第1章 CPU架构

1.1 概 述

dsPIC30F系列的CPU采用了改良型哈佛架构，其数据总线和程序总线是独立的，这样有效地消除了数据传输的瓶颈。说它是改良型哈佛结构，主要在于：数据总线宽度为16位，程序总线宽度为24位；程序区和数据区也可以交换数据(PSV、表读/表写)等。同时该芯片包含了强大的DSP引擎支持，拥有一个增强功能的指令集，为数字信号处理提供了硬件支持。由于CPU拥有24位宽度的程序指令字，指令字带有长度可变的操作码字段。程序计数器(PC)的长度为23位，其最低位强制为0，因此可以寻址高达4 M×24位的用户程序存储器空间。单周期指令和预取机制可以提供最大的吞吐量。除了改变程序流的指令(比如GOTO、CALL)、双字移动指令(MOV. D)和表操作指令以外，所有指令都在单个周期内执行。硬件支持DO和REPEAT指令，意味着可以免除大多数影响程序执行的“家务活”，比如维护循环次数(加或减)、判断循环是否到达预定值、跳转等工作。更为重要的是，DO和REPEAT在执行的时候可以被中断。

dsPIC30F系列DSC拥有16个16位长度的工作寄存器堆。每个工作寄存器都可以充当数据、地址或地址偏移寄存器(指针)。其中W15被分配作为软件堆栈的指针。

dsPIC30F指令集有两类指令：MCU类指令和DSP类指令。这两类指令无缝地集成到CPU架构中并从同一个执行单元开始执行。指令集包括很多寻址模式，并专门针对C编译器进行了优化，这样一来用户使用C语言编程的时候就可以达到最佳的代码效率。

数据存储器的寻址范围为64 KB(32 KW)，并可以被分成两个数据区，即X数据区和Y数据区。每个存储器区有各自独立的地址发生单元AGU(Address Generation Unit)。MCU类指令通过X数据AGU对数据进行操作，这时整个数据存储器映射空间被作为一个线性数据空间访问。当执行某些DSP类指令(比如乘加运算：MAC)时，将同时使用X数据AGU、Y数据AGU进行操作，意味着可以同时操作两个数据，这样会将数据空间分成X和Y两个部

分。不同型号芯片的数据空间大小可能不同。

芯片支持程序空间可视化操作(Program Space Visibility,PSV),也就是说可以使用任何访问RAM的方式访问 Flash 里的内容。使用一个专门的 8 位程序空间可视化页寄存器(Program Space Visibility Page,PSVPAG)可以定义任何以 16 KB 程序字为单位的空间,这些空间里的数值(当然是 24 位程序字的低 16 位)将被映射到数据存储器空间的高 32 KB 地址范围内。另外,在带有外部总线的芯片上,RAM 可被连接到程序存储器总线并用来扩展数据 RAM。

利用 X 数据 AGU 和 Y 数据 AGU,可以实现"零家务活"循环缓冲器(模寻址)。模寻址完全避免了做 DSP 算法的时候进行边界检查。此外,X 数据 AGU 的循环寻址可以与任何 MCU 类指令一起使用。X 数据 AGU 还支持位反转寻址,避免了 FFT 蝶形运算算法对输入、输出数据的重新排序。

寻址方式多样,包括固有(无操作数)寻址、相对寻址、立即寻址、存储器直接寻址、寄存器直接寻址和寄存器间接寻址。每条指令最多支持 6 种不同的寻址模式。

对于大多数指令,在每个指令周期 dsPIC30F 能执行一次数据(或程序数据)存储器读操作、一次工作寄存器(数据)读操作、一次数据存储器写操作和一次程序(指令)存储器读操作。所以,可以支持 3 个操作数的指令,使 A+B=C 操作能在单个周期内执行。

DSP 引擎拥有一个高速 17 位×17 位乘法器、一个 40 位 ALU、两个 40 位饱和累加器和一个 40 位双向桶形移位器。该桶形移位器在单个周期内至多可将一个 40 位的值右移 15 位或左移 16 位。DSP 指令可以无缝地与所有其他指令一起操作,具有最佳的实时性能,方便、直接、快速。当两个 W 寄存器相乘时,MAC 指令和其他相关的指令可以同时从存储器(X 区域和 Y 区域)取出两个数据操作数。这要求数据存储器在遇到 DSP 指令时拆分为 X、Y 两块区域,而对所有其他指令保持线性(X 区域)。这得益于在硬件上为每个数据空间设置可专用工作寄存器,使得拆分动作透明而灵活。

dsPIC30F 具有完善的中断系统。支持多达 8 个非屏蔽中断入口和 54 个常规中断入口。非屏蔽中断的引入可以增强系统可靠性,加快 MCU 处理极端情况的响应时间。对于常规中断,MCU 可以为每个中断源分配 7 个优先级之一。

图 1.1 所示为 dsPIC30F 系列单片机的原理框图。从图 1.1 中看到 dsPIC30F 系列单片机的结构和 PIC 系列单片机有非常类似之处,但是加入了更多外设,比如电机控制 PWM、QEI 接口、DCI 接口等。

dsPIC30F 系列单片机具有更完善的指令系统、软件堆栈管理系统、中断系统,具有更多的 RAM 和 Flash 存储器,适合于更高端的应用。

它延续了 PIC 系列 2.5～5.5 V 宽电压供电、低功耗的传统,在外设和端口的配置上也非常类似,甚至连端口的定义都完全一样。与 PIC 系列一样,很多模块比如上电/掉电时序控制、看门狗、振荡器控制等和系统可靠性息息相关的模块成为标准配置。这让大多数 PIC 爱好者觉得非常亲切,而且感觉上手很快。

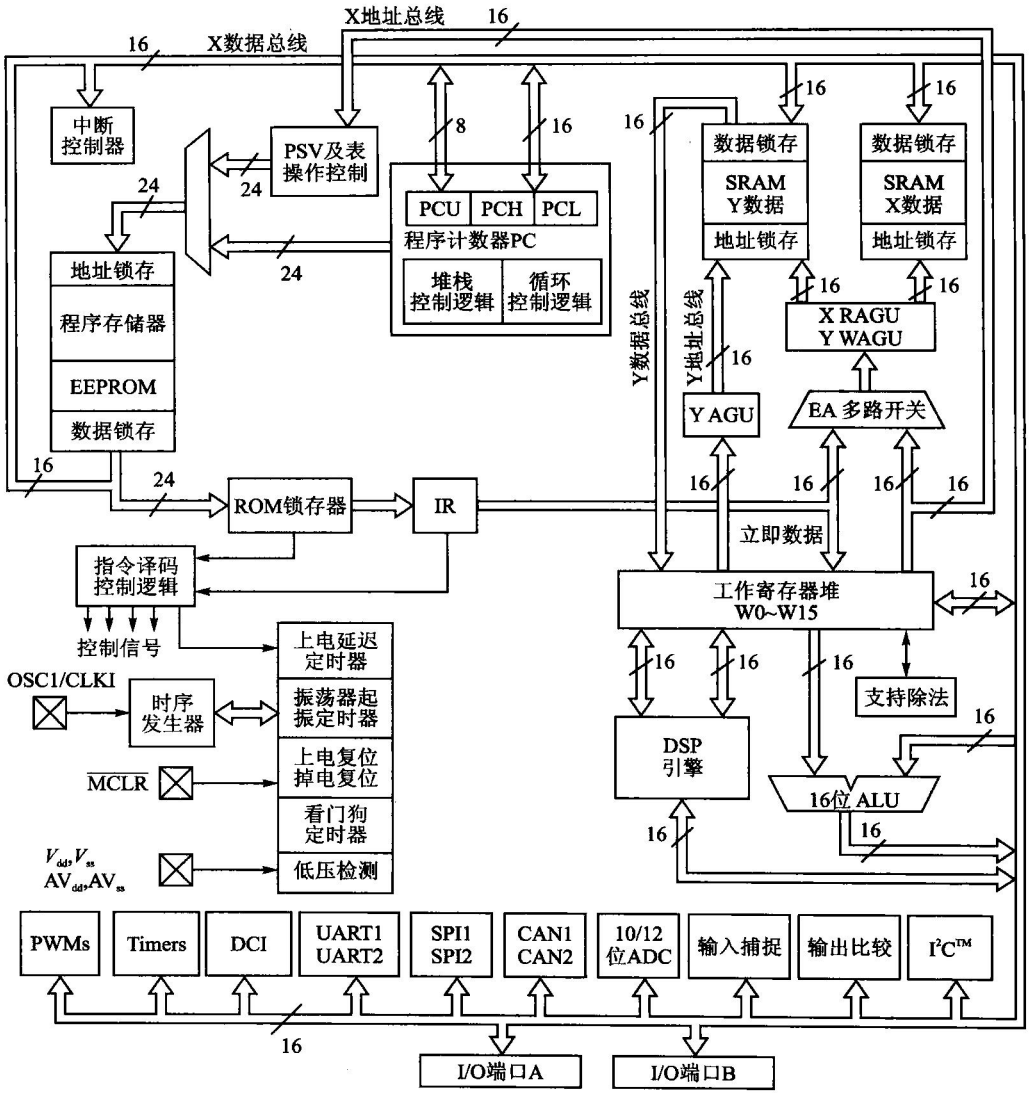

图 1.1　dsPIC30F 系列单片机框图

与 PIC 系列 8 位单片机不同的是，不再有程序空间分页和数据空间分块的概念，操作完全线性，大大提高了操作效率和速度。这一提高让用户感觉非常好，避免了因为忘记切换页而错误寻址。

当然，dsPIC 系列单片机完全是一个 16 位架构，而且加入了 DSP 运算核，使得在做数字信号处

理的时候更快速。同时运行速度也大幅度提高，可达到 30 MIPS。对于 dsPIC33F 系列的数字信号处理器处理速度可达 40 MIPS。很多用 8 位 MCU 无法完成的任务，dsPIC 可以轻松胜任。

当然，dsPIC 的结构要比 PIC 系列 8 位 MCU 的结构要复杂很多，在以后的章节里将分别详细介绍。

1.2 编程者模型(Programmer's Model)

图 1.2 所示为 dsPIC30F 的编程者模型。所谓编程者模型就是在编写程序时编程者最关心的一些特殊功能寄存器，比如：PC 指针、累加器、堆栈深度限制寄存器、工作寄存器、状态寄存器和核心控制寄存器等。这些特殊功能寄存器决定了整个芯片的功能、结构和效率。编程者对它们的了解是至关重要的。不管是使用汇编语言还是使用 C 语言编写程序，所有 dsPIC 编程者都必须了解编程者模型。

编程者模型中的所有特殊寄存器都是存储器映射的，并且可以由指令直接访问。用户可以在链接器脚本文件(.gld 扩展名)里找到每个特殊功能寄存器的绝对地址。这些特殊功能寄存器同样是在文件寄存器范畴，全部位于 2 KB 地址空间范围内。

表 1.1 中提供了编程者模型里各个相关寄存器的名字和相应的描述。

表 1.1 与编程者模式相关的主要寄存器

寄存器	功能描述
W0,W1,...,W15	工作寄存器堆（DSP 和 MCU 共用）
ACCA,ACCB	DSP 专用累加器（40 位长度）
PC	程序指针（23 位长度）
SR	状态寄存器（DSP 和 ALU 操作标志位）
SPLIM	堆栈深度限制寄存器
TBLPAG	表操作页指针
PSVPAG	程序可视化页寄存器
RCOUNT	REPEAT 指令循环次数寄存器
DCOUNT	DO 指令次数寄存器
DOSTART	DO 指令起始地址寄存器
DOEND	DO 指令结束地址寄存器
CORCON	核心控制寄存器(包含 DSP 和 DO 相关设置)

除了编程者模型中包含的寄存器之外，dsPIC30F 还包含模数寻址、位反转寻址和中断控制寄存器。这些寄存器用于数字滤波器(FIR、IIR)和快速傅里叶变换(FFT)等场合，为复杂

的数字信号处理提供硬件支持。这些寄存器将在本书随后的章节中介绍。

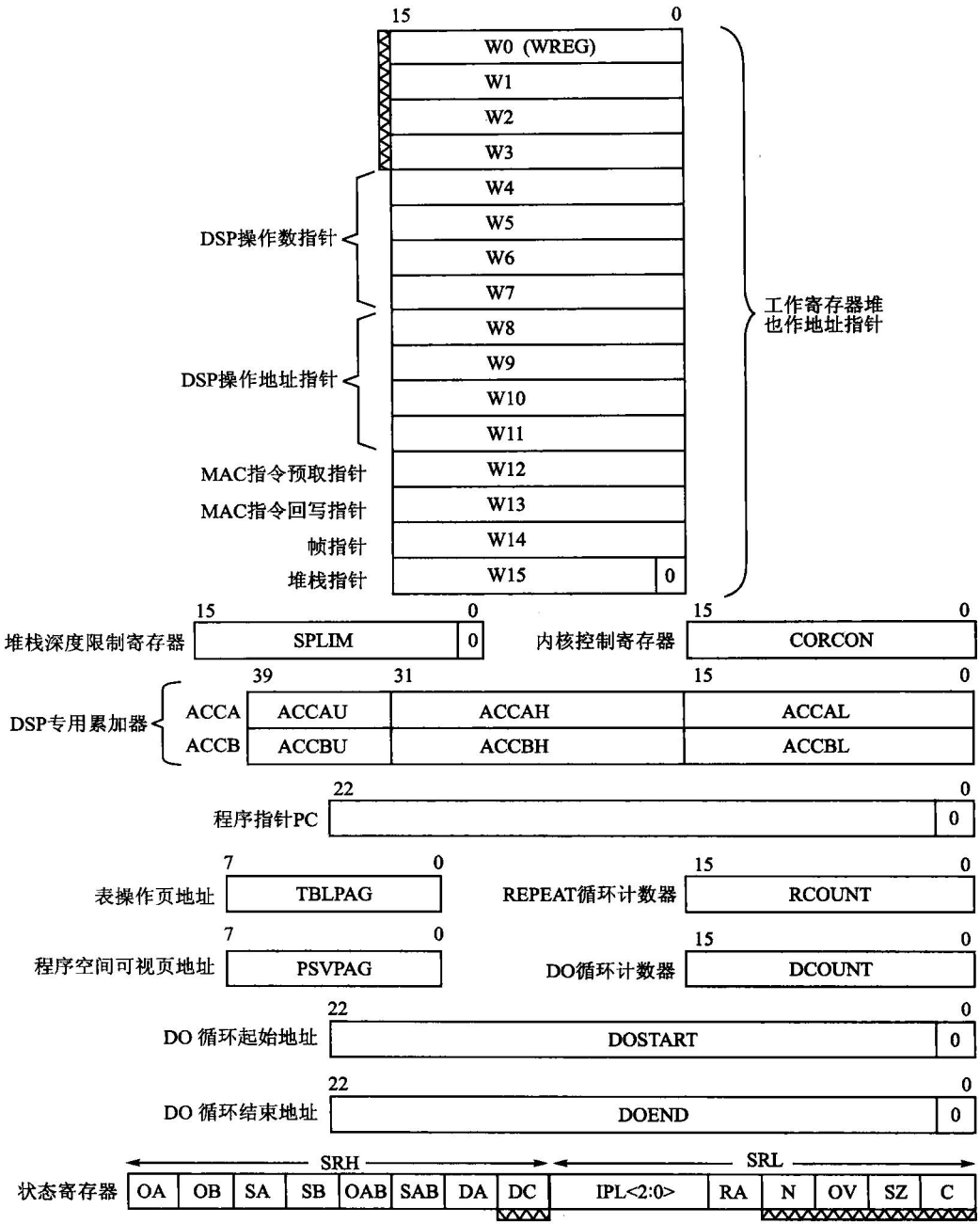

图 1.2　编程者模型图示

从图 1.2 编程者模型中可以看到有些寄存器或某些位画有阴影，这些寄存器（W0、W1、W2、W3）和 SR 寄存器中的一些位（DC、N、OV、SZ、C）可以使用 PUSH.S 指令保存到一级深度的影子寄存器里，也可以用 POP.S 指令从影子寄存器里恢复这些寄存器的值。

从图 1.2 中还看到有 3 个特殊功能寄存器，即 DCOUNT、DOSTART、DOEND，它们用于 DO 循环的设置。这 3 个寄存器都有一级深度的影子寄存器，当 DO 循环嵌套的时候可以用来保存这些寄存器的值，以便进入下一级的 DO 嵌套。

1.2.1 工作寄存器堆

工作寄存器堆共有 16 个工作寄存器（W0～W15）可以作为数据、地址或地址偏移寄存器。寻址模式决定了 W 寄存器的不同功能。

dsPIC30F 指令集可被分成两种指令类型：寄存器指令和文件寄存器指令。寄存器指令可以把每个 W 寄存器用作数据值或地址偏移值。例如：

```
MOV     W0,W1              ; 将 W0 里的数值送入 W1 寄存器
MOV     W0,[W1]            ; 将 W0 里的数值送入 W1 指向的地址里
ADD     W0,[W4],W5         ; 将 W0 里的数值和 W4 指向地址里的数值相加，结果存入 W5
```

1. W0 和文件寄存器指令

W0 是一个特殊的工作寄存器，因为它是可在文件寄存器指令中使用的唯一工作寄存器。文件寄存器指令对包含在指令操作码和 W0 中指定的存储器地址进行操作。在文件寄存器指令中，W1～W15 不可被指定为目标寄存器。

众所周知，现有的 PIC 单片机只有一个 W 寄存器，因此 dsPIC 的文件寄存器指令可以提供向后兼容性。在 dsPIC 的汇编器语法中使用标号“WREG”来表示文件寄存器指令中的 W0。例如：

```
MOV     WREG,0x0100        ; 将 W0 里的数值送入 0x0100 地址单元里
ADD     0x0100,WREG        ; 将 W0 里的数值和 0x0100 里的数值相加，结果存入 W0
```

2. W 寄存器存储器映射

由于 W 寄存器是存储器映射的，每个 W 寄存器都有自己的地址。因此寄存器寻址指令中可以直接使用该寄存器的地址进行寻址。但是这种方法并不方便记忆。例如：

```
MOV     0x0004, W10        ;其中 0x0004 是 W2 存储器中的地址，等同于 MOV W2, W10
```

在一条指令中 W 寄存器甚至可以同时被当作寻址的地址指针和目的地址使用。例如：

```
MOV     W1, [W2 + +]       ;[W2] 指针对文件寄存器 W2 进行寻址
```

假如 W1＝0x1234，W2＝0x0004。这里 W2 被用作间接寻址的地址指针，它指向文件寄存器中的单元 0x0004。而刚好 W2 寄存器的地址也是 0x0004。显然实际使用的时候用户不

大可能故意使用这种情况，但它确实可以运行，写操作总是能实现。因此最后 W2＝0x1234。

3. W 寄存器和字节型指令

对于那些把 W 寄存器作为目标寄存器的字节型指令，其操作结果只影响目标寄存器的低位字节。因为 W 寄存器是存储器映射的，其高位字节和低位字节都有相应的地址。所以用户可以使用字节型指令对数据存储器空间里存储单元的低位和高位字节进行操作。

这种寻址方式非常方便以前用户已有的 8 位运算程序，其程序迁移非常容易。同时，在某些场合字节型运算也可以帮助节省存储空间。

1.2.2 影子寄存器(Shadow Register)

图 1.2 中可以看到阴影部分的寄存器或相应的位可以被保存到影子寄存器里。编程模型中的许多寄存器都有相关的影子寄存器。影子寄存器在文件寄存器里没有相应的映射地址，因此不能被用户直接访问。目前共有两种类型的影子寄存器：一类是可以被 PUSH. S 和 POP. S 指令使用的影子寄存器，另一类是可以被 DO 指令使用的影子寄存器。

1. 可以用 PUSH. S 和 POP. S 指令操作的影子寄存器

在执行函数调用和中断服务子程序(ISR)过程中，使用 PUSH. S 和 POP. S 指令可以快速地进行现场保存或恢复。使用 PUSH. S 指令可以将工作寄存器 W0、W1、W2、W3 以及 SR 寄存器里 N、OV、Z、C、DC 位的值传输到它们相应的影子寄存器里保存起来。使用 POP. S 指令可以将影子寄存器里保存的信息恢复到相应的寄存器单元或位里面。至于是如何恢复的完全由硬件操作。

下面是使用 PUSH. S 和 POP. S 指令的范例：

```
MyFunction:
PUSH.S                      ; 保存 W 寄存器，MCU 状态寄存器
MOV     #0x03,W0            ; 给 W0 寄存器赋予立即数
ADD     RAM100              ; 将 W0 里的数据和 RAM100 里的数据相加
BTSC    SR,#Z               ; 结果为 0?
BSET    Flags,#IsZero       ; 是，设置标志位
POP.S                       ; 恢复 W 寄存器，MCU 状态寄存器
RETURN
```

注意：只要使用了 PUSH. S 指令一定会改写先前保存在影子寄存器中的内容。而影子寄存器深度只有一级，所以如果多个软件任务都可能用到影子寄存器的时候必须十分小心。

用户必须确保任何使用影子寄存器的任务不会被同样使用该影子寄存器且具有更高优先级的任务中断。如果允许较高优先级的任务中断较低优先级的任务，则影子寄存器在较低优先级任务中保存的内容将被较高优先级任务改写。假如允许任务嵌套，则必须对低优先级的现场参数手动保护。这有点类似于 PIC18 系列单片机里的影子寄存器在中断嵌套时遇到的

问题。

2. DO 循环专用影子寄存器

当执行 DO 指令时，DOSTART(DO 起始地址)、DOEND(DO 结束地址)、DCOUNT(DO 计数器)这 3 个寄存器的内容将自动保存在影子寄存器中。DO 影子寄存器的深度为一级，允许两个 DO 循环自动嵌套。具体应用将在后面详细介绍。

1.2.3 未初始化的 W 寄存器的复位

所有的 W 寄存器(除了 W15 之外)在发生任何类型的复位时将被清零，同时假如这些寄存器没有被使用指令对其进行写操作之前，则认为它们未经初始化。假如用户试图把未初始化的寄存器用作地址指针时将会导致芯片复位。

请小心：用户必须执行字写操作(Word Write)来初始化 W 寄存器。字节写(Byte Write)操作不会影响 CPU 的初始化检测逻辑。

1.3 软件堆栈(Software Stack)

和 PIC16 或 PIC18 等 8 位单片机的硬件堆栈不同，dsPIC30F 系列 DSC 采用软件堆栈。所有 16 位 PIC 和 dsPIC 的堆栈都是建立在 RAM 里面，因此堆栈的深度完全决定于某颗芯片片上 RAM 的多少。硬件堆栈数量较少(8 级或 31 级)、简单可靠、操作快速、不占用系统 RAM，但是存在堆栈级数少、不能被用户直接操作、不支持递归运算、不能用来保存用户变量等缺点。而基于 RAM 的软件堆栈深度可调、可保存用户变量、操作灵活、支持递归等复杂算法。软件堆栈的缺点是占用系统 RAM、芯片成本增加。

W15 被用作堆栈指针，因此用户一般不要使用 W15 作别的用途。为了避免错误的堆栈访问，W15 的最低位被硬件强制设置为“0”。该指针在中断处理、子程序调用与返回等情况下将被自动修改。与操作所有其他 W 寄存器的方式一样，W15 也可以使用任何指令对其进行操作。这样可以简化对堆栈指针的读、写和控制操作。例如用户可以建立堆栈帧(Stack Frame)。

当芯片发生复位时(任何类型的复位)W15 都被初始化为指向 0x0800，也就是片内 RAM 的起始地址(0～7FF 为 SFR 的范围)。这样可以确保芯片一复位即可获得有效的堆栈指针，指向有效的 RAM 地址。这样的设计非常有利于处理一些极端情况，比如单片机复位后，在软件还没有来得及初始化 SP 之前就发生了非屏蔽陷阱的时候，堆栈可以用来保存断点数据。在初始化期间，用户可以根据需要将 SP 重新指向 RAM 空间内的任何地址单元。

图 1.3 所示为 dsPIC 等 16 位单片机的堆栈结构。可以看到这种堆栈采用“向上生长型”，也就是说每压栈一次，SP 指针会指向高一级地址，从低地址到高地址填充软件堆栈。堆栈指针 SP 总是指向下一个可用的堆栈空间。堆栈出栈(POP)时，堆栈指针先减；堆栈进栈

(PUSH)时,堆栈指针后加。

图1.3所示的堆栈操作是执行CALL指令期间的PC进栈操作。PC压栈时,PC<15:0>这16位数据首先被自动压入第一个可用的堆栈字里,紧接着被压入第二个可用堆栈单元的16位数据中低7位来自PC<22:16>,高9位全部填"0"。至此现场信息也即23位PC指针被保存起来了。

发生中断事件时的压栈操作和执行CALL操作时有所不同。发生中断时首先压栈的也是PC的低16位,不同之处在于第二次压栈时,16位数据中除了PC高7位以外,还分别把1位IPL3(位于CORCON里)和8位SRL寄存器的信息也组合到这个字的高9位。这个操作都是硬件自动完成的,不需要用户操心。

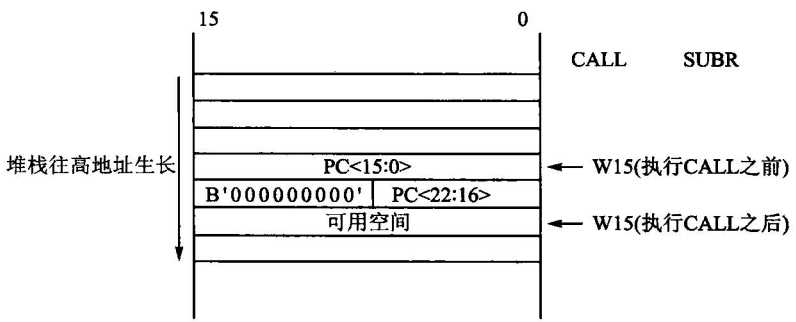

图1.3 执行CALL指令时堆栈操作情况

1.3.1 软件堆栈示例

用户可以使用PUSH和POP指令控制软件堆栈的压栈和出栈操作。同时PUSH和POP指令也可以等价于将W15用作目标指针的MOV指令。

例如:用户想要把W0的数据内容压入堆栈可通过执行"PUSH W0"指令来实现,也可以通过执行"MOV W0,[W15++]"指令来实现。

再例如:要把栈顶的内容弹出到W0寄存器里,可通过执行"POP W0"指令来实现,也可以通过执行"MOV [--W15], W0"指令来实现。

图1.4给出了如何使用软件堆栈的示例,可以看到当芯片初始化时,软件堆栈W15已经初始化为0x0800,并且W0和W1中有相应的初始值,此示例假设值0x5A5A和0x3636已被分别写入W0和W1。当执行第一个"PUSH W0"指令,堆栈第一次进栈,W0中包含的值被复制到堆栈中。W15自动更新以指向下一个可用的堆栈单元(0x0802)。用户还看到,当第二次执行"PUSH W1"指令时,W1的内容被压入软件堆栈。最后当执行"POP W3"指令时,堆栈里的内容出栈,即栈顶的数据(先前从W1压入的)被写入到W3里了。

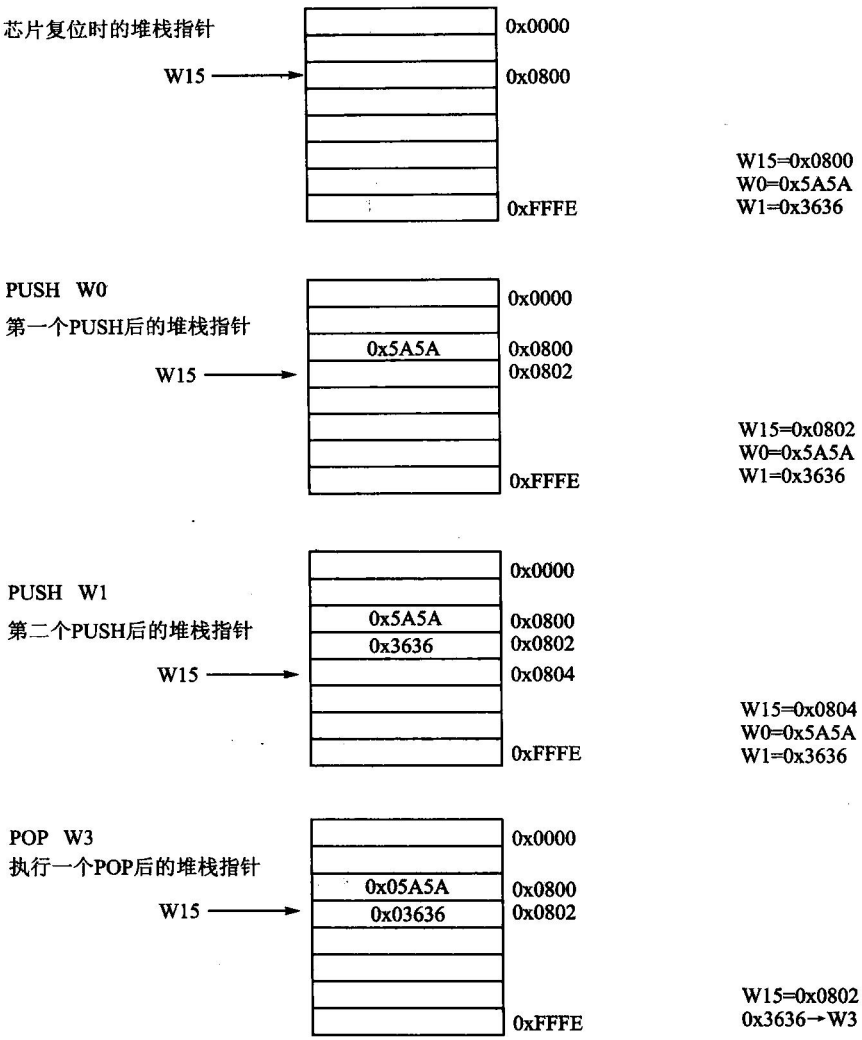

图 1.4　软件堆栈的操作过程示范

1.3.2　W14 软件堆栈帧指针

大家知道在调用子程序时可能需要保存一些用户定义的变量信息。帧(Frame)是堆栈中用户定义的供某个子程序使用的存储器段,这些信息包括用户定义的一些需要保存的变量等。在堆栈操作时帧指针(Frame Pointer)是指向用户变量和系统参数的分界点,用来对用户变量进行操作。

W14是特殊工作寄存器，通过使用LNK(link，连接)和ULNK(unlink，不连接)指令可以把W14用作堆栈帧指针。当W14不用作帧指针时，依然可以作为普通的工作寄存器使用。用户可参考指令集里对于这两条指令的描述。

1.3.3 堆栈指针上溢(Overflow)和下溢(Underflow)

栈顶限制寄存器(SPLIM)用于设置堆栈的深度，复位时为0x0000。SPLIM是一个16位的寄存器，因为所有的堆栈操作必须按照字对齐，因此SPLIM<0>被硬件强制为"0"。

芯片复位后上溢出检查是被禁止的，只有用户使用字写操作(word write)对SPLIM进行初始化后才会使能堆栈上溢检查。一旦使能上溢检查之后，只有复位芯片才能禁止上溢检查。所有将W15用作源或目标寄存器而产生的有效地址将与SPLIM中的值作比较。假如堆栈不断向上生长直到某一时刻W15的值等于SPLIM的值，此时堆栈指针指向最后一个可用空间，还可以压栈，当进行了再一次入栈操作后，W15的值比SPLIM的值大2，则此时堆栈到达最高使用极限。虽然此时不会产生堆栈上溢错误，但是不能再次压栈了，只要再做一次入栈操作就会产生上溢，生成堆栈错误陷阱。

例如：某颗芯片RAM有8 KB，现在要求堆栈指针递增超出0x2000时产生堆栈溢出错误，那么需将SPLIM初始化为0x1FFE。当最后一个堆栈位置被使用后W15指向0x2000。再次压栈将导致堆栈上溢，同时会产生堆栈错误陷阱。

注意： 堆栈错误陷阱可以由任何使用W15寄存器的内容来产生有效地址(EA)的指令引起。所以如果W15的内容比SPLIM寄存器的内容大2，并且执行了一条CALL指令或发生了中断，那么将产生堆栈错误陷阱。

如果已经使能了堆栈上溢检查，W15有效地址计算绕过了数据空间的末尾(0xFFFF)，堆栈错误陷阱仍将产生。

注意： 对SPLIM进行赋值操作指令的下一条指令不能是任何使用W15的间接读操作指令。

发生复位时，堆栈初始化为0x0800(0x0000～0x07FF之间的空间保留给SFR)。如果堆栈指针地址小于0x0800会产生堆栈下溢出，并产生堆栈错误陷阱。

1.4 与核心相关的寄存器

1.4.1 状态寄存器(SR)

表1.2所列为状态寄存器(SR)各位的定义。状态寄存器是一个相当重要的特殊功能寄存器，其长度为16位，分为高8位和低8位两个部分。其中低字节称为低状态寄存器(SRL)，

高字节称为高状态寄存器(SRH)。两个部分分别负责 DSP 和 MCU 的状态信息。这些状态位给用户提供各种运算信息或状态信息,从而决定程序的分支跳转情况。从指令集里可以看到,很多判断跳转指令都和状态位信息相关。

SRL 主要包含了所有的与 MCU 核心 ALU 操作状态相关的标志,还有 CPU 中断优先级状态位 IPL<2:0>和 REPEAT 循环有效状态位 RA(SR<4>)。中断处理期间,SRL 的 8 位与 IPL3 位,加上 PC 的高 7 位相连以形成一个完整的 16 位字被压入堆栈中保护起来。

SRH 主要包含了与 DSP 相关的加法器/减法器状态位、DO 循环有效位 DA(SR<9>)和辅助进位标志位 DC(SR<8>)。

状态寄存器里有些位可读或写,但有些位却例外,比如 DA、RA 是只读位;OA、OB、OAB 位只读而且只能被 DSP 引擎硬件修改;SA、SB、SAB 位是只读只清零的,而且只能被 DSP 引擎硬件置"1",一旦被置"1",它们就保持置位状态直到被用户清零,与任何随后的 DSP 操作的结果无关。

注意: 清零 SAB 位的同时将清零 SA 位和 SB 位。

表 1.2 状态寄存器(SR)

R-0	R-0	R/C-0	R/C-0	R-0	R/C-0	R-0	R-0
OA	OB	SA	SB	OAB	SAB	DA	DC
bit 15							bit 8
R/W-0	R/W-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
IPL<2:0>			RA	N	OV	Z	C
bit 7							bit 0

其中:R=可读位,W=可写位,C=只能被清零,U=未用(读作 0),-n=上电复位时的值

Bit15	**OA**:累加器 A 溢出状态位	1=溢出 0=未溢出
Bit14	**OB**:累加器 B 溢出状态位	1=溢出 0=未溢出
Bit13	**SA**:累加器 A 饱和位	1=饱和或曾经饱和 0=未饱和
Bit12	**SB**:累加器 B 饱和位	1=饱和或曾经饱和 0=未饱和
Bit11	**OAB**:累加器溢出标志位	1=累加器 A 或 B 有溢出 0=累加器 A 或 B 没有溢出
Bit10	**SAB**:累加器饱和标志位	1=A 或 B 饱和或曾经饱和 0=没有饱和

续表 1.2

Bit9	**DA**：DO 指令状态位	1＝DO 循环在运行中 0＝DO 循环没有运行
Bit8	**DC**：MCU 的半进位/借位	1＝运算时第 4 位有进位(字节)，或第 8 位有进位(字) 0＝没有半进位
Bit7～5	**IPL**＜2：0＞： CPU 中断优先级状态位	111＝优先级 7(15)，禁止用户中断　011＝优先级 3(11) 110＝优先级 6(14)　010＝优先级 2(10) 101＝优先级 5(13)　001＝优先级 1(9) 100＝优先级 4(12)　000＝优先级 0(8)
Bit4	**RA**：REPEAT 指令状态位	1＝REPEAT 循环在运行中 0＝REPEAT 循环没有运行
Bit3	**N**：MCU 的 ALU 负标志位	1＝ALU 运算结果为负 0＝ALU 运算结果非负
Bit2	**OV**：MCU 的 ALU 溢出标志位	1＝有符号数学运算发生溢出 0＝没有溢出
Bit1	**Z**：MCU 的 ALU 零标志位	1＝运算结果为零 0＝运算结果非零
Bit0	**C**：MCU 的 ALU 进位/借位	1＝运算结果有进位 0＝运算结果无进位

1.4.2 核心控制寄存器(CORCON)

表 1.3 所列为核心控制寄存器各位的分布和相应的含义。核心控制寄存器也是一个至关重要的寄存器，其中包含了与 DSP 乘法器、DO 循环硬件操作相关的各种信息。

同时在 CORCON 寄存器里还包含 IPL3 状态位，它与 IPL＜2：0＞(SR＜7：5＞)相连以形成 CPU 中断优先级的级别。请注意对于用户来说 IPL3 是只读的。

可通过核心控制寄存器(CORCON)选择 DSP 引擎的各种特性。这些特性包括：

- 小数或整数乘法操作；
- 传统或收敛舍入；
- 用于 ACCA 的自动饱和度开/关；
- 用于 ACCB 的自动饱和度开/关；
- 用于写数据存储器的自动饱和度开/关；
- 自动饱和度模式选择。

表 1.3　芯片控制寄存器(CORCON)

U-0	U-0	U-0	R/W-0	R/W-0	R-0	R-0	R-0
—	—	—	US	EDT	DL<1:0>		
bit 15							bit 8

R/W-0	R/W-0	R/W-1	R/W-0	R/C-0	R/W-0	R/W-0	R/W-0
SATA	SATB	SATDW	ACCSAT	IPL3	PSV	RND	IF
bit 7							bit 0

其中：R＝可读位，W ＝可写位，C ＝只能被清零，U ＝未用(读作 0)，－n ＝上电复位时的值

Bit15～13	未用	读作 0
Bit12	**US**： DSP 乘法无符号/带符号控制位	1＝DSP 引擎乘法带符号 0＝DSP 引擎乘法无符号
Bit11	**EDT**： DO 循环提前终止控制位	1＝当前 DO 循环结束后终止 DO 循环 0＝无影响
Bit10～8	**DL<2：0>**： DO 循环嵌套级状态位	111＝7 个 DO 循环有效　011＝3 个 DO 循环有效 110＝6 个 DO 循环有效　010＝2 个 DO 循环有效 101＝5 个 DO 循环有效　001＝1 个 DO 循环有效 100＝4 个 DO 循环有效　000＝0 个 DO 循环有效
Bit7	**SATA**： ACCA 饱和使能位	1＝使能累加器 A 饱和 0＝禁止累加器 A 饱和
Bit6	**SATB**： ACCB 饱和使能位	1＝使能累加器 B 饱和 0＝禁止累加器 B 饱和
Bit5	**SATDW**： 来自 DSP 引擎的数据空间写操作饱和使能位	1＝使能数据空间写操作饱和 0＝禁止数据空间写操作饱和
Bit4	**ACCSAT**： 累加器饱和模式选择位	1＝9.31 饱和(超级饱和) 0＝1.31 饱和(正常饱和)
Bit3	**IPL3**： CPU 中断优先级状态位 3	1＝CPU 优先级高于 7 0＝CPU 优先级等于或低于 7
Bit2	**PSV**： 数据空间可视使能位	1＝允许 PSV 0＝禁止 PSV
Bit1	**RND**： 舍入模式选择位	1＝使能带偏置的(传统)舍入 0＝使能非偏置(收敛)舍入
Bit0	**IF**： 整数或小数乘法器模式选择位	1＝使能 DSP 乘法运算器的整数模式 0＝使能 DSP 乘法运算器的小数模式

1.4.3 其他CPU控制寄存器

与内核相关的寄存器还有以下几个。这里提出来简单说明一下，本书的其他章节会对它们进行更详细的描述。

1. TBLPAG：表页寄存器

TBLPAG寄存器用于在表读和表写操作过程中保存程序存储器地址的高8位。当然用户可以用宏汇编指令来获取表页寄存器，简化程序设计。

2. PSVPAG：PSV页寄存器

程序空间可视性(PSV)允许用户将程序存储空间的32 KB窗口映射到数据空间的高32 KB范围。这样就可以用访问RAM的指令去访问Flash里的数据(透明访问)。PSVPAG寄存器用于选择将Flash空间里的某一个32 KB区域映射到数据空间。

3. MODCON：模控制寄存器

MODCON寄存器用于模寻址(循环缓冲)的使能和各项配置。

4. XMODSRT，XMODEND：X模起始和结束地址寄存器

在X数据空间中执行模数寻址时，XMODSRT、XMODEND寄存器用于存放模缓冲区的起始地址和结束地址。

5. YMODSRT，YMODEND：Y模起始和结束地址寄存器

在Y数据空间中执行模数寻址时，YMODSRT、YMODEND寄存器用于存放模缓冲区的起始地址和结束地址。

6. XBREV：X模位反转寄存器

XBREV寄存器用于设置位反转寻址的缓冲区大小。

7. DISICNT：禁止中断计数寄存器

使用DISI指令可以禁止优先级为1～6的中断，禁止时间为DISICNT寄存器指定周期数。

1.5 算术逻辑部件(ALU)

算术逻辑部件(ALU)是CPU的核心，所有的算术运算和逻辑运算都需要经过ALU的操作。PIC24和dsPIC系列16位单片机的ALU宽度为16位，能进行加、减、移位、逻辑操作。除非特别指明，算术运算一般是以二进制补码形式进行的。根据不同的操作，ALU可能会影响SR寄存器中的进位标志位(C)、零标志位(Z)、负标志位(N)、溢出标志位(OV)和辅助进位标志位(DC)的值。在减法操作中，C位和DC位分别作为借位和辅助借位使用。

根据所使用的指令不同，ALU可以执行8位或16位操作。根据指令的寻址模式，ALU操作的数据可以来自W寄存器阵列或数据存储器。同样ALU的输出数据可以被写入W寄

存器阵列或数据存储单元。

以下两点需要提醒用户特别注意：

① 使用 16 位 ALU 进行的字节操作可以产生超过 8 位的结果。为了保持 PIC 系列的向后兼容性，字节操作结果只取低字节结果(不修改高字节)。SR 寄存器的状态只根据运算结果低字节的状态进行更新。

② 字节模式中执行的所有寄存器指令只会影响 W 寄存器的低字节。可以使用访问 W 寄存器的存储器映射内容的文件寄存器指令修改任何 W 寄存器的高字节。

有两条指令可以进行 8 位和 16 位的转换操作：第一条是符号扩展(SE)指令，ALU 可以将 W 寄存器或 RAM 中的一个 8 位字节进行符号扩展处理，将 16 位结果保存在 W 寄存器中。第二条是零扩展(ZE)指令，ALU 可以将 W 寄存器或 RAM 里的一个 16 位数据的高 8 位清零，并将结果存放在 16 位的 W 寄存器中。

1.6 DSP 引擎

DSP 引擎是一个相对独立的硬件模块，但是它和 MCU 核融合在一起，共享很多公共资源。DSP 核同样使用 W 寄存器堆为自己的运算服务，但它有一些自己专用的结果寄存器。DSP 引擎与 MCU 有着相同的指令译码器单元。W 寄存器堆用于产生有效地址(EA)。尽管 MCU 和 DSP 引擎共享很多资源并相对独立，但是指令流是顺序的，DSP 和 MCU 指令并不能同步运行。执行到 DSP 类指令时 DSP 引擎动作，执行到 MCU 类指令时 MCU 核动作。

图 1.5 所示为 DSP 引擎原理框图。可以看到 DSP 引擎的核心由一个高速 17 位×17 位乘法器、40 位加法/减法器、两个 40 位累加器组成，配合 40 位长度的桶形移位寄存器、带可选模式的舍入逻辑、带可选模式的饱和逻辑，整个 DSP 核可以进行快速有效率的 DSP 运算。

DSP 引擎的输入数据可能来自两个途径：首先，假如执行双操作数 DSP 类指令时，数据通过 X 总线和 Y 总线同时预取得到(使用寄存器 W4、W5、W6 或 W7)；其次，假如执行所有其他 DSP 类指令时，数据来自 X 总线。

DSP 引擎的输出数据可能被输出到两个地方：首先，可能输出到由该 DSP 类指令指定的累加器(ACCA 或 ACCB)里；其次，还可能通过 X 总线被输出到 RAM 中的任何单元里。

MCU 的移位和乘法指令使用了 DSP 引擎的一些硬件。这些操作中数据的读写是通过 X 总线实现的。

1.6.1 累加器(Accumulators)

所有的 dsPIC 单片机内部的 DSP 引擎都有两个 40 位数据累加器 ACCA 和 ACCB，它们是 DSP 指令的结果寄存器。每个累加器可以分为 3 个寄存器，分别是 ACCxL(ACCx<

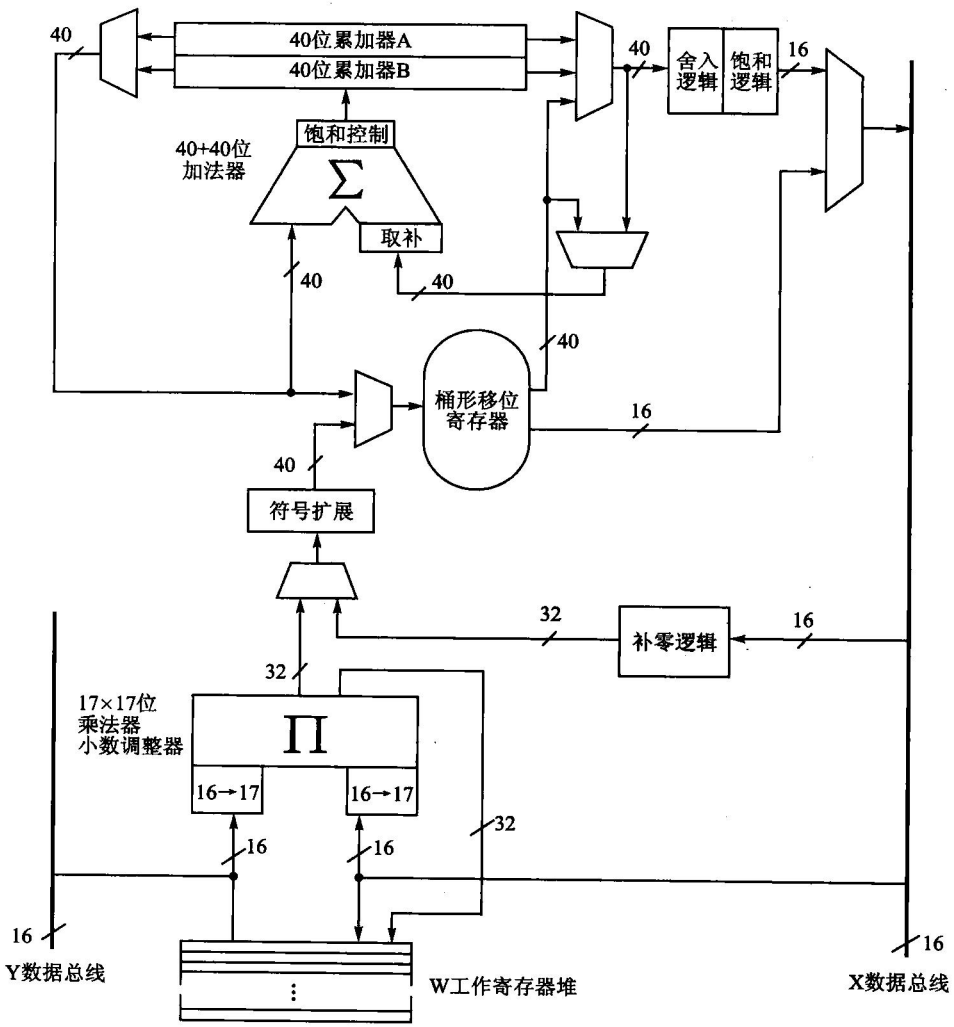

图 1.5　DSP 引擎原理框图

15：0>)、ACCxH(ACCx<31：16>)、ACCxU(ACCx<39：32>)。其中“x”表示某一个累加器(A 或 B)。这三部分都有自己的映射地址。

对于使用累加器的小数操作，小数点位于第 31 位的右边。存储在每个累加器中的小数值范围在 $-256.0\sim(256.0-2^{-31})$。对于使用累加器的整数操作，小数点位于第 0 位的右边。存储在每个累加器中的整数值范围为−549 755 813 888 到+549 755 813 887。