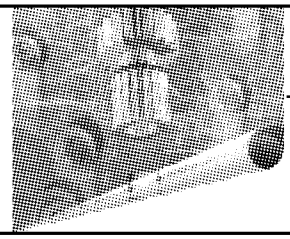

第 3 章

PIC18 微控制器指令集

本章是全书最重要的章节之一，将讲解 PIC18 微控制器的指令集中各指令的操作。尽管大多数编程使用高级编程语言(例如 C 语言)，但是理解每一个指令操作仍很重要，只有这样，才能更好地理解微控制器。本章也将介绍 PIC18 微控制器的一些使用限制。

在本章中，每一指令的讲解都配以短小的应用程序示例，这些应用程序都可以输入 IDE 并执行。因而，凭借 IDE 中的仿真器就可以更好地探究和理解指令的操作。本章的讲解有利于加深读者对第 1 章和第 2 章中提出的思想和概念的印象。深刻理解指令集之后，后面几章将讲解如何编写访问 I/O 设备的应用程序。

本章包含以下内容：

- 如何使用微控制器中可用的寻址模式。
- 微控制器指令集中的各条指令。
- 使用 IDE 开发和仿真小应用程序。
- 如何使用编程结构进行编程。
- 如何使用间接寻址访问数据存储器。
- 如何使用表寻址访问程序存储器。
- 如何在程序中生成和使用宏序列。

3.1 立即数指令

在程序中使用指令之前，必须理解指令的寻址模式。本节讲解 PIC18 系列微控制器的各种寻址模式。

3.1.1 立即数指令详述

立即数寻址是最容易理解的寻址类型，所以我们首先讲解立即数指令。一个立即数就是一个常数，例如数字或 ASCII 符。大多数立即数寻址指令使用工作寄存器，也就是 WREG。表 3-1 列出了 PIC18 系列微控制器指令集中的全部立即数指令。

注意，从表 3-1 可以看出，需要学习的立即数指令并不多。大多数立即数指令的前 3 个字母表示指令所执行的操作。第 1 章中已经讲过，CPU 只能完成一些简单的操作：ADD(加法)、AND(逻辑与)、IOR(或)、MOV(复

制)、MUL(乘法)、SUB(减法)和 XOR(异或)。操作码的后两个字母表示指令本身的信息。例如,字母 L 表示立即数(Literal),字母 W 表示工作寄存器(WREG)。这样,ADDLW 指令的作用就是将立即数与工作寄存器中的数字相加。命名方式不遵循上述规则的立即数指令只有两个,就是 LFSR 和 MOVLB 指令。LFSR 指令将立即数(通常是文件寄存器地址)送入 3 个文件选择寄存器(FSR)之一。文件选择寄存器用于间接寻址数据存储器。MOVLB 指令将立即数(0～15 之间的数字,0 表示存储区 0,15 表示存储区 15)送入存储区选择寄存器(BSR)。

表 3-1　立即数指令

操作码	操作数 1	操作数 2	示　例	说　明
ADDLW	立即数		ADDLW 0x20 ADDLW .100	将 0x20 加到 WREG 将 100 加到 WREG
ANDLW	立即数		ANDLW 0x0F ANDLW .15 ANDLW 0b00001111	0x0F 和 WREG 与 十进制数 15 和 WREG 与 二进制 00001111 和 WREG 与
IORLW	立即数		IORLW 0x80 IORLW 1	0x80 和 WREG 或 1 和 WREG 或
LFSR	FSR 寄存器编号	立即数	LFSR 0,0x123 LFSR 2,0x10	将 0x123 送入 FSR0 将 0x01 送入 FSR2
MOVLB	立即数		MOVLB 2 MOVLB 0	将 2 送入 BSR 将 0 送入 BSR
MOVLW	立即数		MOVLW 3 MOVLW 0x34	将 3 送入 WREG 将 0x34 送入 WREG
MULLW	立即数		MULLW .100 MULLW 2	将 WREG 乘以 100 将 WREG 乘以 2
RETLW	立即数		RETLW 2 RETLW 0x2A	返回 WREG=2 返回 WREG=0x2A
SUBLW	立即数		SUBLW 5 SUBLW .19	从 5 减去 WREG 从 19 减去 WREG
XORLW	立即数		XORLW 4 XORLW 0xF0	4 和 WREG 异或 0xF0 和 WREG 异或

PIC 微控制器的算术和逻辑指令并不完整。PIC 系列的微控制器没有除法指令,但有乘法指令。如果系统需要除法运算,必须为除法开发相应的软件程序。

一般而言,算术和逻辑指令会改变状态寄存器的标志位,但乘法指令例外,乘法指令不影响状态寄存器的任何标志位。(已经讲过,状态寄存器的标志位十分重要,因为微控制器就是使用这些标志位进行决策的。)为了演示状态寄存器标志位的变化,请执行例 3-1 所示的简单程序。虽然这段程序比较简单,但是,由于 ADDLW 指令的作用,状态寄存器发生了变化。示例程序使用 MOVLW 指令将 0x7F 复制到工作寄存器(WREG),然后使用 ADDLW 指令将 1 与 WREG 中的数字相加,在 WREG 中得到和 0x80。注意示例中是如何使用 GOTO Stop 指令结束程序的。因为微控制器唯一能做的事情就是不停地取指令和执行指令,所以时常采用这种技巧(GOTO Stop)来结束程序。在实际的系统软件中,则不需要这样做,因为大多数操作系统使用无限循环的 While 结构连续地重复执行程序。一旦操

作系统停止了，程序也就停止了。

例 3-1：

```
        MOVLW   0x7F        ;load W with 0x7F
        ADDLW   1           ;add 1 to W

Stop:   GOTO    Stop        ;Stop here
```

将这段简单的示例程序放在 IDE 模板文件（参考第 2 章的例 2-6）中的 Main 部分。执行程序，然后从 View 菜单下选择显示特殊功能寄存器中的数据。注意，WREG（位于地址 0xFE8）中的数据是 0x80，状态寄存器中的数据是 0x1A。要查看特殊功能寄存器，先运行程序，然后停止程序，再查看特殊功能寄存器。如果查看屏幕底部，就会发现不但显示了 W 寄存器的内容，而且还显示了其他一些有用信息。如果没有停止运行程序，就不能使用仿真器看到寄存器中数据的正确值。仿真器仅在程序停止运行的时候才会显示更新过的信息。

状态寄存器中的数据 0x1A 的负数标志位是逻辑 1，表示运算结果是一个负数。工作寄存器中的 0x80，也就是二进制的 1000 0000 确实是一个负数。前面已经讲过，状态寄存器最右边的 5 位是标志位（例如负数标志位）。从左至右，从第 4 位开始，这些标志位依次为负数标志位（N）、溢出标志位（OV）、进位标志位（C）、半进位标志位（DC）和零标志位（Z）。状态寄存器中 0x1A 表示，N＝1、OV＝1、C＝0、DC＝1 和 Z＝0。结果是负数，溢出工作寄存器，没有进位，发生了半进位或数位进位，结果不为零。

本例中之所以发生溢出，是因为＋1（0x01）与＋127（0x7F）相加的结果是－128（0x80）。8 位寄存器所能容纳的最大正数是＋127（0x7F），所能容纳的最小负数是－128（0x80）。大于最大正数或小于最小负数，结果就会溢出。注意，仅在使用有符号数时才会发生溢出。如果将本例中使用的数字作为无符号数，那么 0x80 就是无符号数 128，结果是正确的。对于无符号数操作，负数标志位（N）和溢出标志位（OV）没有意义。图 3-1 说明了例 3-1 中的相加过程和两个进位标志位的位置。

进行减法运算时，进位标志位（C）和半进位标志位（DC）实际上保存借位（0＝借位，1＝没有借位）。执行例 3-2 中的程序，工作寄存器中的结果是 0xFF，状态寄存器中的数据是 0x10。SUBLW 指令从立即数中减去 WREG 的值，这看起来与加法正好相反。事实正是如此，因而请注意 PIC18 微控制器中进行减法运算的方式。在本例中，从 5 中减去 6，结果是－1，也就是 0xFF。标识寄存器中进位标志位（C）和半进位标志位（DC）都是 0。这意味着完成这一减法运算，发生了如图 3-2 所示的借位。

例 3-2：

```
        MOVLW   6       ;move 6 into W
        SUBLW   5       ;subtract W (6) from 5

Stop:   GOTO    Stop
```

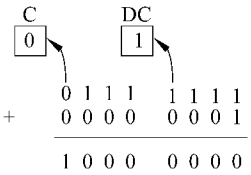

图 3-1　例 3-1 中加法运算的进位标志位(C)和半进位标志位(DC)

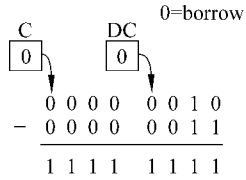

图 3-2　例 3-2 中加法运算的进位标志位(C)和半进位标志位(DC)

立即数逻辑指令。立即数逻辑指令有 3 个：与(AND)、或(IOR)和异或(XOR)。3 种逻辑操作的真值表如图 3-3 所示。在程序中执行时，这些操作都是按位进行的。与(AND)操作常用来有选择地将某一位或某几位置 0，因为任何逻辑值和逻辑 0 进行与操作之后，结果都是逻辑 0。或(IOR)操作常用来有选择地将某一位或某几位置 1，因为任何逻辑值和逻辑 1 进行或操作之后，结果都是逻辑 1。异或(XOR)操作常用来有选择地将某一位或某几位取反(从 0 变为 1，或者从 1 变为 0)，因为任何逻辑值和逻辑 0 进行异或操作之后，结果都会取反。这些操作在图 3-3 中也有说明，图 3-3 使用 X 代表任意逻辑值，演示了这些逻辑操作是如何改变 X 的逻辑值的。

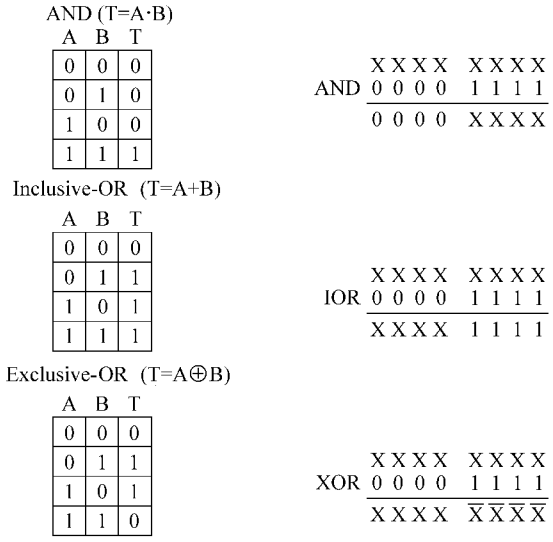

图 3-3　与(AND)、或(IOR)和异或(XOR)操作

假设需要将工作寄存器最右边的两位(第 0 位和第 1 位)清零，而将第 6 位和第 7 位置 1。则可以使用 AND 指令将第 0 位和第 1 位清零，使用 IOR 指令将第 6 位和第 7 位置 1。例 3-3 中的示例程序演示了这一过程。该例中，首先将 0x1F 送入工作寄存器，作为测试值。在 IDE 中执行该程序，WREG 寄存器中的数据变为 0xDC。

例 3-3：

```
        MOVLW   0x1F
        ANDLW   0xFC        ;clear bits 0 and 1
```

```
        IORLW   0xC0        ;set bits 6 and 7

Stop:   GOTO    Stop
```

另一个例子要求工作寄存器(WREG)最左边的三位取反。使用 XOR 和立即数 0xE0(1110 0000),就可以实现对工作寄存器最左边的三位取反。例 3-4 演示了将 0x90(1001 0000)最左边的三位取反的实验程序,程序运行的结果是 0x70(0111 0000)。

例 3-4:

```
        MOVLW   0x90
        XORLW   0xE0        ;invert left 3 bits

Stop:   GOTO    Stop
```

比较各种逻辑指令的常用程度,与(AND)操作使用得最为频繁,多用来屏蔽寄存器的位。例如,假设工作寄存器中含有数字 0x4A,但只需要使用该数字最右边的 4 位(A)。那么,如何屏蔽最左边的 4 位呢? 可以使用指令 ANDLW 0x0F 擦除 0x4A 中的 4,而结果就是 0x0A。同样,如果只需要使用该数字最左边的 4 位,可以使用指令 ANDLW 0xF0 擦除 0x4A 中的 A,工作寄存器中的结果就是 0x40。

3.2　位操作指令

本节将讲解逻辑操作中的位操作指令。位操作指令用于设置、清除和测试某个位,而刚刚讲解过的 AND、IOR 和 XOR 等指令是字节操作指令,可以改变一个或多个位。表 3-2 给出了 PIC 微控制器的位操作指令。许多微控制器应用程序需要位操作指令,以便对程序中的某个位单独进行控制或测试。在对 I/O 设备进行接口和控制时,情况尤为如此。

表 3-2　位操作指令

指　令	操作数 1	操作数 2	操作数 3	示　　例	说　　明
BCF	寄存器	位#	a 位	BCF WREG,7 BCF 0x10,1,0 BCF 0x10,1,ACCESS	清除 WREG 第 7 位 清除访问区域寄存器 0x10 第 1 位 清除访问区域第 1 位
BSF	寄存器	位#	a 位	BSF WREG,1	设置 WREG 第 1 位
BTFSC	寄存器	位#	a 位	BTFSC WREG,2	如果 WREG 第 2 位为 0,跳过下一条指令
BTFSS	寄存器	位#	a 位	BTFSS WREG,7	如果 WREG 第 7 位为 1,跳过下一条指令
BTG	寄存器	位#	a 位	BTG WREG,4	将 WREG 第 4 位取反

在汇编器中,当在指令中使用数字形式的文件寄存器地址来指定访问区或存储区选择寄存器时才会用到表 3-2 中的 a 位。如果指令通过寄存器的名称或存储器位置的名称

引用寄存器，而不是采用数字形式的地址引用寄存器，那么，在程序中就不必使用 a 位。注意，表 3-2 中将 WREG 作为操作数 1 使用的时候，也不必指定 a 位。汇编器很“聪明”，它知道访问区域就在 WREG 的位置，因而在指令中使用特殊寄存器的名称时，就不需要使用 a 位。

为了解释 BCF（位清除）和 BSF（位设置）指令。在例 3-5 中，使用位操作指令替代 ANDLW 和 IORLW 指令重写例 3-3。比较这两个例子就会发现，尽管两个例子的结果是一样的，例 3-5 更容易理解一些。这两个例子确实有些不同：其中一个不同就是位操作指令不会修改状态寄存器，而 ANDLW 和 IORLW 指令则会修改状态寄存器。另外，由于要改变多个位，每设置或清除一个位就需要一条位操作指令。由此可见，要改变多个位时，使用 ANDLW 和 IORLW 指令更加有效。

例 3-5：

```
        MOVLW   0x1F
        BCF     WREG, 0         ;clear bit 0
        BCF     WREG, 1         ;clear bit 1
        BCF     WREG, 6         ;set bit 6
        BCF     WREG, 7         ;set bit 7

Stop:   GOTO    Stop
```

例 3-6 使用条件位操作指令设置 WREG 的位，仅在第 7 位为 0 的条件下才将该位设置为 0。BTFSS 指令测试 WREG 的第 7 位，如果该位是逻辑 1，就跳过下一条指令（BCF）。如果 WREG 的第 7 位是逻辑 0，就执行下面的 BCF 指令，将该位清零。要测试这一过程，使用 IDE 和仿真器，并在 MOVLW 指令中使用不同的值，例如 0x7F 和 0xFF。

例 3-6：

```
        MOVLW   0x7F            ;load test data
        BTFSS   WREG, 7
        BCF     WREG, 0         ;clear bit 0

Stop:   GOTO    Stop
```

假设程序需要在 PICDEM2 PLUS 演示板的 LED 上显示数字。LED 连接到 Port B 的第 0 位～第 3 位。演示板上有一个按钮，连接到 Port A 的第 4 位。没有按下按钮时，在 LED 上显示数字 0x05，按下按钮时，在 LED 上显示数字 0x03。按钮被按下时产生逻辑 0，没有被按下时产生逻辑 1。例 3-7 给出了完成这一任务的程序。如果使用的是其他演示板，则可能需要修改 I/O 端口和位的位置。所寻址的 I/O 端口应当是 Port A 或 Port B，用来传送数据。在相应的 TRISA 或 TRISB 寄存器中放置 0，将端口针脚的方向设置为输出；在相应的 TRISA 或 TRISB 寄存器中放置 1，将端口针脚的方向设置为输入。ADCON1 寄存器用于将针脚编程为模拟针脚或数字针脚。在 ADCON1 寄存器中放置 0x7F，将针脚设置为

数字针脚；在 ADCON1 寄存器中放置 0x00，将针脚设置为模拟针脚。注意，在各种的 PIC18 系列微控制器中，ADCON1 寄存器的功能和使用的值都有所不同。本示例程序是针对 PIC18F1320 编写的。如果使用其他微控制器，所设置的值可能会有所不同，甚至根本就不需要使用 ADCON1 寄存器。

例 3-7：

```
Main:
        MOVLW   0x7F            ;program all ports ass digital
        MOVWF   ADCON1

        MOVLW   0x00
        MOVWF   TRISB           ;Port B is output

        MOVLW   0xFF
        MOVWF   TRISA           ;Port A is input

Main1:                          ;main program loop
        MOVLW   0x05

        BTFSS   PORTA, 0        ;test Pushbutton
        MOVLW   0x03            ;and skip this if Pushbutton is up

        MOVWF   PORTB           ;change Port B and the LEDS

        GOTO    Main1           ;repeat
```

如果在指令中使用状态寄存器，常用标志位的名称来寻址某个位。例如，要清除进位标志位，可以使用指令 BCF STATUS,C，而无需使用指令 BCF STATUS,2。但是，这两条指令都可以使用。

3.3 字节指令

面向字节的指令（Byte-oriented Instruction）是 PIC18 微控制器指令集中数量最多的指令，常常也是程序中使用得最多的指令。在程序中，面向字节的指令可以使用变量（Variable）数据，而立即数指令使用常量（Constant）数据。面向字节的指令通常使用工作寄存器和寄存器文件中的某个位置来完成某项操作。表 3-3 给出了所有的面向字节的指令，每条指令给出了两个示例。大多数面向字节的指令有 3 个操作数：第 1 个操作数是寄存器文件位置，第 2 个操作数决定程序的目标位置，第 3 个操作数选择访问区或寄存器文件存储区，访问区或寄存器文件存储区的位置由存储区选择寄存器决定。如果第 2 个操作数（d 位）是 0，则目标位置是 WREG；如果第 2 个操作数是 1，则目标位置是寄存器堆位置。如果第 3 个操作数（a 位）是 0，则使用访问区作为寄存器文件位置；如果第 3 个操作数是 1，则使

用 BSF 决定的存储区作为寄存器文件位置。只有在使用数字引用文件寄存器时，才需要使用 a 位。如果使用标记引用文件寄存器，就不需要在指令中使用 a 位。默认情况下，a＝1。只有要将操作结果送入工作寄存器时，才需要使用 d 位。默认情况下，d＝1。在汇编语言中，如果需要使用 d 位，使用字母 W 或 F 表示 d 位。表 3-3 给出了一些相关示例。

表 3-3　面向字节的指令

指　令	操作数 1	操作数 2	操作数 3	示　例	说　明
ADDWF	寄存器	d 位	a 位	ADDWF 0x10,W,0 ADDWF 0x10,F,0	将 W 寄存器和访问区位置 0x10 相加，结果保存在 W 寄存器中 将 W 寄存器和访问区位置 0x10 相加，结果保存访问区位置 0x10
ADDWFC	寄存器	d 位	a 位	ADDWFC 0x10,0,0 ADDWFC 0x10,1,1	将带进位的 W 寄存器和访问区位置 0x10 相加，结果保存在 W 寄存器中 将带进位的 W 寄存器和 BSR 存储区位置 0x10 相加，结果保存在 BSR 存储区位置 0x10
ANDWF	寄存器	d 位	a 位	ANDWF 0x10,0,1	将 W 寄存器和 BSR 存储区位置 0x10 相与，结果保存在 W 寄存器中
CLRF	寄存器	a 位		CLRF WREG CLRF 0x10,ACCESS CLRF BOB	将 W 寄存器清零为 0x00 将访问区位置 0x10 清零为 0x00 清除位置 BOB
COMF	寄存器	d 位	a 位	COMF WREG COMF 0x10,0,0 COMF 0x10,1,1	求 W 寄存器的反码 求访问区位置 0x10 的反码 求 BSR 存储区位置 0x10 的反码
CPFSEQ	寄存器	a 位		CPFSEQ 0x10,0 CPFSEQ 0x11,1	比较 W 寄存器和访问区位置 0x10，如果相等就跳过下一条指令 比较 W 寄存器和 BSR 存储区位置 0x11，如果相等就跳过下一条指令
CPFSGT	寄存器	a 位		CPFSGT 0x12,0	比较 W 寄存器和访问区位置 0x12，如果访问区位置 0x12 的内容比 W 寄存器大，就跳过下一条指令
CPFSLT	寄存器	a 位		CPFSLT 0x13,1	比较 W 寄存器和 BSR 存储区位置 0x13，如果 BSR 存储区位置 0x13 的内容比 W 寄存器小，就跳过下一条指令
DECF	寄存器	d 位	a 位	DECF WREG DECF 0x10,0,0 DECF 0x10,1,0	从 W 寄存器减 1 从访问区位置 0x10 减 1，并将结果保存在访问区位置 0x10 从访问区位置 0x10 减 1，并将结果保存在访问区位置 0x10
DECFSZ	寄存器	d 位	a 位	DECFSZ WREG	将 W 寄存器减 1，如果结果是 0，则跳过下一条指令
DCFSNZ	寄存器	d 位	a 位	DFNSZ 0x10,1,0	将访问区位置 0x10 减 1，如果结果不是 0，则跳过下一条指令

续表

指　令	操作数1	操作数2	操作数3	示　例	说　明
INCF	寄存器	d位	a位	INCF WREG INC 0x10,0,0	将W寄存器增1 将访问区位置0x10增1
INCFSZ	寄存器	d位	a位	INCFSZ WREG	将W寄存器增1，如果结果是0，则跳过下一条指令
INFSNZ	寄存器	d位	a位	INFSNZ WREG	将W寄存器增1，如果结果不是0，则跳过下一条指令
IORWF	寄存器	d位	a位	IORWF 0x10,0,0	将W寄存器和访问区位置0x10进行或，并将结果保存在访问区位置0x10
MOVF	寄存器	d位	a位	MOVF 0x10,0,0	将访问区位置0x10的内容复制到W寄存器中
MOVFF	寄存器源位置	寄存器目标位置		MOVFF WREG,0x130	将W寄存器的内容复制到位置0x130
MOVWF	寄存器	a位		MOVWF 0x10,0	将W寄存器的内容复制到访问区位置0x10
MULWF	寄存器	a位		MULWF 0x10,0	将W寄存器和访问区位置0x10相乘，将乘积保存在PRODL和PRODH中
NEGF	寄存器	a位		NEGF WREG	求W寄存器的补码（相反数）
RLCF	寄存器	d位	a位	RLCF WREG	带进位对W寄存器进行左移位
RLNCF	寄存器	d位	a位	RLNCF WREG	不带进位对W寄存器进行左移位
RRCF	寄存器	d位	a位	RRCF WREG	带进位对W寄存器进行右移位
RRNCF	寄存器	d位	a位	RRNCF WREG	不带进位对W寄存器进行右移位
SETF	寄存器	a位		SETF WREG SETF 0x11,ACCESS	将0xFF保存在W寄存器中 将0xFF保存在访问区位置0x11
SUBFWB	寄存器	d位	a位	SUBFWB 0x10,0,0	从W寄存器带借位减去访问区位置0x10，将结果保存在访问区位置0x10
SUBWF	寄存器	d位	a位	SUBWF 0x10,0,0	从访问区位置0x10减去W寄存器，将结果保存在访问区位置0x10
SUBWFB	寄存器	d位	a位	SUBWFB 0x10,0,0	从访问区位置0x10带借位减去W寄存器，将结果保存在访问区位置0x10
SWAPF	寄存器	d位	a位	SWAPF WREG	交换W寄存器中的半字节
TSTFSZ	寄存器	d位		TSTFSZ WREG	如果W寄存器的值是0，则跳过下一条指令
XORWF	寄存器	d位	a位	XORWF 0x10,0,0	将W寄存器和访问区位置0x10进行异或，并将结果保存在访问区位置0x10

假如要将两个 16 位数字相加，得到一个 16 位的和。在本节之前，完成这一操作要更加困难一些，但是，因为面向字节的指令中有一个带进位的加法(Addition With Carry)指令，所以就可以更加简便地将 8 位以上宽度的数字相加。例 3-8 用一段简单的程序演示了将访问区位置 0x10 和 0x11 处的 16 位数与访问区位置 0x12 和 0x13 处的 16 位数相加的过程。这两个 16 位数字都是以小结尾格式存储的，最低有效位保存在低位位置。相加后的 16 位结果保存在访问区位置 0x14 和 0x15 处。程序看起来有些长，但是完成加法的程序部分(最后 6 条指令)却很短。注意最低有效位是如何相加的，以及最高有效位是如何进位相加的。使用同样的方法，任何宽度的数字都可以相加。图 3-4 说明了程序的操作，以及如何使用进位形成 16 位的和。

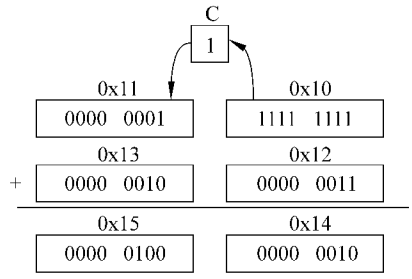

图 3-4　16 位数相加示例

例 3-8：

```
        MOVLW   0xFF                    ;store 0x01FF
        MOVWF   0x10                    ;into 0x10 and 0x11
        MOVLW   0x01
        MOVWF   0x11

        MOVLW   0x03                    ;store 0x0203
        MOVWF   0x12                    ;into 0x12 and 0x13
        MOVLW   0x02
        MOVWF   0x13

        MOVF    0x10, W, ACCESS         ;add 0x10 and 0x12
        ADDWF   0x12, W, ACCESS
        MOVWF   0x14                    ;save result at 0x14

        MOVF    0x11, W, ACCESS         ;add 0x11 and 0x13 with carry
        ADDWF   0x13, W, ACCESS
        MOVWF   0x15                    ;save result at 0x15

Stop:   GOTO    Stop
```

该程序更简洁、更易编写的版本如例 3-9 所示。例 3-8 和例 3-9 的不同之处在于，例 3-9 使用汇编器定义存储器位置，避免了使用例 3-8 所示的操作数。在程序模板文件中已经定义了 UDATA 区域，因而无需再使用 UDATA(用户数据)伪指令就可以将例 3-9 中的代码段添加到模板文件中。如果打算将数据放在访问区，那么在 RES(保留存储器)指令之前应当使用 UDATA_ACS 伪指令。这样就可以选择在哪里保存数据。保存在访问区的数据通常执行得更快，因为不需要使用存储区地址初始化存储区选择寄存器。