Section 2. CPU

HIGHLIGHTS

This section of the manual contains the following topics:

2.1 Introduction

The dsPIC30F CPU module has a 16-bit (data) modified Harvard architecture with an enhanced instruction set, including significant support for DSP. The CPU has a 24-bit instruction word, with a variable length opcode field. The program counter (PC) is 24-bits wide and addresses up to 4M x 24 bits of user program memory space. A single cycle instruction prefetch mechanism is used to help maintain throughput and provides predictable execution. All instructions execute in a single cycle, with the exception of instructions that change the program flow, the double-word move (MOV.D) instruction and the table instructions. Overhead free program loop constructs are supported using the DO and REPEAT instructions, both of which are interruptible at any point.

The dsPIC30F devices have sixteen 16-bit working registers in the programmer's model. Each of the working registers can act as a data, address or address offset register. The 16th working register (W15) operates as a software stack pointer for interrupts and calls.

The dsPIC30F instruction set has two classes of instructions; the MCU class of instructions and the DSP class of instructions. These two instruction classes are seamlessly integrated into the architecture and execute from a single execution unit. The instruction set includes many Addressing modes and was designed for optimum C compiler efficiency.

The data space can be addressed as 32K words or 64 Kbytes and is split into two blocks, referred to as X and Y data memory. Each memory block has its own independent Address Generation Unit (AGU). The MCU class of instructions operate solely through the X memory AGU, which accesses the entire memory map as one linear data space. Certain DSP instructions operate through the X and Y AGUs to support dual operand reads, which splits the data address space into two parts. The X and Y data space boundary is device specific.

The upper 32 Kbytes of the data space memory map can optionally be mapped into program space at any 16K program word boundary defined by the 8-bit Program Space Visibility Page (PSVPAG) register. The program to data space mapping feature lets any instruction access program space as if it were data space. Furthermore, RAM may be connected to the program memory bus on devices with an external bus and used to extend the internal data RAM.

Overhead free circular buffers (modulo addressing) are supported in both X and Y address spaces. The modulo addressing removes the software boundary checking overhead for DSP algorithms. Furthermore, the X AGU circular addressing can be used with any of the MCU class of instructions. The X AGU also supports bit-reverse addressing to greatly simplify input or output data reordering for radix-2 FFT algorithms.

The CPU supports Inherent (no operand), Relative, Literal, Memory Direct, Register Direct and Register Indirect Addressing modes. Each instruction is associated with a predefined Addressing mode group depending upon its functional requirements. As many as six Addressing modes are supported for each instruction.

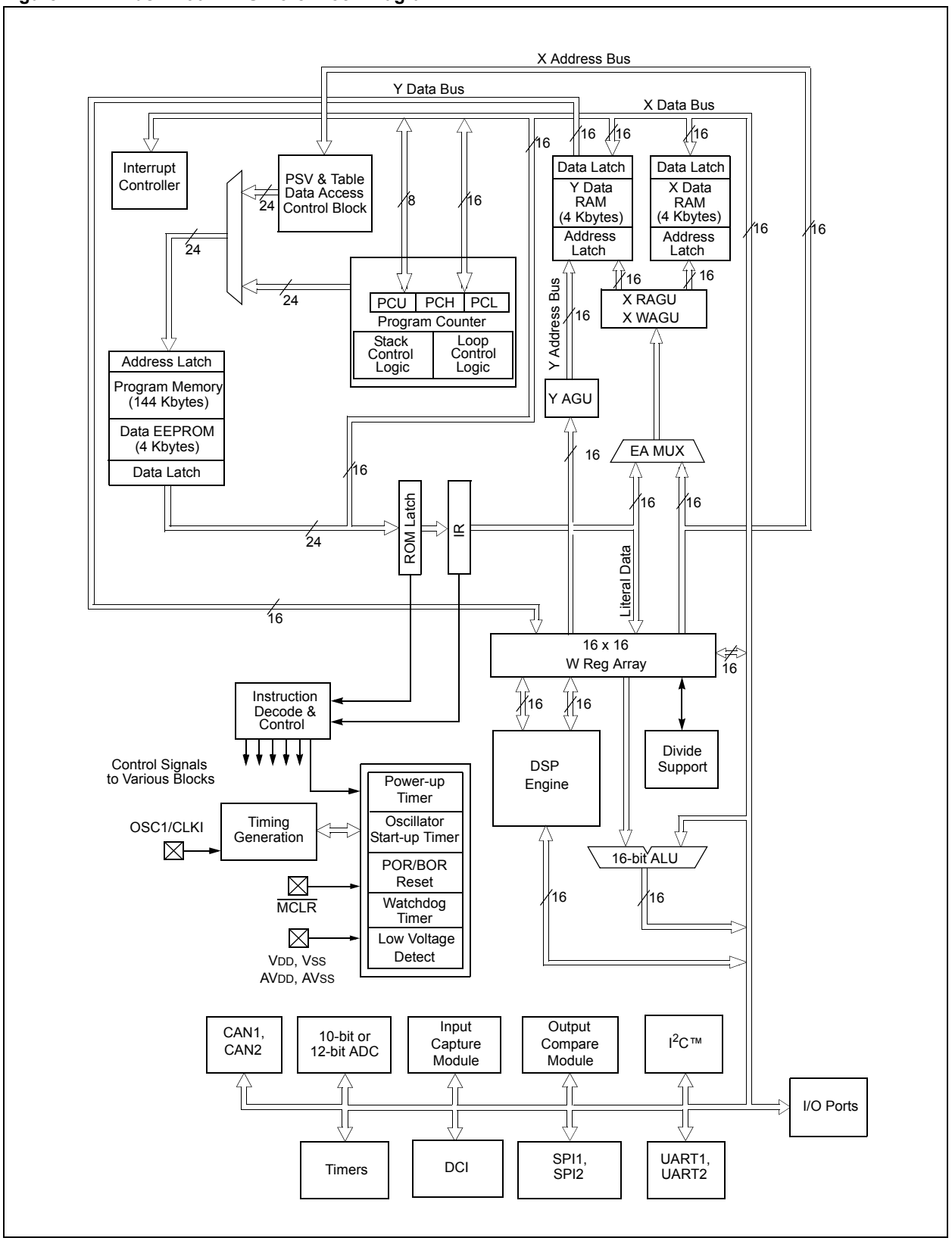
For most instructions, the dsPIC30F is capable of executing a data (or program data) memory read, a working register (data) read, a data memory write and a program (instruction) memory read per instruction cycle. As a result, 3 operand instructions can be supported, allowing A+B =C operations to be executed in a single cycle.

The DSP engine features a high speed, 17-bit by 17-bit multiplier, a 40-bit ALU, two 40-bit saturating accumulators and a 40-bit bi-directional barrel shifter. The barrel shifter is capable of shifting a 40-bit value up to 15 bits right, or up to 16 bits left, in a single cycle. The DSP instructions operate seamlessly with all other instructions and have been designed for optimal real-time performance. The MAC instruction and other associated instructions can concurrently fetch two data operands from memory while multiplying two W registers. This requires that the data space be split for these instructions and linear for all others. This is achieved in a transparent and flexible manner through dedicating certain working registers to each address space.

The dsPIC30F has a vectored exception scheme with up to 8 sources of non-maskable traps and 54 interrupt sources. Each interrupt source can be assigned to one of seven priority levels.

Figure 2-1 shows a block diagram of the CPU.

Figure 2-1: dsPIC30F CPU Core Block Diagram



2.2 Programmer's Model

The programmer's model for the dsPIC30F is shown in Figure 2-2. All registers in the programmer's model are memory mapped and can be manipulated directly by instructions. A description of each register is provided in Table 2-1.

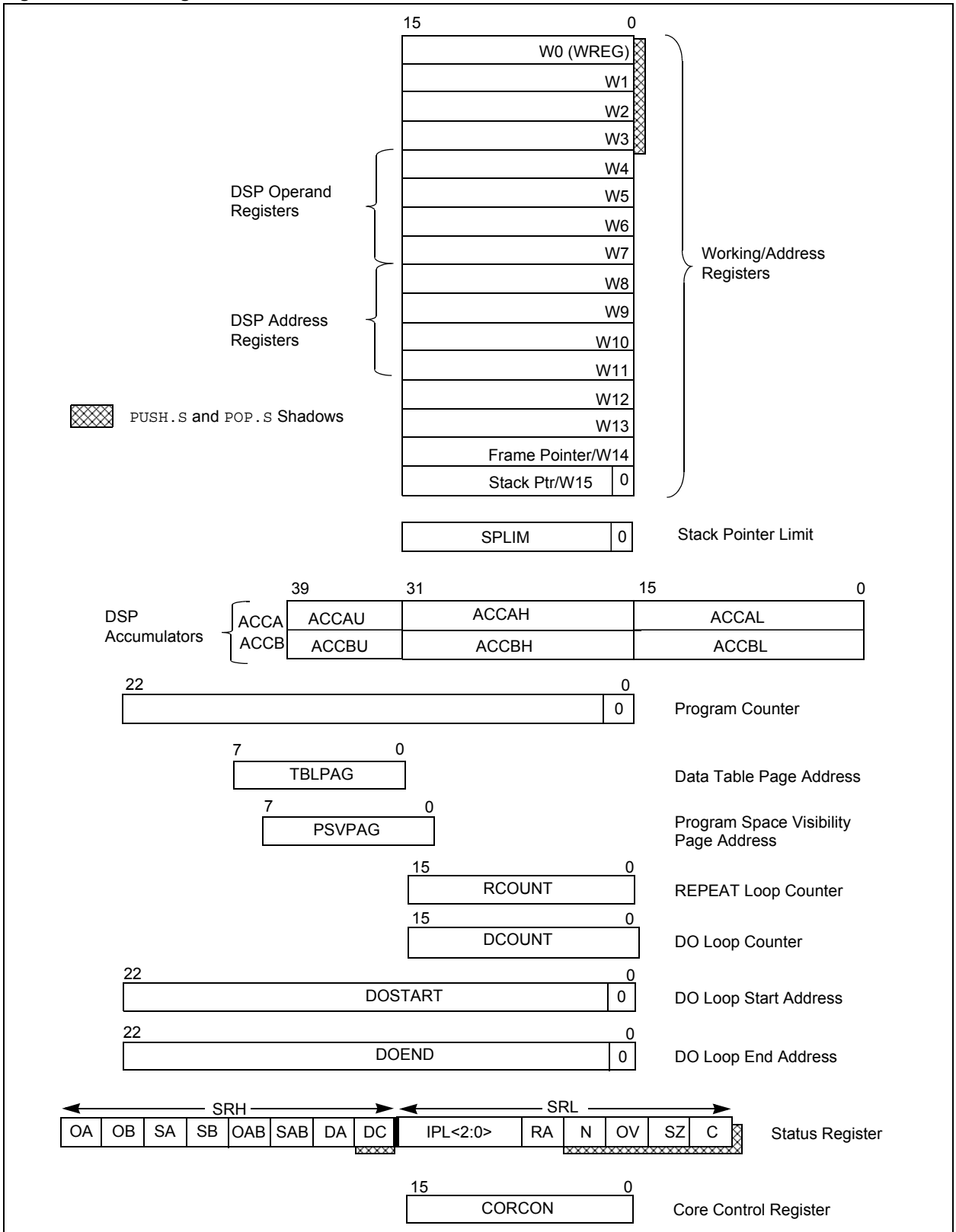
Table 2-1: Programmer's Model Register Descriptions

Register(s) Name	Description
W0 through W15	Working register array
ACCA, ACCB	40-bit DSP Accumulators
PC	23-bit Program Counter
SR	ALU and DSP Engine Status register
SPLIM	Stack Pointer Limit Value register
TBLPAG	Table Memory Page Address register
PSVPAG	Program Space Visibility Page Address register
RCOUNT	REPEAT Loop Count register
DCOUNT	DO Loop Count register
DOSTART	DO Loop Start Address register
DOEND	DO Loop End Address register
CORCON	Contains DSP Engine and DO Loop control bits

In addition to the registers contained in the programmer's model, the dsPIC30F contains control registers for modulo addressing, bit-reversed addressing and interrupts. These registers are described in subsequent sections of this document.

All registers associated with the programmer's model are memory mapped, as shown in Table 2-8.

Figure 2-2: Programmer's Model



Note: DCOUNT, DOSTART and DOEND have one level of shadow registers (not shown) for nested DO loops.

2.2.1 Working Register Array

The 16 working (W) registers can function as data, address or address offset registers. The function of a W register is determined by the Addressing mode of the instruction that accesses it.

The dsPIC30F instruction set can be divided into two instruction types: register and file register instructions. Register instructions can use each W register as a data value or an address offset value. For example:

```
MOV    W0,W1      ; move contents of W0 to W1
MOV    W0,[W1]    ; move W0 to address contained in W1
ADD    W0,[W4],W5 ; add contents of W0 to contents pointed
                  ; to by W4.  Place result in W5.
```

2.2.1.1 W0 and File Register Instructions

W0 is a special working register because it is the only working register that can be used in file register instructions. File register instructions operate on a specific memory address contained in the instruction opcode and W0. W1-W15 cannot be specified as a target register in file register instructions.

The file register instructions provide backward compatibility with existing PIC® devices which have only one W register. The label 'WREG' is used in the assembler syntax to denote W0 in a file register instruction. For example:

```
MOV    WREG,0x0100   ; move contents of W0 to address 0x0100
ADD    0x0100,WREG   ; add W0 to address 0x0100, store in W0
```

Note:	For a complete description of Addressing modes and instruction syntax, please refer to the "*dsPIC30F Programmer's Reference Manual*" (DS70157).

2.2.1.2 W Register Memory Mapping

Since the W registers are memory mapped, it is possible to access a W register in a file register instruction as shown below:

```
MOV    0x0004, W10   ; equivalent to MOV W2, W10
```

where `0x0004` is the address in memory of W2.

Further, it is also possible to execute an instruction that will attempt to use a W register as both an address pointer and operand destination. For example:

```
MOV    W1,[W2++]
```

where:

```
W1 = 0x1234
W2 = 0x0004      ;[W2] addresses W2
```

In the example above, the contents of W2 are 0x0004. Since W2 is used as an address pointer, it points to location `0x0004` in memory. W2 is also mapped to this address in memory. Even though this is an unlikely event, it is impossible to detect until run-time. The dsPIC30F ensures that the data write will dominate, resulting in W2 = 0x1234 in the example above.

2.2.1.3 W Registers and Byte Mode Instructions

Byte instructions which target the W register array only affect the Least Significant Byte of the target register. Since the working registers are memory mapped, the Least and Most Significant Bytes can be manipulated through byte wide data memory space accesses.

2.2.2 Shadow Registers

Many of the registers in the programmer's model have an associated shadow register as shown in Figure 2-2. None of the shadow registers are accessible directly. There are two types of shadow registers: those utilized by the `PUSH.S` and `POP.S` instructions and those utilized by the `DO` instruction.

2.2.2.1 PUSH.S and POP.S Shadow Registers

The PUSH.S and POP.S instructions are useful for fast context save/restore during a function call or Interrupt Service Routine (ISR). The PUSH.S instruction will transfer the following register values into their respective shadow registers:

- W0...W3
- SR (N, OV, Z , C, DC bits only)

The POP.S instruction will restore the values from the shadow registers into these register locations. A code example using the PUSH.S and POP.S instructions is shown below:

```
MyFunction:

    PUSH.S              ; Save W registers, MCU status
    MOV    #0x03,W0     ; load a literal value into W0
    ADD    RAM100       ; add W0 to contents of RAM100
    BTSC   SR,#Z        ; is the result 0?
    BSET   Flags,#IsZero; Yes, set a flag
    POP.S               ; Restore W regs, MCU status
    RETURN
```

The PUSH.S instruction will overwrite the contents previously saved in the shadow registers. The shadow registers are only one level in depth, so care must be taken if the shadow registers are to be used for multiple software tasks.

The user must ensure that any task using the shadow registers will not be interrupted by a higher priority task that also uses the shadow registers. If the higher priority task is allowed to interrupt the lower priority task, the contents of the shadow registers saved in the lower priority task will be overwritten by the higher priority task.

2.2.2.2 DO Loop Shadow Registers

The following registers are automatically saved in shadow registers when a DO instruction is executed:

- DOSTART
- DOEND
- DCOUNT

The DO shadow registers are one level in depth, permitting two loops to be automatically nested. Refer to **Section 2.9.2.2 "DO Loop Nesting"** for further details.

2.2.3 Uninitialized W Register Reset

The W register array (with the exception of W15) is cleared during all Resets and is considered uninitialized until written to. An attempt to use an uninitialized register as an address pointer will reset the device.

A word write must be performed to initialize a W register. A byte write will not affect the initialization detection logic.

2.3 Software Stack Pointer

W15 serves as a dedicated software stack pointer and is automatically modified by exception processing, subroutine calls and returns. However, W15 can be referenced by any instruction in the same manner as all other W registers. This simplifies reading, writing and manipulating the stack pointer (for example, creating stack frames).

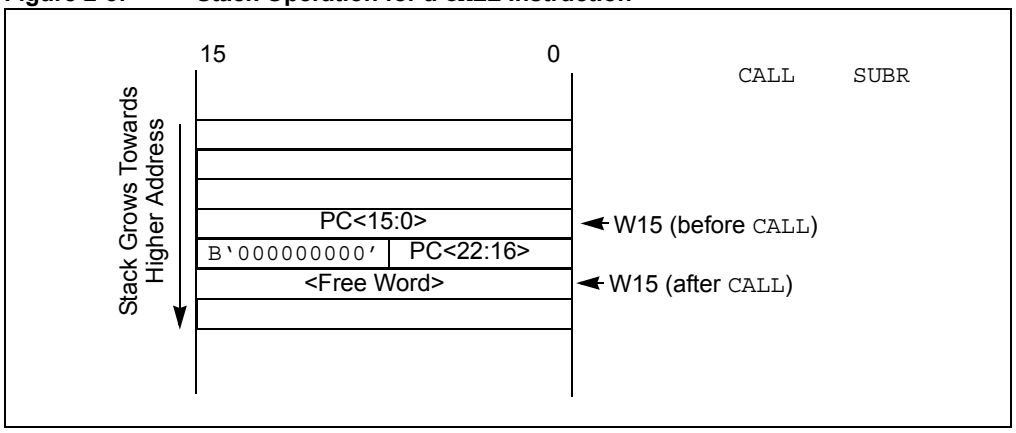
Note:	In order to protect against misaligned stack accesses, W15<0> is fixed to '0' by the hardware.

W15 is initialized to `0x0800` during all Resets. This address ensures that the stack pointer (SP) will point to valid RAM in all dsPIC30F devices and permits stack availability for non-maskable trap exceptions, which may occur before the SP is initialized by the user software. The user may reprogram the SP during initialization to any location within the data space.

The stack pointer always points to the first available free word and fills the software stack working from lower towards higher addresses. It pre-decrements for a stack pop (read) and post-increments for a stack push (writes), as shown in Figure 2-3.

When the PC is pushed onto the stack, PC<15:0> is pushed onto the first available stack word, then PC<22:16> is pushed into the second available stack location. For a PC push during any `CALL` instruction, the MSByte of the PC is zero-extended before the push as shown in Figure 2-3. During exception processing, the MSByte of the PC is concatenated with the lower 8 bits of the CPU status register, SR. This allows the contents of SRL to be preserved automatically during interrupt processing.

Figure 2-3: Stack Operation for a `CALL` Instruction



2.3.1 Software Stack Examples

The software stack is manipulated using the PUSH and POP instructions. The PUSH and POP instructions are the equivalent of a MOV instruction with W15 used as the destination pointer. For example, the contents of W0 can be pushed onto the stack by:

```
PUSH W0
```

This syntax is equivalent to:

```
MOV W0,[W15++]
```

The contents of the top-of-stack can be returned to W0 by:

```
POP W0
```

This syntax is equivalent to:

```
MOV [--W15],W0
```

Figure 2-4 through Figure 2-7 show examples of how the software stack is used. Figure 2-4 shows the software stack at device initialization. W15 has been initialized to 0x0800. Furthermore, this example assumes the values 0x5A5A and 0x3636 have been written to W0 and W1, respectively. The stack is pushed for the first time in Figure 2-5 and the value contained in W0 is copied to the stack. W15 is automatically updated to point to the next available stack location (0x0802). In Figure 2-6, the contents of W1 are pushed onto the stack. In Figure 2-7, the stack is popped and the top-of-stack value (previously pushed from W1) is written to W3.

Figure 2-4: Stack Pointer at Device Reset

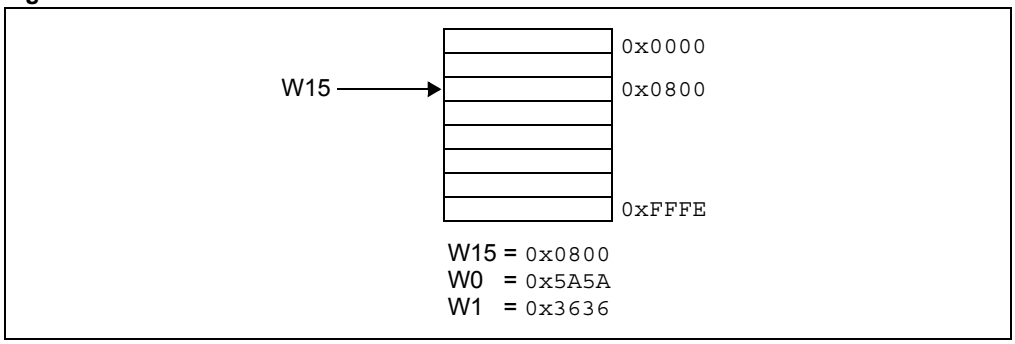


Figure 2-5: Stack Pointer After the First PUSH Instruction

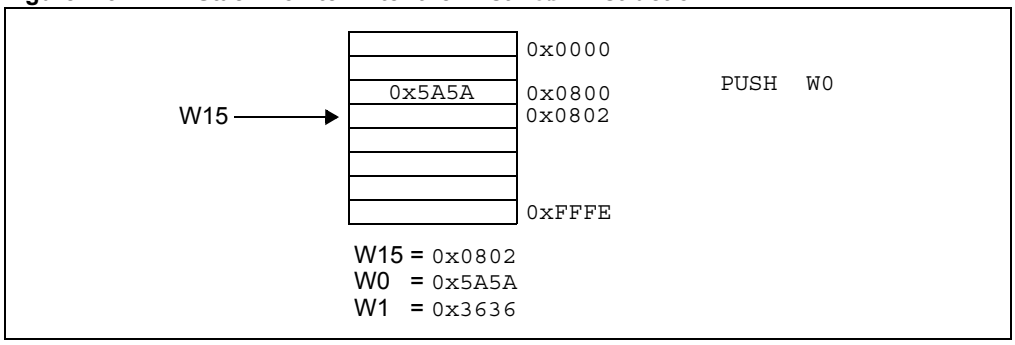


Figure 2-6: Stack Pointer After the Second PUSH Instruction

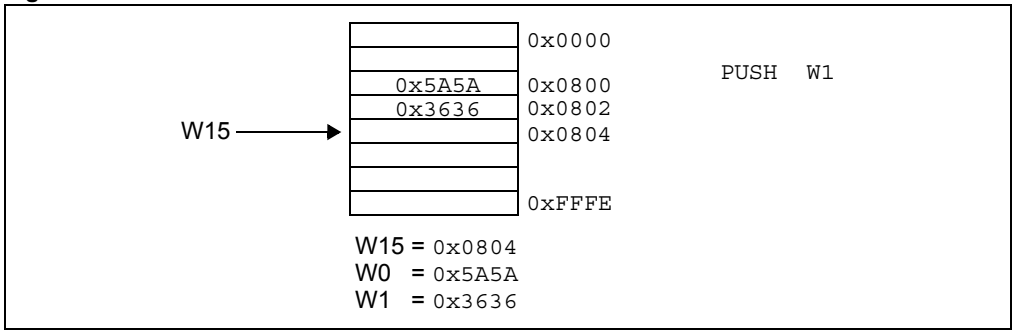


Figure 2-7: Stack Pointer After a POP Instruction

2.3.2 W14 Software Stack Frame Pointer

A frame is a user defined section of memory in the stack that is used by a single subroutine. W14 is a special working register because it can be used as a stack frame pointer with the LNK (link) and ULNK (unlink) instructions. W14 can be used in a normal working register by instructions when it is not used as a frame pointer.

Refer to the "*dsPIC30F Programmer's Reference Manual*" (DS70157) for software examples that use W14 as a stack frame pointer.

2.3.3 Stack Pointer Overflow

There is a stack limit register (SPLIM) associated with the stack pointer that is reset to 0x0000. SPLIM is a 16-bit register, but SPLIM<0> is fixed to '0' because all stack operations must be word aligned.

The stack overflow check will not be enabled until a word write to SPLIM occurs, after which time it can only be disabled by a device Reset. All effective addresses generated using W15 as a source or destination are compared against the value in SPLIM. If the contents of the Stack Pointer (W15) are greater than the contents of the SPLIM register by 2 and a push operation is performed, a Stack Error Trap will not occur. The Stack Error Trap will occur on a subsequent push operation. Thus, for example, if it is desirable to cause a Stack Error Trap when the stack grows beyond address 0x2000 in RAM, initialize the SPLIM with the value, 0x1FFE.

Note:	A Stack Error Trap may be caused by any instruction that uses the contents of the W15 register to generate an effective address (EA). Thus, if the contents of W15 are greater than the contents of the SPLIM register by 2, and a CALL instruction is executed, or if an interrupt occurs, a Stack Error Trap will be generated.

If stack overflow checking has been enabled, a stack error trap will also occur if the W15 effective address calculation wraps over the end of data space (0xFFFF).

Note:	A write to the Stack Pointer Limit register, SPLIM, should not be followed by an indirect read operation using W15.

Refer to **Section 6. "Reset Interrupts"** for more information on the stack error trap.

2.3.4 Stack Pointer Underflow

The stack is initialized to 0x0800 during Reset. A stack error trap will be initiated should the stack pointer address ever be less than 0x0800.

Note:	Locations in data space between 0x0000 and 0x07FF are, in general, reserved for core and peripheral special function registers.

2.4 CPU Register Descriptions

2.4.1 SR: CPU Status Register

The dsPIC30F CPU has a 16-bit status register (SR), the LSByte of which is referred to as the lower status register (SRL). The upper byte of SR is referred to as SRH. A detailed description of SR is shown in Register 2-1.

SRL contains all the MCU ALU operation status flags, plus the CPU interrupt priority status bits, IPL<2:0> and the REPEAT loop active status bit, RA (SR<4>). During exception processing, SRL is concatenated with the MSByte of the PC to form a complete word value, which is then stacked.

SRH contains the DSP Adder/Subtractor status bits, the DO loop active bit, DA (SR<9>) and the Digit Carry bit, DC (SR<8>).

The SR bits are readable/writable with the following exceptions:

1. The DA bit (SR<8>): DA is a read only bit.
2. The RA bit (SR<4>): RA is a read only bit.
3. The OA, OB (SR<15:14>) and OAB (SR<11>) bits: These bits are read only and can only be modified by the DSP engine hardware.
4. The SA, SB (SR<13:12>) and SAB (SR<10>) bits: These are read and clear only and can only be set by the DSP engine hardware. Once set, they remain set until cleared by the user, irrespective of the results from any subsequent DSP operations.

Note:	Clearing the SAB bit will also clear both the SA and SB bits.

Note:	A description of the SR bits affected by each instruction is provided in the "*dsPIC30F Programmer's Reference Manual*" (DS70157).

2.4.2 CORCON: Core Control Register

The CORCON register contains bits that control the operation of the DSP multiplier and DO loop hardware. The CORCON register also contains the IPL3 status bit, which is concatenated with IPL<2:0> (SR<7:5>), to form the CPU Interrupt Priority Level.

Register 2-1: SR: CPU Status Register

Upper Byte:							
R-0	R-0	R/C-0	R/C-0	R-0	R/C-0	R -0	R/W-0
OA	OB	SA	SB	OAB	SAB	DA	DC
bit 15							bit 8

Lower Byte: (SRL)							
R/W-0(2)	R/W-0(2)	R/W-0(2)	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

bit 15 **OA:** Accumulator A Overflow Status bit
1 = Accumulator A overflowed
0 = Accumulator A has not overflowed

bit 14 **OB:** Accumulator B Overflow Status bit
1 = Accumulator B overflowed
0 = Accumulator B has not overflowed

bit 13 **SA:** Accumulator A Saturation 'Sticky' Status bit
1 = Accumulator A is saturated or has been saturated at some time
0 = Accumulator A is not saturated

Note: This bit may be read or cleared (not set).

bit 12 **SB:** Accumulator B Saturation 'Sticky' Status bit
1 = Accumulator B is saturated or has been saturated at some time
0 = Accumulator B is not saturated

Note: This bit may be read or cleared (not set).

bit 11 **OAB:** OA || OB Combined Accumulator Overflow Status bit
1 = Accumulators A or B have overflowed
0 = Neither Accumulators A or B have overflowed

bit 10 **SAB:** SA || SB Combined Accumulator 'Sticky' Status bit
1 = Accumulators A or B are saturated or have been saturated at some time in the past
0 = Neither Accumulator A or B are saturated

Note: This bit may be read or cleared (not set). Clearing this bit will clear SA and SB.

bit 9 **DA:** DO Loop Active bit
1 = DO loop in progress
0 = DO loop not in progress

bit 8 **DC:** MCU ALU Half Carry/$\overline{\text{Borrow}}$ bit
1 = A carry-out from the 4th low order bit (for byte-sized data) or 8th low order bit (for word-sized data) of the result occurred
0 = No carry-out from the 4th low order bit (for byte-sized data) or 8th low order bit (for word-sized data) of the result occurred

Register 2-1: SR: CPU Status Register (Continued)

bit 7-5 **IPL<2:0>:** CPU Interrupt Priority Level Status bits(1)
111 = CPU Interrupt Priority Level is 7 (15). User interrupts disabled.
110 = CPU Interrupt Priority Level is 6 (14)
101 = CPU Interrupt Priority Level is 5 (13)
100 = CPU Interrupt Priority Level is 4 (12)
011 = CPU Interrupt Priority Level is 3 (11)
010 = CPU Interrupt Priority Level is 2 (10)
001 = CPU Interrupt Priority Level is 1 (9)
000 = CPU Interrupt Priority Level is 0 (8)

Note 1: The IPL<2:0> bits are concatenated with the IPL<3> bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL if IPL<3> = 1. User interrupts are disabled when IPL<3> = 1.

2: The IPL<2:0> status bits are read only when NSTDIS = 1 (INTCON1<15>).

bit 4 **RA:** REPEAT Loop Active bit
1 = REPEAT loop in progress
0 = REPEAT loop not in progress

bit 3 **N:** MCU ALU Negative bit
1 = Result was negative
0 = Result was non-negative (zero or positive)

bit 2 **OV:** MCU ALU Overflow bit
This bit is used for signed arithmetic (2's complement). It indicates an overflow of the magnitude which causes the sign bit to change state.
1 = Overflow occurred for signed arithmetic (in this arithmetic operation)
0 = No overflow occurred

bit 1 **Z:** MCU ALU Zero bit
1 = An operation which effects the Z bit has set it at some time in the past
0 = The most recent operation which effects the Z bit has cleared it (i.e., a non-zero result)

bit 0 **C:** MCU ALU Carry/$\overline{\text{Borrow}}$ bit
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred

Legend:		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
C = Clear only bit	S = Set only bit	-n = Value at POR
'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Register 2-2: CORCON: Core Control Register

Upper Byte:							
U-0	U-0	U-0	R/W-0	R/W-0	R-0	R-0	R-0
—	—	—	US	EDT	DL<2:0>		
bit 15							bit 8

Lower Byte:							
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

bit 15-13 **Unimplemented:** Read as '0'

bit 12 **US:** DSP Multiply Unsigned/Signed Control bit
1 = DSP engine multiplies are unsigned
0 = DSP engine multiplies are signed

bit 11 **EDT:** Early DO Loop Termination Control bit
1 = Terminate executing DO loop at end of current loop iteration
0 = No effect

Note: This bit will always read as '0'.

bit 10-8 **DL<2:0>:** DO Loop Nesting Level Status bits
111 = 7 DO loops active
•
•
001 = 1 DO loop active
000 = 0 DO loops active

bit 7 **SATA:** AccA Saturation Enable bit
1 = Accumulator A saturation enabled
0 = Accumulator A saturation disabled

bit 6 **SATB:** AccB Saturation Enable bit
1 = Accumulator B saturation enabled
0 = Accumulator B saturation disabled

bit 5 **SATDW:** Data Space Write from DSP Engine Saturation Enable bit
1 = Data space write saturation enabled
0 = Data space write saturation disabled

bit 4 **ACCSAT:** Accumulator Saturation Mode Select bit
1 = 9.31 saturation (super saturation)
0 = 1.31 saturation (normal saturation)

bit 3 **IPL3:** CPU Interrupt Priority Level Status bit 3
1 = CPU interrupt priority level is greater than 7
0 = CPU interrupt priority level is 7 or less

Note: The IPL3 bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU interrupt priority level.

Register 2-2: CORCON: Core Control Register (Continued)

bit 2 **PSV:** Program Space Visibility in Data Space Enable bit
1 = Program space visible in data space
0 = Program space not visible in data space

bit 1 **RND:** Rounding Mode Select bit
1 = Biased (conventional) rounding enabled
0 = Unbiased (convergent) rounding enabled

bit 0 **IF:** Integer or Fractional Multiplier Mode Select bit
1 = Integer mode enabled for DSP multiply ops
0 = Fractional mode enabled for DSP multiply ops

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	C = Bit can be cleared
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

2.4.3 Other dsPIC30F CPU Control Registers

The registers listed below are associated with the dsPIC30F CPU core, but are described in further detail in other sections of this manual.

2.4.3.1 TBLPAG: Table Page Register

The TBLPAG register is used to hold the upper 8 bits of a program memory address during table read and write operations. Table instructions are used to transfer data between program memory space and data memory space. Refer to **Section 4. "Program Memory"** for further details.

2.4.3.2 PSVPAG: Program Space Visibility Page Register

Program space visibility allows the user to map a 32-Kbyte section of the program memory space into the upper 32 Kbytes of data address space. This feature allows transparent access of constant data through dsPIC30F instructions that operate on data memory. The PSVPAG register selects the 32 Kbyte region of program memory space that is mapped to the data address space. Refer to **Section 4. "Program Memory"** for more information on the PSVPAG register.

2.4.3.3 MODCON: Modulo Control Register

The MODCON register is used to enable and configure modulo addressing (circular buffers). Refer to **Section 3. "Data Memory"** for further details on modulo addressing.

2.4.3.4 XMODSRT, XMODEND: X Modulo Start and End Address Registers

The XMODSRT and XMODEND registers hold the start and end addresses for modulo (circular) buffers implemented in the X data memory address space. Refer to **Section 3. "Data Memory"** for further details on modulo addressing.

2.4.3.5 YMODSRT, YMODEND: Y Modulo Start and End Address Registers

The YMODSRT and YMODEND registers hold the start and end addresses for modulo (circular) buffers implemented in the Y data memory address space. Refer to **Section 3. "Data Memory"** for further details on modulo addressing.

2.4.3.6 XBREV: X Modulo Bit-Reverse Register

The XBREV register is used to set the buffer size used for bit-reversed addressing. Refer to **Section 3. "Data Memory"** for further details on bit-reversed addressing.

2.4.3.7 DISICNT: Disable Interrupts Count Register

The DISICNT register is used by the `DISI` instruction to disable interrupts of priority 1-6 for a specified number of cycles. See **Section 6. "Reset Interrupts"** for further information.

2.5 Arithmetic Logic Unit (ALU)

The dsPIC30F ALU is 16-bits wide and is capable of addition, subtraction, single bit shifts and logic operations. Unless otherwise mentioned, arithmetic operations are 2's complement in nature. Depending on the operation, the ALU may affect the values of the Carry (C), Zero (Z), Negative (N), Overflow (OV) and Digit Carry (DC) status bits in the SR register. The C and DC status bits operate as a Borrow and Digit Borrow bits, respectively, for subtraction operations.

The ALU can perform 8-bit or 16-bit operations, depending on the mode of the instruction that is used. Data for the ALU operation can come from the W register array or data memory depending on the Addressing mode of the instruction. Likewise, output data from the ALU can be written to the W register array or a data memory location.

Refer to the "*dsPIC30F Programmer's Reference Manual* (DS70157)" for information on the SR bits affected by each instruction, Addressing modes and 8-bit/16-bit Instruction modes.

Note 1:	Byte operations use the 16-bit ALU and can produce results in excess of 8 bits. However, to maintain backward compatibility with PIC devices, the ALU result from all byte operations is written back as a byte (i.e., MSByte not modified), and the SR register is updated based only upon the state of the LSByte of the result.
2:	All register instructions performed in Byte mode only affect the LSByte of the W registers. The MSByte of any W register can be modified by using file register instructions that access the memory mapped contents of the W registers.

2.5.1 Byte to Word Conversion

The dsPIC30F has two instructions that are helpful when mixing 8-bit and 16-bit ALU operations. The sign-extend (SE) instruction takes a byte value in a W register or data memory and creates a sign-extended word value that is stored in a W register. The zero-extend (ZE) instruction clears the 8 MSbs of a word value in a W register or data memory and places the result in a destination W register.

2.6 DSP Engine

The DSP engine is a block of hardware which is fed data from the W register array but contains its own specialized result registers. The DSP engine is driven from the same instruction decoder that directs the MCU ALU. In addition, all operand effective addresses (EAs) are generated in the W register array. Concurrent operation with MCU instruction flow is not possible, though both the MCU ALU and DSP engine resources may be shared by all instructions in the instruction set.

The DSP engine consists of the following components:

- high speed 17-bit x 17-bit multiplier
- barrel shifter
- 40-bit adder/subtractor
- two target accumulator registers
- rounding logic with Selectable modes
- saturation logic with Selectable modes

Data input to the DSP engine is derived from one of the following sources:

- Directly from the W array (registers W4, W5, W6 or W7) for dual source operand DSP instructions. Data values for the W4, W5, W6 and W7 registers are prefetched via the X and Y memory data buses.
- From the X memory data bus for all other DSP instructions.

Data output from the DSP engine is written to one of the following destinations:

- The target accumulator, as defined by the DSP instruction being executed.
- The X memory data bus to any location in the data memory address space.

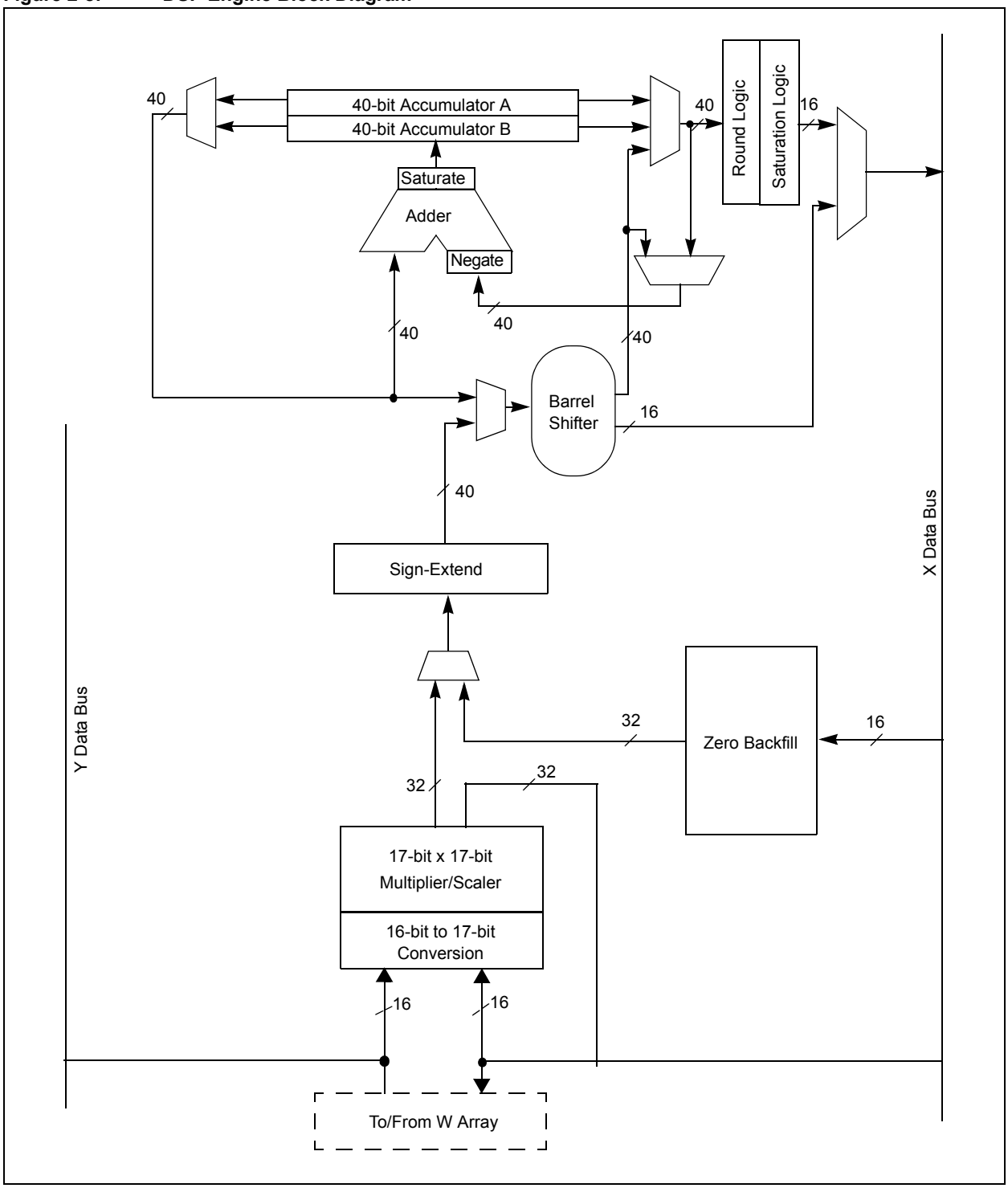
The DSP engine has the capability to perform inherent accumulator to accumulator operations which require no additional data.

The MCU shift and multiply instructions use the DSP engine hardware to obtain their results. The X memory data bus is used for data reads and writes in these operations.

A block diagram of the DSP engine is shown in Figure 2-6.

Note:	For detailed code examples and instruction syntax related to this section, refer to the "*dsPIC30F Programmer's Reference Manual*" (DS70157).

Figure 2-8: DSP Engine Block Diagram



2.6.1 Data Accumulators

There are two 40-bit data accumulators, ACCA and ACCB, that are the result registers for the DSP instructions listed in Table 2-3. Each accumulator is memory mapped to three registers, where 'x' denotes the particular accumulator:

- ACCxL: ACCx<15:0>
- ACCxH: ACCx<31:16>
- ACCxU: ACCx<39:32>

For fractional operations that use the accumulators, the radix point is located to the right of bit 31. The range of fractional values that be stored in each accumulator is -256.0 to ($256.0 - 2^{-31}$). For integer operations that use the accumulators, the radix point is located to the right of bit 0. The range of integer values that can be stored in each accumulator is -549,755,813,888 to 549,755,813,887.

2.6.2 Multiplier

The dsPIC30F features a 17-bit x 17-bit multiplier which is shared by both the MCU ALU and the DSP engine. The multiplier is capable of signed or unsigned operation and can support either 1.31 fractional (Q.31) or 32-bit integer results.

The multiplier takes in 16-bit input data and converts the data to 17-bits. Signed operands to the multiplier are sign-extended. Unsigned input operands are zero-extended. The 17-bit conversion logic is transparent to the user and allows the multiplier to support mixed sign and unsigned/ unsigned multiplication.

The IF control bit (CORCON<0>) determines integer/fractional operation for the instructions listed in Table 2-3. The IF bit does not affect MCU multiply instructions listed in Table 2-4, which are always integer operations. The multiplier scales the result one bit to the left for fractional operation. The Least Significant bit (LSb) of the result is always cleared. The multiplier defaults to Fractional mode for DSP operations at a device Reset.

The representation of data in hardware for each of these modes is as follows:

- Integer data is inherently represented as a signed two's complement value, where the MSbit is defined as a sign bit. Generally speaking, the range of an N-bit two's complement integer is -2^{N-1} to $2^{N-1} - 1$.
- Fractional data is represented as a two's complement fraction where the MSbit is defined as a sign bit and the radix point is implied to lie just after the sign bit (Q.X format). The range of an N-bit two's complement fraction with this implied radix point is -1.0 to ($1 - 2^{1-N}$).

Figure 2-9 and Figure 2-10 illustrate how the multiplier hardware interprets data in Integer and Fractional modes. The range of data in both Integer and Fractional modes is listed in Table 2-2.

Figure 2-9: Integer and Fractional Representation of 0x4001

Different Representations of 0x4001

Integer:

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

-2^{15} 2^{14} 2^{13} 2^{12} . . . . 2^{0}

0x4001 = $2^{14} + 2^{0} = 16385$

1.15 Fractional:

0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

-2^{0} ▪ 2^{-1} 2^{-2} 2^{-3} . . . 2^{-15}

Implied Radix Point

0x4001 = $2^{-1} + 2^{-15} = 0.500030518$

Figure 2-10: Integer and Fractional Representation of 0xC002

Different Representations of 0xC002

Integer:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

-2^{15} 2^{14} 2^{13} 2^{12} . . . . 2^{0}

0xC002 = $-2^{15} + 2^{14} + 2^{0} = -32768 + 16384 + 2 = -16382$

1.15 Fractional:

1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

-2^{0} ▪ 2^{-1} 2^{-2} 2^{-3} . . . 2^{-15}

Implied Radix Point

0xC002 = $-2^{0} + 2^{-1} + 2^{-14} = -1 + 0.5 + 0.000061035 = -0.499938965$

Table 2-2: dsPIC30F Data Ranges

Register Size	Integer Range	Fraction Range	Fraction Resolution
16-bit	-32768 to 32767	-1.0 to $(1.0 - 2^{-15})$ (Q.15 Format)	3.052×10^{-5}
32-bit	-2,147,483,648 to 2,147,483,647	-1.0 to $(1.0 - 2^{-31})$ (Q.31 Format)	4.657×10^{-10}
40-bit	-549,755,813,888 to 549,755,813,887	-256.0 to $(256.0 - 2^{-31})$ (Q.31 Format with 8 Guard bits)	4.657×10^{-10}

2.6.2.1 DSP Multiply Instructions

The DSP instructions that utilize the multiplier are summarized in Table 2-3.

Table 2-3: DSP Instructions that Utilize the Multiplier

DSP Instruction	Description	Algebraic Equivalent
MAC	Multiply and Add to Accumulator OR Square and Add to Accumulator	$a = a + b*c$ $a = a + b^2$
MSC	Multiply and Subtract from Accumulator	$a = a - b*c$
MPY	Multiply	$a = b*c$
MPY.N	Multiply and Negate Result	$a = -b*c$
ED	Partial Euclidean Distance	$a = (b - c)^2$
EDAC	Add Partial Euclidean Distance to the Accumulator	$a = a + (b - c)^2$

Note: DSP instructions using the multiplier can operate in Fractional (1.15) or Integer modes.

The US control bit (CORCON<12>) determines whether DSP multiply instructions are signed (default) or unsigned. The US bit does not influence the MCU multiply instructions which have specific instructions for signed or unsigned operation. If the US bit is set, the input operands for instructions shown in Table 2-3 are considered as unsigned values which are always zero-extended into the 17th bit of the multiplier value.

2.6.2.2 MCU Multiply Instructions

The same multiplier is used to support the MCU multiply instructions, which include integer 16-bit signed, unsigned and mixed sign multiplies as shown in Table 2-4. All multiplications performed by the MUL instruction produce integer results. The MUL instruction may be directed to use byte or word sized operands. Byte input operands will produce a 16-bit result and word input operands will produce a 32-bit result to the specified register(s) in the W array.

Table 2-4: MCU Instructions that Utilize the Multiplier

MCU Instruction	Description
MUL/MUL.UU	Multiply two unsigned integers
MUL.SS	Multiply two signed integers
MUL.SU/MUL.US	Multiply a signed integer with an unsigned integer

Note 1: MCU instructions using the multiplier operate only in Integer mode.
2: Result of an MCU multiply is 32-bits long and is stored in a pair of W registers.

2.6.3 Data Accumulator Adder/Subtractor

The data accumulators have a 40-bit adder/subtractor with automatic sign extension logic for the multiplier result (if signed). It can select one of the two accumulators (A or B) as its pre-accumulation source and post-accumulation destination. For the ADD (accumulator) and LAC instructions, the data to be accumulated or loaded can optionally be scaled via the barrel shifter prior to accumulation.

The 40-bit adder/subtractor may optionally negate one of its operand inputs to change the sign of the result (without changing the operands). The negate is used during multiply and subtract (MSC), or multiply and negate (MPY.N) operations.

The 40-bit adder/subtractor has an additional saturation block which controls accumulator data saturation, if enabled.

2.6.3.1 Accumulator Status Bits

Six Status register bits have been provided to support saturation and overflow. They are located in the CPU Status register, SR and are listed below:

Table 2-5: Accumulator Overflow and Saturation Status Bits

Status Bit	Location	Description
OA	SR<15>	Accumulator A overflowed into guard bits (ACCA<39:32>)
OB	SR<14>	Accumulator B overflowed into guard bits(ACCB<39:32>)
SA	SR<13>	ACCA saturated (bit 31 overflow and saturation) *or* ACCA overflowed into guard bits and saturated (bit 39 overflow and saturation)
SB	SR<12>	ACCB saturated (bit 31 overflow and saturation) *or* ACCB overflowed into guard bits and saturated (bit 39 overflow and saturation)
OAB	SR<11>	OA logically ORed with OB
SAB	SR<10>	SA logically ORed with SB. Clearing SAB will also clear SA and SB.

The OA and OB bits are read only and are modified each time data passes through the accumulator add/subtract logic. When set, they indicate that the most recent operation has overflowed into the accumulator guard bits (bits 32 through 39). This type of overflow is not catastrophic; the guard bits preserve the accumulator data. The OAB status bit is the logically ORed value of OA and OB.

The OA and OB bits, when set, can optionally generate an arithmetic error trap. The trap is enabled by setting the corresponding overflow trap flag enable bit OVATE:OVBTE (INTCON1<10:9>). The trap event allows the user to take immediate corrective action, if desired.

The SA and SB bits can be set each time data passes through the accumulator saturation logic. Once set, these bits remain set until cleared by the user. The SAB status bit indicates the logically ORed value of SA and SB. The SA and SB bits will be cleared when SAB is cleared. When set, these bits indicate that the accumulator has overflowed its maximum range (bit 31 for 32-bit saturation or bit 39 for 40-bit saturation) and will be saturated (if saturation is enabled).

When saturation is not enabled, the SA and SB bits indicate that a catastrophic overflow has occurred (the sign of the accumulator has been destroyed). If the COVTE (INTCON1<8>) bit is set, SA and SB bits will generate an arithmetic error trap when saturation is disabled.

Note: See **Section 6. "Reset Interrupts"** for further information on arithmetic warning traps.

Note: The user must remember that SA, SB and SAB status bits can have different meanings depending on whether accumulator saturation is enabled. The Accumulator Saturation mode is controlled via the CORCON register.

2.6.3.2 Saturation and Overflow Modes

The device supports three Saturation and Overflow modes.

Accumulator 39-bit Saturation
In this mode, the saturation logic loads the maximally positive 9.31 value (0x7FFFFFFFFF), or maximally negative 9.31 value (0x8000000000), into the target accumulator. The SA or SB bit is set and remains set until cleared by the user. This Saturation mode is useful for extending the dynamic range of the accumulator.

To configure for this mode of saturation, the ACCSAT(CORCON<4>) bit must be set. Additionally, the SATA and/or SATB (CORCON<7 and/or 6>) bits must be set to enable accumulator saturation.

- **Accumulator 31-bit Saturation**
  In this mode, the saturation logic loads the maximally positive 1.31 value (0x007FFFFFFF) or maximally negative 1.31 value (0xFF80000000) into the target accumulator. The SA or SB bit is set and remains set until cleared by the user. When this Saturation mode is in effect, the guard bits 32 through 39 are not used, except for sign-extension of the accumulator value. Consequently, the OA, OB or OAB bits in SR will never be set.

  To configure for this mode of overflow and saturation, the ACCSAT (CORCON<4>) bit must be cleared. Additionally, the SATA and/or SATB (CORCON<7 and/or 6>) bits must be set to enable accumulator saturation.

- **Accumulator Catastrophic Overflow**
  If the SATA and/or SATB (CORCON<7 and/or 6>) bits are not set, then no saturation operation is performed on the accumulator and the accumulator is allowed to overflow all the way up to bit 39 (destroying its sign). If the COVTE bit (INTCON1<8>) is set, a catastrophic overflow will initiate an arithmetic error trap.

Note that accumulator saturation and overflow detection can only result from the execution of a DSP instruction that modifies one of the two accumulators via the 40-bit DSP ALU. Saturation and overflow detection will not take place when the accumulators are accessed as memory mapped registers via MCU class instructions. Furthermore, the accumulator status bits shown in Table 2-5 will not be modified. However, the MCU status bits (Z, N, C, OV, DC) will be modified depending on the MCU instruction that accesses the accumulator.

Note: See **Section 6. "Reset Interrupts"** for further information on arithmetic error traps.

2.6.3.3 Data Space Write Saturation

In addition to adder/subtractor saturation, writes to data space can be saturated without affecting the contents of the source accumulator. This feature allows data to be limited while not sacrificing the dynamic range of the accumulator during intermediate calculation stages. Data space write saturation is enabled by setting the SATDW control bit (CORCON<5>). Data space write saturation is enabled by default at a device Reset.

The data space write saturation feature works with the `SAC` and `SAC.R` instructions. The value held in the accumulator is never modified when these instructions are executed. The hardware takes the following steps to obtain the saturated write result:

1. The read data is scaled based upon the arithmetic shift value specified in the instruction.
2. The scaled data is rounded (`SAC.R` only).
3. The scaled/rounded value is saturated to a 16-bit result based on the value of the guard bits. For data values greater than `0x007FFF`, the data written to memory is saturated to the maximum positive 1.15 value, `0x7FFF`. For input data less than `0xFF8000`, data written to memory is saturated to the maximum negative 1.15 value, `0x8000`.

2.6.3.4 Accumulator 'Write Back'

The MAC and MSC instructions can optionally write a rounded version of the accumulator that is not the target of the current operation into data space memory. The write is performed across the X-bus into combined X and Y address space. This accumulator write back feature is beneficial in certain FFT and LMS algorithms.

The following Addressing modes are supported by the accumulator write back hardware:

- W13, register direct:
  The rounded contents of the non-target accumulator are written into W13 as a 1.15 fractional result.
- [W13]+=2, register indirect with post-increment:
  The rounded contents of the non-target accumulator are written into the address pointed to by W13 as a 1.15 fraction. W13 is then incremented by 2.

2.6.4 Round Logic

The round logic can perform a conventional (biased) or convergent (unbiased) round function during an accumulator write (store). The Round mode is determined by the state of the RND (CORCON<1>) bit. It generates a 16-bit, 1.15 data value, which is passed to the data space write saturation logic. If rounding is not indicated by the instruction, a truncated 1.15 data value is stored.

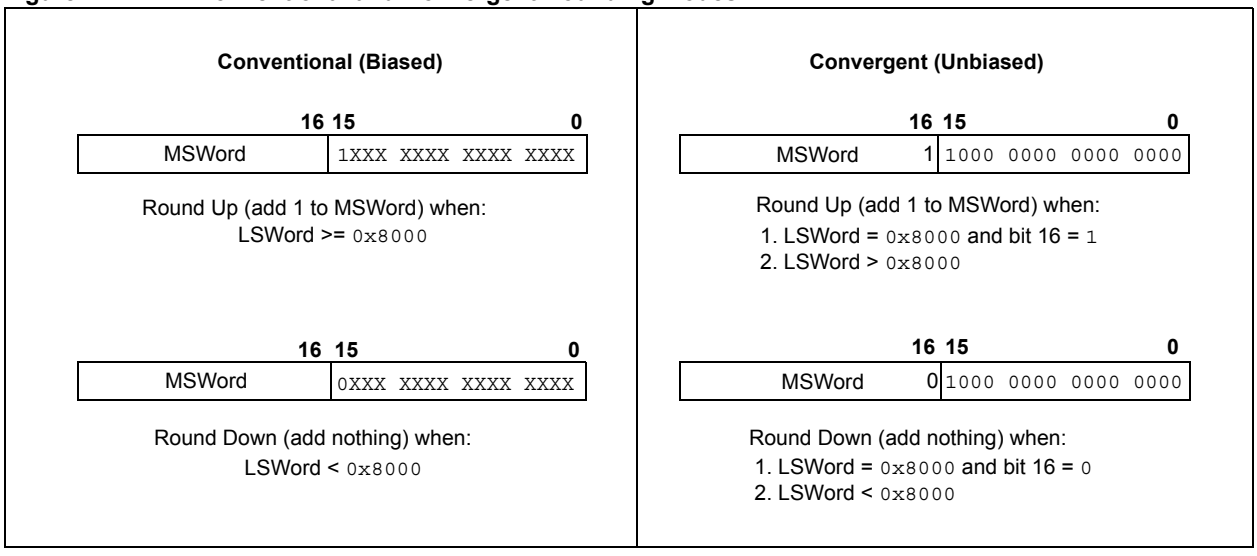
The two Rounding modes are shown in Figure 2-11. Conventional rounding takes bit 15 of the accumulator, zero-extends it and adds it to the most significant Word excluding the guard or overflow bits (bits 16 through 31). If the least significant word of the accumulator is between 0x8000 and 0xFFFF (0x8000 included), the MSWord is incremented. If the LSWord of the accumulator is between 0x0000 and 0x7FFF, the MSWord is left unchanged. A consequence of this algorithm is that over a succession of random rounding operations, the value will tend to be biased slightly positive.

Convergent (or unbiased) rounding operates in the same manner as conventional rounding except when the least significant word equals 0x8000. If this is the case, the LSbit of the most significant word (bit 16 of the accumulator) is examined. If it is '1', the most significant word is incremented. If it is '0', the most significant word is not modified. Assuming that bit 16 is effectively random in nature, this scheme will remove any rounding bias that may accumulate.

The SAC and SAC.R instructions store either a truncated (SAC) or rounded (SAC.R) version of the contents of the target accumulator to data memory via the X-bus (subject to data saturation, see **Section 2.6.3.3 "Data Space Write Saturation"**).

Note that for the MAC class of instructions, the accumulator write back data path is always subject to rounding.

Figure 2-11: Conventional and Convergent Rounding Modes



2.6.5 Barrel Shifter

The barrel shifter is capable of performing up to a 16-bit arithmetic right shift, or up to a 16-bit left shift, in a single cycle. The barrel shifter can be used by DSP instructions or MCU instructions for multi-bit shifts.

The shifter requires a signed binary value to determine both the magnitude (number of bits) and direction of the shift operation:

- A positive value will shift the operand right
- A negative value will shift the operand left
- A value of '0' will not modify the operand

The barrel shifter is 40-bits wide to accommodate the width of the accumulators. A 40-bit output result is provided for DSP shift operations, and a 16-bit result for MCU shift operations.

A summary of instructions that use the barrel shifter is provided below in Table 2-6.

Table 2-6: Instructions that Utilize the DSP Engine Barrel Shifter

Instruction	Description
ASR	Arithmetic multi-bit right shift of data memory location
LSR	Logical multi-bit right shift of data memory location
SL	Multi-bit shift left of data memory location
SAC	Store DSP accumulator with optional shift
SFTAC	Shift DSP accumulator

2.6.6 DSP Engine Mode Selection

The various operational characteristics of the DSP engine discussed in previous sub-sections can be selected through the CPU Core Configuration register (CORCON). These are listed below:

- Fractional or integer multiply operation
- Conventional or convergent rounding
- Automatic saturation on/off for ACCA
- Automatic saturation on/off for ACCB
- Automatic saturation on/off for writes to data memory
- Accumulator Saturation mode selection

2.6.7 DSP Engine Trap Events

The various arithmetic error traps that can be generated for handling exceptions in the DSP engine are selected through the Interrupt Control register (INTCON1). These are listed below:

- Trap on ACCA overflow enable, using OVATE (INTCON1<10>)
- Trap on ACCB overflow enable, using OVBTE (INTCON1<9>)
- Trap on catastrophic ACCA and/or ACCB overflow enable, using COVTE (INTCON1<8>).

An arithmetic error trap will also be generated when the user attempts to shift a value beyond the maximum allowable range (+/- 16 bits) using the SFTAC instruction. This trap source cannot be disabled. The execution of the instruction will complete, but the results of the shift will not be written to the target accumulator.

For further information on bits in the INTCON1 register and arithmetic error traps, please refer to **Section 6. "Reset Interrupts"**.

2.7 Divide Support

The dsPIC30F supports the following types of division operations:

- DIVF: 16/16 signed fractional divide
- DIV.SD: 32/16 signed divide
- DIV.UD: 32/16 unsigned divide
- DIV.SW: 16/16 signed divide
- DIV.UW: 16/16 unsigned divide

The quotient for all divide instructions is placed in W0, and the remainder in W1. The 16-bit divisor can be located in any W register. A 16-bit dividend can be located in any W register and a 32-bit dividend must be located in an adjacent pair of W registers.

All divide instructions are iterative operations and must be executed 18 times within a REPEAT loop. The user is responsible for programming the REPEAT instruction. A complete divide operation takes 19 instruction cycles to execute.

The divide flow is interruptible, just like any other REPEAT loop. All data is restored into the respective data registers after each iteration of the loop, so the user will be responsible for saving the appropriate W registers in the ISR. Although they are important to the divide hardware, the intermediate values in the W registers have no meaning to the user. The divide instructions must be executed 18 times in a REPEAT loop to produce a meaningful result.

Refer to the "*dsPIC30F Programmer's Reference Manual*" (DS70157) for more information and programming examples for the divide instructions.

2.8 Instruction Flow Types

Most instructions in the dsPIC30F architecture occupy a single word of program memory and execute in a single cycle. An instruction prefetch mechanism facilitates single cycle (1 TCY) execution. However, some instructions take 2 or 3 instruction cycles to execute. Consequently, there are seven different types of instruction flow in the dsPIC® DSC architecture. These are described below:

1. **1 Instruction Word, 1 Instruction Cycle:**

   These instructions will take one instruction cycle to execute as shown in Figure 2-12. Most instructions are 1-word, 1-cycle instructions.

Figure 2-12: Instruction Flow – 1-Word, 1-Cycle

	TCY0	TCY1	TCY2	TCY3	TCY4	TCY5
1. MOV #0x55AA,W0	Fetch 1	Execute 1				
2. MOV W0,PORTA		Fetch 2	Execute 2			
3. MOV W0,PORTB			Fetch 3	Execute 3		

2. **1 Instruction Word, 2 Instruction Cycles:**

   In these instructions, there is no prefetch flush. The only instructions of this type are the MOV.D instructions (load and store double-word). Two cycles are required to complete these instructions, as shown in Figure 2-13.

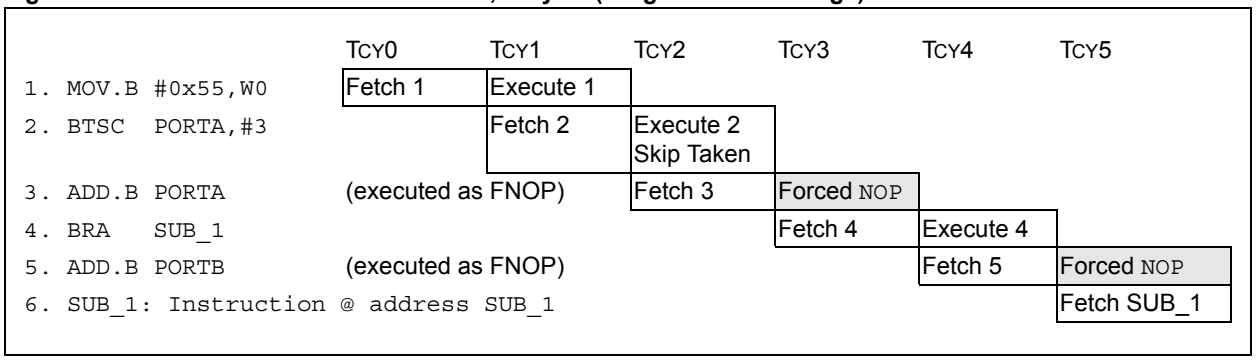
Figure 2-13: Instruction Flow – 1-Word, 2-Cycle (MOV.D Operation)

	TCY0	TCY1	TCY2	TCY3	TCY4	TCY5
1. MOV #0x1234,W0	Fetch 1	Execute 1				
2. MOV.D [W0++],W1		Fetch 2	Execute 2 R/W Cycle 1			
3. MOV #0x00AA,W1			Fetch 3	Execute 2 R/W Cycle2		
				No Fetch	Execute 3	
4. MOV #0x00CC,W0					Fetch 4	Execute 4

3. 1 Instruction Word, 2 or 3 Instruction Cycle Program Flow Changes:

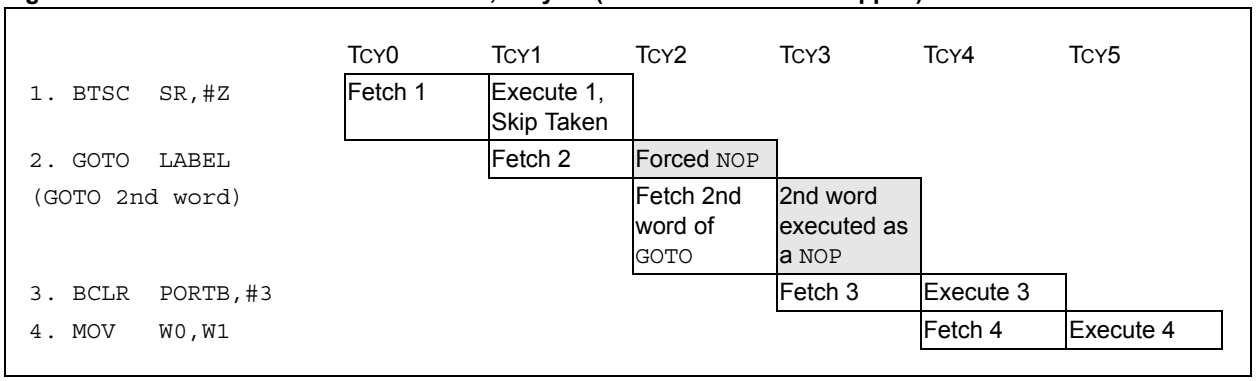
These instructions include relative call and branch instructions, and skip instructions. When an instruction changes the PC (other than to increment it), the program memory prefetch data must be discarded. This makes the instruction take two effective cycles to execute, as shown in Figure 2-14.

Figure 2-14: Instruction Flow – 1-Word, 2-Cycle (Program Flow Change)



Three cycles will be taken when a two-word instruction is skipped. In this case, the program memory prefetch data is discarded and the second word of the two-word instruction is fetched. The second word of the instruction will be executed as a NOP, as shown in Figure 2-15.

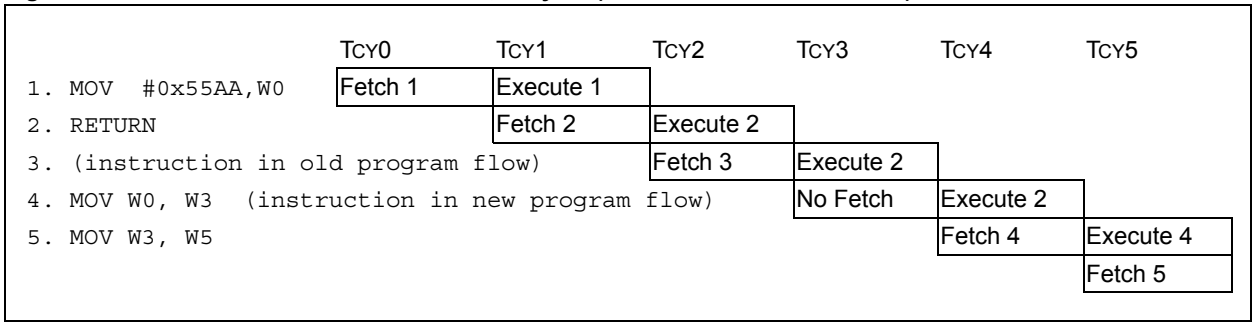
Figure 2-15: Instruction Flow – 1-Word, 3-Cycle (2-Word Instruction Skipped)



4. 1 Instruction Word, 3 Instruction Cycles (RETFIE, RETURN, RETLW):

The RETFIE, RETURN and RETLW instructions, that are used to return from a subroutine call or an Interrupt Service Routine, take 3 instruction cycles to execute, as shown in Figure 2-16.

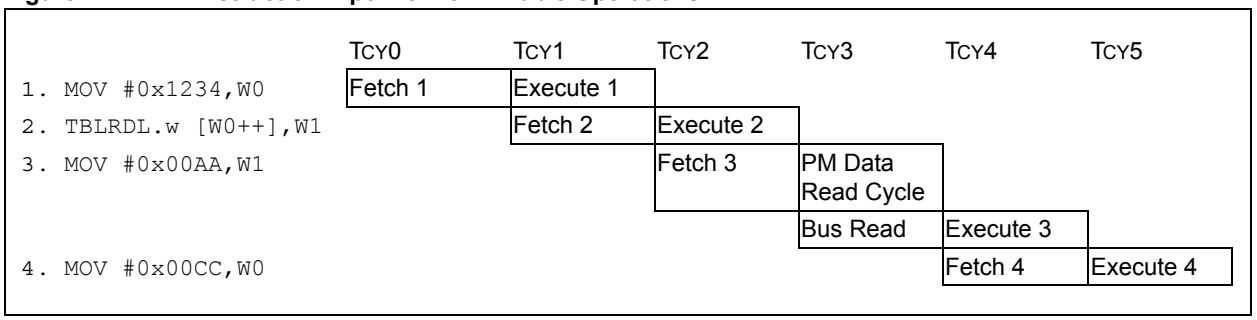
Figure 2-16: Instruction Flow – 1-Word, 3-Cycle (RETURN, RETFIE, RETLW)



5. Table Read/Write Instructions:

These instructions will suspend fetching to insert a read or write cycle to the program memory. The instruction fetched while executing the table operation is saved for 1 cycle and executed in the cycle immediately after the table operation as shown in Figure 2-17.

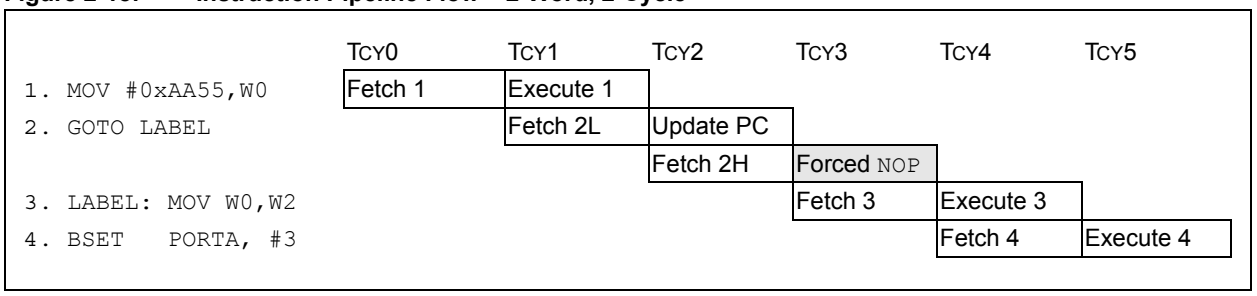
Figure 2-17: Instruction Pipeline Flow – Table Operations



6. 2 Instruction Words, 2 Instruction Cycles:

In these instructions, the fetch after the instruction contains data. This results in a 2-cycle instruction as shown in Figure 2-18. The second word of a two-word instruction is encoded so that it will be executed as a NOP, should it be fetched by the CPU without first fetching the first word of the instruction. This is important when a two-word instruction is skipped by a skip instruction (see Figure 2-15).

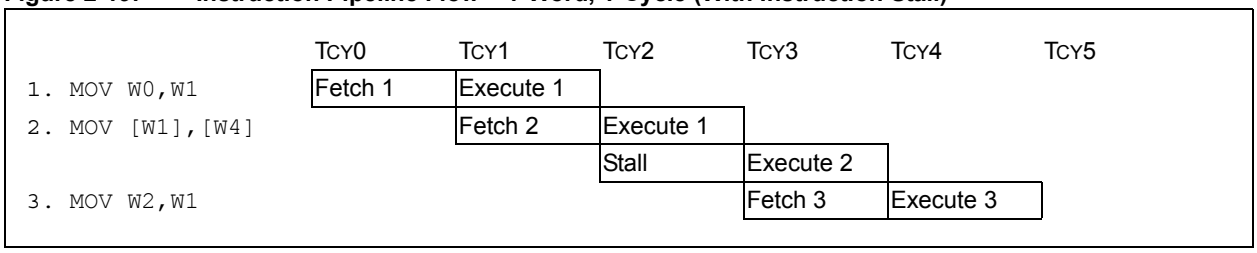
Figure 2-18: Instruction Pipeline Flow – 2-Word, 2-Cycle



7. Address Register Dependencies:

These are the instructions that are subjected to a stall due to a data address dependency between the X-data space read and write operations. An additional cycle is inserted to resolve the resource conflict as discussed in **Section 2.10 "Address Register Dependencies"**.

Figure 2-19: Instruction Pipeline Flow – 1-Word, 1-Cycle (With Instruction Stall)



2.9 Loop Constructs

The dsPIC30F supports both REPEAT and DO instruction constructs to provide unconditional automatic program loop control. The REPEAT instruction is used to implement a single instruction program loop. The DO instruction is used to implement a multiple instruction program loop. Both instructions use control bits within the CPU Status register, SR, to temporarily modify CPU operation.

2.9.1 Repeat Loop Construct

The REPEAT instruction causes the instruction that follows it to be repeated a number of times. A literal value contained in the instruction or a value in one of the W registers can be used to specify the repeat count value. The W register option enables the loop count to be a software variable.

An instruction in a REPEAT loop will be executed at least once. The number of iterations for a repeat loop will be the 14-bit literal value + 1 or Wn + 1.

The syntax for the two forms of the REPEAT instruction is given below:

```
REPEAT #lit14     ; RCOUNT <-- lit14
(Valid target Instruction)
or
REPEAT Wn         ; RCOUNT <-- Wn
(Valid target Instruction)
```

2.9.1.1 Repeat Operation

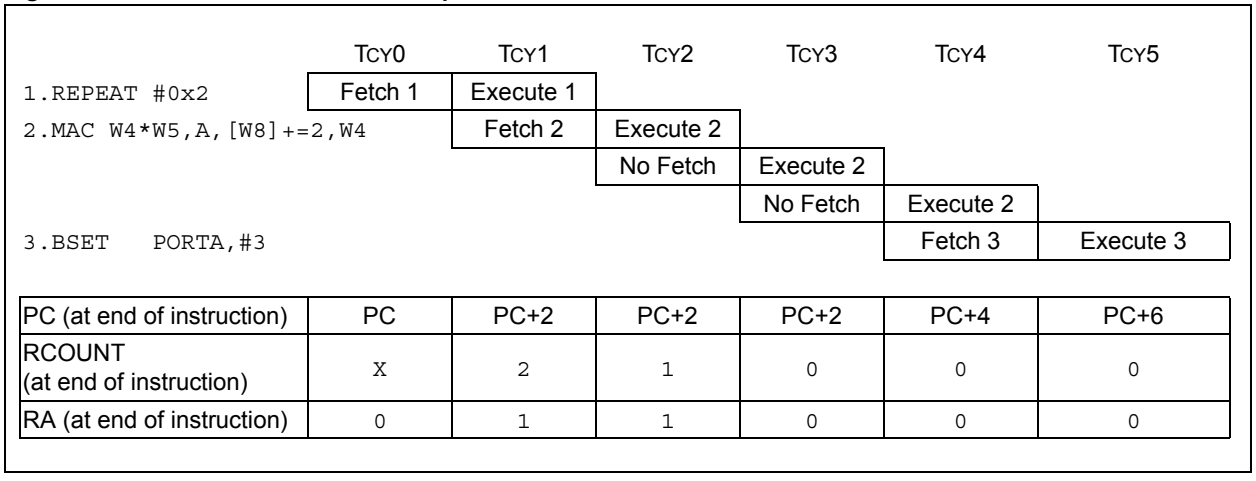
The loop count for Repeat operations is held in the 14-bit RCOUNT register, which is memory mapped. RCOUNT is initialized by the REPEAT instruction. The REPEAT instruction sets the Repeat Active, or RA (SR<4>) status bit to '1', if the RCOUNT is a non-zero value.

RA is a read only bit and cannot be modified through software. For repeat loop count values greater than '0', the PC is not incremented. Further PC increments are inhibited until RCOUNT = 0. See Figure 2-20 for an instruction flow example of a Repeat loop.

For a loop count value equal to '0', REPEAT has the effect of a NOP and the RA (SR<4>) bit is not set. The Repeat loop is essentially disabled before it begins, allowing the target instruction to execute only once while prefetching the subsequent instruction (i.e., normal execution flow).

Note:	The instruction immediately following the REPEAT instruction (i.e., the target instruction) is always executed at least one time. It is always executed one time more than the value specified in the 14-bit literal or the W register operand.

Figure 2-20: REPEAT Instruction Pipeline Flow



PC (at end of instruction)	PC	PC+2	PC+2	PC+2	PC+4	PC+6
RCOUNT (at end of instruction)	X	2	1	0	0	0
RA (at end of instruction)	0	1	1	0	0	0

2.9.1.2 Interrupting a REPEAT Loop

A REPEAT instruction loop may be interrupted at any time.

The RA state is preserved on the stack during exception processing to allow the user to execute further REPEAT loops from within (any number) of nested interrupts. After SRL is stacked, the RA status bit is cleared to restore normal execution flow within the ISR.

Note: If a Repeat loop has been interrupted and an ISR is being processed, the user must stack the RCOUNT (Repeat Count register) prior to executing another REPEAT instruction within an ISR.

Note: If Repeat was used within an ISR, the user must unstack RCOUNT prior to executing RETFIE.

Returning into a Repeat loop from an ISR using RETFIE requires no special handling. Interrupts will prefetch the repeated instruction during the third cycle of the RETFIE. The stacked RA bit will be restored when the SRL register is popped and, if set, the interrupted Repeat loop will be resumed.

Note: Should the repeated instruction (target instruction in the Repeat loop) be accessing data from PS using PSV, the first time it is executed after a return from an exception will require 2 instruction cycles. Similar to the first iteration of a loop, timing limitations will not allow the first instruction to access data residing in PS in a single instruction cycle.

2.9.1.2.1 Early Termination of a REPEAT Loop

An interrupted Repeat loop can be terminated earlier than normal in the ISR by clearing the RCOUNT register in software.

2.9.1.3 Restrictions on the REPEAT Instruction

Any instruction can immediately follow a REPEAT except for the following:

1. Program Flow Control instructions (any branch, compare and skip, subroutine calls, returns, etc.).
2. Another REPEAT or DO instruction.
3. DISI, ULNK, LNK, PWRSAV and RESET.
4. MOV.D instruction.

Note: There are some instructions and/or Instruction Addressing modes that can be executed within a Repeat loop, but make little sense when repeated.

2.9.2 DO Loop Construct

The DO instruction can execute a group of instructions that follow it a specified number of times without software overhead. The set of instructions up to and including the end address will be repeated. The repeat count value for the DO instruction can be specified by a 14-bit literal or by the contents of a W register declared within the instruction. The syntax for the two forms of the DO instruction is given below:

```
          DO     #lit14,LOOP_END     ; DCOUNT <-- lit14
          Instruction1
          Instruction2
          :
          :
LOOP_END: Instruction n

          DO     Wn,LOOP_END         ; DCOUNT <-- Wn<13:0>
          Instruction1
          Instruction2
          :
          :
LOOP_END: Instruction n
```

The following features are provided in the DO loop construct:

- A W register can be used to specify the loop count. This allows the loop count to be defined at run-time.
- The instruction execution order need not be sequential (i.e., there can be branches, subroutine calls, etc.).
- The loop end address does not have to be greater than the start address.

2.9.2.1 DO Loop Registers and Operation

The number of iterations executed by a DO loop will be the (14-bit literal value +1) or the (Wn value + 1). If a W register is used to specify the number of iterations, the two MSbits of the W register are not used to specify the loop count. The operation of a DO loop is similar to the 'do-while' construct in the C programming language because the instructions in the loop will always be executed at least once.

The dsPIC30F has three registers associated with DO loops: DOSTART, DOEND and DCOUNT. These registers are memory mapped and automatically loaded by the hardware when the DO instruction is executed. DOSTART holds the starting address of the DO loop while DOEND holds the end address of the DO loop. The DCOUNT register holds the number of iterations to be executed by the loop. DOSTART and DOEND are 22-bit registers that hold the PC value. The MSbits and LSbits of these registers is fixed to '0'. Refer to Figure 2-2 for further details. The LSbit is not stored in these registers because PC<0> is always forced to '0'.

The DA status bit (SR<9>) indicates that a single DO loop (or nested DO loops) is active. The DA bit is set when a DO instruction is executed and enables a PC address comparison with the DOEND register on each subsequent instruction cycle. When PC matches the value in DOEND, DCOUNT is decremented. If the DCOUNT register is not zero, the PC is loaded with the address contained in the DOSTART register to start another iteration of the DO loop.

The DO loop will terminate when DCOUNT = 0. If there are no other nested DO loops in progress, then the DA bit will also be cleared.

Note:	The group of instructions in a DO loop construct is always executed at least one time. The DO loop is always executed one time more than the value specified in the literal or W register operand.

2.9.2.2 DO Loop Nesting

The DOSTART, DOEND and DCOUNT registers each have a shadow register associated with them, such that the DO loop hardware supports one level of automatic nesting. The DOSTART, DOEND and DCOUNT registers are user accessible and they may be manually saved to permit additional nesting, where required.

The DO Level bits, DL<2:0> (CORCON<10:8>) indicate the nesting level of the DO loop currently being executed. When the first `DO` instruction is executed, DL<2:0> is set to `B'001'` to indicate that one level of DO loop is underway. The DA (SR<9>) is also set. When another `DO` instruction is executed within the first DO loop, the DOSTART, DOEND and DCOUNT registers are transferred into the shadow registers, prior to being updated with the new loop values. The DL<2:0> bits are set to `B'010'` indicating that a second, nested DO loop is in progress. The DA (SR<9>) bit also remains set.

If no more than one level of DO loop nesting is required in the application, no special attention is required. Should the user require more than one level of DO loop nesting, this may be achieved through manually saving the DOSTART, DOEND and DCOUNT registers prior to executing the next `DO` instruction. These registers should be saved whenever DL<2:0> is `B'010'` or greater.

The DOSTART, DOEND and DCOUNT registers will automatically be restored from their shadow registers when a DO loop terminates and DL<2:0> = `B'010'`.

Note:	The DL<2:0> (CORCON<10:8>) bits are combined (logically ORed) to form the DA (SR<9>) bit. If nested DO loops are being executed, the DA bit is cleared only when the loop count associated with the outer most loop expires.

2.9.2.3 Interrupting a DO Loop

DO loops may be interrupted at any time. If another DO loop is to be executed during the ISR, the user must check the DL<2:0> status bits and save the DOSTART, DOEND and DCOUNT registers as required.

No special handling is required if the user can ensure that only one level of DO loop will ever be executed in:

- both background and any one ISR handler (if interrupt nesting is enabled) or
- both background and any ISR (if interrupt nesting is disabled)

Alternatively, up to two (nested) DO loops may be executed in either background or within any

- one ISR handler (if interrupt nesting is enabled) or
- in any ISR (if interrupt nesting is disabled)

It is assumed that no DO loops are used within any trap handlers.

Returning to a DO loop from an ISR, using the `RETFIE` instruction, requires no special handling.

2.9.2.4 Early Termination of the DO loop

There are two ways to terminate a DO loop, earlier than normal:

1. The EDT (CORCON<11>) bit provides a means for the user to terminate a DO loop before it completes all loops. Writing a '1' to the EDT bit will force the loop to complete the iteration underway and then terminate. If EDT is set during the penultimate or last instruction of the loop, one more iteration of the loop will occur. EDT will always read as a '0'; clearing it has no effect. After the EDT bit is set, the user can optionally branch out of the DO loop.
2. Alternatively, the code may branch out of the loop at any point except from the last instruction, which cannot be a flow control instruction. Although the DA bit enables the DO loop hardware, it will have no effect unless the address of the penultimate instruction is encountered during an instruction prefetch. This is not a recommended method for terminating a DO loop.

Note:	Exiting a DO loop without using EDT is not recommended because the hardware will continue to check for DOEND addresses.

2.9.2.5 DO Loop Restrictions

DO loops have the following restrictions imposed:

- choice of last instruction in the loop
- the loop length (offset from the first instruction)
- reading of the DOEND register

All DO loops must contain at least two instructions because the loop termination tests are performed in the penultimate instruction. REPEAT should be used for single instruction loops.

The special function register, DOEND, cannot be read by user software in the instruction that immediately follows either a DO instruction, or a file register write operation to the DOEND SFR.

The instruction that is executed two instructions before the last instruction in a DO loop should not modify any of the following:

- CPU priority level governed by the IPL (SR<7:5>) bits
- Peripheral Interrupt Enable bits governed by the IEC0, IEC1 and IEC2 registers
- Peripheral Interrupt Priority bits governed by the IPC0 through IPC11 registers

If the restrictions above are not followed, the DO loop may execute incorrectly.

2.9.2.5.1 Last Instruction Restrictions

There are restrictions on the last instruction executed in a DO loop. The last instruction in a DO loop should not be:

1. Flow control instruction (for e.g., any branch, compare and skip, GOTO, CALL, RCALL, TRAP).
2. RETURN, RETFIE and RETLW will work correctly as the last instruction of a DO loop, but the user must be responsible for returning into the loop to complete it.
3. Another REPEAT or DO instruction.
4. Target instruction within a REPEAT loop. This restriction implies that the penultimate instruction also cannot be a REPEAT.
5. Any instruction that occupies two words in program space.
6. DISI instruction

2.9.2.5.2 Loop Length Restrictions

Loop length is defined as the signed offset of the last instruction from the first instruction in the DO loop. The loop length when added to the address of the first instruction in the loop forms the address of the last instruction of the loop.There are some loop length values that should be avoided.

1. **Loop Length = -2**

   Execution will start at the first instruction in the loop (i.e., at [PC]) and will continue until the loop end address (in this case [PC – 4]) is prefetched. As this is the first word of the DO instruction, it will execute the DO instruction again, re-initializing the DCOUNT and prefetching [PC]. This will continue forever as long as the loop end address [PC – 4] is prefetched. This value of n has the potential of creating an infinite loop (subject to a Watchdog Timer Reset).

```
end_loop: DO #33, end_loop ;DO is a two-word instruction
          NOP              ;2nd word of DO executes as a NOP
          ADD W2,W3,W4     ;First instruction in DO loop([PC])
```

2. Loop Length = -1

Execution will start at the first instruction in the loop (i.e., at [PC]) and will continue until the loop end address ([PC – 2]) is prefetched. Since the loop end address is the second word of the DO instruction, it will execute as a NOP but will still prefetch [PC]. The loop will then execute again. This will continue as long as the loop end address [PC – 2] is prefetched and the loop does not terminate. Should the value in the DCOUNT register reach zero and on a subsequent decrement generate a borrow, the loop will terminate. However, in such a case the initial instruction outside the loop will once again be the first loop instruction.

```
          DO #33, end_loop ;DO is a two-word instruction
end_loop: NOP              ;2nd word of DO executes as a NOP
          ADD W2,W3,W4     ;First instruction in DO loop([PC])
```

3. Loop Length = 0

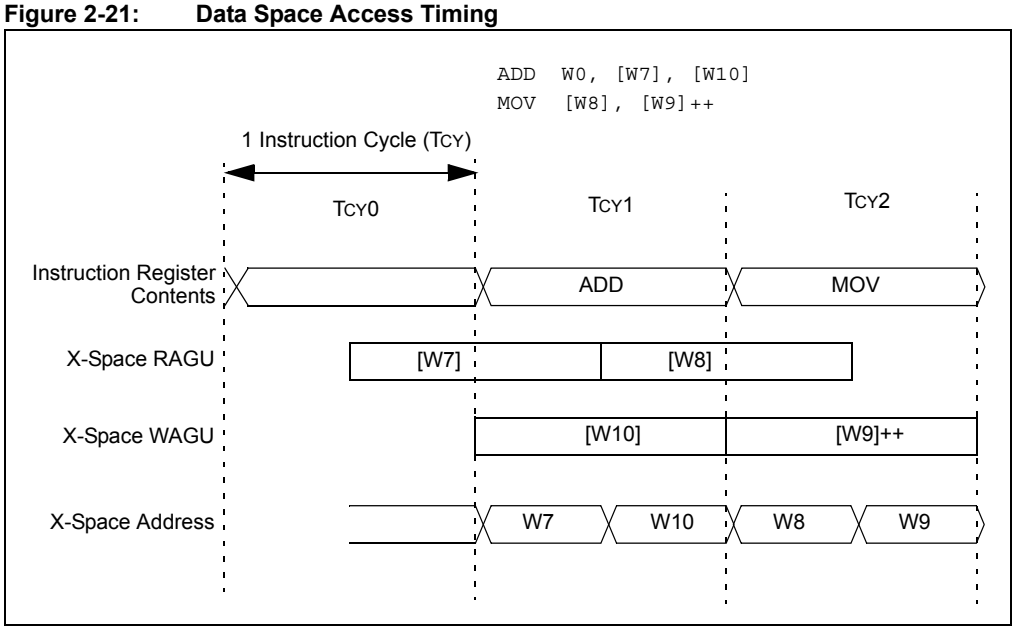
Execution will start at the first instruction in the loop (i.e., at [PC]) and will continue until the loop end address ([PC]) is prefetched. If the loop is to continue, this prefetch will cause the DO loop hardware to load the DOEND address ([PC]) into the PC for the next fetch (which will be [PC] again). After the first true iteration of the loop, the first instruction in the loop will be executed repeatedly until the loop count underflows and the loop terminates. When this occurs, the initial instruction outside the loop will be the instruction after [PC].

```
          DO #33, end_loop ;DO is a two-word instruction
          NOP              ;2nd word of DO executes as a NOP
end_loop: ADD W2,W3,W4     ;First instruction in DO loop([PC])
```

2.10 Address Register Dependencies

The dsPIC30F architecture supports a data space read (source) and a data space write (destination) for most MCU class instructions. The effective address (EA) calculation by the AGU and subsequent data space read or write, each take a period of 1 instruction cycle to complete. This timing causes the data space read and write operations for each instruction to partially overlap, as shown in Figure 2-21. Because of this overlap, a 'Read-After-Write' (RAW) data dependency can occur across instruction boundaries. RAW data dependencies are detected and handled at run-time by the dsPIC30F CPU.

Figure 2-21: Data Space Access Timing



2.10.1 Read-After-Write Dependency Rules

If the W register is used as a write operation destination in the current instruction and the W register being read in the prefetched instruction are the same, the following rules will apply:

- If the destination write (current instruction) does not modify the contents of Wn, no stalls will occur.

  or

- If the source read (prefetched instruction) does not calculate an EA using Wn, no stalls will occur.

During each instruction cycle, the dsPIC30F hardware automatically checks to see whether a RAW data dependency is about to occur. If the conditions specified above are not satisfied, the CPU will automatically add one instruction cycle delay before executing the prefetched instruction. The instruction stall provides enough time for the destination W register write to take place before the next (prefetched) instruction has to use the written data.

Table 2-7: Read-After-Write Dependency Summary

Destination Addressing Mode using Wn	Source Addressing Mode using Wn	Status	Examples (Wn = W2)
Direct	Direct	Allowed	`ADD.w W0, W1, W2` `MOV.w W2, W3`
Direct	Indirect	Stall	`ADD.w W0, W1, W2` `MOV.w [W2], W3`
Direct	Indirect with modification	Stall	`ADD.w W0, W1, W2` `MOV.w [W2++], W3`
Indirect	Direct	Allowed	`ADD.w W0, W1, [W2]` `MOV.w W2, W3`
Indirect	Indirect	Allowed	`ADD.w W0, W1, [W2]` `MOV.w [W2], W3`
Indirect	Indirect with modification	Allowed	`ADD.w W0, W1, [W2]` `MOV.w [W2++], W3`
Indirect with modification	Direct	Allowed	`ADD.w W0, W1, [W2++]` `MOV.w W2, W3`

Indirect	Indirect	Stall	`ADD.w W0, W1, [W2]`
			`MOV.w [W2], W3`
			`; W2=0x0004 (mapped W2)`

Indirect	Indirect with modification	Stall	`ADD.w W0, W1, [W2]`
			`MOV.w [W2++], W3`
			`; W2=0x0004 (mapped W2)`

Indirect with modification	Indirect	Stall	`ADD.w W0, W1, [W2++]` `MOV.w [W2], W3`
Indirect with modification	Indirect with modification	Stall	`ADD.w W0, W1, [W2++]` `MOV.w [W2++], W3`

2.10.2 Instruction Stall Cycles

An instruction stall is essentially a one-instruction cycle wait period appended in front of the read phase of an instruction, in order to allow the prior write to complete before the next read operation. For the purposes of interrupt latency, it should be noted that the stall cycle is associated with the instruction following the instruction where it was detected (i.e., stall cycles always precede instruction execution cycles).

If a RAW data dependency is detected, the dsPIC30F will begin an instruction stall. During an instruction stall, the following events occur:

1. The write operation underway (for the previous instruction) is allowed to complete as normal.
2. Data space is not addressed until after the instruction stall.
3. PC increment is inhibited until after the instruction stall.
4. Further instruction fetches are inhibited until after the instruction stall.

2.10.2.1 Instruction Stall Cycles and Interrupts

When an interrupt event coincides with two adjacent instructions that will cause an instruction stall, one of two possible outcomes could occur:

- The interrupt could be coincident with the first instruction. In this situation, the first instruction will be allowed to complete and the second instruction will be executed after the ISR completes. In this case, the stall cycle is eliminated from the second instruction because the exception process provides time for the first instruction to complete the write phase.
- The interrupt could be coincident with the second instruction. In this situation, the second instruction and the appended stall cycle will be allowed to execute prior to the ISR. In this case, the stall cycle associated with the second instruction executes normally. However, the stall cycle will be effectively absorbed into the exception process timing. The exception process proceeds as if an ordinary two-cycle instruction was interrupted.

2.10.2.2 Instruction Stall Cycles and Flow Change Instructions

The `CALL` and `RCALL` instructions write to the stack using W15 and may, therefore, force an instruction stall prior to the next instruction, if the source read of the next instruction uses W15.

The `RETFIE and RETURN` instructions can never force an instruction stall prior to the next instruction because they only perform read operations. However, the user should note that the `RETLW` instruction could force a stall, because it writes to a W register during the last cycle.

The `GOTO` and branch instructions can never force an instruction stall because they do not perform write operations.

2.10.2.3 Instruction Stalls and DO and REPEAT Loops

Other than the addition of instruction stall cycles, RAW data dependencies will not affect the operation of either DO or REPEAT loops.

The prefetched instruction within a REPEAT loop does not change until the loop is complete or an exception occurs. Although register dependency checks occur across instruction boundaries, the dsPIC30F effectively compares the source and destination of the same instruction during a REPEAT loop.

The last instruction of a DO loop either prefetches the instruction at the loop start address or the next instruction (outside the loop). The instruction stall decision will be based on the last instruction in the loop and the contents of the prefetched instruction.

2.10.2.4 Instruction Stalls and Program Space Visibility (PSV)

When program space (PS) is mapped to data space by enabling the PSV (CORCON<2>) bit, and the X space EA falls within the visible program space window, the read or write cycle is redirected to the address in program space. Accessing data from program space takes up to 3 instruction cycles.

Instructions operating in PSV address space are subject to RAW data dependencies and consequent instruction stalls, just like any other instruction.

Consider the following code segment:

```
ADD    W0,[W1],[W2++]   ; PSV = 1, W1=0x8000, PSVPAG=0xAA
MOV    [W2],[W3]
```

This sequence of instructions would take 5 instruction cycles to execute. Two instruction cycles are added to perform the PSV access via W1. Furthermore, an instruction stall cycle is inserted to resolve the RAW data dependency caused by W2.

2.11 Register Maps

A summary of the registers associated with the dsPIC30F CPU core is provided in Table 2-8.

Table 2-8: dsPIC30F Core Register Map

Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset State
W0	0000	W0 (WREG)																0000 0000 0000 0000
W1	0002	W1																0000 0000 0000 0000
W2	0004	W2																0000 0000 0000 0000
W3	0006	W3																0000 0000 0000 0000
W4	0008	W4																0000 0000 0000 0000
W5	000A	W5																0000 0000 0000 0000
W6	000C	W6																0000 0000 0000 0000
W7	000E	W7																0000 0000 0000 0000
W8	0010	W8																0000 0000 0000 0000
W9	0012	W9																0000 0000 0000 0000
W10	0014	W10																0000 0000 0000 0000
W11	0016	W11																0000 0000 0000 0000
W12	0018	W12																0000 0000 0000 0000
W13	001A	W13																0000 0000 0000 0000
W14	001C	W14																0000 0000 0000 0000
W15	001E	W15																0000 0000 0000 0000
SPLIM	0020	SPLIM																0000 0000 0000 0000
ACCAL	0022	ACCAL																0000 0000 0000 0000
ACCAH	0024	ACCAH																0000 0000 0000 0000
ACCAU	0026	Sign-extension of ACCA<39>								ACCAU								0000 0000 0000 0000
ACCBL	0028	ACCBL																0000 0000 0000 0000
ACCBH	002A	ACCBH																0000 0000 0000 0000
ACCBU	002C	Sign-extension of ACCB<39>								ACCBU								0000 0000 0000 0000
PCL	002E	PCL															0	0000 0000 0000 0000
PCH	0030	—	—	—	—	—	—	—	—	—	PCH							0000 0000 0000 0000
TBLPAG	0032	—	—	—	—	—	—	—	—	TBLPAG								0000 0000 0000 0000
PSVPAG	0034	—	—	—	—	—	—	—	—	PSVPAG								0000 0000 0000 0000
RCOUNT	0036	RCOUNT																xxxx xxxx xxxx xxxx
DCOUNT	0038	DCOUNT																xxxx xxxx xxxx xxxx
DOSTARTL	003A	DOSTARTL															0	xxxx xxxx xxxx xxx0
DOSTARTH	003C	—	—	—	—	—	—	—	—	—	—	DOSTARTH						0000 0000 00xx xxxx
DOENDL	003E	DOENDL															0	xxxx xxxx xxxx xxx0
DOENDH	0040	—	—	—	—	—	—	—	—	—	—	DOENDH						0000 0000 00xx xxxx
SR	0042	OA	OB	SA	SB	OAB	SAB	DA	DC	IPL2	IPL1	IPL0	RA	N	OV	Z	C	0000 0000 0000 0000
CORCON	0044	—	—	—	US	EDT	DL2	DL<1:0>		SATA	SATB	SATDW	ACCSAT	IPL3	PSV	RND	IF	0000 0000 0010 0000
MODCON	0046	XMODEN	YMODEN	—	—	BWM<3:0>				YWM<3:0>				XWM<3:0>				0000 0000 0000 0000
XMODSRT	0048	XMODSRT<15:0>															0	xxxx xxxx xxxx xxx0

Table 2-8: dsPIC30F Core Register Map (Continued)

Name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Reset State
XMODEND	004A	XMODEND<15:0>															1	xxxx xxxx xxxx xxx1
YMODSRT	004C	YMODSRT<15:0>															0	xxxx xxxx xxxx xxx0
YMODEND	004E	YMODEND<15:0>															1	xxxx xxxx xxxx xxx1
XBREV	0050	BREN	XBREV<14:0>															xxxx xxxx xxxx xxxx
DISICNT	0052	—	—	DISICNT<13:0>														0000 0000 0000 0000
Reserved	0054 - 007E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000 0000 0000 0000

Legend: x = uninitiated

Note: Refer to the device data sheet for specific Core Register Map details.

2.12 Related Application Notes

This section lists application notes that are related to this section of the manual. These application notes may not be written specifically for the dsPIC30F Product Family, but the concepts are pertinent and could be used with modification and possible limitations. The current application notes related to the dsPIC30F CPU module are:

Title	Application Note #
No related application notes at this time.	

Note: Please visit the Microchip web site (www.microchip.com) for additional Application Notes and code examples for the dsPIC30F Family of devices.

2.13 Revision History

Revision A

This is the initial released revision of this document.

Revision B

This revision incorporates additional technical content for the dsPIC30F CPU module.

Revision C

This revision incorporates all known errata at the time of this document update.

Revision D (August 2007)

This revision corrects LS-bit definitions for PCH and PCL registers (see register maps in Table 2-8).