Section 2. CPU

HIGHLIGHTS

This section of the manual contains the following topics:

2.1 INTRODUCTION

The PIC24F CPU module has a 16-bit (data) modified Harvard architecture with an enhanced instruction set. 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 `REPEAT` instructions, which are interruptible at any point.

The PIC24F 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 15th working register (W14) can be used as a Stack Frame Pointer when used with `LNK` and `UNLK` instructions.

The upper 32 Kbytes of the data space memory map can optionally be mapped into program space at any 16K word program boundary defined by the 8-bit Program Space Visibility Page (PSVPAG) register. The data to program space mapping feature lets any instruction access program space as if it were data space. Refer to **Section 4.4 "Program Space Visibility from Data Space"** for more information on Program Space Visibility.

The Instruction Set Architecture (ISA) is significantly enhanced beyond that of the PIC18F but maintains an acceptable level of backward compatibility. All PIC18F instructions and addressing modes are supported either directly or through simple macros. Many of the ISA enhancements are driven by compiler efficiency needs.

The core supports Inherent (no operand), Relative, Literal and Memory Direct Addressing modes, and 3 groups of addressing modes (MODE1, MODE2 and MODE3). All modes support Register Direct and various Register Indirect Addressing modes. Each group offers up to seven addressing modes. Instructions are associated with predefined addressing modes depending upon their functional requirements.

There is also a 'Register Indirect with Signed 10-Bit Offset' Addressing mode dedicated to two special move instructions, `LDWLO` and `STWLO`. Refer to **Section 32. "Instruction Set"** for more details.

For most instructions, the core 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, three parameter instructions can be supported, allowing A + B = C operations to be executed in a single cycle.

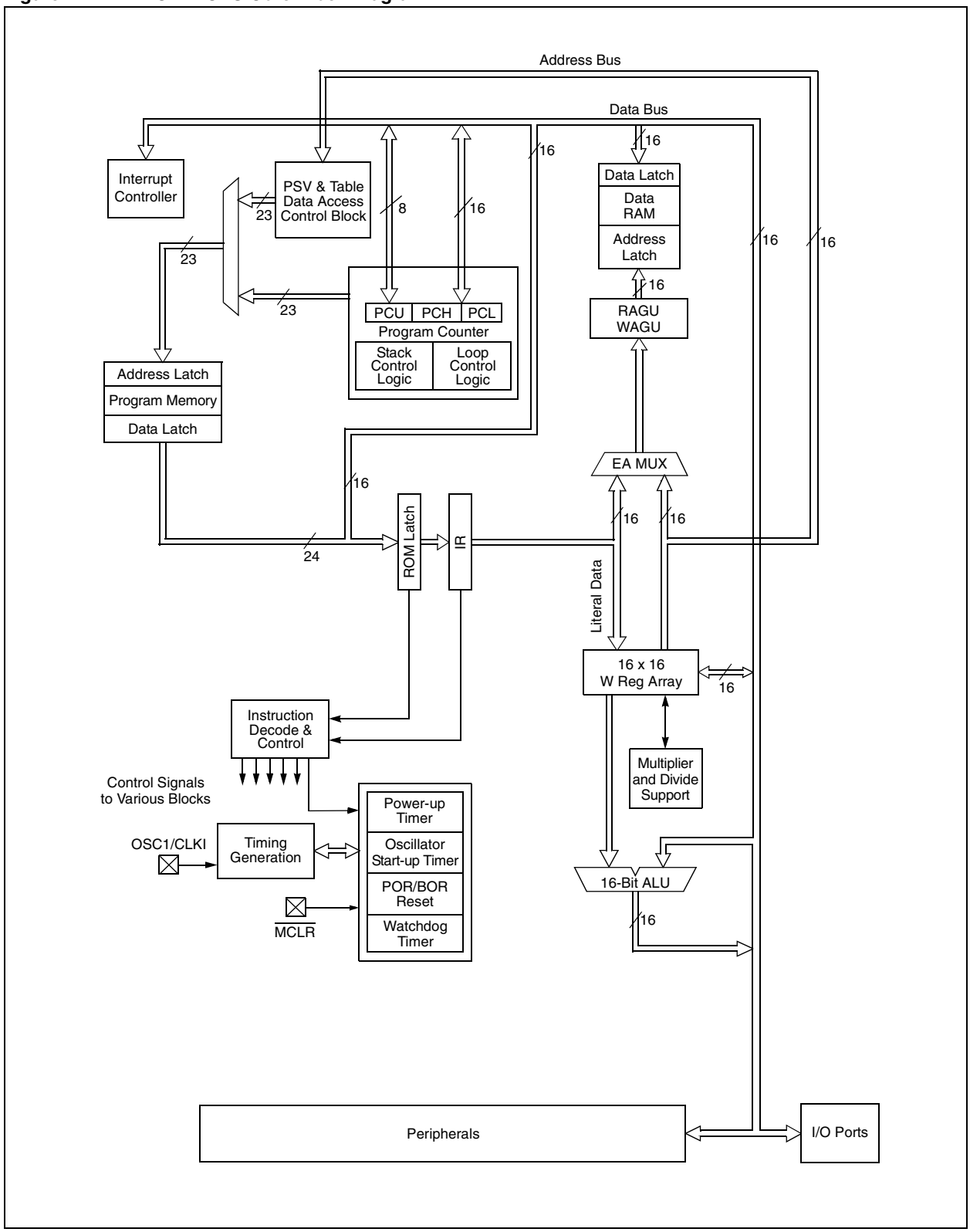
A high-speed, 17-bit by 17-bit multiplier is included to significantly enhance the core arithmetic capability and throughput. The multiplier supports Signed, Unsigned and Mixed mode, 16-bit by 16-bit or 8-bit by 8-bit integer multiplication. All multiply instructions execute in a single cycle.

The 16-bit ALU is enhanced with integer divide assist hardware that supports an iterative non-restoring divide algorithm. It operates in conjunction with the `REPEAT` instruction looping mechanism, and a selection of iterative divide instructions, to support 32-bit (or 16-bit) divided by 16-bit integer signed and unsigned division. All divide operations require 19 cycles to complete, but are interruptible at any cycle boundary.

The PIC24F has a vectored exception scheme with up to 8 sources of non-maskable traps and interrupt sources. Each interrupt source can be assigned to one of seven priority levels.

A block diagram of the CPU is shown in Figure 2-1.

Figure 2-1: PIC24F CPU Core Block Diagram



2.2 PROGRAMMER'S MODEL

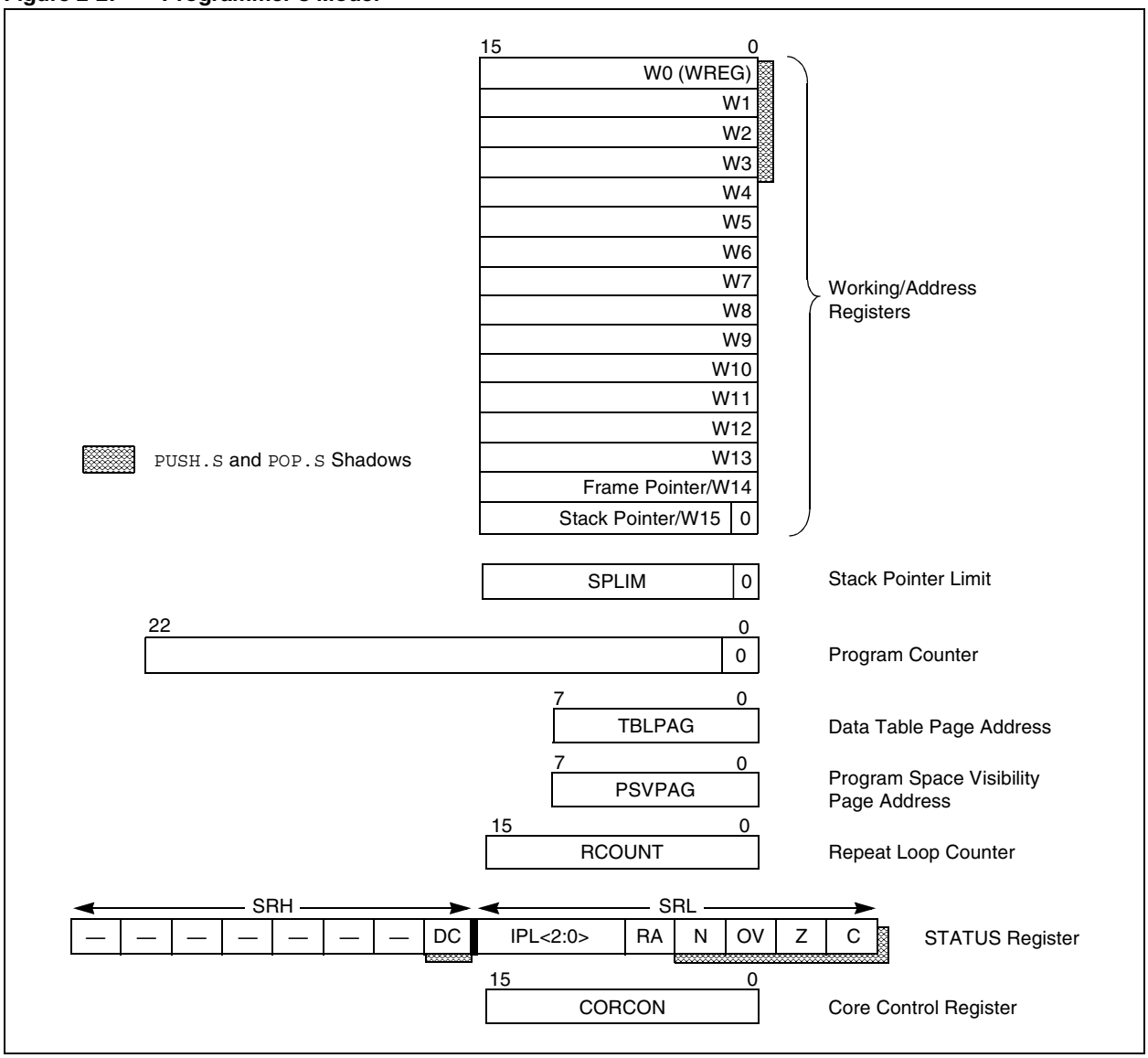
The programmer's model for the PIC24F is shown in Figure 2-2. All registers in the programmer's model are memory mapped and can be manipulated directly by instructions. A description of each register is provided in Table 2-1.

Table 2-1: Programmer's Model Register Descriptions

Register(s) Name	Description
W0 through W15	Working register array
PC	23-bit Program Counter
SR	ALU STATUS register
SPLIM	Stack Pointer Limit Value register
TBLPAG	Table Memory Page Address register
PSVPAG	Program Space Visibility Page Address register
RCOUNT	Repeat Loop Counter register
CORCON	CPU Control register

All registers associated with the programmer's model are memory mapped, as shown in Table 2-5.

Figure 2-2: Programmer's Model



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 PIC24F 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:

Example 2-1: Register Instructions

```
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 PICmicro® 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:

Example 2-2: File Register Instructions

```
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"* (DS70030).

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:

Example 2-3: Access W Register in File Register Instruction

```
    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:

Example 2-4: W Register Used as Address Pointer and Operand Destination

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

where:

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

In Example 2-4, 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 PIC24F 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

Some of the registers have a shadow register associated with them as shown in Table 2-5. The shadow register is used as a temporary holding register and can transfer its contents to or from its host register upon some occurring event. None of the shadow registers are accessible directly. The `PUSH.S` and `POP.S` shadow rule is applied to register transfer into and out of shadows.

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 in Example 2-5.

Example 2-5: `PUSH.S` and `POP.S` Instructions

```
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.

2.2.3 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. Refer to **Section 7. "Reset"** for more details (check Microchip web site for availability: www.microchip.com).

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 (e.g., creating stack frames).

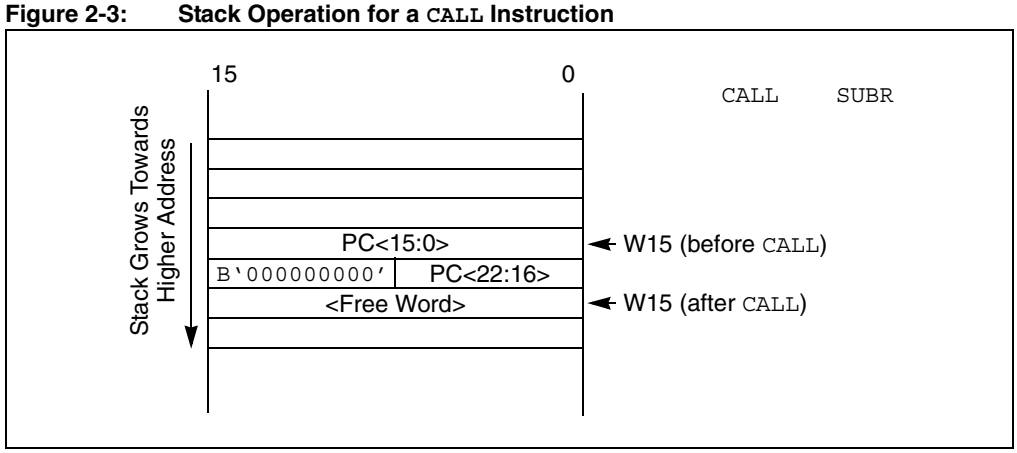
Note:	In order to protect against misaligned stack accesses, W15<0> is fixed to '0' by the hardware.

W15 is initialized to 0x0800 during all Resets. This address ensures that the Stack Pointer (SP) will point to valid RAM in all PIC24F 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 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, the PC<15:0> bits are pushed onto the first available stack word; then, the PC<22:16> bits are pushed onto the second available stack location. For a PC push during any CALL instruction, the MSB of the PC is zero-extended before the push, as shown in Figure 2-3. During exception processing, the MSB 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 (TOS) 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 is initialized to 0x0800. Furthermore, this example assumes that the values 0x5A5A and 0x3636 are 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

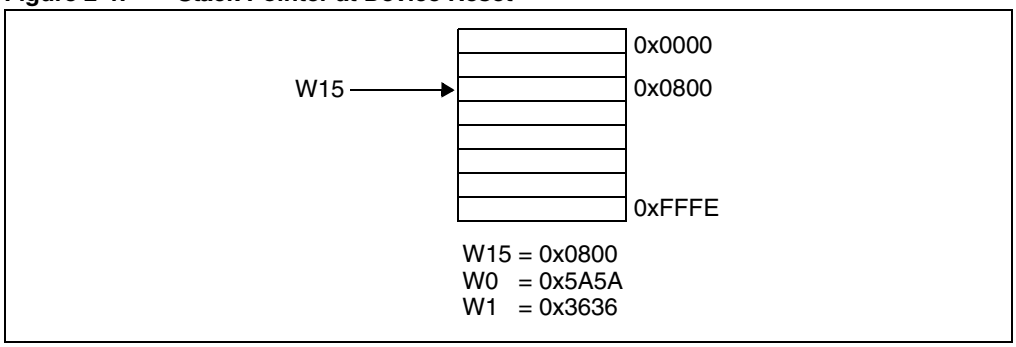


Figure 2-5: Stack Pointer After the First PUSH Instruction

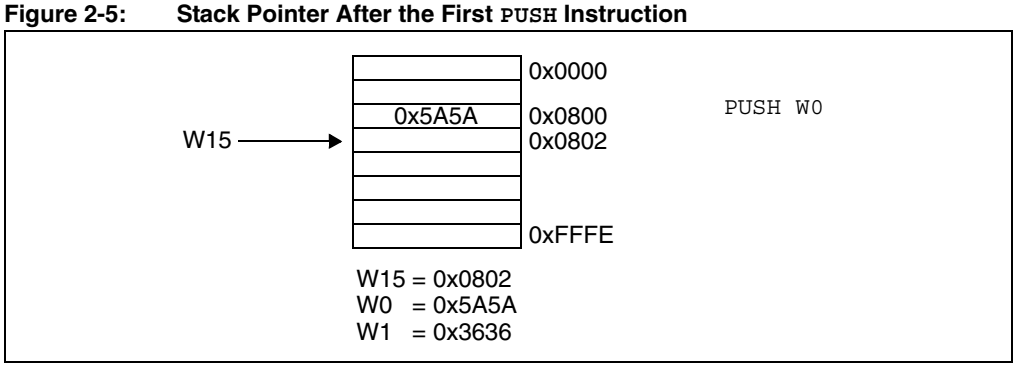


Figure 2-6: Stack Pointer After the Second PUSH Instruction

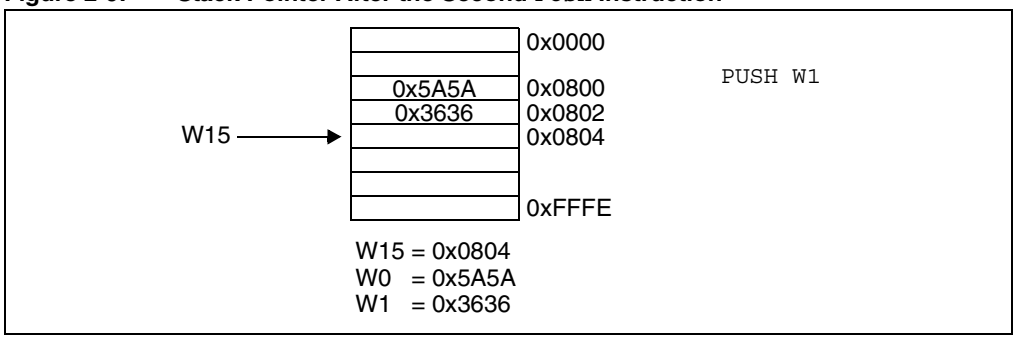
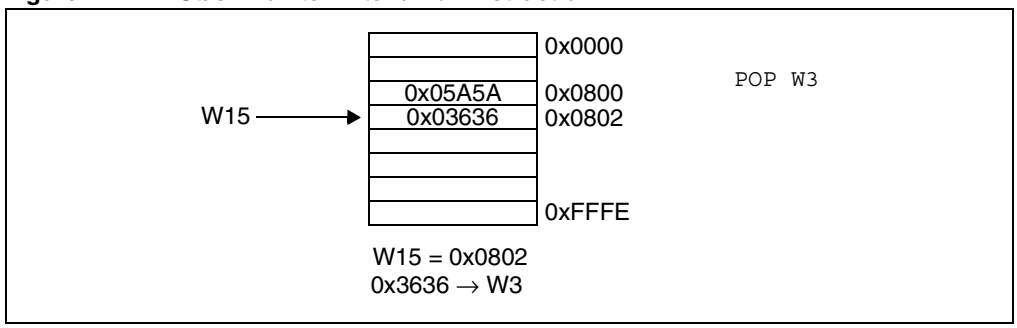


Figure 2-7: Stack Pointer After a POP Instruction



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 Stack Frame Pointer.

Refer to the *"dsPIC30F Programmer's Reference Manual"* (DS70030) for software examples that use W14 as a Stack Frame Pointer.

2.3.3 Stack Pointer Overflow

There is a Stack Pointer 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 an interrupt occurs, a stack error trap will be generated.

If stack overflow checking is enabled, a stack error trap also occurs 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 8. "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 PIC24F CPU has a 16-bit STATUS register (SR), the LSB 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 Level Status bits, IPL<2:0>, and the REPEAT Loop Active Status bit, RA (SR<4>). During exception processing, SRL is concatenated with the MSB of the PC to form a complete word value which is then stacked.

SRH contains only the Digit Carry bit, DC (SR<8>).

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

1. The RA bit (SR<4>): RA is a read-only bit.
2. IPL<2:0>: When register is disabled (NSTDIS = 1), IPL<2:0> bits become read-only.

Note: A description of the SR bits affected by each instruction is provided in the *"dsPIC30F Programmer's Reference Manual"* (DS70030).

Register 2-1: SR: CPU STATUS Register

U-0	U-0	U-0	U-0	U-0	U-0	U -0	R/W-0
—	—	—	—	—	—	—	DC
bit 15							bit 8

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

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

bit 15-9 **Unimplemented:** Read as '0'

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

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)

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 = Last operation resulted in zero
0 = Last operation did not result in zero

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

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>).

Register 2-2: CORCON: Core Control Register

U-0	U-0	U-0	U-0	U-0	U-0	U -0	U -0
—	—	—	—	—	—	—	—
bit 15							bit 8

U-0	U-0	U-0	U-0	R/C-0	R/W-0	U-0	U-0
—	—	—	—	IPL3(1)	PSV	—	—
bit 7							bit 0

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

bit 15-4 **Unimplemented:** Read as '0'

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

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 1-0 **Unimplemented:** Read as '0'

Note 1: User interrupts are disabled when IPL3 = 1.

2.4.2 Other PIC24F CPU Control Registers

The registers listed below are associated with the PIC24F CPU core, but are described in further detail in other sections of this manual.

2.4.2.1 TBLPAG: TABLE PAGE ADDRESS POINTER

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 (check Microchip web site for availability: www.microchip.com).

2.4.2.2 PSVPAG: PROGRAM MEMORY VISIBILITY PAGE ADDRESS POINTER

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 PIC24F 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 (check Microchip web site for availability: www.microchip.com).

2.4.2.3 DISICNT: DISABLE INTERRUPTS COUNTER 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 8. "Interrupts"** for further information.

2.5 ARITHMETIC LOGIC UNIT (ALU)

The PIC24F 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/Borrow (C), Zero (Z), Negative (N), Overflow (OV) and Half Carry/Borrow (DC) Status bits in the SR register. The C and DC Status bits operate as 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"* (DS70030) 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 PICmicro devices, the ALU result from all byte operations is written back as a byte (i.e., MSB not modified) and the CPU STATUS register, SR, is updated based only upon the state of the LSB of the result.
2:	All register instructions performed in Byte mode only affect the LSB of the W registers. The MSB of any W register can be modified by using file register instructions that access the memory mapped contents of the W registers.

2.6 MULTIPLICATION AND DIVIDE SUPPORT

2.6.1 Overview

The PIC24F core contains a 17-bit x 17-bit multiplier and is capable of unsigned, signed or mixed sign operation with the following multiplication modes:

1. 16-Bit x 16-Bit Signed
2. 16-Bit x 16-Bit Unsigned
3. 16-Bit Signed x 5-Bit (literal) Unsigned
4. 16-Bit Unsigned x 16-Bit Unsigned
5. 16-Bit Unsigned x 5-Bit (literal) Unsigned
6. 16-Bit Unsigned x 16-Bit Signed
7. 8-Bit Unsigned x 8-Bit Unsigned

The divide block is capable of supporting 32-bit/16-bit and 16-bit/16-bit signed and unsigned integer divide operation with the following data sizes:

1. 32-bit signed/16-bit signed divide
2. 32-bit unsigned/16-bit unsigned divide
3. 16-bit signed/16-bit signed divide
4. 16-bit unsigned/16-bit unsigned divide

2.6.2 Multiplier

A block diagram of the multiplier is shown in Figure 2-8. It is used to support the multiply instructions which include integer 16-bit signed, unsigned and mixed sign multiplies, including the PIC18F unsigned multiply, `MULWF` (`MUL.w` and `MUL.b`). All multiply instructions only support Register Direct Addressing mode for the result. A 32-bit result (from all multiplies other than `MULWF`) is written to any two aligned consecutive W register pairs, except W15:W14, which are not allowed.

The `MULWF` instruction may be directed to use byte or word sized operands. The destination is always the W3:W2 register pair in the W array. Byte multiplicands will direct a 16-bit result to W2 (W3 is not changed), and word multiplicands will direct a 32-bit result to W3:W2.

Note:	The destination register pair for multiply instructions must be ‘aligned’ (i.e., odd:even), where ‘odd’ contains the most significant result word and ‘even’ contains the least significant result word. For example, W3:W2 is acceptable, whereas W4:W3 is not.

The multiplicands for all multiply instructions (other than `MULWF` which is a special case) are derived from the W array (1st word) and data space (2nd word). `MULWF` derives its multiplicands from W2 (1st word or byte) and data space (2nd word or byte) using a zero-extended, 13-bit absolute address.

Additional data paths are provided to allow these instructions to write the result back into the W array and data bus (via the W array) as shown in Figure 2-8.

2.6.2.1 SINGLE AND MIXED MODE INTEGERS

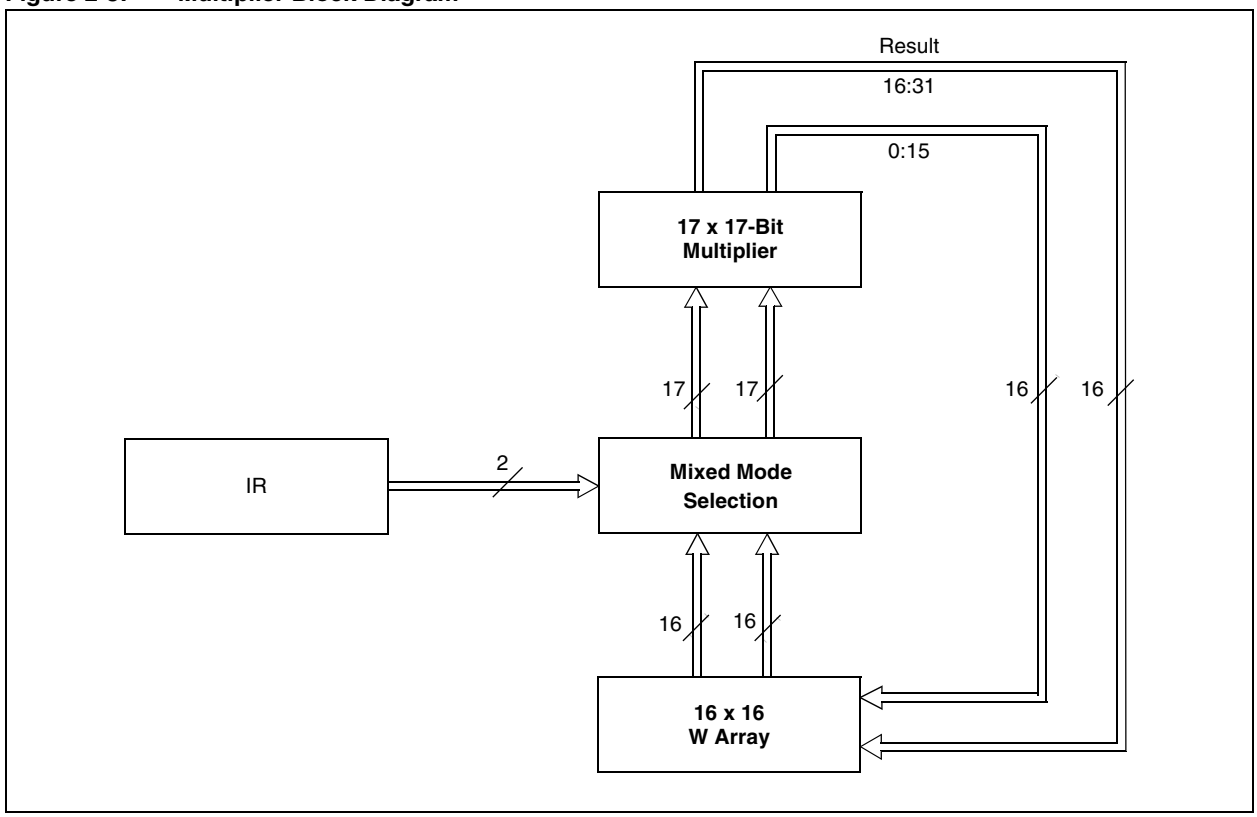
Simple data preprocessing logic either zero or sign-extends all operands to 17 bits, such that unsigned, signed or mixed sign multiplications can be executed as signed values. All unsigned operands are always zero-extended into the 17th bit of the multiplier input value. All signed operands are always sign-extended into the 17th bit of the multiplier input value.

For unsigned 16-bit multiplies, the multiplier produces a 32-bit, unsigned result.

For signed 16-bit multiplies, the multiplier produces 30 bits of data and 2 bits of sign.

For 16-Bit Mixed mode (signed/unsigned) multiplies, the multiplier produces 31 bits of data and 1 bit of sign.

Figure 2-8: Multiplier Block Diagram



2.6.3 Divider

The PIC24F features both 32-bit/16-bit and 16-bit/16-bit signed and unsigned, integer divide operations which are implemented as single instruction iterative divides.

The quotient for all divide instructions ends up in W0 and the remainder in W1. 16-bit signed and unsigned DIV instructions can specify any W register for both the 16-bit divisor (Wn) and any W register (aligned) pair (W(m + 1):Wm) for the 32-bit dividend. The divide algorithm takes one cycle per bit of divisor, so both 32-bit/16-bit and 16-bit/16-bit instructions take the same number of cycles to execute.

The divide instructions must be executed within a REPEAT loop. Any other form of execution (e.g., a series of discrete divide instructions) will not function correctly because the instruction flow function is conditional on RCOUNT. The divide flow does not automatically set up the REPEAT, which must therefore, be explicitly executed with the correct operand value as shown in Table 2-2 (REPEAT will execute the target instruction {operand value + 1} time).

Table 2-2: Divide Execution Time

Instruction	Description	Iterations	REPEAT Operand Value	Total Execution Time (including REPEAT)
DIV.SD	Signed divide: W(m + 1):Wm/Wn → W0; Rem → W1	18	17	19
DIV.SW	Signed divide: Wm/Wn → W0; Rem → W1	18	17	19
DIV.UD	Unsigned divide: W(m + 1):Wm/Wn → W0; Rem → W1	18	17	19
DIV.UW	Unsigned divide: Wm/Wn → W0; Rem → W1	18	17	19

All intermediate data is saved in W1:W0 after each iteration. The N, C and Z Status flags are used to convey control information between iterations. Consequently, although the divide instructions are listed as 19 cycle operations, the divide iterative sequence is interruptible, just like any other REPEAT loop.

Dividing by zero will initiate an arithmetic error trap. The divisor is evaluated during the first cycle of the divide instruction, so the first cycle will be executed prior to the start of exception processing for the trap. Refer to **Section 8. "Interrupts"** for more details.

2.7 COMPILER FRIENDLY ARCHITECTURE

The core architecture is designed to produce an efficient (code size and speed) C compiler.

- For most instructions, the core 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 parameter instructions are supported, allowing A + B = C operations to be executed in a single cycle.
- Instruction addressing modes are very flexible and are matched closely to compiler needs.
- There are sixteen, 16 x 16-bit working register arrays, each of which can act as data, address or offset registers. One working register (W15) operates as a software stack for interrupts and calls.
- Linear indirect access of all data space is supported, plus the memory direct address range is extended to 8 Kbytes, with the addition of 16-bit direct address load and store instructions.
- Linear indirect access of 32K word (64 Kbyte) pages within program space (user and test space) is supported using any working register via new table read and write instructions.
- Part of the data space can be mapped into program space, allowing constant data to be accessed as if it were in data space using PSV mode.

2.8 MULTI-BIT SHIFT SUPPORT

The PIC24F core supports single-cycle, multi-bit arithmetic and logic shifts using a shifter block. It also supports single bit shifts through the ALU. The multi-bit shifter is capable of performing up to a 15-bit arithmetic right shift, or up to a 15-bit left shift, in a single cycle.

A full summary of instructions that use the shift operation is provided below in Table 2-3.

Table 2-3: Instructions Using Single and Multi-Bit Shift Operations

Instruction	Description
ASR	Arithmetic shift right source register by one or more bits.
SL	Shift left source register by one or more bits.
LSR	Logical shift right source register by one or more bits.

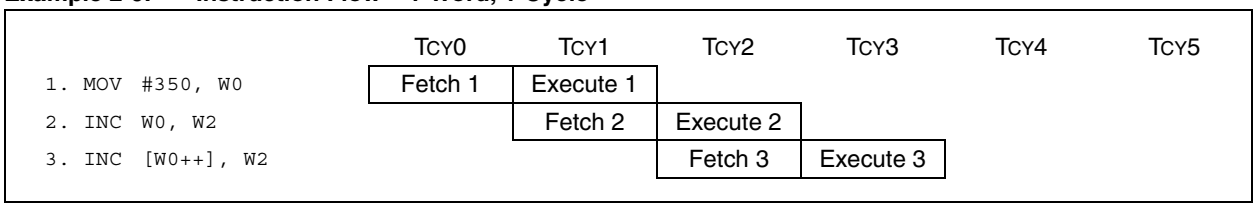
All multi-bit shift instructions only support Register Direct Addressing mode for both the operand source and result destination.

2.9 INSTRUCTION FLOW TYPES

Most instructions in the PIC24F 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 six different types of instruction flow in the PIC24F architecture. These are described below:

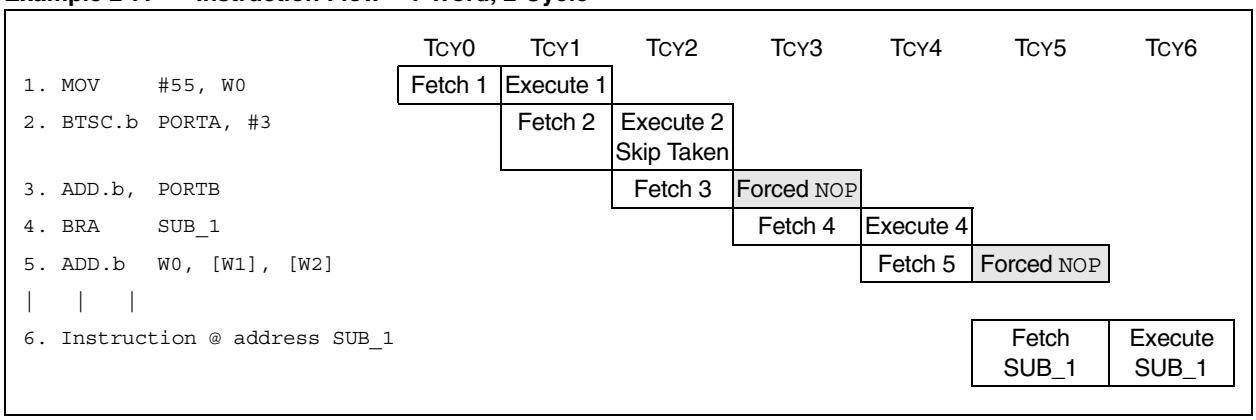
1. **1 Instruction Word, 1 Instruction Cycle:**
   These instructions will take one instruction cycle to execute as shown in Example 2-6.

Example 2-6: Instruction Flow – 1-Word, 1-Cycle



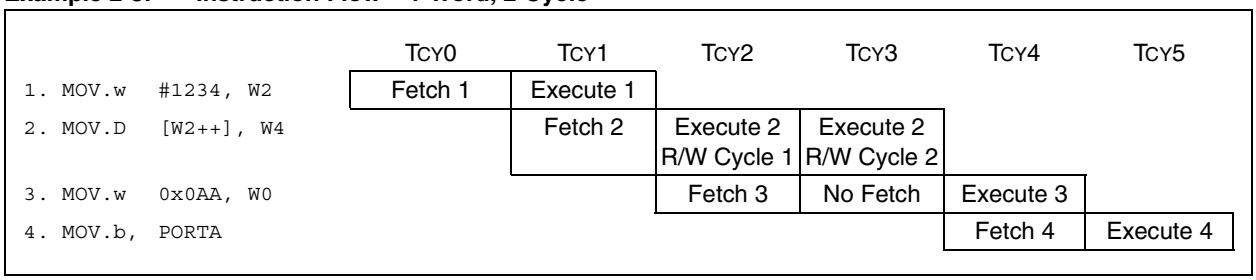
2. **1 Instruction Word, 2 Instruction Cycles:**
   These instructions include the relative branches, relative call, skips and returns. When an instruction changes the PC (other than to increment it), the pipelined fetch is discarded. This makes the instruction take two effective cycles to execute as shown in Example 2-7.

Example 2-7: Instruction Flow – 1-Word, 2-Cycle



3. **1 Instruction Word, 2 Instruction Cycles (Double Operation):**
   The only instructions of this type are the LDDW and STDW (load and store double word). As the data access has to be sequential, two cycles are required to complete these instructions as shown in Example 2-8.

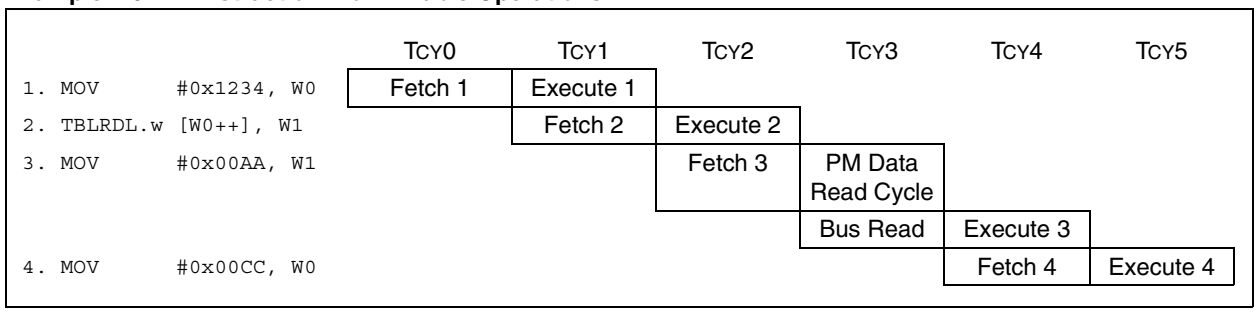
Example 2-8: Instruction Flow – 1-Word, 2-Cycle



4. 1 Instruction Word, 2 Instruction Cycles Table Operations:

These instructions will suspend fetching to insert a read or write cycle into the program memory. The instruction fetched while executing the table operation is saved for one cycle and executed in the cycle immediately after the table operation as shown in Example 2-9.

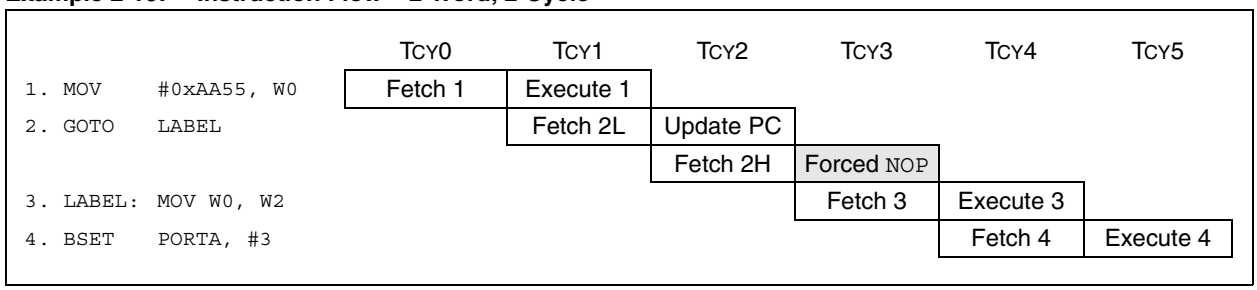
Example 2-9: Instruction Flow – Table Operations



5. 2 Instruction Words, 2 Instruction Cycles (GOTO, CALL):

In these instructions, the fetch after the instruction contains data. This results in a 2-cycle instruction as shown in Example 2-10. The second word of a 2-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 2-word instruction is skipped by a skip instruction (see Example 2-13).

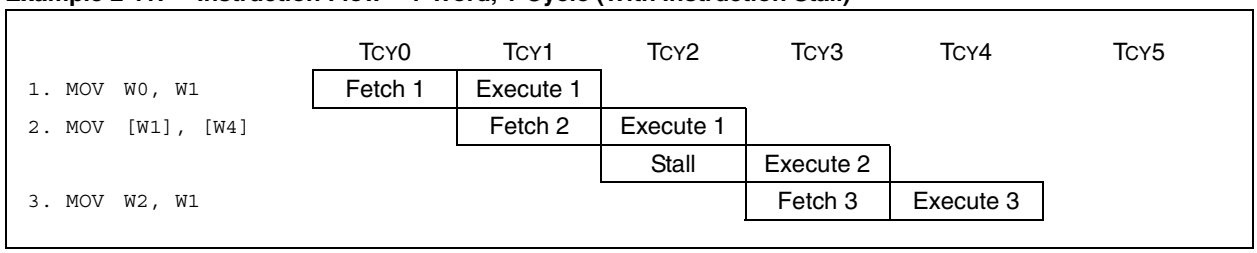
Example 2-10: Instruction Flow – 2-Word, 2-Cycle



6. Address Register Dependencies:

These are instructions that are subjected to a stall due to a data address dependency between the data space read and write operations. An additional cycle is inserted to resolve the resource conflict as discussed in **Section 2.11 "Address Register Dependencies"**.

Example 2-11: Instruction Flow – 1-Word, 1-Cycle (With Instruction Stall)



2.10 PROGRAM FLOW LOOP CONTROL

The PIC24F supports REPEAT instruction construct to provide unconditional automatic program loop control. The REPEAT instruction is used to implement a single instruction program loop. The instruction uses control bits within the CPU STATUS register, SR, to temporarily modify CPU operation.

2.10.1 REPEAT Loop

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:

Example 2-12: REPEAT Instruction Syntax

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

or

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

2.10.1.1 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 Status bit, RA (SR<4>), 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 Example 2-13 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.

Example 2-13: REPEAT Instruction Pipeline Flow

	TCY0	TCY1	TCY2	TCY3	TCY4	TCY5
1.REPEAT #0x2	Fetch 1	Execute 1				
2.MOV [W0++], [W1++]		Fetch 2	Execute 2			
			No Fetch	Execute 2		
				No Fetch	Execute 2	
3.BSET PORTA, #3					Fetch 3	Execute 3

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

2.10.1.2 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 Loop Counter) 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 Program Space (PS) using Program Space Visibility (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.

2.10.1.2.1 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.

2.10.1.3 RESTRICTIONS ON THE REPEAT INSTRUCTION

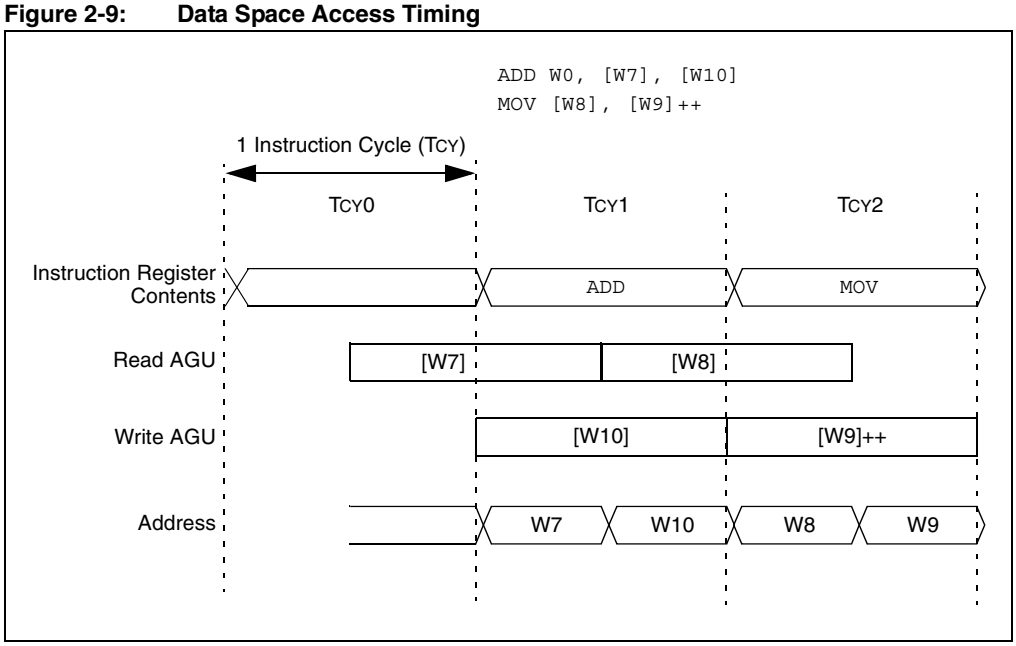
Any instruction can immediately follow a REPEAT except for the following:

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

2.11 ADDRESS REGISTER DEPENDENCIES

The PIC24F architecture supports a data space read (source) and a data space write (destination) for most instructions. The Effective Address (EA) calculation by the Address Generator Unit (AGU), and subsequent data space read or write, each take a period of one instruction cycle to complete. This timing causes the data space read and write operations for each instruction to partially overlap, as shown in Figure 2-9. 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 PIC24F CPU.

Figure 2-9: Data Space Access Timing



2.11.1 Read-After-Write Dependency Rules

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

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

   or

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

During each instruction cycle, the PIC24F hardware automatically checks to see if a RAW data dependency is about to occur. If the conditions specified above are not satisfied, the CPU will automatically add a 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 2-4: 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`

2.11.2 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 PIC24F 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.

2.11.2.1 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:

1. 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.
2. 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 2-cycle instruction was interrupted.

2.11.2.2 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.

2.11.2.3 INSTRUCTION STALLS AND REPEAT LOOPS

Other than the addition of instruction stall cycles, RAW data dependencies will not affect the operation of 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 PIC24F effectively compares the source and destination of the same instruction during a REPEAT loop.

2.11.2.4 INSTRUCTION STALLS AND PROGRAM SPACE VISIBILITY (PSV)

When Program Space Visibility (PSV) is enabled and the Effective Address (EA) falls within the visible PSV 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 instruction stalls, just like any other instruction. Although the instruction stall and PSV cycles both occur at the beginning of an instruction, it is not possible to combine them. If a stall occurs coincidentally with a PSV cycle, the stall cycle will be forced first, then the PSV cycle and finally, the instruction cycle.

Consider the following code segment:

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

This sequence of instructions would take five 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.

During a stalled instruction, the ROM Latch is transferred to the IR on the rising Q1 of the first cycle, and the Flash data read is transferred to the ROM Latch on the rising Q3 of the 2nd cycle of the instruction, as shown in Figure 2-9.

2.12 REGISTER MAPS

A summary of the registers associated with the PIC24F CPU core is provided in Table 2-5.

Table 2-5: Core SFR Memory Map (User Mode)

Name	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	All Resets
W0	Working Register 0																0000
W1	Working Register 1																0000
W2	Working Register 2																0000
W3	Working Register 3																0000
W4	Working Register 4																0000
W5	Working Register 5																0000
W6	Working Register 6																0000
W7	Working Register 7																0000
W8	Working Register 8																0000
W9	Working Register 9																0000
W10	Working Register 10																0000
W11	Working Register 11																0000
W12	Working Register 12																0000
W13	Working Register 13																0000
W14	Working Register 14																0000
W15	Working Register 15																0800
SPLIM	Stack Pointer Limit																xxxx
PCL	Program Counter, Low Word																0000
PCH	—	—	—	—	—	—	—	—	Program Counter, High Byte								0000
TBLPAG	—	—	—	—	—	—	—	—	Table Page Addess Pointer								0000
PSVPAG	—	—	—	—	—	—	—	—	Program Memory Visibility Page Address Pointer								0000
RCOUNT	Repeat Loop Counter																xxxx
SR	—	—	—	—	—	—	—	DC	IPL2	IPL1	IPL0	RA	N	OV	Z	C	0000
CORCON	—	—	—	—	—	—	—	—	—	—	—	—	IPL3	PSV	—	—	0000
DISICNT	—	—	Disable Interrupts Counter														xxxx

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexidecimal.

Note 1: Refer to the device data sheet for specific core register map details.

2.13 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 PIC24F device family, but the concepts are pertinent and could be used with modification and possible limitations. The current application notes related to the CPU 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 PIC24F family of devices.

2.14 REVISION HISTORY

Revision A (April 2006)

This is the initial released revision of this document.

NOTES:

