Information contained in this publication regarding device applications and the like is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION, INCLUDING BUT NOT LIMITED TO ITS CONDITION, QUALITY, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR PURPOSE. Microchip disclaims all liability arising from this information and its use. Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights.

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip’s Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as “unbreakable.”

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip’s code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.
# Table of Contents

## SECTION 1. INTRODUCTION

- Introduction .................................................................................................................. 6
- Manual Objective ............................................................................................................ 6
- Development Support .................................................................................................... 6
- Style and Symbol Conventions ..................................................................................... 7
- Instruction Set Symbols ................................................................................................. 8

## SECTION 2. PROGRAMMER'S MODEL

- 16-bit MCU and DSC Core Architecture Overview ....................................................... 10
- Programmer's Model ........................................................................................................ 14
- Working Register Array .................................................................................................. 18
- Default Working Register (WREG) ............................................................................... 18
- Software Stack Frame Pointer ....................................................................................... 18
- Software Stack Pointer .................................................................................................. 19
- Stack Pointer Limit Register (SPLIM) .......................................................................... 19
- Accumulator A and Accumulator B (dsPIC30F, dsPIC33F and dsPIC33E Devices) .... 19
- Program Counter ............................................................................................................ 19
- TBLPAG Register .......................................................................................................... 19
- PHSVAG Register (PIC24F, PIC24H, dsPIC30F and dsPIC33F) .................................. 19
- RCOUNT Register ........................................................................................................... 20
- DCOUNT Register (dsPIC30F, dsPIC33F and dsPIC33E Devices) .............................. 20
- DOSTART Register (dsPIC30F, dsPIC33F and dsPIC33E Devices) ............................ 20
- DOEND Register (dsPIC30F, dsPIC33F and dsPIC33E Devices) ................................. 21
- STATUS Register ............................................................................................................ 21
- Core Control Register .................................................................................................... 24
- Shadow Registers .......................................................................................................... 24
- DO Stack (dsPIC33E Devices) ........................................................................................ 25

## SECTION 3. INSTRUCTION SET OVERVIEW

- Introduction .................................................................................................................. 38
- Instruction Set Overview ............................................................................................... 38
- Instruction Set Summary Tables ..................................................................................... 40

## SECTION 4. INSTRUCTION SET DETAILS

- Data Addressing Modes .................................................................................................. 52
- Program Addressing Modes ........................................................................................... 61
- Instruction Stalls ............................................................................................................ 62
- Byte Operations ............................................................................................................. 64
- Word Move Operations .................................................................................................. 66
- Using 10-bit Literal Operands ....................................................................................... 69
- Software Stack Pointer and Frame Pointer ................................................................... 70
- Conditional Branch Instructions ................................................................................... 76
- Z Status Bit .................................................................................................................... 77
- Assigned Working Register Usage ................................................................................ 78
- DSP Data Formats (dsPIC30F, dsPIC33F and dsPIC33E Devices) ............................... 81
- Accumulator Usage (dsPIC30F, dsPIC33F and dsPIC33E Devices) ........................... 83
- Accumulator Access (dsPIC30F, dsPIC33F and dsPIC33E Devices) ........................... 84
- DSP MAC Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices) ....................... 84
- DSP Accumulator Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices) .......... 88
- Scaling Data with the FBCL Instruction (dsPIC30F, dsPIC33F and dsPIC33E Devices) . 88
- Normalizing the Accumulator with the FBCL Instruction (dsPIC30F, dsPIC33F and dsPIC33E Devices) .................................................................................. 90
Section 1. Introduction

HIGHLIGHTS

This section of the manual contains the following major topics:

1.1 Introduction ....................................................................................................................... 6
1.2 Manual Objective ............................................................................................................. 6
1.3 Development Support ...................................................................................................... 6
1.4 Style and Symbol Conventions ....................................................................................... 7
1.5 Instruction Set Symbols ................................................................................................. 8
1.1 INTRODUCTION

Microchip Technology focuses on products for the embedded control market. Microchip is a leading supplier of the following devices and products:

• 8-bit General Purpose Microcontrollers (PIC® MCUs)
• 16-bit Digital Signal Controllers (dsPIC® DSCs)
• 16-bit and 32-bit Microcontrollers (MCUs)
• Speciality and Standard Nonvolatile Memory Devices
• Security Devices (K EELoo® Security ICs)
• Application-specific Standard Products

Information about these devices and products, with corresponding technical documentation, is available on the Microchip web site (www.microchip.com).

1.2 MANUAL OBJECTIVE

This manual is a software developer’s reference for the 16-bit MCU and DSC device families. It describes the Instruction Set in detail and also provides general information to assist the development of software for the 16-bit MCU and DSC device families.

This manual does not include detailed information about the core, peripherals, system integration or device-specific information. The user should refer to the specific device family reference manual for information about the core, peripherals and system integration. For device-specific information, the user should refer to the specific device data sheets. The information that can be found in the data sheets includes:

• Device memory map
• Device pinout and packaging details
• Device electrical specifications
• List of peripherals included on the device

Code examples are given throughout this manual. These examples are valid for any device in the 16-bit MCU and DSC families.

1.3 DEVELOPMENT SUPPORT

Microchip offers a wide range of development tools that allow users to efficiently develop and debug application code. Microchip’s development tools can be broken down into four categories:

• Code generation
• Hardware/Software debug
• Device programmer
• Product evaluation boards

Information about the latest tools, product briefs and user guides can be obtained from the Microchip web site (www.microchip.com) or from your local Microchip Sales Office.

Microchip offers other reference tools to speed the development cycle. These include:

• Application Notes
• Reference Designs
• Microchip web site
• Local Sales Offices with Field Application Support
• Corporate Support Line

The Microchip web site also lists other sites that may be useful references.
### 1.4 STYLE AND SYMBOL CONVENTIONS

Throughout this document, certain style and font format conventions are used. Table 1-1 provides a description of the conventions used in this document.

<table>
<thead>
<tr>
<th>Symbol or Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>To force a bit/register to a value of logic ‘1’.</td>
</tr>
<tr>
<td>clear</td>
<td>To force a bit/register to a value of logic ‘0’.</td>
</tr>
</tbody>
</table>
| Reset          | 1. To force a register/bit to its default state.  
                2. A condition in which the device places itself after a device Reset occurs. Some bits will be forced to ‘0’ (such as interrupt enable bits), while others will be forced to ‘1’ (such as the I/O data direction bits). |
| 0xnnnn         | Designates the number ‘nnnn’ in the hexadecimal number system. These conventions are used in the code examples. For example, 0x013F or 0xA800. |
| (colon)        | Used to specify a range or the concatenation of registers/bits/pins. One example is ACCAU:ACCAH:ACCAL, which is the concatenation of three registers to form the 40-bit Accumulator. Concatenation order (left-right) usually specifies a positional relationship (MSb to LSb, higher to lower). |
| < >            | Specifies bit locations in a particular register. One example is SR<7:5> (or IPL<2:0>), which specifies the register and associated bits or bit locations. |
| LSB, MSb       | Indicates the Least Significant or Most Significant bit in a field. |
| LSB, MSB       | Indicates the Least/Most Significant Byte in a field of bits. |
| ls, ms        | Indicates the least/most significant word in a field of bits. |
| Courier New Font | Used for code examples, binary numbers and for Instruction mnemonics in the text. |
| Times New Roman Font, Italic | Used for equations and variables. |
| Times New Roman Font, Bold Italic | Used in explanatory text for items called out from a figure, equation, or example. |
| Note:          | A Note presents information that we want to re-emphasize, either to help you avoid a common pitfall, or make you aware of operating differences between some device family members. A Note can be in a box, or when used in a table or figure, it is located at the bottom of the table or figure. |
1.5 INSTRUCTION SET SYMBOLS

The summary tables in Section 3.2 “Instruction Set Overview” and Section 7.2 “Instruction Set Summary Table”, and the instruction descriptions in Section 5.4 “Instruction Descriptions” utilize the symbols shown in Table 1-2.

Table 1-2: Symbols Used in Instruction Summary Tables and Descriptions

<table>
<thead>
<tr>
<th>Symbol(1)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>Optional field or operation</td>
</tr>
<tr>
<td>[text]</td>
<td>The location addressed by text</td>
</tr>
<tr>
<td>(text)</td>
<td>The contents of text</td>
</tr>
<tr>
<td>#text</td>
<td>The literal defined by text</td>
</tr>
<tr>
<td>a ∈ [b, c, d]</td>
<td>“a” must be in the set of [b, c, d]</td>
</tr>
<tr>
<td>&lt;n:m&gt;</td>
<td>Register bit field</td>
</tr>
<tr>
<td>(label:)</td>
<td>Optional label name</td>
</tr>
<tr>
<td>Acc</td>
<td>Accumulator A or Accumulator B</td>
</tr>
<tr>
<td>AWB</td>
<td>Accumulator Write Back</td>
</tr>
<tr>
<td>bit4</td>
<td>4-bit wide bit position (0:7 in Byte mode, 0:15 in Word mode)</td>
</tr>
<tr>
<td>Expr</td>
<td>Absolute address, label or expression (resolved by the linker)</td>
</tr>
<tr>
<td>f</td>
<td>File register address</td>
</tr>
<tr>
<td>lit1</td>
<td>1-bit literal (0:1)</td>
</tr>
<tr>
<td>lit4</td>
<td>4-bit literal (0:15)</td>
</tr>
<tr>
<td>lit5</td>
<td>5-bit literal (0:31)</td>
</tr>
<tr>
<td>lit8</td>
<td>8-bit literal (0:255)</td>
</tr>
<tr>
<td>lit10</td>
<td>10-bit literal (0:255 in Byte mode, 0:1023 in Word mode)</td>
</tr>
<tr>
<td>lit14</td>
<td>14-bit literal (0:16383)</td>
</tr>
<tr>
<td>lit16</td>
<td>16-bit literal (0:65535)</td>
</tr>
<tr>
<td>lit23</td>
<td>23-bit literal (0:8388607)</td>
</tr>
<tr>
<td>Slit4</td>
<td>Signed 4-bit literal (-8:7)</td>
</tr>
<tr>
<td>Slit6</td>
<td>Signed 6-bit literal (-32:31) (range is limited to -16:16)</td>
</tr>
<tr>
<td>Slit10</td>
<td>Signed 10-bit literal (-512:511)</td>
</tr>
<tr>
<td>Slit16</td>
<td>Signed 16-bit literal (-32768:32767)</td>
</tr>
<tr>
<td>TOS</td>
<td>Top-of-Stack</td>
</tr>
<tr>
<td>Wb</td>
<td>Base working register</td>
</tr>
<tr>
<td>Wd</td>
<td>Destination working register (direct and indirect addressing)</td>
</tr>
<tr>
<td>Wdo</td>
<td>Destination working register (direct and indirect addressing, including indirect addressing with offset)</td>
</tr>
<tr>
<td>Wm, Wn</td>
<td>Working register divide pair (dividend, divisor)</td>
</tr>
<tr>
<td>Wm * Wm</td>
<td>Working register multiplier pair (same source register)</td>
</tr>
<tr>
<td>Wm * Wn</td>
<td>Working register multiplier pair (different source registers)</td>
</tr>
<tr>
<td>Wn</td>
<td>Both source and destination working register (direct addressing)</td>
</tr>
<tr>
<td>Wnd</td>
<td>Destination working register (direct addressing)</td>
</tr>
<tr>
<td>Wns</td>
<td>Source working register (direct addressing)</td>
</tr>
<tr>
<td>WREG</td>
<td>Default working register (assigned to W0)</td>
</tr>
<tr>
<td>Ws</td>
<td>Source working register (direct and indirect addressing)</td>
</tr>
<tr>
<td>Wso</td>
<td>Source working register (direct and indirect addressing, including indirect addressing with offset)</td>
</tr>
<tr>
<td>Wx</td>
<td>Source Addressing mode and working register for X data bus prefetch</td>
</tr>
<tr>
<td>Wxd</td>
<td>Destination working register for X data bus prefetch</td>
</tr>
<tr>
<td>Wy</td>
<td>Source Addressing mode and working register for Y data bus prefetch</td>
</tr>
<tr>
<td>Wyd</td>
<td>Destination working register for Y data bus prefetch</td>
</tr>
</tbody>
</table>

Note 1: The range of each symbol is instruction dependent. Refer to Section 5. “Instruction Descriptions” for the specific instruction range.
Section 2. Programmer’s Model

HIGHLIGHTS

This section of the manual contains the following major topics:

2.1 16-bit MCU and DSC Core Architecture Overview ....................................................... 10
2.2 Programmer’s Model..................................................................................................... 14
2.3 Working Register Array............................................................................................... 18
2.4 Default Working Register (WREG) .............................................................................. 18
2.5 Software Stack Frame Pointer..................................................................................... 18
2.1 16-BIT MCU AND DSC CORE ARCHITECTURE OVERVIEW

This section provides an overview of the 16-bit architecture features and capabilities for the following families of devices:

- 16-bit Microcontrollers (MCU):
  - PIC24F
  - PIC24H
  - PIC24E
- 16-bit Digital Signal Controllers (DSC):
  - dsPIC30F
  - dsPIC33F
  - dsPIC33E

2.1.1 Features Specific to 16-bit MCU and DSC Core

The core of the 16-bit MCU and DSC devices is a 16-bit (data) modified Harvard architecture with an enhanced instruction set. The core has a 24-bit instruction word, with an 8-bit Op code field. The Program Counter (PC) is 23 bits wide and addresses up to 4M x 24 bits of user program memory space. An instruction prefetch mechanism is used to help maintain throughput and provides predictable execution. The majority of instructions execute in a single cycle.

2.1.1.1 REGISTERS

The 16-bit MCU and DSC devices have sixteen 16-bit working registers. Each of the working registers can act as a data, address or offset register. The 16th working register (W15) operates as a software Stack Pointer for interrupts and calls.

2.1.1.2 INSTRUCTION SET

The instruction set is almost identical for the 16-bit MCU and DSC architectures. The instruction set includes many Addressing modes and was designed for optimum C compiler efficiency.

2.1.1.3 DATA SPACE ADDRESSING

The data space can be addressed as 32K words or 64 Kbytes. The upper 32 Kbytes of the data space memory map can optionally be mapped into program space at any 16K program word boundary, which is a feature known as Program Space Visibility (PSV). The program to data space mapping feature lets any instruction access program space as if it were the data space, which is useful for storing data coefficients.

Note: Some devices families support Extended Data Space (EDS) addressing. See the specific device data sheet and family reference manual for more details on this feature.

2.1.1.4 ADDRESSING MODES

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

For most instructions, the CPU 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.
2.1.1.5 ARITHMETIC AND LOGIC UNIT

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

The 16-bit Arithmetic Logic Unit (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.

2.1.1.6 EXCEPTION PROCESSING

The 16-bit MCU and DSC devices have a vectored exception scheme with support for up to 8 sources of non-maskable traps and up to 246 interrupt sources. In both families, each interrupt source can be assigned to one of seven priority levels.

2.1.2 PIC24E and dsPIC33E Features

In addition to the information provided in Section 2.1.1 “Features Specific to 16-bit MCU and DSC Core”, this section describes the enhancements that are available in the PIC24E and dsPIC33E families of devices.

2.1.2.1 DATA SPACE ADDRESSING

The Base Data Space address is used in conjunction with a read or write page register (DSRPAG or DSWPAG) to form an Extended Data Space (EDS) address, which can also be used for PSV access. The EDS can be addressed as 8 M words or 16 Mbytes. Refer to Section 3. “Data Memory” (DS70595) in the “dsPIC33E/PIC24E Family Reference Manual” for more details on EDS, PSV, and table accesses.

Note: Some PIC24F devices also support Extended Data Space. Refer to Section 44. “CPU with EDS” (DS39732) and Section 45. “Data Memory with EDS” (DS39733) of the PIC24F Family Reference Manual for details.

2.1.2.2 AUTOMATIC MIXED-SIGN MULTIPLICATION MODE (dsPIC33E ONLY)

In addition to signed and unsigned DSP multiplications, dsPIC33E devices support mixed-sign (unsigned-signed and signed-unsigned) multiplications without the need to dynamically reconfigure the multiplication mode and shift data to account for the difference in operand formats. This mode is particularly beneficial for executing extended-precision (32-bit and 64-bit) algorithms. Besides DSP instructions, MCU multiplication (MUL) instructions can also utilize either accumulator as a result destination, thereby enabling faster extended-precision arithmetic. Refer to 4.10.1 “Implied DSP Operands (dsPIC30F, dsPIC33F and dsPIC33E Devices)” and 4.18 “Extended-precision Arithmetic using mixed-sign multiplications (dsPIC33E only)” for more details on mixed-sign DSP multiplications.

2.1.2.3 MCU MULTIPLICATIONS WITH 16-BIT RESULT

16x16-bit MUL instructions include an option to store the product in a single 16-bit working register rather than a pair of registers. This feature helps free up a register for other purposes, in cases where the numbers being multiplied are small in magnitude and therefore expected to provide a 16-bit result. See the individual MUL instruction descriptions in 5.4 “Instruction Descriptions” for more details.

2.1.2.4 HARDWARE STACK FOR DO LOOPS (dsPIC33E ONLY)

The single-level DO loop shadow register-set has been replaced by 4-level deep DO loop hardware stack. This provides automatic DO loop register save/restore for up to 3 levels of DO loop nesting, resulting in more efficient implementation of nested loops. Refer to 2.19 “DO Stack (dsPIC33E Devices)” for more details on DO loop nesting in dsPIC33E devices.
2.1.2.5 DSP CONTEXT SWITCH SUPPORT (dsPIC33E ONLY)

In dsPIC33E devices, the DSP overflow and saturation status bits are writable. This allows the state of the DSP Engine to be efficiently saved and restored while switching between DSP tasks. See 2.16.4 “DSP ALU Status Bits (dsPIC30F, dsPIC33F and dsPIC33E Devices)” for more details on DSP status bits.

2.1.2.6 EXTENDED CALL AND GOTO INSTRUCTIONS

The new CALL.L Wn and GOTO.L Wn instructions extend the capabilities of the CALL Wn and GOTO Wn by enabling 32-bit addresses for computed branch/call destinations. In these enhanced instructions, the destination address is provided by a pair of working registers rather than a single 16-bit register. See the CALL.L and GOTO.L instruction descriptions in 5.4 “Instruction Descriptions” for more details.

2.1.2.7 COMPARE-BRANCH INSTRUCTIONS

dsPIC33E/PIC24E devices feature conditional Compare-Branch (CPBxx) instructions. These instructions extend the capabilities of the Compare-Skip (CPSxx) instructions by allowing branches rather than only skipping over a single instruction. See the CPBEQ, CPBNE, CPBGT and CPBLT instruction descriptions in 5.4 “Instruction Descriptions” for more details on compare-branch instructions.

2.1.3 dsPIC30F, dsPIC33F, and dsPIC33E Features

In addition to the information provided in Section 2.1.1 “Features Specific to 16-bit MCU and DSC Core”, this section describes the DSP enhancements that are available in the dsPIC30F, dsPIC33F, and dsPIC33E families of devices.

2.1.3.1 PROGRAMMING LOOP CONSTRUCTS

Overhead free program loop constructs are supported using the DO instruction, which is interruptible.

2.1.3.2 DSP INSTRUCTION CLASS

The DSP class of instructions are seamlessly integrated into the architecture and execute from a single execution unit.

2.1.3.3 DATA SPACE ADDRESSING

The data space 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. The DSP dual source class of instructions operates through the X and Y AGUs, which splits the data address space into two parts. The X and Y data space boundary is arbitrary and device-specific.

2.1.3.4 MODULO AND BIT-REVERSED ADDRESSING

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.

2.1.3.5 DSP ENGINE

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 bidirectional barrel shifter. The barrel shifter is capable of shifting a 40-bit value, up to 16 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 working 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.

2.1.3.6 EXCEPTION PROCESSING

The dsPIC30F devices have a vectored exception scheme with support for up to 8 sources of non-maskable traps and up to 54 interrupt sources. The dsPIC33F and dsPIC33E have a similar exception scheme, but support up to 118, and up to 246 interrupt sources, respectively. In all three families, each interrupt source can be assigned to one of seven priority levels.

2.2  PROGRAMMER’S MODEL

Figure 2-1 through Figure 2-4 show the programmer’s model diagrams for the 16-bit MCU and DSC families of devices.

Figure 2-1: PIC24F and PIC24H Programmer’s Model Diagram
Section 2. Programmer’s Model

Figure 2-2: PIC24E Programmer’s Model Diagram

- **W0/WREG**
- **W1**
- **W2**
- **W3**
- **W4**
- **W5**
- **W6**
- **W7**
- **W8**
- **W9**
- **W10**
- **W11**
- **W12**
- **W13**
- **W14/Frame Pointer**
- **W15/Stack Pointer**

- **DIV and MUL Result Registers**

- **Program Counter**
- **Data Table Page Address**
- **Data Space Read Page Address**
- **Data Space Write Page Address**
- **REPEAT Loop Counter**
- **CPU Core Control Register**
- **Status Register**

- **Stack Pointer Limit Register**

- **Working Registers**

**Legend**
- **PUSH.S and POP.S**
- **Shadow Registers**
- **TABPAG**
- **PSVPAG**
- **TBLPAG**
- **DSRPAG**
- **DSWPAG**
- **RCOUNT**
- **CORCON**
- **SRH**
- **SRL**
Figure 2-3: dsPIC30F and dsPIC33F Programmer's Model Diagram

<table>
<thead>
<tr>
<th>DIV and MUL Result Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0/WREG</td>
</tr>
<tr>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
</tr>
<tr>
<td>W3</td>
</tr>
<tr>
<td>W4</td>
</tr>
<tr>
<td>W5</td>
</tr>
<tr>
<td>W6</td>
</tr>
<tr>
<td>W7</td>
</tr>
<tr>
<td>W8</td>
</tr>
<tr>
<td>W9</td>
</tr>
<tr>
<td>W10</td>
</tr>
<tr>
<td>W11</td>
</tr>
<tr>
<td>W12</td>
</tr>
<tr>
<td>W13</td>
</tr>
<tr>
<td>W14/Frame Pointer</td>
</tr>
<tr>
<td>W15/Stack Pointer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAC Operand Registers</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>MAC Address Registers</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Working Registers</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Stack Pointer Limit Register</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Program Counter</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Data Table Page Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Program Space Visibility Page Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Repeat Loop Counter</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Do Loop Counter</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Do Loop Start Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Do Loop End Address</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>CPU Core Control Register</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Status Register</th>
</tr>
</thead>
</table>

Legend:
- PUSH.S Shadow Register
- DO Shadow Register
- Legend
- DIV and MUL Result Registers
- MAC Operand Registers
- MAC Address Registers
- Working Registers
- Stack Pointer Limit Register
- Program Counter
- Data Table Page Address
- Program Space Visibility Page Address
- Repeat Loop Counter
- Do Loop Counter
- Do Loop Start Address
- Do Loop End Address
- CPU Core Control Register
- Status Register
Section 2. Programmer’s Model

Figure 2-4: dsPIC33E Programmer’s Model Diagram
All registers in the programmer’s model are memory mapped and can be manipulated directly by the instruction set. A description of each register is provided in Table 2-1.

**Note:** Unless otherwise specified, the Programmer’s Model Register Descriptions in Table 2-1 apply to all MCU and DSC device families.

### Table 2-1: Programmer’s Model Register Descriptions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORCON</td>
<td>CPU Core Configuration register</td>
</tr>
<tr>
<td>PC</td>
<td>23-bit Program Counter</td>
</tr>
<tr>
<td>PSVPAG(^1)</td>
<td>Program Space Visibility Page Address register</td>
</tr>
<tr>
<td>DSRPAG(^2)</td>
<td>Extended Data Space (EDS) Read Page register</td>
</tr>
<tr>
<td>DSWPAG(^2)</td>
<td>Extended Data Space (EDS) Write Page register</td>
</tr>
<tr>
<td>RCOUNT</td>
<td>REPEAT Loop Count register</td>
</tr>
<tr>
<td>SPLIM</td>
<td>Stack Pointer Limit Value register</td>
</tr>
<tr>
<td>SR</td>
<td>ALU and DSP Engine STATUS register</td>
</tr>
<tr>
<td>TBLPAG</td>
<td>Table Memory Page Address register</td>
</tr>
<tr>
<td>W0-W15</td>
<td>Working register array</td>
</tr>
<tr>
<td>ACCA, ACCB(^3)</td>
<td>40-bit DSP Accumulators</td>
</tr>
<tr>
<td>DCOUNT(^3)</td>
<td>DO Loop Count register</td>
</tr>
<tr>
<td>DOSTART(^3)</td>
<td>DO Loop Start Address register</td>
</tr>
<tr>
<td>DOEND(^3)</td>
<td>DO Loop End Address register</td>
</tr>
</tbody>
</table>

**Note:**
1. This register is only available on PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
2. This register is only available on PIC24E and dsPIC33E devices.
3. This register is only available on dsPIC30F, dsPIC33F, and dsPIC33E devices.

### 2.3 WORKING REGISTER ARRAY

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

Byte instructions, which target the working register array, only affect the Least Significant Byte (LSB) 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.

### 2.4 DEFAULT WORKING REGISTER (WREG)

The instruction set can be divided into two instruction types: working register instructions and file register instructions. The working register instructions use the working register array as data values or as addresses that point to a memory location. In contrast, file register instructions operate on a specific memory address contained in the instruction opcode.

File register instructions that also utilize a working register do not specify the working register that is to be used for the instruction. Instead, a default working register (WREG) is used for these file register instructions. Working register, W0, is assigned to be the WREG. The WREG assignment is not programmable.

### 2.5 SOFTWARE STACK FRAME POINTER

A frame is a user-defined section of memory in the stack, used by a function to allocate memory for local variables. W14 has been assigned for use as a Stack Frame Pointer with the link (LNK) and unlink (ULNK) instructions. However, if a Stack Frame Pointer and the LNK and ULNK instructions are not used, W14 can be used by any instruction in the same manner as all other W registers. On dsPIC33E and PIC24E devices, a Stack Frame Active (SFA) Status bit is used to support nested stack frames. See Section 4.7.2 “Software Stack Frame Pointer” for detailed information about the Frame Pointer.
2.6 SOFTWARE STACK POINTER

W15 serves as a dedicated Software Stack Pointer, and will be automatically modified by function calls, exception processing 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. Refer to Section 4.7.1 “Software Stack Pointer” for detailed information about the Stack Pointer.

2.7 STACK POINTER LIMIT REGISTER (SPLIM)

The SPLIM is a 16-bit register associated with the Stack Pointer. It is used to prevent the Stack Pointer from overflowing and accessing memory beyond the user allocated region of stack memory. Refer to Section 4.7.3 “Stack Pointer Overflow” for detailed information about the SPLIM.

2.8 ACCUMULATOR A AND ACCUMULATOR B (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

Accumulator A (ACCA) and Accumulator B (ACCB) are 40-bit wide registers, utilized by DSP instructions to perform mathematical and shifting operations. Each accumulator is composed of 3 memory mapped registers:

- AccxU (bits 39-32)
- AccxH (bits 31-16)
- AccxL (bits 15-0)

In dsPIC33E devices, Accumulator A and Accumulator B can also be used as destination registers in MCU MUL.xx instructions. This helps reduce the execution time of extended-precision arithmetic operations.

Refer to Section 4.12 “Accumulator Usage (dsPIC30F, dsPIC33F and dsPIC33E Devices)” for details on using ACCA and ACCB.

2.9 PROGRAM COUNTER

The Program Counter (PC) is 23 bits wide. Instructions are addressed in the 4M x 24-bit user program memory space by PC<22:1>, where PC<0> is always set to ‘0’ to maintain instruction word alignment and provide compatibility with data space addressing. This means that during normal instruction execution, the PC increments by 2.

Program memory located at 0x800000 and above is utilized for device configuration data, Unit ID and Device ID. This region is not available for user code execution and the PC can not access this area. However, one may access this region of memory using table instructions. For details on accessing the configuration data, Unit ID, and Device ID, refer to the specific device family reference manual.

2.10 TBLPAG 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. For details on accessing program memory with the table instructions, refer to the family reference manual of the specific device.

2.11 PSVPAG REGISTER (PIC24F, PIC24H, dsPIC30F AND dsPIC33F)

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 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. For details on program space visibility, refer to the specific device family reference manual.
2.12  RCOUNT REGISTER

The 14-bit RCOUNT register (16-bit for PIC24E and dsPIC33E devices) register contains the loop counter for the REPEAT instruction. When a REPEAT instruction is executed, RCOUNT is loaded with the repeat count of the instruction, either "lit14" for the "REPEAT #lit14" instruction ("lit15" for the "REPEAT #lit15" instruction for PIC24E and dsPIC33E devices), or the 14 LSb of the Wn register for the "REPEAT Wn" instruction (entire Wn for PIC24E and dsPIC33E devices). The REPEAT loop will be executed RCOUNT + 1 time.

Note 1: If a REPEAT loop is executing and gets interrupted, RCOUNT may be cleared by the Interrupt Service Routine to break out of the REPEAT loop when the foreground code is re-entered.

2: Refer to the specific device family reference manual for complete details about REPEAT loops.

2.13  DCOUNT REGISTER (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The 14-bit DCOUNT register (16-bit for dsPIC33E devices) contains the loop counter for hardware DO loops. When a DO instruction is executed, DCOUNT is loaded with the loop count of the instruction, either "lit14" for the "DO #lit14,Expr" instruction ("lit15" for the "DO #lit15,Expr" instruction for dsPIC33E devices) or the 14 LSb of the Ws register for the "DO Ws,Expr" instruction (entire Wn for dsPIC33E devices). The DO loop will be executed DCOUNT + 1 times.

Note 1: In dsPIC30F and dsPIC33F devices, the DCOUNT register contains a shadow register. See Section 2.18 “Shadow Registers” for information on shadow registers.

2: The dsPIC33E devices have a 4-level-deep, nested DO stack instead of a shadow register.

3: Refer to the specific device family reference manual for complete details about DO loops.

2.14  DOSTART REGISTER (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The DOSTART register contains the starting address for a hardware DO loop. When a DO instruction is executed, DOSTART is loaded with the address of the instruction that follows the DO instruction. This location in memory is the start of the DO loop. When looping is activated, program execution continues with the instruction stored at the DOSTART address after the last instruction in the DO loop is executed. This mechanism allows for zero overhead looping.

Note 1: For dsPIC30F and dsPIC33F devices, DOSTART has a shadow register. See Section 2.18 “Shadow Registers” for information on shadowing.

2: The dsPIC33E devices have a 4-level-deep, nested DO stack instead of a shadow register. The DOSTART register is read-only in dsPIC33E devices.

3: Refer to the specific device family reference manual for complete details about DO loops.
2.15  DOEND REGISTER (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The DOEND register contains the ending address for a hardware **DO** loop. When a **DO** instruction is executed, DOEND is loaded with the address specified by the expression in the **DO** instruction. This location in memory specifies the last instruction in the **DO** loop. When looping is activated and the instruction stored at the DOEND address is executed, program execution will continue from the **DO** loop start address (stored in the DOSTART register).

| Note 1: | For dsPIC30F and dsPIC33F devices, DOEND has a shadow register. See Section 2.18 “Shadow Registers” for information on shadow registers. |
| 2: | The dsPIC33E devices have a 4-level-deep, nested **DO** stack instead of a shadow register. |
| 3: | Refer to the specific device family reference manual for complete details about **DO** loops. |

2.16  STATUS REGISTER

The 16-bit STATUS register maintains status information for the instructions which have been executed most recently. Operation Status bits exist for MCU operations, loop operations and DSP operations. Additionally, the STATUS register contains the CPU Interrupt Priority Level bits, IPL<2:0>, which are used for interrupt processing.

Depending on the MCU and DSC family, one of the following STATUS registers is used:

- Register 2-1 for PIC24F, PIC24H, and PIC24E devices
- Register 2-2 for dsPIC30F and dsPIC33F devices
- Register 2-3 for dsPIC33E devices

2.16.1  MCU ALU Status Bits

The MCU operation Status bits are either affected or used by the majority of instructions in the instruction set. Most of the logic, math, rotate/shift and bit instructions modify the MCU Status bits after execution, and the conditional Branch instructions use the state of individual Status bits to determine the flow of program execution. All conditional branch instructions are listed in Section 4.8 “Conditional Branch Instructions”.

The Carry (C), Zero (Z), Overflow (OV), Negative (N), and Digit Carry (DC) bits show the immediate status of the MCU ALU by indicating whether an operation has resulted in a Carry, Zero, Overflow, Negative result, or Digit Carry. When a subtract operation is performed, the C flag is used as a Borrow flag.

The Z Status bit is useful for extended precision arithmetic. The Z Status bit functions like a normal Z flag for all instructions except those that use a carry or borrow input (**ADDC**, **CPB**, **SUBB** and **SUBBR**). See Section 4.9 “Z Status Bit” for more detailed information.

| Note 1: | All MCU bits are shadowed during execution of the **PUSH.S** instruction and they are restored on execution of the **POP.S** instruction. |
| 2: | All MCU bits, except the DC flag (which is not in the SRL), are stacked during exception processing (see Section 4.7.1 “Software Stack Pointer”). |

2.16.2  REPEAT Loop Status Bit

The REPEAT Active bit (RA) is used to indicate when looping is active. The RA flag indicates that a REPEAT instruction is being executed, and it is only affected by the REPEAT instructions. The RA flag is set to ‘1’ when the instruction being repeated begins execution, and it is cleared when the instruction being repeated completes execution for the last time.

Since the RA flag is also read-only, it may not be directly cleared. However, if a REPEAT or its target instruction is interrupted, the Interrupt Service Routine may clear the RA flag of the SRL, which resides on the stack. This action will disable looping once program execution returns from the Interrupt Service Routine, because the restored RA will be ‘0’.
2.16.3 DO Active bit (DA) (dsPIC30F, dsPIC33F and dsPIC33E Devices)

The DO Active bit (DA) is used to indicate when looping is active. The DO instructions affect the DA flag, which indicates that a DO loop is active. The DA flag is set to '1' when the first instruction of the DO loop is executed, and it is cleared when the last instruction of the loop completes final execution.

The DA flag is read-only. This means that looping is not initiated by writing a '1' to DA, nor is it terminated by writing a '0' to DA. If a DO loop must be terminated prematurely, the EDT bit, CORCON<11>, should be used.
2.16.4 DSP ALU Status Bits (dsPIC30F, dsPIC33F and dsPIC33E Devices)

The high byte of the STATUS Register (SRH) is used by the DSP class of instructions, and it is modified when data passes through one of the adders. The SRH provides status information about overflow and saturation for both accumulators. The Saturate A, Saturate B, Overflow A and Overflow B (SA, SB, OA, OB) bits provide individual accumulator status, while the Saturate AB and Overflow AB (SAB, OAB) bits provide combined accumulator status. The SAB and OAB bits provide an efficient method for the software developer to check the register for saturation or overflow.

The OA and OB bits are used to indicate when an operation has generated an overflow into the guard bits (bits 32 through 39) of the respective accumulator. This condition can only occur when the processor is in Super Saturation mode, or if saturation is disabled. It indicates that the operation has generated a number which cannot be represented with the lower 31 bits of the accumulator. The OA and OB bits are writable in dsPIC33E devices.

The SA and SB bits are used to indicate when an operation has generated an overflow out of the MSb of the respective accumulator. The SA and SB bits are active, regardless of the Saturation mode (Disabled, Normal or Super) and may be considered “sticky”. Namely, once the SA or SB bit is set to ‘1’, it can only be cleared manually by software, regardless of subsequent DSP operations. When it is required, the BCLR instruction can be used to clear the SA or SB bit.

In addition, the SA and SB bits can be set by software in dsPIC33E devices, enabling efficient context state switching.

For convenience, the OA and OB bits are logically ORed together to form the OAB flag, and the SA and SB bits are logically ORed to form the SAB flag. These cumulative Status bits provide efficient overflow and saturation checking when an algorithm is implemented. Instead of interrogating the OA and the OB bits independently for arithmetic overflows, a single check of OAB can be performed. Likewise, when checking for saturation, SAB may be examined instead of checking both the SA and SB bits. Note that clearing the SAB flag will clear both the SA and SB bits.

2.16.5 Interrupt Priority Level Status Bits

The three Interrupt Priority Level (IPL) bits of the SRL, SR<7:5>, and the IPL3 bit, CORCON<3>, set the CPU's IPL which is used for exception processing. Exceptions consist of interrupts and hardware traps. Interrupts have a user-defined priority level between 0 and 7, while traps have a fixed priority level between 8 and 15. The fourth Interrupt Priority Level bit, IPL3, is a special IPL bit that may only be read or cleared by the user. This bit is only set when a hardware trap is activated and it is cleared after the trap is serviced.

The CPU's IPL identifies the lowest level exception which may interrupt the processor. The interrupt level of a pending exception must always be greater than the CPU's IPL for the CPU to process the exception. This means that if the IPL is 0, all exceptions at priority Level 1 and above may interrupt the processor. If the IPL is 7, only hardware traps may interrupt the processor.

When an exception is serviced, the IPL is automatically set to the priority level of the exception being serviced, which will disable all exceptions of equal and lower priority. However, since the IPL field is read/write, one may modify the lower three bits of the IPL in an Interrupt Service Routine to control which exceptions may preempt the exception processing. Since the SRL is stacked during exception processing, the original IPL is always restored after the exception is serviced. If required, one may also prevent exceptions from nesting by setting the NSTDIS bit (INTCON1<15>).

Note: For more detailed information on exception processing, refer to the family reference manual of the specific device.
2.17 CORE CONTROL REGISTER

For all MCU and DSC devices, the 16-bit CPU Core Control register (CORCON), is used to set the configuration of the CPU. This register provides the ability to map program space into data space.

In addition to setting CPU modes, the CORCON register contains status information about the IPL<3> Status bit, which indicates if a trap exception is being processed.

Depending on the MCU and DSC family, one of the following CORCON registers is used:

- Register 2-4 for PIC24F and PIC24H devices
- Register 2-5 for PIC24E devices
- Register 2-6 for dsPIC30F and dsPIC33F devices
- Register 2-7 for dsPIC33E devices

2.17.1 dsPIC30F, dsPIC33F, and dsPIC33E Specific bits

In addition to setting CPU modes, the following features are available through the CORCON register:

- Set the ACCA and ACCB saturation enable
- Set the Data Space Write Saturation mode
- Set the Accumulator Saturation and Rounding modes
- Set the Multiplier mode for DSP operations
- Terminate DO loops prematurely
- Provide status information about the DO loop nesting level (DL<2:0>)
- Select fixed or variable interrupt latency (dsPIC33E only)

2.17.1.1 PIC24E and dsPIC33E SPECIFIC BITS

A Status bit (SFA) is available that indicates whether the Stack Frame is active.

Note: PIC24E and dsPIC33E devices do not have a PSV control bit, it has been replaced by the SFA bit.

2.18 SHADOW REGISTERS

A shadow register is used as a temporary holding register and can transfer its contents to or from the associated host register when instructed. Some of the registers in the programmer’s model have a shadow register, which is utilized during the execution of a DO, POP.S, or PUSH.S instruction. Shadow register usage is shown in Table 2-2.

Note: The DO instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

<table>
<thead>
<tr>
<th>Location</th>
<th>DO(1)</th>
<th>POP.S/PUSH.S</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCOUNT(1)</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>DOSTART(1)</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>DOEND(1)</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>STATUS Register – DC, N, OV, Z and C bits</td>
<td>—</td>
<td>Yes</td>
</tr>
<tr>
<td>W0-W3</td>
<td>—</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note 1: The DO shadow registers are only available in dsPIC30F and dsPIC33F devices.

For dsPIC30F and dsPIC33F devices, since the DCOUNT, DOSTART and DOEND registers are shadowed, the ability to nest DO loops without additional overhead is provided. Since all shadow registers are one register deep, up to one level of DO loop nesting is possible. Further nesting of DO loops is possible in software, with support provided by the DO Loop Nesting Level Status bits (DL<2:0>) in the CORCON register (CORCON<10:8>).

Note: All shadow registers are one register deep and not directly accessible. Additional shadowing may be performed in software using the software stack.
2.19  DO STACK (dsPIC33E DEVICES)

The DO stack is used to preserve the following elements associated with a DO loop underway when another DO loop is encountered (i.e., a nested DO loop).

- DOSTART register value
- DOEND register value
- DCOUNT register value

Note that the DO level status field (DL<2:0>) also acts as a pointer to address the DO stack. After the DO instruction is executed, the DO level status field (DL<2:0>) points to the next free entry.

The DOSTART, DOEND, and DCOUNT registers each have an associated hardware stack that allows the DO loop hardware to support up to three levels of nesting. A conceptual representation of the DO stack is shown in Figure 2-5.

Figure 2-5: DO Stack Conceptual Diagram

<table>
<thead>
<tr>
<th>DL&lt;2:0&gt;</th>
<th>DOSTART</th>
<th>DOEND</th>
<th>DCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>Level 1 Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>Level 2 Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>Level 3 Registers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note 1: For DO register entries, DL<2:0> represents the value before the DO stack is executed.

2: For DO instruction buffer entries, DL<2:0> represents the value after the DO stack is executed.

3: If DL<2:0> = 0, no DO loops are active (DA = 0).
Register 2-1: SR: CPU STATUS Register (PIC24H, PIC24F and PIC24E Devices)

| Bit 15-9 | Unimplemented: Read as ‘0’ |
| Bit 8   | DC: MCU ALU Half Carry/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<sup>1,2</sup> |
|         | 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 that 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 that affects the Z bit has set it at some time in the past |
|         | 0 = The most recent operation that affects the Z bit has cleared it (i.e., a non-zero result) |
| Bit 0   | C: MCU ALU Carry/Borrow bit |
|         | 1 = A carry-out from the MSb occurred |
|         | 0 = No carry-out from the MSb occurred |

**Legend:**

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

**Note 1:** The IPL<2:0> bits are concatenated with the IPL3 bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL, if IPL3 = 1. User interrupts are disabled when IPL<3> = 1.

**Note 2:** The IPL<2:0> Status bits are read only when the NSTDIS bit (INTCON1<15>) = 1. Refer to the family reference manual of the specific device family to see the associated interrupt register.
## Register 2-2: SR: CPU STATUS Register (dsPIC30F and dsPIC33F Devices)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Value at POR</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>OA: Accumulator A Overflow bit</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>1 = Accumulator A overflowed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Accumulator A has not overflowed</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>OB: Accumulator B Overflow bit</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>1 = Accumulator B overflowed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Accumulator B has not overflowed</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SA: Accumulator A Saturation bit</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>1 = Accumulator A is saturated or has been saturated since this bit was last cleared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Accumulator A is not saturated</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>SB: Accumulator B Saturation bit</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>1 = Accumulator B is saturated or has been saturated at since this bit was last cleared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Accumulator B is not saturated</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>OAB: OA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Accumulator A or B has overflowed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Neither Accumulator A nor B has overflowed</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SAB: SA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 = Accumulator A or B has been saturated since this bit was last cleared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Neither Accumulator A nor B has been saturated since this bit was last cleared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Neither Accumulator A nor B has been saturated since this bit was last cleared</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = Neither Accumulator A nor B has been saturated since this bit was last cleared</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DA: DO Loop Active bit</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>1 = DO loop in progress</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = DO loop not in progress</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>DC: MCU ALU Half Carry bit</td>
<td>'1'</td>
</tr>
<tr>
<td></td>
<td>1 = A carry-out from the MSb of the lower nibble occurred</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 = No carry-out from the MSb of the lower nibble occurred</td>
<td></td>
</tr>
<tr>
<td>7-5</td>
<td>IPL&lt;2:0&gt;: Interrupt Priority Level bits</td>
<td></td>
</tr>
<tr>
<td></td>
<td>111 = CPU Interrupt Priority Level is 7 (15). User interrupts disabled</td>
<td></td>
</tr>
<tr>
<td></td>
<td>110 = CPU Interrupt Priority Level is 6 (14)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>101 = CPU Interrupt Priority Level is 5 (13)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100 = CPU Interrupt Priority Level is 4 (12)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>011 = CPU Interrupt Priority Level is 3 (11)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>010 = CPU Interrupt Priority Level is 2 (10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>001 = CPU Interrupt Priority Level is 1 (9)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>000 = CPU Interrupt Priority Level is 0 (8)</td>
<td></td>
</tr>
</tbody>
</table>

### Note
1: This bit may be read or cleared, but not set.
2: Once this bit is set, it must be cleared manually by software.
3: Clearing this bit will clear SA and SB.
4: This bit is read-only.
5: The IPL<2:0> bits are concatenated with the IPL3 bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL, if IPL3 = 1.
Register 2-2: SR: CPU STATUS Register (dsPIC30F and dsPIC33F Devices) (Continued)

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 = The result of the operation was negative
      0 = The result of the operation was not negative

bit 2  OV: MCU ALU Overflow bit
      1 = Overflow occurred
      0 = No overflow occurred

bit 1  Z: MCU ALU Zero bit
      1 = The result of the operation was zero
      0 = The result of the operation was not zero

bit 0  C: MCU ALU Carry/Borrow bit
      1 = A carry-out from the MSb occurred
      0 = No carry-out from the MSb occurred

Note 1: This bit may be read or cleared, but not set.
2: Once this bit is set, it must be cleared manually by software.
3: Clearing this bit will clear SA and SB.
4: This bit is read-only.
5: The IPL<2:0> bits are concatenated with the IPL3 bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL, if IPL3 = 1.
## Register 2-3: SR: CPU STATUS Register (dsPIC33E Devices)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>OA: Accumulator A Overflow Status bit</td>
</tr>
<tr>
<td>14</td>
<td>OB: Accumulator B Overflow Status bit</td>
</tr>
<tr>
<td>13</td>
<td>SA: Accumulator A Saturation Status bit</td>
</tr>
<tr>
<td>12</td>
<td>SB: Accumulator B Saturation Status bit</td>
</tr>
<tr>
<td>11</td>
<td>OAB: OA</td>
</tr>
<tr>
<td>10</td>
<td>SAB: SA</td>
</tr>
<tr>
<td>9</td>
<td>DA: DO Loop Active bit</td>
</tr>
<tr>
<td>8</td>
<td>DC: MCU ALU Half Carry/Borrow bit</td>
</tr>
</tbody>
</table>

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

### Bit Descriptions:
- **bit 15**: OA: Accumulator A Overflow Status bit
  - 1 = Accumulator A has overflowed
  - 0 = Accumulator A has not overflowed
- **bit 14**: OB: Accumulator B Overflow Status bit
  - 1 = Accumulator B has overflowed
  - 0 = Accumulator B has not overflowed
- **bit 13**: SA: Accumulator A Saturation Status bit
  - 1 = Accumulator A is saturated or has been saturated since this bit was last cleared
  - 0 = Accumulator A is not saturated
- **bit 12**: SB: Accumulator B Saturation Status bit
  - 1 = Accumulator B is saturated or has been saturated since this bit was last cleared
  - 0 = Accumulator B is not saturated
- **bit 11**: OAB: OA || OB Combined Accumulator Overflow Status bit
  - 1 = Accumulator A or B has overflowed
  - 0 = Neither Accumulator A nor B has overflowed
- **bit 10**: SAB: SA || SB Combined Accumulator Status bit
  - 1 = Accumulator A or B is saturated or has been saturated since this bit was last cleared
  - 0 = Neither Accumulator A nor B is saturated
- **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/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

### Notes:
1. The IPL<2:0> bits are concatenated with the IPL3 bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL, if IPL3 = 1. User interrupts are disabled when IPL3 = 1.
2. The IPL<2:0> Status bits are read only when NSTDIS bit (INTCON1<15>) = 1. Refer to the family reference manual of the specific device family to see the associated interrupt register.
3. A data write to SR can modify the SA or SB bits by either a data write to SA and SB or by clearing the SAB bit. To avoid a possible SAVSB bit write race-condition, the SA and SB bits should not be modified using bit operations.
Register 2-3: SR: CPU STATUS Register (dsPIC33E Devices) (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-5</td>
<td>IPL&lt;2:0&gt;: CPU Interrupt Priority Level Status bits&lt;sup&gt;(1,2)&lt;/sup&gt;</td>
<td>111 = CPU Interrupt Priority Level is 7 (15). User interrupts disabled&lt;br&gt;110 = CPU Interrupt Priority Level is 6 (14)&lt;br&gt;101 = CPU Interrupt Priority Level is 5 (13)&lt;br&gt;100 = CPU Interrupt Priority Level is 4 (12)&lt;br&gt;011 = CPU Interrupt Priority Level is 3 (11)&lt;br&gt;010 = CPU Interrupt Priority Level is 2 (10)&lt;br&gt;001 = CPU Interrupt Priority Level is 1 (9)&lt;br&gt;000 = CPU Interrupt Priority Level is 0 (8)</td>
</tr>
<tr>
<td>4</td>
<td>RA: REPEAT Loop Active bit</td>
<td>1 = REPEAT loop in progress&lt;br&gt;0 = REPEAT loop not in progress</td>
</tr>
<tr>
<td>3</td>
<td>N: MCU ALU Negative bit</td>
<td>1 = Result was negative&lt;br&gt;0 = Result was non-negative (zero or positive)</td>
</tr>
<tr>
<td>2</td>
<td>OV: MCU ALU Overflow bit</td>
<td>1 = Overflow occurred for signed arithmetic (in this arithmetic operation)&lt;br&gt;0 = No overflow occurred</td>
</tr>
<tr>
<td>1</td>
<td>Z: MCU ALU Zero bit</td>
<td>1 = The result of the operation was zero&lt;br&gt;0 = The result of the operation was not zero</td>
</tr>
<tr>
<td>0</td>
<td>C: MCU ALU Carry/Borrow bit</td>
<td>1 = A carry-out from the MSb of the result occurred&lt;br&gt;0 = No carry-out from the MSb of the result occurred</td>
</tr>
</tbody>
</table>

**Note 1:** The IPL<2:0> bits are concatenated with the IPL3 bit (CORCON<3>) to form the CPU Interrupt Priority Level. The value in parentheses indicates the IPL, if IPL3 = 1. User interrupts are disabled when IPL3 = 1.

**Note 2:** The IPL<2:0> Status bits are read only when NSTDIS bit (INTCON1<15>) = 1. Refer to the family reference manual of the specific device family to see the associated interrupt register.

**Note 3:** A data write to SR can modify the SA or SB bits by either a data write to SA and SB or by clearing the SAB bit. To avoid a possible SA/SB bit write race-condition, the SA and SB bits should not be modified using bit operations.
### Register 2-4: CORCON: Core Control Register (PIC24F and PIC24H Devices)

<table>
<thead>
<tr>
<th>bit 15</th>
<th>bit 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bit 7</th>
<th>bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>R/C-0</td>
</tr>
<tr>
<td>R/W-0</td>
<td>U-0</td>
</tr>
<tr>
<td>U-0</td>
<td>U-0</td>
</tr>
</tbody>
</table>

**Legend:**

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

**bit 15-4:** **Unimplemented:** Read as ‘0’

**bit 3:** **IPL3:** Interrupt Priority Level 3 Status bit\(^{(1,2)}\)
- 1 = CPU Interrupt Priority Level is 8 or greater (trap exception activated)
- 0 = CPU Interrupt Priority Level is 7 or less (no trap exception activated)

**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:** This bit may be read or cleared, but not set.

**2:** This bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU Interrupt Priority Level.
Register 2-5: CORCON: Core Control Register (PIC24E Devices)

<table>
<thead>
<tr>
<th>R/W-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
<th>U-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bit 15  
VAR: Variable Exception Processing Latency Control bit
1 = Variable (bounded deterministic) exception processing latency
0 = Fixed (fully deterministic) exception processing latency

bit 14-4 Unimplemented: Read as '0'

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

bit 2  
SFA: Stack Frame Active Status bit
1 = Stack frame is active. W14 and W15 address 0x0000 to 0xFFFF, regardless of DSRPAG and DSWPAG values.
0 = Stack frame is not active. W14 and W15 address of EDS or Base Data Space

bit 1-0 Unimplemented: Read as '0'

Note 1: This bit may be read or cleared, but not set.

2: The IPL3 bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU interrupt priority level.
# Section 2. Programmer’s Model

## Register 2-6: CORCON: Core Control Register (dsPIC30F and dsPIC33F Devices)

| Bit 15-13 | **Unimplemented**: Read as ‘0’ |
| Bit 12 | **US**: Unsigned or Signed Multiplier Mode Select bit |
| 1 = Unsigned mode enabled for DSP multiply operations |
| 0 = Signed mode enabled for DSP multiply operations |
| Bit 11 | **EDT**: Early DO Loop Termination Control bit\(^{(1)}\) |
| 1 = Terminate executing DO loop at end of current iteration |
| 0 = No effect |
| Bit 10-8 | **DL<2:0>**: DO Loop Nesting Level Status bits\(^{(2,3)}\) |
| 111 = DO looping is nested at 7 levels |
| 110 = DO looping is nested at 6 levels |
| 110 = DO looping is nested at 5 levels |
| 110 = DO looping is nested at 4 levels |
| 011 = DO looping is nested at 3 levels |
| 010 = DO looping is nested at 2 levels |
| 001 = DO looping is active, but not nested (just 1 level) |
| 000 = DO looping is not 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**: Interrupt Priority Level 3 Status bit\(^{(4,5)}\) |
| 1 = CPU Interrupt Priority Level is 8 or greater (trap exception activated) |
| 0 = CPU Interrupt Priority Level is 7 or less (no trap exception activated) |
| 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 |

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

Note 1: This bit will always read ‘0’.

2: DL<2:1> are read-only.

3: The first two levels of DO loop nesting are handled by hardware.

4: This bit may be read or cleared, but not set.

5: This bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU Interrupt Priority Level.
Register 2-6: **CORCON: Core Control Register (dsPIC30F and dsPIC33F Devices) (Continued)**

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 operations
       0 = Fractional mode enabled for DSP multiply operations

**Note 1:** This bit will always read ‘0’.

**2:** DL<2:1> are read-only.

**3:** The first two levels of do loop nesting are handled by hardware.

**4:** This bit may be read or cleared, but not set.

**5:** This bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU Interrupt Priority Level.
Register 2-7: CORCON: Core Control Register (dsPIC33E Devices)

<table>
<thead>
<tr>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-1</th>
<th>R/W-0</th>
<th>R/C-0</th>
<th>R-0</th>
<th>R/W-0</th>
<th>R/W-0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR</td>
<td>US&lt;1:0&gt;</td>
<td>EDT&lt;1&gt;</td>
<td>DL&lt;2:0&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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**
- VAR: Variable Exception Processing Latency Control bit
  - 1 = Variable (bounded deterministic) exception processing latency
  - 0 = Fixed (fully deterministic) exception processing latency

**bit 14**
- Unimplemented: Read as '0'

**bit 13-12**
- US<1:0>: DSP Multiply Unsigned/Signed Control bits
  - 11 = Reserved
  - 10 = DSP engine multiplies are mixed-sign
  - 01 = DSP engine multiplies are unsigned
  - 00 = DSP engine multiplies are signed

**bit 11**
- EDT: Early DO Loop Termination Control bit<sup>(1)</sup>
  - 1 = Terminate executing DO loop at end of current loop iteration
  - 0 = No effect

**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<sup>(2)</sup>
  - 1 = CPU interrupt priority level is greater than 7
  - 0 = CPU interrupt priority level is 7 or less

**Note:**
1. This bit always reads as ‘0’.
2. This bit may be read or cleared, but not set.
3. The IPL3 bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU interrupt priority level.
Register 2-7: CORCON: Core Control Register (dsPIC33E Devices) (Continued)

bit 2  SFA: Stack Frame Active Status bit
       1 = Stack frame is active. W14 and W15 address 0x0000 to 0xFFFF, regardless of DSRPAG and DSWPAG values.
       0 = Stack frame is not active. W14 and W15 address of EDS or Base 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
       0 = Fractional mode enabled for DSP multiply

Note 1: This bit always reads as ‘0’.
2: This bit may be read or cleared, but not set.
3: The IPL3 bit is concatenated with the IPL<2:0> bits (SR<7:5>) to form the CPU interrupt priority level.
Section 3. Instruction Set Overview

HIGHLIGHTS

This section of the manual contains the following major topics:

3.1 Introduction ................................................................................................................ .... 38
3.2 Instruction Set Overview .............................................................................................. 38
3.3 Instruction Set Summary Tables .................................................................................... 40
3.1 INTRODUCTION

The 16-bit MCU and DSC instruction set provides a broad suite of instructions that support traditional microcontroller applications, and a class of instructions that support math intensive applications. Since almost all of the functionality of the 8-bit PIC MCU instruction set has been maintained, this hybrid instruction set allows an easy 16-bit migration path for users already familiar with the PIC microcontroller.

3.2 INSTRUCTION SET OVERVIEW

Depending on the device family, the 16-bit MCU and DSC instruction set contains up to 84 instructions, which can be grouped into the functional categories shown in Table 3-1. Table 1-2 defines the symbols used in the instruction summary tables, Table 3-2 through Table 3-11. These tables define the syntax, description, storage and execution requirements for each instruction. Storage requirements are represented in 24-bit instruction words and execution requirements are represented in instruction cycles.

Table 3-1: Instruction Groups

<table>
<thead>
<tr>
<th>Functional Group</th>
<th>Summary Table</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Instructions</td>
<td>Table 3-2</td>
<td>40</td>
</tr>
<tr>
<td>Math Instructions</td>
<td>Table 3-3</td>
<td>41</td>
</tr>
<tr>
<td>Logic Instructions</td>
<td>Table 3-4</td>
<td>43</td>
</tr>
<tr>
<td>Rotate/Shift Instructions</td>
<td>Table 3-5</td>
<td>44</td>
</tr>
<tr>
<td>Bit Instructions</td>
<td>Table 3-6</td>
<td>45</td>
</tr>
<tr>
<td>Compare/Skip and Compare/Branch Instructions</td>
<td>Table 3-7</td>
<td>46</td>
</tr>
<tr>
<td>Program Flow Instructions</td>
<td>Table 3-8</td>
<td>47</td>
</tr>
<tr>
<td>Shadow/Stack Instructions</td>
<td>Table 3-9</td>
<td>49</td>
</tr>
<tr>
<td>Control Instructions</td>
<td>Table 3-10</td>
<td>49</td>
</tr>
<tr>
<td>DSP Instructions (1)</td>
<td>Table 3-11</td>
<td>50</td>
</tr>
</tbody>
</table>

Note 1: DSP instructions are only available in the dsPIC30F, dsPIC33F, and dsPIC33E device families.

Most instructions have several different Addressing modes and execution flows, which require different instruction variants. For instance, depending on the device family, there are up to six unique ADD instructions and each instruction variant has its own instruction encoding. Instruction format descriptions and specific instruction operation are provided in Section 5, “Instruction Descriptions”. Additionally, a composite alphabetized instruction set table is provided in Section 7, “Reference”.
3.2.1 Multi-Cycle Instructions

As the instruction summary tables show, most instructions execute in a single cycle, with the following exceptions:

**Note:** The **DO** and **DIVF** instructions are only available in the dsPIC30F, dsPIC33F, and dsPIC33E device families.

- **Instructions** **DO, MOV.D, POP.D, PUSH.D, TBLRDH, TBLRDL, TBLWTH and TBLWTL** require 2 cycles to execute
- **Instructions** **DIV.S, DIV.U and DIVF** are single-cycle instructions, which should be executed 18 consecutive times as the target of a **REPEAT** instruction
- **Instructions** that change the program counter also require 2 cycles to execute, with the extra cycle executed as a **NOP**. Compare-skip instructions, which skip over a 2-word instruction, require 3 instruction cycles to execute, with 2 cycles executed as a **NOP**. Compare-branch instructions (dsPIC33E/PIC24E devices only) require 5 instruction cycles to execute when the branch is taken.
- **The RETFIE, RETLW and RETURN** are a special case of an instruction that changes the program counter. These execute in 3 cycles, unless an exception is pending and then they execute in 2 cycles.

**Note 1:** Instructions which access program memory as data, using Program Space Visibility (PSV), will incur a one or two cycle delay for PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices, whereas using PSV in dsPIC33E and PIC24E devices incurs a 4-cycle delay based on Flash memory access time. However, regardless of which device is being used, when the target instruction of a **REPEAT** loop accesses program memory as data, only the first execution of the target instruction is subject to the delay. See the specific device family reference manual for details.

2: All instructions may incur an additional delay on some device families, depending on Flash memory access time. For example, PIC24E and dsPIC33E devices have a 3-cycle Flash memory access time. However, instruction pipelining increases the effective instruction execution throughput. Refer to Section 2. “CPU” of the specific device family reference manual for details on instruction timing.

3: All read and read-modify-write operations (including bit operations) on non-CPU Special Function Registers (e.g., I/O Port, peripheral control, or status registers; interrupt flags, etc.) in PIC24E and dsPIC33E devices require 2 instruction cycles to execute. However, all write operations on both CPU and non-CPU Special Function Registers, and all read and read-modify-write operations on CPU Special Function Registers require 1 instruction cycle.

3.2.2 Multi-Word Instructions

As defined by Table 3-2, almost all instructions consume one instruction word (24 bits), with the exception of the **CALL, DO and GOTO** instructions, which are Program Flow Instructions, listed in Table 3-8. These instructions require two words of memory because their opcodes embed large literal operands.
### 3.3 INSTRUCTION SET SUMMARY TABLES

#### Table 3-2: Move Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCH Wns,Wnd</td>
<td>Swap Wns and Wnd</td>
<td>1</td>
<td>1</td>
<td>243</td>
</tr>
<tr>
<td>MOV f {,WREG}(^{(1)})</td>
<td>Move f to destination</td>
<td>1</td>
<td>1</td>
<td>279</td>
</tr>
<tr>
<td>MOV WREG,f</td>
<td>Move WREG to f</td>
<td>1</td>
<td>1</td>
<td>280</td>
</tr>
<tr>
<td>MOV f,Wnd</td>
<td>Move f to Wnd</td>
<td>1</td>
<td>1(^{(4)})</td>
<td>281</td>
</tr>
<tr>
<td>MOV Wns,f</td>
<td>Move Wns to f</td>
<td>1</td>
<td>1</td>
<td>282</td>
</tr>
<tr>
<td>MOV.B #lit8,Wnd</td>
<td>Move 8-bit literal to Wnd</td>
<td>1</td>
<td>1</td>
<td>283</td>
</tr>
<tr>
<td>MOV #lit16,Wnd</td>
<td>Move 16-bit literal to Wnd</td>
<td>1</td>
<td>1</td>
<td>284</td>
</tr>
<tr>
<td>MOV [Ws+Slit10],Wnd</td>
<td>Move [Ws + signed 10-bit offset] to Wnd</td>
<td>1</td>
<td>1(^{(4)})</td>
<td>285</td>
</tr>
<tr>
<td>MOV Wns,[Wd+Slit10]</td>
<td>Move Wns to [Wd + signed 10-bit offset]</td>
<td>1</td>
<td>1</td>
<td>286</td>
</tr>
<tr>
<td>MOV Wso,Wdo</td>
<td>Move Wso to Wdo</td>
<td>1</td>
<td>1(^{(4)})</td>
<td>287</td>
</tr>
<tr>
<td>MOV.D Ws,Wnd</td>
<td>Move double Ws to Wnd:Wns + 1</td>
<td>1</td>
<td>2(^{(4)})</td>
<td>289</td>
</tr>
<tr>
<td>MOVPAG #lit10,DSRPAG(^{(2)})</td>
<td>Move 10-bit literal to DSRPAG</td>
<td>1</td>
<td>1</td>
<td>291</td>
</tr>
<tr>
<td>MOVPAG #lit9,DSWPAG(^{(2)})</td>
<td>Move 9-bit literal to DSWPAG</td>
<td>1</td>
<td>1</td>
<td>291</td>
</tr>
<tr>
<td>MOVPAG #lit8,TBLPAG(^{(2)})</td>
<td>Move 8-bit literal to TBLPAG</td>
<td>1</td>
<td>1</td>
<td>291</td>
</tr>
<tr>
<td>MOVPAG Wn, DSRPAG(^{(2)})</td>
<td>Move Wn to DSRPAG</td>
<td>1</td>
<td>1</td>
<td>292</td>
</tr>
<tr>
<td>MOVPAG Wn, DSWPAG(^{(2)})</td>
<td>Move Wn to DSWPAG</td>
<td>1</td>
<td>1</td>
<td>292</td>
</tr>
<tr>
<td>MOVPAG Wn, TBLPAC(^{(2)})</td>
<td>Move Wn to TBLPAC</td>
<td>1</td>
<td>1</td>
<td>292</td>
</tr>
<tr>
<td>SWAP Wn</td>
<td>Wn = byte or nibble swap Wn</td>
<td>1</td>
<td>1</td>
<td>426</td>
</tr>
<tr>
<td>TBLRDH [Ws],Wd</td>
<td>Read high program word to Wd</td>
<td>1</td>
<td>2(^{(3)})</td>
<td>427</td>
</tr>
<tr>
<td>TBLRDL [Ws],Wd</td>
<td>Read low program word to Wd</td>
<td>1</td>
<td>2(^{(3)})</td>
<td>429</td>
</tr>
<tr>
<td>TBLWTH Ws,[Wd]</td>
<td>Write Ws to high program word</td>
<td>1</td>
<td>2(^{(4)})</td>
<td>431</td>
</tr>
<tr>
<td>TBLWTL Ws,[Wd]</td>
<td>Write Ws to low program word</td>
<td>1</td>
<td>2(^{(4)})</td>
<td>433</td>
</tr>
</tbody>
</table>

**Note 1:** When the optional \{,WREG\} operand is specified, the destination of the instruction is WREG. When \{,WREG\} is not specified, the destination of the instruction is the file register f.

**2:** The MOVPAG instruction is only available in dsPIC33E and PIC24E devices.

**3:** In dsPIC33E and PIC24E devices, these instructions require 3 additional cycles – compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.

**4:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
### Table 3-3: Math Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD f {,WREG}</td>
<td>Destination = f + WREG</td>
<td>1</td>
<td>1(5)</td>
<td>99</td>
</tr>
<tr>
<td>ADD #lit10,Wn</td>
<td>Wn = lit10 + Wn</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>ADD Wb,#lit5,Wd</td>
<td>Wd = Wb + lit5</td>
<td>1</td>
<td>1</td>
<td>101</td>
</tr>
<tr>
<td>ADD Wb,Ws,Wd</td>
<td>Wd = Wb + Ws</td>
<td>1</td>
<td>1(5)</td>
<td>102</td>
</tr>
<tr>
<td>ADDC f {,WREG}</td>
<td>Destination = f + WREG + (C)</td>
<td>1</td>
<td>1(5)</td>
<td>106</td>
</tr>
<tr>
<td>ADDC #lit10,Wn</td>
<td>Wn = lit10 + Wn + (C)</td>
<td>1</td>
<td>1</td>
<td>107</td>
</tr>
<tr>
<td>ADDC Wb,#lit5,Wd</td>
<td>Wd = Wb + lit5 + (C)</td>
<td>1</td>
<td>1</td>
<td>108</td>
</tr>
<tr>
<td>ADDC Wb,Ws,Wd</td>
<td>Wd = Wb + Ws + (C)</td>
<td>1</td>
<td>1(5)</td>
<td>110</td>
</tr>
<tr>
<td>DAW.B Wn</td>
<td>Wn = decimal adjust Wn</td>
<td>1</td>
<td>1</td>
<td>216</td>
</tr>
<tr>
<td>DEC f {,WREG}</td>
<td>Destination = f - 1</td>
<td>1</td>
<td>1(5)</td>
<td>217</td>
</tr>
<tr>
<td>DEC Ws,Wd</td>
<td>Wd = Ws - 1</td>
<td>1</td>
<td>1(5)</td>
<td>218</td>
</tr>
<tr>
<td>DEC2 f {,WREG}</td>
<td>Destination = f - 2</td>
<td>1</td>
<td>1(5)</td>
<td>220</td>
</tr>
<tr>
<td>DEC2 Ws,Wd</td>
<td>Wd = Ws - 2</td>
<td>1</td>
<td>1(5)</td>
<td>221</td>
</tr>
<tr>
<td>DIV.S Wm, Wn</td>
<td>Signed 16/16-bit integer divide, Q → W0, R → W1</td>
<td>1</td>
<td>18(2)</td>
<td>224</td>
</tr>
<tr>
<td>DIV.SD Wm, Wn</td>
<td>Signed 32/16-bit integer divide, Q → W0, R → W1</td>
<td>1</td>
<td>18(2)</td>
<td>224</td>
</tr>
<tr>
<td>DIV.U Wm, Wn</td>
<td>Unsigned 16/16-bit integer divide, Q → W0, R → W1</td>
<td>1</td>
<td>18(2)</td>
<td>226</td>
</tr>
<tr>
<td>DIV.UD Wm, Wn</td>
<td>Unsigned 32/16-bit integer divide, Q → W0, R → W1</td>
<td>1</td>
<td>18(2)</td>
<td>226</td>
</tr>
<tr>
<td>DIVF Wm, Wn</td>
<td>Signed 16/16-bit fractional divide, Q → W0, R → W1</td>
<td>1</td>
<td>18(2)</td>
<td>228</td>
</tr>
<tr>
<td>INC f {,WREG}</td>
<td>Destination = f + 1</td>
<td>1</td>
<td>1(5)</td>
<td>254</td>
</tr>
<tr>
<td>INC Ws,Wd</td>
<td>Wd = Ws + 1</td>
<td>1</td>
<td>1(5)</td>
<td>255</td>
</tr>
<tr>
<td>INC2 f {,WREG}</td>
<td>Destination = f + 2</td>
<td>1</td>
<td>1(5)</td>
<td>257</td>
</tr>
<tr>
<td>INC2 Ws,Wd</td>
<td>Wd = Ws + 2</td>
<td>1</td>
<td>1(5)</td>
<td>258</td>
</tr>
<tr>
<td>MUL f</td>
<td>W3:W2 = f * WREG</td>
<td>1</td>
<td>1(5)</td>
<td>303</td>
</tr>
<tr>
<td>MUL.SS Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>305</td>
</tr>
<tr>
<td>MUL.SS Wb,Ws,Acc</td>
<td>Accumulator = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>307</td>
</tr>
<tr>
<td>MUL.SU Wb,#lit5,Wnd</td>
<td>(Wnd + 1, Wnd) = signed(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>308</td>
</tr>
<tr>
<td>MUL.SU Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = signed(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>310</td>
</tr>
<tr>
<td>MUL.SU Wb,Ws,Acc</td>
<td>Accumulator = signed(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>312</td>
</tr>
<tr>
<td>MUL.SU Wb,#lit5,Acc</td>
<td>Accumulator = signed(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>314</td>
</tr>
<tr>
<td>MUL.US Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = unsigned(Wb) * signed(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>315</td>
</tr>
<tr>
<td>MUL.US Wb,Ws,Acc</td>
<td>Accumulator = unsigned(Wb) * signed(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>317</td>
</tr>
<tr>
<td>MUL.UU Wb,#lit5,Wnd</td>
<td>(Wnd + 1, Wnd) = unsigned(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>319</td>
</tr>
<tr>
<td>MUL.UU Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = unsigned(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>320</td>
</tr>
<tr>
<td>MUL.UU Wb,Ws,Acc</td>
<td>Accumulator = unsigned(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>322</td>
</tr>
<tr>
<td>MUL.UU Wb,#lit5,Acc</td>
<td>Accumulator = unsigned(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>323</td>
</tr>
<tr>
<td>MULW.SS Wb,Ws,Wnd</td>
<td>Wnd = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1(5)</td>
<td>324</td>
</tr>
</tbody>
</table>

**Note 1:** When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

**Note 2:** The divide instructions must be preceded with a “REPEAT #17” instruction, such that they are executed 18 consecutive times.

**Note 3:** These instructions are only available in dsPIC33E and PIC24E devices.

**Note 4:** These instructions are only available in dsPIC33E devices.

**Note 5:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
### Table 3-3: Math Instructions (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULW.SU Wb,Ws,Wnd</td>
<td>( Wnd = \text{signed}(Wb) \times \text{unsigned}(Ws) )</td>
<td>1</td>
<td>( f )</td>
<td>326</td>
</tr>
<tr>
<td>MULW.SU Wb,#lit5,Wnd</td>
<td>( Wnd = \text{signed}(Wb) \times \text{unsigned}(\text{lit5}) )</td>
<td>1</td>
<td>1</td>
<td>328</td>
</tr>
<tr>
<td>MULW.US Wb,Ws,Wnd</td>
<td>( Wnd = \text{unsigned}(Wb) \times \text{signed}(Ws) )</td>
<td>1</td>
<td>( f )</td>
<td>329</td>
</tr>
<tr>
<td>MULW.UU Wb,Ws,Wnd</td>
<td>( Wnd = \text{unsigned}(Wb) \times \text{unsigned}(Ws) )</td>
<td>1</td>
<td>( f )</td>
<td>331</td>
</tr>
<tr>
<td>MULW.UU Wb,#lit5,Wnd</td>
<td>( Wnd = \text{unsigned}(Wb) \times \text{unsigned}(\text{lit5}) )</td>
<td>1</td>
<td>1</td>
<td>332</td>
</tr>
<tr>
<td>SE Ws,Wnd</td>
<td>( Wnd = \text{signed-extended} Ws )</td>
<td>1</td>
<td>( f )</td>
<td>393</td>
</tr>
<tr>
<td>SUB f {,WREG}</td>
<td>Destination = ( f - \text{WREG} )</td>
<td>1</td>
<td>( f )</td>
<td>405</td>
</tr>
<tr>
<td>SUB #lit10,Wn</td>
<td>( Wn = Wn - \text{lit10} )</td>
<td>1</td>
<td>1</td>
<td>406</td>
</tr>
<tr>
<td>SUB Wb,#lit5,Wd</td>
<td>( Wd = Wb - \text{lit5} )</td>
<td>1</td>
<td>1</td>
<td>407</td>
</tr>
<tr>
<td>SUB Wb,Ws,Wd</td>
<td>( Wd = Wb - Ws )</td>
<td>1</td>
<td>( f )</td>
<td>408</td>
</tr>
<tr>
<td>SUBB f {,WREG}</td>
<td>Destination = ( f - \text{WREG} - (\bar{C}) )</td>
<td>1</td>
<td>( f )</td>
<td>411</td>
</tr>
<tr>
<td>SUBB #lit10,Wn</td>
<td>( Wn = Wn - \text{lit10} - (\bar{C}) )</td>
<td>1</td>
<td>1</td>
<td>412</td>
</tr>
<tr>
<td>SUBB Wb,#lit5,Wd</td>
<td>( Wd = Wb - \text{lit5} - (\bar{C}) )</td>
<td>1</td>
<td>1</td>
<td>413</td>
</tr>
<tr>
<td>SUBB Wb,Ws,Wd</td>
<td>( Wd = Wb - Ws - (\bar{C}) )</td>
<td>1</td>
<td>( f )</td>
<td>415</td>
</tr>
<tr>
<td>SUBBR f {,WREG}</td>
<td>Destination = ( \text{WREG} - f - (\bar{C}) )</td>
<td>1</td>
<td>( f )</td>
<td>417</td>
</tr>
<tr>
<td>SUBBR Wb,#lit5,Wd</td>
<td>( Wd = \text{lit5} - Wb - (\bar{C}) )</td>
<td>1</td>
<td>1</td>
<td>418</td>
</tr>
<tr>
<td>SUBBR Wb,Ws,Wd</td>
<td>( Wd = Ws - Wb - (\bar{C}) )</td>
<td>1</td>
<td>( f )</td>
<td>420</td>
</tr>
<tr>
<td>SUBR f {,WREG}</td>
<td>Destination = ( \text{WREG} - f )</td>
<td>1</td>
<td>( f )</td>
<td>422</td>
</tr>
<tr>
<td>SUBR Wb,#lit5,Wd</td>
<td>( Wd = \text{lit5} - Wb )</td>
<td>1</td>
<td>1</td>
<td>423</td>
</tr>
<tr>
<td>SUBR Wb,Ws,Wd</td>
<td>( Wd = Ws - Wb )</td>
<td>1</td>
<td>( f )</td>
<td>424</td>
</tr>
<tr>
<td>ZE Ws,Wnd</td>
<td>( Wnd = \text{zero-extended} Ws )</td>
<td>1</td>
<td>( f )</td>
<td>442</td>
</tr>
</tbody>
</table>

**Note 1:** When the optional \{,WREG\} operand is specified, the destination of the instruction is WREG. When \{,WREG\} is not specified, the destination of the instruction is the file register f.

**Note 2:** The divide instructions must be preceded with a “REPEAT #17” instruction, such that they are executed 18 consecutive times.

**Note 3:** These instructions are only available in dsPIC33E and PIC24E devices.

**Note 4:** These instructions are only available in dsPIC33E devices.

**Note 5:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
### Table 3-4: Logic Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND f {,WREG}</td>
<td>Destination = f .AND. WREG</td>
<td>1</td>
<td>1[2]</td>
<td>112</td>
</tr>
<tr>
<td>AND #lit10,Wn</td>
<td>Wn = lit10 .AND. Wn</td>
<td>1</td>
<td>1</td>
<td>113</td>
</tr>
<tr>
<td>AND Wb,#lit5,Wd</td>
<td>Wd = Wb .AND. lit5</td>
<td>1</td>
<td>1</td>
<td>114</td>
</tr>
<tr>
<td>AND Wb,Ws,Wd</td>
<td>Wd = Wb .AND. Ws</td>
<td>1</td>
<td>1[2]</td>
<td>115</td>
</tr>
<tr>
<td>CLR f</td>
<td>f = 0x0000</td>
<td>1</td>
<td>1</td>
<td>184</td>
</tr>
<tr>
<td>CLR WREG</td>
<td>WREG = 0x0000</td>
<td>1</td>
<td>1</td>
<td>184</td>
</tr>
<tr>
<td>CLR Wd</td>
<td>Wd = 0x0000</td>
<td>1</td>
<td>1</td>
<td>185</td>
</tr>
<tr>
<td>COM f {,WREG}</td>
<td>Destination = ~f</td>
<td>1</td>
<td>1[2]</td>
<td>189</td>
</tr>
<tr>
<td>COM Ws,Wd</td>
<td>Wd = Ws</td>
<td>1</td>
<td>1</td>
<td>190</td>
</tr>
<tr>
<td>IOR f {,WREG}</td>
<td>Destination = f .IOR. WREG</td>
<td>1</td>
<td>1[2]</td>
<td>260</td>
</tr>
<tr>
<td>IOR #lit10,Wn</td>
<td>Wn = lit10 .IOR. Wn</td>
<td>1</td>
<td>1</td>
<td>261</td>
</tr>
<tr>
<td>IOR Wb,#lit5,Wd</td>
<td>Wd = Wb .IOR. lit5</td>
<td>1</td>
<td>1</td>
<td>262</td>
</tr>
<tr>
<td>IOR Wb,Ws,Wd</td>
<td>Wd = Wb .IOR. Ws</td>
<td>1</td>
<td>1[2]</td>
<td>263</td>
</tr>
<tr>
<td>NEG f {,WREG}</td>
<td>Destination = f + 1</td>
<td>1</td>
<td>1[2]</td>
<td>333</td>
</tr>
<tr>
<td>NEG Ws,Wd</td>
<td>Wd = Ws + 1</td>
<td>1</td>
<td>1</td>
<td>333</td>
</tr>
<tr>
<td>SETM f</td>
<td>f = 0xFFFF</td>
<td>1</td>
<td>1</td>
<td>395</td>
</tr>
<tr>
<td>SETM WREG</td>
<td>WREG = 0xFFFF</td>
<td>1</td>
<td>1</td>
<td>395</td>
</tr>
<tr>
<td>SETM Wd</td>
<td>Wd = 0xFFFF</td>
<td>1</td>
<td>1</td>
<td>396</td>
</tr>
<tr>
<td>XOR f {,WREG}</td>
<td>Destination = f .XOR. WREG</td>
<td>1</td>
<td>1[2]</td>
<td>437</td>
</tr>
<tr>
<td>XOR #lit10,Wn</td>
<td>Wn = lit10 .XOR. Wn</td>
<td>1</td>
<td>1</td>
<td>438</td>
</tr>
<tr>
<td>XOR Wb,#lit5,Wd</td>
<td>Wd = Wb .XOR. lit5</td>
<td>1</td>
<td>1</td>
<td>439</td>
</tr>
<tr>
<td>XOR Wb,Ws,Wd</td>
<td>Wd = Wb .XOR. Ws</td>
<td>1</td>
<td>1[2]</td>
<td>440</td>
</tr>
</tbody>
</table>

**Note 1:** When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

**Note 2:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
### Table 3-5: Rotate/Shift Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR f {,WREG}</td>
<td>Destination = arithmetic right shift f, LSb → C</td>
<td>1</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td>ASR Ws, Wd</td>
<td>Wd = arithmetic right shift Ws, LSb → C</td>
<td>1</td>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td>ASR Wb, #lit4, Wnd</td>
<td>Wnd = arithmetic right shift Wb by lit4, LSb → C</td>
<td>1</td>
<td>1</td>
<td>121</td>
</tr>
<tr>
<td>ASR Wb, Wns, Wnd</td>
<td>Wnd = arithmetic right shift Wb by Wns, LSb → C</td>
<td>1</td>
<td>1</td>
<td>122</td>
</tr>
<tr>
<td>LSR f {,WREG}</td>
<td>Destination = logical right shift f, LSb → C</td>
<td>1</td>
<td>1</td>
<td>269</td>
</tr>
<tr>
<td>LSR Ws, Wd</td>
<td>Wd = logical right shift Ws, LSb → C</td>
<td>1</td>
<td>1</td>
<td>271</td>
</tr>
<tr>
<td>LSR Wb, #lit4, Wnd</td>
<td>Wnd = logical right shift Wb by lit4, LSb → C</td>
<td>1</td>
<td>1</td>
<td>273</td>
</tr>
<tr>
<td>LSR Wb, Wns, Wnd</td>
<td>Wnd = logical right shift Wb by Wns, LSb → C</td>
<td>1</td>
<td>1</td>
<td>274</td>
</tr>
<tr>
<td>RLC f {,WREG}</td>
<td>Destination = rotate left through Carry f</td>
<td>1</td>
<td>1</td>
<td>373</td>
</tr>
<tr>
<td>RLC Ws, Wd</td>
<td>Wd = rotate left through Carry Ws</td>
<td>1</td>
<td>1</td>
<td>375</td>
</tr>
<tr>
<td>RLNC f {,WREG}</td>
<td>Destination = rotate left (no Carry) f</td>
<td>1</td>
<td>1</td>
<td>377</td>
</tr>
<tr>
<td>RLNC Ws, Wd</td>
<td>Wd = rotate left (no Carry) Ws</td>
<td>1</td>
<td>1</td>
<td>379</td>
</tr>
<tr>
<td>RRC f {,WREG}</td>
<td>Destination = rotate right through Carry f</td>
<td>1</td>
<td>1</td>
<td>381</td>
</tr>
<tr>
<td>RRC Ws, Wd</td>
<td>Wd = rotate right through Carry Ws</td>
<td>1</td>
<td>1</td>
<td>383</td>
</tr>
<tr>
<td>RRNC f {,WREG}</td>
<td>Destination = rotate right (no Carry) f</td>
<td>1</td>
<td>1</td>
<td>385</td>
</tr>
<tr>
<td>RRNC Ws, Wd</td>
<td>Wd = rotate right (no Carry) Ws</td>
<td>1</td>
<td>1</td>
<td>387</td>
</tr>
<tr>
<td>SL f {,WREG}</td>
<td>Destination = left shift f, MSb → C</td>
<td>1</td>
<td>1</td>
<td>399</td>
</tr>
<tr>
<td>SL Ws, Wd</td>
<td>Wd = left shift Ws, MSb → C</td>
<td>1</td>
<td>1</td>
<td>401</td>
</tr>
<tr>
<td>SL Wb, #lit4, Wnd</td>
<td>Wnd = left shift Wb by lit4, MSb → C</td>
<td>1</td>
<td>1</td>
<td>403</td>
</tr>
<tr>
<td>SL Wb, Wns, Wnd</td>
<td>Wnd = left shift Wb by Wns, MSb → C</td>
<td>1</td>
<td>1</td>
<td>404</td>
</tr>
</tbody>
</table>

**Note 1:** When the optional {,WREG} operand is specified, the destination of the instruction is WREG. When {,WREG} is not specified, the destination of the instruction is the file register f.

**Note 2:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
### Section 3. Instruction Set Overview

#### Table 3-6: Bit Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCLR f,#bit4</td>
<td>Bit clear f</td>
<td>1</td>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td>BCLR Ws,#bit4</td>
<td>Bit clear Ws</td>
<td>1</td>
<td>1</td>
<td>124</td>
</tr>
<tr>
<td>BSET f,#bit4</td>
<td>Bit set f</td>
<td>1</td>
<td>1</td>
<td>152</td>
</tr>
<tr>
<td>BSET Ws,#bit4</td>
<td>Bit set Ws</td>
<td>1</td>
<td>1</td>
<td>153</td>
</tr>
<tr>
<td>BSW.C Ws,Wb</td>
<td>Write C bit to Ws&lt;Wb&gt;</td>
<td>1</td>
<td>1</td>
<td>155</td>
</tr>
<tr>
<td>BSW.Z Ws,Wb</td>
<td>Write Z bit to Ws&lt;Wb&gt;</td>
<td>1</td>
<td>1</td>
<td>155</td>
</tr>
<tr>
<td>BTG f,#bit4</td>
<td>Bit toggle f</td>
<td>1</td>
<td>1</td>
<td>157</td>
</tr>
<tr>
<td>BTG Ws,#bit4</td>
<td>Bit toggle Ws</td>
<td>1</td>
<td>1</td>
<td>158</td>
</tr>
<tr>
<td>BTST f,#bit4</td>
<td>Bit test f to Z</td>
<td>1</td>
<td>1</td>
<td>168</td>
</tr>
<tr>
<td>BTST.Ws,#bit4</td>
<td>Bit test Ws to C</td>
<td>1</td>
<td>1</td>
<td>169</td>
</tr>
<tr>
<td>BTST.Ws,Wb</td>
<td>Bit test Ws&lt;Wb&gt; to C</td>
<td>1</td>
<td>1</td>
<td>171</td>
</tr>
<tr>
<td>BTSTS f,#bit4</td>
<td>Bit test f to Z, then set f</td>
<td>1</td>
<td>1</td>
<td>173</td>
</tr>
<tr>
<td>BTSTS.Ws,#bit4</td>
<td>Bit test Ws to C then set Ws</td>
<td>1</td>
<td>1</td>
<td>175</td>
</tr>
<tr>
<td>FBCL Ws,Wnde</td>
<td>Find bit change from left (MSb) side</td>
<td>1</td>
<td>1</td>
<td>244</td>
</tr>
<tr>
<td>FF1L Ws,Wnde</td>
<td>Find first one from left (MSb) side</td>
<td>1</td>
<td>1</td>
<td>246</td>
</tr>
<tr>
<td>FF1R Ws,Wnde</td>
<td>Find first one from right (LSb) side</td>
<td>1</td>
<td>1</td>
<td>248</td>
</tr>
</tbody>
</table>

**Note 1:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
Table 3-7: Compare/Skip and Compare/Branch Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles$^{(1)}$</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTSC f,#bit4</td>
<td>Bit test f, skip if clear</td>
<td>1</td>
<td>1 (2 or 3)$^5$</td>
<td>160</td>
</tr>
<tr>
<td>BTSC Ws,#bit4</td>
<td>Bit test Ws, skip if clear</td>
<td>1</td>
<td>1 (2 or 3)$^5$</td>
<td>162</td>
</tr>
<tr>
<td>BTSS f,#bit4</td>
<td>Bit test f, skip if set</td>
<td>1</td>
<td>1 (2 or 3)$^5$</td>
<td>164</td>
</tr>
<tr>
<td>BTSS Ws,#bit4</td>
<td>Bit test Ws, skip if set</td>
<td>1</td>
<td>1 (2 or 3)$^5$</td>
<td>166</td>
</tr>
<tr>
<td>CP f</td>
<td>Compare (f – WREG)</td>
<td>1</td>
<td>1$^5$</td>
<td>191</td>
</tr>
<tr>
<td>CP Wb,#lit5</td>
<td>Compare (Wb – lit5)</td>
<td>1</td>
<td>1</td>
<td>192</td>
</tr>
<tr>
<td>CP Wb,#lit8</td>
<td>Compare (Wb – lit8)</td>
<td>1</td>
<td>1</td>
<td>193</td>
</tr>
<tr>
<td>CP Wb,Ws</td>
<td>Compare (Wb – Ws)</td>
<td>1</td>
<td>1$^5$</td>
<td>194</td>
</tr>
<tr>
<td>CP0 f</td>
<td>Compare (f – 0x0000)</td>
<td>1</td>
<td>1$^5$</td>
<td>196</td>
</tr>
<tr>
<td>CP0 Ws</td>
<td>Compare (Ws – 0x0000)</td>
<td>1</td>
<td>1$^5$</td>
<td>197</td>
</tr>
<tr>
<td>CPB f</td>
<td>Compare with Borrow (f – WREG – $\overline{C}$)</td>
<td>1</td>
<td>1$^5$</td>
<td>198</td>
</tr>
<tr>
<td>CPB Wb,#lit5</td>
<td>Compare with Borrow (Wb – lit5 – $\overline{C}$)</td>
<td>1</td>
<td>1</td>
<td>199</td>
</tr>
<tr>
<td>CPB Wb,#lit8</td>
<td>Compare with Borrow (Wb – lit8 – $\overline{C}$)</td>
<td>1</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>CPB Wb,Ws</td>
<td>Compare with Borrow (Wb – Ws – $\overline{C}$)</td>
<td>1</td>
<td>1$^5$</td>
<td>201</td>
</tr>
<tr>
<td>CPBEQ Wb,Wn,Expr</td>
<td>Compare Wb with Wn, branch if =</td>
<td>1</td>
<td>1 (5)$^4$</td>
<td>203</td>
</tr>
<tr>
<td>CPBGT Wb,Wn,Expr</td>
<td>Signed compare Wb with Wn, branch if &gt;</td>
<td>1</td>
<td>1 (5)$^4$</td>
<td>204</td>
</tr>
<tr>
<td>CPBLT Wb,Wn,Expr</td>
<td>Signed compare Wb with Wn, branch if &lt;</td>
<td>1</td>
<td>1 (5)$^4$</td>
<td>205</td>
</tr>
<tr>
<td>CPBNE Wb,Wn,Expr</td>
<td>Compare Wb with Wn, branch if $\neq$</td>
<td>1</td>
<td>1 (5)$^4$</td>
<td>204</td>
</tr>
<tr>
<td>CPSGT Wb, Wn</td>
<td>Signed compare (Wb – Wn), skip if &gt;</td>
<td>1</td>
<td>1 (2 or 3)</td>
<td>207</td>
</tr>
<tr>
<td>CPSLT Wb, Wn</td>
<td>Signed compare (Wb – Wn), skip if &lt;</td>
<td>1</td>
<td>1 (2 or 3)</td>
<td>211</td>
</tr>
<tr>
<td>CPSNE Wb, Wn</td>
<td>Compare (Wb – Wn), skip if $\neq$</td>
<td>1</td>
<td>1 (2 or 3)</td>
<td>213</td>
</tr>
</tbody>
</table>

Note 1: Conditional skip instructions execute in 1 cycle if the skip is not taken, 2 cycles if the skip is taken over a one-word instruction and 3 cycles if the skip is taken over a two-word instruction.

2: This instruction is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.

3: This instruction is only available in dsPIC33E and PIC24E devices.

4: Compare-branch instructions in dsPIC33E/PIC24E devices execute in 1 cycle if the branch is not taken and 5 cycles if the branch is taken.

5: In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
### Table 3-8: Program Flow Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRA Expr</td>
<td>Branch unconditionally</td>
<td>1</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>126</td>
</tr>
<tr>
<td>BRA Wn</td>
<td>Computed branch</td>
<td>1</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>128</td>
</tr>
<tr>
<td>BRA C,Expr</td>
<td>Branch if Carry (no Borrow)</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>130</td>
</tr>
<tr>
<td>BRA GE,Expr</td>
<td>Branch if greater than or equal</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>132</td>
</tr>
<tr>
<td>BRA GEU,Expr</td>
<td>Branch if unsigned greater than or equal</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>134</td>
</tr>
<tr>
<td>BRA GT,Expr</td>
<td>Branch if greater than</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>135</td>
</tr>
<tr>
<td>BRA GTU,Expr</td>
<td>Branch if unsigned greater than</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>136</td>
</tr>
<tr>
<td>BRA LE,Expr</td>
<td>Branch if less than or equal</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>137</td>
</tr>
<tr>
<td>BRA LEU,Expr</td>
<td>Branch if unsigned less than or equal</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>138</td>
</tr>
<tr>
<td>BRA LT,Expr</td>
<td>Branch if less than</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>139</td>
</tr>
<tr>
<td>BRA LTU,Expr</td>
<td>Branch if unsigned less than</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>140</td>
</tr>
<tr>
<td>BRA N,Expr</td>
<td>Branch if Negative</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>141</td>
</tr>
<tr>
<td>BRA NC,Expr</td>
<td>Branch if not Carry (Borrow)</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>142</td>
</tr>
<tr>
<td>BRA NN,Expr</td>
<td>Branch if not Negative</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>143</td>
</tr>
<tr>
<td>BRA NOV,Expr</td>
<td>Branch if not Overflow</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>144</td>
</tr>
<tr>
<td>BRA NZ,Expr</td>
<td>Branch if not Overflow</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>145</td>
</tr>
<tr>
<td>BRA OA,Expr</td>
<td>Branch if Accumulator A Overflow</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>146</td>
</tr>
<tr>
<td>BRA OB,Expr</td>
<td>Branch if Accumulator B Overflow</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>147</td>
</tr>
<tr>
<td>BRA OV,Expr</td>
<td>Branch if Overflow</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>148</td>
</tr>
<tr>
<td>BRA SA,Expr</td>
<td>Branch if Accumulator A Saturate</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>149</td>
</tr>
<tr>
<td>BRA SB,Expr</td>
<td>Branch if Accumulator B Saturate</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>150</td>
</tr>
<tr>
<td>BRA Z,Expr</td>
<td>Branch if Zero</td>
<td>1</td>
<td>1&lt;sup&gt;(2)&lt;/sup&gt;</td>
<td>151</td>
</tr>
<tr>
<td>CALL Expr</td>
<td>Call subroutine</td>
<td>2</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>177</td>
</tr>
<tr>
<td>CALL Wn</td>
<td>Call indirect subroutine</td>
<td>1</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>180</td>
</tr>
<tr>
<td>CALL.L Wn&lt;sup&gt;(4)&lt;/sup&gt;</td>
<td>Call indirect subroutine (long address)</td>
<td>1</td>
<td>4</td>
<td>183</td>
</tr>
<tr>
<td>DO #lit14,Expr&lt;sup&gt;(6)&lt;/sup&gt;</td>
<td>Do code through PC + Expr, (lit14 + 1) times</td>
<td>2</td>
<td>2</td>
<td>230</td>
</tr>
<tr>
<td>DO #lit15,Expr&lt;sup&gt;(7)&lt;/sup&gt;</td>
<td>Do code through PC + Expr, (lit15 + 1) times</td>
<td>2</td>
<td>2</td>
<td>233</td>
</tr>
<tr>
<td>DO Wn,Expr&lt;sup&gt;(3)&lt;/sup&gt;</td>
<td>Do code through PC + Expr, (Wn + 1) times</td>
<td>2</td>
<td>2</td>
<td>235</td>
</tr>
<tr>
<td>GOTO Expr</td>
<td>Go to address</td>
<td>2</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>250</td>
</tr>
<tr>
<td>GOTO Wn</td>
<td>Go to address indirectly</td>
<td>1</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>251</td>
</tr>
<tr>
<td>GOTO.L Wn&lt;sup&gt;(4)&lt;/sup&gt;</td>
<td>Go to indirect (long address)</td>
<td>1</td>
<td>4</td>
<td>253</td>
</tr>
<tr>
<td>RCALL Expr</td>
<td>Relative call</td>
<td>1</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>347</td>
</tr>
<tr>
<td>RCALL Wn</td>
<td>Computed call</td>
<td>1</td>
<td>2&lt;sup&gt;(8)&lt;/sup&gt;</td>
<td>351</td>
</tr>
<tr>
<td>REPEAT #lit14&lt;sup&gt;(5)&lt;/sup&gt;</td>
<td>Repeat next instruction (lit14 + 1) times</td>
<td>1</td>
<td>1</td>
<td>355</td>
</tr>
</tbody>
</table>

**Note 1:** Conditional branch instructions execute in 1 cycle if the branch is not taken, or 2 cycles if the branch is taken.

2: RETURN instructions execute in 3 cycles, but if an exception is pending, they execute in 2 cycles.

3: This instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

4: This instruction is only available in dsPIC33E and PIC24E devices.

5: This instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

6: This instruction is only available in dsPIC30F and dsPIC33F devices.

7: This instruction is only available in dsPIC33E devices.

8: In dsPIC33E and PIC24E devices, these instructions require 2 additional cycles (4 cycles overall) when the branch is taken.

9: In dsPIC33E and PIC24E devices, these instructions require 3 additional cycles.
### Table 3-8: Program Flow Instructions (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPEAT #lit15(^{(4)})</td>
<td>Repeat next instruction (lit15 + 1) times</td>
<td>1</td>
<td>1</td>
<td>357</td>
</tr>
<tr>
<td>REPEAT Wn</td>
<td>Repeat next instruction (Wn + 1) times</td>
<td>1</td>
<td>1</td>
<td>359</td>
</tr>
<tr>
<td>RETFIE</td>
<td>Return from interrupt enable</td>
<td>1</td>
<td>3 (2)(^{(2,3)})</td>
<td>365</td>
</tr>
<tr>
<td>RETLW #lit10,Wn</td>
<td>Return with lit10 in Wn</td>
<td>1</td>
<td>3 (2)(^{(2,3)})</td>
<td>367</td>
</tr>
<tr>
<td>RETURN</td>
<td>Return from subroutine</td>
<td>1</td>
<td>3 (2)(^{(2,3)})</td>
<td>371</td>
</tr>
</tbody>
</table>

**Note 1:** Conditional branch instructions execute in 1 cycle if the branch is not taken, or 2 cycles if the branch is taken.

2: **RETURN** instructions execute in 3 cycles, but if an exception is pending, they execute in 2 cycles.

3: This instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

4: This instruction is only available in dsPIC33E and PIC24E devices.

5: This instruction is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.

6: This instruction is only available in dsPIC30F and dsPIC33F devices.

7: This instruction is only available in dsPIC33E devices.

8: In dsPIC33E and PIC24E devices, these instructions require 2 additional cycles (4 cycles overall) when the branch is taken.

9: In dsPIC33E and PIC24E devices, these instructions require 3 additional cycles.
### Table 3-9: Shadow/Stack Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNK #lit14</td>
<td>Link Frame Pointer</td>
<td>1</td>
<td>1</td>
<td>267</td>
</tr>
<tr>
<td>POP f</td>
<td>POP TOS to f</td>
<td>1</td>
<td>1</td>
<td>337</td>
</tr>
<tr>
<td>POP Wdo</td>
<td>POP TOS to Wdo</td>
<td>1</td>
<td>1</td>
<td>338</td>
</tr>
<tr>
<td>POP.D Wnd</td>
<td>Double POP from TOS to Wnd:Wnd + 1</td>
<td>1</td>
<td>2</td>
<td>339</td>
</tr>
<tr>
<td>POP.S</td>
<td>POP shadow registers</td>
<td>1</td>
<td>1</td>
<td>340</td>
</tr>
<tr>
<td>PUSH f</td>
<td>PUSH f to TOS</td>
<td>1</td>
<td>1($^{(f)}$)</td>
<td>341</td>
</tr>
<tr>
<td>PUSH Wso</td>
<td>PUSH Wso to TOS</td>
<td>1</td>
<td>1($^{(f)}$)</td>
<td>342</td>
</tr>
<tr>
<td>PUSH.D Wns</td>
<td>PUSH double Wns:Wns + 1 to TOS</td>
<td>1</td>
<td>2</td>
<td>343</td>
</tr>
<tr>
<td>PUSH.S</td>
<td>PUSH shadow registers</td>
<td>1</td>
<td>1</td>
<td>345</td>
</tr>
<tr>
<td>ULNK</td>
<td>Unlink Frame Pointer</td>
<td>1</td>
<td>1</td>
<td>435</td>
</tr>
</tbody>
</table>

**Note 1:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.

### Table 3-10: Control Instructions

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRWDT</td>
<td>Clear Watchdog Timer</td>
<td>1</td>
<td>1</td>
<td>188</td>
</tr>
<tr>
<td>DISI #lit14</td>
<td>Disable interrupts for (lit14 + 1) instruction cycles</td>
<td>1</td>
<td>1</td>
<td>223</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>1</td>
<td>1</td>
<td>336</td>
</tr>
<tr>
<td>NOPR</td>
<td>No operation</td>
<td>1</td>
<td>1</td>
<td>336</td>
</tr>
<tr>
<td>FWRSAV #lit1</td>
<td>Enter Power-saving mode lit1</td>
<td>1</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>RESET</td>
<td>Software device Reset</td>
<td>1</td>
<td>1</td>
<td>363</td>
</tr>
</tbody>
</table>
Table 3-11: DSP Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices)

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Acc</td>
<td>Add accumulators</td>
<td>1</td>
<td>1</td>
<td>103</td>
</tr>
<tr>
<td>ADD Wso,#Slit4,Acc</td>
<td>16-bit signed add to Acc</td>
<td>1</td>
<td>1</td>
<td>104</td>
</tr>
<tr>
<td>CLR Acc,[Wx],Wxd,[Wy],Wyd,AWB</td>
<td>Clear Acc</td>
<td>1</td>
<td>1</td>
<td>186</td>
</tr>
<tr>
<td>ED Wm*Wm,Acc,[Wx],[Wy],Wxd</td>
<td>Euclidean distance (no accumulate)</td>
<td>1</td>
<td>1</td>
<td>239</td>
</tr>
<tr>
<td>EDAC Wm*Wm,Acc,[Wx],[Wy],Wxd</td>
<td>Euclidean distance</td>
<td>1</td>
<td>1</td>
<td>241</td>
</tr>
<tr>
<td>LAC Wso,#Slit4,Acc</td>
<td>Load Acc</td>
<td>1</td>
<td>1</td>
<td>265</td>
</tr>
<tr>
<td>MAC Wm*Wn,Acc,[Wx],Wxd,[Wy],Wyd,AWB</td>
<td>Multiply and accumulate</td>
<td>1</td>
<td>1</td>
<td>275</td>
</tr>
<tr>
<td>MAC Wm*Wm,Acc,[Wx],Wxd,[Wy],Wyd</td>
<td>Square and accumulate</td>
<td>1</td>
<td>1</td>
<td>277</td>
</tr>
<tr>
<td>MOVSA C Acc,[Wx],Wxd,[Wy],Wyd,AWB</td>
<td>Move Wx to Wxd and Wy to Wyd</td>
<td>1</td>
<td>1</td>
<td>293</td>
</tr>
<tr>
<td>MPY Wm*Wn,Acc,[Wx],Wxd,[Wy],Wyd</td>
<td>Multiply Wn by Wm to Wyd</td>
<td>1</td>
<td>1</td>
<td>295</td>
</tr>
<tr>
<td>MPY Wm*Wm,Acc,[Wx],Wxd,[Wy],Wyd</td>
<td>Square to Acc</td>
<td>1</td>
<td>1</td>
<td>297</td>
</tr>
<tr>
<td>MPY.N Wm*Wn,Acc,[Wx],Wxd,[Wy],Wyd</td>
<td>-(Multiply Wn by Wm) to Acc</td>
<td>1</td>
<td>1</td>
<td>299</td>
</tr>
<tr>
<td>MSC Wm*Wn,Acc,[Wx],Wxd,[Wy],Wyd,AWB</td>
<td>Multiply and subtract from Acc</td>
<td>1</td>
<td>1</td>
<td>301</td>
</tr>
<tr>
<td>NEG Acc</td>
<td>Negate Acc</td>
<td>1</td>
<td>1</td>
<td>335</td>
</tr>
<tr>
<td>SAC Acc,#Slit4,Wdo</td>
<td>Store Acc</td>
<td>1</td>
<td>1</td>
<td>389</td>
</tr>
<tr>
<td>SAC.R Acc,#Slit4,Wdo</td>
<td>Store rounded Acc</td>
<td>1</td>
<td>1</td>
<td>391</td>
</tr>
<tr>
<td>SFTAC Acc,#Slit6</td>
<td>Arithmetic shift Acc by Slit6</td>
<td>1</td>
<td>1</td>
<td>397</td>
</tr>
<tr>
<td>SFTAC Acc,Wn</td>
<td>Arithmetic shift Acc by (Wn)</td>
<td>1</td>
<td>1</td>
<td>398</td>
</tr>
<tr>
<td>SUB Acc</td>
<td>Subtract accumulators</td>
<td>1</td>
<td>1</td>
<td>410</td>
</tr>
</tbody>
</table>

**Note 1:** In dsPIC33E and PIC24E devices, read and read-modify-write operations on non-CPU Special Function Registers require an additional cycle when compared to dsPIC30F, dsPIC33F, PIC24F and PIC24H devices.
HIGHLIGHTS

This section of the manual contains the following major topics:

4.1 Data Addressing Modes ......................................................... 52
4.2 Program Addressing Modes .................................................... 61
4.3 Instruction Stalls ................................................................. 62
4.4 Byte Operations ................................................................. 64
4.5 Word Move Operations ....................................................... 66
4.6 Using 10-bit Literal Operands ............................................. 69
4.7 Software Stack Pointer and Frame Pointer ......................... 70
4.8 Conditional Branch Instructions .......................................... 76
4.9 Z Status Bit ........................................................................... 77
4.10 Assigned Working Register Usage ...................................... 78
4.11 DSP Data Formats (dsPIC30F, dsPIC33F and dsPIC33E Devices) ......................................................... 81
4.12 Accumulator Usage (dsPIC30F, dsPIC33F and dsPIC33E Devices) ......................................................... 83
4.13 Accumulator Access (dsPIC30F, dsPIC33F and dsPIC33E Devices) ......................................................... 84
4.14 DSP MAC Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices) ......................................................... 84
4.15 DSP Accumulator Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices) ......................................................... 88
4.16 Scaling Data with the FBCL Instruction (dsPIC30F, dsPIC33F and dsPIC33E Devices) 88
4.17 Normalizing the Accumulator with the FBCL Instruction (dsPIC30F, dsPIC33F and dsPIC33E Devices) ............ 90
4.1 DATA ADDRESSING MODES

The 16-bit MCU and DSC devices support three native Addressing modes for accessing data memory, along with several forms of immediate addressing. Data accesses may be performed using file register addressing, register direct or indirect addressing, and immediate addressing, allowing a fixed value to be used by the instruction.

File register addressing provides the ability to operate on data stored in the lower 8K of data memory (Near RAM), and also move data between the working registers and the entire 64K data space. Register direct addressing is used to access the 16 memory mapped working registers, W0:W15. Register indirect addressing is used to efficiently operate on data stored in the entire 64K data space (and also Extended Data Space, in the case of dsPIC33E/PIC24E), using the contents of the working registers as an effective address. Immediate addressing does not access data memory, but provides the ability to use a constant value as an instruction operand. The address range of each mode is summarized in Table 4-1.

### Table 4-1: 16-bit MCU and DSC Addressing Modes

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Register</td>
<td>0x0000–0xFFFF (1)</td>
</tr>
<tr>
<td>Register Direct</td>
<td>0x0000–0x001F (working register array W0:W15)</td>
</tr>
<tr>
<td>Register Indirect</td>
<td>0x0000–0xFFFF</td>
</tr>
<tr>
<td>Immediate</td>
<td>N/A (constant value)</td>
</tr>
</tbody>
</table>

**Note 1:** The address range for the File Register MOV is 0x0000–0xFFFF.

4.1.1 File Register Addressing

File register addressing is used by instructions which use a predetermined data address as an operand for the instruction. The majority of instructions that support file register addressing provide access to the lower 8 Kbytes of data memory, which is called the Near RAM. However, the MOV instruction provides access to all 64 Kbytes of memory using file register addressing. This allows the loading of the data from any location in data memory to any working register, and storing the contents of any working register to any location in data memory. It should be noted that file register addressing supports both byte and word accesses of data memory, with the exception of the MOV instruction, which accesses all 64K of memory as words. Examples of file register addressing are shown in Example 4-1.

Most instructions, which support file register addressing, perform an operation on the specified file register and the default working register WREG (see Section 2.4 “Default Working Register (WREG)”). If only one operand is supplied in the instruction, WREG is an implied operand and the operation results are stored back to the file register. In these cases, the instruction is effectively a read-modify-write instruction. However, when both the file register and the WREG register are specified in the instruction, the operation results are stored in the WREG register and the contents of the file register are unchanged. Sample instructions that show the interaction between the file register and the WREG register are shown in Example 4-2.

**Note:** Instructions which support file register addressing use ‘f’ as an operand in the instruction summary tables of Section 3, “Instruction Set Overview”.
Example 4-1: File Register Addressing

```
DEC 0x1000 ; decrement data stored at 0x1000
Before Instruction:
    Data Memory 0x1000 = 0x5555
After Instruction:
    Data Memory 0x1000 = 0x5554
```

```
MOV 0x27FE, W0 ; move data stored at 0x27FE to W0
Before Instruction:
    W0 = 0x5555
    Data Memory 0x27FE = 0x1234
After Instruction:
    W0 = 0x1234
    Data Memory 0x27FE = 0x1234
```

Example 4-2: File Register Addressing and WREG

```
AND 0x1000 ; AND 0x1000 with WREG, store to 0x1000
Before Instruction:
    W0 (WREG) = 0x332C
    Data Memory 0x1000 = 0x5555
After Instruction:
    W0 (WREG) = 0x332C
    Data Memory 0x1000 = 0x1104
```

```
AND 0x1000, WREG ; AND 0x1000 with WREG, store to WREG
Before Instruction:
    W0 (WREG) = 0x332C
    Data Memory 0x1000 = 0x5555
After Instruction:
    W0 (WREG) = 0x1104
    Data Memory 0x1000 = 0x5555
```

4.1.2 Register Direct Addressing

Register direct addressing is used to access the contents of the 16 working registers (W0:W15). The Register Direct Addressing mode is fully orthogonal, which allows any working register to be specified for any instruction that uses register direct addressing, and it supports both byte and word accesses. Instructions which employ register direct addressing use the contents of the specified working register as data to execute the instruction, therefore this Addressing mode is useful only when data already resides in the working register core. Sample instructions which utilize register direct addressing are shown in Example 4-3.

Another feature of register direct addressing is that it provides the ability for dynamic flow control. Since variants of the DO and REPEAT instruction support register direct addressing, flexible looping constructs may be generated using these instructions.

Note: Instructions which must use register direct addressing, use the symbols Wb, Wn, Wns and Wnd in the summary tables of Section 3. “Instruction Set Overview”. Commonly, register direct addressing may also be used when register indirect addressing may be used. Instructions which use register indirect addressing, use the symbols Wd and Ws in the summary tables of Section 3. “Instruction Set Overview”.
4.1.3 Register Indirect Addressing

Register indirect addressing is used to access any location in data memory by treating the contents of a working register as an Effective Address (EA) to data memory. Essentially, the contents of the working register become a pointer to the location in data memory which is to be accessed by the instruction.

This Addressing mode is powerful, because it also allows one to modify the contents of the working register, either before or after the data access is made, by incrementing or decrementing the EA. By modifying the EA in the same cycle that an operation is being performed, register indirect addressing allows for the efficient processing of data that is stored sequentially in memory. The modes of indirect addressing supported by the 16-bit MCU and DSC devices are shown in Table 4-2.

Table 4-2: Indirect Addressing Modes

<table>
<thead>
<tr>
<th>Indirect Mode</th>
<th>Syntax</th>
<th>Function (Byte Instruction)</th>
<th>Function (Word Instruction)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Increment</td>
<td>[+Wn]</td>
<td>EA = [Wn + = 1]</td>
<td>EA = [Wn + = 2]</td>
<td>Wn is pre-incremented to form the EA.</td>
</tr>
<tr>
<td>Pre-Decrement</td>
<td>[-Wn]</td>
<td>EA = [Wn - = 1]</td>
<td>EA = [Wn - = 2]</td>
<td>Wn is pre-decremented to form the EA.</td>
</tr>
<tr>
<td>Post-Decrement</td>
<td>[Wn-]</td>
<td>EA = [Wn] - = 1</td>
<td>EA = [Wn] - = 2</td>
<td>The contents of Wn forms the EA, then Wn is post-decremented.</td>
</tr>
<tr>
<td>Register Offset</td>
<td>[Wn+Wb]</td>
<td>EA = [Wn + Wb]</td>
<td>EA = [Wn + Wb]</td>
<td>The sum of Wn and Wb forms the EA. Wn and Wb are not modified.</td>
</tr>
</tbody>
</table>
Table 4-2 shows that four Addressing modes modify the EA used in the instruction, and this allows the following updates to be made to the working register: post-increment, post-decrement, pre-increment and pre-decrement. Since all EAs must be given as byte addresses, support is provided for Word mode instructions by scaling the EA update by 2. Namely, in Word mode, pre/post-decrements subtract 2 from the EA stored in the working register, and pre/post-increments add 2 to the EA. This feature ensures that after an EA modification is made, the EA will point to the next adjacent word in memory. Example 4-4 shows how indirect addressing may be used to update the EA.

Table 4-2 also shows that the Register Offset mode addresses data which is offset from a base EA stored in a working register. This mode uses the contents of a second working register to form the EA by adding the two specified working registers. This mode does not scale for Word mode instructions, but offers the complete offset range of 64 Kbytes. Note that neither of the working registers used to form the EA are modified. Example 4-5 shows how register offset indirect addressing may be used to access data memory.

### Note:

The MOV with offset instructions (see pages 285 and 286) provides a literal addressing offset ability to be used with indirect addressing. In these instructions, the EA is formed by adding the contents of a working register to a signed 10-bit literal. Example 4-6 shows how these instructions may be used to move data to and from the working register array.

#### Example 4-4: Indirect Addressing with Effective Address Update

MOV.B [W0++], [W13--] ; byte move [W0] to [W13]
; post-inc W0, post-dec W13

**Before Instruction:**

- W0 = 0x2300
- W13 = 0x2708

Data Memory 0x2300 = 0x7783
Data Memory 0x2708 = 0x904E

**After Instruction:**

- W0 = 0x2301
- W13 = 0x2707

Data Memory 0x2300 = 0x7783
Data Memory 0x2708 = 0x9083

ADD W1, [--W5], [++W8] ; pre-dec W5, pre-inc W8
; add W1 to [W5], store in [W8]

**Before Instruction:**

- W1 = 0x0800
- W5 = 0x2200
- W8 = 0x2400

Data Memory 0x21FE = 0x7783
Data Memory 0x2402 = 0xAACC

**After Instruction:**

- W1 = 0x0800
- W5 = 0x21FE
- W8 = 0x2402

Data Memory 0x21FE = 0x7783
Data Memory 0x2402 = 0x7F83
Example 4-5: Indirect Addressing with Register Offset

MOV.B [W0+W1], [W7++] ; byte move [W0+W1] to W7, post-inc W7

Before Instruction:
W0 = 0x2300
W1 = 0x01FE
W7 = 0x1000
Data Memory 0x24FE = 0x7783
Data Memory 0x1000 = 0x11DC

After Instruction:
W0 = 0x2300
W1 = 0x01FE
W7 = 0x1001
Data Memory 0x24FE = 0x7783
Data Memory 0x1000 = 0x1183

LAC [W0+W8], A ; load ACCA with [W0+W8]
; (sign-extend and zero-backfill)

Before Instruction:
W0 = 0x2344
W8 = 0x0008
ACCA = 0x00 7877 9321
Data Memory 0x234C = 0xE290

After Instruction:
W0 = 0x2344
W8 = 0x0008
ACCA = 0xFF E290 0000
Data Memory 0x234C = 0xE290

Example 4-6: Move with Literal Offset Instructions

MOV [W0+0x20], W1 ; move [W0+0x20] to W1

Before Instruction:
W0 = 0x1200
W1 = 0x01FE
Data Memory 0x1220 = 0xFD27

After Instruction:
W0 = 0x1200
W1 = 0xFD27
Data Memory 0x1220 = 0xFD27

MOV W4, [W8-0x300] ; move W4 to [W8-0x300]

Before Instruction:
W4 = 0x3411
W8 = 0x2944
Data Memory 0x2644 = 0xCB98

After Instruction:
W4 = 0x3411
W8 = 0x2944
Data Memory 0x2644 = 0x3411
4.1.3.1 REGISTER INDIRECT ADDRESSING AND THE INSTRUCTION SET

The Addressing modes presented in Table 4-2 demonstrate the Indirect Addressing mode capability of the 16-bit MCU and DSC devices. Due to operation encoding and functional considerations, not every instruction which supports indirect addressing supports all modes shown in Table 4-2. The majority of instructions which use indirect addressing support the No Modify, Pre-Increment, Pre-Decrement, Post-Increment and Post-Decrement Addressing modes. The MOV instructions, and several accumulator-based DSP instructions (dsPIC30F, dsPIC33F, and dsPIC33E devices only), are also capable of using the Register Offset Addressing mode.

Note: Instructions which use register indirect addressing use the operand symbols Wd and Ws in the summary tables of Section 3. “Instruction Set Overview”.

4.1.3.2 DSP MAC INDIRECT ADDRESSING MODES (dsPIC30F, dsPIC33F, AND dsPIC33E DEVICES)

A special class of Indirect Addressing modes is utilized by the DSP MAC instructions. As is described later in Section 4.14 “DSP MAC Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices)”, the DSP MAC class of instructions are capable of performing two fetches from memory using effective addressing. Since DSP algorithms frequently demand a broader range of address updates, the Addressing modes offered by the DSP MAC instructions provide greater range in the size of the effective address update which may be made. Table 4-3 shows that both X and Y prefetches support Post-Increment and Post-Decrement Addressing modes, with updates of 2, 4 and 6 bytes. Since DSP instructions only execute in Word mode, no provisions are made for odd sized EA updates.

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>X Memory</th>
<th>Y Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indirect with no modification</td>
<td>EA = [Wx]</td>
<td>EA = [Wy]</td>
</tr>
<tr>
<td>Indirect with Post-Increment by 2</td>
<td>EA = [Wx] + = 2</td>
<td>EA = [Wy] + = 2</td>
</tr>
<tr>
<td>Indirect with Post-Increment by 4</td>
<td>EA = [Wx] + = 4</td>
<td>EA = [Wy] + = 4</td>
</tr>
<tr>
<td>Indirect with Post-Increment by 6</td>
<td>EA = [Wx] + = 6</td>
<td>EA = [Wy] + = 6</td>
</tr>
</tbody>
</table>

Note: As described in Section 4.14 “DSP MAC Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices)”, only W8 and W9 may be used to access X Memory, and only W10 and W11 may be used to access Y Memory.

4.1.3.3 MODULO AND BIT-REVERSED ADDRESSING MODES (dsPIC30F, dsPIC33F, AND dsPIC33E DEVICES)

The 16-bit DSC architecture provides support for two special Register Indirect Addressing modes, which are commonly used to implement DSP algorithms. Modulo (or circular) addressing provides an automated means to support circular data buffers in X and/or Y memory. Modulo buffers remove the need for software to perform address boundary checks, which can improve the performance of certain algorithms. Similarly, bit-reversed addressing allows one to access the elements of a buffer in a nonlinear fashion. This Addressing mode simplifies data re-ordering for radix-2 FFT algorithms and provides a significant reduction in FFT processing time.

Both of these Addressing modes are powerful features of the dsPIC30F, dsPIC33F, and dsPIC33E architectures, which can be exploited by any instruction that uses indirect addressing. Refer to the specific device family reference manual for details on using modulo and bit-reversed addressing.
4.1.4 Immediate Addressing

In immediate addressing, the instruction encoding contains a predefined constant operand, which is used by the instruction. This Addressing mode may be used independently, but it is more frequently combined with the File Register, Direct and Indirect Addressing modes. The size of the immediate operand which may be used varies with the instruction type. Constants of size 1-bit (#lit1), 4-bit (#bit4, #lit4 and #Slit4), 5-bit (#lit5), 6-bit (#Slit6), 8-bit (#lit8), 10-bit (#lit10 and #Slit10), 14-bit (#lit14) and 16-bit (#lit16) may be used. Constants may be signed or unsigned and the symbols #Slit4, #Slit6 and #Slit10 designate a signed constant. All other immediate constants are unsigned. Table 4-4 shows the usage of each immediate operand in the instruction set.

Table 4-4: Immediate Operands in the Instruction Set

<table>
<thead>
<tr>
<th>Operand</th>
<th>Instruction Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>#lit1</td>
<td>PWRSAV</td>
</tr>
<tr>
<td>#bit4</td>
<td>BCLR, BSET, BTG, BTSC, BTST, BTST.C, BTST.Z, BTSTS, BTSTS.C, BTSTS.Z</td>
</tr>
<tr>
<td>#lit4</td>
<td>ASR, LSR, SL</td>
</tr>
<tr>
<td>#Slit4</td>
<td>ADD, LAC, SAC, SAC.R</td>
</tr>
<tr>
<td>#lit5</td>
<td>ADD, ADDC, AND, CP(5), CPB(5), IOR, MUL.SU, MUL.UU, SUB, SUBB, SUBBR, SUBR, XOR</td>
</tr>
<tr>
<td>#Slit6(1)</td>
<td>SFTAC</td>
</tr>
<tr>
<td>#lit8</td>
<td>MOV.B, CP(4), CPB(4)</td>
</tr>
<tr>
<td>#lit10</td>
<td>ADD, ADDC, AND, CP, CPB, IOR, RETLW, SUB, SUBB, XOR</td>
</tr>
<tr>
<td>#Slit10</td>
<td>MOV</td>
</tr>
<tr>
<td>#lit14</td>
<td>DISI, DO(2), LNK, REPEAT(5)</td>
</tr>
<tr>
<td>#lit15</td>
<td>DO(3), REPEAT(4)</td>
</tr>
<tr>
<td>#lit16</td>
<td>MOV</td>
</tr>
</tbody>
</table>

Note 1: This operand or instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

Note 2: This operand or instruction is only available in dsPIC30F and dsPIC33F devices.

Note 3: This operand or instruction is only available in dsPIC33E devices.

Note 4: This operand or instruction is only available in dsPIC33E and PIC24E devices.

Note 5: This operand or instruction is only available in dsPIC30F, dsPIC33F, PIC24F, and PIC24H devices.

The syntax for immediate addressing requires that the number sign (#) must immediately precede the constant operand value. The “#” symbol indicates to the assembler that the quantity is a constant. If an out-of-range constant is used with an instruction, the assembler will generate an error. Several examples of immediate addressing are shown in Example 4-7.
Example 4-7: Immediate Addressing

\[
\begin{align*}
\text{PWRSAV} &\ #1 &; \text{Enter IDLE mode} \\
\text{ADD.B} &\ #0x10, \ W0 &; \text{Add 0x10 to W0 (byte mode)}
\end{align*}
\]

Before Instruction:
\[\text{W0} = 0x12A9\]

After Instruction:
\[\text{W0} = 0x12B9\]

\[
\begin{align*}
\text{XOR} &\ W0, \ #1, \ [W1++] &; \text{Exclusive-OR W0 and 0x1} \\
&\; \text{Store the result to [W1]} \\
&\; \text{Post-increment W1}
\end{align*}
\]

Before Instruction:
\[\begin{align*}
\text{W0} &= 0xFFFF \\
\text{W1} &= 0x0890 \\
\text{Data Memory 0x0890} &= 0x0032
\end{align*}\]

After Instruction:
\[\begin{align*}
\text{W0} &= 0xFFFF \\
\text{W1} &= 0x0892 \\
\text{Data Memory 0x0890} &= 0xFFFE
\end{align*}\]

### 4.1.5 Data Addressing Mode Tree

The Data Addressing modes of the PIC24F, PIC24H, and PIC24E families are summarized in Figure 4-1.

**Figure 4-1: Data Addressing Mode Tree (PIC24F, PIC24H, and PIC24E)**

<table>
<thead>
<tr>
<th>Data Addressing Modes</th>
<th>Immediate</th>
<th>No Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Register</td>
<td>Pre-Increment</td>
<td>Pre-Decrement</td>
</tr>
<tr>
<td>Direct</td>
<td>Post-Increment</td>
<td>Post-Decrement</td>
</tr>
<tr>
<td>Indirect</td>
<td>Literal Offset</td>
<td>Register Offset</td>
</tr>
</tbody>
</table>

The Data Addressing modes of the dsPIC30F, dsPIC33F, and dsPIC33E are summarized in Figure 4-2.
Figure 4-2: Data Addressing Mode Tree (dsPIC30F, dsPIC33F, and dsPIC33E)
4.2 PROGRAM ADDRESSING MODES

The 16-bit MCU and DSC devices have a 24-bit Program Counter (PC). The PC addresses the 24-bit wide program memory to fetch instructions for execution, and it may be loaded in several ways. For byte compatibility with the table read and table write instructions, each instruction word consumes two locations in program memory. This means that during serial execution, the PC is loaded with PC + 2.

Several methods may be used to modify the PC in a non-sequential manner, and both absolute and relative changes may be made to the PC. The change to the PC may be from an immediate value encoded in the instruction, or a dynamic value contained in a working register. In dsPIC30F, dsPIC33F, and dsPIC33E devices, when DO looping is active, the PC is loaded with the address stored in the DOSTART register, after the instruction at the DOEND address is executed. For exception handling, the PC is loaded with the address of the exception handler, which is stored in the interrupt vector table. When required, the software stack is used to return scope to the foreground process from where the change in program flow occurred.

Table 4-5 summarizes the instructions which modify the PC. When performing function calls, it is recommended that RCALL be used instead of CALL, since RCALL only consumes 1 word of program memory.

<table>
<thead>
<tr>
<th>Condition/Instruction</th>
<th>PC Modification</th>
<th>Software Stack Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Execution</td>
<td>PC = PC + 2</td>
<td>None</td>
</tr>
<tr>
<td>BRA Expr (Branch Unconditionally)</td>
<td>PC = PC + 2 * Slit16</td>
<td>None</td>
</tr>
<tr>
<td>BRA Condition, Expr (Branch Conditionally)</td>
<td>PC = PC + 2 (condition false) PC = PC + 2 * Slit16 (condition true)</td>
<td>None</td>
</tr>
<tr>
<td>CALL Expr (Call Subroutine)</td>
<td>PC = lit23</td>
<td>PC + 4 is PUSHed on the stack(2)</td>
</tr>
<tr>
<td>CALL Wn (Call Subroutine Indirect)</td>
<td>PC = Wn</td>
<td>PC + 2 is PUSHed on the stack(2)</td>
</tr>
<tr>
<td>CALL.L Wn (Call Indirect Subroutine Long)</td>
<td>PC = (Wn+1:Wn)</td>
<td>None</td>
</tr>
<tr>
<td>GOTO Expr (Unconditional Jump)</td>
<td>PC = lit23</td>
<td>None</td>
</tr>
<tr>
<td>GOTO Wn (Unconditional Indirect Jump)</td>
<td>PC = Wn</td>
<td>None</td>
</tr>
<tr>
<td>GOTO.L Wn (Unconditional Indirect Long Jump)</td>
<td>PC = (Wn+1:Wn)</td>
<td>None</td>
</tr>
<tr>
<td>RCALL Expr (Relative Call)</td>
<td>PC = PC + 2 * Slit16</td>
<td>PC + 2 is PUSHed on the stack(2)</td>
</tr>
<tr>
<td>RCALL Wn (Computed Relative Call)</td>
<td>PC = PC + 2 * Wn</td>
<td>PC + 2 is PUSHed on the stack(2)</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>PC = address of the exception handler (read from vector table)</td>
<td>PC + 2 is PUSHed on the stack(3)</td>
</tr>
<tr>
<td>PC = Target REPEAT instruction (REPEAT Looping)</td>
<td>PC not modified (if REPEAT active)</td>
<td>None</td>
</tr>
<tr>
<td>PC = DOEND address (DO Looping)</td>
<td>PC = DOSTART (if DO active)</td>
<td>None</td>
</tr>
</tbody>
</table>

Note 1: For BRA, CALL and GOTO, the Expr may be a label, absolute address, or expression, which is resolved by the linker to a 16-bit or 23-bit value (Slit16 or lit23). See Section 5. “Instruction Descriptions” for details.

2: After CALL or RCALL is executed, RETURN or RETLW will POP the Top-of-Stack (TOS) back into the PC.

3: After an exception is processed, RETFIE will POP the Top-of-Stack (TOS) back into the PC.

4: This condition/instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

5: This condition instruction is only available in dsPIC33E and PIC24E devices.
4.3 INSTRUCTION STALLS

In order to maximize the data space EA calculation and operand fetch time, the X data space read and write accesses are partially pipelined. A consequence of this pipelining is that address register data dependencies may arise between successive read and write operations using common registers.

‘Read After Write’ (RAW) dependencies occur across instruction boundaries and are detected by the hardware. An example of a RAW dependency would be a write operation that modifies W5, followed by a read operation that uses W5 as an Address Pointer. The contents of W5 will not be valid for the read operation until the earlier write completes. This problem is resolved by stalling the instruction execution for one instruction cycle, which allows the write to complete before the next read is started.

4.3.1 RAW Dependency Detection

During the instruction pre-decode, the core determines if any address register dependency is imminent across an instruction boundary. The stall detection logic compares the W register (if any) used for the destination EA of the instruction currently being executed with the W register to be used by the source EA (if any) of the prefetched instruction. When a match between the destination and source registers is identified, a set of rules are applied to decide whether or not to stall the instruction by one cycle. Table 4-6 lists various RAW conditions which cause an instruction execution stall.

Table 4-6: Raw Dependency Rules (Detection By Hardware)

<table>
<thead>
<tr>
<th>Destination Address Mode Using Wn</th>
<th>Source Address Mode Using Wn</th>
<th>Stall Required?</th>
<th>Examples (Wn = W2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct</td>
<td>Direct</td>
<td>No Stall</td>
<td>ADD.W W0, W1, W2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W W2, W3</td>
</tr>
<tr>
<td>Indirect</td>
<td>Direct</td>
<td>No Stall</td>
<td>ADD.W W0, W1, [W2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W W2, W3</td>
</tr>
<tr>
<td>Indirect</td>
<td>Indirect</td>
<td>No Stall</td>
<td>ADD.W W0, W1, [W2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2], W3</td>
</tr>
<tr>
<td>Indirect</td>
<td>Indirect with pre/post-modification</td>
<td>No Stall</td>
<td>ADD.W W0, W1, [W2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2++], W3</td>
</tr>
<tr>
<td>Indirect with pre/post-modification</td>
<td>Direct</td>
<td>No Stall</td>
<td>ADD.W W0, W1, [W2++]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W W2, W3</td>
</tr>
<tr>
<td>Direct</td>
<td>Indirect</td>
<td>Stall(1)</td>
<td>ADD.W W0, W1, W2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2], W3</td>
</tr>
<tr>
<td>Direct</td>
<td>Indirect with pre/post-modification</td>
<td>Stall(1)</td>
<td>ADD.W W0, W1, W2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2++], W3</td>
</tr>
<tr>
<td>Indirect</td>
<td>Indirect</td>
<td>Stall(1)</td>
<td>ADD.W W0, W1, [W2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2], W3</td>
</tr>
<tr>
<td>Indirect</td>
<td>Indirect with pre/post-modification</td>
<td>Stall(1)</td>
<td>ADD.W W0, W1, [W2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2++], W3</td>
</tr>
<tr>
<td>Indirect with pre/post-modification</td>
<td>Indirect</td>
<td>Stall(1)</td>
<td>ADD.W W0, W1, [W2++]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2], W3</td>
</tr>
<tr>
<td>Indirect with pre/post-modification</td>
<td>Indirect with pre/post-modification</td>
<td>Stall(1)</td>
<td>ADD.W W0, W1, [W2++]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MOV.W [W2++], W3</td>
</tr>
</tbody>
</table>

Note 1: When stalls are detected, one cycle is added to the instruction execution time.
Note 2: For these examples, the contents of W2 = the mapped address of W2 (0x0004).
4.3.2 Instruction Stalls and Exceptions

In order to maintain deterministic operation, instruction stalls are allowed to happen, even if they occur immediately prior to exception processing.

4.3.3 Instruction Stalls and Instructions that Change Program Flow

CALL and RCALL write to the stack using W15 and may, therefore, be subject to an instruction stall if the source read of the subsequent instruction uses W15.

GOTO, RETFIE and RETURN instructions are never subject to an instruction stall because they do not perform write operations to the working registers.

4.3.4 Instruction Stalls and DO/REPEAT Loops

Instructions operating in a DO or REPEAT loop are subject to instruction stalls, just like any other instruction. Stalls may occur on loop entry, loop exit and also during loop processing.

Note: DO loops are only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.

4.3.5 Instruction Stalls and PSV

Instructions operating in PSV address space are subject to instruction stalls, just like any other instruction. Should a data dependency be detected in the instruction immediately following the PSV data access, the second cycle of the instruction will initiate a stall. Should a data dependency be detected in the instruction immediately before the PSV data access, the last cycle of the previous instruction will initiate a stall.

Note: Refer to the specific device family reference manual for more detailed information about RAW instruction stalls.
4.4 BYTE OPERATIONS

Since the data memory is byte addressable, most of the base instructions may operate in either Byte mode or Word mode. When these instructions operate in Byte mode, the following rules apply:

- All direct working register references use the Least Significant Byte of the 16-bit working register and leave the Most Significant Byte (MSB) unchanged.
- All indirect working register references use the data byte specified by the 16-bit address stored in the working register.
- All file register references use the data byte specified by the byte address.
- The STATUS Register is updated to reflect the result of the byte operation.

It should be noted that data addresses are always represented as byte addresses. Additionally, the native data format is little-endian, which means that words are stored with the Least Significant Byte at the lower address, and the Most Significant Byte at the adjacent, higher address (as shown in Figure 4-3). Example 4-8 shows sample byte move operations and Example 4-9 shows sample byte math operations.

**Note:** Instructions that operate in Byte mode must use the ".b" or ".B" instruction extension to specify a byte instruction. For example, the following two instructions are valid forms of a byte clear operation:

- CLR.b W0
- CLR.B W0

**Example 4-8:** Sample Byte Move Operations

<table>
<thead>
<tr>
<th>MOV.B #0x30, W0 ; move the literal byte 0x30 to W0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Instruction:</td>
</tr>
<tr>
<td>W0 = 0x5555</td>
</tr>
<tr>
<td>After Instruction:</td>
</tr>
<tr>
<td>W0 = 0x5530</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV.B 0x1000, W0 ; move the byte at 0x1000 to W0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Instruction:</td>
</tr>
<tr>
<td>W0 = 0x5555</td>
</tr>
<tr>
<td>Data Memory 0x1000 = 0x1234</td>
</tr>
<tr>
<td>After Instruction:</td>
</tr>
<tr>
<td>W0 = 0x5534</td>
</tr>
<tr>
<td>Data Memory 0x1000 = 0x1234</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV.B W0, 0x1001 ; byte move W0 to address 0x1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Instruction:</td>
</tr>
<tr>
<td>W0 = 0x1234</td>
</tr>
<tr>
<td>Data Memory 0x1000 = 0x5555</td>
</tr>
<tr>
<td>After Instruction:</td>
</tr>
<tr>
<td>W0 = 0x1234</td>
</tr>
<tr>
<td>Data Memory 0x1000 = 0x3455</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MOV.B W0, [W1++] ; byte move W0 to [W1], then post-inc W1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before Instruction:</td>
</tr>
<tr>
<td>W0 = 0x1234</td>
</tr>
<tr>
<td>W1 = 0x1001</td>
</tr>
<tr>
<td>Data Memory 0x1000 = 0x5555</td>
</tr>
<tr>
<td>After Instruction:</td>
</tr>
<tr>
<td>W0 = 0x1234</td>
</tr>
<tr>
<td>W1 = 0x1002</td>
</tr>
<tr>
<td>Data Memory 0x1000 = 0x3455</td>
</tr>
</tbody>
</table>
Example 4-9: Sample Byte Math Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR.B [W6--]</td>
<td>W6 = 0x1001</td>
<td>W6 = 0x1000</td>
</tr>
<tr>
<td></td>
<td>Data Memory 0x1000 = 0x5555</td>
<td>Data Memory 0x1000 = 0x0055</td>
</tr>
<tr>
<td>SUB.B W0, #0x10, W1</td>
<td>W0 = 0x1234, W1 = 0xFFFF</td>
<td>W0 = 0x1234, W1 = 0xFF24</td>
</tr>
<tr>
<td>ADD.B W0, W1, [W2++]</td>
<td>W0 = 0x1234, W1 = 0x5678, W2 = 0x1000, Data Memory 0x1000 = 0x5555</td>
<td>W0 = 0x1234, W1 = 0x5678, W2 = 0x1001, Data Memory 0x1000 = 0x55AC</td>
</tr>
</tbody>
</table>
4.5 WORD MOVE OPERATIONS

Even though the data space is byte addressable, all move operations made in Word mode must be word-aligned. This means that for all source and destination operands, the Least Significant address bit must be ‘0’. If a word move is made to or from an odd address, an address error exception is generated. Likewise, all double words must be word-aligned. Figure 4-3 shows how bytes and words may be aligned in data memory. Example 4-10 contains several legal word move operations.

When an exception is generated due to a misaligned access, the exception is taken after the instruction executes. If the illegal access occurs from a data read, the operation will be allowed to complete, but the Least Significant bit of the source address will be cleared to force word alignment. If the illegal access occurs during a data write, the write will be inhibited. Example 4-11 contains several illegal word move operations.

Figure 4-3: Data Alignment in Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1001</td>
<td>b0</td>
</tr>
<tr>
<td>0x1003</td>
<td>b1</td>
</tr>
<tr>
<td>0x1005</td>
<td>b3:b2</td>
</tr>
<tr>
<td>0x1007</td>
<td>b5</td>
</tr>
<tr>
<td>0x1009</td>
<td>b7:b6</td>
</tr>
<tr>
<td>0x100B</td>
<td>b8</td>
</tr>
<tr>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td>0x1002</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td></td>
</tr>
<tr>
<td>0x1006</td>
<td></td>
</tr>
<tr>
<td>0x1008</td>
<td></td>
</tr>
<tr>
<td>0x100A</td>
<td></td>
</tr>
</tbody>
</table>

Legend:
- b0 – byte stored at 0x1000
- b1 – byte stored at 0x1003
- b3:b2 – word stored at 0x1005:1004 (b2 is LSB)
- b7:b4 – double word stored at 0x1009:0x1006 (b4 is LSB)
- b8 – byte stored at 0x100A

Note: Instructions that operate in Word mode are not required to use an instruction extension. However, they may be specified with an optional “.w” or “.W” extension, if desired. For example, the following instructions are valid forms of a word clear operation:
- CLR W0
- CLR.w W0
- CLR.W W0

Instructions that operate in Word mode are not required to use an instruction extension. However, they may be specified with an optional “.w” or “.W” extension, if desired. For example, the following instructions are valid forms of a word clear operation:
- CLR W0
- CLR.w W0
- CLR.W W0
### Example 4-10: Legal Word Move Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #0x30, W0 ; move the literal word 0x30 to W0</td>
<td>W0 = 0x5555</td>
<td>W0 = 0x0030</td>
</tr>
<tr>
<td>MOV 0x1000, W0 ; move the word at 0x1000 to W0</td>
<td>Data Memory 0x1000 - 0x1234</td>
<td>Data Memory 0x1000 - 0x1234</td>
</tr>
<tr>
<td>MOV [W0], [W1++] ; word move [W0] to [W1], then post-inc W1</td>
<td>W0 = 0x1234 W1 = 0x1000 Data Memory 0x1000 - 0x5555 Data Memory 0x1234 - 0xA AAA</td>
<td>W0 = 0x1234 W1 = 0x1002 Data Memory 0x1000 - 0xA AAA Data Memory 0x1234 - 0xA AAA</td>
</tr>
</tbody>
</table>
Example 4-11: Illegal Word Move Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Before Instruction</th>
<th>After Instruction</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| MOV 0x1001, W0 ; move the word at 0x1001 to W0 | W0 = 0x5555  
Data Memory 0x1000 = 0x1234  
Data Memory 0x1002 = 0x5678 | W0 = 0x1234  
Data Memory 0x1000 = 0x1234  
Data Memory 0x1002 = 0x5678 | ADDRESS ERROR TRAP GENERATED  
(source address is misaligned, so MOV is performed) |
| MOV W0, 0x1001 ; move W0 to the word at 0x1001 | W0 = 0x1234  
Data Memory 0x1000 = 0x5555  
Data Memory 0x1002 = 0x6666 | W0 = 0x1234  
Data Memory 0x1000 = 0x5555  
Data Memory 0x1002 = 0x6666 | ADDRESS ERROR TRAP GENERATED  
(destination address is misaligned, so MOV is not performed) |
| MOV [W0], [W1++] ; word move [W0] to [W1],  
then post-inc W1 | W0 = 0x1235  
W1 = 0x1000  
Data Memory 0x1000 = 0x1234  
Data Memory 0x1234 = 0xAAAA  
Data Memory 0x1236 = 0xBAAA | W0 = 0x1235  
W1 = 0x1002  
Data Memory 0x1000 = 0xAAAA  
Data Memory 0x1234 = 0xAAAA  
Data Memory 0x1236 = 0xBBBB | ADDRESS ERROR TRAP GENERATED  
(source address is misaligned, so MOV is performed) |
4.6 USING 10-BIT LITERAL OPERANDS

Several instructions that support Byte and Word mode have 10-bit operands. For byte instructions, a 10-bit literal is too large to use. So when 10-bit literals are used in Byte mode, the range of the operand must be reduced to 8 bits or the assembler will generate an error. Table 4-7 shows that the range of a 10-bit literal is 0:1023 in Word mode and 0:255 in Byte mode.

Instructions which employ 10-bit literals in Byte and Word mode are: ADD, ADDC, AND, IOR, RETLW, SUB, SUBB, and XOR. Example 4-12 shows how positive and negative literals are used in Byte mode for the ADD instruction.

<table>
<thead>
<tr>
<th>Literal Value</th>
<th>Word Mode kk kkkk kkkk</th>
<th>Byte Mode kkkk kkkk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 0000 0000</td>
<td>0000 0000</td>
</tr>
<tr>
<td>1</td>
<td>00 0000 0001</td>
<td>0000 0001</td>
</tr>
<tr>
<td>2</td>
<td>00 0000 0010</td>
<td>0000 0010</td>
</tr>
<tr>
<td>127</td>
<td>00 0111 1111</td>
<td>0111 1111</td>
</tr>
<tr>
<td>128</td>
<td>00 1000 0000</td>
<td>1000 0000</td>
</tr>
<tr>
<td>255</td>
<td>00 1111 1111</td>
<td>1111 1111</td>
</tr>
<tr>
<td>256</td>
<td>01 0000 0000 N/A</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>10 0000 0000 N/A</td>
<td></td>
</tr>
<tr>
<td>1023</td>
<td>11 1111 1111 N/A</td>
<td></td>
</tr>
</tbody>
</table>

Example 4-12: Using 10-bit Literals for Byte Operands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.B #0x80, W0</td>
<td>add 128 (or -128) to W0</td>
</tr>
<tr>
<td>ADD.B #0x380, W0</td>
<td>ERROR... Illegal syntax for byte mode</td>
</tr>
<tr>
<td>ADD.B #0xFF, W0</td>
<td>add 255 (or -1) to W0</td>
</tr>
<tr>
<td>ADD.B #0x3FF, W0</td>
<td>ERROR... Illegal syntax for byte mode</td>
</tr>
<tr>
<td>ADD.B #0xF, W0</td>
<td>add 15 to W0</td>
</tr>
<tr>
<td>ADD.B #0x7F, W0</td>
<td>add 127 to W0</td>
</tr>
<tr>
<td>ADD.B #0x100, W0</td>
<td>ERROR... Illegal syntax for byte mode</td>
</tr>
</tbody>
</table>

Note: Using a literal value greater than 127 in Byte mode is functionally identical to using the equivalent negative two's complement value, since the Most Significant bit of the byte is set. When operating in Byte mode, the Assembler will accept either a positive or negative literal value (i.e., #10).
4.7 SOFTWARE STACK POINTER AND FRAME POINTER

4.7.1 Software Stack Pointer

The 16-bit MCU and DSC devices feature a software stack which facilitates function calls and exception handling. W15 is the default Stack Pointer (SP) and after any Reset, it is initialized to 0x0800 (0x1000 for PIC24E and dsPIC33E devices). This ensures that the SP will point to valid RAM and permits stack availability for exceptions, which may occur before the SP is set by the user software. The user may reprogram the SP during initialization to any location within data space.

The SP always points to the first available free word (Top-of-Stack) and fills the software stack, working from lower addresses towards higher addresses. It pre-decrements for a stack POP (read) and post-increments for a stack PUSH (write).

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 Top-of-Stack (TOS) by:

```
PUSH W0
```

This syntax is equivalent to:

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

The contents of the TOS can be returned to W0 by:

```
POP W0
```

This syntax is equivalent to:

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

During any CALL instruction, the PC is PUSHed onto the stack, such that when the subroutine completes execution, program flow may resume from the correct location. 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. When PC<22:16> is PUSHed, the Most Significant 7 bits of the PC are zero-extended before the PUSH is made, as shown in Figure 4-4. During exception processing, the Most Significant 7 bits of the PC are concatenated with the lower byte of the STATUS register (SRL) and IPL<3>, CORCON<3>. This allows the primary STATUS register contents and CPU Interrupt Priority Level to be automatically preserved during interrupts.

Note: In order to protect against misaligned stack accesses, W15<0> is always clear.

Figure 4-4: Stack Operation for CALL Instruction

![Figure 4-4: Stack Operation for CALL Instruction](image)
4.7.1.1 STACK POINTER EXAMPLE

Figure 4-5 through Figure 4-8 show how the software stack is modified for the code snippet shown in Example 4-13. Figure 4-5 shows the software stack before the first PUSH has executed. Note that the SP has the initialized value of 0x0800. Furthermore, the example loads 0x5A5A and 0x3636 to W0 and W1, respectively. The stack is PUSHed for the first time in Figure 4-6 and the value contained in W0 is copied to TOS. W15 is automatically updated to point to the next available stack location, and the new TOS is 0x0802. In Figure 4-7, the contents of W1 are PUSHed onto the stack, and the new TOS becomes 0x0804. In Figure 4-8, the stack is POPped, which copies the last PUSHed value (W1) to W3. The SP is decremented during the POP operation, and at the end of the example, the final TOS is 0x0802.

Example 4-13: Stack Pointer Usage

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #0x5A5A, W0</td>
<td>Load W0 with 0x5A5A</td>
</tr>
<tr>
<td>MOV #0x3636, W1</td>
<td>Load W1 with 0x3636</td>
</tr>
<tr>
<td>PUSH W0</td>
<td>Push W0 to TOS (see Figure 4-5)</td>
</tr>
<tr>
<td>PUSH W1</td>
<td>Push W1 to TOS (see Figure 4-7)</td>
</tr>
<tr>
<td>POP W3</td>
<td>Pop TOS to W3 (see Figure 4-8)</td>
</tr>
</tbody>
</table>

Figure 4-5: Stack Pointer Before The First PUSH

```
0x0000
0x0800
0xFFFE
W15 = 0x0800
W0 = 0x5A5A
W1 = 0x3636
```

Figure 4-6: Stack Pointer After “PUSH W0” Instruction

```
0x0000
0x0800
0x0802
0xFFFE
W15 = 0x0802
W0 = 0x5A5A
W1 = 0x3636
W15 = 0x0800
```

Figure 4-7: Stack Pointer Before The Second PUSH

```
0x0000
0x0800
0xFFFE
W15 = 0x0804
W0 = 0x5A5A
W1 = 0x3636
```

Figure 4-8: Stack Pointer After “PUSH W1” Instruction

```
0x0000
0x0800
0xFFFE
W15 = 0x0806
W0 = 0x5A5A
W1 = 0x3636
W15 = 0x0802
```
4.7.2 Software Stack Frame Pointer

A Stack Frame is a user-defined section of memory residing in the software stack. It is used to allocate memory for temporary variables which a function uses, and one Stack Frame may be created for each function. W14 is the default Stack Frame Pointer (FP) and it is initialized to 0x0000 on any Reset. If the Stack Frame Pointer is not used, W14 may be used like any other working register.

The link (LNK) and unlink (ULNK) instructions provide Stack Frame functionality. The LNK instruction is used to create a Stack Frame. It is used during a call sequence to adjust the SP, such that the stack may be used to store temporary variables utilized by the called function. After the function completes execution, the ULNK instruction is used to remove the Stack Frame created by the LNK instruction. The LNK and ULNK instructions must always be used together to avoid stack overflow.

Note: The contents of 0x802, the new TOS, remain unchanged (0x3636).
4.7.2.1  STACK FRAME POINTER EXAMPLE

Figure 4-9 through Figure 4-11 show how a Stack Frame is created and removed for the code snippet shown in Example 4-14. This example demonstrates how a Stack Frame operates and is not indicative of the code generated by the compiler. Figure 4-9 shows the stack condition at the beginning of the example, before any registers are pushed to the stack. Here, W15 points to the first free stack location (TOS) and W14 points to a portion of stack memory allocated for the routine that is currently executing.

Before calling the function "COMPUTE", the parameters of the function (W0, W1 and W2) are PUSHed on the stack. After the "CALL COMPUTE" instruction is executed, the PC changes to the address of "COMPUTE" and the return address of the function "TASKA" is placed on the stack (Figure 4-10). Function "COMPUTE" then uses the "LNK #4" instruction to PUSH the calling routine’s Frame Pointer value onto the stack and the new Frame Pointer will be set to point to the current Stack Pointer. Then, the literal 4 is added to the Stack Pointer address in W15, which reserves memory for two words of temporary data (Figure 4-11).

Inside the function "COMPUTE", the FP is used to access the function parameters and temporary (local) variables. [W14 + n] will access the temporary variables used by the routine and [W14 – n] is used to access the parameters. At the end of the function, the ULNK instruction is used to copy the Frame Pointer address to the Stack Pointer and then POP the calling subroutine’s Frame Pointer back to the W14 register. The ULNK instruction returns the stack back to the state shown in Figure 4-10.

A RETURN instruction will return to the code that called the subroutine. The calling code is responsible for removing the parameters from the stack. The RETURN and POP instructions restore the stack to the state shown in Figure 4-9.

Example 4-14:  Frame Pointer Usage

```
TASKA:
    ...
    PUSH W0 ; Push parameter 1
    PUSH W1 ; Push parameter 2
    PUSH W2 ; Push parameter 3
    CALL COMPUTE ; Call COMPUTE function
    POP W2 ; Pop parameter 3
    POP W1 ; Pop parameter 2
    POP W0 ; Pop parameter 1
    ...

COMPUTE:
    LNK #4 ; Stack FP, allocate 4 bytes for local variables
    ...
    ULNK ; Free allocated memory, restore original FP
    RETURN ; Return to TASKA
```

Figure 4-9:  Stack at the Beginning of Example 4-14
Figure 4-10: Stack After “CALL COMPUTE” Executes

Figure 4-11: Stack After “LNK #4” Executes

4.7.3 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. Should the effective address be greater than the contents of SPLIM, then a stack error trap is 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).

Refer to the specific device family reference manual for more information on the stack error trap.
Section 4. Instruction Set Details

4.7.4 Stack Pointer Underflow

The stack is initialized to 0x0800 during Reset (0x1000 for PIC24E and dsPIC33E devices). A stack error trap will be initiated should the Stack Pointer address ever be less than 0x0800 (0x1000 for PIC24E and dsPIC33E devices).

Note: Locations in data space between 0x0000 and 0x07FF (0x0FFF for PIC24E and dsPIC33E devices) are, in general, reserved for core and peripheral Special Function Registers (SFRs).

4.7.5 Stack Frame Active (SFA) Control (dsPIC33E and PIC24E Devices)

W15 is never subject to paging and is therefore restricted to address range 0x000000 to 0x00FFFFFF. However, the Stack Frame Pointer (W14) for any user software function is only dedicated to that function when a stack frame addressed by W14 is active (i.e., after a LNK instruction). Therefore, it is desirable to have the ability to dynamically switch W14 between use as a general purpose W register, and use as a Stack Frame Pointer. The SFA Status bit (CORCON<2>) achieves this function without additional software overhead.

When the SFA bit is clear, W14 may be used with any page register. When SFA is set, W14 is not subject to paging and is locked into the same address range as W15 (0x000000 to 0x00FFFFFF). Operation of the SFA register lock is as follows:

- The LNK instruction sets SFA (and creates a stack frame)
- The ULNK instruction clears SFA (and deletes the stack frame)
- The CALL, CALL.L, and RCALL instructions also stack the SFA bit (placing it in the LSb of the stacked PC), and clear the SFA bit after the stacking operation is complete. The called procedure is now free to either use W14 as a general purpose register, or create another stack frame using the LNK instruction.
- The RETURN, RETLW and RETFIE instructions all restore the SFA bit from its previously stacked value

The SFA bit is a read-only bit. It can only be set by execution of the LNK instruction, and cleared by the ULNK, CALL, CALL.L, and RCALL instructions.

Note: In dsPIC33E and PIC24E devices, the SFA bit is stacked instead of PC<0>. 
4.8 CONDITIONAL BRANCH INSTRUCTIONS

Conditional branch instructions are used to direct program flow, based on the contents of the STATUS register. These instructions are generally used in conjunction with a Compare class instruction, but they may be employed effectively after any operation that modifies the STATUS register.

The compare instructions CP, CP0, and CPB, perform a subtract operation (minuend – subtrahend), but do not actually store the result of the subtraction. Instead, compare instructions just update the flags in the STATUS register, such that an ensuing conditional branch instruction may change program flow by testing the contents of the updated STATUS register. If the result of the STATUS register test is true, the branch is taken. If the result of the STATUS register test is false, the branch is not taken.

The conditional branch instructions supported by the dsPIC30F, dsPIC33F, and dsPIC33E devices are shown in Table 4-8. This table identifies the condition in the STATUS register which must be true for the branch to be taken. In some cases, just a single bit is tested (as in BRA C), while in other cases, a complex logic operation is performed (as in BRA GT). For dsPIC30F, dsPIC33F, and dsPIC33E devices, it is worth noting that both signed and unsigned conditional tests are supported, and that support is provided for DSP algorithms with the OA, OB, SA and SB condition mnemonics.

<table>
<thead>
<tr>
<th>Condition Mnemonic(1)</th>
<th>Description</th>
<th>Status Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Carry (not Borrow)</td>
<td>C</td>
</tr>
<tr>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>(NZ&amp;&amp;OV)</td>
</tr>
<tr>
<td>GEU(2)</td>
<td>Unsigned greater than or equal</td>
<td>C</td>
</tr>
<tr>
<td>GT</td>
<td>Signed greater than</td>
<td>(Z&amp;&amp;NZ&amp;&amp;OV)</td>
</tr>
<tr>
<td>GTU</td>
<td>Unsigned greater than</td>
<td>C&amp;&amp;Z</td>
</tr>
<tr>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Z</td>
</tr>
<tr>
<td>LEU</td>
<td>Unsigned less than or equal</td>
<td>C</td>
</tr>
<tr>
<td>LT</td>
<td>Signed less than</td>
<td>(NZ&amp;&amp;OV)</td>
</tr>
<tr>
<td>LTU(3)</td>
<td>Unsigned less than</td>
<td>C</td>
</tr>
<tr>
<td>N</td>
<td>Negative</td>
<td>N</td>
</tr>
<tr>
<td>NC</td>
<td>Not Carry (Borrow)</td>
<td>C</td>
</tr>
<tr>
<td>NN</td>
<td>Not Negative</td>
<td>N</td>
</tr>
<tr>
<td>NOV</td>
<td>Not Overflow</td>
<td>OV</td>
</tr>
<tr>
<td>NZ</td>
<td>Not Zero</td>
<td>Z</td>
</tr>
<tr>
<td>OA(4)</td>
<td>Accumulator A overflow</td>
<td>OA</td>
</tr>
<tr>
<td>OB(4)</td>
<td>Accumulator B overflow</td>
<td>OB</td>
</tr>
<tr>
<td>OV</td>
<td>Overflow</td>
<td>OV</td>
</tr>
<tr>
<td>SA(4)</td>
<td>Accumulator A saturate</td>
<td>SA</td>
</tr>
<tr>
<td>SB(4)</td>
<td>Accumulator B saturate</td>
<td>SB</td>
</tr>
<tr>
<td>Z</td>
<td>Zero</td>
<td>Z</td>
</tr>
</tbody>
</table>

Note: The “Compare and Skip” instructions (CPBEQ, CPBGT, CPBLT, CPBNE, CPSEQ, CPSGT, CPSLT, and CPSNE) do not modify the STATUS register.
4.9  Z STATUS BIT

The Z Status bit is a special zero Status bit that is useful for extended precision arithmetic. The Z bit functions like a normal Z flag for all instructions, except those that use the Carry/Borrow input (ADDC, CPB, SUBB and SUBBR). For the ADDC, CPB, SUBB and SUBBR instructions, the Z bit can only be cleared and never set. If the result of one of these instructions is non-zero, the Z bit will be cleared and will remain cleared, regardless of the result of subsequent ADDC, CPB, SUBB or SUBBR operations. This allows the Z bit to be used for performing a simple zero check on the result of a series of extended precision operations.

A sequence of instructions working on multi-precision data (starting with an instruction with no Carry/Borrow input), will automatically logically AND the successive results of the zero test. All results must be zero for the Z flag to remain set at the end of the sequence of operations. If the result of the ADDC, CPB, SUBB or SUBBR instruction is non-zero, the Z bit will be cleared and remain cleared for all subsequent ADDC, CPB, SUBB or SUBBR instructions. Example 4-15 shows how the Z bit operates for a 32-bit addition. It shows how the Z bit is affected for a 32-bit addition implemented with an ADD/ADDC instruction sequence. The first example generates a zero result for only the most significant word, and the second example generates a zero result for both the least significant word and most significant word.

Example 4-15: ‘Z’ Status bit Operation for 32-bit Addition

; Add two doubles (W0:W1 and W2:W3)
; Store the result in W5:W4
ADD  W0, W2, W4 ; Add LSWord and store to W4
ADDC W1, W3, W5 ; Add MSWord and store to W5

Before 32-bit Addition (zero result for the most significant word):

W0 = 0x2342
W1 = 0xFFF0
W2 = 0x39AA
W3 = 0x0010
W4 = 0x0000
W5 = 0x0000
SR = 0x0000

After 32-bit Addition:

W0 = 0x2342
W1 = 0xFFF0
W2 = 0x39AA
W3 = 0x0010
W4 = 0x5CEC
W5 = 0x0000
SR = 0x0201 (DC,C=1)

Before 32-bit Addition (zero result for the least significant word and most significant word):

W0 = 0xB76E
W1 = 0xFB7B
W2 = 0x4892
W3 = 0x0484
W4 = 0x0000
W5 = 0x0000
SR = 0x0000

After 32-bit Addition:

W0 = 0xB76E
W1 = 0xFB7B
W2 = 0x4892
W3 = 0x0485
W4 = 0x0000
W5 = 0x0000
SR = 0x0103 (DC,Z,C=1)
4.10 ASSIGNED WORKING REGISTER USAGE

The 16 working registers of the 16-bit MCU and DSC devices provide a large register set for efficient code generation and algorithm implementation. In an effort to maintain an instruction set that provides advanced capability, a stable run-time environment and backwards compatibility with earlier Microchip processor cores, some working registers have a preassigned usage. Table 4-9 summarizes these working register assignments. For the dsPIC30F, dsPIC33F, and dsPIC33E, additional details are provided in subsections Section 4.10.1 “Implied DSP Operands (dsPIC30F, dsPIC33F and dsPIC33E Devices)” through Section 4.10.3 “PIC® Microcontroller Compatibility”.

Table 4-9: Special Working Register Assignments

<table>
<thead>
<tr>
<th>Register</th>
<th>Special Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>Default WREG, Divide Quotient</td>
</tr>
<tr>
<td>W1</td>
<td>Divide Remainder</td>
</tr>
<tr>
<td>W2</td>
<td>“MUL f” Product least significant word</td>
</tr>
<tr>
<td>W3</td>
<td>“MUL f” Product most significant word</td>
</tr>
<tr>
<td>W4</td>
<td>MAC Operand(1)</td>
</tr>
<tr>
<td>W5</td>
<td>MAC Operand(1)</td>
</tr>
<tr>
<td>W6</td>
<td>MAC Operand(1)</td>
</tr>
<tr>
<td>W7</td>
<td>MAC Operand(1)</td>
</tr>
<tr>
<td>W8</td>
<td>MAC Prefetch Address (X Memory)(1)</td>
</tr>
<tr>
<td>W9</td>
<td>MAC Prefetch Address (X Memory)(1)</td>
</tr>
<tr>
<td>W10</td>
<td>MAC Prefetch Address (Y Memory)(1)</td>
</tr>
<tr>
<td>W11</td>
<td>MAC Prefetch Address (Y Memory)(1)</td>
</tr>
<tr>
<td>W12</td>
<td>MAC Prefetch Offset(1)</td>
</tr>
<tr>
<td>W13</td>
<td>MAC Write Back Destination(1)</td>
</tr>
<tr>
<td>W14</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>W15</td>
<td>Stack Pointer</td>
</tr>
</tbody>
</table>

Note 1: This assignment is only applicable in dsPIC30F, dsPIC33F, and dsPIC33E devices.

4.10.1 Implied DSP Operands (dsPIC30F, dsPIC33F and dsPIC33E Devices)

To assist instruction encoding and maintain uniformity among the DSP class of instructions, some working registers have pre-assigned functionality. For all DSP instructions which have prefetch ability, the following 10 register assignments must be adhered to:

- W4-W7 are used for arithmetic operands
- W8-W11 are used for prefetch addresses (pointers)
- W12 is used for the prefetch register offset index
- W13 is used for the accumulator Write Back destination

These restrictions only apply to the DSP MAC class of instructions, which utilize working registers and have prefetch ability (described in Section 4.15 “DSP Accumulator Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices”)’). These instructions are CLR, ED, EDAC, MAC, MOVAC, MPY, MPY, N and MSC.

In dsPIC33E devices, mixed-sign DSP multiplication operations are supported without the need to dynamically modify the US<1:0> bits. In this mode (US<1:0> = ‘10’), each input operand is treated as unsigned or signed based on which register is being used for that operand. W4 and W6 are always unsigned operand, whereas W5 and W7 are always signed operands. This feature can be used to efficiently execute extended-precision DSP multiplications.

The DSP Accumulator class of instructions (described in Section 4.15 “DSP Accumulator Instructions (dsPIC30F, dsPIC33F and dsPIC33E Devices”)’) are not required to follow the working register assignments in Table 4-9 and may freely use any working register when required.
4.10.2 Implied Frame and Stack Pointer

To accommodate software stack usage, W14 is the implied Frame Pointer (used by the LNK and ULNK instructions) and W15 is the implied Stack Pointer (used by the CALL, LNK, POP, PUSH, RCALL, RETFIE, RETLW, RETURN, TRAP and ULNK instructions). Even though W14 and W15 have this implied usage, they may still be used as generic operands in any instruction, with the exceptions outlined in Section 4.10.1 “Implied DSP Operands (dsPIC30F, dsPIC33F and dsPIC33E Devices)”. If W14 and W15 must be used for other purposes (it is strongly advised that they remain reserved for the Frame and Stack Pointer), extreme care must be taken such that the run-time environment is not corrupted.

4.10.3 PIC® Microcontroller Compatibility

4.10.3.1 DEFAULT WORKING REGISTER WREG

To ease the migration path for users of the Microchip 8-bit PIC MCU families, the 16-bit MCU and DSC devices have matched the functionality of the PIC MCU instruction sets as closely as possible. One major difference between the 16-bit MCU and DSC and the 8-bit PIC MCU processors is the number of working registers provided. The 8-bit PIC MCU families only provide one 8-bit working register, while the 16-bit MCU and DSC families provide sixteen, 16-bit working registers. To accommodate for the one working register of the 8-bit PIC MCU, the 16-bit MCU and DSC device instruction set has designated one working register to be the default working register for all legacy file register instructions. The default working register is set to W0, and it is used by all instructions which use file register addressing.

Additionally, the syntax used by the 16-bit MCU and DSC device assembler to specify the default working register is similar to that used by the 8-bit PIC MCU assembler. As shown in the detailed instruction descriptions in Section 5. “Instruction Descriptions”, “WREG” must be used to specify the default working register. Example 4-16 shows several instructions that use WREG.

Example 4-16: Using the Default Working Register WREG

```
ADD    RAM100 ; add RAM100 and WREG, store in RAM100
ASR    RAM100, WREG ; shift RAM100 right, store in WREG
CLR.B  WREG ; clear the WREG LS Byte
DEC    RAM100, WREG ; decrement RAM100, store in WREG
MOVL   WREG, RAM100 ; move WREG to RAM100
SETM   WREG ; set all bits in the WREG
XOR    RAM100 ; XOR RAM100 and WREG, store in RAM100
```

4.10.3.2 PRODH:PRODL REGISTER PAIR

Another significant difference between the Microchip 8-bit PIC MCU and 16-bit MCU and DSC architectures is the multiplier. Some PIC MCU families support an 8-bit x 8-bit multiplier, which places the multiply product in the PRODH:PRODL register pair. The 16-bit MCU and DSC devices have a 17-bit x 17-bit multiplier, which may place the result into any two successive working registers (starting with an even register), or an accumulator.

Despite this architectural difference, the 16-bit MCU and DSC devices still support the legacy file register multiply instruction (MULWF) with the “MUL{.B} f” instruction (described on page 303). Supporting the legacy MULWF instruction has been accomplished by mapping the PRODH:PRODL registers to the working register pair W3:W2. This means that when “MUL{.B} f” is executed in Word mode, the multiply generates a 32-bit product which is stored in W3:W2, where W3 has the most significant word of the product and W2 has the least significant word of the product. When “MUL{.B} f” is executed in Byte mode, the 16-bit product is stored in W2, and W3 is unaffected. Examples of this instruction are shown in Example 4-17.
Example 4-17:  Unsigned f and WREG Multiply (Legacy \texttt{MULWF} Instruction)

```assembly
MUL.B 0x100 ; (0x100)\texttt{\*}WREG (byte mode), store to W2
```

Before Instruction:
- W0 (WREG) = 0x7705
- W2 = 0x1235
- W3 = 0x1000
- Data Memory 0x0100 = 0x1255

After Instruction:
- W0 (WREG) = 0x7705
- W2 = 0x01A9
- W3 = 0x1000
- Data Memory 0x0100 = 0x1255

```assembly
MUL 0x100 ; (0x100)\texttt{\*}WREG (word mode), store to W3:W2
```

Before Instruction:
- W0 (WREG) = 0x7705
- W2 = 0x1235
- W3 = 0x1000
- Data Memory 0x0100 = 0x1255

After Instruction:
- W0 (WREG) = 0x7705
- W2 = 0xDEA9
- W3 = 0x0885
- Data Memory 0x0100 = 0x1255

4.10.3.3 MOVING DATA WITH WREG

The "\texttt{MOV\{.B\} f \{,WREG\}}" instruction (described on page 279) and "\texttt{MOV\{.B\} WREG, f}" instruction (described on page 280) allow for byte or word data to be moved between file register memory and the WREG (working register W0). These instructions provide equivalent functionality to the legacy Microchip PIC MCU \texttt{MOVF} and \texttt{MOVWF} instructions.

The "\texttt{MOV\{.B\} f \{,WREG\}}" and "\texttt{MOV\{.B\} WREG, f}" instructions are the only \texttt{MOV} instructions which support moves of byte data to and from file register memory. Example 4-18 shows several \texttt{MOV} instruction examples using the WREG.

**Note:** When moving word data between file register memory and the working register array, the "\texttt{MOV Wns, f}" and "\texttt{MOV f, Wnd}" instructions allow any working register (W0:W15) to be used as the source or destination register, not just WREG.

Example 4-18:  Moving Data with WREG

```assembly
MOV.B 0x1001, WREG ; move the byte stored at location 0x1001 to W0
MOV 0x1000, WREG ; move the word stored at location 0x1000 to W0
MOV.B WREG, TBLPAG ; move the byte stored at W0 to the TBLPAG register
MOV WREG, 0x804 ; move the word stored at W0 to location 0x804
```
4.11 DSP DATA FORMATS (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

4.11.1 Integer and Fractional Data

The dsPIC30F, dsPIC33F, and dsPIC33E devices support both integer and fractional data types. Integer data is inherently represented as a signed two's complement value, where the Most Significant bit is defined as a sign bit. Generally speaking, the range of an N-bit two's complement integer is \(-2^{N-1}\) to \(2^{N-1} - 1\). For a 16-bit integer, the data range is \(-32768\) (0x8000) to \(32767\) (0x7FFF), including '0'. For a 32-bit integer, the data range is \(-2,147,483,648\) (0x8000 0000) to \(2,147,483,647\) (0x7FFF FFFF).

Fractional data is represented as a two's complement number, where the Most Significant bit is defined as a sign bit, and the radix point is implied to lie just after the sign bit. This format is commonly referred to as 1.15 (or Q15) format, where 1 is the number of bits used to represent the integer portion of the number, and 15 is the number of bits used to represent the fractional portion. The range of an N-bit two's complement fraction with this implied radix point is \(-1.0\) to \((1 - 2^{-15})\). For a 16-bit fraction, the 1.15 data range is \(-1.0\) (0x8000) to \(0.999969482\) (0x7FFF), including 0.0 and it has a precision of \(3.05176 \times 10^{-5}\). In Normal Saturation mode, the 32-bit accumulators use a 1.31 format, which enhances the precision to \(4.6566 \times 10^{-10}\).

The dynamic range of the accumulators can be expanded by using the 8 bits of the Upper Accumulator register (ACCxU) as guard bits. Guard bits are used if the value stored in the accumulator overflows beyond the 32nd bit, and they are useful for implementing DSP algorithms. This mode is enabled when the ACCSAT bit (CORCON<4>) is set to ‘1’ and it expands the accumulators to 40 bits. The guard bits are also used when the accumulator saturation is disabled. The accumulators then support an integer range of \(-5.498 \times 10^{11}\) (0x80 0000 0000) to \(5.498 \times 10^{11}\) (0x7F FFFF FFFF). In Fractional mode, the guard bits of the accumulator do not modify the location of the radix point and the 40-bit accumulators use a 9.31 fractional format. Note that all fractional operation results are stored in the 40-bit Accumulator, justified with a 1.31 radix point. As in Integer mode, the guard bits merely increase the dynamic range of the accumulator. 9.31 fractions have a range of \(-256.0\) (0x80 0000 0000) to \((256.0 - 4.65661 \times 10^{-10})\) (0x7F FFFF FFFF). Table 4-10 identifies the range and precision of integers and fractions on the dsPIC30F/33F/33E devices for 16-bit, 32-bit and 40-bit registers.

It should be noted that, with the exception of DSP multiplies, the ALU operates identically on integer and fractional data. Namely, an addition of two integers will yield the same result (binary number) as the addition of two fractional numbers. The only difference is how the result is interpreted by the user. However, multiplies performed by DSP operations are different. In these instructions, data format selection is made by the IF bit (CORCON<0>), and it must be set accordingly (‘0’ for Fractional mode, ‘1’ for Integer mode). This is required because of the implied radix point used by dsPIC30F/33F/33E fractional numbers. In Integer mode, multiplying two 16-bit integers produces a 32-bit integer result. However, multiplying two 1.15 values generates a 2.30 result. Since the dsPIC30F, dsPIC33F, and dsPIC33E devices use a 1.31 format for the accumulators, a DSP multiply in Fractional mode also includes a left shift of one bit to keep the radix point properly aligned. This feature reduces the resolution of the DSP multiplier to \(2^{-30}\), but has no other effect on the computation (e.g., \(0.5 \times 0.5 = 0.25\)).

Table 4-10: dsPIC30F/33F/33E Data Ranges

<table>
<thead>
<tr>
<th>Register Size</th>
<th>Integer Range</th>
<th>Fraction Range</th>
<th>Fraction Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td>-32768 to 32767</td>
<td>-1.0 to (1.0 - 2^{-15}) ()</td>
<td>(3.052 \times 10^{-5})</td>
</tr>
<tr>
<td>32-bit</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>-1.0 to (1.0 - 2^{-31}) ()</td>
<td>(4.657 \times 10^{-10})</td>
</tr>
<tr>
<td>40-bit</td>
<td>-549,755,813,888 to 549,755,813,887</td>
<td>-256.0 to (256.0 - 2^{-31}) ()</td>
<td>(4.657 \times 10^{-10})</td>
</tr>
</tbody>
</table>
### 4.11.2 Integer and Fractional Data Representation

Having a working knowledge of how integer and fractional data are represented on the dsPIC30F, dsPIC33F, and dsPIC33E is fundamental to working with the device. Both integer and fractional data treat the Most Significant bit as a sign bit, and the binary exponent decreases by one as the bit position advances toward the Least Significant bit. The binary exponent for an N-bit integer starts at \((N-1)\) for the Most Significant bit, and ends at \('0'\) for the Least Significant bit. For an N-bit fraction, the binary exponent starts at \('0'\) for the Most Significant bit, and ends at \((1-N)\) for the Least Significant bit (as shown in Figure 4-12 for a positive value and in Figure 4-13 for a negative value).

Conversion between integer and fractional representations can be performed using simple division and multiplication. To go from an N-bit integer to a fraction, divide the integer value by \(2^{N-1}\). Similarly, to convert an N-bit fraction to an integer, multiply the fractional value by \(2^{N-1}\).

#### Figure 4-12: Different Representations of 0x4001

<table>
<thead>
<tr>
<th>Integer:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) 10 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1\</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-2^{15}) 2^{14} 2^{13} 2^{12} \ldots \ldots\</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0x4001 = 2^{14} + 2^0 = 16384 + 1 = 16385)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.15 Fractional:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0) 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1\</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-2^0 \cdot 2^{-1} 2^{-2} 2^{-3} \ldots \ldots\</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0x4001 = 2^{-1} + 2^{-15} = 0.5 + .000030518 = 0.500030518)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Figure 4-13: Different Representations of 0xC002

| Integer:                                      |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----------------------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0\          |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| \(-2^{15}\) 2^{14} 2^{13} 2^{12} \ldots \ldots\ |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| \(0xC002 = -2^{15} + 2^{14} + 2^1 = -32768 + 16384 + 2 = -16382\) |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1.15 Fractional:                              |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0\          |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| \(-2^0 \cdot 2^{-1} 2^{-2} 2^{-3} \ldots \ldots\ |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| \(0xC002 = -2^0 + 2^{-1} + 2^{-14} = -1.0 + 0.5 + 0.000061035 = -0.499938965\) |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
4.12 ACCUMULATOR USAGE (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

Accumulators A and B are utilized by DSP instructions to perform mathematical and shifting operations. Since the accumulators are 40 bits wide and the X and Y data paths are only 16 bits, the method to load and store the accumulators must be understood.

Item A in Figure 4-14 shows that each 40-bit Accumulator (ACCA and ACCB) consists of an 8-bit Upper register (ACCxU), a 16-bit High register (ACCxH) and a 16-bit Low register (ACCxL). To address the bus alignment requirement and provide the ability for 1.31 math, ACCxH is used as a destination register for loading the accumulator (with the LAC instruction), and also as a source register for storing the accumulator (with the SAC.R instruction). This is represented by Item B, Figure 4-14, where the upper and lower portions of the accumulator are shaded. In reality, during accumulator loads, ACCxL is zero backfilled and ACCxU is sign-extended to represent the sign of the value loaded in ACCxH.

When Normal (31-bit) Saturation is enabled, DSP operations (such as ADD, MAC, MSC, etc.) utilize solely ACCxH:ACCxL (Item C in Figure 4-14) and ACCxU is only used to maintain the sign of the value stored in ACCxH:ACCxL. For instance, when a MPY instruction is executed, the result is stored in ACCxH:ACCxL, and the sign of the result is extended through ACCxU.

When Super Saturation is enabled, or when saturation is disabled, all registers of the accumulator may be used (Item D in Figure 4-14) and the results of DSP operations are stored in ACCxU:ACCxH:ACCxL. The benefit of ACCxU is that it increases the dynamic range of the accumulator, as described in Section 4.11.1 “Integer and Fractional Data”. Refer to Table 4-10 to see the range of values which may be stored in the accumulator when in Normal and Super Saturation modes.

Figure 4-14: Accumulator Alignment and Usage

A) 40-bit Accumulator consists of ACCxU:ACCxH:ACCxL
B) Load and Store operations
C) Operations in Normal Saturation mode
D) Operations in Super Saturation mode or with saturation disabled
4.13 ACCUMULATOR ACCESS (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The six registers of Accumulator A and Accumulator B are memory mapped like any other Special Function Register. This feature allows them to be accessed with file register or indirect addressing, using any instruction which supports such addressing. However, it is recommended that the DSP instructions LAC, SAC and SAC.R be used to load and store the accumulators, since they provide sign-extension, shifting and rounding capabilities. LAC, SAC and SAC.R instruction details are provided in Section 5. “Instruction Descriptions”.

Note 1: For convenience, ACCAU and ACCBU are sign-extended to 16 bits. This provides the flexibility to access these registers using either Byte or Word mode (when file register or indirect addressing is used).

2: The OA, OB, SA or SB bit cannot be set by writing overflowed values to the memory mapped accumulators using MOV instructions, as these status bits are only affected by DSP operations.

4.14 DSP MAC INSTRUCTIONS (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The DSP Multiply and Accumulate (MAC) operations are a special suite of instructions which provide the most efficient use of the dsPIC30F, dsPIC33F, and dsPIC33E architectures. The DSP MAC instructions, shown in Table 4-11, utilize both the X and Y data paths of the CPU core, which enables these instructions to perform the following operations all in one cycle:

- two reads from data memory using prefetch working registers (MAC Prefetches)
- two updates to prefetch working registers (MAC Prefetch Register Updates)
- one mathematical operation with an accumulator (MAC Operations)

In addition, four of the ten DSP MAC instructions are also capable of performing an operation with one accumulator, while storing out the rounded contents of the alternate accumulator. This feature is called accumulator Write Back (WB) and it provides flexibility for the software developer. For instance, the accumulator WB may be used to run two algorithms concurrently, or efficiently process complex numbers, among other things.

### Table 4-11: DSP MAC Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Accumulator WB?</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>Clear accumulator</td>
<td>Yes</td>
</tr>
<tr>
<td>ED</td>
<td>Euclidean distance (no accumulate)</td>
<td>No</td>
</tr>
<tr>
<td>EDAC</td>
<td>Euclidean distance</td>
<td>No</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply and accumulate</td>
<td>Yes</td>
</tr>
<tr>
<td>MAC</td>
<td>Square and accumulate</td>
<td>No</td>
</tr>
<tr>
<td>MOVSA</td>
<td>Move from X and Y bus</td>
<td>Yes</td>
</tr>
<tr>
<td>MPY</td>
<td>Multiply to accumulator</td>
<td>No</td>
</tr>
<tr>
<td>MPY</td>
<td>Square to accumulator</td>
<td>No</td>
</tr>
<tr>
<td>MPY.N</td>
<td>Negative multiply to accumulator</td>
<td>No</td>
</tr>
<tr>
<td>MSC</td>
<td>Multiply and subtract</td>
<td>Yes</td>
</tr>
</tbody>
</table>

4.14.1 MAC Prefetches

Prefetches (or data reads) are made using the effective address stored in the working register. The two prefetches from data memory must be specified using the working register assignments shown in Table 4-9. One read must occur from the X data bus using W8 or W9, and one read must occur from the Y data bus using W10 or W11. The allowed destination registers for both prefetches are W4-W7.

As shown in Table 4-3, one special Addressing mode exists for the MAC class of instructions. This mode is the Register Offset Addressing mode and utilizes W12. In this mode, the prefetch is made using the effective address of the specified working register, plus the 16-bit signed value stored in W12. Register Offset Addressing may only be used in the X space with W9, and in the Y-space with W11.
4.14.2 MAC Prefetch Register Updates

After the MAC prefetches are made, the effective address stored in each prefetch working register may be modified. This feature enables efficient single-cycle processing for data stored sequentially in X and Y memory. Since all DSP instructions execute in Word mode, only even numbered updates may be made to the effective address stored in the working register. Allowable address modifications to each prefetch register are -6, -4, -2, 0 (no update), +2, +4 and +6. This means that effective address updates may be made up to 3 words in either direction.

When the Register Offset Addressing mode is used, no update is made to the base prefetch register (W9 or W11), or the offset register (W12).

4.14.3 MAC Operations

The mathematical operations performed by the MAC class of DSP instructions center around multiplying the contents of two working registers and either adding or storing the result to either Accumulator A or Accumulator B. This is the operation of the MAC, MPY, MPY.N and MSC instructions. Table 4-9 shows that W4-W7 must be used for data source operands in the MAC class of instructions. W4-W7 may be combined in any fashion, and when the same working register is specified for both operands, a square or square and accumulate operation is performed.

For the ED and EDAC instructions, the same multiplicand operand must be specified by the instruction, because this is the definition of the Euclidean Distance operation. Another unique feature about this instruction is that the values prefetched from X and Y memory are not actually stored in W4-W7. Instead, only the difference of the prefetched data words is stored in W4-W7.

The two remaining MAC class instructions, CLR and MOVSAC, are useful for initiating or completing a series of MAC or EDAC instructions and do not use the multiplier. CLR has the ability to clear Accumulator A or B, prefetch two values from data memory and store the contents of the other accumulator. Similarly, MOVSAC has the ability to prefetch two values from data memory and store the contents of either accumulator.

4.14.4 MAC Write Back

The write back ability of the MAC class of DSP instructions facilitates efficient processing of algorithms. This feature allows one mathematical operation to be performed with one accumulator, and the rounded contents of the other accumulator to be stored in the same cycle. As indicated in Table 4-9, register W13 is assigned for performing the write back, and two Addressing modes are supported: Direct and Indirect with Post-Increment.

The CLR, MOVSAC and MSC instructions support accumulator Write Back, while the ED, EDAC, MPY and MPY.N instructions do not support accumulator Write Back. The MAC instruction, which multiplies two working registers which are not the same, also supports accumulator Write Back. However, the square and accumulate MAC instruction does not support accumulator Write Back (see Table 4-11).

4.14.5 MAC Syntax

The syntax of the MAC class of instructions can have several formats, which depend on the instruction type and the operation it is performing, with respect to prefetches and accumulator Write Back. With the exception of the CLR and MOVSAC instructions, all MAC class instructions must specify a target accumulator along with two multiplicands, as shown in Example 4-19.

Example 4-19: Base MAC Syntax

```
; MAC with no prefetch
MAC W4*W5, A

; MAC with no prefetch
MAC W7*W7, B

Multiply W7*W7, Accumulate to ACCB
```
If a prefetch is used in the instruction, the assembler is capable of discriminating between the X or Y data prefetch based on the register used for the effective address. \([W8]\) or \([W9]\) specifies the X prefetch and \([W10]\) or \([W11]\) specifies the Y prefetch. Brackets around the working register are required in the syntax, and they designate that indirect addressing is used to perform the prefetch. When address modification is used, it must be specified using a minus-equals or plus-equals “C”-like syntax (i.e., “[W8] – = 2” or “[W8] + = 6”). When Register Offset Addressing is used for the prefetch, W12 is placed inside the brackets ([W9 + W12] for X prefetches and [W11 + W12] for Y prefetches). Each prefetch operation must also specify a prefetch destination register (W4-W7). In the instruction syntax, the destination register appears before the prefetch register. Legal forms of prefetch are shown in Example 4-20.

**Example 4-20: MAC Prefetch Syntax**

<table>
<thead>
<tr>
<th>; MAC with X only prefetch</th>
<th>MAC W5*W6, A, [W8]+=2, W5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACCA=ACCA+W5*W6</td>
</tr>
<tr>
<td></td>
<td>X([W8]+=2)→W5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>; MAC with Y only prefetch</th>
<th>MAC W5*W5, B, [W11+W12], W5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACCB=ACCB+W5*W5</td>
</tr>
<tr>
<td></td>
<td>Y([W11+W12])→W5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>; MAC with X/Y prefetch</th>
<th>MAC W6*W7, B, [W9], W6, [W10]+=4, W7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ACCB=ACCB+W6*W7</td>
</tr>
<tr>
<td></td>
<td>X([W9])→W6</td>
</tr>
<tr>
<td></td>
<td>Y([W10]+=4)→W7</td>
</tr>
</tbody>
</table>

If an accumulator Write Back is used in the instruction, it is specified last. The Write Back must use the W13 register, and allowable forms for the Write Back are “W13” for direct addressing and “[W13] + = 2” for indirect addressing with post-increment. By definition, the accumulator not used in the mathematical operation is stored, so the Write Back accumulator is not specified in the instruction. Legal forms of accumulator Write Back (WB) are shown in Example 4-21.
Putting it all together, an MSC instruction which performs two prefetches and a write back is shown in Example 4-22.

Example 4-22: MSC Instruction with Two Prefetches and Accumulator Write Back

; MSC with X/Y prefetch, indirect WB of ACCA

ACCB=ACCB-W6*W7
X([W8]+=2)→W6
Y([W10]=-6)→W7
Y([W10]+=2)→W4
ACCA→W13

ACCB=ACCB+W4*W5

ACCA=ACCA+W4*W5

Y([W10]+=2)→W4
ACCA→W13

ACCB=W13
4.15 DSP ACCUMULATOR INSTRUCTIONS (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The DSP Accumulator instructions do not have prefetch or accumulator WB ability, but they do provide the ability to add, negate, shift, load and store the contents of either 40-bit Accumulator. In addition, the ADD and SUB instructions allow the two accumulators to be added or subtracted from each other. DSP Accumulator instructions are shown in Table 4-12 and instruction details are provided in Section 5. "Instruction Descriptions".

Table 4-12: DSP Accumulator Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Accumulator WB?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add accumulators</td>
<td>No</td>
</tr>
<tr>
<td>ADD</td>
<td>16-bit signed accumulator add</td>
<td>No</td>
</tr>
<tr>
<td>LAC</td>
<td>Load accumulator</td>
<td>No</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate accumulator</td>
<td>No</td>
</tr>
<tr>
<td>SAC</td>
<td>Store accumulator</td>
<td>No</td>
</tr>
<tr>
<td>SAC,R</td>
<td>Store rounded accumulator</td>
<td>No</td>
</tr>
<tr>
<td>SFTAC</td>
<td>Arithmetic shift accumulator by Literal</td>
<td>No</td>
</tr>
<tr>
<td>SFTAC</td>
<td>Arithmetic shift accumulator by (Wn)</td>
<td>No</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract accumulators</td>
<td>No</td>
</tr>
</tbody>
</table>

4.16 SCALING DATA WITH THE FBCL INSTRUCTION (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

To minimize quantization errors that are associated with data processing using DSP instructions, it is important to utilize the complete numerical result of the operations. This may require scaling data up to avoid underflow (i.e., when processing data from a 12-bit ADC), or scaling data down to avoid overflow (i.e., when sending data to a 10-bit DAC). The scaling, which must be performed to minimize quantization error, depends on the dynamic range of the input data which is operated on, and the required dynamic range of the output data. At times, these conditions may be known beforehand and fixed scaling may be employed. In other cases, scaling conditions may not be fixed or known, and then dynamic scaling must be used to process data.

The FBCL instruction (Find First Bit Change Left) can efficiently be used to perform dynamic scaling, because it determines the exponent of a value. A fixed point or integer value's exponent represents the amount which the value may be shifted before overflowing. This information is valuable, because it may be used to bring the data value to "full scale", meaning that its numeric representation utilizes all the bits of the register it is stored in.

The FBCL instruction determines the exponent of a word by detecting the first bit change starting from the value's sign bit and working towards the LSB. Since the dsPIC DSC device's barrel shifter uses negative values to specify a left shift, the FBCL instruction returns the negated exponent of a value. If the value is being scaled up, this allows the ensuing shift to be performed immediately with the value returned by FBCL. Additionally, since the FBCL instruction only operates on signed quantities, FBCL produces results in the range of -15:0. When the FBCL instruction returns '0', it indicates that the value is already at full scale. When the instruction returns -15, it indicates that the value cannot be scaled (as is the case with 0x0 and 0xFFFF).

Table 4-13 shows word data with various dynamic ranges, their exponents, and the value after scaling each data to maximize the dynamic range. Example 4-23 shows how the FBCL instruction may be used for block processing.
As a practical example, assume that block processing is performed on a sequence of data with very low dynamic range stored in 1.15 fractional format. To minimize quantization errors, the data may be scaled up to prevent any quantization loss which may occur as it is processed. The FBCL instruction can be executed on the sample with the largest magnitude to determine the optimal scaling value for processing the data. Note that scaling the data up is performed by left shifting the data. This is demonstrated with the code snippet below.

**Example 4-23: Scaling with FBCL**

```assembly
; assume W0 contains the largest absolute value of the data block
; assume W4 points to the beginning of the data block
; assume the block of data contains BLOCK_SIZE words

FBCL W0, W2 ; store exponent in W2

; scale the entire data block before processing
DO #(BLOCK_SIZE-1), SCALE
  LAC [W4], A ; move the next data sample to ACCA
  SFTAC A, W2 ; shift ACCA by W2 bits
SCALE:
  SAC A, [W4++] ; store scaled input (overwrite original)

; now process the data
; (processing block goes here)
```

Table 4-13: Scaling Examples

<table>
<thead>
<tr>
<th>Word Value</th>
<th>Exponent</th>
<th>Full Scale Value (Word Value &lt;&lt; Exponent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>14</td>
<td>0x4000</td>
</tr>
<tr>
<td>0x0002</td>
<td>13</td>
<td>0x4000</td>
</tr>
<tr>
<td>0x0004</td>
<td>12</td>
<td>0x4000</td>
</tr>
<tr>
<td>0x0100</td>
<td>6</td>
<td>0x4000</td>
</tr>
<tr>
<td>0x01FF</td>
<td>6</td>
<td>0x7FC0</td>
</tr>
<tr>
<td>0x0806</td>
<td>3</td>
<td>0x4030</td>
</tr>
<tr>
<td>0x2007</td>
<td>1</td>
<td>0x400E</td>
</tr>
<tr>
<td>0x4800</td>
<td>0</td>
<td>0x4800</td>
</tr>
<tr>
<td>0x7000</td>
<td>0</td>
<td>0x7000</td>
</tr>
<tr>
<td>0x8000</td>
<td>0</td>
<td>0x8000</td>
</tr>
<tr>
<td>0x900A</td>
<td>0</td>
<td>0x900A</td>
</tr>
<tr>
<td>0xE001</td>
<td>2</td>
<td>0x8004</td>
</tr>
<tr>
<td>0xFF07</td>
<td>7</td>
<td>0x8380</td>
</tr>
</tbody>
</table>

**Note:** For the word values 0x0000 and 0xFFFF, the FBCL instruction returns -15.
4.17 NORMALIZING THE ACCUMULATOR WITH THE FBCL INSTRUCTION (dsPIC30F, dsPIC33F AND dsPIC33E DEVICES)

The process of scaling a quantized value for its maximum dynamic range is known as normalization (the data in the third column in Table 4-13 contains normalized data). Accumulator normalization is a technique used to ensure that the accumulator is properly aligned before storing data from the accumulator, and the FBCL instruction facilitates this function.

The two 40-bit accumulators each have 8 guard bits from the ACCxU register, which expands the dynamic range of the accumulators from 1.31 to 9.31, when operating in Super Saturation mode (see Section 4.11.1 “Integer and Fractional Data”). However, even in Super Saturation mode, the Store Rounded Accumulator (SAC.R) instruction only stores 16-bit data (in 1.15 format) from ACCxH, as described in Section 4.12 “Accumulator Usage (dsPIC30F, dsPIC33F and dsPIC33E Devices)”. Under certain conditions, this may pose a problem.

Proper data alignment for storing the contents of the accumulator may be achieved by scaling the accumulator down if ACCxU is in use, or scaling the accumulator up if all of the ACCxH bits are not being used. To perform such scaling, the FBCL instruction must operate on the ACCxU byte and it must operate on the ACCxH word. If a shift is required, the ALU’s 40-bit shifter is employed, using the SFTAC instruction to perform the scaling. Example 4-24 contains a code snippet for accumulator normalization.

**Example 4-24: Normalizing with FBCL**

```
; assume an operation in ACCA has just completed (SR intact)
; assume the processor is in super saturation mode
; assume ACCAH is defined to be the address of ACCAH (0x24)

MOV   #ACCAH,  W5 ; W5 points to ACCAH
BRA   OA,   FBCL_GUARD ; if overflow we right shift

FBCL_HI:
    FBCL   [W5],  W0 ; extract exponent for left shift
    BRA   SHIFT_ACC ; branch to the shift
FBCL_GUARD:
    FBCL  [++W5], W0 ; extract exponent for right shift
    ADD.B W0, #15, W0 ; adjust the sign for right shift
SHIFT_ACC:
    SFTAC A,  W0 ; shift ACCA to normalize
```
4.18 EXTENDED-PRECISION ARITHMETIC USING MIXED-SIGN MULTIPLICATIONS (dsPIC33E ONLY)

Many DSP algorithms utilize extended-precision arithmetic operations (operations with 32-bit or 64-bit operands and results) to enhance the resolution and accuracy of computations. These can be implemented using 16-bit signed or unsigned multiplications; however, this would require some additional processing and shifting of the data to obtain the correct results. To enable such extended-precision algorithms to be computed faster, dsPIC33E devices support an optional implicit mixed-sign multiplication mode, which is selected by setting US<1:0> (CORCON<13:12>) = '10'.

In this mode, mixed-sign (unsigned x signed and signed x unsigned) multiplications can be performed without the need to dynamically reconfigure the US<1:0> bits and shift data to account for the difference in operand formats. Moreover, signed x signed and unsigned x unsigned multiplications can also be performed without changing the multiplication mode. Each input operand is implicitly treated as an unsigned number if the working register being used to specify the operand is either W4 or W6. Similarly, an operand is treated as a signed number if the register used is either W5 or W7. The DSP Engine selects the type of multiplication to be performed based on the operand registers used, thereby eliminating the need for the user software to modify the US<1:0> bits.

The execution time reductions provided by the implicit mixed-sign multiplication feature is illustrated in the following code example, where the instruction cycle count for performing a 32-bit multiplication is reduced from 7 cycles to 4 cycles when the mixed-sign multiplication mode is enabled.

Example 4-25: 32-bit Signed Multiplication using Implicit Mixed-Sign Mode

<table>
<thead>
<tr>
<th>Case A</th>
<th>Mixed-Sign Multiplication Mode Not Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL.SU W5, W6, W0; Word1 (signed) x Word2 (unsigned)</td>
<td></td>
</tr>
<tr>
<td>MUL.US W4, W7, W2; Word0 (unsigned) x Word3 (signed)</td>
<td></td>
</tr>
<tr>
<td>CLR B ; Clear Accumulator B</td>
<td></td>
</tr>
<tr>
<td>ADD W1, B</td>
<td></td>
</tr>
<tr>
<td>ADD W3, B</td>
<td></td>
</tr>
<tr>
<td>SFTAC B, #15 ; Shift right by 15 bits to align for Q31 format</td>
<td></td>
</tr>
<tr>
<td>MAC W5*W7, B; Word1 (signed) x Word 3 (signed)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case B</th>
<th>Mixed-Sign Multiplication Mode Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPY W5*W6, B; Word1 (signed) x Word2 (unsigned)</td>
<td></td>
</tr>
<tr>
<td>MAC W4*W7, B; Word0 (unsigned) x Word3 (signed)</td>
<td></td>
</tr>
<tr>
<td>SFTAC B, #15 ; Shift right by 15 bits to align for Q31 format</td>
<td></td>
</tr>
<tr>
<td>MAC W5*W7, B; Word1 (signed) x Word 3 (signed)</td>
<td></td>
</tr>
</tbody>
</table>

Besides DSP instructions, MCU multiplication (MUL) instructions can also utilize Accumulator A or Accumulator B as a result destination, which enables faster extended-precision arithmetic even when not using DSP multiplication instructions such as MPY or MAC.
Section 5. Instruction Descriptions

HIGHLIGHTS

This section of the manual contains the following major topics:

5.1 Instruction Symbols........................................................................................................... 94
5.2 Instruction Encoding Field Descriptors Introduction........................................................... 94
5.3 Instruction Description Example ........................................................................................ 98
5.4 Instruction Descriptions................................................................................................. 99
## 5.1 INSTRUCTION SYMBOLS

All the symbols used in Section 5.4 “Instruction Descriptions” are listed in Table 1-2.

## 5.2 INSTRUCTION ENCODING FIELD DESCRIPTORS INTRODUCTION

All instruction encoding field descriptors used in Section 5.4 “Instruction Descriptions” are shown in Table 5-2 through Table 5-12.

### Table 5-1: Instruction Encoding Field Descriptors

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>Accumulator selection bit: 0 = ACCA; 1 = CCB</td>
</tr>
<tr>
<td>aa&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>Accumulator Write Back mode (see Table 5-12)</td>
</tr>
<tr>
<td>B</td>
<td>Byte mode selection bit: 0 = word operation; 1 = byte operation</td>
</tr>
<tr>
<td>bbbb</td>
<td>4-bit bit position select: 0000 = LSB; 1111 = MSB</td>
</tr>
<tr>
<td>D</td>
<td>Destination address bit: 0 = result stored in WREG; 1 = result stored in file register</td>
</tr>
<tr>
<td>dddd</td>
<td>Wd destination register select: 0000 = W0; 1111 = W15</td>
</tr>
<tr>
<td>f fffff fffff fffff</td>
<td>13-bit register file address (0x0000 to 0xFFFF)</td>
</tr>
<tr>
<td>fffff fffff fffff fffff</td>
<td>16-bit register file word address (implied 0 LSB) (0x0000 to 0xFFFF)</td>
</tr>
<tr>
<td>ggg</td>
<td>Register Offset Addressing mode for Ws source register (see Table 5-4)</td>
</tr>
<tr>
<td>hhh</td>
<td>Register Offset Addressing mode for Wd destination register (see Table 5-5)</td>
</tr>
<tr>
<td>iiiii&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>Prefetch X Operation (see Table 5-6)</td>
</tr>
<tr>
<td>jjjj&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>Prefetch Y Operation (see Table 5-8)</td>
</tr>
<tr>
<td>k</td>
<td>1-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>kkkk</td>
<td>4-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>kkkkk</td>
<td>6-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>kkkkk kkkkk</td>
<td>8-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>kk kkkk kkkkk</td>
<td>10-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>kk kkkk kkkk</td>
<td>14-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>kkkkk kkkk kkkk</td>
<td>16-bit literal field, constant data or expression</td>
</tr>
<tr>
<td>mmm</td>
<td>Multiplier source select with same working registers (see Table 5-10)</td>
</tr>
<tr>
<td>nnnn nnnn nnnn nnn0 nnnn</td>
<td>23-bit program address for CALL and GOTO instructions</td>
</tr>
<tr>
<td>nnnn nnnn nnnn nnnn</td>
<td>16-bit program offset field for relative branch/call instructions</td>
</tr>
<tr>
<td>ppp</td>
<td>Addressing mode for Ws source register (see Table 5-2)</td>
</tr>
<tr>
<td>qqq</td>
<td>Addressing mode for Wd destination register (see Table 5-3)</td>
</tr>
<tr>
<td>rrrrr</td>
<td>Barrel shift count</td>
</tr>
<tr>
<td>sssss</td>
<td>Ws source register select: 0000 = W0; 1111 = W15</td>
</tr>
<tr>
<td>ttttt</td>
<td>Dividend select, most significant word</td>
</tr>
<tr>
<td>vvvvv</td>
<td>Dividend select, least significant word</td>
</tr>
<tr>
<td>W</td>
<td>Double Word mode selection bit: 0 = word operation; 1 = double word operation</td>
</tr>
<tr>
<td>wwwww</td>
<td>Wb base register select: 0000 = W0; 1111 = W15</td>
</tr>
<tr>
<td>xxx&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>Prefetch X Destination (see Table 5-7)</td>
</tr>
<tr>
<td>yyyy&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>Prefetch Y Destination (see Table 5-9)</td>
</tr>
<tr>
<td>z</td>
<td>Bit test destination: 0 = C flag bit; 1 = Z flag bit</td>
</tr>
</tbody>
</table>

**Note 1:** This field is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
### Table 5-2: Addressing Modes for Ws Source Register

<table>
<thead>
<tr>
<th>PPP</th>
<th>Addressing Mode</th>
<th>Source Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Register Direct</td>
<td>Ws</td>
</tr>
<tr>
<td>001</td>
<td>Indirect</td>
<td>[Ws]</td>
</tr>
<tr>
<td>010</td>
<td>Indirect with Post-Decrement</td>
<td>[Ws--]</td>
</tr>
<tr>
<td>011</td>
<td>Indirect with Post-Increment</td>
<td>[Ws++]</td>
</tr>
<tr>
<td>100</td>
<td>Indirect with Pre-Decrement</td>
<td>[--Ws]</td>
</tr>
<tr>
<td>101</td>
<td>Indirect with Pre-Increment</td>
<td>[++Ws]</td>
</tr>
<tr>
<td>11x</td>
<td>Unused</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5-3: Addressing Modes for Wd Destination Register

<table>
<thead>
<tr>
<th>PPP</th>
<th>Addressing Mode</th>
<th>Destination Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Register Direct</td>
<td>Wd</td>
</tr>
<tr>
<td>001</td>
<td>Indirect</td>
<td>[Wd]</td>
</tr>
<tr>
<td>010</td>
<td>Indirect with Post-Decrement</td>
<td>[Wd--]</td>
</tr>
<tr>
<td>011</td>
<td>Indirect with Post-Increment</td>
<td>[Wd++]</td>
</tr>
<tr>
<td>100</td>
<td>Indirect with Pre-Decrement</td>
<td>[--Wd]</td>
</tr>
<tr>
<td>101</td>
<td>Indirect with Pre-Increment</td>
<td>[++Wd]</td>
</tr>
<tr>
<td>11x</td>
<td>Unused (an attempt to use this Addressing mode will force a <strong>RESET</strong> instruction)</td>
<td></td>
</tr>
</tbody>
</table>

### Table 5-4: Offset Addressing Modes for Ws Source Register (with Register Offset)

<table>
<thead>
<tr>
<th>ggg</th>
<th>Addressing Mode</th>
<th>Source Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Register Direct</td>
<td>Ws</td>
</tr>
<tr>
<td>001</td>
<td>Indirect</td>
<td>[Ws]</td>
</tr>
<tr>
<td>010</td>
<td>Indirect with Post-Decrement</td>
<td>[Ws--]</td>
</tr>
<tr>
<td>011</td>
<td>Indirect with Post-Increment</td>
<td>[Ws++]</td>
</tr>
<tr>
<td>100</td>
<td>Indirect with Pre-Decrement</td>
<td>[--Ws]</td>
</tr>
<tr>
<td>101</td>
<td>Indirect with Pre-Increment</td>
<td>[++Ws]</td>
</tr>
<tr>
<td>11x</td>
<td>Indirect with Register Offset</td>
<td>[Ws+Wb]</td>
</tr>
</tbody>
</table>

### Table 5-5: Offset Addressing Modes for Wd Destination Register (with Register Offset)

<table>
<thead>
<tr>
<th>hhh</th>
<th>Addressing Mode</th>
<th>Source Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>Register Direct</td>
<td>Wd</td>
</tr>
<tr>
<td>001</td>
<td>Indirect</td>
<td>[Wd]</td>
</tr>
<tr>
<td>010</td>
<td>Indirect with Post-Decrement</td>
<td>[Wd--]</td>
</tr>
<tr>
<td>011</td>
<td>Indirect with Post-Increment</td>
<td>[Wd++]</td>
</tr>
<tr>
<td>100</td>
<td>Indirect with Pre-Decrement</td>
<td>[--Wd]</td>
</tr>
<tr>
<td>101</td>
<td>Indirect with Pre-Increment</td>
<td>[++Wd]</td>
</tr>
<tr>
<td>11x</td>
<td>Indirect with Register Offset</td>
<td>[Wd+Wb]</td>
</tr>
</tbody>
</table>
### Table 5-6: X Data Space Prefetch Operation (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>(\text{xff})</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>(W_{xd} = [W_8])</td>
</tr>
<tr>
<td>0001</td>
<td>(W_{xd} = [W_8], W_8 = W_8 + 2)</td>
</tr>
<tr>
<td>0010</td>
<td>(W_{xd} = [W_8], W_8 = W_8 + 4)</td>
</tr>
<tr>
<td>0011</td>
<td>(W_{xd} = [W_8], W_8 = W_8 + 6)</td>
</tr>
<tr>
<td>0100</td>
<td>No Prefetch for X Data Space</td>
</tr>
<tr>
<td>0101</td>
<td>(W_{xd} = [W_8], W_8 = W_8 - 6)</td>
</tr>
<tr>
<td>0110</td>
<td>(W_{xd} = [W_8], W_8 = W_8 - 4)</td>
</tr>
<tr>
<td>0111</td>
<td>(W_{xd} = [W_8], W_8 = W_8 - 2)</td>
</tr>
<tr>
<td>1000</td>
<td>(W_{xd} = [W_9])</td>
</tr>
<tr>
<td>1001</td>
<td>(W_{xd} = [W_9], W_9 = W_9 + 2)</td>
</tr>
<tr>
<td>1010</td>
<td>(W_{xd} = [W_9], W_9 = W_9 + 4)</td>
</tr>
<tr>
<td>1011</td>
<td>(W_{xd} = [W_9], W_9 = W_9 + 6)</td>
</tr>
<tr>
<td>1100</td>
<td>(W_{xd} = [W_9 + W_{12}])</td>
</tr>
<tr>
<td>1101</td>
<td>(W_{xd} = [W_9], W_9 = W_9 - 6)</td>
</tr>
<tr>
<td>1110</td>
<td>(W_{xd} = [W_9], W_9 = W_9 - 4)</td>
</tr>
<tr>
<td>1111</td>
<td>(W_{xd} = [W_9], W_9 = W_9 - 2)</td>
</tr>
</tbody>
</table>

### Table 5-7: X Data Space Prefetch Destination (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>(\text{xff})</th>
<th>(W_{xd})</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(W_4)</td>
</tr>
<tr>
<td>01</td>
<td>(W_5)</td>
</tr>
<tr>
<td>10</td>
<td>(W_6)</td>
</tr>
<tr>
<td>11</td>
<td>(W_7)</td>
</tr>
</tbody>
</table>

### Table 5-8: Y Data Space Prefetch Operation (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>(\text{jjjj})</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>(W_{yd} = [W_{10}])</td>
</tr>
<tr>
<td>0001</td>
<td>(W_{yd} = [W_{10}], W_{10} = W_{10} + 2)</td>
</tr>
<tr>
<td>0010</td>
<td>(W_{yd} = [W_{10}], W_{10} = W_{10} + 4)</td>
</tr>
<tr>
<td>0011</td>
<td>(W_{yd} = [W_{10}], W_{10} = W_{10} + 6)</td>
</tr>
<tr>
<td>0100</td>
<td>No Prefetch for Y Data Space</td>
</tr>
<tr>
<td>0101</td>
<td>(W_{yd} = [W_{10}], W_{10} = W_{10} - 6)</td>
</tr>
<tr>
<td>0110</td>
<td>(W_{yd} = [W_{10}], W_{10} = W_{10} - 4)</td>
</tr>
<tr>
<td>0111</td>
<td>(W_{yd} = [W_{10}], W_{10} = W_{10} - 2)</td>
</tr>
<tr>
<td>1000</td>
<td>(W_{yd} = [W_{11}])</td>
</tr>
<tr>
<td>1001</td>
<td>(W_{yd} = [W_{11}], W_{11} = W_{11} + 2)</td>
</tr>
<tr>
<td>1010</td>
<td>(W_{yd} = [W_{11}], W_{11} = W_{11} + 4)</td>
</tr>
<tr>
<td>1011</td>
<td>(W_{yd} = [W_{11}], W_{11} = W_{11} + 6)</td>
</tr>
<tr>
<td>1100</td>
<td>(W_{yd} = [W_{11} + W_{12}])</td>
</tr>
<tr>
<td>1101</td>
<td>(W_{yd} = [W_{11}], W_{11} = W_{11} - 6)</td>
</tr>
<tr>
<td>1110</td>
<td>(W_{yd} = [W_{11}], W_{11} = W_{11} - 4)</td>
</tr>
<tr>
<td>1111</td>
<td>(W_{yd} = [W_{11}], W_{11} = W_{11} - 2)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

Table 5-9: Y Data Space Prefetch Destination (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>yy</th>
<th>Wyd</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>W4</td>
</tr>
<tr>
<td>01</td>
<td>W5</td>
</tr>
<tr>
<td>10</td>
<td>W6</td>
</tr>
<tr>
<td>11</td>
<td>W7</td>
</tr>
</tbody>
</table>

Table 5-10: MAC or MPY Source Operands (Same Working Register) (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>mm</th>
<th>Multiplicands</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>W4 * W4</td>
</tr>
<tr>
<td>01</td>
<td>W5 * W5</td>
</tr>
<tr>
<td>10</td>
<td>W6 * W6</td>
</tr>
<tr>
<td>11</td>
<td>W7 * W7</td>
</tr>
</tbody>
</table>

Table 5-11: MAC or MPY Source Operands (Different Working Register) (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>mmm</th>
<th>Multiplicands</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>W4 * W5</td>
</tr>
<tr>
<td>001</td>
<td>W4 * W6</td>
</tr>
<tr>
<td>010</td>
<td>W4 * W7</td>
</tr>
<tr>
<td>011</td>
<td>Invalid</td>
</tr>
<tr>
<td>100</td>
<td>W5 * W6</td>
</tr>
<tr>
<td>101</td>
<td>W5 * W7</td>
</tr>
<tr>
<td>110</td>
<td>W6 * W7</td>
</tr>
<tr>
<td>111</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Table 5-12: MAC Accumulator Write Back Selection (dsPIC30F, dsPIC33F and dsPIC33E)

<table>
<thead>
<tr>
<th>aa</th>
<th>Write Back Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>W13 = Other Accumulator (Direct Addressing)</td>
</tr>
<tr>
<td>01</td>
<td>[W13] + = 2 = Other Accumulator (Indirect Addressing with Post-Increment)</td>
</tr>
<tr>
<td>10</td>
<td>No Write Back</td>
</tr>
<tr>
<td>11</td>
<td>Invalid</td>
</tr>
</tbody>
</table>

Table 5-13: MOVPAG Destination Selection

<table>
<thead>
<tr>
<th>PP</th>
<th>Target Page Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>DSRPAG</td>
</tr>
<tr>
<td>01</td>
<td>DSWPAG</td>
</tr>
<tr>
<td>10</td>
<td>TBLPAG</td>
</tr>
<tr>
<td>11</td>
<td>Reserved – do not use</td>
</tr>
</tbody>
</table>

Table 5-14: Accumulator Selection

<table>
<thead>
<tr>
<th>A</th>
<th>Target Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Accumulator A</td>
</tr>
<tr>
<td>1</td>
<td>Accumulator B</td>
</tr>
</tbody>
</table>
5.3 INSTRUCTION DESCRIPTION EXAMPLE

The example description below is for the fictitious instruction FOO. The following example instruction was created to demonstrate how the table fields (syntax, operands, operation, etc.) are used to describe the instructions presented in Section 5.4 “Instruction Descriptions”.

### FOO

The Header field summarizes what the instruction does

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cells marked with an ‘X’ indicate the instruction is implemented for that device family.

**Syntax:**
The Syntax field consists of an optional label, the instruction mnemonic, any optional extensions which exist for the instruction and the operands for the instruction. Most instructions support more than one operand variant to support the various Addressing modes. In these circumstances, all possible instruction operands are listed beneath each other and are enclosed in braces.

**Operands:**
The Operands field describes the set of values which each of the operands may take. Operands may be accumulator registers, file registers, literal constants (signed or unsigned), or working registers.

**Operation:**
The Operation field summarizes the operation performed by the instruction.

**Status Affected:**
The Status Affected field describes which bits of the STATUS Register are affected by the instruction. Status bits are listed by bit position in descending order.

**Encoding:**
The Encoding field shows how the instruction is bit encoded. Individual bit fields are explained in the Description field, and complete encoding details are provided in Table 5.2.

**Description:**
The Description field describes in detail the operation performed by the instruction. A key for the encoding bits is also provided.

**Words:**
The Words field contains the number of program words that are used to store the instruction in memory.

**Cycles:**
The Cycles field contains the number of instruction cycles that are required to execute the instruction.

**Examples:**
The Examples field contains examples that demonstrate how the instruction operates. “Before” and “After” register snapshots are provided, which allow the user to clearly understand what operation the instruction performs.
### 5.4 INSTRUCTION DESCRIPTIONS

#### ADD

**Add f to WREG**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} ADD{.B} f {,WREG}
```

**Operands:**

- `f` ∈ [0 ... 8191]

**Operation:**

\[(f) + (WREG) \rightarrow \text{destination designated by } D\]

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

```
1011 0100 0BDf efee efee efee
```

**Description:**

Add the contents of the default working register WREG to the contents of the file register, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

**Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**Example 1:**

```
ADD.B RAM100 ; Add WREG to RAM100 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>CC80</td>
</tr>
<tr>
<td>RAM100</td>
<td>FFC0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>WREG</td>
<td>CC80</td>
</tr>
<tr>
<td>RAM100</td>
<td>FF40</td>
</tr>
<tr>
<td>SR</td>
<td>0005</td>
</tr>
</tbody>
</table>

**Example 2:**

```
ADD RAM200, WREG ; Add RAM200 to WREG (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>CC80</td>
</tr>
<tr>
<td>RAM200</td>
<td>FFC0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>WREG</td>
<td>CC40</td>
</tr>
<tr>
<td>RAM200</td>
<td>FFC0</td>
</tr>
<tr>
<td>SR</td>
<td>0001</td>
</tr>
</tbody>
</table>

---

© 2005-2011 Microchip Technology Inc. DS70157F-page 99
ADD  Add Literal to Wn

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
```
{label:} ADD{.B}  #lit10,   Wn
```

Operands:  
lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation:  
lit10 + (Wn) → Wn

Status Affected:  
DC, N, OV, Z, C

Encoding:  
```
1011  0000  0Bkk  kkkk  kkkk  dddd
```

Description:  
Add the 10-bit unsigned literal operand to the contents of the working register Wn, and place the result back into the working register Wn.

The ’B’ bit selects byte or word operation (’0’ for word, ’1’ for byte).
The ’k’ bits specify the literal operand.
The ’d’ bits select the address of the working register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**2:** For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

Words: 1
Cycles: 1

**Example 1:**
```
ADD.B #0xFF, W7 ; Add -1 to W7 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>12C0</td>
</tr>
</tbody>
</table>
| SR | 0000 | SR | 0009 | (N, C = 1)

**Example 2:**
```
ADD #0xFF, W1 ; Add 255 to W1 (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>12C0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
ADD

Add Wb to Short Literal

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} ADD{.B} Wb, #lit5, Wd

- [Wd]
- [Wd++]
- [Wd--]
- [++Wd]
- [--Wd]

Operands:

- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wd ∈ [W0 ... W15]

Operation: (Wb) + lit5 → Wd

Status Affected: DC, N, OV, Z, C

Encoding: 0100 0www wBqq qddd d11k kkkk

Description: Add the contents of the base register Wb to the 5-bit unsigned short literal operand, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The ‘w’ bits select the address of the base register.

- The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
- The ‘q’ bits select the destination Address mode.
- The ‘d’ bits select the destination register.
- The ‘k’ bits provide the literal operand, a five-bit integer number.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1: ADD.B W0, #0x1F, W7 ; Add W0 and 31 (Byte mode)

; Store the result in W7

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 2290</td>
<td>W0 2290</td>
</tr>
<tr>
<td>W7 12C0</td>
<td>W7 12AF</td>
</tr>
</tbody>
</table>
| SR 0000          | SR 0008          (N = 1)
Example 2: 
ADD W3, #0x6, [--W4]; Add W3 and 6 (Word mode)  
; Store the result in [--W4]

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>6006</td>
<td>6006</td>
</tr>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>1000</td>
<td>0FFE</td>
</tr>
<tr>
<td>Data 0FFE</td>
<td>Data 0FFE</td>
</tr>
<tr>
<td>DDEE</td>
<td>600C</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>DDEE</td>
<td>DDEE</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

## ADD
Add Wb to Ws

### Implemented in:
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

### Syntax:

```
({label:}) ADD{.B} Wb, Ws, Wd
```

- Wb ∈ [W0 ... W15]
- Ws ∈ [W0 ... W15]
- Wd ∈ [W0 ... W15]

### Operation:

\((Wb) + (Ws) \rightarrow Wd\)

### Status Affected:
- DC, N, OV, Z, C

### Encoding:

```
0100 0www wBqq qddd dppp ssss
```

### Description:

Add the contents of the source register Ws and the contents of the base  
register Wb, and place the result in the destination register Wd. Register  
direct addressing must be used for Wb. Either register direct or indirect  
addressing may be used for Ws and Wd.

The 'w' bits select the address of the base register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'q' bits select the destination Address mode.

The 'd' bits select the destination register.

The 'p' bits select the source Address mode.

The 's' bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation  
rather than a word operation. You may use a .W extension to  
denote a word operation, but it is not required.

### Words:
1

### Cycles:
1(1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and  
read-modify-write operations on non-CPU Special Function Registers. For more  
details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

---

© 2005-2011 Microchip Technology Inc.
Example 1: ADD.B W5, W6, W7 ; Add W5 to W6, store result in W7
      ; (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5  AB00</td>
<td>W5  AB00</td>
</tr>
<tr>
<td>W6  0030</td>
<td>W6  0030</td>
</tr>
<tr>
<td>W7  FFFF</td>
<td>W7  FF30</td>
</tr>
<tr>
<td>SR  0000</td>
<td>SR  0000</td>
</tr>
</tbody>
</table>

Example 2: ADD W5, W6, W7 ; Add W5 to W6, store result in W7
            ; (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5  AB00</td>
<td>W5  AB00</td>
</tr>
<tr>
<td>W6  0030</td>
<td>W6  0030</td>
</tr>
<tr>
<td>W7  FFFF</td>
<td>W7  AB30</td>
</tr>
<tr>
<td>SR  0000</td>
<td>SR  0008          (N = 1)</td>
</tr>
</tbody>
</table>

ADD
Add Accumulators

### Implemented in:
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

### Syntax:
{label:} ADD Acc

### Operands:
Acc ∈ [A,B]

### Operation:
\[
\text{If (Acc = A):} \\
(ACCA) + (ACCB) \rightarrow ACCA \\
\text{Else:} \\
(ACCA) + (ACCB) \rightarrow ACCB
\]

### Status Affected:
OA, OB, QAB, SA, SB, SAB

### Encoding:

<table>
<thead>
<tr>
<th></th>
<th>1100</th>
<th>1011</th>
<th>A000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
</tr>
</thead>
</table>

### Description:
Add the contents of Accumulator A to the contents of Accumulator B and place the result in the selected accumulator. This instruction performs a 40-bit addition.

The 'A' bit specifies the destination accumulator.

### Words:
1

### Cycles:
1

Example 1: ADD A ; Add ACCB to ACCA

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCA  00 0022 3300</td>
<td>ACCA  00 1855 7858</td>
</tr>
<tr>
<td>ACCB  00 1833 4558</td>
<td>ACCB  00 1833 4558</td>
</tr>
<tr>
<td>SR    0000</td>
<td>SR    0000</td>
</tr>
</tbody>
</table>
Example 2:  

```
ADD B ; Add ACCA to ACCB
; Assume Super Saturation mode enabled
; (ACCSAT = 1, SATA = 1, SATB = 1)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCA 00 E111 2222</td>
<td>ACCA 00 E111 2222</td>
</tr>
<tr>
<td>ACCB 00 7654 3210</td>
<td>ACCB 01 5765 5432</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 4800</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADD</th>
<th>16-bit Signed Add to Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented in: PIC24F</td>
<td>PIC24H</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Syntax:</td>
<td>{label:}</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Operands:</td>
<td>Ws ∈ [W0 ... W15]</td>
</tr>
<tr>
<td></td>
<td>Wb ∈ [W0 ... W15]</td>
</tr>
<tr>
<td></td>
<td>Slit4 ∈ [-8 ... +7]</td>
</tr>
<tr>
<td></td>
<td>Acc ∈ [A,B]</td>
</tr>
<tr>
<td>Operation:</td>
<td>Shift_{Slit4}(Extend(Ws)) + (Acc) → Acc</td>
</tr>
<tr>
<td>Status Affected:</td>
<td>OA, OB, OAB, SA, SB, SAB</td>
</tr>
<tr>
<td>Encoding:</td>
<td>1100</td>
</tr>
<tr>
<td>Description:</td>
<td>Add a 16-bit value specified by the source working register to the most significant word of the selected accumulator. The source operand may specify the direct contents of a working register or an effective address. The value specified is added to the most significant word of the accumulator by sign-extending and zero backfilling the source operand prior to the operation. The value added to the accumulator may also be shifted by a 4-bit signed literal before the addition is made.</td>
</tr>
<tr>
<td>Note:</td>
<td>Positive values of operand Slit4 represent an arithmetic shift right and negative values of operand Slit4 represent an arithmetic shift left. The contents of the source register are not affected by Slit4.</td>
</tr>
</tbody>
</table>
ADD

16-bit Signed Add to Accumulator

Words: 1
Cycles: 1 (1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: ADD W0, #2, A ; Add W0 right-shifted by 2 to ACCA

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>8000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 7000 0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 5000 0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2: ADD [W5++], A ; Add the effective value of W5 to ACCA ; Post-increment W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>2000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0067 2345</td>
</tr>
<tr>
<td>Data 2000</td>
<td>5000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 5067 2345</td>
</tr>
<tr>
<td>Data 2000</td>
<td>5000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
ADDCC
Add f to WREG with Carry

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} ADDC{.B} f {,WREG}`

Operands: `f ∈ [0 ... 8191]`

Operation: `(f) + (WREG) + (C) → destination designated by D`

Status Affected: DC, N, OV, Z, C

Encoding:
```
+1011 0100 1BDf ffff ffff ffff
```

Description: Add the contents of the default working register WREG, the contents of the file register and the Carry bit and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

Note 1: The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

3: The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1

Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:
```
ADDC.B RAM100 ; Add WREG and C bit to RAM100
 ; (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>WREG</td>
</tr>
<tr>
<td>RAM100</td>
<td>RAM100</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>CC60</td>
<td>CC60</td>
</tr>
<tr>
<td>8006</td>
<td>8067</td>
</tr>
<tr>
<td>0001 (C=1)</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:
```
ADDC RAM200, WREG ; Add RAM200 and C bit to the WREG
 ; (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>WREG</td>
</tr>
<tr>
<td>RAM200</td>
<td>RAM200</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>5600</td>
<td>8A01</td>
</tr>
<tr>
<td>3400</td>
<td>3400</td>
</tr>
<tr>
<td>0001 (C=1)</td>
<td>0000C (N, OV = 1)</td>
</tr>
</tbody>
</table>
### ADDC

**Add Literal to Wn with Carry**

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td>{label:} ADDC{.B} #lit10, Wn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operands:</td>
<td>lit10 ∈ [0 ... 255] for byte operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lit10 ∈ [0 ... 1023] for word operation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wn ∈ [W0 ... W15]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation:</td>
<td>lit10 + (Wn) + (C) → Wn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Affected:</td>
<td>DC, N, OV, Z, C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Encoding:**

```
1011 0000 1Bkk kkkk kkkk dddd
```

**Description:**
Add the 10-bit unsigned literal operand, the contents of the working register Wn and the Carry bit, and place the result back into the working register Wn.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

**Note 1:**
The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:**
For byte operations, the literal must be specified as an unsigned value [0:255]. See **Section 4.6 “Using 10-bit Literal Operands”** for information on using 10-bit literal operands in Byte mode.

**Note 3:**
The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

**Words:** 1
**Cycles:** 1

---

**Example 1:**
ADDC.B #0xFF, W7 ; Add -1 and C bit to W7 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>W7</td>
</tr>
<tr>
<td>12C0</td>
<td>12BF</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000 (C = 0)</td>
<td>0009 (N,C = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**
ADDC #0xFF, W1 ; Add 255 and C bit to W1 (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>12C0</td>
<td>13C0</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0001 (C = 1)</td>
<td>0000</td>
</tr>
</tbody>
</table>
**ADDC**

Add Wb to Short Literal with Carry

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: `{label:} ADDC{.B} Wb, #lit5, Wd

- [Wd]
- [Wd++]
- [Wd--]
- [++Wd]
- [--Wd]

Operands:

- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wd ∈ [W0 ... W15]

Operation:

(Wb) + lit5 + (C) → Wd

Status Affected:

DC, N, OV, Z, C

Encoding:

| 0100 | lwww | wBqq | qddd | d11k | kkkk |

Description:

Add the contents of the base register Wb, the 5-bit unsigned short literal operand and the Carry bit, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Wd.

The 'w' bits select the address of the base register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'q' bits select the destination Address mode.

The 'd' bits select the destination register.

The 'k' bits provide the literal operand, a five-bit integer number.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The Z flag is "sticky" for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1

Cycles: 1

**Example 1:** ADDC.B W0, #0x1F, [W7]; Add W0, 31 and C bit (Byte mode)

; Store the result in [W7]

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 CC80</td>
<td>W0 CC80</td>
</tr>
<tr>
<td>W7 12C0</td>
<td>W7 12C0</td>
</tr>
<tr>
<td>Data 12C0 B000</td>
<td>Data 12C0 B09F</td>
</tr>
<tr>
<td>SR 0000 (C = 0)</td>
<td>SR 0008 (N = 1)</td>
</tr>
</tbody>
</table>
**Example 2:**

```
ADDC W3, #0x6, [--W4] ; Add W3, 6 and C bit (Word mode) ; Store the result in [--W4]
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>W3 6006</td>
</tr>
<tr>
<td>W4</td>
<td>W4 0FFE</td>
</tr>
<tr>
<td>Data 0FFE</td>
<td>Data 0FFE</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>SR</td>
<td>SR 0001 (C = 1)</td>
</tr>
<tr>
<td></td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
**ADDC** Add Wb to Ws with Carry

**Implemented in:**
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**
```
{label:} ADDC{.B}  Wb,  Ws,  Wd
[Wb],      [Ws],      [Wd]
[Ws++],    [Wd++]
[Ws--],    [Wd--]
[++Ws],    [+Wd]
[--Ws],    [--Wd]
```

**Operands:**
- Wb ∈ [W0 ... W15]
- Ws ∈ [W0 ... W15]
- Wd ∈ [W0 ... W15]

**Operation:**

```
(Wb) + (Ws) + (C) → Wd
```

**Status Affected:**
- DC
- N
- OV
- Z
- C

**Encoding:**
```
0100 1www wBqq qddd dppp ssss
```

**Description:**
Add the contents of the source register Ws, the contents of the base register Wb and the Carry bit, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The Z flag is "sticky" for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

**Words:** 1
**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

---

DS70157F-page 110 © 2005-2011 Microchip Technology Inc.
### Example 1:

**ADD.C B W0,[W1++],[W2++]**  
; Add W0, [W1] and C bit (Byte mode)  
; Store the result in [W2]  
; Post-increment W1, W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>Data 0800</td>
<td>Data 0800</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td></td>
<td>(C = 1)</td>
</tr>
</tbody>
</table>

| Data 0800         | AB25              |
| Data 1000         | FFFF              |

| SR                | 0001              |
|                   | (C = 1)           |
|                   | 0000              |

### Example 2:

**ADD.C W3,[W2++],[W1++]**  
; Add W3, [W2] and C bit (Word mode)  
; Store the result in [W1]  
; Post-increment W1, W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td></td>
<td>(C = 1)</td>
</tr>
</tbody>
</table>

| Data 1000         | 8000              |
| Data 2000         | 2500              |

| SR                | 0001              |
|                   | (C = 1)           |
|                   | 0000              |
AND f and WREG

Implemented in: 

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

\{label\}: AND{.B} f \{,WREG\}

Operands: 

\(f \in [0 \ldots 8191]\)

Operation: 

\((f).\text{AND}(\text{WREG}) \rightarrow \text{destination designated by D}\)

Status Affected: 

\(N, Z\)

Encoding:

| 1011 | 0110 | 0BDF | fffe | fffe | fffe |

Description:

Compute the logical AND operation of the contents of the default working register WREG and the contents of the file register, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register). The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1(1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

AND.B RAM100 ; AND WREG to RAM100 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>CC80</td>
</tr>
<tr>
<td>RAM100</td>
<td>FFC0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

AND RAM200, WREG ; AND RAM200 to WREG (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>CC80</td>
</tr>
<tr>
<td>RAM200</td>
<td>12C0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### AND

#### AND Literal and Wn

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

```
{label:} AND{.B} #lit10, Wn
```

#### Operands:

- lit10 $\in [0 \ldots 255]$ for byte operation
- lit10 $\in [0 \ldots 1023]$ for word operation
- Wn $\in [W0 \ldots W15]$

#### Operation:

```
lit10.AND.(Wn) \rightarrow Wn
```

#### Status Affected:

- N, Z

#### Encoding:

```
1011 | 0010 | 0Bkk | kkkk | kkkk | dddd
```

#### Description:

Compute the logical AND operation of the 10-bit literal operand and the contents of the working register Wn and place the result back into the working register Wn. Register direct addressing must be used for Wn.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'k' bits specify the literal operand.

The 'd' bits select the address of the working register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

#### Words: 1

#### Cycles: 1

#### Example 1:

```
AND.B #0x83, W7 ; AND 0x83 to W7 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>12C0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

#### Example 2:

```
AND #0x333, W1 ; AND 0x333 to W1 (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>12D0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### AND

**AND Wb and Short Literal**

- **Implemented in:** PIC24F, PIC24H, PIC24E, dsPIC30F, dsPIC33F, dsPIC33E
- **Syntax:** `{label:} AND{.B} Wb, #lit5, Wd
  - [Wd]
  - [Wd++]
  - [Wd--]
  - [++Wd]
  - [--Wd]

- **Operands:**
  - Wb ∈ [W0 ... W15]
  - lit5 ∈ [0 ... 31]
  - Wd ∈ [W0 ... W15]

- **Operation:** (Wb).AND.lit5 → Wd

- **Status Affected:** N, Z

- **Encoding:**
  - 0110 0www wBqq qddd d11k kkkk

- **Description:** Compute the logical AND operation of the contents of the base register Wb and the 5-bit literal and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

  The 'w' bits select the address of the base register.
  The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
  The 'q' bits select the destination Address mode.
  The 'd' bits select the destination register.
  The 'k' bits provide the literal operand, a five-bit integer number.

  **Note:** The extension {.B} in the instruction denotes a byte operation rather than a word operation. You may use a {.W} extension to denote a word operation, but it is not required.

- **Words:** 1
- **Cycles:** 1

**Example 1:**

```
AND.B W0,#0x3,[W1++] ; AND W0 and 0x3 (Byte mode)
; Store to [W1]
; Post-increment W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>23A5</td>
<td>23A5</td>
</tr>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>2211</td>
<td>2212</td>
</tr>
<tr>
<td>Data 2210</td>
<td>Data 2210</td>
</tr>
<tr>
<td>9999</td>
<td>0199</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
AND W0,#0x1F,W1   ; AND W0 and 0x1F (Word mode)
; Store to W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>6723</td>
<td>6723</td>
</tr>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>7878</td>
<td>0003</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**AND**

And Wb and Ws

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} AND{.B} Wb, Ws, Wd

[Ws], [Wd]

[Ws++], [Wd++]

[Ws--], [Wd--]

[++Ws], [++Wd]

[--Ws], [--Wd]

**Operands:**

Wb ∈ [W0 ... W15]

Ws ∈ [W0 ... W15]

Wd ∈ [W0 ... W15]

**Operation:**

(Wb).AND.(Ws) → Wd

**Status Affected:**

N, Z

**Encoding:**

| 0110 | 0wwww | wBqq | qddd | dppp | ssss |

**Description:**

Compute the logical AND operation of the contents of the source register Ws and the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
### Example 1:

```
AND.B W0, W1 [W2++] ; AND W0 and W1, and
; store to [W2] (Byte mode)
; Post-increment W2
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0  AA55</td>
</tr>
<tr>
<td>W1</td>
<td>W1  2211</td>
</tr>
<tr>
<td>W2</td>
<td>W2  1001</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>SR</td>
<td>SR  0000</td>
</tr>
</tbody>
</table>

- Data: 1000 FFFF
- SR: 0000

### Example 2:

```
AND  W0, [W1++], W2 ; AND W0 and [W1], and
; store to W2 (Word mode)
; Post-increment W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0  AA55</td>
</tr>
<tr>
<td>W1</td>
<td>W1  1000</td>
</tr>
<tr>
<td>W2</td>
<td>W2  55AA</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>SR</td>
<td>SR  0000</td>
</tr>
</tbody>
</table>

- Data: 1000 2634
- SR: 0000
### ASR

#### Arithmetic Shift Right \( f \)

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\[
\{\text{label:}\} \text{ASR}\{.B\} \ f \ \{.WREG\}
\]

**Operands:**

\( f \in [0 \ldots 8191] \)

**Operation:**

- **For byte operation:**
  
  \[
  \begin{align*}
  \text{Dest}<7> & \rightarrow (f<7>) \\
  \text{Dest}<6> & \rightarrow (f<7>) \\
  \text{Dest}<5:0> & \rightarrow (f<6:1>) \\
  C & \rightarrow (f<0>)
  \end{align*}
  \]

- **For word operation:**
  
  \[
  \begin{align*}
  \text{Dest}<15> & \rightarrow (f<15>) \\
  \text{Dest}<14> & \rightarrow (f<15>) \\
  \text{Dest}<13:0> & \rightarrow (f<14:1>) \\
  C & \rightarrow (f<0>)
  \end{align*}
  \]

**Status Affected:**

\( N, Z, C \)

**Encoding:**

|     | 1101 | 0101 | 1BDf | ffff | ffff | ffff |

**Description:**

Shift the contents of the file register one bit to the right and place the result in the destination register. The Least Significant bit of the file register is shifted into the Carry bit of the STATUS Register. After the shift is performed, the result is sign-extended. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

**Note 1:** The extension \( .B \) in the instruction denotes a byte operation rather than a word operation. You may use a \( .W \) extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

**Words:**

1

**Cycles:**

1\(^{(1)}\)

---

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

---

\( \)
Example 1: \texttt{ASR.B RAM400, WREG}; ASR RAM400 and store to WREG; (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG 0600</td>
<td>WREG 0611</td>
</tr>
<tr>
<td>RAM400 0823</td>
<td>RAM400 0823</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

Example 2: \texttt{ASR RAM200}; ASR RAM200 (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM200 8009</td>
<td>RAM200 C004</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0009 (N, C = 1)</td>
</tr>
</tbody>
</table>
ASR
Arithmetic Shift Right Ws

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} ASR{.B} Ws, Wd

- [Ws], [Wd]
- [Ws++], [Wd++]
- [Ws--], [Wd--]
- [++Ws], [++Wd]
- [--Ws], [--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:
- (Ws<7>) → Wd<7>
- (Ws<7>) → Wd<6>
- (Ws<6:1>) → Wd<5:0>
- (Ws<0>) → C

For word operation:
- (Ws<15>) → Wd<15>
- (Ws<15>) → Wd<14>
- (Ws<14:1>) → Wd<13:0>
- (Ws<0>) → C

Status Affected: N, Z, C

Encoding: 1101 0001 1Bqq qddd dppp ssss

Description: Shift the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Least Significant bit of Ws is shifted into the Carry bit of the STATUS register. After the shift is performed, the result is sign-extended. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). The ‘q’ bits select the destination Address mode. The ‘d’ bits select the destination register. The ‘p’ bits select the source Address mode. The ‘s’ bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1(f)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
**Example 1:**

```
ASR.B [W0++], [W1++] ; ASR [W0] and store to [W1]
(Byte mode)
; Post-increment W0 and W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0600</td>
<td>W0 0601</td>
</tr>
<tr>
<td>W1 0801</td>
<td>W1 0802</td>
</tr>
<tr>
<td>Data 600 2366</td>
<td>Data 600 2366</td>
</tr>
<tr>
<td>Data 800 FFC0</td>
<td>Data 800 33C0</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
ASR W12, W13 ; ASR W12 and store to W13
(Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W12 AB01</td>
<td>W12 AB01</td>
</tr>
<tr>
<td>W13 0322</td>
<td>W13 D580</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0009</td>
</tr>
</tbody>
</table>

(N, C = 1)
## ASR

### Arithmetic Shift Right by Short Literal

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} ASR Wb, #lit4, Wnd

**Operands:**

Wb ∈ [W0 ... W15]

lit4 ∈ [0...15]

Wnd ∈ [W0 ... W15]

**Operation:**

lit4<3:0> → Shift_Val

Wb<15> → Wnd<15:15-Shift_Val + 1>

Wb<15:Shift_Val> → Wnd<15-Shift_Val:0>

**Status Affected:** N, Z

**Encoding:**

| 1101 | 1110 | lwww | wddd | d100 | kkkk |

**Description:**

Arithmetic shift right the contents of the source register Wb by the 4-bit unsigned literal, and store the result in the destination register Wnd. After the shift is performed, the result is sign-extended. Direct addressing must be used for Wb and Wnd.

The ‘w’ bits select the address of the base register.

The ‘d’ bits select the destination register.

The ‘k’ bits provide the literal operand.

**Note:** This instruction operates in Word mode only.

**Words:** 1

**Cycles:** 1

**Example 1:**

ASR W0, #0x4, W1 ; ASR W0 by 4 and store to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>060F</td>
</tr>
<tr>
<td>W1</td>
<td>1234</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>060F</td>
</tr>
<tr>
<td>W1</td>
<td>0060</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

ASR W0, #0x6, W1 ; ASR W0 by 6 and store to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>80FF</td>
</tr>
<tr>
<td>W1</td>
<td>0060</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>80FF</td>
</tr>
<tr>
<td>W1</td>
<td>FE03</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>

**Example 3:**

ASR W0, #0xF, W1 ; ASR W0 by 15 and store to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>70FF</td>
</tr>
<tr>
<td>W1</td>
<td>CC26</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>70FF</td>
</tr>
<tr>
<td>W1</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0002 (Z = 1)</td>
</tr>
</tbody>
</table>
**ASR**

**Arithmetic Shift Right by Wns**

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

```{label:} ASR Wb, Wns, Wnd```

Operands:

- `Wb` ∈ [W0 ... W15]
- `Wns` ∈ [W0 ... W15]
- `Wnd` ∈ [W0 ... W15]

Operation:

- `Wns<3:0>` → Shift_Val
- `Wb<15>` → Wnd<15:15-Shift_Val + 1>
- `Wb<15:Shift_Val>` → Wnd<15-Shift_Val:0>

Status Affected:

- N, Z

Encoding:

```
1101 1110 1www wddd d000 ssss
```

Description:

Arithmetic shift right the contents of the source register `Wb` by the 4 Least Significant bits of `Wns` (up to 15 positions) and store the result in the destination register `Wnd`. After the shift is performed, the result is sign-extended. Direct addressing must be used for `Wb`, `Wns` and `Wnd`.

- The 'w' bits select the address of the base register.
- The 'd' bits select the destination register.
- The 's' bits select the source register.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** If `Wns` is greater than 15, `Wnd` = 0x0 if `Wb` is positive, and `Wnd` = 0xFFFF if `Wb` is negative.

Words: 1

Cycles: 1

**Example 1:**

`ASR W0, W5, W6 ; ASR W0 by W5 and store to W6`

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 80FF</td>
<td>W0 80FF</td>
</tr>
<tr>
<td>W5 0004</td>
<td>W5 0004</td>
</tr>
<tr>
<td>W6 2633</td>
<td>W6 F80F</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

`ASR W0, W5, W6 ; ASR W0 by W5 and store to W6`

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 6688</td>
<td>W0 6688</td>
</tr>
<tr>
<td>W5 000A</td>
<td>W5 000A</td>
</tr>
<tr>
<td>W6 FF00</td>
<td>W6 0019</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 3:**

`ASR W11, W12, W13 ; ASR W11 by W12 and store to W13`

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W11 8765</td>
<td>W11 8765</td>
</tr>
<tr>
<td>W12 88E4</td>
<td>W12 88E4</td>
</tr>
<tr>
<td>W13 A5A5</td>
<td>W13 F876</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008 (N = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

BCLR

Bit Clear f

Implemented in:  
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{labeled:} BCLR{.B} f, #bit4

Operands:  
f ∈ [0 ... 8191] for byte operation  
f ∈ [0 ... 8190] (even only) for word operation  
bit4 ∈ [0 ... 7] for byte operation  
bit4 ∈ [0 ... 15] for byte operation

Operation:  
0 → f<bit4>

Status Affected: None

Encoding:  
1010 1001 bbbf ffff ffff fffb

Description:  
Clear the bit in the file register f specified by ‘bit4’. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The ‘b’ bits select value bit4 of the bit position to be cleared. 
The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word-aligned.

3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1  
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:  
BCLR.B 0x800, #0x7  ; Clear bit 7 in 0x800

Before Instruction  
Data 0800 66EF  
SR 0000

After Instruction  
Data 0800 666F  
SR 0000

Example 2:  
BCLR 0x400, #0x9  ; Clear bit 9 in 0x400

Before Instruction  
Data 0400 AA55  
SR 0000

After Instruction  
Data 0400 A855  
SR 0000
BCLR | Bit Clear in Ws

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```plaintext
{label:} BCLR{.B} Ws, #bit4

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],
```

**Operands:**

- `Ws ∈ [W0 ... W15]`
- `bit4 ∈ [0 ... 7]` for byte operation
- `bit4 ∈ [0 ... 15]` for word operation

**Operation:**

```
0 → Ws<bit4>
```

**Status Affected:** None

**Encoding:**

```
1010 0001 bbbb 0B00 0ppp ssss
```

**Description:**

Clear the bit in register `Ws` specified by `bit4`. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations). Register direct or indirect addressing may be used for `Ws`.

The ‘b’ bits select value `bit4` of the bit position to be cleared.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘s’ bits select the source/destination register.

The ‘p’ bits select the source Address mode.

**Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**Note 2:** When this instruction operates in Word mode, the source register address must be word-aligned.

**Note 3:** When this instruction operates in Byte mode, `bit4` must be between 0 and 7.

**Note 4:** In dsPIC33E and PIC24E devices, this instruction uses the DSRPAG register for indirect address generation in Extended Data Space.

**Words:** 1

**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

### Section 5. Instruction Descriptions

**Example 1:**

```assembly
BCLR.B W2, #0x2 ; Clear bit 3 in W2
```

**Before Instruction** | **After Instruction**
--- | ---
W2 | F234 | F230
SR | 0000 | 0000

**Example 2:**

```assembly
BCLR [W0++], #0x0 ; Clear bit 0 in [W0]
; Post-increment W0
```

**Before Instruction** | **After Instruction**
--- | ---
W0 | 2300 | 2302
Data 2300 | 5607 | 5606
SR | 0000 | 0000
BRA

Branch Unconditionally

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
| X | X | X | X | X | X |

Syntax: {label:} BRA Expr

Operands: Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: (PC + 2) + 2 * Slit16 → PC

Status Affected: None

Encoding:

0011 0111 nnnn nnnn nnnn nnnn

Description: The program will branch unconditionally, relative to the next PC. The offset of the branch is the two's complement number '2 * Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression. After the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The 'n' bits are a signed literal that specifies the number of program words offset from (PC + 2).

Words: 1

Cycles: 2 (PIC24F, PIC24H, dsPIC30F, dsPIC33F) 4 (PIC24E, dsPIC33E)

Example 1:

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 200A</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2:

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 200C</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Example 3:

002000 HERE:  BRA 0x1366  ; Branch to 0x1366
002002        . .
002004        . .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 1366</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
# BRA

## Computed Branch

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

## Syntax:

```
{label:} BRA Wn
```

## Operands:

$Wn \in [W0 ... W15]$  

## Operation:

```
(PC + 2) + (2 \times Wn) \rightarrow PC 
```

NOP $\rightarrow$ Instruction Register

## Status Affected:

None

## Encoding:

```
0000 0001 0110 0000 0000 ssss
```

## Description:

The program branches unconditionally, relative to the next PC. The offset of the branch is the sign-extended 17-bit value ($2 \times Wn$), which supports branches up to 32K instructions forward or backward. After this instruction executes, the new PC will be $(PC + 2) + 2 \times Wn$, since the PC will have incremented to fetch the next instruction.

The 's' bits select the source register.

## Words:

1

## Cycles:

2

### Example 1:

```
002000 HERE: BRA W7 ; Branch forward (2+2*W7)
002002 . . .
. . .
. . .
002108 . . .
00210A TABLE7: . . .
00210C . . .
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>002000</td>
<td>00210A</td>
</tr>
<tr>
<td>W7</td>
<td>W7</td>
</tr>
<tr>
<td>0084</td>
<td>0084</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
## Section 5. Instruction Descriptions

### BRA

**Computed Branch**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\[
\text{\{label:}\} \quad \text{BRA} \quad \text{Wn}
\]

**Operands:**

\[Wn \in \{W0 \ldots W15\}\]

**Operation:**

\[(PC + 2) + (2 \times Wn) \rightarrow PC\]

NOP \rightarrow Instruction Register

**Status Affected:**

None

**Encoding:**

| 0000 | 0001 | 0000 | 0110 | 0000 | ssss |

**Description:**

The program branches unconditionally, relative to the next PC. The offset of the branch is the sign-extended 17-bit value \((2 \times Wn)\), which supports branches up to 32K instructions forward or backward. After this instruction executes, the new PC will be \((PC + 2) + 2 \times Wn\), since the PC will have incremented to fetch the next instruction.

The ‘s’ bits select the source register.

**Words:**

1

**Cycles:**

4

**Example 1:**

002000 HERE: BRA W7 ; Branch forward \((2+2\times W7)\)

002002 ... ...

... ... ...

002108 ... ...

00210A TABLE7: ...

00210C ...

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 210A</td>
</tr>
<tr>
<td>W7 0084</td>
<td>W7 0084</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
BRA C  
Branch if Carry

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} BRA C, Expr

Operands:  
Expr may be a label, absolute address or expression.  
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767).

Operation:  
Condition = C  
If (Condition)  
(PC + 2) + 2 * Slit16 → PC  
NOP → Instruction Register

Status Affected:  
None

Encoding:  
0011 0001 nnnn nnnn nnnn nnnn

Description:  
If the Carry flag bit is '1', then the program will branch relative to the next PC.  
The offset of the branch is the two's complement number '2 * Slit16', which  
supports branches up to 32K instructions forward or backward. The Slit16  
value is resolved by the linker from the supplied label, absolute address or  
expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the  
PC will have incremented to fetch the next instruction. The instruction then  
becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a 16-bit signed literal that specify the offset from (PC + 2) in  
instruction words.

Words:  
1

Cycles:  
1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F  
1 (4 if branch taken) – PIC24E, dsPIC33E

Example 1:  
002000 HERE:  BRA C, CARRY  ; If C is set, branch to CARRY  
002002 NO_C:  . . .  ; Otherwise... continue  
002004 . . .  
002006 GOTO THERE  
002008 CARRY:  . . .  
00200A . . .  
00200C THERE:  . . .  
00200E . . .

Before Instruction  
<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>002000</td>
<td>0001</td>
<td>1</td>
</tr>
</tbody>
</table>

After Instruction  
<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>002008</td>
<td>0001</td>
<td>1</td>
</tr>
</tbody>
</table>
**Example 2:**

```
002000 HERE:     BRA C, CARRY ; If C is set, branch to CARRY
002002 NO_C:     ... ; Otherwise... continue
002004 ... 
002006 GOTO THERE
002008 CARRY:    ... 
00200A ... 
00200C THERE:    ... 
00200E ... 

Before Instruction
PC  00 2000
SR  0000

After Instruction
PC  00 2002
SR  0000
```

**Example 3:**

```
006230 HERE:     BRA C, CARRY ; If C is set, branch to CARRY
006232 NO_C:     ... ; Otherwise... continue
006234 ... 
006236 GOTO THERE
006238 CARRY:    ... 
00623A ... 
00623C THERE:    ... 
00623E ... 

Before Instruction
PC  00 6230
SR  0001(C = 1)

After Instruction
PC  00 6238
SR  0001(C = 1)
```

**Example 4:**

```
006230 START:    ... 
006232 ... 
006234 CARRY:    ... 
006236 ... 
006238 ... 
00623A ... 
00623C HERE:     BRA C, CARRY ; If C is set, branch to CARRY
00623E ... ; Otherwise... continue

Before Instruction
PC  00 623C
SR  0001(C = 1)

After Instruction
PC  00 6234
SR  0001(C = 1)
```
BRA GE  
Branch if Signed Greater Than or Equal

Implemented in:  
<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:
{label;} BRA GE, Expr

Operands:
Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation:
Condition = (N&&OV)||(N&&!OV)
If (Condition)
   (PC + 2) + 2 * Slit16 → PC
   NOP → Instruction Register

Status Affected:
None

Encoding:

| 0011 | 1101 | nnnn | nnnn | nnnn | nnnn |

Description:
If the logical expression (N&&OV)||(N&&!OV) is true, then the program will branch relative to the next PC. The offset of the branch is the two’s complement number ‘2 * Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a 16-bit signed literal that specify the offset from (PC + 2) in instruction words.

Note: The assembler will convert the specified label into the offset to be used.

Words: 1
Cycles: 1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
1 (4 if branch taken) – PIC24E, dsPIC33E

Example 1:

007600 LOOP: . . .
007602 . . .
007604 . . .
007606 . . .
007608 HERE: BRA GE, LOOP ; If GE, branch to LOOP
00760A NO_GE: . . . ; Otherwise... continue

Before Instruction   After Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>00 7608</th>
<th>PC</th>
<th>00 7600</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0000</td>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:

```
007600  LOOP:   . . .
007602          . . .
007604          . . .
007606          . . .
007608  HERE:  BRA GE, LOOP ; If GE, branch to LOOP
00760A  NO_GE: . . . ; Otherwise... continue
```

---

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 7608</td>
<td>00 760A</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0008 (N = 1)</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
**BRA GEU**  
Branch if Unsigned Greater Than or Equal

### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Syntax:

\{label: \} BRA GEU, Expr

### Operands:

Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16 offset that supports an offset range of [-32768 ... +32767] program words.

### Operation:

Condition = C  
If (Condition)  
\((PC + 2) + 2 \times \text{Slit16} \rightarrow PC\)  
\text{NOP} \rightarrow \text{Instruction Register}\n
### Status Affected:

None

### Encoding:

| 0011 | 0001 | nnnn | nnnn | nnnn | nnnn |

### Description:

If the Carry flag is ‘1’, then the program will branch relative to the next PC. The offset of the branch is the two's complement number ‘\(2 \times \text{Slit16}\)’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 \times \text{Slit16}\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a 16-bit signed literal that specify the offset from \((PC + 2)\) in instruction words.

**Note:** This instruction is identical to the BRA C, Expr (Branch if Carry) instruction and has the same encoding. It will reverse assemble as BRA C, Slit16.

### Words:

1

### Cycles:

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
1 (4 if branch taken) – PIC24E, dsPIC33E

### Example 1:

```
002000 HERE:   BRA GEU, BYPASS    ; If C is set, branch
002002 NO_GEU: . . .    ; to BYPASS
002004 . . .    ; Otherwise... continue
002006 . . .
002008 . . .
00200A GOTO THERE
00200C BYPASS: . . .
00200E . . .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 2000</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 200C</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

### BRA GT

**Branch if Signed Greater Than**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} BRA GT, Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

**Operation:**

\[
\text{Condition} = (\neg Z \land \neg N \land \neg OV) \lor (\neg Z \land \neg \neg N \land \neg OV) \\
\text{If} \ (\text{Condition}) \\
\quad (\text{PC} + 2) + 2 \times \text{Slit16} \rightarrow \text{PC} \\
\quad \text{NOP} \rightarrow \text{Instruction Register}
\]

**Status Affected:** None

**Encoding:**

<table>
<thead>
<tr>
<th></th>
<th>0011</th>
<th>1100</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
</tr>
</thead>
</table>

**Description:**

If the logical expression \((\neg Z \land \neg N \land \neg OV) \lor (\neg Z \land \neg \neg N \land \neg OV)\) is true, then the program will branch relative to the next PC. The offset of the branch is the two’s complement number \(2 \times \text{Slit16}\), which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((\text{PC} + 2) + 2 \times \text{Slit16}\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a 16-bit signed literal that specify the offset from \((\text{PC} + 2)\) in instruction words.

**Words:** 1

**Cycles:**

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F

1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

```
002000 HERE:     BRA GT, BYPASS ; If GT, branch to BYPASS
002002 NO_GT:    . . . ; Otherwise... continue
002004 . . .
002006 . . .
002008 . . .
00200A GOTO THERE
00200C BYPASS:  . . .
00200E . . .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0001 ((C = 1))</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 200C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0001 ((C = 1))</td>
</tr>
</tbody>
</table>
### BRA GTU

**Branch if Unsigned Greater Than**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} BRA GTU, Expr
```

**Operands:**

- Expr may be a label, absolute address or expression.
- Expr is resolved by the linker to a Silt16, where Silt16 ∈ [-32768 ... +32767].

**Operation:**

Condition = (C&&!Z)

If (Condition)

```
(PC + 2) + 2 * Silt16 → PC
NOP → Instruction Register
```

**Status Affected:** None

**Encoding:**

```
0011  1110  nnnn  nnnn  nnnn  nnnn
```

**Description:**

If the logical expression (C&&Iz) is true, then the program will branch relative to the next PC. The offset of the branch is the two's complement number ‘2 * Silt16’, which supports branches up to 32K instructions forward or backward. The Silt16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Silt16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC + 2).

**Words:** 1

**Cycles:**

- 1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
- 1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

```
002000 HERE:    BRA GTU, BYPASS    ; If GTU, branch to BYPASS
002002 NO_GTU:  . . .                ; Otherwise... continue
002004
002006
002008
00200A         GOTO THERE
00200C BYPASS:  . . .
00200E

Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 200C</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
```
### Section 5. Instruction Descriptions

#### BRA LE

**Branch if Signed Less Than or Equal**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} BRA LE, Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

**Operation:**

Condition = Z||(N&&!OV)||(!N&&OV)

If (Condition)

(PC + 2) + 2 * Slit16 → PC

NOP → Instruction Register

**Status Affected:**

None

**Encoding:**

```
0011 0100 nnnn nnnn nnnn nnnn
```

**Description:**

If the logical expression (Z||(N&&!OV)||(!N&&OV)) is true, then the program will branch relative to the next PC. The offset of the branch is the two’s complement number ‘2 * Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC + 2).

**Words:**

1

**Cycles:**

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F

1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

```
002000 HERE:  BRA LE, BYPASS          ; If LE, branch to BYPASS
002002 NO_LE:  . . .                  ; Otherwise... continue
002004        . . .
002006        . . .
002008        . . .
00200A        GOTO THERE
00200C BYPASS: . . .
00200E        . . .
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

© 2005-2011 Microchip Technology Inc.
### BRA LEU

**Branch if Unsigned Less Than or Equal**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**
```
{label:} BRA LEU, Expr
```

**Operands:**
- Expr may be a label, absolute address or expression.
- Expr is resolved by the linker to a Slit16, where $Slit16 \in [-32768 ... +32767]$.

**Operation:**
```
Condition = !C||Z
If (Condition)
(PC + 2) + 2 * Slit16 \rightarrow PC
NOP \rightarrow Instruction Register
```

**Status Affected:**
None

**Encoding:**
```
0011 0110 nnnn nnnn nnnn nnnn
```

**Description:**
If the logical expression (!C||Z) is true, then the program will branch relative to the next PC. The offset of the branch is the two's complement number '2 * Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC + 2).

**Words:**
1

**Cycles:**
- 1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
- 1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**
```
002000 HERE:     BRA LEU, BYPASS    ; If LEU, branch to BYPASS
002002 NO_LEU:   . . .              ; Otherwise... continue
002004 . . .      . . .              . . .
002006 . . .      GOTO THERE
00200C BYPASS:   . . .              . . .
00200E . . .
```

**Before Instruction**
```
<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>002000</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
```

**After Instruction**
```
<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00200C</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
```
BRA LT
Branch if Signed Less Than

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} BRA LT, Expr

Operands: Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation:

1. Condition = (N!&!OV)|(|N&!OV)
2. If (Condition)
   - (PC + 2) + 2 * Slit16 → PC
   - NOP → Instruction Register

Status Affected: None

Encoding:

|          | 0011 | 0101 | nnnn | nnnn | nnnn | nnnn |

Description:

If the logical expression ((N&!OV)|(|N&!OV)) is true, then the program will branch relative to the next PC. The offset of the branch is the two’s complement number ‘2 * Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from (PC + 2).

Words: 1

Cycles: 1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
1 (4 if branch taken) – PIC24E, dsPIC33E

Example 1:

002000 HERE: BRA LT, BYPASS ; If LT, branch to BYPASS
002002 NO_LT: ... ; Otherwise... continue
002004 ... 002006 ... 002008 ... 00200A GOTO THERE
00200C BYPASS: ...
00200E ...

Before Instruction

| PC | 00 2000 |
| SR | 0001 (C = 1) |

After Instruction

| PC | 00 2002 |
| SR | 0001 (C = 1) |
BRA LTU  

Branch if Unsigned Less Than

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

{label:}  BRA  LTU,  Expr  

Operands:  

Expr may be a label, absolute address or expression.  
Expr is resolved by the linker to a Slit16, where  
Slit16 ∈ [-32768 ... +32767].

Operation:  

Condition = !C  
If (Condition)  
(\(PC + 2\) + 2 * Slit16 → PC)  
NOP → Instruction Register

Status Affected:  

None

Encoding:

0011  1001  nnnn  nnnn  nnnn  nnnn  nnnn

Description:  

If the Carry flag is ’0’, then the program will branch relative to the next PC.  
The offset of the branch is the two’s complement number ’2 * Slit16’,  
which supports branches up to 32K instructions forward or backward. The  
Slit16 value is resolved by the linker from the supplied label, absolute  
address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 * Slit16\), since  
the PC will have incremented to fetch the next instruction. The instruction  
then becomes a two-cycle instruction, with a NOP executed in the second  
cycle.

The ’n’ bits are a signed literal that specifies the number of instructions  
offset from \((PC + 2)\).

Note :  

This instruction is identical to the BRA NC, Expr (Branch if Not  
Carry) instruction and has the same encoding. It will reverse  
assemble as BRA NC, Slit16.

Words:  

1

Cycles:  

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F  
1 (4 if branch taken) – PIC24E, dsPIC33E

Example 1:

002000 HERE:     BRA LTU, BYPASS ; If LTU, branch to BYPASS  
002002 NO_LTU:   . . . ; Otherwise... continue  
002004 . . .  
002006 . . .  
002008 . . .  
00200A GOTO THERE  
00200C BYPASS:  . . .  
00200E . . .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>
### BRA N

**Branch if Negative**

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label\} BRA N, Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slt16, where Slt16 \(\in [-32768 \ldots +32767]\).

**Operation:**

\[\text{Condition} = N\]

If (Condition)

\[(PC + 2) + 2 \times \text{Slt16} \rightarrow PC\]

\[\text{NOP} \rightarrow \text{Instruction Register} . \]

**Status Affected:** None

**Encoding:**

| 0011 | 0011 | nnnn | nnnn | nnnn | nnnn |

**Description:**

If the Negative flag is ‘1’, then the program will branch relative to the next PC. The offset of the branch is the two’s complement number ‘2 * Slt16’, which supports branches up to 32K instructions forward or backward. The Slt16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 \times \text{Slt16}\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from \((PC + 2)\).

**Words:** 1

**Cycles:**

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F

1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

| 002000 HERE: | BRA N, BYPASS ; If N, branch to BYPASS |
| 002002 NO_N: | . . . ; Otherwise... continue |
| 002004 | . . . |
| 002006 | . . . |
| 002008 | . . . |
| 00200A GOTO THERE |
| 00200C BYPASS: | . . . |
| 00200E | . . . |

**Before Instruction**

| PC | 002000 |
| SR | 0008 \((N = 1)\) |

**After Instruction**

| PC | 00200C |
| SR | 0008 \((N = 1)\) |
### BRA NC

**Branch if Not Carry**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label:\}  BRA  NC,  Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Sli16 ∈ [-32768 ... +32767].

**Operation:**

Condition = 1C
If (Condition)

(\(PC + 2\) + 2 * Sli16 \(\rightarrow PC\))

NOP \(\rightarrow\) Instruction Register

**Status Affected:** None

**Encoding:**

0011 1001 nnnn nnnn nnnn nnnn

**Description:**

If the Carry flag is ‘0’, then the program will branch relative to the next PC. The offset of the branch is the two’s complement number ‘2 * Sli16’, which supports branches up to 32K instructions forward or backward. The Sli16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 * Sli16\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions offset from \((PC + 2)\).

**Words:** 1

**Cycles:** 1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

```
002000 HERE:     BRA NC, BYPASS ; If NC, branch to BYPASS
002002 NO_NC:    . . . ; Otherwise... continue
002004 . . .
002006 . . .
002008 . . .
00200A GOTO THERE
00200C BYPASS:  . . .
00200E . . .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
**BRA NN**

**Branch if Not Negative**

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
   | X | X | X | X | X | X |

Syntax: \{label:\} BRA NN, Expr

Operands: Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: Condition = !N
If (Condition)

\[(PC + 2) + 2 \times \text{Slit16} \rightarrow \text{PC}\]

NOP \rightarrow \text{Instruction Register}

Status Affected: None

Encoding:

<table>
<thead>
<tr>
<th></th>
<th>0011</th>
<th>1011</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
</tr>
</thead>
</table>

Description: If the Negative flag is '0', then the program will branch relative to the next PC. The offset of the branch is the two’s complement number ‘2 * Slit16’, which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC + 2).

Words: 1

Cycles: 1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

```
002000 HERE:     BRA NN, BYPASS ; If NN, branch to BYPASS
002002 NO_NN:    . . . ; Otherwise... continue
002004          . . .
002006          . . .
002008          . . .
00200A         GOTO THERE
00200C BYPASS: . . .
00200E . . .
```

Before Instruction  | After Instruction
---|---
PC  00 2000  | PC  00 200C
SR  0000   | SR  0000
### BRA NOV

**Branch if Not Overflow**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{(label:}\ BRA NOV, Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

**Operation:**

Condition = !OV

If (Condition)

\[(PC + 2) + 2 \times \text{Slit16} \rightarrow \text{PC}\]

NOP \rightarrow \text{Instruction Register}

**Status Affected:** None

**Encoding:**

<table>
<thead>
<tr>
<th>0011</th>
<th>1000</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
</tr>
</thead>
</table>

**Description:**

If the Overflow flag is '0', then the program will branch relative to the next PC. The offset of the branch is the two's complement number '2 \times \text{Slit16}', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 \times \text{Slit16}\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from \((PC + 2)\).

**Words:** 1

**Cycles:**

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F

1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

```
002000 HERE:     BRA NOV, BYPASS ; If NOV, branch to BYPASS
002002 NO_NOV:   . . . ; Otherwise... continue
002004          . . .
002006          . . .
002008          . . .
00200A BYPASS:   GOTO THERE
00200C          . . .
00200E          . . .
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 002000</td>
<td>PC 00200C</td>
</tr>
<tr>
<td>SR 0008 (N = 1)</td>
<td>SR 0008 (N = 1)</td>
</tr>
</tbody>
</table>
BRA NZ  
Branch if Not Zero

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---------------|--------|--------|--------|----------|----------|----------|
              | X      | X      | X      | X        | X        | X        |

Syntax:  
{label:} BRA NZ, Expr

Operands:  
Expr may be a label, absolute address or expression.  
Expr is resolved by the linker to a Slit16, where  
Slit16 ∈ [-32768 ... +32767].

Operation:  
Condition = !Z  
If (Condition)  
(PC + 2) + 2 * Slit16 → PC  
NOP → Instruction Register

Status Affected: None

Encoding:  
0011 1010 nnnn nnnn nnnn nnnn

Description:  
If the Z flag is '0', then the program will branch relative to the next PC. The  
offset of the branch is the two’s complement number '2 * Slit16', which  
supports branches up to 32K instructions forward or backward. The Slit16  
value is resolved by the linker from the supplied label, absolute address or  
expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since  
the PC will have incremented to fetch the next instruction. The instruction  
then becomes a two-cycle instruction, with a NOP executed in the second  
cycle.

The 'n' bits are a signed literal that specifies the number of instructions  
offset from (PC + 2).

Words: 1

Cycles:  
1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F  
1 (4 if branch taken) – PIC24E, dsPIC33E

Example 1:  
002000 HERE: BRA NZ, BYPASS ; If NZ, branch to BYPASS  
002002 NO_NZ: . . . ; Otherwise... continue  
002004  . . .  
002006  . . .  
002008  . . .  
00200A GOTO THERE  
00200C BYPASS: . . .  
00200E . . .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 2000</td>
<td>00 2002</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0002 (Z = 1)</td>
<td>0002 (Z = 1)</td>
</tr>
</tbody>
</table>
**BRA OA**  
Branch if Overflow Accumulator A

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**  
{label:} BRA OA, Expr

**Operands:**  
Expr may be a label, absolute address or expression.  
Expr is resolved by the linker to a `Slit16`, where  
`Slit16 ∈ [-32768 ... +32767]`.

**Operation:**  
Condition = OA  
If (Condition)  
(PC + 2) + 2 * Slit16 → PC  
NOP → Instruction Register

**Status Affected:**  
None

**Encoding:**

<table>
<thead>
<tr>
<th></th>
<th>0000</th>
<th>1100</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
</tr>
</thead>
</table>

**Description:**  
If the Overflow Accumulator A flag is ‘1’, then the program will branch  
relative to the next PC. The offset of the branch is the two’s complement  
number ‘2 * Slit16’, which supports branches up to 32K instructions  
forward or backward. The Slit16 value is resolved by the linker from the  
supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since  
the PC will have incremented to fetch the next instruction. The instruction  
then becomes a two-cycle instruction, with a NOP executed in the second  
cycle.

The ‘n’ bits are a signed literal that specifies the number of instructions  
offset from (PC + 2).

**Example 1:**

```
002000 HERE: BRA OA, BYPASS ; If OA, branch to BYPASS
002002 NO_OA: . . . ; Otherwise... continue
002004 . . .
002006 . . .
002008 . . .
00200A GOTO THERE
00200C BYPASS: . . .
00200E . . .
```

**Example 1:**

Before Instruction:  
| PC | 00 2000 |
| SR | 8800   |

After Instruction:  
| PC | 00 200C |
| SR | 8800   |

Note: The assembler will convert the specified label into the offset to  
be used.

- **Words:** 1
- **Cycles:** 1 (2 if branch taken) – dsPIC30F, dsPIC33F  
1 (4 if branch taken) – dsPIC33E
BRA OB

Branch if Overflow Accumulator B

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Syntax: \{label:} BRA OB, Expr

Operands: Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: Condition = OB
If (Condition)
(PS + 2) + 2 * Slit16 → PC
NOP → Instruction Register

Status Affected: None

Encoding:

| 0000 | 1101 | nnnn | nnnn | nnnn | nnnn |

Description: If the Overflow Accumulator B flag is '1', then the program will branch relative to the next PC. The offset of the branch is the two's complement number '2 * Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC + 2).

Words: 1
Cycles: 1 (2 if branch taken) – dsPIC30F, dsPIC33F
1 (4 if branch taken) – dsPIC33E

Example 1:

002000 HERE: BRA OB, BYPASS ; If OB, branch to BYPASS
002002 NO OB: . . . ; Otherwise... continue
002004 . . .
002006 . . .
002008 . . .
00200A GOTO THERE
00200C BYPASS: . . .
00200E . . .

Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>8800 (OA, OAB = 1)</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>8800 (OA, OAB = 1)</td>
</tr>
</tbody>
</table>
**BRA OV**

Branch if Overflow

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{
(label:) BRA OV, Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

**Operation:**

Condition = OV
If (Condition)

\[(PC + 2) + 2 \times Slit16 \rightarrow PC\]

NOP \rightarrow Instruction Register

**Status Affected:** None

**Encoding:**

0011 0000 nnnn nnnn nnnn nnnn

**Description:**

If the Overflow flag is '1', then the program will branch relative to the next PC. The offset of the branch is the two’s complement number \(2 \times Slit16\), which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 \times Slit16\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from \((PC + 2)\).

**Words:** 1

**Cycles:**

1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F

1 (4 if branch taken) – PIC24E, dsPIC33E

**Example 1:**

002000 HERE: BRA OV, BYPASS ; If OV, branch to BYPASS
002002 NO_OV . . . ; Otherwise... continue
002004 .
002006 .
002008 .
00200A GOTO THERE
00200C BYPASS: .
00200E .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>SR 0002 (Z = 1)</td>
<td>SR 0002 (Z = 1)</td>
</tr>
</tbody>
</table>
**BRA SA**

**Branch if Saturation Accumulator A**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label\} BRA SA, Expr

**Operands:**

Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 \(\in [\text{-32768 ... +32767}]\).

**Operation:**

\[
\text{Condition} = \text{SA} \\
\text{If (Condition)} \\
(PC + 2) + 2 \times \text{Slit16} \to \text{PC} \\
\text{NOP} \to \text{Instruction Register}
\]

**Status Affected:**

None

**Encoding:**

| 0000 | 1110 | nnnn | nnnn | nnnn | nnnn |

**Description:**

If the Saturation Accumulator A flag is '1', then the program will branch relative to the next PC. The offset of the branch is the two's complement number \(2 \times \text{Slit16}\), which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be \((PC + 2) + 2 \times \text{Slit16}\), since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a \text{NOP} executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from \((PC + 2)\).

**Words:**

1

**Cycles:**

1 (2 if branch taken) – dsPIC30F, dsPIC33F
1 (4 if branch taken) – dsPIC33E

**Example 1:**

| 002000 HERE: | BRA SA, BYPASS | ; If SA, branch to BYPASS |
| 002002 NO_SA: | ... | ; Otherwise... continue |
| 002004 | ... |
| 002006 | ... |
| 002008 | ... |
| 00200A | GOTO THERE |
| 00200C BYPASS: | ... |
| 00200E | ... |

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 200C</td>
</tr>
<tr>
<td>SR 2400 (SA, SAB = 1)</td>
<td>SR 2400 (SA, SAB = 1)</td>
</tr>
</tbody>
</table>
BRA SB  
Branch if Saturation Accumulator B

Implemented in:  
<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax: 
{label:} BRA SB, Expr

Operands: 
Expr may be a label, absolute address or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: 
Condition = SB  
if (Condition)  
(PC + 2) + 2 * Slit16 → PC  
NOP → Instruction Register

Status Affected: 
None

Encoding:  
0000 | llll | nnnn | nnnn | nnnn | nnnn

Description: 
If the Saturation Accumulator B flag is '1', then the program will branch relative to the next PC. The offset of the branch is the two's complement number '2 * Slit16', which supports branches up to 32K instructions forward or backward. The Slit16 value is resolved by the linker from the supplied label, absolute address or expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since the PC will have incremented to fetch the next instruction. The instruction then becomes a two-cycle instruction, with a NOP executed in the second cycle.

The 'n' bits are a signed literal that specifies the number of instructions offset from (PC + 2).

Words: 
1

Cycles: 
1 (2 if branch taken) – dsPIC30F, dsPIC33F  
1 (4 if branch taken) – dsPIC33E

Example 1:  
002000 HERE:   BRA SB, BYPASS ; If SB, branch to BYPASS  
002002 NO_SB:   . . . ; Otherwise... continue  
002004 . . .  
002006 . . .  
002008 . . .  
00200A GOTO THERE  
00200C BYPASS: . . .  
00200E . . .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Before Instruction | After Instruction

PC 00 2000          | PC 00 2002        |
SR 0000             | SR 0000           |
Section 5. Instruction Descriptions

BRA Z  Branch if Zero

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E
                                X    X    X    X    X    X

Syntax:  {label:} BRA Z, Expr

Operands:  Expr may be a label, absolute address or expression.
Expr is resolved by the linker to a Slit16, where
Slit16 ∈ {-32768 ... +32767}.

Operation:  Condition = Z
if (Condition)
             (PC + 2) + 2 * Slit16 →PC
             NOP →Instruction Register

Status Affected:  None

Encoding:

<table>
<thead>
<tr>
<th>Code</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
<td>0010</td>
</tr>
<tr>
<td>nnnn</td>
<td>nnnn</td>
</tr>
<tr>
<td>nnnn</td>
<td>nnnn</td>
</tr>
<tr>
<td>nnnn</td>
<td>nnnn</td>
</tr>
</tbody>
</table>

Description:  If the Zero flag is '1', then the program will branch relative to the next PC.
The offset of the branch is the two’s complement number ‘2 * Slit16’, which
supports branches up to 32K instructions forward or backward. The Slit16
value is resolved by the linker from the supplied label, absolute address or
expression.

If the branch is taken, the new address will be (PC + 2) + 2 * Slit16, since
the PC will have incremented to fetch the next instruction. The instruction
then becomes a two-cycle instruction, with a NOP executed in the second
cycle.

The 'n' bits are a signed literal that specifies the number of instructions
offset from (PC + 2).

Words:  1
Cycles:  1 (2 if branch taken) – PIC24F, PIC24H, dsPIC30F, dsPIC33F
         1 (4 if branch taken) – PIC24E, dsPIC33E

Example 1:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>002000</td>
<td>HERE: BRA Z, BYPASS</td>
<td>If Z, branch to BYPASS</td>
</tr>
<tr>
<td>002002</td>
<td>NO_Z:</td>
<td>Otherwise... continue</td>
</tr>
<tr>
<td>002004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>002006</td>
<td></td>
<td></td>
</tr>
<tr>
<td>002008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00200A</td>
<td>GOTO THERE</td>
<td></td>
</tr>
<tr>
<td>00200C</td>
<td>BYPASS:</td>
<td></td>
</tr>
<tr>
<td>00200E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Before Instruction:

| PC  | 00 2000 |
| SR  | 0002 (Z = 1) |

After Instruction:

| PC  | 00 200C |
| SR  | 0002 (Z = 1) |
## BSET

**Bit Set f**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} BSET{.B} f, #bit4
```

**Operands:**

- `f ∈ [0 ... 8191]` for byte operation
- `f ∈ [0 ... 8190]` (even only) for word operation
- `bit4 ∈ [0 ... 7]` for byte operation
- `bit4 ∈ [0 ... 15]` for word operation

**Operation:**

1 → `f<bit4>`

**Status Affected:** None

**Encoding:**

```
1010 1000  bbbf  ffff  ffff  fffb
```

**Description:**

Set the bit in the file register `f` specified by `bit4`. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The 'b' bits select value `bit4` of the bit position to be set.

The 'f' bits select the address of the file register.

**Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**Note 2:** When this instruction operates in Word mode, the file register address must be word-aligned.

**Note 3:** When this instruction operates in Byte mode, `bit4` must be between 0 and 7.

**Words:** 1

**Cycles:** 1

**Example 1:**

```
BSET.B 0x601, #0x3 ; Set bit 3 in 0x601
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 0600</td>
<td>Data 0600</td>
</tr>
<tr>
<td>F234</td>
<td>FA34</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
BSET 0x444, #0xF ; Set bit 15 in 0x444
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 0444</td>
<td>Data 0444</td>
</tr>
<tr>
<td>5604</td>
<td>D604</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Note:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 "Multi-Cycle Instructions".
Section 5. Instruction Descriptions

### BSET

**Bit Set in Ws**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} BSET{.B} Ws, #bit4

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],
```

<table>
<thead>
<tr>
<th>Operands:</th>
<th>Ws ∈ [W0 ... W15]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit4 ∈ [0 ... 7] for byte operation</td>
<td></td>
</tr>
<tr>
<td>bit4 ∈ [0 ... 15] for word operation</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation:</th>
<th>1 → Ws&lt;bit4&gt;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Status Affected:</th>
<th>None</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Encoding:</th>
<th>1010</th>
<th>0000</th>
<th>bbbb</th>
<th>0B00</th>
<th>0ppp</th>
<th>ssss</th>
</tr>
</thead>
</table>

**Description:**

Set the bit in register Ws specified by ‘bit4’. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations). Register direct or indirect addressing may be used for Ws.

The ‘b’ bits select value bit4 of the bit position to be cleared.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source/destination register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** When this instruction operates in Word mode, the source register address must be word-aligned.

**Note 3:** When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

**Note 4:** In dsPIC33E and PIC24E devices, this instruction uses the DSRPAG register for indirect address generation in Extended Data Space.

**Words:**

1

**Cycles:**

1(1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

---

© 2005-2011 Microchip Technology Inc.  DS70157F-page 153
Example 1: \( \text{BSET.B W3, \#0x7} \) ; Set bit 7 in W3

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>0026</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W3</td>
<td>00A6</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2: \( \text{BSET [W4++], \#0x0} \) ; Set bit 0 in [W4] ; Post-increment W4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>6700</td>
</tr>
<tr>
<td>Data 6700</td>
<td>1734</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W4</td>
<td>6702</td>
</tr>
<tr>
<td>Data 6700</td>
<td>1735</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
BSW Bit Write in Ws

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

{label:} BSW.C Ws, Wb
BSW.Z [Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],

Operands:

Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]

Operation:

For ".C" operation:
C → Ws<(Wb)>
For ".Z" operation (default):
Z → Ws<(Wb)>

Status Affected: None

Encoding:

```
 1010 1101 Zwww w000 0ppp ssss
```

Description:
The (Wb) bit in register Ws is written with the value of the C or Z flag from the STATUS register. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the working register. Only the four Least Significant bits of Wb are used to determine the destination bit number. Register direct addressing must be used for Wb, and either register direct, or indirect addressing may be used for Ws.

The ‘Z’ bit selects the C or Z flag as source.
The ‘w’ bits select the address of the bit select register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note: This instruction only operates in Word mode. If no extension is provided, the ".Z" operation is assumed.

Words: 1
Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: BSW.C W2, W3 ; Set bit W3 in W2 to the value ; of the C bit

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Instruction</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>W2 F234</td>
<td>7234</td>
</tr>
<tr>
<td>W3 111F</td>
<td>111F</td>
</tr>
<tr>
<td>SR 0002 (Z = 1, C = 0)</td>
<td>SR 0002 (Z = 1, C = 0)</td>
</tr>
</tbody>
</table>
Example 2: BSW.Z W2, W3 ; Set bit W3 in W2 to the complement of the Z bit

Before Instruction | After Instruction
---|---
W2 | E235 | E234
W3 | 0550 | 0550
SR | 0002 (Z = 1, C = 0) | 0002 (Z = 1, C = 0)

Example 3: BSW.C [++W0], W6 ; Set bit W6 in [W0++] to the value of the C bit

Before Instruction | After Instruction
---|---
W0 | 1000 | 1002
W6 | 34A3 | 34A3
Data | 2380 | 2388
SR | 0001 (Z = 0, C = 1) | 0001 (Z = 0, C = 1)

Example 4: BSW.Z [W1--], W5 ; Set bit W5 in [W1] to the complement of the Z bit ; Post-decrement W1

Before Instruction | After Instruction
---|---
W1 | 1000 | 0FFE
W5 | 888B | 888B
Data | C4DD | CCDD
SR | 0001 (C = 1) | 0001 (C = 1)
Section 5. Instruction Descriptions

BTG

Bit Toggle f

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} BTG(.B) f, #bit4

Operands:
- \( f \in [0 \ldots 8191] \) for byte operation
- \( f \in [0 \ldots 8190] \) (even only) for word operation
- \( \text{bit4} \in [0 \ldots 7] \) for byte operation
- \( \text{bit4} \in [0 \ldots 15] \) for word operation

Operation:
- \((f) < \text{bit4}> \rightarrow (f) < \text{bit4}>\)

Status Affected: None

Encoding:

| 1010 | 1010 | bbbf | ffff | ffff | fffb |

Description:
Bit 'bit4' in file register 'f' is toggled (complemented). For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operation, bit 15 for word operation) of the byte.

The 'b' bits select value bit4, the bit position to toggle.
The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word-aligned.

3: When this instruction operates in Byte mode, 'bit4' must be between 0 and 7.

Words: 1

Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: BTG.B 0x1001, #0x4 ; Toggle bit 4 in 0x1001

Before Instruction
| Data 1000 | F234 |
| SR 0000 |      |

After Instruction
| Data 1000 | E234 |
| SR 0000 |      |

Example 2: BTG 0x1660, #0x8 ; Toggle bit 8 in RAM660

Before Instruction
| Data 1660 | 5606 |
| SR 0000 |      |

After Instruction
| Data 1660 | 5706 |
| SR 0000 |      |
BTG

Bit Toggle in Ws

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented in</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} BTG(B) Ws, #bit4

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 7] for byte operation
bit4 ∈ [0 ... 15] for word operation

Operation: (Ws)<bit4> → Ws<bit4>

Status Affected: None

Encoding:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>0010</td>
<td>bbbb</td>
<td>0B00</td>
<td>0ppp</td>
<td>ssss</td>
<td></td>
</tr>
</tbody>
</table>

Description: Bit ‘bit4’ in register Ws is toggled (complemented). For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations). Register direct or indirect addressing may be used for Ws.

The ‘b’ bits select value bit4, the bit position to test.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘s’ bits select the source/destination register.
The ‘p’ bits select the source Address mode.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the source register address must be word-aligned.

3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

4: In dsPIC33E and PIC24E devices, this instruction uses the DSRPAG register for indirect address generation in Extended Data Space.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: BTG W2, #0x0 ; Toggle bit 0 in W2

<table>
<thead>
<tr>
<th></th>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>F234</td>
<td>F235</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2: \( \text{BTG }[W0++],#0x0 \) ; Toggle bit 0 in [W0] ; Post-increment W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>Data 2300</td>
<td>Data 2300</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
<tr>
<td>2300</td>
<td>2302</td>
</tr>
<tr>
<td>5606</td>
<td>5607</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
BTSC  Bit Test f, Skip if Clear

Implemented in:  
<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} BTSC.{B} f, #bit4

Operands:  
f ∈ [0 ... 8191] for byte operation  
f ∈ [0 ... 8190] (even only) for word operation  
bit4 ∈ [0 ... 7] for byte operation  
bit4 ∈ [0 ... 15] for word operation

Operation:  
Test (f)<bit4>, skip if clear

Status Affected:  
None

Encoding:  
1010 1111 bbbf ffff ffff fffb

Description:  
Bit ‘bit4’ in the file register is tested. If the tested bit is ‘0’, the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. If the tested bit is ‘1’, the next instruction is executed as normal. In either case, the contents of the file register are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operations, bit 15 for word operations).

The ‘b’ bits select value bit4, the bit position to test.  
The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word-aligned.

3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words:  
1

Cycles:  
1 (2 or 3)(f)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**  
002000 HERE: BTSC.B 0x1201, #2 ; If bit 2 of 0x1201 is 0,  
002002      GOTO      BYPASS ; skip the GOTO  
002004      ...  
002006      ...  
002008      BYPASS: ...  
00200A      ...  

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 2000</td>
<td>00 2002</td>
</tr>
<tr>
<td>Data 1200</td>
<td>Data 1200</td>
</tr>
<tr>
<td>264F</td>
<td>264F</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:

```
002000 HERE:    BTSC    0x804, #14  ; If bit 14 of 0x804 is 0,
002002          GOTO    BYPASS  ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

```markdown
<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2006</td>
</tr>
<tr>
<td>Data 0804</td>
<td>Data 0804</td>
</tr>
<tr>
<td>2647</td>
<td>2647</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
```
BTSC

Bit Test Ws, Skip if Clear

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} BTSC Ws, #bit4

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 15]

Operation: Test (Ws)<bit4>, skip if clear

Status Affected: None

Encoding:

|      | 1010 | 0111 | bbbb | 0000 | 0ppp | ssss |

Description:

Bit 'bit4' in Ws is tested. If the tested bit is '0', the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. If the tested bit is '1', the next instruction is executed as normal. In either case, the contents of Ws are not changed.

For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the word. Either register direct or indirect addressing may be used for Ws.

The 'b' bits select value bit4, the bit position to test.
The 'p' bits select the source Address mode.
The 's' bits select the source register.

Note: This instruction operates in Word mode only.

Words: 1
Cycles: 1 (2 or 3 if the next instruction is skipped)(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: 002000 HERE: BTSC W0, #0x0 ; If bit 0 of W0 is 0,
002002 GOTO BYPASS ; skip the GOTO
002004 ... 002006 ... 002008 BYPASS: ... 00200A ...

Before Instruction After Instruction

| PC | 00 2000 | PC | 00 2002 |
| W0 | 264F | W0 | 264F |
| SR | 0000 | SR | 0000 |
Section 5. Instruction Descriptions

Example 2:
002000 HERE: BTSC W6, #0xF ; If bit 15 of W6 is 0,
002002 GOTO BYPASS ; skip the GOTO
002004 . . .
002006 . . .
002008 BYPASS: . . .
00200A . . .

Before Instruction
PC  00 2000
W6  264F
SR  0000

After Instruction
PC  00 2006
W6  264F
SR  0000

Example 3:
003400 HERE: BTSC [W6++], #0xC ; If bit 12 of [W6] is 0,
003402 GOTO BYPASS ; skip the GOTO
003404 . . .
003406 . . .
003408 BYPASS: . . .
00340A . . .

Before Instruction
PC  00 3400
W6  1800
Data 1800 1000
SR  0000

After Instruction
PC  00 3402
W6  1802
Data 1800 1000
SR  0000
BTSS: Bit Test f, Skip if Set

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} BTSS{.B} f, #bit4

Operands: f ∈ [0 ... 8191] for byte operation
          f ∈ [0 ... 8190] (even only) for word operation
          bit4 ∈ [0 ... 7] for byte operation
          bit4 ∈ [0 ... 15] for word operation

Operation: Test (f)<bit4>, skip if set

Status Affected: None

Encoding: 1010 1110 bbbf ffff ffff fffb

Description: Bit ‘bit4’ in the file register ‘f’ is tested. If the tested bit is ‘1’, the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. If the tested bit is ‘0’, the next instruction is executed as normal. In either case, the contents of the file register are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operation, bit 15 for word operation).

The ‘b’ bits select value bit4, the bit position to test.
The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: When this instruction operates in Word mode, the file register address must be word-aligned.

3: When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

Words: 1
Cycles: 1 (2 or 3 if the next instruction is skipped)(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

(1) Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
Section 5. Instruction Descriptions

Example 1:
007100 HERE: BTSS.B 0x1401, #0x1 ; If bit 1 of 0x1401 is 1,
007102 CLR WREG ; don’t clear WREG
007104 .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 7100</td>
<td>PC 00 7104</td>
</tr>
<tr>
<td>Data 1400 0280</td>
<td>Data 1400 0280</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2:
007100 HERE: BTSS 0x890, #0x9 ; If bit 9 of 0x890 is 1,
007102 GOTO BYPASS ; skip the GOTO
007104 .
007106 BYPASS: .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 7100</td>
<td>PC 00 7102</td>
</tr>
<tr>
<td>Data 0890 00FE</td>
<td>Data 0890 00FE</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
## BTSS

**Bit Test Ws, Skip if Set**

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} BTSS Ws, #bit4
[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],
```

**Operands:**

- `Ws ∈ [W0 ... W15]`
- `bit4 ∈ [0 ... 15]`

**Operation:**

Test `(Ws)<bit4>`, skip if set.

**Status Affected:**

None

**Encoding:**

```
1010 0110 bbbb 0000 0ppp ssss
```

**Description:**

Bit `bit4` in `Ws` is tested. If the tested bit is '1', the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a **NOP** is executed instead. If the tested bit is '0', the next instruction is executed as normal. In either case, the contents of `Ws` are not changed. For the `bit4` operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the word. Either register direct or indirect addressing may be used for `Ws`.

The 'b' bits select the value `bit4`, the bit position to test.

The 's' bits select the source register.

The 'p' bits select the source Address mode.

**Note:** This instruction operates in Word mode only.

**Words:**

1

**Cycles:**

1 (2 or 3 if the next instruction is skipped)(1)

**Example 1:**

```
002000 HERE:    BTSS    W0, #0x0    ; If bit 0 of W0 is 1,
002002          GOTO    BYPASS    ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

### Example:

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 2000</td>
<td>00 2006</td>
</tr>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>264F</td>
<td>264F</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:

```
002000 HERE:    BTSS    W6, #0xF    ; If bit 15 of W6 is 1,
002002          GOTO    BYPASS      ; skip the GOTO
002004          . . .
002006          . . .
002008 BYPASS:  . . .
00200A          . . .
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>W6 264F</td>
<td>W6 264F</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 3:

```
003400 HERE:    BTSS    [W6++], 0xC    ; If bit 12 of [W6] is 1,
003402          GOTO    BYPASS      ; skip the GOTO
003404          . . .
003406          . . .
003408 BYPASS:  . . .
00340A          . . .
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 3400</td>
<td>PC 00 3406</td>
</tr>
<tr>
<td>W6 1800</td>
<td>W6 1802</td>
</tr>
<tr>
<td>Data 1800 1000</td>
<td>Data 1800 1000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
**BTST**  
**Bit Test**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} BTST{.B} f, #bit4
```

**Operands:**

- `f` ∈ [0 ... 8191] for byte operation
- `f` ∈ [0 ... 8190] (even only) for word operation
- `bit4` ∈ [0 ... 7] for byte operation
- `bit4` ∈ [0 ... 15] for word operation

**Operation:**

```
(f)<bit4> → Z
```

**Status Affected:**

Z

**Encoding:**

```
1010 1011 bbbf ffff ffff fffb
```

**Description:**

Bit ‘bit4’ in file register ‘f’ is tested and the complement of the tested bit is stored to the Z flag in the STATUS register. The contents of the file register are not changed. For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 7 for byte operation, bit 15 for word operation).

The ‘b’ bits select value bit4, the bit position to be tested.
The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** When this instruction operates in Word mode, the file register address must be word-aligned.

**Note 3:** When this instruction operates in Byte mode, ‘bit4’ must be between 0 and 7.

**Words:**

1

**Cycles:**

1

---

**Example 1:**

```
BTST.B 0x1201, #0x3
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1200</td>
<td>Data 1200</td>
</tr>
<tr>
<td>F7FF</td>
<td>F7FF</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0002 (Z = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

```
BTST 0x1302, #0x7
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1302</td>
<td>Data 1302</td>
</tr>
<tr>
<td>F7FF</td>
<td>F7FF</td>
</tr>
<tr>
<td>SR 0002 (Z = 1)</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

---

**Note:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

---

© 2005-2011 Microchip Technology Inc.
## BTST

### Bit Test in Ws

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Syntax:

- `{label:}` BTST.C Ws, #bit4
- BTST.Z [Ws], [Ws++], [Ws--], [++Ws], [--Ws],

### Operands:

- Ws ∈ [W0 ... W15]
- bit4 ∈ [0 ... 15]

### Operation:

- For “.C” operation:
  \((Ws)<bit4> \rightarrow C\)
- For “.Z” operation (default):
  \((Ws)<bit4> \rightarrow Z\)

### Status Affected:

- Z or C

### Encoding:

| 1010 | 0011 | bbbb | Z000 | 0ppp | ssss |

### Description:

Bit ‘bit4’ in register Ws is tested. If the “.Z” option of the instruction is specified, the complement of the tested bit is stored to the Zero flag in the STATUS register. If the “.C” option of the instruction is specified, the value of the tested bit is stored to the Carry flag in the STATUS register. In either case, the contents of Ws are not changed.

For the bit4 operand, bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the word. Either register direct or indirect addressing may be used for Ws.

The ‘b’ bits select value bit4, the bit position to test.
The ‘Z’ bit selects the C or Z flag as destination.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

**Note:** This instruction only operates in Word mode. If no extension is provided, the “.Z” operation is assumed.

### Words:

1

### Cycles:

1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.
Example 1:  
```c
BTST.C [W0++], #0x3 ; Set C - bit 3 in [W0] ; Post-increment W0
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>Data 1200</td>
<td>Data 1200</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 2:  
```c
BTST.Z W0, #0x7 ; Set Z - complement of bit 7 in W0
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0002 (Z = 1)</td>
</tr>
</tbody>
</table>

```
BTST  Bit Test in Ws

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

- `{label;}
  BTST.C Ws, Wb
- BTST.Z [Ws],
  [Ws++],
  [Ws--],
  [++Ws],
  [--Ws],

Operands:

- Ws ∈ [W0 ... W15]
- Wb ∈ [W0 ... W15]

Operation:

For ".C" operation:

(Ws)<(Wb)> → C

For ".Z" operation (default):

(Ws)<(Wb)> → Z

Status Affected: Z or C

Encoding:

| 1010 | 0101 | Zwww | w000 | 0ppp | ssss |

Description:

The (Wb) bit in register Ws is tested. If the ".C" option of the instruction is specified, the value of the tested bit is stored to the Carry flag in the STATUS register. If the ".Z" option of the instruction is specified, the complement of the tested bit is stored to the Zero flag in the STATUS register. In either case, the contents of Ws are not changed.

Only the four Least Significant bits of Wb are used to determine the bit number. Bit numbering begins with the Least Significant bit (bit 0) and advances to the Most Significant bit (bit 15) of the working register. Register direct or indirect addressing may be used for Ws.

The ‘Z’ bit selects the C or Z flag as destination.
The ‘w’ bits select the address of the bit select register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note: This instruction only operates in Word mode. If no extension is provided, the ".Z" operation is assumed.

Words: 1
Cycles: 1\(^{(1)}\)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

© 2005-2011 Microchip Technology Inc.
Example 1:  
BTST.C W2, W3 ; Set C = bit W3 of W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>F234</td>
</tr>
<tr>
<td>W3</td>
<td>2368</td>
</tr>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>

Example 2:  
BTST.Z [W0++], W1 ; Set Z = complement of bit W1 in [W0], ; Post-increment W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>1200</td>
</tr>
<tr>
<td>W1</td>
<td>CCC0</td>
</tr>
<tr>
<td>Data 1200</td>
<td>6243</td>
</tr>
<tr>
<td>SR</td>
<td>0002 (Z = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

BTSTS                  Bit Test/Set f

Implemented in:        PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E
                        X      X      X      X       X       X

Syntax:                {label:} BTSTS(.B) f,    #bit4

Operands:              f ∈ [0 ... 8191] for byte operation
                        f ∈ [0 ... 8190] (even only) for word operation
                        bit4 ∈ [0 ... 7] for byte operation
                        bit4 ∈ [0 ... 15] for word operation

Operation:             (f)<bit4> → Z
                        1 →(f)<bit4>

Status Affected:       Z

Encoding:              1010  1100  bbbf  ffff  ffff  fffbf

Description:           Bit ‘bit4’ in file register ‘f’ is tested and the complement of the
tested bit is stored to the Zero flag in the STATUS register. The tested bit is then set
to ‘1’ in the file register. For the bit4 operand, bit numbering begins with
the Least Significant bit (bit 0) and advances to the Most Significant bit
(bit 7 for byte operations, bit 15 for word operations).

The ‘b’ bits select value bit4, the bit position to test/set.
The ‘f’ bits select the address of the file register.

Note 1:                The extension .B in the instruction denotes a byte operation
                        rather than a word operation. You may use a .W extension to
denote a word operation, but it is not required.

2:                     When this instruction operates in Word mode, the file register
                        address must be word-aligned.

3:                     When this instruction operates in Byte mode, ‘bit4’ must be
                        between 0 and 7.

4:                     The file register ‘f’ must not be the CPU Status register (SR).

Words:                 1

Cycles:                1(1)

Note 1:                In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read
                        and read-modify-write operations on non-CPU Special Function Registers. For more
details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

© 2005-2011 Microchip Technology Inc.
Example 1:  
```
BTSTS.B 0x1201, #0x3 ; Set Z = complement of bit 3 in 0x1201,
               ; then set bit 3 of 0x1201 = 1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1200</td>
<td>Data 1200</td>
</tr>
<tr>
<td>F7FF</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0002</td>
</tr>
</tbody>
</table>

(Z = 1)

Example 2:  
```
BTSTS 0x808, #15 ; Set Z = complement of bit 15 in 0x808,
                   ; then set bit 15 of 0x808 = 1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM300</td>
<td>RAM300</td>
</tr>
<tr>
<td>8050</td>
<td>8050</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0002</td>
<td>0000</td>
</tr>
</tbody>
</table>

(Z = 1)
Section 5. Instruction Descriptions

BTSTS  Bit Test/Set in Ws

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} BTSTS.C Ws, #bit4
BTSTS.Z [Ws], [Ws++], [Ws--], [+Ws], [--Ws],

Operands: Ws ∈ [W0 ... W15]
bit4 ∈ [0 ... 15]

Operation: For "C" operation:
(Ws)<bit4> → C
1 → Ws<bit4>
For "Z" operation (default):
(Ws)<bit4> → Z
1 → Ws<bit4>

Status Affected: Z or C

Encoding: 1010 0100 bbbb Z000 0ppp ssss

Description: Bit ‘bit4’ in register Ws is tested. If the "Z" option of the instruction is specified, the complement of the tested bit is stored to the Zero flag in the STATUS register. If the "C" option of the instruction is specified, the value of the tested bit is stored to the Carry flag in the STATUS register. In both cases, the tested bit in Ws is set to ‘1’.

The ‘b’ bits select the value bit4, the bit position to test/set.
The ‘Z’ bit selects the C or Z flag as destination.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: This instruction only operates in Word mode. If no extension is provided, the "Z" operation is assumed.

2: If Ws is used as a pointer, it must not contain the address of the CPU Status register (SR).

3: In dsPIC33E and PIC24E devices, this instruction uses the DSRPAG register for indirect address generation in Extended Data Space.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
Example 1: BTSTS.C [W0++], #0x3
; Set C = bit 3 in [W0]
; Set bit 3 in [W0] = 1
; Post-increment W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 1200</td>
<td>W0 1202</td>
</tr>
<tr>
<td>Data 1200</td>
<td>Data 1200</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2: BTSTS.Z W0, #0x7
; Set Z = complement of bit 7
; in W0, and set bit 7 in W0 = 1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 F234</td>
<td>W0 F2BC</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0002 (Z = 1)</td>
</tr>
</tbody>
</table>
## CALL

### Call Subroutine

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} CALL Expr

**Operands:**

Expr may be a label or expression (but not a literal). Expr is resolved by the linker to a lit23, where lit23 ∈ [0 ... 8388606].

**Operation:**

\[
\begin{align*}
(\text{PC}) + 4 & \rightarrow \text{PC} \\
(\text{PC}<15:0>) & \rightarrow \text{TOS} \\
(W15) + 2 & \rightarrow W15 \\
(\text{PC}<23:16>) & \rightarrow \text{TOS} \\
(W15) + 2 & \rightarrow W15 \\
\text{lit23} & \rightarrow \text{PC}
\end{align*}
\]

**Status Affected:** None

**Encoding:**

<table>
<thead>
<tr>
<th>1st word</th>
<th>0000</th>
<th>0010</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnnn</th>
<th>nnn0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2nd word</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0nnn</td>
<td>nnnn</td>
</tr>
</tbody>
</table>

**Description:**

Direct subroutine call over the entire 4-Mbyte instruction program memory range. Before the CALL is made, the 24-bit return address (PC + 4) is PUSHed onto the stack. After the return address is stacked, the 23-bit value 'lit23' is loaded into the PC.

The 'n' bits form the target address.

**Note:** The linker will resolve the specified expression into the lit23 to be used.

**Words:** 2

**Cycles:** 2

**Example 1:**

026000 CALL _FIR ; Call _FIR subroutine
026004 MOV W0, W1

... ; Call _FIR subroutine start

026844 _FIR: MOV #0x400, W2
026846 ...

**Before Instruction** | **After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>02 6000</th>
<th>PC</th>
<th>02 6844</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15</td>
<td>A268</td>
<td>W15</td>
<td>A26C</td>
</tr>
<tr>
<td>Data A268</td>
<td>FFFF</td>
<td>Data A268</td>
<td>6004</td>
</tr>
<tr>
<td>Data A26A</td>
<td>FFFF</td>
<td>Data A26A</td>
<td>0002</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:  
072000         CALL    _G66         ; call routine _G66  
072004         MOV     W0, W1  
               .           ...  
077A28 _G66:   INC     W6, [W7++]   ; routine start  
077A2A        .           ...  
077A2C

Before Instruction                                      After Instruction
PC  07 2000                                               PC  07 7A28
W15     9004                                               W15     9008
Data 9004 FFFF                                             Data 9004 2004
Data 9006 FFFF                                             Data 9006 0007
SR  0000                                                  SR  0000

CALL Call Subroutine

Implemented in:   PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E
                   X         X

Syntax:           {label:} CALL Expr

Operands:         Expr may be a label or expression (but not a literal).
                  Expr is resolved by the linker to a lit23, where lit23 ∈ [0 ... 8388606].

Operation:        (PC) + 4 ➝ PC
                  (PC<15:1>) ➝ TOS<15:1>, SFA bit ➝ TOS<0>
                  (W15) + 2 ➝ W15
                  (PC<23:16>) ➝ TOS
                  (W15) + 2 ➝ W15
                  0 ➝ SFA bit
                  lit23 ➝ PC
                  NOP ➝ Instruction Register

Status Affected:  SFA

Encoding:         1st word 0000 0010 nnnn nnnn nnnn nnnn 0
                  2nd word 0000 0000 0000 0000 0nnn nnnn

Description:      Direct subroutine call over the entire 4-Mbyte instruction program memory range. Before the CALL is made, the 24-bit return address (PC + 4) is PUSHed onto the stack. After the return address is stacked, the 23-bit value 'lit23' is loaded into the PC.

The 'n' bits form the target address.

Note: The linker will resolve the specified expression into the lit23 to be used.

Words:  2
Cycles: 4
### Section 5. Instruction Descriptions

**Example 1:**

```
Example 1:
026000     CALL   _FIR          ; Call _FIR subroutine
026004     MOV    W0, W1
.           ...
.           ...
026844 _FIR: MOV   #0x400, W2   ; _FIR subroutine start
026846     ...
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>02 6000</td>
<td>02 6844</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>A268</td>
<td>A26C</td>
</tr>
<tr>
<td>Data A268</td>
<td>Data A268</td>
</tr>
<tr>
<td>FFFF</td>
<td>6004</td>
</tr>
<tr>
<td>Data A26A</td>
<td>Data A26A</td>
</tr>
<tr>
<td>FFFF</td>
<td>0002</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
Example 2:
072000     CALL    _G66         ; call routine _G66
072004     MOV     W0, W1
.           ...
077A28 _G66: INC     W6, [W7++] ; routine start
077A2A     ...
077A2C
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>07 2000</td>
<td>07 7A28</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>9004</td>
<td>9008</td>
</tr>
<tr>
<td>Data 9004</td>
<td>Data 9004</td>
</tr>
<tr>
<td>FFFF</td>
<td>2004</td>
</tr>
<tr>
<td>Data 9006</td>
<td>Data 9006</td>
</tr>
<tr>
<td>FFFF</td>
<td>0007</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
**CALL**  
Call Indirect Subroutine

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**  
\{label:} CALL Wn

**Operands:**  
Wn ∈ [W0 ... W15]

**Operation:**  
(PC) + 2 → PC  
(PC<15:0>) → TOS  
(W15) + 2 → W15  
(PC<23:16>) → TOS  
(W15) + 2 → W15  
0 → PC<22:16>  
(Wn<15:1>) → PC<15:1>  
NOP → Instruction Register

**Status Affected:** None

**Encoding:**  
| 0000 | 0001 | 0000 | 0000 | 0000 | ssss |

**Description:** Indirect subroutine call over the first 32K instructions of program memory. Before the CALL is made, the 24-bit return address (PC + 2) is PUSHed onto the stack. After the return address is stacked, Wn<15:1> is loaded into PC<15:1> and PC<22:16> is cleared. Since PC<0> is always '0', Wn<0> is ignored.

The 's' bits select the source register.

**Words:** 1

**Cycles:** 2

**Example 1:**
```
001002 CALL W0 ; Call BOOT subroutine indirectly
001004 ... ; using W0
... 001600 _BOOT: MOV #0x400, W2 ; _BOOT starts here
001602 MOV #0x300, W6
...  
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 1002</td>
<td>PC 00 1600</td>
</tr>
<tr>
<td>W0 1600</td>
<td>W0 1600</td>
</tr>
<tr>
<td>W15 6F00</td>
<td>W15 6F04</td>
</tr>
<tr>
<td>Data 6F00 FFFF</td>
<td>Data 6F00 1004</td>
</tr>
<tr>
<td>Data 6F02 FFFF</td>
<td>Data 6F02 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
### Example 2:
```
004200 CALL W7 ; Call TEST subroutine indirectly
004202 ... ; using W7
005500 _TEST: INC W1, W2 ; _TEST starts here
005502 DEC W1, W3 ;
```

#### Before Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>00 4200</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>5500</td>
</tr>
<tr>
<td>W15</td>
<td>6F00</td>
</tr>
<tr>
<td>Data 6F00</td>
<td>FFFF</td>
</tr>
<tr>
<td>Data 6F02</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

#### After Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>00 5500</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>5500</td>
</tr>
<tr>
<td>W15</td>
<td>6F04</td>
</tr>
<tr>
<td>Data 6F00</td>
<td>4202</td>
</tr>
<tr>
<td>Data 6F02</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

---

### CALL

**Call Indirect Subroutine**

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

**Syntax:**
```
{label:} CALL Wn
```

**Operands:**
- Wn ∈ [W0 ... W15]

**Operation:**
- (PC) + 2 → PC
- (PC<15:1>) → TOS, SFA bit → TOS<0>
- (W15) + 2 → W15
- (PC<23:16>) → TOS
- (W15) + 2 → W15
- 0 → SFA bit
- 0 → PC<22:16>
- (Wn<15:1>) → PC<15:1>
- NOP → Instruction Register

**Status Affected:**
- SFA

**Encoding:**
```
0000 0001 0000 0000 0000 sszzz
```

**Description:**
Indirect subroutine call over the first 32K instructions of program memory. Before the CALL is made, the 24-bit return address (PC + 2) is PUSHed onto the stack. After the return address is stacked, Wn<15:1> is loaded into PC<15:1> and PC<22:16> is cleared. Since PC<0> is always '0', Wn<0> is ignored.

The ‘s’ bits select the source register.

**Words:** 1

**Cycles:** 4
Example 1:

```
001002          CALL W0           ; Call BOOT subroutine indirectly
001004          ...               ; using W0
               ...               
001600 _BOOT:  MOV #0x400, W2    ; _BOOT starts here
001602          MOV #0x300, W6   
               ...               

; Call BOOT subroutine indirectly
; _BOOT starts here
```

```
<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>00 1002</td>
</tr>
<tr>
<td>W0</td>
<td>1600</td>
</tr>
<tr>
<td>W15</td>
<td>6F00</td>
</tr>
<tr>
<td>Data 6F00</td>
<td>FFFF</td>
</tr>
<tr>
<td>Data 6F02</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
```

Example 2:

```
004200          CALL W7           ; Call TEST subroutine indirectly
004202          ...               ; using W7
               ...               
005500 __TEST: INC W1, W2        ; __TEST starts here
005502          DEC W1, W3       ;
               ...               

; Call TEST subroutine indirectly
; __TEST starts here
```

```
<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>00 4200</td>
</tr>
<tr>
<td>W7</td>
<td>5500</td>
</tr>
<tr>
<td>W15</td>
<td>6F00</td>
</tr>
<tr>
<td>Data 6F00</td>
<td>FFFF</td>
</tr>
<tr>
<td>Data 6F02</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
CALL.L  Call Indirect Subroutine Long

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:  

{label:} CALL.L  Wn

Operands:  

Wn ∈ [W0, W2, W4, W6, W8, W10, W12]

Operation:

(PC) +2 → PC,  
(PC<15:1>) → TOS<15:1>, SFA bit → TOS<0>  
(W15)+2 → W15  
(PC<23:16>) → TOS,  
(W15)+2 → W15  
0 → SFA bit,  
PC<23> → PC<23> (see text); (Wn+1)<6:0> → PC<22:16>; (Wn) → PC<15:0>  
NOP → Instruction Register

Status Affected:  

SFA

Encoding:

| 0000 | 0001 | lwww | w000 | 0000 | ssss |

Description:

Indirect subroutine call to any User program memory address. First, return address (PC+2) and the state of the Stack Frame Active bit (SFA) is pushed onto the system stack, after which the SFA bit is cleared. Then, the LS 7-bits of (Wn+1) are loaded in PC<22:16>, and the 16-bit value (Wn) is loaded into PC<15:0>. PC<23> is not modified by this instruction. The contents of (Wn+1)<15:7> are ignored. The value of Wn<0> is also ignored and PC<0> is always set to 0. The ‘s’ bits specify the address of the Wn source register. The ‘w’ bits specify the address of the Wn+1 source register.

Words: 1  
Cycles: 4

Example 1:

| 026000 | CALL.L W4 | ; Call _FIR subroutine  
| 026004 | MOV W0, W1 |  
|       | ... |  
|       | ... |  
| 026844 | _FIR: MOV #0x400, W2 | ; _FIR subroutine start  
| 026846 | ... |

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Instruction</td>
</tr>
<tr>
<td>PC</td>
<td>02 6000</td>
</tr>
<tr>
<td>W4</td>
<td>6844</td>
</tr>
<tr>
<td>W5</td>
<td>0002</td>
</tr>
<tr>
<td>W15</td>
<td>A268</td>
</tr>
<tr>
<td>Data A268</td>
<td>FFFF</td>
</tr>
<tr>
<td>Data A26A</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
## CLR

**Clear f or WREG**

### Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Syntax:

```
{label:} CLR(.B) f
```

```
WREG
```

### Operands:

`f ∈ [0 ... 8191]`

### Operation:

`0` → destination designated by D

### Status Affected:

None

### Encoding:

```
1110 1111 0BDf ffff ffff ffff
```

### Description:

Clear the contents of a file register or the default working register WREG. If WREG is specified, the WREG is cleared. Otherwise, the specified file register `f` is cleared.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

**Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**2:** The WREG is set to working register W0.

### Words:

1

### Cycles:

1

### Example 1:

```
CLR.B RAM200 ; Clear RAM200 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM200 8009</td>
<td>RAM200 8000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

### Example 2:

```
CLR WREG ; Clear WREG (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG 0600</td>
<td>WREG 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

CLR

Clear Wd

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented in:</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label;} CLR{.B} Wd

- [Wd]
- [Wd++]
- [Wd--]
- [++Wd]
- [--Wd]

Operands: Wd ∈ [W0 ... W15]

Operation: 0 → Wd

Status Affected: None

Encoding:

| 1110 | 1011 | OBqq | qddd | d000 | 0000 |

Description:
Clear the contents of register Wd. Either register direct or indirect
addressing may be used for Wd.

The ‘B’ bit select byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.

Note: The extension .B in the instruction denotes a byte operation
rather than a word operation. You may use a .W extension to
denote a word operation, but it is not required.

Words: 1

Cycles: 1

Example 1:
CLR.B W2
; Clear W2 (Byte mode)

Before Instruction | After Instruction
--- | ---
W2 | 3300
SR | 0000

Example 2:
CLR [W0++]
; Clear [W0]
; Post-increment W0

Before Instruction | After Instruction
--- | ---
W0 | 2300
Data 2300 | 5607
SR | 0000

DS70157F-page 185
**CLR**

**Clear Accumulator, Prefetch Operands**

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

(label:)

CLR Acc {,[Wx],Wxd} {,[Wy],Wyd} {,AWB}

{,[Wx] + = kx,Wxd} {,[Wy] + = ky,Wyd}

{,[Wx] – = kx,Wxd} {,[Wy] – = ky,Wyd}

{,[W9 + W12],Wxd} {,[W11 + W12],Wyd}

Operands:

Acc ∈ [A,B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13, [W13] + = 2]

Operation:

0 → Acc(A or B)

([Wx]) → Wxd; (Wx) +/- kx → Wx

([Wy]) → Wyd; (Wy) +/- ky → Wy

(Acc(B or A)) rounded → AWB

Status Affected:

OA, OB, SA, SB

Encoding:

| 1100 | 0011 | A0xx | yyii | iijj | jjaa |

Description:

Clear all 40 bits of the specified accumulator, optionally prefetch operands in preparation for a MAC type instruction and optionally store the non-specified accumulator results. This instruction clears the respective overflow and saturate flags (either OA, SA or OB, SB).

Operands Wx, Wxd, Wy and Wyd specify optional prefetch operations, which support indirect and register offset addressing, as described in **Section 4.14.1 “MAC Prefetches”**. Operand AWB specifies the optional register direct or indirect store of the convergently rounded contents of the “other” accumulator, as described in **Section 4.14.4 “MAC Write Back”**.

The ‘A’ bit selects the other accumulator used for write back.
The ‘x’ bits select the prefetch Wxd destination.
The ‘y’ bits select the prefetch Wyd destination.
The ‘i’ bits select the Wx prefetch operation.
The ‘j’ bits select the Wy prefetch operation.
The ‘a’ bits select the accumulator Write Back destination.

Words: 1
Cycles: 1
### Example 1:
```
CLR A, [W8]+2, W4, W13  ; Clear ACCA
; Load W4 with [W8], post-inc W8
; Store ACCB to W13
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>F001</td>
</tr>
<tr>
<td>W8</td>
<td>2000</td>
</tr>
<tr>
<td>W13</td>
<td>C623</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0067 2345</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 5420 3BDD</td>
</tr>
<tr>
<td>Data 2000</td>
<td>1221</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W4</td>
<td>1221</td>
</tr>
<tr>
<td>W8</td>
<td>2002</td>
</tr>
<tr>
<td>W13</td>
<td>5420</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0000 0000</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 5420 3BDD</td>
</tr>
<tr>
<td>Data 2000</td>
<td>1221</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

### Example 2:
```
; Load W6 with [W8]
; Load W7 with [W10]
; Save ACCA to [W13]
; Post-inc W8,W10,W13
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>F001</td>
</tr>
<tr>
<td>W7</td>
<td>C783</td>
</tr>
<tr>
<td>W8</td>
<td>2000</td>
</tr>
<tr>
<td>W10</td>
<td>3000</td>
</tr>
<tr>
<td>W13</td>
<td>4000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0067 2345</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 5420 ABDD</td>
</tr>
<tr>
<td>Data 2000</td>
<td>1221</td>
</tr>
<tr>
<td>Data 3000</td>
<td>FF80</td>
</tr>
<tr>
<td>Data 4000</td>
<td>FF00</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W6</td>
<td>1221</td>
</tr>
<tr>
<td>W7</td>
<td>FF80</td>
</tr>
<tr>
<td>W8</td>
<td>2002</td>
</tr>
<tr>
<td>W10</td>
<td>3002</td>
</tr>
<tr>
<td>W13</td>
<td>4002</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0067 2345</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 0000 0000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>1221</td>
</tr>
<tr>
<td>Data 3000</td>
<td>FF80</td>
</tr>
<tr>
<td>Data 4000</td>
<td>0067</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
**CLRWD T**

Clear Watchdog Timer

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} CLRWD T

Operands: None

Operation:

- `0 → WDT count register`
- `0 → WDT prescaler A count`
- `0 → WDT prescaler B count`

Status Affected: None

Encoding:

```
1111 1110 0110 0000 0000 0000
```

Description: Clear the contents of the Watchdog Timer count register and the prescaler count registers. The Watchdog Prescaler A and Prescaler B settings, set by configuration fuses in the FWDT, are not changed.

Words: 1

Cycles: 1

**Example 1:**

```
CLRWDT     ; Clear Watchdog Timer
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**COM**  
Complement f

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} COM{.B} f {,WREG}

Operands:  
f ∈ [0 ... 8191]

Operation:  
(\(f\)) → destination designated by D

Status Affected:  
N, Z

Encoding:  

| 1110 | 1110 | 1BDF | fffe | fffe | fffe |

Description:  
Compute the 1’s complement of the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1\(^{(1)}\)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.

**Example 1:**  
COM.b RAM200 ; COM RAM200 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM200</td>
<td>RAM200</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>80FF</td>
<td>8000</td>
</tr>
<tr>
<td>0000</td>
<td>0002 ((Z))</td>
</tr>
</tbody>
</table>

**Example 2:**  
COM RAM400, WREG ; (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>WREG</td>
</tr>
<tr>
<td>RAM400</td>
<td>RAM400</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>1211</td>
<td>F7DC</td>
</tr>
<tr>
<td>0823</td>
<td>0823</td>
</tr>
<tr>
<td>0000</td>
<td>0008 ((N = 1))</td>
</tr>
</tbody>
</table>
**COM**

**Complement Ws**

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

```text
{label:} COM{.B} Ws, Wd

[Ws], [Wd]

[Ws++], [Wd++]

[Ws--], [Wd--]

[++Ws], [+Wd]

[--Ws], [--Wd]
```

Operands:

- \( Ws \in [W0 \ldots W15] \)
- \( Wd \in [W0 \ldots W15] \)

Operation:

\((Ws) \rightarrow Wd\)

Status Affected:

- \( N, Z \)

Encoding:

```
1110 1010 1Bqq qddd dppp ssss
```

Description:

Compute the 1’s complement of the contents of the source register \( Ws \) and place the result in the destination register \( Wd \). Either register direct or indirect addressing may be used for both \( Ws \) and \( Wd \).

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: \( 1^{(1)} \)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.

**Example 1:**

```
COM.B [W0++], [W1++] ; COM [W0] and store to [W1] (Byte mode)

; Post-increment W0, W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 2301</td>
<td>W0 2302</td>
</tr>
<tr>
<td>W1 2400</td>
<td>W1 2401</td>
</tr>
<tr>
<td>Data 2300</td>
<td>Data 2300</td>
</tr>
<tr>
<td>5607</td>
<td>5607</td>
</tr>
<tr>
<td>Data 2400</td>
<td>Data 2400</td>
</tr>
<tr>
<td>ABCD</td>
<td>ABA9</td>
</tr>
</tbody>
</table>
| SR 0000            | SR 0008           | (N = 1)
Example 2: COM W0, [W1++] ; COM W0 and store to [W1] (Word mode) ; Post-increment W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>D004</td>
</tr>
<tr>
<td>W1</td>
<td>1000</td>
</tr>
<tr>
<td>Data 1000</td>
<td>ABA9</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

CP

Compare f with WREG, Set Status Flags

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
</table>

Syntax: \{label:\} CP{.B} f

Operands: f ∈ [0 ...8191]
Operation: (f) – (WREG)
Status Affected: DC, N, OV, Z, C
Encoding: 1110 0011 0010 C000 0000 0000 0000

Description: Compute (f) – (WREG) and update the STATUS register. This instruction is equivalent to the SUBWF instruction, but the result of the subtraction is not stored.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1
Cycles: 1(4)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: CP.B RAM400 ; Compare RAM400 with WREG (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>8823</td>
</tr>
<tr>
<td>RAM400</td>
<td>0823</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2: CP 0x1200 ; Compare (0x1200) with WREG (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>2377</td>
</tr>
<tr>
<td>Data 1200</td>
<td>2277</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

(C = 1) (N = 1)
**CP**

Compare Wb with lit5, Set Status Flags

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label:\} CP\{.B\} Wb, #lit5

**Operands:**

Wb ∈ [W0 ... W15]
lit5 ∈ [ 0 ... 31]

**Operation:**

(Wb) – lit5

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

```
1110 0001 0www wB00 011k kkkk
```

**Description:**

Compute (Wb) – lit5, and update the STATUS register. This instruction is equivalent to the `SUB` instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb.

The ‘w’ bits select the address of the Wb base register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘k’ bits provide the literal operand, a five-bit integer number.

**Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1

**Example 1:**

```
CP.B W4, #0x12 ; Compare W4 with 0x12 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>7711</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W4</td>
<td>7711</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

```
CP W4, #0x12 ; Compare W4 with 0x12 (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>7713</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W4</td>
<td>7713</td>
</tr>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
## CP

**Compare Wb with lit8, Set Status Flags**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} CP(.B) Wb, #lit8

**Operands:**

Wb ∈ [W0 ... W15]
lit8 ∈ [0 ... 255]

**Operation:**

(Wb) – lit8

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

| 1110 | 0001 | 0www | wBkk | k11k | kkkk |

**Description:**

Compute (Wb) – lit8, and update the STATUS register. This instruction is equivalent to the `SUB` instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb.

The 'w' bits select the address of the Wb base register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'k' bits provide the literal operand, a five-bit integer number.

*Note:* The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:**

1

**Cycles:**

1

**Example 1:**

CP.B W4, #0x12 ; Compare W4 with 0x12 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>7711</td>
<td>7711</td>
</tr>
<tr>
<td>0000</td>
<td>0009 (N, C = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

CP W4, #0x12 ; Compare W4 with 0x12 (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>7713</td>
<td>7713</td>
</tr>
<tr>
<td>0000</td>
<td>0001 (C = 1)</td>
</tr>
</tbody>
</table>
CP

Compare Wb with Ws, Set Status Flags

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

{label:} CP{.B} Wb, Ws

[Ws]

[Ws++]

[Ws--]

[++Ws]

[--Ws]

Operands:

Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]

Operation:

(Wb) – (Ws)

Status Affected:

DC, N, OV, Z, C

Encoding:

| 1110 | 0001 | 0www | wB00 | 0ppp | ssss |

Description:

Compute (Wb) – (Ws), and update the STATUS register. This instruction is equivalent to the SUB instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws.

The 'w' bits select the address of the Wb source register.
The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'p' bits select the source Address mode.
The 's' bits select the address of the Ws source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1

Example 1:

CP.B W0, [W1++] ; Compare [W1] with W0 (Byte mode)

; Post-increment W1

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 ABA9</td>
<td>W0 ABA9</td>
</tr>
<tr>
<td>W1 2000</td>
<td>W1 2001</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0009</td>
</tr>
</tbody>
</table>

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
Example 2:  CP    W5, W6        ; Compare W6 with W5 (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5  2334</td>
<td>W5  2334</td>
</tr>
<tr>
<td>W6  8001</td>
<td>W6  8001</td>
</tr>
</tbody>
</table>
| SR  0000           | SR  000C          (N, OV = 1)
**CP0**

Compare f with 0x0, Set Status Flags

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} CP0{.B} f

Operands:

f ∈ [0 ... 8191]

Operation:

(f) – 0x0

Status Affected:

DC, N, OV, Z, C

Encoding:

```
1110 0010 0B0f ffff ffff ffff
```

Description:

Compute (f) – 0x0 and update the STATUS register. The result of the subtraction is not stored.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘f’ bits select the address of the file register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in *Section 3.2.1 “Multi-Cycle Instructions”*.

**Example 1:**

```
CP0.B  RAM100  ; Compare RAM100 with 0x0 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM100 44C3</td>
<td>RAM100 44C3</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0009</td>
</tr>
</tbody>
</table>

(N, C = 1)

**Example 2:**

```
CP0  0x1FFE  ; Compare (0x1FFE) with 0x0 (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1FFE 0001</td>
<td>Data 1FFE 0001</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001</td>
</tr>
</tbody>
</table>

(C = 1)
CP0

Compare Ws with 0x0, Set Status Flags

Implemented in:  
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} CP0{.B} Ws
         [Ws]
         [Ws++]
         [Ws--]
         [+Ws]
         [--Ws]

Operands: Ws ∈ [W0 ... W15]
Operation: (Ws) – 0x0000
Status Affected: DC, N, OV, Z, C
Encoding:

| 1110 | 0000 | 0000 | 0B00 | 0ppp | ssss |

Description: Compute (Ws) – 0x0000 and update the STATUS register. The result of the subtraction is not stored. Register direct or indirect addressing may be used for Ws.

The 'B' bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The 'p' bits select the source Address mode.
The 's' bits select the address of the Ws source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: CP0.B [W4--] ; Compare [W4] with 0 (Byte mode)
            ; Post-decrement W4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>1001</td>
<td>1000</td>
</tr>
<tr>
<td>0034</td>
<td>0034</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>(C = 1)</td>
<td></td>
</tr>
</tbody>
</table>

Example 2: CP0 [--W5] ; Compare [--W5] with 0 (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>W5</td>
</tr>
<tr>
<td>Data 23FE</td>
<td>Data 23FE</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>2400</td>
<td>23FE</td>
</tr>
<tr>
<td>9000</td>
<td>9000</td>
</tr>
<tr>
<td>0000</td>
<td>0009</td>
</tr>
<tr>
<td>(N, C = 1)</td>
<td></td>
</tr>
</tbody>
</table>
**CPB**

**Compare f with WREG using Borrow, Set Status Flags**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} CPB{.B} f
```

**Operands:**

`f ∈ [0 ...8191]`

**Operation:**

`\((f) - (WREG) - (C)\)`

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

```
1110 0011 1B0f ffff ffff ffff
```

**Description:**

Compute `(f) - (WREG) - (C)`, and update the STATUS register. This instruction is equivalent to the SUBB instruction, but the result of the subtraction is not stored.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'f' bits select the address of the file register.

**Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

3: The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

**Words:**

1

**Cycles:**

1

---

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

```c
CPB.B RAM400 ; Compare RAM400 with WREG using C (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>8823</td>
</tr>
<tr>
<td>RAM400</td>
<td>0823</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```c
CPB 0x1200 ; Compare (0x1200) with WREG using C (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>2377</td>
</tr>
<tr>
<td>Data 1200</td>
<td>2377</td>
</tr>
</tbody>
</table>
| SR                 | 0001              | (C = 1)
**Section 5. Instruction Descriptions**

**CPB**  
**Compare Wb with lit5 using Borrow, Set Status Flags**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**  
{label:} CPB(B) Wb, #lit5

**Operands:**  
Wb ∈ [W0 ... W15]  
lit5 ∈ [0 ... 31]

**Operation:**  
(Wb) – lit5 – (C)

**Status Affected:** DC, N, OV, Z, C

**Encoding:**

<table>
<thead>
<tr>
<th></th>
<th>1110</th>
<th>0001</th>
<th>lwww</th>
<th>wB00</th>
<th>01lk</th>
<th>kkkk</th>
</tr>
</thead>
</table>

**Description:**  
Compute (Wb) – lit5 – (C), and update the STATUS register. This instruction is equivalent to the SUBB instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb.

The ‘w’ bits select the address of the Wb source register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘k’ bits provide the literal operand, a five bit integer number.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

**Words:** 1  
**Cycles:** 1

**Example 1:**  
CPB.B W4, #0x12  
; Compare W4 with 0x12 using C (Byte mode)

Before Instruction  
W4 7711  
SR 0001 (C = 1)

After Instruction  
W4 7711  
SR 0008 (N = 1)

**Example 2:**  
CPB.B W4, #0x12  
; Compare W4 with 0x12 using C (Byte mode)

Before Instruction  
W4 7711  
SR 0000

After Instruction  
W4 7711  
SR 0008 (N = 1)

**Example 3:**  
CPB W12, #0x1F  
; Compare W12 with 0x1F using C (Word mode)

Before Instruction  
W12 0020  
SR 0002 (Z = 1)

After Instruction  
W12 0020  
SR 0003 (Z, C = 1)

**Example 4:**  
CPB W12, #0x1F  
; Compare W12 with 0x1F using C (Word mode)

Before Instruction  
W12 0020  
SR 0003 (Z, C = 1)

After Instruction  
W12 0020  
SR 0001 (C = 1)
CPB

Compare Wb with lit8 using Borrow, Set Status Flags

Implemented in:

- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

Syntax:

{label:} CPB(.B) Wb, #lit8

Operands:

- Wb ∈ [W0 ... W15]
- lit8 ∈ [0 ... 255]

Operation:

(Wb) – lit8 – (C)

Status Affected:

DC, N, OV, Z, C

Encoding:

1110 0001 1www wBkk k11k kkkk

Description:

Compute (Wb) – lit8 – (C), and update the STATUS register. This instruction is equivalent to the SUBB instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb.

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘k’ bits provide the literal operand, a five bit integer number.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1
Cycles: 1

Example 1:

CPB.B W4, #0x12 ; Compare W4 with 0x12 using C (Byte mode)

Before Instruction

| W4 | 7711 |
| SR | 0001 | (C = 1)

After Instruction

| W4 | 7711 |
| SR | 0008 | (N = 1)

Example 2:

CPB.B W4, #0x12 ; Compare W4 with 0x12 using C (Byte mode)

Before Instruction

| W4 | 7711 |
| SR | 0000 |

After Instruction

| W4 | 7711 |
| SR | 0008 | (N = 1)

Example 3:

CPB W12, #0x1F ; Compare W12 with 0x1F using C (Word mode)

Before Instruction

| W12 | 0020 |
| SR  | 0002 | (Z = 1)

After Instruction

| W12 | 0020 |
| SR  | 0003 | (Z, C = 1)

Example 4:

CPB W12, #0x1F ; Compare W12 with 0x1F using C (Word mode)

Before Instruction

| W12 | 0020 |
| SR  | 0003 | (Z, C = 1)

After Instruction

| W12 | 0020 |
| SR  | 0001 | (C = 1)
## Section 5. Instruction Descriptions

### CPB

**Compare Ws with Wb using Borrow, Set Status Flags**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} CPB{.B} Wb, Ws

- [Ws]
- [Ws++]
- [Ws--]
- [++Ws]
- [-Ws]

**Operands:**

- Wb ∈ [W0 ... W15]
- Ws ∈ [W0 ... W15]

**Operation:**

(Wb) – (Ws) – (C)

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

| 1110 | 0001 | lwww | WB00 | 0ppp | ssss |

**Description:**

Compute (Wb) – (Ws) – (C), and update the STATUS register. This instruction is equivalent to the SUBB instruction, but the result of the subtraction is not stored. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws.

The ‘w’ bits select the address of the Wb source register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the address of the Ws source register.

**Note 1:**

- The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

- The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

**Words:**

1

**Cycles:**

1(1)

**Note 1:**

In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

CPB.B W0, [W1++] ; Compare [W1] with W0 using (Byte mode) ; Post-increment W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 ABA9</td>
<td>W0 ABA9</td>
</tr>
<tr>
<td>W1 1000</td>
<td>W1 1001</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>D0A9</td>
<td>D0A9</td>
</tr>
<tr>
<td>SR 0002 (Z = 1)</td>
<td>SR 0008 (N = 1)</td>
</tr>
</tbody>
</table>
Example 2: CPB.B W0, [W1++] ; Compare [W1] with W0 using C (Byte mode)
; Post-increment W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0  ABA9</td>
<td>W0  ABA9</td>
</tr>
<tr>
<td>W1  1000</td>
<td>W1  1001</td>
</tr>
<tr>
<td>Data 1000  D0A9</td>
<td>Data 1000  D0A9</td>
</tr>
<tr>
<td>SR  0001 (C = 1)</td>
<td>SR  0001 (C = 1)</td>
</tr>
</tbody>
</table>

Example 3: CPB W4, W5 ; Compare W5 with W4 using C (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4  4000</td>
<td>W4  4000</td>
</tr>
<tr>
<td>W5  3000</td>
<td>W5  3000</td>
</tr>
<tr>
<td>SR  0001 (C = 1)</td>
<td>SR  0001 (C = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**CPBEQ**

**Compare Wb with Wn, Branch if Equal (Wb = Wn)**

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} CPBEQ{.B} Wb, Wn, Expr

**Operands:**

Wb ∈ [W0 ... W15]
Wn ∈ [W0 ... W15]

**Operation:**

(Wb) − (Wn)
If (Wb) = (Wn), [(PC+2) + 2 * Expr] →PC and NOP →Instruction Register

**Status Affected:** None

**Encoding:**

```
| 1110 | 0111 | lwww | wBnn | nnnn | ssss |
```

**Description:**

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) − (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded, the PC is recalculated based on the 6-bit signed offset specified by Expr, and on the next cycle, a NOP is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal (branch is not taken).

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘s’ bits select the address of the Wn source register.
The ‘n’ bits select the offset of the branch destination.

**Note:**
The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1 (5 if branch taken)

**Example 1:**

002000 HERE:CPBEQ.B W0, W1, BYPASS; If W0 = W1 (Byte mode),
002002 ADD W2, W3, W4; Perform branch to BYPASS
002004 . . .
002006 . . .
002008 BYPASS: . . .
00200A . . .

### Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>1000</td>
</tr>
<tr>
<td>W1</td>
<td>1000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

### After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>1000</td>
</tr>
<tr>
<td>W1</td>
<td>1000</td>
</tr>
<tr>
<td>SR</td>
<td>0002(z = 1)</td>
</tr>
</tbody>
</table>
## CPBGT

### Signed Compare Wb with Wn, Branch if Greater Than (Wb > Wn)

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} CPBGT{.B} Wb, Wn, Expr

**Operands:**

Wb ∈ [W0 ... W15]
Wn ∈ [W0 ... W15]

**Operation:**

(Wb) – (Wn)

If (Wb) = (Wn), [(PC+2) + 2 * Expr] → PC and NOP → Instruction Register

**Status Affected:** None

**Encoding:**

```
1110 0110 0www wBnn nnnn ssss
```

**Description:**

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded, the PC is recalculated based on the 6-bit signed offset specified by Expr, and on the next cycle, a NOP is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal (branch is not taken).

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘s’ bits select the address of the Wn source register.
The ‘n’ bits select the offset of the branch destination.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1 (5 if branch taken)

### Example 1:

002000 HERE: CPBGT.B W0, W1, BYPASS ; If W0 > W1 (Byte mode),
002002        ADD W2, W3, W4 ; Perform branch to BYPASS
002004 . . .
002006 . . .
002008 BYPASS . . .
00200A . . .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2008</td>
</tr>
<tr>
<td>W0 30FF</td>
<td>W0 00FF</td>
</tr>
<tr>
<td>W1 26FE</td>
<td>W1 26FE</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000 (N, C = 0)</td>
</tr>
</tbody>
</table>
### CPBLT

**Signed Compare Wb with Wn, Branch if Less Than (Wb < Wn)**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Syntax:** `{label:} CPBLT{.B} Wb, Wn, Expr
- **Operands:** Wb ∈ [W0 ... W15]  
  Wn ∈ [W0 ... W15]
- **Operation:** (Wb) – (Wn)  
  If (Wb) = (Wn), [(PC+2) + 2 * Expr] →PC and NOP →Instruction Register  
  If (Wb) ≠ (Wn), the next instruction is executed as normal (branch is not taken).
- **Status Affected:** None
- **Encoding:**
  ```
  1110 0110 1WWW 1WNN 1LLL
  ```
- **Description:** Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded, the PC is recalculated based on the 6-bit signed offset specified by Expr, and on the next cycle, a NOP is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal (branch is not taken).

  The ‘w’ bits select the address of the Wb source register.  
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).  
The ‘s’ bits select the address of the Wn source register.  
The ‘n’ bits select the offset of the branch destination.

  **Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

- **Words:** 1
- **Cycles:** 1 (5 if branch taken)

**Example 1:**

002000 HERE: CPBLT.B W8, W9, BYPASS; If W8 < W9 (Byte mode),  
002002 ADD W2, W3, W4; Perform branch to BYPASS  
002004 ...  
002006 ...  
002008 BYPASS: ...  
00200A ...  

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>W8</td>
<td>W8</td>
</tr>
<tr>
<td>W9</td>
<td>W9</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>00 2000</td>
<td>00 2008</td>
</tr>
<tr>
<td>00FF</td>
<td>00FF</td>
</tr>
<tr>
<td>26FE</td>
<td>26FE</td>
</tr>
<tr>
<td>0000</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
CPBNE Compare Wb with Wn, Branch if Not Equal (Wb ≠ Wn)

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

\{(label: \) CPBNE(.B) Wb, Wn, Expr \}

Operands: 

Wb ∈ [W0 ... W15]  
Wn ∈ [W0 ... W15]

Operation: 

(Wb) – (Wn)  
If (Wb) = (Wn), [(PC+2) + 2 * Expr] → PC and NOP → Instruction Register  
If (Wb) ≠ (Wn), the next instruction is executed as normal (branch is not taken).

Status Affected: None

Encoding: 

1110 0111 0www wBnn nnnn ssss

Description: 

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded, the PC is recalculated based on the 6-bit signed offset specified by Expr, and on the next cycle, a NOP is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal (branch is not taken).

The ‘w’ bits select the address of the Wb source register. 
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). 
The ‘s’ bits select the address of the Wn source register. 
The ‘n’ bits select the offset of the branch destination.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1  
Cycles: 1 (5 if branch taken)

Example 1: 

002000 HERE:   CPBNE.B W2, W3, BYPASS ; If W2 !- W3 (Byte mode),  
002002 ADD W2, W3, W4 ; Perform branch to BYPASS  
002004 . . .  
002006 . . .  
002008 BYPASS: . . .  
00200A . . .

Before Instruction | After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>W2</th>
<th>W3</th>
<th>SR</th>
<th>PC</th>
<th>W2</th>
<th>W3</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>002000</td>
<td>00FF</td>
<td>26FE</td>
<td>0000</td>
<td>00200A</td>
<td>00FF</td>
<td>26FE</td>
<td>0001</td>
</tr>
</tbody>
</table>

(C = 1)
Section 5. Instruction Descriptions

**CPSEQ**

**Compare Wb with Wn, Skip if Equal (Wb = Wn)**

Implemented in:

<table>
<thead>
<tr>
<th>Family</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

\{label:\} CPSEQ{.B} Wb, Wn

Operands:

- Wb ∈ [W0 ... W15]
- Wn ∈ [W0 ... W15]

Operation:

- (Wb) – (Wn)
- Skip if (Wb) = (Wn)

Status Affected:

None

Encoding:

<table>
<thead>
<tr>
<th>0000</th>
<th>0001</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>wWw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td>ssss</td>
</tr>
</tbody>
</table>

Description:

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a **NOP** is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘s’ bits select the address of the Wn source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1 (2 or 3 if skip taken)

**Example 1:**

002000 HERE: CPSEQ.BW0, W1; If W0 = W1 (Byte mode),
002002 GOTO BYPASS; skip the GOTO
002004 . . .
002006 . . .
002008 BYPASS: . . .
00200A . . .

**Example 2:**

018000 HERE: CPSEQ W4, W8; If W4 = W8 (Word mode),
018002 CALL _FIR; skip the subroutine call
018006 . . .
018008 . . .

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>00 2000</td>
</tr>
<tr>
<td>W0</td>
<td>1001</td>
</tr>
<tr>
<td>W1</td>
<td>1000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>01 8000</td>
</tr>
<tr>
<td>W4</td>
<td>3344</td>
</tr>
<tr>
<td>W8</td>
<td>3344</td>
</tr>
<tr>
<td>SR</td>
<td>0002 (Z = 1)</td>
</tr>
</tbody>
</table>
**CPSEQ**

Compare Wb with Wn, Skip if Equal (Wb = Wn)

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{(label:)} CPSEQ\{.B\} Wb, Wn

**Operands:**

Wb ∈ \{W0 ... W15\}
Wn ∈ \{W0 ... W15\}

**Operation:**

(Wb) – (Wn)
Skip if (Wb) = (Wn)

**Status Affected:** None

**Encoding:**

| 1110 | 0111 | lwww | wB00 | 0001 | ssss |

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) = (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a \texttt{NOP} is executed instead. If (Wb) ≠ (Wn), the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘s’ bits select the address of the Wn source register.

**Note:** The extension \texttt{.B} in the instruction denotes a byte operation rather than a word operation. You may use a \texttt{.W} extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1 (2 or 3 if skip taken)

**Example 1:**

```
002000 HERE:CPSEQ.BW0, W1; If W0 = W1 (Byte mode),
002002GOTOBYPASS; skip the GOTO
002004 . . .
002006 . . .
002008 BYPASS: . . .
00200A . . .
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>W0 1001</td>
<td>W0 1001</td>
</tr>
<tr>
<td>W1 1000</td>
<td>W1 1000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Example 2: 018000 HERE: CPSEQ W4, W8; If W4 = W8 (Word mode),
018002 CALL _FIR; skip the subroutine call
018006 ... 
018008 ...

Before
Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>01 8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>3344</td>
</tr>
<tr>
<td>W8</td>
<td>3344</td>
</tr>
<tr>
<td>SR</td>
<td>0002 (Z = 1)</td>
</tr>
</tbody>
</table>

After
Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>01 8006</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>3344</td>
</tr>
<tr>
<td>W8</td>
<td>3344</td>
</tr>
<tr>
<td>SR</td>
<td>0002 (Z = 1)</td>
</tr>
</tbody>
</table>
CPSGT  Signed Compare Wb with Wn, Skip if Greater Than (Wb > Wn)

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Syntax:  
{lable:} CPSGT{.B} Wb, Wn

Operands:  
Wb ∈ [W0 ... W15]  
Wn ∈ [W0 ... W15]

Operation:  
(Wb) – (Wn)  
Skip if (Wb) > (Wn)

Status Affected:  
None

Encoding:  
1110 0110 0www wB00 0000 ssss

Description:  
Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) > (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. Otherwise, the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.  
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).  
The ‘s’ bits select the address of the Wn source register.

Note:  
The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words:  1
Cycles:  1 (2 or 3 if skip taken)

Example 1:  
002000 HERE: CPSGT.B W0, W1; If W0 > W1 (Byte mode),  
002002    GOTO      BYPASS; skip the GOTO  
002006    . . .  
002008    . . .  
00200A BYPASS . . .  
00200C . . .

Before Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
<th>W0</th>
<th>00FF</th>
<th>W1</th>
<th>26FE</th>
<th>SR</th>
<th>0009 (N, C = 1)</th>
</tr>
</thead>
</table>

After Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>00 2006</th>
<th>W0</th>
<th>00FF</th>
<th>W1</th>
<th>26FE</th>
<th>SR</th>
<th>0009 (N, C = 1)</th>
</tr>
</thead>
</table>

Example 2:  
018000 HERE: CPSGT W4, W5; If W4 > W5 (Word mode),  
018002    CALL      _FIR; skip the subroutine call  
018006    . . .  
018008    . . .

Before Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>01 8000</th>
<th>W4</th>
<th>2600</th>
<th>W5</th>
<th>2600</th>
<th>SR</th>
<th>0004 (OV = 1)</th>
</tr>
</thead>
</table>

After Instruction
<table>
<thead>
<tr>
<th>PC</th>
<th>01 8002</th>
<th>W4</th>
<th>2600</th>
<th>W5</th>
<th>2600</th>
<th>SR</th>
<th>0004 (OV = 1)</th>
</tr>
</thead>
</table>
CPSGT  Signed Compare Wb with Wn, Skip if Greater Than (Wb > Wn)

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} CPSGT{.B} Wb, Wn

**Operands:**

Wb ∈ [W0 ... W15]

Wn ∈ [W0 ... W15]

**Operation:**

(Wb) – (Wn)

Skip if (Wb) > (Wn)

**Status Affected:**

None

**Encoding:**

```
1110 0110 0www wB00 0001 ssss
```

**Description:**

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) > (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a **NOP** is executed instead. Otherwise, the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.

The ‘B’ bit selects byte or word operation (’0’ for word, ’1’ for byte).

The ‘s’ bits select the address of the Wn source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1 (2 or 3 if skip taken)

**Example 1:**

```
002000 HERE:  CPSGT.B  W0, W1; If W0 > W1 (Byte mode),
002002        GOTO      BYPASS; skip the GOTO
002006        .  .  .
002008        .  .  .
00200A BYPASS .  .  .
00200C        .  .  .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>W0</th>
<th>W1</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>002000</td>
<td>00FF</td>
<td>26FE</td>
<td>0009 (N, C = 1)</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>W0</th>
<th>W1</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>002006</td>
<td>00FF</td>
<td>26FE</td>
<td>0009 (N, C = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

```
018000 HERE:  CPSGT  W4, W5; If W4 > W5 (Word mode),
018002        CALL      _FIR; skip the subroutine call
018006        .  .  .
018008        .  .  .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>W4</th>
<th>W5</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>018000</td>
<td>2600</td>
<td>2600</td>
<td>0004 (OV = 1)</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>W4</th>
<th>W5</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>018002</td>
<td>2600</td>
<td>2600</td>
<td>0004 (OV = 1)</td>
</tr>
</tbody>
</table>
CPSLT Signed Compare Wb with Wn, Skip if Less Than (Wb < Wn)

### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Syntax:

\{(label:)} CPSLT{.B} Wb, Wn

### Operands:

- \( Wb \in [W0 \ldots W15] \)
- \( Wn \in [W0 \ldots W15] \)

### Operation:

- \( (Wb) - (Wn) \)
- Skip if \( (Wb) < (Wn) \)

### Status Affected:

None

### Encoding:

| 1110 | 0110 | lwww | wB00 | 0000 | ssss |

### Description:

Compare the contents of Wb with the contents of Wn by performing the subtraction \((Wb) - (Wn)\), but do not store the result. If \((Wb) < (Wn)\), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. Otherwise, the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘s’ bits select the address of the Wn source register.

**Note:** The extension \(.B\) in the instruction denotes a byte operation rather than a word operation. You may use a \(.W\) extension to denote a word operation, but it is not required.

### Words:

1

### Cycles:

1 (2 or 3 if skip taken)

#### Example 1:

002000 HERE: CPSLT.B W8, W9; If W8 < W9 (Byte mode),
002002 GOTO BYPASS; skip the GOTO
002006 . . .
002008 . . .
00200A BYPASS: . . .
00200C . . .

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8</td>
<td>00FF</td>
</tr>
<tr>
<td>W9</td>
<td>26FE</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8</td>
<td>00FF</td>
</tr>
<tr>
<td>W9</td>
<td>26FE</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>

#### Example 2:

018000 HERE: CPSLT W3, W6; If W3 < W6 (Word mode),
018002 CALL _FIR; skip the subroutine call
018006 . . .
018008 . . .

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>01 8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>2600</td>
</tr>
<tr>
<td>W6</td>
<td>3000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>01 8006</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>2600</td>
</tr>
<tr>
<td>W6</td>
<td>3000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### CPSLT

**Signed Compare Wb with Wn, Skip if Less Than (Wb < Wn)**

**Implemented in:**

- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

**Syntax:**

\{label:\} CPSLT{.B} Wb, Wn

**Operands:**

- Wb ∈ \{W0 ... W15\}
- Wn ∈ \{W0 ... W15\}

**Operation:**

(Wb) – (Wn)
Skip if (Wb) < (Wn)

**Status Affected:**

None

**Encoding:**

```
1110 0110 1www wB00 0001 ssss
```

**Description:**

Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) < (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a **NOP** is executed instead. Otherwise, the next instruction is executed as normal.

The ‘w’ bits select the address of the Wb source register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘s’ bits select the address of the Wn source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1 (2 or 3 if skip taken)

**Example 1:**

```
002000 HERE:   CPSLT.B   W8, W9; If W8 < W9 (Byte mode),
002002        GOTO      BYPASS; skip the GOTO
002006         . . .
002008         . . .
00200A BYPASS: . . .
00200C         . . .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8</td>
<td>00FF</td>
</tr>
<tr>
<td>W9</td>
<td>26FE</td>
</tr>
<tr>
<td>SR</td>
<td>0008(N = 1)</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8</td>
<td>00FF</td>
</tr>
<tr>
<td>W9</td>
<td>26FE</td>
</tr>
<tr>
<td>SR</td>
<td>0008(N = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

```
018000 HERE:  CPSLT    W3, W6; If W3 < W6 (Word mode),
018002        CALL     _FIR; skip the subroutine call
018006         . . .
018008         . . .
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>01 8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>2600</td>
</tr>
<tr>
<td>W6</td>
<td>3000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>01 8006</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>2600</td>
</tr>
<tr>
<td>W6</td>
<td>3000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
CPSNE

Signed Compare \( W_b \) with \( W_n \), Skip if Not Equal (\( W_b \neq W_n \))

Implemented in:  
\[
\begin{array}{ccccccc}
\text{PIC24F} & \text{PIC24H} & \text{PIC24E} & \text{dsPIC30F} & \text{dsPIC33F} & \text{dsPIC33E} \\
X & X & X & X & X & X \\
\end{array}
\]

Syntax:  
\[
\{\text{label:}\} \quad \text{CPSNE}\{.B\} \quad W_b, \quad W_n
\]

Operands:  
\( W_b \in [W0 \ldots W15] \)  
\( W_n \in [W0 \ldots W15] \)

Operation:  
\( (W_b) - (W_n) \)  
Skip if \( (W_b) \neq (W_n) \)

Status Affected: None

Encoding:  
\[
\begin{array}{cccccccc}
1110 & 0111 & 0\text{www} & wB00 & 0000 & \text{ssss} \\
\end{array}
\]

Description: Compare the contents of \( W_b \) with the contents of \( W_n \) by performing the subtraction \( (W_b) - (W_n) \), but do not store the result. If \( (W_b) \neq (W_n) \), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a \textit{NOP} is executed instead. Otherwise, the next instruction is executed as normal.

The ‘w’ bits select the address of the \( W_b \) source register.  
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).  
The ‘s’ bits select the address of the \( W_n \) source register.

Note: The extension \( .B \) in the instruction denotes a byte operation rather than a word operation. You may use a \( .W \) extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1 (2 or 3 if skip taken)

Example 1:  
\[
\begin{array}{l}
002000 \text{ HERE: CPSNE.B W2, W3 ; If W2 !- W3 (Byte mode),} \\
002002 \text{ GOTO BYPASS ; skip the GOTO} \\
002006 \quad \ldots \\
002008 \quad \ldots \\
00200A \text{ BYPASS: \ldots} \\
00200C \quad \ldots \\
\end{array}
\]

Before Instruction  
\[
\begin{array}{l}
\text{PC} \quad 002000 \\
W2 \quad 00FF \\
W3 \quad 26FE \\
\text{SR} \quad 0001 (C = 1)
\end{array}
\]

After Instruction  
\[
\begin{array}{l}
\text{PC} \quad 002006 \\
W2 \quad 00FF \\
W3 \quad 26FE \\
\text{SR} \quad 0001 (C = 1)
\end{array}
\]

Example 2:  
\[
\begin{array}{l}
018000 \text{ HERE: CPSNE W0, W8 ; If W0 !- W8 (Word mode),} \\
018002 \text{ CALL _FIR ; skip the subroutine call} \\
018006 \quad \ldots \\
018008 \quad \ldots \\
\end{array}
\]

Before Instruction  
\[
\begin{array}{l}
\text{PC} \quad 018000 \\
W0 \quad 3000 \\
W8 \quad 3000 \\
\text{SR} \quad 0000
\end{array}
\]

After Instruction  
\[
\begin{array}{l}
\text{PC} \quad 018002 \\
W0 \quad 3000 \\
W8 \quad 3000 \\
\text{SR} \quad 0000
\end{array}
\]
Section 5. Instruction Descriptions

CPSNE
Signed Compare Wb with Wn, Skip if Not Equal (Wb ≠ Wn)

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  {label:} CPSNE{.B} Wb, Wn

Operands:  Wb ∈ [W0 ... W15]
Wn ∈ [W0 ... W15]

Operation:  (Wb) – (Wn)
Skip if (Wb) ≠ (Wn)

Status Affected:  None

Encoding:

```
1110 0111 0www wB00 0001 ssss
```

Description: Compare the contents of Wb with the contents of Wn by performing the subtraction (Wb) – (Wn), but do not store the result. If (Wb) ≠ (Wn), the next instruction (fetched during the current instruction execution) is discarded and on the next cycle, a NOP is executed instead. Otherwise, the next instruction is executed as normal.

The 'w' bits select the address of the Wb source register.
The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 's' bits select the address of the Wn source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words:  1
Cycles:  1 (2 or 3 if skip taken)

Example 1:

```
002000 HERE:   CPSNE.B  W2, W3 ; If W2 != W3 (Byte mode),
002002         GOTO     BYPASS ; skip the GOTO
002006         . . .
002008         . . .
00200A BYPASS: . . .
00200C         . . .
```

Before Instruction | After Instruction
---|---
PC 00 2000 | PC 00 2006
W2 00FF | W2 00FF
W3 26FE | W3 26FE
SR 0001 | SR 0001

Example 2:

```
018000 HERE:   CPSNE  W0, W8  ; If W0 != W8 (Word mode),
018002         CALL   _FIR  ; skip the subroutine call
018006         . . .
018008         . . .
```

Before Instruction | After Instruction
---|---
PC 01 8000 | PC 01 8002
W0 3000 | W0 3000
W8 3000 | W8 3000
SR 0000 | SR 0000
DAW.B  

Decimal Adjust Wn

Implemented in:  

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

{label:} DAW.B Wn

Operands:  

Wn ∈ [W0 ... W15]

Operation:  

If (Wn<3:0> > 9) or (DC = 1)  
  (Wn<3:0>) + 6 → Wn<3:0>

Else  
  (Wn<3:0>) → Wn<3:0>

If (Wn<7:4> > 9) or (C = 1)  
  (Wn<7:4>) + 6 → Wn<7:4>

Else  
  (Wn<7:4>) → Wn<7:4>

Status Affected:  

C

Encoding:  

```
1111 1101 0100 0000 0000 ssss
```

Description:  

Adjust the Least Significant Byte in Wn to produce a binary coded decimal (BCD) result. The Most Significant Byte of Wn is not changed, and the Carry flag is used to indicate any decimal rollover. Register direct addressing must be used for Wn.

The 's' bits select the source/destination register.

**Note 1:** This instruction is used to correct the data format after two packed BCD bytes have been added.

**2:** This instruction operates in Byte mode only and the .B extension must be included with the opcode.

Words: 1

Cycles: 1

Example 1:  

```
DAW.B W0 ; Decimal adjust W0
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 771A</td>
<td>W0 7720</td>
</tr>
<tr>
<td>SR 0002 (DC = 1)</td>
<td>SR 0002 (DC = 1)</td>
</tr>
</tbody>
</table>

Example 2:  

```
DAW.B W3 ; Decimal adjust W3
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3 77AA</td>
<td>W3 7710</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>
DEC  Decrement f

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E
               X      X      X      X      X      X

 Syntax:  {label:} DEC{.B} f {,WREG}

 Operands:  f ∈ [0 ... 8191]

 Operation:  (f) – 1 → destination designated by D

 Status Affected:  DC, N, OV, Z, C

 Encoding:  

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>1101</td>
<td>0BDF</td>
<td>ffff</td>
<td>ffff</td>
<td>ffff</td>
<td></td>
</tr>
</tbody>
</table>

 Description:  Subtract one from the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

 The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
 The 'D' bit selects the destination ('0' for WREG, '1' for file register).
 The 'f' bits select the address of the file register.

 Note 1:  The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

 Words:  1
 Cycles:  1

 Note 1:  In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

 Example 1:  DEC.B  0x200 ; Decrement (0x200) (Byte mode)

 Before Instruction          After Instruction
 Data 200 80FF                Data 200 80FE
 SR 0000                         SR 0009 (N, C = 1)

 Example 2:  DEC  RAM400, WREG ; Decrement RAM400 and store to WREG ; (Word mode)

 Before Instruction          After Instruction
 WREG 1211                    WREG 0822
 RAM400 0823                  RAM400 0823
 SR 0000                      SR 0000
DEC Decrement Ws

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E
---|---|---|---|---|---|---
X | X | X | X | X | X | X

Syntax: 

```
[label:] DEC{.B} Ws, Wd

[Ws], [Wd]

[Ws++,] [Wd++]

[Ws--], [Wd--]

[++Ws], [++Wd]

[--Ws], [--Wd]
```

Operands: 

- **Ws**: ∈ [W0 ... W15]
- **Wd**: ∈ [W0 ... W15]

Operation: 

\( (Ws) - 1 \rightarrow Wd \)

Status Affected: DC, N, OV, Z, C

Encoding: 

```
1110 1001 0Bqq qddd dppp ssss
```

Description: Subtract one from the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used by Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension {.B} in the instruction denotes a byte operation rather than a word operation. You may use a {.W} extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

```
DEC.B [W7++], [W8++] ; DEC [W7] and store to [W8] (Byte mode)
; Post-increment W7, W8
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>2301</td>
</tr>
<tr>
<td>W8</td>
<td>2400</td>
</tr>
<tr>
<td>Data 2300</td>
<td>5607</td>
</tr>
<tr>
<td>Data 2400</td>
<td>ABCD</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
**Example 2:** DEC W5, [W6++] ; Decrement W5 and store to [W6] (Word mode)
; Post-increment W6

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>W5</td>
</tr>
<tr>
<td>D004</td>
<td>D004</td>
</tr>
<tr>
<td>W6</td>
<td>W6</td>
</tr>
<tr>
<td>2000</td>
<td>2002</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>ABA9</td>
<td>D003</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0009</td>
</tr>
</tbody>
</table>

(N, C = 1)
DEC2

Decrement f by 2

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} DEC2{.B} f {,WREG}

Operands:

f ∈ [0 ... 8191]

Operation:

(f) – 2 → destination designated by D

Status Affected:

DC, N, OV, Z, C

Encoding:

1110 1101 1BDF ffff ffff ffff

Description:

Subtract two from the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

Note:

The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

Note 1:

In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

DEC2.B 0x200 ; Decrement (0x200) by 2 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 200</td>
<td>Data 200</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
<tr>
<td>80FF</td>
<td>80FD</td>
</tr>
<tr>
<td>(N, C = 1)</td>
<td></td>
</tr>
</tbody>
</table>

Example 2:

DEC2 RAM400, WREG ; Decrement RAM400 by 2 and ; store to WREG (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>WREG</td>
</tr>
<tr>
<td>RAM400</td>
<td>RAM400</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
<tr>
<td>1211</td>
<td>0821</td>
</tr>
<tr>
<td>0823</td>
<td>0823</td>
</tr>
<tr>
<td>(N, C = 0)</td>
<td></td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

DEC2  Decrement Ws by 2

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

\{label:\} DEC2{.B} [Ws], [Wd]

Operands:  

Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation:  

(Ws) – 2 → Wd

Status Affected:  
DC, N, OV, Z, C

Encoding:

| 1110 | 1001 | 1Bqq | qddd | dppp | ssss |

Description:  
Subtract two from the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used by Ws and Wd.

- The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
- The ‘q’ bits select the destination Address mode.
- The ‘d’ bits select the destination register.
- The ‘p’ bits select the source Address mode.
- The ‘s’ bits select the source register.

Note:  
The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words:  1
Cycles:  1(1)

Note 1:  In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:  
DEC2.B [W7--], [W8--]; DEC [W7] by 2, store to [W8] (Byte mode)
; Post-decrement W7, W8

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W7  2301</td>
<td>W7  2300</td>
</tr>
<tr>
<td>W8  2400</td>
<td>W8  23FF</td>
</tr>
<tr>
<td>Data 2300 0107</td>
<td>Data 2300 0107</td>
</tr>
<tr>
<td>Data 2400 ABCD</td>
<td>Data 2400 ABFF</td>
</tr>
<tr>
<td>SR  0000</td>
<td>SR  0008 (N = 1)</td>
</tr>
</tbody>
</table>
; Post-increment W6

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>D004</td>
</tr>
<tr>
<td>W6</td>
<td>1000</td>
</tr>
<tr>
<td>Data 1000</td>
<td>ABA9</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W5</th>
<th>D004</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>1002</td>
</tr>
<tr>
<td>Data 1000</td>
<td>D002</td>
</tr>
<tr>
<td>SR</td>
<td>0009 (N, C = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

### DISI

**Disable Interrupts Temporarily**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} DISI #lit14

**Operands:**

lit14 ∈ [0 ... 16383]

**Operation:**

lit14 → DISICNT
1 → DISI

Disable interrupts for (lit14 + 1) cycles

**Status Affected:** None

**Encoding:**

1111 1100 00kk kkkk kkkk kkkk

**Description:**

Disable interrupts of priority 0 through priority 6 for (lit14 + 1) instruction cycles. Priority 0 through priority 6 interrupts are disabled starting in the cycle that DISI executes, and remain disabled for the next (lit 14) cycles. The lit14 value is written to the DISICNT register, and the DISI flag (INTCON2<14>) is set to ‘1’. This instruction can be used before executing time critical code, to limit the effects of interrupts.

**Note 1:** This instruction does not prevent priority 7 interrupts and traps from running. See the specific device family reference manual for details.

**Note 2:** This instruction does not prevent any interrupts when the device is in Sleep mode.

**Words:** 1

**Cycles:** 1

**Example 1:**

002000 HERE: DISI #100 ; Disable interrupts for 101 cycles
002002 ; next 100 cycles protected by DISI
002004 . . .

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2002</td>
</tr>
<tr>
<td>DISICNT 0000</td>
<td>DISICNT 0100</td>
</tr>
<tr>
<td>INTCON2 0000</td>
<td>INTCON2 4000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

(DISI = 1)
DIV.S Signed Integer Divide

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} DIV.S(W) Wm, Wn
DIV.SD Wm, Wn

Operands: Wm ∈ [W0 ... W15] for word operation
Wm ∈ [W0, W2, W4 ... W14] for double operation
Wn ∈ [W2 ... W15]

Operation: For word operation (default):
Wm → W0
If (Wm<15> = 1):
0xFFFF → W1
Else:
0x0 → W1
W1:W0 / Wn → W0
Remainder → W1

For double operation (DIV.SD):
Wm + 1:Wm → W1:W0
W1:W0 / Wn → W0
Remainder → W1

Status Affected: N, OV, Z, C

Encoding: | 1101 | 1000 | 0ttt | tvvv | vW00 | ssss |

Description: Iterative, signed integer divide, where the dividend is stored in Wm (for a 16-bit by 16-bit divide) or Wm + 1:Wm (for a 32-bit by 16-bit divide) and the divisor is stored in Wn. In the default word operation, Wm is first copied to W0 and sign-extended through W1 to perform the operation. In the double operation, Wm + 1:Wm is first copied to W1:W0. The 16-bit quotient of the divide operation is stored in W0, and the 16-bit remainder is stored in W1.

This instruction must be executed 18 times using the REPEAT instruction (with an iteration count of 17) to generate the correct quotient and remainder. The N flag will be set if the remainder is negative and cleared otherwise. The OV flag will be set if the divide operation resulted in an overflow and cleared otherwise. The Z flag will be set if the remainder is ‘0’ and cleared otherwise. The C flag is used to implement the divide algorithm and its final value should not be used. The ‘t’ bits select the most significant word of the dividend for the double operation. These bits are clear for the word operation. The ‘v’ bits select the least significant word of the dividend. The ‘W’ bit selects the dividend size (‘0’ for 16-bit, ‘1’ for 32-bit). The ‘s’ bits select the divisor register.
Note 1: The extension .D in the instruction denotes a double word (32-bit) dividend rather than a word dividend. You may use a .W extension to denote a word operation, but it is not required.

2: Unexpected results will occur if the quotient cannot be represented in 16 bits. When this occurs for the double operation (DIV.SD), the OV Status bit will be set and the quotient and remainder should not be used. For the word operation (DIV.S), only one type of overflow may occur (0x8000/0xFFFF = +32768 or 0x00008000), which allows the OV Status bit to interpret the result.

3: Dividing by zero will initiate an arithmetic error trap during the first cycle of execution.

4: This instruction is interruptible on each instruction cycle boundary.

Words: 1
Cycles: 18 (plus 1 for REPEAT execution)

Example 1: REPEAT #17 ; Execute DIV.S 18 times DIV.S W3, W4 ; Divide W3 by W4 ; Store quotient to W0, remainder to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 5555</td>
<td>W0 013B</td>
</tr>
<tr>
<td>W1 1234</td>
<td>W1 0003</td>
</tr>
<tr>
<td>W3 3000</td>
<td>W3 3000</td>
</tr>
<tr>
<td>W4 0027</td>
<td>W4 0027</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2: REPEAT #17 ; Execute DIV.SD 18 times DIV.SD W0, W12 ; Divide W1:W0 by W12 ; Store quotient to W0, remainder to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 2500</td>
<td>W0 FA6B</td>
</tr>
<tr>
<td>W1 FF42</td>
<td>W1 EF00</td>
</tr>
<tr>
<td>W12 2200</td>
<td>W12 2200</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008 (N = 1)</td>
</tr>
</tbody>
</table>
DIV.U

Unsigned Integer Divide

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

\{label:\} DIV.U{W} Wm, Wn

DIV.UD Wm, Wn

Operands:

- \(Wm \in [\ W0 \ldots W15]\) for word operation
- \(Wm \in [\ W0, W2, W4 \ldots W14]\) for double operation
- \(Wn \in [\ W2 \ldots W15]\)

Operation:

**For word operation (default):**

- \(Wm \rightarrow W0\)
- \(0x0 \rightarrow W1\)
- \(W1:W0/Wn \rightarrow W0\)
- Remainder \(\rightarrow W1\)

**For double operation (DIV.UD):**

- \(Wm + 1:Wm \rightarrow W1:W0\)
- \(W1:W0/Wns \rightarrow W0\)
- Remainder \(\rightarrow W1\)

Status Affected: \(N, OV, Z, C\)

Encoding:

| 1101 | 1000 | lttt | tvvv | vW00 | ssss |

Description:

Iterative, unsigned integer divide, where the dividend is stored in \(Wm\) (for a 16-bit by 16-bit divide), or \(Wm + 1:Wm\) (for a 32-bit by 16-bit divide) and the divisor is stored in \(Wn\). In the word operation, \(Wm\) is first copied to \(W0\) and \(W1\) is cleared to perform the divide. In the double operation, \(Wm + 1:Wm\) is first copied to \(W1:W0\). The 16-bit quotient of the divide operation is stored in \(W0\), and the 16-bit remainder is stored in \(W1\).

This instruction must be executed 18 times using the REPEAT instruction (with an iteration count of 17) to generate the correct quotient and remainder. The \(N\) flag will always be cleared. The \(OV\) flag will be set if the divide operation resulted in an overflow and cleared otherwise. The \(Z\) flag will be set if the remainder is ‘0’ and cleared otherwise. The \(C\) flag is used to implement the divide algorithm and its final value should not be used.

The ‘t’ bits select the most significant word of the dividend for the double operation. These bits are clear for the word operation.

The ‘v’ bits select the least significant word of the dividend.

The ‘W’ bit selects the dividend size (‘0’ for 16-bit, ‘1’ for 32-bit).

The ‘s’ bits select the divisor register.

**Note 1:** The extension .D in the instruction denotes a double word (32-bit) dividend rather than a word dividend. You may use a .W extension to denote a word operation, but it is not required.

**2:** Unexpected results will occur if the quotient cannot be represented in 16 bits. This may only occur for the double operation (DIV.UD). When an overflow occurs, the OV Status bit will be set and the quotient and remainder should not be used.

**3:** Dividing by zero will initiate an arithmetic error trap during the first cycle of execution.

**4:** This instruction is interruptible on each instruction cycle boundary.

Words: 1

Cycles: 18 (plus 1 for REPEAT execution)
### Example 1:

REPEAT #17 ; Execute DIV.U 18 times
DIV.U W2, W4 ; Divide W2 by W4
; Store quotient to W0, remainder to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0: 5555</td>
<td>W0: 0040</td>
</tr>
<tr>
<td>W1: 1234</td>
<td>W1: 0000</td>
</tr>
<tr>
<td>W2: 8000</td>
<td>W2: 8000</td>
</tr>
<tr>
<td>W4: 0200</td>
<td>W4: 0200</td>
</tr>
<tr>
<td>SR: 0000</td>
<td>SR: 0002 (Z = 1)</td>
</tr>
</tbody>
</table>

### Example 2:

REPEAT #17 ; Execute DIV.UD 18 times
DIV.UD W10, W12 ; Divide W11:W10 by W12
; Store quotient to W0, remainder to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0: 5555</td>
<td>W0: 01F2</td>
</tr>
<tr>
<td>W1: 1234</td>
<td>W1: 0100</td>
</tr>
<tr>
<td>W10: 2500</td>
<td>W10: 2500</td>
</tr>
<tr>
<td>W11: 0042</td>
<td>W11: 0042</td>
</tr>
<tr>
<td>W12: 2200</td>
<td>W12: 2200</td>
</tr>
<tr>
<td>SR: 0000</td>
<td>SR: 0000</td>
</tr>
</tbody>
</table>
DIVF Fractional Divide

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

Syntax: {label:} DIVF Wm, Wn

Operands: Wm ∈ [ W0 ... W15] Wn ∈ [ W2 ... W15]

Operation:
0x0 → W0
Wm → W1
W1:W0/Wn → W0
Remainder → W1

Status Affected: N, OV, Z, C

Encoding: 1101 1001 0ttt t000 0000 ssss

Description:
Iterative, signed fractional 16-bit by 16-bit divide, where the dividend is stored in Wm and the divisor is stored in Wn. To perform the operation, W0 is first cleared and Wm is copied to W1. The 16-bit quotient of the divide operation is stored in W0, and the 16-bit remainder is stored in W1. The sign of the remainder will be the same as the sign of the dividend.

This instruction must be executed 18 times using the REPEAT instruction (with an iteration count of 17) to generate the correct quotient and remainder. The N flag will be set if the remainder is negative and cleared otherwise. The OV flag will be set if the divide operation resulted in an overflow and cleared otherwise. The Z flag will be set if the remainder is ‘0’ and cleared otherwise. The C flag is used to implement the divide algorithm and its final value should not be used.

The 't' bits select the dividend register.
The 's' bits select the divisor register.

Note 1: For the fractional divide to be effective, Wm must be less than Wn. If Wm is greater than or equal to Wn, unexpected results will occur because the fractional result will be greater than or equal to 1.0. When this occurs, the OV Status bit will be set and the quotient and remainder should not be used.

2: Dividing by zero will initiate an arithmetic error trap during the first cycle of execution.

3: This instruction is interruptible on each instruction cycle boundary.

Words: 1
Cycles: 18 (plus 1 for REPEAT execution)
### Example 1:
```
REPEAT #17 ; Execute DIVF 18 times
DIVF W8, W9 ; Divide W8 by W9
; Store quotient to W0, remainder to W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 8000</td>
<td>W0 2000</td>
</tr>
<tr>
<td>W1 1234</td>
<td>W1 0000</td>
</tr>
<tr>
<td>W8 1000</td>
<td>W8 1000</td>
</tr>
<tr>
<td>W9 4000</td>
<td>W9 4000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0002</td>
</tr>
</tbody>
</table>

### Example 2:
```
REPEAT #17 ; Execute DIVF 18 times
DIVF W8, W9 ; Divide W8 by W9
; Store quotient to W0, remainder to W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 8000</td>
<td>W0 0000</td>
</tr>
<tr>
<td>W1 1234</td>
<td>W1 0000</td>
</tr>
<tr>
<td>W8 1000</td>
<td>W8 1000</td>
</tr>
<tr>
<td>W9 8000</td>
<td>W9 8000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0002</td>
</tr>
</tbody>
</table>

### Example 3:
```
REPEAT #17 ; Execute DIVF 18 times
DIVF W0, W1 ; Divide W0 by W1
; Store quotient to W0, remainder to W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 8002</td>
<td>W0 7FFE</td>
</tr>
<tr>
<td>W1 8001</td>
<td>W1 8002</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008</td>
</tr>
</tbody>
</table>
DO Initialize Hardware Loop Literal

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Syntax: \{label:\} DO #lit14, Expr

Operands: lit14 ∈ [0 ... 16383]
Expr may be an absolute address, label or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: PUSH DO shadows (DCOUNT, DOEND, DOSTART)
(lit14) → DCOUNT
(PC) + 4 → PC
(PC) → DOSTART
(PC) + (2 * Slit16) → DOEND
Increment DL<2:0> (CORCON<10:8>)

Status Affected: DA

Encoding:

<table>
<thead>
<tr>
<th>0000</th>
<th>1000</th>
<th>00kk</th>
<th>kkkk</th>
<th>kkkk</th>
<th>kkkk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>nnnn</td>
<td>nnnn</td>
<td>nnnn</td>
<td>nnnn</td>
</tr>
</tbody>
</table>
Description: Initiate a no overhead hardware \texttt{DO} loop, which is executed \((\text{lit14} + 1)\) times. The \texttt{DO} loop begins at the address following the \texttt{DO} instruction, and ends at the address \(2 \times \text{Slit16}\) instruction words away. The 14-bit count value (lit14) supports a maximum loop count value of 16384, and the 16-bit offset value (Slit16) supports offsets of 32K instruction words in both directions.

When this instruction executes, DCOUNT, DOSTART and DOEND are first PUSHed into their respective shadow registers, and then updated with the new \texttt{DO} loop parameters specified by the instruction. The \texttt{DO} level count, DL<2:0> (CORCON<8:10>), is then incremented. After the \texttt{DO} loop completes execution, the PUSHed DCOUNT, DOSTART and DOEND registers are restored, and DL<2:0> is decremented.

The \('k'\) bits specify the loop count.

The \('n'\) bits are a signed literal that specifies the number of instructions that are offset from the PC to the last instruction executed in the loop.

**Special Features, Restrictions:**

The following features and restrictions apply to the \texttt{DO} instruction.

1. Using a loop count of \(0\) will result in the loop being executed one time.
2. Using a loop size of -2, -1 or 0 is invalid. Unexpected results may occur if these offsets are used.
3. The very last two instructions of the \texttt{DO} loop cannot be:
   - an instruction which changes program control flow
   - a \texttt{DO} or \texttt{REPEAT} instruction

   Unexpected results may occur if any of these instructions are used.
4. If a hard trap occurs in the second to last instruction or third to last instruction of a \texttt{DO} loop, the loop will not function properly. The hard trap includes exceptions of priority level 13 through level 15, inclusive.

**Note 1:** The \texttt{DO} instruction is interruptible and supports 1 level of hardware nesting. Nesting up to an additional 5 levels may be provided in software by the user. See the specific device family reference manual for details.

**Note 2:** The linker will convert the specified expression into the offset to be used.

---

**Example 1:**

002000 LOOP6: \texttt{DO} \ #5, \texttt{END6}; Initiate DO loop (6 reps)

002004 \texttt{ADD} \ W1, W2, W3; First instruction in loop

002006 \texttt{...}

002008 \texttt{...}

00200A END6: \texttt{SUB} \ W2, W3, W4; Last instruction in loop

00200C \texttt{...}

---

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC \ 00 2000</td>
<td>PC \ 00 2004</td>
</tr>
<tr>
<td>DCOUNT \ 0000</td>
<td>DCOUNT \ 0005</td>
</tr>
<tr>
<td>DOSTART \ FF FFFF</td>
<td>DOSTART \ 00 004</td>
</tr>
<tr>
<td>DOEND \ FF FFFF</td>
<td>DOEND \ 00 200A</td>
</tr>
<tr>
<td>CORCON \ 0000</td>
<td>CORCON \ 0100</td>
</tr>
<tr>
<td>(\text{DL} = 1)</td>
<td>(\text{DA, C} = 1)</td>
</tr>
<tr>
<td>SR \ 0001 (C = 1)</td>
<td>SR \ 0201</td>
</tr>
</tbody>
</table>
Example 2:

01C000 LOOP12: DO #0x160, END12; Init DO loop (353 reps)
01C004   DEC W1, W2; First instruction in loop
01C006   
01C008   
01C00A   
01C00C   
01C00E   CALL _FIR88; Call the FIR88 subroutine
01C012   NOP
01C014 END12: NOP; Last instruction in loop

; (Required NOP filler)

Before Instruction                                      After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>01 C000</th>
<th>PC</th>
<th>01 C004</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCOUNT</td>
<td>0000</td>
<td>DCOUNT</td>
<td>0160</td>
</tr>
<tr>
<td>DOSTART</td>
<td>FF FFFF</td>
<td>DOSTART</td>
<td>01 C004</td>
</tr>
<tr>
<td>DOEND</td>
<td>FF FFFF</td>
<td>DOEND</td>
<td>01 C014</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
<td>CORCON</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td>(N = 1)</td>
<td></td>
<td>(DL = 1)</td>
</tr>
<tr>
<td></td>
<td>(DL = 1)</td>
<td></td>
<td>(DL = 1)</td>
</tr>
<tr>
<td>SR</td>
<td>0008</td>
<td>SR</td>
<td>0208</td>
</tr>
<tr>
<td></td>
<td>(N = 1)</td>
<td></td>
<td>(DA, N = 1)</td>
</tr>
</tbody>
</table>
DO Initialize Hardware Loop Literal


Syntax: {label:} DO #lit15, Expr

Operands: lit15 ∈ [0 ... 32767]
Expr may be an absolute address, label or expression.
Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... +32767].

Operation: PUSH DO shadow s (DCOUNT, DOEND, DOSTART)
(lit15) → DCOUNT
(PC) + 4 → PC
(PC) → DOSTART
(PC) + (2 * Slit16) → DOEND
Increment DL<2:0> (CORCON<10:8>)

Status Affected: DA

Encoding:

<table>
<thead>
<tr>
<th></th>
<th>0000</th>
<th>1000</th>
<th>0kkk</th>
<th>kkkk</th>
<th>kkkk</th>
<th>kkkk</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000</td>
<td>0000</td>
<td>nnnn</td>
<td>nnnn</td>
<td>nnnn</td>
<td>nnnn</td>
</tr>
</tbody>
</table>

Description: Initiate a no overhead hardware DO loop, which is executed (lit15 + 1) times.
The DO loop begins at the address following the DO instruction, and ends at the address 2 * Slit16 instruction words away. The 15-bit count value (lit15) supports a maximum loop count value of 32768, and the 16-bit offset value (Slit16) supports offsets of 32K instruction words in both directions.

When this instruction executes, DCOUNT, DOSTART and DOEND are first PUSHed into their respective shadow registers, and then updated with the new DO loop parameters specified by the instruction. The DO level count, DL<2:0> bits (CORCON<8:10>), is then incremented. After the DO loop completes execution, the PUSHed DCOUNT, DOSTART and DOEND registers are restored, and DL<2:0> is decremented.

The 'k' bits specify the loop count.
The 'n' bits are a signed literal that specifies the number of instructions that are offset from the PC to the last instruction executed in the loop.

Special Features, Restrictions:
The following features and restrictions apply to the DO instruction.
1. Using a loop count of '0' will result in the loop being executed one time.
2. Using a loop size of -2, -1 or 0 is invalid. Unexpected results may occur if these offsets are used.
3. The very last two instructions of the DO loop cannot be:
   - an instruction which changes program control flow
   - a DO or REPEAT instruction
   Unexpected results may occur if any of these instructions are used.
4. If a hard trap occurs in the second to last instruction or third to last instruction of a DO loop, the loop will not function properly. The hard trap includes exceptions of priority level 13 through level 15, inclusive.
5. The first instruction of the DO loop cannot be a PSV read or Table read.

Note 1: The DO instruction is interruptible and supports 1 level of hardware nesting. Nesting up to an additional 5 levels may be provided in software by the user. See the specific device family reference manual for details.
2: The linker will convert the specified expression into the offset to be used.
Example 1:
002000 LOOP6:   DO   #5, END6; Initiate DO loop (6 reps)
002004   ADD   W1, W2, W3; First instruction in loop
002006   .   .
002008   .   .
00200A END6:    SUB   W2, W3, W4; Last instruction in loop
00200C   .   .

Example 2:
01C000 LOOP12: DO   #0x160, END12; Init DO loop (353 reps)
01C004   DEC   W1, W2; First instruction in loop
01C006   .   .
01C008   .   .
01C00A   .   .
01C00C   .   .
01C00E   CALL   _FIR88; Call the FIR88 subroutine
01C012   NOP
01C014 END12: NOP; Last instruction in loop
; (Required NOP filler)
## DO

**Initialize Hardware Loop Wn**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label: \} DO Wn, Expr

**Operands:**

- Wn ∈ [W0 ... W15]
- Expr may be an absolute address, label or expression.

**Operation:**

- PUSH Shadows (DCOUNT, DOEND, DOSTART)
  - (Wn<13:0>) → DCOUNT
  - (PC) + 4 → PC
  - (PC) → DOSTART
  - (PC) + (2 * Slit16) → DOEND
  - Increment DL<2:0> (CORCON<10:8>)

**Status Affected:** DA

**Encoding:**

| 0000 | 0000 | nnnn | nnnn | nnnn | nnnn | ssss |

**Description:**

Initiate a no overhead hardware DO loop, which is executed (Wn + 1) times. The DO loop begins at the address following the DO instruction, and ends at the address 2 * Slit16 instruction words away. The lower 14 bits of Wn support a maximum count value of 16384, and the 16-bit offset value (Slit16) supports offsets of 32K instruction words in both directions.

When this instruction executes, DCOUNT, DOSTART and DOEND are first PUSHed into their respective shadow registers, and then updated with the new DO loop parameters specified by the instruction. The DO level count, DL<2:0> (CORCON<10:8>), is then incremented. After the DO loop completes execution, the PUSHed DCOUNT, DOSTART and DOEND registers are restored, and DL<2:0> is decremented.

The ‘s’ bits specify the register Wn that contains the loop count. The ‘n’ bits are a signed literal that specifies the number of instructions that are offset from (PC + 4), which is the last instruction executed in the loop.

**Special Features, Restrictions:**

The following features and restrictions apply to the DO instruction.

1. Using a loop count of ‘0’ will result in the loop being executed one time.
2. Using an offset of -2, -1 or 0 is invalid. Unexpected results may occur if these offsets are used.
3. The very last two instructions of the DO loop cannot be:
   - an instruction which changes program control flow
   - a DO or REPEAT instruction

Unexpected results may occur if these last instructions are used.

**Note 1:** The DO instruction is interruptible and supports 1 level of nesting. Nesting up to an additional 5 levels may be provided in software by the user. See the specific device family reference manual for details.

**Note 2:** The linker will convert the specified expression into the offset to be used.

**Words:** 2

**Cycles:** 2
Example 1:

```
002000 LOOP6:   DO     W0, END6   ; Initiate DO loop (W0 reps)
002004       ADD    W1, W2, W3 ; First instruction in loop
002006       . . .
002008       . . .
00200A       . . .
00200C       REPEAT #6
00200E       SUB    W2, W3, W4
002010 END6:   NOP               ; Last instruction in loop
                ; (Required NOP filler)
```

Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0012</td>
</tr>
<tr>
<td>DCOUNT</td>
<td>0000</td>
</tr>
<tr>
<td>DOSTART</td>
<td>FF FFFF</td>
</tr>
<tr>
<td>DOEND</td>
<td>FF FFFF</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0012</td>
</tr>
<tr>
<td>DCOUNT</td>
<td>0012</td>
</tr>
<tr>
<td>DOSTART</td>
<td>00 2004</td>
</tr>
<tr>
<td>DOEND</td>
<td>00 2010</td>
</tr>
<tr>
<td>CORCON</td>
<td>0100</td>
</tr>
<tr>
<td>SR</td>
<td>0080</td>
</tr>
</tbody>
</table>

Example 2:

```
002000 LOOPA:   DO     W7, ENDA   ; Initiate DO loop (W7 reps)
002004       SWAP   W0         ; First instruction in loop
002006       . . .
002008       . . .
00200A       . . .
002010 ENDA:    MOV    W1, [W2++] ; Last instruction in loop
```

Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>E00F</td>
</tr>
<tr>
<td>DCOUNT</td>
<td>0000</td>
</tr>
<tr>
<td>DOSTART</td>
<td>FF FFFF</td>
</tr>
<tr>
<td>DOEND</td>
<td>FF FFFF</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 2004</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>E00F</td>
</tr>
<tr>
<td>DCOUNT</td>
<td>200F</td>
</tr>
<tr>
<td>DOSTART</td>
<td>00 2004</td>
</tr>
<tr>
<td>DOEND</td>
<td>00 2010</td>
</tr>
<tr>
<td>CORCON</td>
<td>0100</td>
</tr>
<tr>
<td>SR</td>
<td>0080</td>
</tr>
</tbody>
</table>
**DO Initialize Hardware Loop Wn**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax:</td>
<td>{label:} DO Wn, Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operands:</td>
<td>Wn ∈ [W0 ... W15] Expr may be an absolute address, label or expression.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation:</td>
<td>PUSH Shadows (DCOUNT, DOEND, DOSTART) (Wn) → DCOUNT (PC) + 4 → PC (PC) → DOSTART (PC) + (2 * Slit16) → DOEND Increment DL&lt;2:0&gt; (CORCON&lt;10:8&gt;)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Affected:</td>
<td>DA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding:</td>
<td>0000 1000 1000 0000 0000 ssss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td>Initialize a no overhead hardware <strong>DO</strong> loop, which is executed (Wn + 1) times. The <strong>DO</strong> loop begins at the address following the <strong>DO</strong> instruction, and ends at the address 2 * Slit16 instruction words away. The 16 bits of Wn support a maximum count value of 65536, and the 16-bit offset value (Slit16) supports offsets of 32K instruction words in both directions. When this instruction executes, DCOUNT, DOSTART and DOEND are first PUSHed into their respective shadow registers, and then updated with the new <strong>DO</strong> loop parameters specified by the instruction. The <strong>DO</strong> level count, DL&lt;2:0&gt; (CORCON&lt;10:8&gt;), is then incremented. After the <strong>DO</strong> loop completes execution, the PUSHed DCOUNT, DOSTART and DOEND registers are restored, and DL&lt;2:0&gt; is decremented. The ‘s’ bits specify the register Wn that contains the loop count. The ‘n’ bits are a signed literal that specifies the number of instructions that are offset from (PC + 4), which is the last instruction executed in the loop.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special Features, Restrictions:</td>
<td>The following features and restrictions apply to the <strong>DO</strong> instruction.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Using a loop count of ‘0’ will result in the loop being executed one time.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Using an offset of -2, -1 or 0 is invalid. Unexpected results may occur if these offsets are used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. The very last two instructions of the <strong>DO</strong> loop cannot be:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a. an instruction which changes program control flow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. a <strong>DO</strong> or <strong>REPEAT</strong> instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unexpected results may occur if these last instructions are used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. The first instruction of the <strong>DO</strong> loop cannot be a PSV read or Table read.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Note 1:</td>
<td>The <strong>DO</strong> instruction is interruptible and supports 1 level of nesting. Nesting up to an additional 5 levels may be provided in software by the user. See the specific device family reference manual for details.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>The linker will convert the specified expression into the offset to be used.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Words:</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DO

Initialize Hardware Loop Wn

Cycles: 2

**Example 1:**

002000 LOOP6:   DO     W0, END6   ; Initiate DO loop (W0 reps)
002004        ADD    W1, W2, W3 ; First instruction in loop
002006        . . .
002008        . . .
00200A        . . .
00200C        REPEAT #6
00200E        SUB    W2, W3, W4
002010 END6:    NOP               ; Last instruction in loop
                 ; (Required NOP filler)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2004</td>
</tr>
<tr>
<td>W0 0012</td>
<td>W0 0012</td>
</tr>
<tr>
<td>DCOUNT 0000</td>
<td>DCOUNT 0012</td>
</tr>
<tr>
<td>DOSTART FF FFFF</td>
<td>DOSTART 00 2004</td>
</tr>
<tr>
<td>DOEND FF FFFF</td>
<td>DOEND 00 2010</td>
</tr>
<tr>
<td>CORCON 0000</td>
<td>CORCON 0100</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0080</td>
</tr>
</tbody>
</table>

Example 2:

002000 LOOPA:   DO     W7, ENDA   ; Initiate DO loop (W7 reps)
002004        SWAP   W0         ; First instruction in loop
002006        . . .
002008        . . .
00200A        . . .
002010 ENDA:    MOV    W1, [W2++] ; Last instruction in loop

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2000</td>
<td>PC 00 2004</td>
</tr>
<tr>
<td>W7 E00F</td>
<td>W7 E00F</td>
</tr>
<tr>
<td>DCOUNT 0000</td>
<td>DCOUNT 200F</td>
</tr>
<tr>
<td>DOSTART FF FFFF</td>
<td>DOSTART 00 2004</td>
</tr>
<tr>
<td>DOEND FF FFFF</td>
<td>DOEND 00 2010</td>
</tr>
<tr>
<td>CORCON 0000</td>
<td>CORCON 0100</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0080</td>
</tr>
</tbody>
</table>
Euclidean Distance (No Accumulate)

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax: {label:} ED Wm * Wm, Acc, [Wx], [Wy], Wxd

[Wx] + = kx, [Wy] + = ky,
[Wx] – = kx, [Wy] – = ky,
[W9 + W12]

Operands: Acc ∈ [A, B]
Wm * Wm ∈ [W4 * W4, W5 * W5, W6 * W6, W7 * W7]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]
Wxd ∈ [W4 ... W7]

Operation:
(Wm) * (Wm) → Acc(A or B)
((Wx) – [Wy]) → Wxd
(Wx) + kx → Wx
(Wy) + ky → Wy

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

| llll | 00mm | Alxx | 00ii | iijj | jjll |

Description: Compute the square of Wm, and compute the difference of the prefetch values specified by [Wx] and [Wy]. The results of Wm * Wm are sign-extended to 40 bits and stored in the specified accumulator. The results of [Wx] – [Wy] are stored in Wxd, which may be the same as Wm.

Operands Wx, Wxd and Wyd specify the prefetch operations which support indirect and register offset addressing as described in Section 4.14.1 “MAC Prefetches”.

The ‘m’ bits select the operand register Wm for the square.
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the prefetch difference Wxd destination.
The ‘i’ bits select the Wx prefetch operation.
The ‘j’ bits select the Wy prefetch operation.

Words: 1
Cycles: 1

Example 1:
ED W4*W4, A, [W8]++2, [W10]--2, W4 ; Square W4 to ACCA
; [W8]--[W10] to W4
; Post-increment W8
; Post-decrement W10

Before Instruction

| W4 | 009A |
| W8 | 1100 |
| W10| 2300 |
| ACCA | 00 3D0A 0000 |
| Data 1100 | 007F |
| Data 2300 | 0028 |
| SR | 0000 |

After Instruction

| W4 | 0057 |
| W8 | 1102 |
| W10| 22FE |
| ACCA | 00 0000 5CA4 |
| Data 1100 | 007F |
| Data 2300 | 0028 |
| SR | 0000 |
Example 2: \[ \text{ED W5+W5, B, [W9]*2, [W11+W12], W5} \]

; Square W5 to ACCB
; [W9]+[W11+W12] to W5
; Post-increment W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>43C2</td>
</tr>
<tr>
<td>W9</td>
<td>1200</td>
</tr>
<tr>
<td>W11</td>
<td>2500</td>
</tr>
<tr>
<td>W12</td>
<td>0008</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 28E3 F14C</td>
</tr>
<tr>
<td>Data 1200</td>
<td>6A7C</td>
</tr>
<tr>
<td>Data 2508</td>
<td>2B3D</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

EDAC  Euclidean Distance

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  \{label:} EDAC  Wm * Wm, Acc,  [Wx],  [Wy],  Wxd

\[Wx\] + =  \[Wy\] + = ky,  
\[Wx\] - = kx,  \[Wy\] = = ky,  
\[W9 + W12\],  \[W11 + W12\],

Operands:
- Acc ∈ [A,B]
- Wm * Wm ∈ [W4 * W4, W5 * W5, W6 * W6, W7 * W7]
- Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]
- Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]
- Wxd ∈ [W4 ... W7]

Operation:
- \((Acc(A \text{ or } B)) + (Wm) * (Wm)) \rightarrow Acc(A \text{ or } B)\)
- \(([Wx] - [Wy]) \rightarrow Wxd\)
- \((Wx) + kx \rightarrow Wx\)
- \((Wy) + ky \rightarrow Wy\)

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

```
  1111
  00mm
  A1xx
  00ii
  iiij
  jj10
```

Description:
Compute the square of Wm, and also the difference of the prefetch values specified by [Wx] and [Wy]. The results of Wm * Wm are sign-extended to 40 bits and added to the specified accumulator. The results of [Wx] - [Wy] are stored in Wxd, which may be the same as Wm.

Operands Wx, Wxd and Wyd specify the prefetch operations which support indirect and register offset addressing as described in Section 4.14.1 “MAC Prefetches”.

The ‘m’ bits select the operand register Wm for the square. The ‘A’ bit selects the accumulator for the result. The ‘x’ bits select the prefetch difference Wxd destination. The ‘i’ bits select the Wx prefetch operation. The ‘j’ bits select the Wy prefetch operation.

Words: 1
Cycles: 1

Example 1:
; Square W4 and add to ACCA
; [W8]+[W10] to W4
; Post-increment W8
; Post-decrement W10

Before Instruction

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>009A</td>
<td>W4</td>
<td>0057</td>
</tr>
<tr>
<td>W8</td>
<td>1100</td>
<td>W8</td>
<td>1102</td>
</tr>
<tr>
<td>W10</td>
<td>2300</td>
<td>W10</td>
<td>22FE</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 3D0A 3D0A</td>
<td>ACCA</td>
<td>00 3D0A 99AE</td>
</tr>
<tr>
<td>Data 1100</td>
<td>007F</td>
<td>Data 1100</td>
<td>007F</td>
</tr>
<tr>
<td>Data 2300</td>
<td>0028</td>
<td>Data 2300</td>
<td>0028</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

© 2005-2011 Microchip Technology Inc.
DS70157F-page 241
Example 2: \( \text{EDAC W5} \times \text{W5}, \text{B, [w9]+2, [W11+W12] to W5} \) ; Square W5 and
; add to ACCB
; [W9]−[W11+W12] to W5
; Post-increment W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5 43C2</td>
<td>W5 3F3F</td>
</tr>
<tr>
<td>W9 1200</td>
<td>W9 1202</td>
</tr>
<tr>
<td>W11 2500</td>
<td>W11 2500</td>
</tr>
<tr>
<td>W12 0008</td>
<td>W12 0008</td>
</tr>
<tr>
<td>ACCB 0028E3F14C</td>
<td>ACCB 003AD31050</td>
</tr>
<tr>
<td>Data 1200 6A7C</td>
<td>Data 1200 6A7C</td>
</tr>
<tr>
<td>Data 2508 2B3D</td>
<td>Data 2508 2B3D</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
EXCH

Exchange Wns and Wnd

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label;} EXCH Wns, Wnd

Operands:

- Wns ∈ [W0 ... W15]
- Wnd ∈ [W0 ... W15]

Operation:

(Wns) ↔ (Wnd)

Status Affected: None

Encoding:

| 1111 | 1101 | 0000 | 0ddd | d000 | ssss |

Description:

Exchange the word contents of two working registers. Register direct addressing must be used for Wns and Wnd.

The ‘d’ bits select the address of the first register.

The ‘s’ bits select the address of the second register.

Note: This instruction only executes in Word mode.

Words: 1
Cycles: 1

Example 1:

EXCH W1, W9 ; Exchange the contents of W1 and W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>55FF</td>
</tr>
<tr>
<td>W9</td>
<td>A3A3</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

EXCH W4, W5 ; Exchange the contents of W4 and W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>ABCD</td>
</tr>
<tr>
<td>W5</td>
<td>4321</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
FBCL  Find First Bit Change from Left

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: \{label: \} FBCL \( Ws, \) \( Wnd \)

<table>
<thead>
<tr>
<th>Operands:</th>
<th>( Ws \in [W0 \ldots W15] )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( Wnd \in [W0 \ldots W15] )</td>
</tr>
</tbody>
</table>

Operation:
- \( \text{Max}_{-}\text{Shift} = 15 \)
- \( \text{Sign} = (Ws) \& 0x8000 \)
- \( \text{Temp} = (Ws) \ll 1 \)
- \( \text{Shift} = 0 \)
- While \((\text{Shift} < \text{Max}_{-}\text{Shift}) \&\&( (\text{Temp} & 0x8000) == \text{Sign}) \) \)
  - \( \text{Temp} = \text{Temp} \ll 1 \)
  - \( \text{Shift} = \text{Shift} + 1 \)
  - \( \text{-Shift} \rightarrow (Wnd) \)

Status Affected: \( C \)

Encoding:

| 1101 | 1111 | 0000 | dddd | dppp | ssss |

Description:
Find the first occurrence of a one (for a positive value), or zero (for a negative value), starting from the Most Significant bit after the sign bit of \( Ws \) and working towards the Least Significant bit of the word operand. The bit number result is sign-extended to 16 bits and placed in \( Wnd \).

The next Most Significant bit after the sign bit is allocated bit number 0 and the Least Significant bit is allocated bit number -14. This bit ordering allows for the immediate use of Wd with the \text{SFTAC} instruction for scaling values up. If a bit change is not found, a result of -15 is returned and the C flag is set. When a bit change is found, the C flag is cleared.

The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: \( 1^{(1)} \)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
### Example 1:

FBCL W1, W9  
; Find 1st bit change from left in W1  
; and store result to W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 55FF</td>
<td>W1 55FF</td>
</tr>
<tr>
<td>W9 FFFF</td>
<td>W9 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

### Example 2:

FBCL W1, W9  
; Find 1st bit change from left in W1  
; and store result to W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 FFFF</td>
<td>W1 FFFF</td>
</tr>
<tr>
<td>W9 BBBB</td>
<td>W9 FFF1</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

### Example 3:

FBCL [W1++], W9  
; Find 1st bit change from left in [W1]  
; and store result to W9  
; Post-increment W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 2000</td>
<td>W1 2002</td>
</tr>
<tr>
<td>W9 BBBB</td>
<td>W9 FFF9</td>
</tr>
<tr>
<td>Data 2000 FF0A</td>
<td>Data 2000 FF0A</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
### FF1L

Find First One from Left

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:
```
{label:} FF1L Ws, Wnd
```

#### Operands:
- `Ws` ∈ [W0 ... W15]
- `Wnd` ∈ [W0 ... W15]

#### Operation:
```
Max_Shift = 17
Temp = (Ws)
Shift = 1
While ((Shift < Max_Shift) && !(Temp & 0x8000))
    Temp = Temp << 1
    Shift = Shift + 1
If (Shift == Max_Shift)
    0 → (Wnd)
Else
    Shift → (Wnd)
```

#### Status Affected:
- C

#### Encoding:
```
1100 1111 1000 0ddd dppp ss
```

#### Description:
Finds the first occurrence of a `'1'` starting from the Most Significant bit of `Ws` and working towards the Least Significant bit of the word operand. The bit number result is zero-extended to 16 bits and placed in `Wnd`.

Bit numbering begins with the Most Significant bit (allocated number 1) and advances to the Least Significant bit (allocated number 16). A result of zero indicates a `'1'` was not found, and the C flag will be set. If a `'1'` is found, the C flag is cleared.

The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

**Note:** This instruction operates in Word mode only.

### Words:
1

### Cycles:
1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.
Example 1:  FF1L W2, W5 ; Find the 1st one from the left in W2
; and store result to W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2  000A</td>
<td>W2  000A</td>
</tr>
<tr>
<td>W5  BBBB</td>
<td>W5  000D</td>
</tr>
<tr>
<td>SR  0000</td>
<td>SR  0000</td>
</tr>
</tbody>
</table>

Example 2:  FF1L [W2++], W5 ; Find the 1st one from the left in [W2]
; and store the result to W5
; Post-increment W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2    2000</td>
<td>W2    2002</td>
</tr>
<tr>
<td>W5    BBBB</td>
<td>W5    0000</td>
</tr>
<tr>
<td>Data 2000   0000</td>
<td>Data 2000   0000</td>
</tr>
<tr>
<td>SR  0000</td>
<td>SR  0001 (C = 1)</td>
</tr>
</tbody>
</table>
**FF1R**

Find First One from Right

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{\textcolor{red}{\text{label}}:} FF1R \ Ws, \ Wnd

\[ \text{Ws}, \quad \text{Ws++}, \quad \text{Ws--}, \quad ++\text{Ws}, \quad -\text{Ws}, \]

**Operands:**

Ws ∈ [W0 ... W15]

Wnd ∈ [W0 ... W15]

**Operation:**

Max\_Shift = 17

Temp = (Ws)

Shift = 1

While ( (Shift < Max\_Shift) && !(Temp & 0x1) )

\[ \text{Temp} = \text{Temp} >> 1 \]

\[ \text{Shift} = \text{Shift} + 1 \]

If (Shift == Max\_Shift)

\[ 0 \rightarrow (\text{Wnd}) \]

Else

\[ \text{Shift} \rightarrow (\text{Wnd}) \]

**Status Affected:**

C

**Encoding:**

| 1100 | 1111 | 0000 | 0ddd | dppp | sszz |

**Description:**

Finds the first occurrence of a ‘1’ starting from the Least Significant bit of Ws and working towards the Most Significant bit of the word operand. The bit number result is zero-extended to 16 bits and placed in Wnd.

Bit numbering begins with the Least Significant bit (allocated number 1) and advances to the Most Significant bit (allocated number 16). A result of zero indicates a ‘1’ was not found, and the C flag will be set. If a ‘1’ is found, the C flag is cleared.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** This instruction operates in Word mode only.

**Words:**

1

**Cycles:**

1\(^{(1)}\)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

---

**FF1R Find First One from Right**

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {\textcolor{red}{\text{label}}:} FF1R \ Ws, \ Wnd

[Ws],

[Ws++],

[Ws--],

[++Ws],

[-Ws],

Operands: Ws ∈ [W0 ... W15]

Wnd ∈ [W0 ... W15]

Operation: Max\_Shift = 17

Temp = (Ws)

Shift = 1

While ( (Shift < Max\_Shift) && !(Temp & 0x1) )

Temp = Temp >> 1

Shift = Shift + 1

If (Shift == Max\_Shift)

0 → (Wnd)

Else

Shift → (Wnd)

Status Affected: C

Encoding: 1100 1111 0000 0ddd dppp sszz

Description: Finds the first occurrence of a ‘1’ starting from the Least Significant bit of Ws and working towards the Most Significant bit of the word operand. The bit number result is zero-extended to 16 bits and placed in Wnd.

Bit numbering begins with the Least Significant bit (allocated number 1) and advances to the Most Significant bit (allocated number 16). A result of zero indicates a ‘1’ was not found, and the C flag will be set. If a ‘1’ is found, the C flag is cleared.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

Note: This instruction operates in Word mode only.

Words: 1

Cycles: 1\(^{(1)}\)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

---

1100 1111 0000 0ddd dppp sszz
Example 1:  
`FF1R W1, W9` ; Find the 1st one from the right in W1  
; and store the result to W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>000A</td>
</tr>
<tr>
<td>W9</td>
<td>BBBB</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:  
`FF1R [W1++], W9` ; Find the 1st one from the right in [W1]  
; and store the result to W9  
; Post-increment W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>2000</td>
</tr>
<tr>
<td>W9</td>
<td>BBBB</td>
</tr>
<tr>
<td>Data 2000</td>
<td>8000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

| W1  | 2002 |
| W9  | 0010 |
| Data 2000 | 8000 |
| SR  | 0000 |
GOTO

Unconditional Jump

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: \{label:} \ GOTO \ Expr

Operands: Expr may be label or expression (but not a literal).
Expr is resolved by the linker to a lit23, where lit23 \in [0 \ldots 8388606].

Operation: lit23 \rightarrow PC

Status Affected: None

Encoding:

1st word

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0100</td>
<td>nnnn</td>
<td>nnnn</td>
<td>nnnn</td>
<td>nnn0</td>
</tr>
</tbody>
</table>

2nd word

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0nnn</td>
<td>nnnn</td>
</tr>
</tbody>
</table>

Description: Unconditional jump to anywhere within the 4M instruction word program memory range. The PC is loaded with the 23-bit literal specified in the instruction. Since the PC must always reside on an even address boundary, lit23<0> is ignored.

The 'n' bits form the target address.

Note: The linker will resolve the specified expression into the lit23 to be used.

Words: 2

Cycles: 2 (PIC24F, PIC24H, dsPIC30F, dsPIC33F)
4 (PIC24E, dsPIC33E)

Example 1:

026000 \ GOTO \ _THERE \ ; \ Jump \ to \ _THERE
026004 \ MOV \ W0, W1
\ ...
\ ...
027844 \_THERE: \ MOV \ #0x400, W2 \ ; \ Code \ execution
027846 \ ...
\ ; \ resumes \ here

Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>02 6000</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>02 7844</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

000100 \_code: \ ...
\ ...
026000 \ GOTO \ _code+2 \ ; \ Jump \ to \ _code+2
026004 \ ...

Before Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>02 6000</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0102</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

GOTO

Unconditional Indirect Jump

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:
{label;} GOTO Wn

Operands:
Wn ∈ [W0 ... W15]

Operation:
0 → PC<22:16>
(Wn<15:1>) → PC<15:1>
0 → PC<0>
NOP → Instruction Register

Status Affected:
None

Encoding:
0000 0001 0100 0000 0000 ssss

Description:
Unconditional indirect jump within the first 32K words of program memory. Zero is loaded into PC<22:16> and the value specified in (Wn) is loaded into PC<15:1>. Since the PC must always reside on an even address boundary, Wn<0> is ignored.

The ‘s’ bits select the source register.

Words: 1
Cycles: 2

Example 1:
006000  GOTO W4                 ; Jump unconditionally
006002  MOV  W0, W1             ; to 16-bit value in W4
          .                    ...
          .                    ...
007844 _THERE: MOV  #0x400, W2 ; Code execution
007846   ...                   ; resumes here

Before
Instruction

<table>
<thead>
<tr>
<th>W4</th>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7844</td>
<td>00 6000</td>
<td>0000</td>
</tr>
</tbody>
</table>

After
Instruction

<table>
<thead>
<tr>
<th>W4</th>
<th>PC</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7844</td>
<td>00 7844</td>
<td>0000</td>
</tr>
</tbody>
</table>
### GOTO

**Unconditional Indirect Jump**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Syntax:

{\text{label:}} \ GOTO \ Wn

#### Operands:

Wn ∈ \{W0 ... W15\}

#### Operation:

0 → PC<22:16>

(Wn<15:1>) → PC<15:1>

0 → PC<0>

NOP → Instruction Register

#### Status Affected:

None

#### Encoding:

```
0000 0001 0000 0100 0000 ssss
```

#### Description:

Unconditional indirect jump within the first 32K words of program memory. Zero is loaded into PC<22:16> and the value specified in (Wn) is loaded into PC<15:1>. Since the PC must always reside on an even address boundary, Wn<0> is ignored.

The ‘s’ bits select the source register.

#### Words:

1

#### Cycles:

4

#### Example 1:

006000 \ GOTO \ W4  \;

Jump unconditionally

006002 \ MOV \ W0, W1  \;

to 16-bit value in W4

\ldots
\ldots

007844 \ THERE: \ MOV \ #0x400, W2  \;

Code execution

007846 \ \ldots

resumes here

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>7844</td>
</tr>
<tr>
<td>PC</td>
<td>00 6000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W4</td>
<td>7844</td>
</tr>
<tr>
<td>PC</td>
<td>00 7844</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
## GOTO.L
Unconditional Indirect Jump Long

### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

### Syntax:

\( \{label;\} \quad \text{GOTO.L} \quad Wn \)

### Operands:

\( Wn \in [W0, W2, W4, W6, W8, W10, W12] \)

### Operation:

\( PC<23> \to PC<23> \text{ (see text)}; \ (Wn+1)<6:0> \to PC<22:16>; \ (Wn) \to PC<15:0> \text{r} \)

### Status Affected:
None

### Encoding:

| 0000 | 0001 | lwww | w100 | 0000 | ssss |

### Description:

Unconditional indirect jump to any user program memory address.

The LS 7-bits of \((Wn+1)\) are loaded in \(PC<22:16>\), and the 16-bit value \((Wn)\) is loaded into \(PC<15:0>\).

\(PC<23>\) is not modified by this instruction.

The contents of \((Wn+1)<15:7>\) are ignored.

The value of \(Wn<0>\) is also ignored and \(PC<0>\) is always set to 0.

GOTO is a two-cycle instruction.

The ‘s’ bits select the address of the \(Wn\) source register.

The ‘w’ bits specify the address of the \(Wn+1\) source register.

### Words:

1

### Cycles:

4

### Example 1:

<table>
<thead>
<tr>
<th>PC</th>
<th>W4</th>
<th>W0</th>
<th>W1</th>
<th>PC</th>
<th>W4</th>
<th>W5</th>
<th>W2</th>
</tr>
</thead>
<tbody>
<tr>
<td>026000</td>
<td></td>
<td></td>
<td></td>
<td>026844</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>026004</td>
<td>MOV</td>
<td>W0</td>
<td>W1</td>
<td>0002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>026844</td>
<td>_FIR: MOV</td>
<td>#0x400, W2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>_FIR subroutine start</td>
<td></td>
<td>_FIR subroutine start</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Before Instruction:

- PC: 02 6000
- W4: 6844
- W5: 0002
- W15: A268
- Data A268: FFFF
- Data A26A: FFFF
- SR: 0000

### After Instruction:

- PC: 02 6844
- W4: 6844
- W5: 0002
- W15: A26C
- Data A268: 6004
- Data A26A: 0002
- SR: 0000
**INC**  Increment f

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label:\} INC(B) f \{,WREG\}

**Operands:**

\( f \in [0 \ldots 8191] \)

**Operation:**

\((f) + 1 \rightarrow \text{destination designated by D}\)

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>1100</td>
<td>0BDf ffff</td>
<td>ffff</td>
<td>ffff</td>
<td>ffff</td>
</tr>
</tbody>
</table>

**Description:**

Add one to the contents of the file register, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

**Words:** 1

**Cycles:** (1)

**Example 1:**

INC.B 0x1000 ; Increment 0x1000 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>8FFF</td>
<td>8F00</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>
| 0000               | 0101              | (DC, C = 1)

**Example 2:**

INC 0x1000, WREG ; Increment 0x1000 and store to WREG ; (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG</td>
<td>WREG</td>
</tr>
<tr>
<td>ABCD</td>
<td>9000</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>8FFF</td>
<td>8F00</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>
| 0000               | 0108              | (DC, N = 1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**
**INC**

**Increment Ws**

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ws</td>
<td>X</td>
<td>X</td>
<td>x</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Wd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

{label:} INC{.B} Ws, Wd

[Ws], [Wd]

[Ws++], [Wd++]

[Ws--], [Wd--]

[++Ws], [++Wd]

[--Ws], [--Wd]

Operands:

Ws ∈ [W0 ... W15]

Wd ∈ [W0 ... W15]

Operation:

(Ws) + 1 → Wd

Status Affected:

DC, N, OV, Z, C

Encoding:

```
 1110  1000  0Bqq qddd dppp ssss
```

Description:

Add 1 to the contents of the source register Ws and place the result in the destination register Wd. Register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

INC.B W1, [++W2] ; Pre-increment W2

; Increment W1 and store to W2

; (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 FF7F</td>
<td>W1 FF7F</td>
</tr>
<tr>
<td>W2 2000</td>
<td>W2 2001</td>
</tr>
<tr>
<td>Data 2000 ABCD</td>
<td>Data 2000 80CD</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 010C (DC, N, OV = 1)</td>
</tr>
</tbody>
</table>
Example 2:

```
INC W1, W2
; Increment W1 and store to W2
; (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 FF7F</td>
<td>W1 FF7F</td>
</tr>
<tr>
<td>W2 2000</td>
<td>W2 FF80</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0108           (DC, N = 1)</td>
</tr>
</tbody>
</table>
**INC2**

Increment f by 2

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

```
{label:} INC2{.B} f {,WREG}
```

Operands:  

`f ∈ [0 ... 8191]`

Operation:  

`(f) + 2 → destination designated by D`

Status Affected:  

DC, N, OV, Z, C

Encoding:

```
| 1110 | 1100 | 1BDF | ffff | ffff | ffff |
```

Description:  

Add 2 to the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register). The ‘f’ bits select the address of the file register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: \(1\) (1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

```
INC2.B 0x1000 ; Increment 0x1000 by 2
               ; (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>8FFF</td>
<td>8F01</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0101</td>
</tr>
</tbody>
</table>

*(DC, C = 1)*

**Example 2:**

```
INC2 0x1000, WREG ; Increment 0x1000 by 2 and store to WREG
               ; (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG ABCD</td>
<td>WREG 9001</td>
</tr>
<tr>
<td>Data 1000</td>
<td>Data 1000</td>
</tr>
<tr>
<td>8FFF</td>
<td>8FFF</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0108</td>
</tr>
</tbody>
</table>

*(DC, N = 1)*
INC2
Increment Ws by 2

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

{label:} INC2{.B} Ws, Wd

[Ws], [Wd]

[Ws++], [Wd++]

[Ws--], [Wd--]

[++Ws], [++Wd]

[--Ws], [--Wd]

Operands:

Ws ∈ [W0 ... W15]

Wd ∈ [W0 ... W15]

Operation:

(Ws) + 2 → Wd

Status Affected:

DC, N, OV, Z, C

Encoding:

| 1110 | 1000 | lBqq | qddd | dppp | ssss |

Description:

Add 2 to the contents of the source register Ws and place the result in the destination register Wd. Register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'q' bits select the destination Address mode.

The 'd' bits select the destination register.

The 'p' bits select the source Address mode.

The 's' bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

INC2.B W1, [++W2] ; Pre-increment W2

; Increment by 2 and store to W1

; (Byte mode)

Before Instruction

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 FF7F</td>
</tr>
<tr>
<td>W2 2000</td>
</tr>
<tr>
<td>Data 2000 ABCD</td>
</tr>
<tr>
<td>SR 0000</td>
</tr>
</tbody>
</table>

After Instruction

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 FF7F</td>
</tr>
<tr>
<td>W2 2001</td>
</tr>
<tr>
<td>Data 2000 81CD</td>
</tr>
<tr>
<td>SR 010C (DC, N, OV = 1)</td>
</tr>
</tbody>
</table>
**Example 2:** INC2 W1, W2 ; Increment W1 by 2 and store to W2 ; (word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1 FF7F</td>
<td>W1 FF7F</td>
</tr>
<tr>
<td>W2 2000</td>
<td>W2 FF81</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0108 (DC, N = 1)</td>
</tr>
</tbody>
</table>
IOR

Inclusive OR f and WREG

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

X X X X X X

Syntax: {label;} IOR{.B} f {,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (f).IOR.(WREG) → destination designated by D

Status Affected: N, Z

Encoding:

| 1011 | 0111 | 0Bdf | ffff | ffff | ffff |

Description: Compute the logical inclusive OR operation of the contents of the working register WREG and the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Example 1:

Example 2:

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

Before Instruction

<table>
<thead>
<tr>
<th>WREG</th>
<th>Data 1000</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>FF00</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>WREG</th>
<th>Data 1000</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>FF34</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

Before Instruction

<table>
<thead>
<tr>
<th>WREG</th>
<th>Data 1000</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>0FAB</td>
<td>0008</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>WREG</th>
<th>Data 1000</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1FBF</td>
<td>0FAB</td>
<td>0000</td>
</tr>
</tbody>
</table>

Note 2: The WREG is set to working register W0.
Section 5. Instruction Descriptions

IOR

Inclusive OR Literal and Wn

Implemented in: PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E
---|---|---|---|---|---
X | X | X | X | X | X

Syntax: `{label;} IOR{.B} #lit10, Wn`

Operands:
- lit10 ∈ [0 ... 255] for byte operation
- lit10 ∈ [0 ... 1023] for word operation
- Wn ∈ [W0 ... W15]

Operation: lit10.IOR.(Wn) → Wn

Status Affected: N, Z

Encoding:

| 1011 | 0011 | 0Bkk | kkkk | kkkk | dddd |

Description: Compute the logical inclusive OR operation of the 10-bit literal operand and the contents of the working register Wn and place the result back into the working register Wn.

The ‘B’ bit selects byte or word operation (’0’ for word, ‘1’ for byte).
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

Words: 1
Cycles: 1

Example 1: IOR.B #0xAA, W9 ; IOR 0xAA to W9 ; (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W9</td>
<td>1234</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2: IOR #0x2AA, W4 ; IOR 0x2AA to W4 ; (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>A34D</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
IOR

Inclusive OR Wb and Short Literal

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} IOR.(B) Wb, #lit5, Wd

- [Wd]
- [Wd++]
- [Wd--]
- [++Wd]
- [--Wd]

Operands:
- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wd ∈ [W0 ... W15]

Operand.set: (Wb).IOR.lit5 → Wd

Status Affected: N, Z

Encoding:

| 0111 | 0www | wBqq | qddd | d11k | kkkk |

Description: Compute the logical inclusive OR operation of the contents of the base register Wb and the 5-bit literal operand and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1

Example 1: IOR.B W1, #0x5, [W9++] ; IOR W1 and 0x5 (Byte mode)
; Store to [W9]
; Post-increment W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>AAAA</td>
</tr>
<tr>
<td>W9</td>
<td>2000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>AAAA</td>
</tr>
<tr>
<td></td>
<td>2001</td>
</tr>
<tr>
<td></td>
<td>00AF</td>
</tr>
</tbody>
</table>
|                   | 0008              | (N = 1)

Example 2: IOR W1, #0x0, W9 ; IOR W1 with 0x0 (Word mode)
; Store to W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>0000</td>
</tr>
<tr>
<td>W9</td>
<td>A34D</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>
|                   | 0002              | (Z = 1)
Section 5. Instruction Descriptions

IOR                      Inclusive OR Wb and Ws

Implemented in:          
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} IOR{.B} Wb, Ws, Wd
         [Ws], [Wd]
         [Ws++], [Wd++]
         [Ws--], [Wd--]
         [++Ws], [++Wd]
         [--Ws], [--Wd]

Operands: Wb ∈ [W0 ... W15]
          Ws ∈ [W0 ... W15]
          Wd ∈ [W0 ... W15]

Operation: (Wb).IOR.(Ws) →Wd

Status Affected: N, Z

Encoding: 0111 0www wBqq qddd dppp ssss

Description: Compute the logical inclusive OR operation of the contents of the source register Ws and the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: IOR.B W1, [W5++], [W9++] ; IOR W1 and [W5] (Byte mode)
            ; Store result to [W9]
            ; Post-increment W5 and W9

Before Instruction          After Instruction
| W1 | AAAA | W1 | AAAA |
| W5 | 2000 | W5 | 2001 |
| W9 | 2400 | W9 | 2401 |
| Data 2000 | 1155 | Data 2000 | 1155 |
| Data 2400 | 0000 | Data 2400 | 00FF |
| SR | 0000 | SR | 0008 (N = 1) |
**Example 2:**

IOR W1, W5, W9 ; IOR W1 and W5 (Word mode)

; Store the result to W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>AAAA</td>
<td>AAAA</td>
</tr>
<tr>
<td>W5</td>
<td>W5</td>
</tr>
<tr>
<td>5555</td>
<td>5555</td>
</tr>
<tr>
<td>W9</td>
<td>W9</td>
</tr>
<tr>
<td>A34D</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
**LAC**

**Load Accumulator**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} LAC Ws, {#Slit4,} Acc

[Ws],
[Ws++],
[Ws--],
[--Ws],
[++Ws],
[Ws+Wb],

Operands: 
Ws ∈ [W0 ... W15]
Wb ∈ [W0 ... W15]
Slit4 ∈ [-8 ... +7]
Acc ∈ [A,B]

Operation: ShiftSlit4(Extend(Ws)) →Acc(A or B)

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

| 1100 | 1010 | Awww | wrrr | rggg | ssss |

Description: Read the contents of the source register, optionally perform a signed 4-bit shift and store the result in the specified accumulator. The shift range is -8:7, where a negative operand indicates an arithmetic left shift and a positive operand indicates an arithmetic right shift. The data stored in the source register is assumed to be 1.15 fractional data and is automatically sign-extended (through bit 39) and zero-backfilled (bits [15:0]), prior to shifting.

The 'A' bit specifies the destination accumulator. The 'w' bits specify the offset register Wb. The 'r' bits encode the accumulator pre-shift. The 'g' bits select the source Address mode. The 's' bits specify the source register Ws.

Note: If the operation moves more than sign-extension data into the upper Accumulator register (AccxU), or causes a saturation, the appropriate overflow and saturation bits will be set.

Words: 1
Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
### Example 1:

LAC [W4++], #-3, B

; Load ACCB with [W4] << 3
; Contents of [W4] do not change
; Post increment W4
; Assume saturation disabled
; (SATB = 0)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 2000</td>
<td>W4 2002</td>
</tr>
<tr>
<td>ACCB 00 5125 ABCD</td>
<td>ACCB FF 9108 0000</td>
</tr>
<tr>
<td>Data 2000 1221</td>
<td>Data 2000 1221</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 4800 (OB, OAB = 1)</td>
</tr>
</tbody>
</table>

### Example 2:

LAC [--W2], #7, A

; Pre-decrement W2
; Load ACCA with [W2] >> 7
; Contents of [W2] do not change
; Assume saturation disabled
; (SATA = 0)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 4002</td>
<td>W2 4000</td>
</tr>
<tr>
<td>ACCA 00 5125 ABCD</td>
<td>ACCA FF FF22 1000</td>
</tr>
<tr>
<td>Data 4000 9108</td>
<td>Data 4000 9108</td>
</tr>
<tr>
<td>Data 4002 1221</td>
<td>Data 4002 1221</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**LNK**

Allocate Stack Frame

Implemented in: PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E
---|---|---|---|---|---
X | X | X | X | X | X

Syntax: {label:} LNK #lit14

Operands: lit14 ∈ [0 ... 16382]

Operation:
- (W14) → (TOS)
- (W15) + 2 → W15
- (W15) → W14
- (W15) + lit14 → W15

Status Affected: None

Encoding: 1111 1010 00kk kkkk kkkk kkk0

Description: This instruction allocates a Stack Frame of size lit14 bytes for a subroutine calling sequence. The Stack Frame is allocated by PUSHing the contents of the Frame Pointer (W14) onto the stack, storing the updated Stack Pointer (W15) to the Frame Pointer and then incrementing the Stack Pointer by the unsigned 14-bit literal operand. This instruction supports a maximum Stack Frame of 16382 bytes.

The 'k' bits specify the size of the Stack Frame.

**Note:** Since the Stack Pointer can only reside on a word boundary, lit14 must be even.

Words: 1
Cycles: 1

**Example 1:**

```
LNK #0xA0 ; Allocate a stack frame of 160 bytes
```

**Before Instruction**

<table>
<thead>
<tr>
<th>W14</th>
<th>W15</th>
<th>Data 2000</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>2000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>W14</th>
<th>W15</th>
<th>Data 2000</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>20A2</td>
<td>2000</td>
<td>0000</td>
</tr>
</tbody>
</table>
**LNK**

Allocate Stack Frame

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} LNK #lit14`

Operands: `lit14 ∈ [0 ... 16382]`

Operation: 
- `(W14) → (TOS)`
- `(W15) + 2 → W15`
- `(W15) → W14`
- `1 → SFA bit`
- `(W15) + lit14 → W15`

Status Affected: SFA

Encoding: `1111 1010 00kk kkkk kkk0`

Description: This instruction allocates a Stack Frame of size `lit14` bytes for a subroutine calling sequence. The Stack Frame is allocated by PUSHing the contents of the Frame Pointer `(W14)` onto the stack, storing the updated Stack Pointer `(W15)` to the Frame Pointer and then incrementing the Stack Pointer by the unsigned 14-bit literal operand. This instruction supports a maximum Stack Frame of 16382 bytes.

The 'k' bits specify the size of the Stack Frame.

**Note:** Since the Stack Pointer can only reside on a word boundary, `lit14` must be even.

Words: 1
Cycles: 1

**Example 1:**

```
LNK   #0xA0   ; Allocate a stack frame of 160 bytes
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W14</td>
<td>W14</td>
</tr>
<tr>
<td>2000</td>
<td>2002</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>2000</td>
<td>20A2</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>0000</td>
<td>2000</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>0000</td>
<td>0004</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

LSR
Logical Shift Right \( f \)

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: \{label:} LSR{.B} \( f \) \{,WREG\}

Operands: \( f \in [0 ... 8191] \)

Operation:
For byte operation:

\[
0 \rightarrow \text{Dest}<7>
\]
\[
(f<7:1>) \rightarrow \text{Dest}<6:0>
\]
\[
(f<0>) \rightarrow \text{C}
\]
For word operation:

\[
0 \rightarrow \text{Dest}<15>
\]
\[
(f<15:1>) \rightarrow \text{Dest}<14:0>
\]
\[
(f<0>) \rightarrow \text{C}
\]

Status Affected: \( N, Z, C \)

Encoding:

|       | 1101 | 0101 | 0BDf | ffff | ffff | ffff |

Description:
Shift the contents of the file register one bit to the right and place the result in the destination register. The Least Significant bit of the file register is shifted into the Carry bit of the STATUS register. Zero is shifted into the Most Significant bit of the destination register.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).
The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

Words: 1
Cycles: 1\(^{(1)}\)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

```
LSR.B 0x600   ; Logically shift right (0x600) by one
             ; (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 600</td>
<td>Data 600</td>
</tr>
<tr>
<td>55FF</td>
<td>557F</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001</td>
</tr>
</tbody>
</table>

\( C = 1 \)
### Example 2:

```
LSR 0x600, WREG ; Logically shift right (0x600) by one
; Store to WREG
; (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 600</td>
<td>Data 600</td>
</tr>
<tr>
<td>WREG 0000</td>
<td>WREG 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0x600</th>
<th>55FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>2AFF</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
</tbody>
</table>
### LSR

**Logical Shift Right Ws**

#### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

```
{label:} LSR{.B} Ws, Wd

[Ws], [Wd]
[Ws++], [Wd++]
[Ws--], [Wd--]
[++Ws], [++Wd]
[--Ws], [--Wd]
```

#### Operands:

- Ws ∈ [W0 ... W15]
- Wd ∈ [W0 ... W15]

#### Operation:

**For byte operation:**

- \(0 \rightarrow Wd<7>\)
- \((Ws<7:1>) \rightarrow Wd<6:0>\)
- \((Ws<0>) \rightarrow C\)

**For word operation:**

- \(0 \rightarrow Wd<15>\)
- \((Ws<15:1>) \rightarrow Wd<14:0>\)
- \((Ws<0>) \rightarrow C\)

#### Status Affected:

- N, Z, C

#### Encoding:

```
1101 0001 0Bqq qddd dppp ssss
```

#### Description:

Shift the contents of the source register Ws one bit to the right, and place the result in the destination register Wd. The Least Significant bit of Ws is shifted into the Carry bit of the STATUS register. Zero is shifted into the Most Significant bit of Wd. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

#### Words:

1

#### Cycles:

1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
**Example 1:**  

```
  LSR.B W0, W1 ; LSR W0 (Byte mode)  
  ; Store result to W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>FF03</td>
</tr>
<tr>
<td>W1</td>
<td>2378</td>
</tr>
</tbody>
</table>
| SR  | 0000 | SR  | 0001 | (C = 1)

**Example 2:**  

```
  LSR W0, W1 ; LSR W0 (Word mode)  
  ; Store the result to W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>8000</td>
</tr>
<tr>
<td>W1</td>
<td>2378</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
LSR
Logical Shift Right by Short Literal

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: \( \{\text{label:}\} \text{LSR}\ Wb, \#\text{lit4}, \ Wnd \)

Operands:
- \( Wb \in [W0 ... W15] \)
- \( \text{lit4} \in [0 ... 15] \)
- \( \text{Wnd} \in [W0 ... W15] \)

Operation:
- \( \text{lit4}<3:0> \rightarrow \text{Shift}_\text{Val} \)
- \( 0 \rightarrow \text{Wnd}<15:15-\text{Shift}_\text{Val} + 1> \)
- \( \text{Wb}<15:15-\text{Shift}_\text{Val}> \rightarrow \text{Wnd}<15-\text{Shift}_\text{Val}:0> \)

Status Affected: \( N, Z \)

Encoding:

\[
\begin{array}{cccccccc}
1101 & 1110 & 0\text{www} & w\text{dd} & d100 & kkkk \\
\end{array}
\]

Description:
Logical shift right the contents of the source register \( Wb \) by the 4-bit unsigned literal and store the result in the destination register \( \text{Wnd} \). Direct addressing must be used for \( Wb \) and \( \text{Wnd} \).

The 'w' bits select the address of the base register.
The 'd' bits select the destination register.
The 'k' bits provide the literal operand.

Note: This instruction operates in Word mode only.

Words: 1
Cycles: 1

Example 1:
\[
\begin{align*}
\text{LSR} & \ W4, \#14, \ W5 \quad ; \text{LSR} \ W4 \text{ by 14} \\
& \quad \text{; Store result to } W5 \\
\end{align*}
\]

Before Instruction | After Instruction
\[
\begin{array}{c|c}
W4 & C800 \\
W5 & 1200 \\
SR & 0000 \\
\end{array}
\rightarrow
\begin{array}{c|c}
W4 & C800 \\
W5 & 0003 \\
SR & 0000 \\
\end{array}
\]

Example 2:
\[
\begin{align*}
\text{LSR} & \ W4, \#1, \ W5 \quad ; \text{LSR} \ W4 \text{ by 1} \\
& \quad \text{; Store result to } W5 \\
\end{align*}
\]

Before Instruction | After Instruction
\[
\begin{array}{c|c}
W4 & 0505 \\
W5 & F000 \\
SR & 0000 \\
\end{array}
\rightarrow
\begin{array}{c|c}
W4 & 0505 \\
W5 & 0282 \\
SR & 0000 \\
\end{array}
\]
**LSR**

**Logical Shift Right by Wns**

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{(label:)} LSR Wb, Wns, Wnd

**Operands:**

- Wb ∈ [W0 ... W15]
- Wns ∈ [W0 ... W15]
- Wnd ∈ [W0 ... W15]

**Operation:**

- Wns<4:0> → Shift_Val
- 0 → Wnd<15:15-Shift_Val + 1>
- Wb<15:Shift_Val> → Wnd<15 - Shift_Val:0>

**Status Affected:** N, Z

**Encoding:**

```
1101 1110 0www wddd d000 ssss
```

**Description:** Logical shift right the contents of the source register Wb by the 5 Least Significant bits of Wns (only up to 15 positions) and store the result in the destination register Wnd. Direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the destination register.
The 's' bits select the source register.

**Note 1:** This instruction operates in Word mode only.
**Note 2:** If Wns is greater than 15, Wnd will be loaded with 0x0.

**Words:** 1

**Cycles:** 1

**Example 1:**

```
LSR W0, W1, W2 ; LSR W0 by W1
; Store result to W2
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 [C00C]</td>
<td>W0 [C00C]</td>
</tr>
<tr>
<td>W1 [0001]</td>
<td>W1 [0001]</td>
</tr>
<tr>
<td>W2 [2390]</td>
<td>W2 [6006]</td>
</tr>
<tr>
<td>SR [0000]</td>
<td>SR [0000]</td>
</tr>
</tbody>
</table>

**Example 2:**

```
LSR W5, W4, W3 ; LSR W5 by W4
; Store result to W3
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3 [DD43]</td>
<td>W3 [0000]</td>
</tr>
<tr>
<td>W4 [000C]</td>
<td>W4 [000C]</td>
</tr>
<tr>
<td>W5 [0800]</td>
<td>W5 [0800]</td>
</tr>
<tr>
<td>SR [0000]</td>
<td>SR [0002] (Z = 1)</td>
</tr>
</tbody>
</table>
### MAC Multiply and Accumulate

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:** 

\{label:\} MAC \(W_m \times W_n, \text{Acc} \{,[W_x], W_{xd}\} \{,[Wy], Wyd\} \{AWB\}

\{,[W_x] + = kx, W_{xd}\} \{,[W_y] + = ky, Wyd\}

\{,[W_x] – = kx, W_{xd}\} \{,[W_y] – = ky, Wyd\}

\{,[W_9 + W_12], W_{xd}\} \{,[W_{11} + W_{12}], Wyd\}

**Operands:**

\(W_m \times W_n \in [W_4 \times W_5, W_4 \times W_6, W_4 \times W_7, W_5 \times W_6, W_5 \times W_7, W_6 \times W_7]\)

\(\text{Acc} \in [A,B]\)

\(W_x \in [W_8, W_9]\); \(k_x \in [-6, -4, -2, 2, 4, 6]\); \(W_{xd} \in [W_4 \ldots W_7]\)

\(W_y \in [W_{10}, W_{11}]\); \(k_y \in [-6, -4, -2, 2, 4, 6]\); \(W_{yd} \in [W_4 \ldots W_7]\)

\(\text{AWB} \in [W_{13}, [W_{13} + = 2]\)

**Operation:**

\((\text{Acc}(A \text{ or } B)) + (\text{Wm}) \times (\text{Wn}) \rightarrow \text{Acc}(A \text{ or } B)\)

\((\text{Wx}) \rightarrow W_{xd}; (W_x) + k_x \rightarrow W_x\)

\((\text{Wy}) \rightarrow W_{yd}; (W_y) + k_y \rightarrow W_y\)

\((\text{Acc}(B \text{ or } A)) \text{ rounded} \rightarrow \text{AWB}\)

**Status Affected:** OA, OB, OAB, SA, SB, SAB

**Encoding:**

| 1100 | 0mmm | A0xx | yiii | ijjj | jjaa |

**Description:**

Multiply the contents of two working registers, optionally prefetch operands in preparation for another MAC type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40 bits and added to the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional prefetch operations, which support indirect and register offset addressing, as described in Section 4.14.1 “MAC Prefetches”. Operand AWB specifies the optional store of the “other” accumulator, as described in Section 4.14.4 “MAC Write Back”.

The ‘m’ bits select the operand registers Wm and Wn for the multiply.

The ‘A’ bit selects the accumulator for the result.

The ‘x’ bits select the prefetch Wxd destination.

The ‘y’ bits select the prefetch Wyd destination.

The ‘i’ bits select the Wx prefetch operation.

The ‘j’ bits select the Wy prefetch operation.

The ‘a’ bits select the accumulator Write Back destination.

**Note 1:** The IF bit (CORCON<0>), determines if the multiply is fractional or an integer.

**Note 2:** The US<1:0> bits (CORCON<13:12> in dsPIC33E, CORCON<12> in dsPIC30F/dsPIC33F) determine if the multiply is unsigned, signed, or mixed-sign. Only dsPIC33E devices support mixed-sign multiplication.

**Words:** 1

**Cycles:** 1
**Example 1:**

; Multiply W4*W5 and add to ACCA
; Fetch [W8] to W4, Post-increment W8 by 6
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x00C0 (fractional multiply, normal saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>2567</td>
</tr>
<tr>
<td>W5</td>
<td>909C</td>
</tr>
<tr>
<td>W8</td>
<td>0A06</td>
</tr>
<tr>
<td>W10</td>
<td>1802</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 1200 0000</td>
</tr>
<tr>
<td>Data 0A00</td>
<td>2567</td>
</tr>
<tr>
<td>Data 1800</td>
<td>909C</td>
</tr>
<tr>
<td>CORCON</td>
<td>00C0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

; Multiply W4*W5 and add to ACCA
; Fetch [W8] to W4, Post-decrement W8 by 2
; Fetch [W10] to W5, Post-increment W10 by 2
; Write Back ACCB to W13
; CORCON = 0x00D0 (fractional multiply, super saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>5BBE</td>
</tr>
<tr>
<td>W5</td>
<td>C967</td>
</tr>
<tr>
<td>W8</td>
<td>09FE</td>
</tr>
<tr>
<td>W10</td>
<td>1802</td>
</tr>
<tr>
<td>W13</td>
<td>0001</td>
</tr>
<tr>
<td>ACCA</td>
<td>23 5000 2000</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 0000 8F4C</td>
</tr>
<tr>
<td>Data 0A00</td>
<td>5BBE</td>
</tr>
<tr>
<td>Data 1800</td>
<td>C967</td>
</tr>
<tr>
<td>CORCON</td>
<td>00D0</td>
</tr>
<tr>
<td>SR</td>
<td>8800 (OA, OAB = 1)</td>
</tr>
</tbody>
</table>
### MAC Square and Accumulate

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} MAC        Wm*Wm, Acc    {,[Wx], Wxd}    {,[Wy], Wyd}
{[Wx] + = kx, Wxd}  {[Wy] + = ky, Wyd}
{[W9 + W12], Wxd}   {[W11 + W12], Wyd}
```

**Operands:**

- \( Wm \times Wm \in \{W4 \times W4, W5 \times W5, W6 \times W6, W7 \times W7\} \)
- \( Acc \in \{A, B\} \)
- \( Wx \in \{W8, W9\} \); \( kx \in \{-6, -4, -2, 2, 4, 6\} \)
- \( Wxd \in \{W4 \ldots W7\} \)
- \( Wy \in \{W10, W11\} \); \( ky \in \{-6, -4, -2, 2, 4, 6\} \)
- \( Wyd \in \{W4 \ldots W7\} \)

**Operation:**

- \((Acc(A \text{ or } B)) + (Wm) \times (Wm) \rightarrow Acc(A \text{ or } B)\)
- \(([Wx]) \rightarrow Wxd; (Wx) + kx \rightarrow Wx\)
- \(([Wy]) \rightarrow Wyd; (Wy) + ky \rightarrow Wy\)

**Status Affected:**

- OA, OB, OAB, SA, SB, SAB

**Encoding:**

| 1111 | 00mm | A0xx | yyii | ii jj | jj00 |

**Description:**

Square the contents of a working register, optionally prefetch operands in preparation for another MAC type instruction. The 32-bit result of the signed multiply is sign-extended to 40 bits and added to the specified accumulator.

Operands \( Wx, Wxd, Wy \) and \( Wyd \) specify optional prefetch operations, which support indirect and register offset addressing, as described in Section 4.14.1 “MAC Prefetches”.

The ‘\( m \)’ bits select the operand register \( Wm \) for the square.
The ‘\( A \)’ bit selects the accumulator for the result.
The ‘\( x \)’ bits select the prefetch \( Wxd \) destination.
The ‘\( y \)’ bits select the prefetch \( Wyd \) destination.
The ‘\( i \)’ bits select the \( Wx \) prefetch operation.
The ‘\( j \)’ bits select the \( Wy \) prefetch operation.

**Note 1:**

The IF bit (CORCON<0>), determines if the multiply is fractional or an integer.

**Note 2:**

The US<1:0> bits (CORCON<13:12> in dsPIC33E, CORCON<12> in dsPIC30F/dsPIC33F) determine if the multiply is unsigned, signed, or mixed-sign. Only dsPIC33E devices support mixed-sign multiplication.

**Words:**

1

**Cycles:**

1
**Example 1:**

MAC \(W4\cdot W4\), B, \([W9+W12]\), W4, \([W10]\)--2, W5

; Square W4 and add to ACCB
; Fetch \([W9+W12]\) to W4
; Fetch \([W10]\) to W5, Post-decrement W10 by 2
; CORCON = 0x00C0 (fractional multiply, normal saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>A022</td>
</tr>
<tr>
<td>W5</td>
<td>B200</td>
</tr>
<tr>
<td>W9</td>
<td>0C00</td>
</tr>
<tr>
<td>W10</td>
<td>1900</td>
</tr>
<tr>
<td>W12</td>
<td>0020</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 2000 0000</td>
</tr>
<tr>
<td>Data 0C20</td>
<td>A230</td>
</tr>
<tr>
<td>Data 1900</td>
<td>650B</td>
</tr>
<tr>
<td>CORCON</td>
<td>00C0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W4</td>
<td>A230</td>
</tr>
<tr>
<td>W5</td>
<td>650B</td>
</tr>
<tr>
<td>W9</td>
<td>0C00</td>
</tr>
<tr>
<td>W10</td>
<td>18FE</td>
</tr>
<tr>
<td>W12</td>
<td>0020</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 67CD 0908</td>
</tr>
<tr>
<td>Data 0C20</td>
<td>A230</td>
</tr>
<tr>
<td>Data 1900</td>
<td>650B</td>
</tr>
<tr>
<td>CORCON</td>
<td>00C0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

MAC \(W7\cdot W7\), A, \([W11]\)--2, W7

; Square W7 and add to ACCA
; Fetch \([W11]\) to W7, Post-decrement W11 by 2
; CORCON = 0x00D0 (fractional multiply, super saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>76AE</td>
</tr>
<tr>
<td>W11</td>
<td>2000</td>
</tr>
<tr>
<td>ACCA</td>
<td>FE 9834 4500</td>
</tr>
<tr>
<td>Data 2000</td>
<td>23FF</td>
</tr>
<tr>
<td>CORCON</td>
<td>00D0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W7</td>
<td>23FF</td>
</tr>
<tr>
<td>W11</td>
<td>1FFE</td>
</tr>
<tr>
<td>ACCA</td>
<td>FF 063E 0188</td>
</tr>
<tr>
<td>Data 2000</td>
<td>23FF</td>
</tr>
<tr>
<td>CORCON</td>
<td>00D0</td>
</tr>
</tbody>
</table>
| SR                 | 8800             (OA, OAB = 1)
Section 5. Instruction Descriptions

MOV

Move f to Destination

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

\{label:\} MOV{.B} f {,WREG}

Operands:

f ∈ [0 .. 8191]

Operation:

(f) → destination designated by D

Status Affected:

N, Z

Encoding:

| 1011 | 1111 | lBdf | ffff | ffff | ffff |

Description:

Move the contents of the specified file register to the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored back to the file register and the only effect is to modify the STATUS register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

3: When moving word data from file register memory, the "MOV f to Wnd" (page 281) instruction allows any working register (W0:W15) to be the destination register.

Words: 1

Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

MOV.B TMR0, WREG ; move (TMR0) to WREG (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>9080</td>
</tr>
<tr>
<td>TMR0</td>
<td>2355</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>WREG (W0)</td>
<td>9055</td>
</tr>
<tr>
<td>TMR0</td>
<td>2355</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

MOV 0x800 ; update SR based on (0x800) (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 0800</td>
<td>B29F</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>Data 0800</td>
<td>B29F</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
MOV
Move WREG to f

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E
X X X X X X X

Syntax: {label:} MOV.{B} WREG, f

Operands: f ∈ [0 ... 8191]

Operation: (WREG) → f

Status Affected: None

Encoding:

| 1011 | 0111 | 1B1f | ffff | ffff | ffff |

Description: Move the contents of the default working register WREG into the specified file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte). The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

2: The WREG is set to working register W0.

3: When moving word data from the working register array to file register memory, the "MOV Wns to f" (page 282) instruction allows any working register (W0:W15) to be the source register.

Words: 1
Cycles: 1

Example 1: MOV.B WREG, 0x801 ; move WREG to 0x801 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>98F3</td>
</tr>
<tr>
<td>Data 0800</td>
<td>4509</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

(N = 1)

Example 2: MOV WREG, DISICNT ; move WREG to DISICNT

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>00A0</td>
</tr>
<tr>
<td>DISICNT</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISICNT</td>
<td>00A0</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MOV
Move f to Wnd

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} MOV f, Wnd

Operands:
- f ∈ [0 ... 65534]
- Wnd ∈ [W0 ... W15]

Operation: (f) → Wnd

Status Affected: None

Encoding:

|   | 1000 | 0fff | ffff | ffff | ffff | dddd |

Description:
Move the word contents of the specified file register to Wnd. The file register may reside anywhere in the 32K words of data memory, but must be word-aligned. Register direct addressing must be used for Wnd.

The 'f' bits select the address of the file register.
The 'd' bits select the destination register.

**Note 1:** This instruction operates on word operands only.

2: Since the file register address must be word-aligned, only the upper 15 bits of the file register address are encoded (bit 0 is assumed to be '0').

3: To move a byte of data from file register memory, the "MOV f to Destination" instruction (page 279) may be used.

Words: 1
Cycles: 1(1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 "Multi-Cycle Instructions".

Example 1:
MOV CORCON, W12 ; move CORCON to W12

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W12</td>
<td>W12</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>78FA</td>
<td>00F0</td>
</tr>
<tr>
<td>00F0</td>
<td>00F0</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:
MOV 0x27FE, W3 ; move (0x27FE) to W3

Before Instruction | After Instruction
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>Data 27FE</td>
<td>Data 27FE</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0035</td>
<td>ABCD</td>
</tr>
<tr>
<td>ABCD</td>
<td>ABCD</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
## MOV

**Move Wns to f**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} MOV Wns, f
```

**Operands:**

- \( f \in [0 \ldots 65534] \)
- \( Wns \in [W0 \ldots W15] \)

**Operation:**

\[(Wns) \rightarrow f\]

**Status Affected:** None

**Encoding:**

| 1000 | lfff | ffff | ffff | ffff | ssss |

**Description:**

Move the word contents of the working register \( Wns \) to the specified file register. The file register may reside anywhere in the 32K words of data memory, but must be word-aligned. Register direct addressing must be used for \( Wn \).

The ‘\( f \)’ bits select the address of the file register.

The ‘\( s \)’ bits select the source register.

**Note 1:** This instruction operates on word operands only.

2. Since the file register address must be word-aligned, only the upper 15 bits of the file register address are encoded (bit 0 is assumed to be ‘0’).

3. To move a byte of data to file register memory, the “MOV WREG to f” instruction (page 280) may be used.

**Words:** 1

**Cycles:** 1

**Example 1:**

MOV W4, XMODSRT ; move W4 to XMODSRT

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 1200</td>
<td>W4 1200</td>
</tr>
<tr>
<td>XMODSRT 1340</td>
<td>XMODSRT 1200</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

MOV W8, 0x1222 ; move W8 to data address 0x1222

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8 F200</td>
<td>W8 F200</td>
</tr>
<tr>
<td>Data 1222 FD88</td>
<td>Data 1222 F200</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
**MOV.B**  
*Move 8-bit Literal to Wnd*

---

### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

### Syntax:

{label:} MOV.B #lit8, Wnd

### Operands:

lit8 ∈ [0 ... 255]  
Wnd ∈ [W0 ... W15]

### Operation:

lit8 → Wnd

### Status Affected:

None

### Encoding:

| 1011 | 0011 | 1100 | kkkk | kkkk | dddd |

### Description:

The unsigned 8-bit literal ‘k’ is loaded into the lower byte of Wnd. The upper byte of Wnd is not changed. Register direct addressing must be used for Wnd.

The ‘k’ bits specify the value of the literal.  
The ‘d’ bits select the address of the working register.

**Note:** This instruction operates in Byte mode and the .B extension must be provided.

### Words:

1

### Cycles:

1

---

**Example 1:**

MOV.B #0x17, W5  
; load W5 with #0x17 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5 7899</td>
<td>W5 7817</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

MOV.B #0xFE, W9  
; load W9 with #0xFE (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W9 AB23</td>
<td>W9 ABFE</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
**MOV**

**Move 16-bit Literal to Wnd**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} MOV #lit16, Wnd

**Operands:**

lit16 ∈ [-32768 ... 65535]

Wnd ∈ [W0 ... W15]

**Operation:**

lit16 → Wnd

**Status Affected:** None

**Encoding:**

| 0010 | kkkk | kkkk | kkkk | kkkk | dddd |

**Description:**

The 16-bit literal 'k' is loaded into Wnd. Register direct addressing must be used for Wnd.

The 'k' bits specify the value of the literal.

The 'd' bits select the address of the working register.

**Note 1:**

This instruction operates only in Word mode.

2: The literal may be specified as a signed value [-32768:32767], or unsigned value [0:65535].

**Words:** 1

**Cycles:** 1

**Example 1:**

MOV #0x4231, W13 ; load W13 with #0x4231

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W13 091B</td>
<td>W13 4231</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

MOV #0x4, W2 ; load W2 with #0x4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 B004</td>
<td>W2 0004</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 3:**

MOV #-1000, W8 ; load W8 with #-1000

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8 23FF</td>
<td>W8 FC18</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**MOV**

Move [Ws with offset] to Wnd

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

\{label:} MOV{.B} [Ws + Slit10], Wnd

Operands:

Ws ∈ [W0 ... W15]
Slit10 ∈ [-512 ... 511] for byte operation
Slit10 ∈ [-1024 ... 1022] (even only) for word operation
Wnd ∈ [W0 ... W15]

Operation:  

[Ws + Slit10] → Wnd

Status Affected:  
None

Encoding:

| 1001 | 0kkk | kBkk | kddd | dkkk | ssss |

Description:

The contents of [Ws + Slit10] are loaded into Wnd. In Word mode, the range of Slit10 is increased to [-1024 ... 1022] and Slit10 must be even to maintain word address alignment. Register indirect addressing must be used for the source, and direct addressing must be used for Wnd.

The 'k' bits specify the value of the literal.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'd' bits select the destination register.

The 's' bits select the source register.

**Note 1:** The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

**Note 2:** In Byte mode, the range of Slit10 is not reduced as specified in Section 4.6 “Using 10-bit Literal Operands”, since the literal represents an address offset from Ws.

Words:  
1

Cycles:  
1(1)

**Example 1:**

MOV.B [W8+0x13], W10 ; load W10 with [W8+0x13] ; (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8 1008</td>
<td>W8 1008</td>
</tr>
<tr>
<td>W10 4009</td>
<td>W10 4033</td>
</tr>
<tr>
<td>Data 101A 3312</td>
<td>Data 101A 3312</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
Example 2: MOV [W4+0x3E8], W2 ; load W2 with [W4+0x3E8] ; (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>Data 0BE8</td>
<td>Data 0BE8</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>9088</td>
<td>5634</td>
</tr>
<tr>
<td>0800</td>
<td>0800</td>
</tr>
<tr>
<td>5634</td>
<td>5634</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

MOV Move Wns to [Wd with offset]

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} MOV{.B} Wns, [Wd + Slit10]

Operands: Wns ∈ [W0 ... W15]
Slit10 ∈ [-512 ... 511] in Byte mode
Slit10 ∈ [-1024 ... 1022] (even only) in Word mode
Wd ∈ [W0 ... W15]

Operation: (Wns) → [Wd + Slit10]

Status Affected: None

Encoding: 1001 1kkk kBkk kddd dkkk ssss

Description: The contents of Wns are stored to [Wd + Slit10]. In Word mode, the range of Slit10 is increased to [-1024 ... 1022] and Slit10 must be even to maintain word address alignment. Register direct addressing must be used for Wns, and indirect addressing must be used for the destination.

The ‘k’ bits specify the value of the literal.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘d’ bits select the destination register.
The ‘s’ bits select the source register.

Note 1: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

2: In Byte mode, the range of Slit10 is not reduced as specified in Section 4.6 “Using 10-bit Literal Operands”, since the literal represents an address offset from Wd.

Words: 1
Cycles: 1

Example 1: MOV.B W0, [W1+0x7] ; store W0 to [W1+0x7] ; (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>Data 1806</td>
<td>Data 1806</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>9015</td>
<td>9015</td>
</tr>
<tr>
<td>1800</td>
<td>1800</td>
</tr>
<tr>
<td>2345</td>
<td>1545</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
### Example 2:

MOV W11, [W1-0x400] ; store W11 to [W1-0x400] ; (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>1000</td>
</tr>
<tr>
<td>W11</td>
<td>8813</td>
</tr>
<tr>
<td>Data 0C00</td>
<td>FFEA</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

#### MOV

**Move Ws to Wd**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label\} MOV{.B} Ws, Wd

- \[Ws\], \[Wd\]
- \[Ws++\], \[Wd++\]
- \[Ws--\], \[Wd--\]
- \[--Ws\], \[--Wd\]
- \[++Ws\], \[++Wd\]
- \[Ws + Wb\], \[Wd + Wb\]

**Operands:**

- \(Ws \in [W0 \ldots W15]\)
- \(Wb \in [W0 \ldots W15]\)
- \(Wd \in [W0 \ldots W15]\)

**Operation:**

\((Ws) \to Wd\)

**Status Affected:**

None

**Encoding:**

| 0111 | 1www | wBhh | hddd | dggg | ssss |

**Description:**

Move the contents of the source register into the destination register. Either register direct or indirect addressing may be used for \(Ws\) and \(Wd\).

The ‘w’ bits define the offset register \(Wb\).

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘h’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘g’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note 1:**

The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

**Note 2:**

When Register Offset Addressing mode is used for both the source and destination, the offset must be the same because the ‘w’ encoding bits are shared by \(Ws\) and \(Wd\).

**Note 3:**

The instruction "PUSH Ws" translates to MOV \(Ws, [W15++]\).

**Note 4:**

The instruction "POP Wd" translates to MOV \([--W15], Wd\).
MOV

Move Ws to Wd

Words: 1
Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: MOV.B  [W0--], W4 ; Move [W0] to W4 (Byte mode) ; Post-decrement W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0A01</td>
<td>W0 0A00</td>
</tr>
<tr>
<td>W4 2976</td>
<td>W4 2989</td>
</tr>
<tr>
<td>Data 0A00</td>
<td>Data 0A00</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 0800</td>
<td>W2 0800</td>
</tr>
<tr>
<td>W3 0040</td>
<td>W3 0040</td>
</tr>
<tr>
<td>W6 1228</td>
<td>W6 122A</td>
</tr>
<tr>
<td>Data 0840</td>
<td>Data 0840</td>
</tr>
<tr>
<td>Data 1228</td>
<td>Data 1228</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

### MOV.D

**Double Word Move from Source to Wnd**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} MOV.D Wns, Wnd

- [Ws],
- [Ws++],
- [Ws--],
- [++Ws],
- [--Ws],

**Operands:**

- Wns ∈ [W0, W2, W4 ... W14]
- Ws ∈ [W0 ... W15]
- Wnd ∈ [W0, W2, W4 ... W14]

**Operation:**

For direct addressing of source:

- Wns → Wnd
- Wns + 1 → Wnd + 1

For indirect addressing of source:

See Description

**Status Affected:** None

**Encoding:**

```
1011 1110 0000 0ddd 0ppp ssss
```

**Description:**

Move the double word specified by the source to a destination working register pair (Wnd:Wnd + 1). If register direct addressing is used for the source, the contents of two successive working registers (Wns:Wns + 1) are moved to Wnd:Wnd + 1. If indirect addressing is used for the source, Ws specifies the effective address for the least significant word of the double word. Any pre/post-increment or pre/post-decrement will adjust Ws by 4 bytes to accommodate for the double word.

The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the address of the first source register.

**Note 1:** This instruction only operates on double words. See Figure 4-3 for information on how double words are aligned in memory.

**Note 2:** Wnd must be an even working register.

**Note 3:** The instruction “POP.D Wnd” translates to MOV.D [--W15], Wnd.

**Words:** 1

**Cycles:** 2

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Note 1: This instruction only operates on double words. See Figure 4-3 for information on how double words are aligned in memory.

2: Wnd must be an even working register.

3: The instruction “POP.D Wnd” translates to MOV.D [--W15], Wnd.
### Example 1:

MOV.D W2, W6 ; Move W2 to W6 (Double mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2: 12FB</td>
<td>W2: 12FB</td>
</tr>
<tr>
<td>W3: 9877</td>
<td>W3: 9877</td>
</tr>
<tr>
<td>W6: 9833</td>
<td>W6: 12FB</td>
</tr>
<tr>
<td>W7: FCC6</td>
<td>W7: 9877</td>
</tr>
<tr>
<td>SR: 0000</td>
<td>SR: 0000</td>
</tr>
</tbody>
</table>

### Example 2:

MOV.D [W7--], W4 ; Move [W7] to W4 (Double mode) ; Post-decrement W7

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4: B012</td>
<td>W4: A319</td>
</tr>
<tr>
<td>W5: FD89</td>
<td>W5: 9927</td>
</tr>
<tr>
<td>W7: 0900</td>
<td>W7: 08FC</td>
</tr>
<tr>
<td>Data 0900: A319</td>
<td>Data 0900: A319</td>
</tr>
<tr>
<td>Data 0902: 9927</td>
<td>Data 0902: 9927</td>
</tr>
<tr>
<td>SR: 0000</td>
<td>SR: 0000</td>
</tr>
</tbody>
</table>
### MOVPAG

**Move Literal to Page Register**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} MOVPAG #lit10, DSRPAG

#lit9, DSWPAG

#lit8, TBLPAG

**Operands:**

lit10 ∈ [0 ... 1023], lit9 ∈ [0 ... 511], lit8 ∈ [0 ... 255]

**Operation:**

lit10 → DSRPAG or lit9 → DSWPAG or lit8 → TBLPAG

**Status Affected:** None

**Encoding:**

| 1111 | 1110 | 1100 | PPkk | kkkk | kkkk |

**Description:**

The appropriate number of bits from the unsigned literal 'k' are loaded into the DSRPAG, DSWPAG, or TBLPAG register. The assembler restricts the literal to a 9-bit unsigned value when the destination is DSWPAG, and an 8-bit unsigned value when the destination is TBLPAG.

The 'P' bits select the destination register.
The 'k' bits specify the value of the literal.

**Note:** This instruction operates in word mode only.

**Words:** 1

**Cycles:** 1

**Example 1:** MOVPAG #0x02, DSRPAG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSRPAG 0000</td>
<td>DSRPAG 0002</td>
</tr>
</tbody>
</table>
### MOVPAG

**Move Ws to Page Register**

- **Implemented in:**
  - PIC24F
  - PIC24H
  - PIC24E
  - dsPIC30F
  - dsPIC33F
  - dsPIC33E

- **Syntax:**
  
  `{label:} MOVPAG Wn, DSRPAG
  
  DSWPAG
  
  TBLPAG

- **Operands:**
  
  \( Wn \in [W0 \ldots W15] \)

- **Operation:**
  
  \( Wn<9:0> \rightarrow \text{DSRPAG or Wn}<8:0> \rightarrow \text{DSWPAG or Wn}<7:0> \rightarrow \text{TBLPAG} \)

- **Status Affected:**
  
  None

- **Encoding:**
  
  | 1111 | 1110 | 1101 | PP00 | 0000 | ssss |

- **Description:**

  The appropriate number of bits from the register Ws are loaded into the DSRPAG, DSWPAG, or TBLPAG register. The assembler restricts the literal to a 9-bit unsigned value when the destination is DSWPAG, and an 8-bit unsigned value when the destination is TBLPAG.

  The ‘P’ bits select the destination register.

  The ‘s’ bits specify the source register.

  **Note:** This instruction operates in word mode only.

- **Words:**
  
  1

- **Cycles:**
  
  1

**Example 1:**

```
Example 1: MOVPAG W2, DSRPAG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSRPAG 0000</td>
<td>DSRPAG 0002</td>
</tr>
<tr>
<td>W2 0002</td>
<td>W2 0002</td>
</tr>
</tbody>
</table>
```
**MOVSA C**  
Prefetch Operands and Store Accumulator

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X |

**Syntax:**  
{label:} MOVSA C Acc {,[Wx], Wxd} {,[Wy], Wyd} {,AWB}  
{,[Wx] + = kx, Wxd} {,[Wy] + = ky, Wyd}  
{,[Wx] – = kx, Wxd} {,[Wy] – = ky, Wyd}  
{,[W9 + W12], Wxd} {,[W11 + W12], Wyd}  

**Operands:**  
Acc ∈ [A,B]  
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]  
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]  
AWB ∈ [W13, [W13] + = 2]  

**Operation:**  
([Wx]) →Wxd; (Wx) + kx →Wx  
([Wy]) →Wyd; (Wy) + ky →Wy  
(Acc(B or A)) rounded →AWB

**Status Affected:** None

**Encoding:**  
```
1100  0111  A0xx  yyii  ijjj  jjaa
```

**Description:**  
Optionally prefetch operands in preparation for another MAC type instruction and optionally store the unspecified accumulator results. Even though an accumulator operation is not performed in this instruction, an accumulator must be specified to designate which accumulator to write back.

Operands Wx, Wxd, Wy and Wyd specify optional prefetch operations which support indirect and register offset addressing, as described in Section 4.14.1 “MAC Prefetches”. Operand AWB specifies the optional store of the “other” accumulator, as described in Section 4.14.4 “MAC Write Back”.

The ‘A’ bit selects the other accumulator used for write back.  
The ‘x’ bits select the prefetch Wxd destination.  
The ‘y’ bits select the prefetch Wyd destination.  
The ‘i’ bits select the Wx prefetch operation.  
The ‘j’ bits select the Wy prefetch operation.  
The ‘a’ bits select the accumulator Write Back destination.

**Words:** 1  
**Cycles:** 1
Example 1:

```
; Fetch [W9] to W6
; Fetch [W11] to W7, Post-increment W11 by 4
; Store ACCA to W13
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>A022</td>
</tr>
<tr>
<td>W7</td>
<td>B200</td>
</tr>
<tr>
<td>W9</td>
<td>0800</td>
</tr>
<tr>
<td>W11</td>
<td>1900</td>
</tr>
<tr>
<td>W13</td>
<td>0020</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 3290 5968</td>
</tr>
<tr>
<td>Data 0800</td>
<td>7811</td>
</tr>
<tr>
<td>Data 1900</td>
<td>B2AF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7811</td>
</tr>
<tr>
<td></td>
<td>B2AF</td>
</tr>
<tr>
<td></td>
<td>1904</td>
</tr>
<tr>
<td></td>
<td>3290</td>
</tr>
<tr>
<td></td>
<td>00 3290 5968</td>
</tr>
<tr>
<td></td>
<td>7811</td>
</tr>
<tr>
<td></td>
<td>B2AF</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

```
; Fetch [W9] to W4, Post-decrement W9 by 2
; Fetch [W11+W12] to W6
; Store ACCB to [W13], Post-increment W13 by 2
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>76AE</td>
</tr>
<tr>
<td>W6</td>
<td>2000</td>
</tr>
<tr>
<td>W9</td>
<td>1200</td>
</tr>
<tr>
<td>W11</td>
<td>2000</td>
</tr>
<tr>
<td>W12</td>
<td>0024</td>
</tr>
<tr>
<td>W13</td>
<td>2300</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 9834 4500</td>
</tr>
<tr>
<td>Data 1200</td>
<td>BB00</td>
</tr>
<tr>
<td>Data 2024</td>
<td>52CE</td>
</tr>
<tr>
<td>Data 2300</td>
<td>23FF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

|                   |                   |
|                   | BB00              |
|                   | 52CE              |
|                   | 23FF              |
|                   | 0000              |
MPY

Multiply Wm by Wn to Accumulator

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax: {label:} MPY Wm * Wn, Acc {[Wx], Wxd} {[Wy], Wyd}
{[Wx] = kx, Wxd} {[Wy] = ky, Wyd}
{[Wx] + = kx, Wxd} {[Wy] – = ky, Wyd}
{[W9 + W12], Wxd} {[W11 + W12], Wyd}

Operands:

Acc ∈ [A, B]
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]
AWB ∈ [W13], [W13] + = 2

Operation:

(Wm) * (Wn) → Acc(A or B)
([Wx]) – Wxd; (Wx) + kx → Wx
([Wy]) – Wyd; (Wy) + ky → Wy

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

| 1100 | 0mmm | A0xx | yyii | iijj | jj11 |

Description:

Multiply the contents of two working registers, optionally prefetch operands in preparation for another MAC type instruction. The 32-bit result of the signed multiply is sign-extended to 40 bits and stored to the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional prefetch operations which support indirect and register offset addressing, as described in Section 4.14.1 “MAC Prefetches”.

The ‘m’ bits select the operand registers Wm and Wn for the multiply:
The ‘A’ bit selects the accumulator for the result.
The ‘x’ bits select the prefetch Wxd destination.
The ‘y’ bits select the prefetch Wyd destination.
The ‘i’ bits select the Wx prefetch operation.
The ‘j’ bits select the Wy prefetch operation.

Note 1: The IF bit, CORCON<0>, determines if the multiply is fractional or an integer.

2: The US<1:0> bits (CORCON<13:12> in dsPIC33E, CORCON<12> in dsPIC30F/dsPIC33F) determine if the multiply is unsigned, signed, or mixed-sign. Only dsPIC33E devices support mixed-sign multiplication.

Words: 1
Cycles: 1
**Example 1:**

```
; Multiply W4*W5 and store to ACCA
; Fetch [W8] to W6, Post-increment W8 by 2
; Fetch [W10] to W7, Post-decrement W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>C000</td>
</tr>
<tr>
<td>W5</td>
<td>9000</td>
</tr>
<tr>
<td>W6</td>
<td>0800</td>
</tr>
<tr>
<td>W7</td>
<td>B200</td>
</tr>
<tr>
<td>W8</td>
<td>1780</td>
</tr>
<tr>
<td>W10</td>
<td>2400</td>
</tr>
<tr>
<td>ACCA</td>
<td>FF F780 2087</td>
</tr>
<tr>
<td>Data 1780</td>
<td>671F</td>
</tr>
<tr>
<td>Data 2400</td>
<td>E3DC</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>C000</td>
</tr>
<tr>
<td>W5</td>
<td>9000</td>
</tr>
<tr>
<td>W6</td>
<td>671F</td>
</tr>
<tr>
<td>W7</td>
<td>E3DC</td>
</tr>
<tr>
<td>W8</td>
<td>1782</td>
</tr>
<tr>
<td>W10</td>
<td>23FE</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 9834 4500</td>
</tr>
<tr>
<td>Data 1780</td>
<td>8FDC</td>
</tr>
<tr>
<td>Data 23FE</td>
<td>0078</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Data 0x671F**

**Example 2:**

```
; Multiply W6*W7 and store to ACCB
; Fetch [W8] to W4, Post-increment W8 by 2
; Fetch [W10] to W5, Post-decrement W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>C000</td>
</tr>
<tr>
<td>W5</td>
<td>9000</td>
</tr>
<tr>
<td>W6</td>
<td>0800</td>
</tr>
<tr>
<td>W7</td>
<td>B200</td>
</tr>
<tr>
<td>W8</td>
<td>1780</td>
</tr>
<tr>
<td>W10</td>
<td>2400</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 3800 0000</td>
</tr>
<tr>
<td>Data 1780</td>
<td>671F</td>
</tr>
<tr>
<td>Data 2400</td>
<td>E3DC</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>8FDC</td>
</tr>
<tr>
<td>W5</td>
<td>0078</td>
</tr>
<tr>
<td>W6</td>
<td>671F</td>
</tr>
<tr>
<td>W7</td>
<td>E3DC</td>
</tr>
<tr>
<td>W8</td>
<td>1784</td>
</tr>
<tr>
<td>W10</td>
<td>23FC</td>
</tr>
<tr>
<td>ACCB</td>
<td>FF E954 3748</td>
</tr>
<tr>
<td>Data 1782</td>
<td>8FDC</td>
</tr>
<tr>
<td>Data 23FE</td>
<td>0078</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
**MPY**  
Square to Accumulator

**Implemented in:**  
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**  
{label:} MPY  Wm * Wm, Acc  Wm {,Wx}, Wxd  Wm {,Wy}, Wyd  
{,[Wx] += kx, Wxd}  {,[Wy] += ky, Wyd}  
{,[Wx] -= kx, Wxd}  {,[Wy] -= ky, Wyd}  
{,[W9 + W12], Wxd}  {,[W11 + W12], Wyd}

**Operands:**  
Wm * Wm ∈ [W4 * W4, W5 * W5, W6 * W6, W7 * W7]  
Acc ∈ [A,B]  
Wx ∈ [W8, W9]; kx ∈ [-6, -4, -2, 2, 4, 6]; Wxd ∈ [W4 ... W7]  
Wy ∈ [W10, W11]; ky ∈ [-6, -4, -2, 2, 4, 6]; Wyd ∈ [W4 ... W7]

**Operation:**  
(Wm) * (Wm) → Acc(A or B)  
([Wx]) → Wxd; (Wx) + kx → Wx  
([Wy]) → Wyd; (Wy) + ky → Wy

**Status Affected:**  
OA, OB, OAB, SA, SB, SAB

**Encoding:**  
| 1111 | 00mm | A0xx | yyii | iijj | jj01 |

**Description:**  
Square the contents of a working register, optionally prefetch operands in preparation for another MAC type instruction. The 32-bit result of the signed multiply is sign-extended to 40 bits and stored in the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional prefetch operations which support indirect and register offset addressing, as described in Section 4.14.1 "MAC Prefetches”.

The 'm' bits select the operand register Wm for the square.  
The 'A' bit selects the accumulator for the result.  
The 'x' bits select the prefetch Wxd destination.  
The 'y' bits select the prefetch Wyd destination.  
The 'i' bits select the Wx prefetch operation.  
The 'j' bits select the Wy prefetch operation.

**Note 1:**  
The IF bit (CORCON<0>), determines if the multiply is fractional or an integer.

2:  
The US<1:0> bits (CORCON<13:12> in dsPIC33E, CORCON<12> in dsPIC30F/dsPIC33F) determine if the multiply is unsigned, signed, or mixed-sign. Only dsPIC33E devices support mixed-sign multiplication.

**Words:** 1

**Cycles:** 1
### Example 1:

**MPY** W6\*W6, A, [W9]+, W6  
; Square W6 and store to ACCA  
; Fetch [W9] to W6, Post-increment W9 by 2  
; CORCON = 0x0000 (fractional multiply, no saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>6500</td>
</tr>
<tr>
<td>W9</td>
<td>0900</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 7C80 0908</td>
</tr>
<tr>
<td>Data 0900</td>
<td>B865</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

### Example 2:

**MPY** W4\*W4, B, [W9+W12], W4, [W10]+, W5  
; Square W4 and store to ACCB  
; Fetch [W9+W12] to W4  
; Fetch [W10] to W5, Post-increment W10 by 2  
; CORCON = 0x0000 (fractional multiply, no saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>E228</td>
</tr>
<tr>
<td>W5</td>
<td>9000</td>
</tr>
<tr>
<td>W9</td>
<td>1700</td>
</tr>
<tr>
<td>W10</td>
<td>1B00</td>
</tr>
<tr>
<td>W12</td>
<td>FF00</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 9834 4500</td>
</tr>
<tr>
<td>Data 1600</td>
<td>8911</td>
</tr>
<tr>
<td>Data 1B00</td>
<td>F678</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MPY.N
Multiply -Wm by Wn to Accumulator

Implemented in:  
<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:  
{label:} MPY.N Wm * Wn, Acc  
{,[Wx], Wxd}  
{,[Wy], Wyd}  
{,[Wx] + = kx, Wxd}  
{,[Wy] + = ky, Wyd}  
{,[Wx] – = kx, Wxd}  
{,[Wy] – = ky, Wyd}  
{,[W9 + W12], Wxd}  
{,[W11 + W12], Wyd}  

Operands:  
Wm * Wn ε [W4 * W5; W4 * W6; W4 * W7; W5 * W6; W5 * W7; W6 * W7]  
Acc ε [A,B]  
Wx ε [W8, W9]; kx ε [-6, -4, -2, 2, 4, 6]; Wxd ε [W4 ... W7]  
Wy ε [W10, W11]; ky ε [-6, -4, -2, 2, 4, 6]; Wyd ε [W4 ... W7]  

Operation:  
-(Wm) * (Wn) →Acc(A or B)  
([Wx]) →Wxd; (Wx) + kx →Wx  
([Wy]) →Wyd; (Wy) + ky →Wy  

Status Affected:  
OA, OB, OAB  

Encoding:  
| 1100 | 0mmm | A1xx | yyii | iijj | jj11 |

Description:  
Multiply the contents of a working register by the negative of the contents of another working register, optionally prefetch operands in preparation for another MAC type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40 bits and stored to the specified accumulator.

The 'm' bits select the operand registers Wm and Wn for the multiply.
The 'A' bit selects the accumulator for the result.
The 'x' bits select the prefetch Wxd destination.
The 'y' bits select the prefetch Wyd destination.
The 'i' bits select the Wx prefetch operation.
The 'j' bits select the Wy prefetch operation.

Note 1:  
The IF bit (CORCON<0>), determines if the multiply is fractional or an integer.

Note 2:  
The US<1:0> bits (CORCON<13:12> in dsPIC33E, CORCON<12> in dsPIC30F/dsPIC33F) determine if the multiply is unsigned, signed, or mixed-sign. Only dsPIC33E devices support mixed-sign multiplication.

Words: 1  
Cycles: 1
Example 1:


; Multiply W4*W5, negate the result and store to ACCA
; Fetch [W8] to W4, Post-increment W8 by 2
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x0001 (integer multiply, no saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>3023</td>
</tr>
<tr>
<td>W5</td>
<td>1290</td>
</tr>
<tr>
<td>W8</td>
<td>0B00</td>
</tr>
<tr>
<td>W10</td>
<td>2000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0000 2387</td>
</tr>
<tr>
<td>Data 0B00</td>
<td>0054</td>
</tr>
<tr>
<td>Data 2000</td>
<td>660A</td>
</tr>
<tr>
<td>CORCON</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:


; Multiply W4*W5, negate the result and store to ACCA
; Fetch [W8] to W4, Post-increment W8 by 2
; Fetch [W10] to W5, Post-increment W10 by 2
; CORCON = 0x0000 (fractional multiply, no saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>3023</td>
</tr>
<tr>
<td>W5</td>
<td>1290</td>
</tr>
<tr>
<td>W8</td>
<td>0B00</td>
</tr>
<tr>
<td>W10</td>
<td>2000</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0000 2387</td>
</tr>
<tr>
<td>Data 0B00</td>
<td>0054</td>
</tr>
<tr>
<td>Data 2000</td>
<td>660A</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MSC

Multiply and Subtract from Accumulator

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:  

\{
\text{label:} \text{MSC} \ Wm \ast Wn, \text{Acc} \ \{,[Wx], \ Wxd\} \ \{,[Wy], Wyd\} \ \{,AWB\}

\{[Wx] + = kx, Wxd\} \ \{,[Wy] + = ky, Wyd\}

\{[Wx] – = kx, Wxd\} \ \{,[Wy] – = ky, Wyd\}

\{[W9 + W12], Wxd\} \ \{,[W11 + W12], Wyd\}

Operands:

\begin{align*}
&Wm \ast Wn \in [W4 \ast W5, W4 \ast W6, W4 \ast W7, W5 \ast W6, W5 \ast W7, W6 \ast W7] \\
&\text{Acc} \in [A,B] \\
&Wx \in [W8, W9]; kx \in [-6, -4, -2, 2, 4, 6]; Wxd \in [W4 \ldots W7] \\
&Wy \in [W10, W11]; ky \in [-6, -4, -2, 2, 4, 6]; Wyd \in [W4 \ldots W7] \\
&\text{AWB} \in [W13, [W13] + = 2]
\end{align*}

Operation:

\begin{align*}
&(\text{Acc}(A \text{ or } B)) \rightarrow (Wm) \ast (Wn) \rightarrow \text{Acc}(A \text{ or } B) \\
&([Wx]) \rightarrow Wxd; (Wx) + kx \rightarrow Wx \\
&([Wy]) \rightarrow Wyd; (Wy) + ky \rightarrow Wy \\
&(\text{Acc}(B \text{ or } A)) \text{ rounded} \rightarrow \text{AWB}
\end{align*}

Status Affected: OA, OB, OAB, SA, SB, SAB

Encoding:

| 1100 | 0mmm | A1xx | yyii | ijjj | jjaa |

Description:

Multiply the contents of two working registers, optionally prefetch operands in preparation for another MAC type instruction and optionally store the unspecified accumulator results. The 32-bit result of the signed multiply is sign-extended to 40 bits and subtracted from the specified accumulator.

Operands Wx, Wxd, Wy and Wyd specify optional prefetch operations which support indirect and register offset addressing as described in Section 4.14.1 “MAC Prefetches”. Operand AWB specifies the optional store of the “other” accumulator as described in Section 4.14.4 “MAC Write Back”.

The ‘m’ bits select the operand registers Wm and Wn for the multiply. The ‘A’ bit selects the accumulator for the result. The ‘x’ bits select the prefetch Wxd destination. The ‘y’ bits select the prefetch Wyd destination. The ‘i’ bits select the Wx prefetch operation. The ‘j’ bits select the Wy prefetch operation. The ‘a’ bits select the accumulator Write Back destination.

Note: The IF bit (CORCON<0>), determines if the multiply is fractional or an integer.

Words: 1

Cycles: 1
Example 1:

```c
; Multiply W6*W7 and subtract the result from ACCA
; Fetch [W8] to W6, Post-decrement W8 by 4
; Fetch [W10] to W7, Post-decrement W10 by 4
; CORCON = 0x0001 (integer multiply, no saturation)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>9051</td>
</tr>
<tr>
<td>W7</td>
<td>7230</td>
</tr>
<tr>
<td>W8</td>
<td>0C00</td>
</tr>
<tr>
<td>W10</td>
<td>1C00</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 0567 8000</td>
</tr>
<tr>
<td>Data 0C00</td>
<td>D309</td>
</tr>
<tr>
<td>Data 1C00</td>
<td>100B</td>
</tr>
<tr>
<td>CORCON</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

```c
MSC W4*W5, B, [W11+W12], W5, W13
; Multiply W4*W5 and subtract the result from ACCB
; Fetch [W11+W12] to W5
; Write Back ACCA to W13
; CORCON = 0x0000 (fractional multiply, no saturation)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>0500</td>
</tr>
<tr>
<td>W5</td>
<td>2000</td>
</tr>
<tr>
<td>W11</td>
<td>1800</td>
</tr>
<tr>
<td>W12</td>
<td>0800</td>
</tr>
<tr>
<td>W13</td>
<td>6233</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 3738 5ED0</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 1000 0000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>3579</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

MUL

Integer Unsigned Multiply f and WREG

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} MUL{.B} f

Operands:

f ∈ [0 ... 8191]

Operation:

For byte operation:

(WREG)<7:0> * (f)<7:0> → W2

For word operation:

(WREG) * (f) → W2:W3

Status Affected:

None

Encoding:

| 1011 | 1100 | 0B0f | ffff | ffff | ffff |

Description:

Multiply the default working register WREG with the specified file register and place the result in the W2:W3 register pair. Both operands and the result are interpreted as unsigned integers. If this instruction is executed in Byte mode, the 16-bit result is stored in W2. In Word mode, the most significant word of the 32-bit result is stored in W3, and the least significant word of the 32-bit result is stored in W2.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). The ‘f’ bits select the address of the file register.

Notes:

1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

3: The IF bit (CORCON<0>), has no effect on this operation.

4: This is the only instruction, which provides for an 8-bit multiply.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: MUL.B 0x800 ; Multiply (0x800)*WREG (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>9823</td>
<td>9823</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>FFFF</td>
<td>13B0</td>
</tr>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>FFFF</td>
<td>FFFF</td>
</tr>
<tr>
<td>Data 0800</td>
<td>Data 0800</td>
</tr>
<tr>
<td>2690</td>
<td>2690</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
**Example 2:**

```
MUL TMR1 ; Multiply (TMR1) * WREG (Word mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td></td>
<td>F001</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>0000</td>
<td>C287</td>
</tr>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>0000</td>
<td>2F5E</td>
</tr>
<tr>
<td>TMR1</td>
<td>TMR1</td>
</tr>
<tr>
<td>3287</td>
<td>3287</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

MUL.SS

Integer 16x16-bit Signed Multiply

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} MUL.SS Wb, Ws, Wnd

- [Ws],
- [Ws++],
- [Ws--],
- [++Ws],
- [--Ws],

Operands:
- Wb ∈ [W0 ... W15]
- Ws ∈ [W0 ... W15]
- Wnd ∈ [W0, W2, W4 ... W12]

Operation: signed (Wb) * signed (Ws) → Wnd:Wnd + 1

Status Affected: None

Encoding: 1011 1001 lwww wddd dppp ssss

Description:
Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The least significant word of the result is stored in Wnd (which must be an even numbered working register), and the most significant word of the result is stored in Wnd + 1. Both source operands and the result Wnd are interpreted as two's complement signed integers. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: This instruction operates in Word mode only.

2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-2 for information on how double words are aligned in memory.

3: Wnd may not be W14, since W15<0> is fixed to zero.

4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1 (1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

© 2005-2011 Microchip Technology Inc.
Example 1: \texttt{MUL.SS W0, W1, W12} \; ; \text{Multiply W0*W1} \\
\hspace{2cm} \text{Store the result to W12:W13}

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 9823</td>
<td>W0 9823</td>
</tr>
<tr>
<td>W1 67DC</td>
<td>W1 67DC</td>
</tr>
<tr>
<td>W12 FFFF</td>
<td>W12 D314</td>
</tr>
<tr>
<td>W13 FFFF</td>
<td>W13 D5DC</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2: \texttt{MUL.SS W2, [--W4], W0} \; ; \text{Pre-decrement W4} \\
\hspace{2cm} \text{Multiply W2*[W4]} \\
\hspace{2cm} \text{Store the result to W0:W1}

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 FFFF</td>
<td>W0 28F8</td>
</tr>
<tr>
<td>W1 FFFF</td>
<td>W1 0000</td>
</tr>
<tr>
<td>W2 0045</td>
<td>W2 0045</td>
</tr>
<tr>
<td>W4 27FE</td>
<td>W4 27FC</td>
</tr>
<tr>
<td>Data 27FC 0098</td>
<td>Data 27FC 0098</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
### MUL.SS

#### Integer 16x16-bit Signed Multiply with Accumulator Destination

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

**Syntax:**
{label:} MUL.SS Wb, Ws, [Ws], A
[Ws++], B
[Ws--],
[++Ws],
[--Ws],

**Operands:**
- Wb ∈ [W0 ... W15]
- Ws ∈ [W0 ... W15]
- ACC ∈ [A, B]

**Operation:**
signed (Wb) * signed (Ws) → ACC(A or B)

**Status Affected:** None

**Encoding:**
1011 1001 lwww w111 Appp ssss

**Description:**
Performs a 16-bit x 16-bit signed multiply with a 32-bit result, which is stored in one of the DSP engine accumulators, ACCA or ACCB. The 32-bit result is sign extended to bit 39 prior to being loaded into the target accumulator.

The source operands are treated as integer or fractional values depending upon the operating mode of the DSP engine (as defined by the IF bit in CORCON<0>). Both source operands are treated as signed values.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the source register.
The 'p' bits select source Address mode 2.
The 'A' bit selects the destination accumulator for the product.

**Note 1:** This instruction operates in Word mode only.
2: The state of the multiplier mode bits (US<1:0> in CORCON) have no effect upon the operation of this instruction.

**Words:** 1
**Cycles:** 1

**Example 1:**

Before Instruction

<table>
<thead>
<tr>
<th></th>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>9823</td>
<td>9823</td>
</tr>
<tr>
<td>W1</td>
<td>67DC</td>
<td>67DC</td>
</tr>
<tr>
<td>Acc A</td>
<td>00 0000 0000</td>
<td>FF D5DC D314</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
MUL.SU  Integer 16x16-bit Signed-Unsigned Short Literal Multiply

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} MUL.SU Wb, #lit5, Wnd

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: signed (Wb) * unsigned lit5 → Wnd:Wnd + 1

Status Affected: None

Encoding: 1011 1001 0www wddd d11k kkkk

Description: Multiply the contents of Wb with the 5-bit literal, and store the 32-bit result in two successive working registers. The least significant word of the result is stored in Wnd (which must be an even numbered working register), and the most significant word of the result is stored in Wnd + 1. The Wb operand and the result Wnd are interpreted as a two's complement signed integer. The literal is interpreted as an unsigned integer. Register direct addressing must be used for Wb and Wnd.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘k’ bits define a 5-bit unsigned integer literal.

Note 1: This instruction operates in Word mode only.

2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-3 for information on how double words are aligned in memory.

3: Wnd may not be W14, since W15<0> is fixed to zero.

4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1

Example 1: MUL.SU W0, #0x1F, W2 ; Multiply W0 by literal 0x1F
            ; Store the result to W2:W3

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 C000</td>
<td>W0 C000</td>
</tr>
<tr>
<td>W2 1234</td>
<td>W2 4000</td>
</tr>
<tr>
<td>W3 C9BA</td>
<td>W3 FFF8</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Example 2:  MUL.SU W2, #0x10, W0  ; Multiply W2 by literal 0x10
             ; Store the result to W0:W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>W0</th>
<th>ABCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>89B3</td>
</tr>
<tr>
<td>W2</td>
<td>F240</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MUL.SU  Integer 16x16-bit Signed-Unsigned Multiply

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: \{label:} MUL.SU Wb, Ws, Wnd

[Ws],

[Ws++],

[Ws--],

[++Ws],

[--Ws],

Operands: Wb ∈ [W0 ... W15]

Ws ∈ [W0 ... W15]

Wnd ∈ [W0, W2, W4 ... W12]

Operation: signed (Wb) * unsigned (Ws) → Wnd:Wnd + 1

Status Affected: None

Encoding: 1011 1001 0www wddd dppp ssss

Description: Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The least significant word of the result is stored in Wnd (which must be an even numbered working register), and the most significant word of the result is stored in Wnd + 1. The Wb operand and the result Wnd are interpreted as a two’s complement signed integer. The Ws operand is interpreted as an unsigned integer. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.

The ‘d’ bits select the address of the lower destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

Note 1: This instruction operates in Word mode only.

2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-3 for information on how double words are aligned in memory.

3: Wnd may not be W14, since W15<0> is fixed to zero.

4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1

Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Note 2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-3 for information on how double words are aligned in memory.

Note 3: Wnd may not be W14, since W15<0> is fixed to zero.

Note 4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.
### Section 5. Instruction Descriptions

**Example 1:** MUL.SU W8, [W9], W0  ; Multiply W8*[W9]
  ; Store the result to W0:W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>68DC</td>
</tr>
<tr>
<td>W1</td>
<td>AA40</td>
</tr>
<tr>
<td>W8</td>
<td>F000</td>
</tr>
<tr>
<td>W9</td>
<td>178C</td>
</tr>
<tr>
<td>Data 178C</td>
<td>F000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:** MUL.SU W2, [++W3], W4  ; Pre-Increment W3
  ; Multiply W2*[W3]
  ; Store the result to W4:W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>0040</td>
</tr>
<tr>
<td>W3</td>
<td>0280</td>
</tr>
<tr>
<td>W4</td>
<td>1819</td>
</tr>
<tr>
<td>W5</td>
<td>2021</td>
</tr>
<tr>
<td>Data 0282</td>
<td>0068</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### MUL.SU

**Integer 16x16-bit Signed-Unsigned Multiply with Accumulator Destination**

Implemented in: PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E
--- | --- | --- | --- | --- | ---
X | X | X | X | X | X

**Syntax:**
```
{label:} MUL.SU Wb, Ws, [Ws], A
[Ws++], B
[Ws--],
[++Ws],
[--Ws],
```

**Operands:**
- Wb $\in [W0 \ldots W15]$
- Ws $\in [W0 \ldots W15]$
- ACC $\in [A, B]$

**Operation:**
Signed (Wb) * unsigned (Ws) $\rightarrow$ ACC(A or B)

**Status Affected:** None

**Encoding:**
```
1011 1001 0www w111 Appp ssss
```

**Description:**
Performs a 16-bit x 16-bit signed multiply with a 32-bit result, which is stored in one of the DSP engine accumulators, ACCA or ACCB. The 32-bit result is sign extended to bit 39 prior to being loaded into the target accumulator.

The source operands are treated as integer or fractional values depending upon the operating mode of the DSP engine (as defined by the IF bit in CORCON<0>). The first source operand is interpreted as a two’s complement signed value and the second source operand is interpreted as an unsigned value.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the source register.
The ‘p’ bits select source Address mode 2.
The ‘A’ bit selects the destination accumulator for the product.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** The state of the multiplier mode bits (US<1:0> in CORCON) have no effect upon the operation of this instruction.

**Words:** 1

**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

---

DS70157F-page 312 © 2005-2011 Microchip Technology Inc.
**Example 1:**  MUL.SU W8, W9, A

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8</td>
<td>F000</td>
</tr>
<tr>
<td>W9</td>
<td>F000</td>
</tr>
<tr>
<td>Acc A</td>
<td>00 0000 0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W8</td>
<td>F000</td>
</tr>
<tr>
<td>W9</td>
<td>F000</td>
</tr>
<tr>
<td>Acc A</td>
<td>FF F100 0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MUL.SU Integer 16x16-bit Signed-Unsigned Short Literal Multiply with Accumulator Destination

Implemented in: PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E

Syntax: {label:} MUL.SU Wb, #lit5, A

Operands: Wb ∈ [W0 ... W15]
lit5 ∈ [0 ... 31]
ACC ∈ [A, B]

Operation: signed (Wb) × unsigned (lit5) → ACC(A or B)
Status Affected: None
Encoding: 1011 1001 wwww w111 A11k kkkk

Description: Performs a 16-bit x 16-bit signed multiply with a 32-bit result, which is stored in one of the DSP engine accumulators, ACCA or ACCB. The 32-bit result is sign extended to bit 39 prior to being loaded into the target accumulator.

The source operands are treated as integer or fractional values depending upon the operating mode of the DSP engine (as defined by the IF bit in CORCON<0>). The first source operand is interpreted as a two’s complement signed value and the second source operand is interpreted as an unsigned value.

The ‘w’ bits select the address of the base register.
The ‘k’ bits select the 5-bit literal value.
The ‘A’ bit selects the destination accumulator for the product.

Note 1: This instruction operates in Word mode only.
2: The state of the multiplier mode bits (US<1:0> in CORCON) have no effect upon the operation of this instruction.

Words: 1
Cycles: 1

Example 1: MUL.SU W8, #0x02, A

Before Instruction

<table>
<thead>
<tr>
<th>W8</th>
<th>Acc A</th>
<th>SR</th>
<th>W8</th>
<th>Acc A</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>042</td>
<td>00000</td>
<td>0000</td>
<td>042</td>
<td>00000</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>W8</th>
<th>Acc A</th>
<th>SR</th>
<th>W8</th>
<th>Acc A</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>042</td>
<td>00000</td>
<td>0000</td>
<td>042</td>
<td>00000</td>
<td>0084</td>
</tr>
</tbody>
</table>

0000
### MUL.US Integer 16x16-bit Unsigned-Signed Multiply

#### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

```plaintext
{label:} MUL.US Wb, Ws, Wnd

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],
```

#### Operands:

- **Wb**: ∈ [W0 ... W15]
- **Ws**: ∈ [W0 ... W15]
- **Wnd**: ∈ [W0, W2, W4 ... W12]

#### Operation:

```
unsigned (Wb) * signed (Ws) → Wnd:Wnd + 1
```

#### Status Affected:

None

#### Encoding:

```
| 1011 | 1000 | lwww | wddd | dppp | ssss |
```

#### Description:

Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The least significant word of the result is stored in Wnd (which must be an even numbered working register), and the most significant word of the result is stored in Wnd + 1.

The Wb operand is interpreted as an unsigned integer. The Ws operand and the result Wnd are interpreted as a two's complement signed integer. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'p' bits select the source Address mode.
The 's' bits select the source register.

#### Note 1:

This instruction operates in Word mode only.

2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-3 for information on how double words are aligned in memory.

3: Wnd may not be W14, since W15<0> is fixed to zero.

4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

#### Words:

1

#### Cycles:

1

#### Note 1:

In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”. 

---

© 2005-2011 Microchip Technology Inc.  DS70157F-page 315
Example 1:  
MUL.US  W0, [W1], W2  ; Multiply W0*[W1] (unsigned-signed)  
; Store the result to W2:W3

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>C000</td>
</tr>
<tr>
<td>W1</td>
<td>2300</td>
</tr>
<tr>
<td>W2</td>
<td>00DA</td>
</tr>
<tr>
<td>W3</td>
<td>CC25</td>
</tr>
<tr>
<td>Data 2300</td>
<td>F000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W0</td>
<td>C000</td>
</tr>
<tr>
<td>W1</td>
<td>2300</td>
</tr>
<tr>
<td>W2</td>
<td>0000</td>
</tr>
<tr>
<td>W3</td>
<td>F400</td>
</tr>
<tr>
<td>Data 2300</td>
<td>F000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:  
MUL.US  W6, [W5++], W10  ; Mult. W6*[W5] (unsigned-signed)  
; Store the result to W10:W11  
; Post-Increment W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>0C00</td>
</tr>
<tr>
<td>W6</td>
<td>FFFF</td>
</tr>
<tr>
<td>W10</td>
<td>0908</td>
</tr>
<tr>
<td>W11</td>
<td>6EEB</td>
</tr>
<tr>
<td>Data 0000</td>
<td>7FFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W5</td>
<td>0C02</td>
</tr>
<tr>
<td>W6</td>
<td>FFFF</td>
</tr>
<tr>
<td>W10</td>
<td>8001</td>
</tr>
<tr>
<td>W11</td>
<td>7FFE</td>
</tr>
<tr>
<td>Data 0000</td>
<td>7FFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
**MUL.US**

**Integer 16x16-bit Unsigned-Signed Multiply with Accumulator Destination**

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

Syntax: {label:} MUL.US Wb, Ws, A

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
ACC ∈ [A, B]

Operation: unsigned (Wb) * signed (Ws) → ACC(A or B)

Status Affected: None

Encoding: 1011 1000 0www w111 Appp ssss

Description: Performs a 16-bit x 16-bit signed multiply with a 32-bit result, which is stored in one of the DSP engine accumulators, ACCA or ACCB. The 32-bit result is sign extended to bit 39 prior to being loaded into the target accumulator.

The source operands are treated as integer or fractional values depending upon the operating mode of the DSP engine (as defined by the IF bit in CORCON<0>). The first source operand is interpreted as an unsigned value and the second source operand is interpreted as a two's complement signed value.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the source register.
The ‘p’ bits select source Address mode 2.
The ‘A’ bit selects the destination accumulator for the product.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** The state of the multiplier mode bits (US<1:0> in CORCON) have no effect upon the operation of this instruction.

Words: 1

Cycles: 1(t)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.
Example 1: MUL.US W0, W1, B

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W1</td>
<td>W1</td>
</tr>
<tr>
<td>Acc B</td>
<td>Acc B</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>C000</td>
<td>0000</td>
</tr>
<tr>
<td>F000</td>
<td>F000</td>
</tr>
<tr>
<td>00 0000 0000</td>
<td>FF F400 0000</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

MUL.UU Integer 16x16-bit Unsigned Short Literal Multiply

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} MUL.UU Wb, #lit5, Wnd

Operands:
- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * unsigned lit5 → Wnd:Wnd + 1

Status Affected: None

Encoding:

| 1011 | 1000 | 0www | wddd | d11k | kkkk |

Description:
Multiply the contents of Wb with the 5-bit literal, and store the 32-bit result in two successive working registers. The least significant word of the result is stored in Wnd (which must be an even numbered working register), and the most significant word of the result is stored in Wnd + 1. Both operands and the result are interpreted as unsigned integers. Register direct addressing must be used for Wb and Wnd.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘k’ bits define a 5-bit unsigned integer literal.

Note 1: This instruction operates in Word mode only.
Note 2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-3 for information on how double words are aligned in memory.
Note 3: Wnd may not be W14, since W15<0> is fixed to zero.
Note 4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1

Example 1:

MUL.UU W0, #0xF, W12 ; Multiply W0 by literal 0xF ; Store the result to W12:W13

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>2323</td>
</tr>
<tr>
<td>W12</td>
<td>4512</td>
</tr>
<tr>
<td>W13</td>
<td>7821</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

MUL.UU W7, #0x1F, W0 ; Multiply W7 by literal 0x1F ; Store the result to W0:W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>780B</td>
</tr>
<tr>
<td>W1</td>
<td>3805</td>
</tr>
<tr>
<td>W7</td>
<td>F240</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MUL.UU

Integer 16x16 Unsigned Multiply

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} MUL.UU Wb, Ws, Wnd

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * unsigned (Ws) → Wnd: Wnd + 1

Status Affected: None

Encoding:

| 1011 | 1000 | 0www | wddd | dppp | ssss |

Description: Multiply the contents of Wb with the contents of Ws, and store the 32-bit result in two successive working registers. The least significant word of the result is stored in Wnd (which must be an even numbered working register), and the most significant word of the result is stored in Wnd + 1. Both source operands and the result are interpreted as unsigned integers. Register direct addressing must be used for Wb and Wnd. Register direct or indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: This instruction operates in Word mode only.
2: Since the product of the multiplication is 32 bits, Wnd must be an even working register. See Figure 4-3 for information on how double words are aligned in memory.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
### Example 1:

MUL.UU W4, W0, W2 ; Multiply W4*W0 (unsigned-unsigned)
; Store the result to W2:W3

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>FFFF</td>
</tr>
<tr>
<td>W2</td>
<td>2300</td>
</tr>
<tr>
<td>W3</td>
<td>00DA</td>
</tr>
<tr>
<td>W4</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

### Example 2:

MUL.UU W0, [W1++], W4 ; Mult. W0*[W1] (unsigned-unsigned)
; Store the result to W4:W5
; Post-Increment W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>1024</td>
</tr>
<tr>
<td>W1</td>
<td>2300</td>
</tr>
<tr>
<td>W4</td>
<td>9654</td>
</tr>
<tr>
<td>W5</td>
<td>BDBC</td>
</tr>
<tr>
<td>Data 2300</td>
<td>D625</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### MUL.UU

**Integer 16x16-bit Unsigned Multiply with Accumulator Destination**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} MUL.UU Wb, Ws, A

[Ws], B

[Ws++],

[Ws--],

[++]Ws,

[--Ws]

**Operands:**

Wb ∈ [W0 ... W15]

Ws ∈ [W0 ... W15]

ACC ∈ [A, B]

**Operation:**

unsigned (Wb) * unsigned (Ws) \(\rightarrow\) ACC(A or B)

**Status Affected:** None

**Encoding:**

| 1011 | 1000 | 0www | w111 | Appp | ssss |

**Description:** Performs a 16-bit x 16-bit unsigned multiply with a 32-bit result, which is stored in one of the DSP engine accumulators, ACCA or ACCB. The 32-bit result is zero extended to bit 39 prior to being loaded into the target accumulator.

The source operands are treated as integer or fractional values depending upon the operating mode of the DSP engine (as defined by the IF bit in CORCON<0>). Both source operands are treated as unsigned values.

The ‘w’ bits select the address of the base register.

The ‘d’ bits select the address of the source register.

The ‘p’ bits select source Address mode 2.

The ‘A’ bit selects the destination accumulator for the product.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** The state of the multiplier mode bits (US<1:0> in CORCON) have no effect upon the operation of this instruction.

**Words:** 1

**Cycles:** 1(1)

**Example 1:** MUL.UU W4, W0, B

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>Acc B</td>
<td>FF FFFE 0001</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.
### MUL.UU

**Integer 16x16-bit Unsigned Short Literal Multiply with Accumulator Destination**

Implemented in: PIC24F  
PIC24H  
PIC24E  
dsPIC30F  
dsPIC33F  
dsPIC33E

Syntax:  
{label:} MUL.UU Wb, #lit5, A

Operands:  
Wb ∈ [W0 ... W15]  
lit5 ∈ [0 ... 31]  
ACC ∈ [A, B]

Operation:  
unsigned (Wb) * unsigned (lit5) → ACC (A or B)

Status Affected: None

Encoding:  
1011 1000 0www w111 A11k kkkk

Description: Performed a 16-bit x 16-bit signed multiply with a 32-bit result, which is stored in one of the DSP engine accumulators, ACCA or ACCB. The 32-bit result is zero extended to bit 39 prior to being loaded into the target accumulator.

The source operands are treated as integer or fractional values depending upon the operating mode of the DSP engine (as defined by the IF bit in CORCON<0>). Both source operands are treated as unsigned values.

The ‘w’ bits select the address of the base register.  
The ‘k’ bits select the 5-bit literal.  
The ‘A’ bit selects the destination accumulator for the product.

**Note 1:** This instruction operates in Word mode only.
**Note 2:** The state of the multiplier mode bits (US<1:0> in CORCON) have no effect upon the operation of this instruction.

| Words: | 1 |
| Cycles: | 1 |

**Example 1:** MUL.UU W8, #0x02, A

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8</td>
<td>0042</td>
</tr>
<tr>
<td>Acc A</td>
<td>00 0000 0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W8</td>
<td>0042</td>
</tr>
<tr>
<td>Acc A</td>
<td>00 0000 0084</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MULW.SS  Integer 16x16-bit Signed Multiply with 16-bit Result

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:

\[
\{\text{label:}\} \; \text{MULW.SS} \; \text{Wb}, \; \text{Ws}, \; \text{Wnd}
\]

[Ws],
[Ws++],
[Ws--],
[++Ws],
[--Ws],

Operands:

- \(\text{Wb} \in [\text{W0 ... W15}]\)
- \(\text{Ws} \in [\text{W0 ... W15}]\)
- \(\text{Wnd} \in [\text{W0, W2, W4 ... W12}]\)

Operation:

\[
\text{signed (Wb)} \times \text{signed (Ws)} \rightarrow \text{Wnd}
\]

Status Affected:

None

Encoding:

| 1011 | 1001 | lwww | wddd | dppp | ssss |

Description:

Multiply the contents of Wb with the contents of Ws, and store the result in a working register, which must be an even numbered working register. Both source operands and the result Wnd are interpreted as two's complement signed integers. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: This instruction operates in Word mode only.
2: Wnd must be an even working register.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

MULW.SS W0, W1, W12 ; Multiply W0*W1
; Store the result to W12

Before Instruction

<table>
<thead>
<tr>
<th>W0</th>
<th>W1</th>
<th>W12</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>9823</td>
<td>67DC</td>
<td>FFFF</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>W0</th>
<th>W1</th>
<th>W12</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>9823</td>
<td>67DC</td>
<td>D314</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:  MULW.SS W2, [--W4], W0  ; Pre-decrement W4
             ; Multiply W2*[W4]
             ; Store the result to W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>W4</td>
<td>W4</td>
</tr>
<tr>
<td>Data 27FC</td>
<td>Data 27FC</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>FFFF</td>
<td>28F8</td>
</tr>
<tr>
<td>0045</td>
<td>0045</td>
</tr>
<tr>
<td>27FE</td>
<td>27FC</td>
</tr>
<tr>
<td>0098</td>
<td>0098</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
# MULW.SU

**Integer 16x16-bit Signed-Unsigned Multiply with 16-bit Result**

- **Implemented in:**
  - PIC24F
  - PIC24H
  - PIC24E
  - dsPIC30F
  - dsPIC33F
  - dsPIC33E

- **Syntax:**
  ```
  {label:} MUL.SU Wb, Ws, Wnd
  [Ws],
  [Ws++],
  [Ws--],
  [++;Ws],
  [--;Ws],
  ```

- **Operands:**
  - `Wb` ∈ `[W0 ... W15]`
  - `Ws` ∈ `[W0 ... W15]`
  - `Wnd` ∈ `[W0, W2, W4 ... W12]`

- **Operation:**
  - `signed (Wb) * unsigned (Ws) → Wnd`

- **Status Affected:** None

- **Encoding:**
  - `1011 1001 0www wddd dppp ssss`

- **Description:**
  Multiply the contents of `Wb` with the contents of `Ws`, and store the result in a working register, which must be an even numbered working register. The `Wb` operand and the result `Wnd` are interpreted as a two's complement signed integer. The `Ws` operand is interpreted as an unsigned integer. Register direct addressing must be used for `Wb` and `Wnd`. Register direct or register indirect addressing may be used for `Ws`.

  The 'w' bits select the address of the base register.
  The 'd' bits select the address of the lower destination register.
  The 'p' bits select the source Address mode.
  The 's' bits select the source register.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** `Wnd` must be an even working register.

**Note 3:** `Wnd` may not be W14, since W15<0> is fixed to zero.

**Note 4:** The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

- **Words:** 1
- **Cycles:** 1<sup>(1)</sup>

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**
```
MULW.SU W8, [W9], W0 ; Multiply W8*{W9}
; Store the result to W0
```
**Example 2:** MULW.SU W2, [+W3], W4 ; Pre-Increment W3
; Multiply W2*W3
; Store the result to W4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 0040</td>
<td>W2 0040</td>
</tr>
<tr>
<td>W3 0280</td>
<td>W3 0282</td>
</tr>
<tr>
<td>W4 1819</td>
<td>W4 1A00</td>
</tr>
<tr>
<td>Data 0282 0068</td>
<td>Data 0282 0068</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
MULW.SU Integer 16x16-bit Signed-Unsigned Short Literal Multiply with 16-bit Result

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

Syntax: {label;} MULW.SU Wb, #lit5, Wnd

Operands: 
- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wnd ∈ [W0, W2, W4 ... W12]

Operation: signed (Wb) * unsigned (lit5) → Wnd

Status Affected: None

Encoding:

<table>
<thead>
<tr>
<th></th>
<th>1011</th>
<th>1001</th>
<th>0www</th>
<th>wddd</th>
<th>d11k</th>
<th>kkkk</th>
</tr>
</thead>
</table>

Description: Multiply the contents of Wb with a 5-bit literal value, and store the result in a working register, which must be an even numbered working register. The Wb operand and the result Wnd are interpreted as a two’s complement signed integer. Register direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the address of the lower destination register.
The 'k' bits select the 5-bit literal value.

Note 1: This instruction operates in Word mode only.
2: Wnd must be an even working register.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1

Example 1: MULW.SU W8, #0x04, W0 ; Multiply W8 * #0x04
; Store the result to W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 68DC</td>
<td>W0 4000</td>
</tr>
<tr>
<td>W8 1000</td>
<td>W8 1000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
MULW.US  Integer 16x16-bit Unsigned-Signed Multiply with 16-bit Result

Implemented in:  
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:  
{label:} MULW.US Wb, Ws, Wnd

Operands:  
Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W12]

Operation:  
unsigned (Wb) * signed (Ws) → Wnd

Status Affected:  
None

Encoding:  

| 1011 | 1000 | lwww | wddd | dppp | ssss |

Description:  
Multiply the contents of Wb with the contents of Ws, and store the result in a working register, which must be an even numbered working register. The Wb operand is interpreted as an unsigned integer. The Ws operand and the result Wnd are interpreted as a two’s complement signed integer. Register direct addressing must be used for Wb and Wnd. Register direct or register indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1:  
This instruction operates in Word mode only.

2:  
Wnd must be an even working register.

3:  
Wnd may not be W14, since W15<0> is fixed to zero.

4:  
The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Example 1:  
MULW.US W0, [W1], W2 ; Multiply W0*[W1] (unsigned-signed) ; Store the result to W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>C000</td>
</tr>
<tr>
<td>W1</td>
<td>2300</td>
</tr>
<tr>
<td>W2</td>
<td>00DA</td>
</tr>
<tr>
<td>Data 2300</td>
<td>F000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Note 1:  
In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Note 1:  
In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Words:  
1

Cycles:  
1(f)
Example 2: \[\text{MULW.US W6, [W5++], W10}\] ; Mult. W6*W5 (unsigned-signed) ; Store the result to W10 ; Post-Increment W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>W5</td>
</tr>
<tr>
<td>W6</td>
<td>W6</td>
</tr>
<tr>
<td>W10</td>
<td>W10</td>
</tr>
<tr>
<td>Data 0C00</td>
<td>Data 0C00</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>

Before Instruction:
- W5: 0C00
- W6: FFFF
- W10: 0908
- Data 0C00: 7FFF
- SR: 0000

After Instruction:
- W5: 0C02
- W6: FFFF
- W10: 8001
- Data 0C00: 7FFF
- SR: 0000
Section 5. Instruction Descriptions

MULW.UU  Integer 16x16-bit Unsigned Multiply with 16-bit Result

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
--- | --- | --- | --- | --- | --- | --- |
| X | X | X | X | X | X | X |

Syntax: `{label:} MULW.UU Wb, Ws, Wnd

[Ws],
[Ws++],
[Ws--],
[+Ws],
[--Ws],

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * unsigned (Ws) → Wnd

Status Affected: None

Encoding: 1011 1000 0www wddd dppp ssss

Description: Multiply the contents of Wb with the contents of Ws, and store the result in a working registers, which must be an even numbered working register. Both source operands and the result are interpreted as unsigned integers. Register direct addressing must be used for Wb and Wnd. Register direct or indirect addressing may be used for Ws.

The ‘w’ bits select the address of the base register.
The ‘d’ bits select the address of the lower destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: This instruction operates in Word mode only.
2: Wnd must be an even working register.
3: Wnd may not be W14, since W15<0> is fixed to zero.
4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1
Cycles: 1 (1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: MULW.UU W4, W0, W2 ; Multiply W4*W0 {unsigned-unsigned}
; Store the result to W2

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Instruction</td>
</tr>
<tr>
<td>W0</td>
<td>FFFF</td>
</tr>
<tr>
<td>W2</td>
<td>2300</td>
</tr>
<tr>
<td>W4</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
MULW.UU Integer 16x16-bit Unsigned Short Literal Multiply with 16-bit Result

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} MULW.UU Wb, #lit5, Wnd

Operands:
- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wnd ∈ [W0, W2, W4 ... W12]

Operation: unsigned (Wb) * unsigned → Wnd

Status Affected: None

Encoding:

| 1011 | 1000 | 0www | wddd | d11k | kkkk |

Description: Multiply the contents of Wb with a 5-bit literal value, and store the result in a working register, which must be an even numbered working register. Both source operands and the result are interpreted as unsigned integers. Register direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.

The 'd' bits select the address of the lower destination register.

The 'k' bits select the 5-bit literal value.

Note 1: This instruction operates in Word mode only.

2: Wnd must be an even working register.

3: Wnd may not be W14, since W15<0> is fixed to zero.

4: The IF bit and the US<1:0> bits in the CORCON register have no effect on this operation.

Words: 1

Cycles: 1

Example 1: MULW.UU W4, #0x04, W2 ; Multiply W4*W0 (unsigned unsigned) ; Store the result to W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>2300</td>
</tr>
<tr>
<td>W4</td>
<td>1000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
NEG

Negate f

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} NEG{.B} f {,WREG}

Operands:

f ∈ [0 ... 8191]

Operation:

(f) + 1 → destination designated by D

Status Affected:

DC, N, OV, Z, C

Encoding:

| 1110 | 1110 | 0BDF | 0000 | 0000 | 0000 | 0000 |

Description:

Compute the two's complement of the contents of the file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

NEG.B 0x880, WREG ; Negate (0x880) (Byte mode)
; Store result to WREG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>9080</td>
<td>90AB</td>
</tr>
<tr>
<td>Data 0880</td>
<td>Data 0880</td>
</tr>
<tr>
<td>2355</td>
<td>2355</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008</td>
</tr>
<tr>
<td></td>
<td>(N = 1)</td>
</tr>
</tbody>
</table>

Example 2:

NEG 0x1200 ; Negate (0x1200) (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1200</td>
<td>Data 1200</td>
</tr>
<tr>
<td>8923</td>
<td>76DD</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

NEG

Negate Ws

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} NEG{.B} Ws, Wd [Ws], [Wd]
NEG

Negate Ws

Operands:

Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation:

(Ws) + 1 → Wd

Status Affected:

DC, N, OV, Z, C

Encoding:

| 1110 | 1010 | 0Bqq | qddd | dpdp | ssss |

Description:

Compute the two’s complement of the contents of the source register Ws and place the result in the destination register Wd. Either register direct or indirect addressing may be used for both Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1

Example 1: NEG.B W3, [W4++] ; Negate W3 and store to [W4] (Byte mode) ; Post-increment W4

Before Instruction

| W3 | 7839 |
| W4 | 1005 |
| Data 1004 | 2355 |
| SR | 0000 |

After Instruction

| W3 | 7839 |
| W4 | 1006 |
| Data 1004 | C755 |
| SR | 0008 (N = 1) |

Example 2: NEG [W2++], [--W4] ; Pre-decrement W4 (Word mode)

Before Instruction

| W2 | 0900 |
| W4 | 1002 |
| Data 0900 | 870F |
| Data 1000 | 5105 |
| SR | 0000 |

After Instruction

| W2 | 0902 |
| W4 | 1000 |
| Data 0900 | 870F |
| Data 1000 | 78F1 |
| SR | 0000 |
Section 5. Instruction Descriptions

### NEG

**Negate Accumulator**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} NEG Acc

**Operands:**

Acc ∈ [A,B]

**Operation:**

If (Acc = A):

- ACCA → ACCA

Else:

- ACCB → ACCB

**Status Affected:** OA, OB, OAB, SA, SB, SAB

**Encoding:**

<table>
<thead>
<tr>
<th></th>
<th>1100</th>
<th>1011</th>
<th>A001</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
</tr>
</thead>
</table>

**Description:**

Compute the two’s complement of the contents of the specified accumulator. Regardless of the Saturation mode, this instruction operates on all 40 bits of the accumulator.

The 'A' bit specifies the selected accumulator.

**Words:** 1

**Cycles:** 1

---

**Example 1:**

NEG A ; Negate ACCA
; Store result to ACCA
; CORCON = 0x0000 (no saturation)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCA</td>
<td>003290</td>
<td>59C8</td>
<td>ACCA</td>
<td>FFCD6F</td>
<td>A638</td>
<td></td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
<td></td>
<td>CORCON</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td></td>
<td>SR</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example 2:**

NEG B ; Negate ACCB
; Store result to ACCB
; CORCON = 0x00C0 (normal saturation)

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCB</td>
<td>FFF230</td>
<td>10DC</td>
<td>ACCB</td>
<td>000DCF</td>
<td>EF24</td>
<td></td>
</tr>
<tr>
<td>CORCON</td>
<td>00C0</td>
<td></td>
<td>CORCON</td>
<td>00C0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td></td>
<td>SR</td>
<td>0000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**NOP**

**No Operation**

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
     | X     | X     | X     | X     | X     | X     |

Syntax: \( \{\text{label:} \) NOP

Operands: None
Operation: No Operation
Status Affected: None
Encoding: 

\[
\begin{array}{ccccccc}
0000 & 0000 & xxxx & xxxx & xxxx & xxxx & xxxx \\
\end{array}
\]

Description: No Operation is performed.

The ‘x’ bits can take any value.

Words: 1
Cycles: 1

Example 1: NOP ; execute no operation

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 1092</td>
<td>PC 00 1094</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2: NOP ; execute no operation

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 08AE</td>
<td>PC 00 08B0</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**NOPR**

**No Operation**

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
     | X     | X     | X     | X     | X     | X     |

Syntax: \( \{\text{label:} \) NOPR

Operands: None
Operation: No Operation
Status Affected: None
Encoding: 

\[
\begin{array}{ccccccc}
1111 & 1111 & xxxx & xxxx & xxxx & xxxx & xxxx \\
\end{array}
\]

Description: No Operation is performed.

The ‘x’ bits can take any value.

Words: 1
Cycles: 1
### Section 5. Instruction Descriptions

**Example 1:** NOPR ; execute no operation

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 2430</td>
<td>PC 00 2432</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:** NOPR ; execute no operation

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 1466</td>
<td>PC 00 1468</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**POP**

**Pop TOS to f**

- **Implemented in:**
  - PIC24F
  - PIC24H
  - PIC24E
  - dsPIC30F
  - dsPIC33F
  - dsPIC33E

- **Syntax:** `{label:} POP f

- **Operands:**
  - f ∈ [0 ... 65534]

- **Operation:**
  - (W15) – 2 → W15
  - (TOS) → f

- **Status Affected:** None

- **Encoding:**
  - 1111 1001 ffff ffff ffff fff0

- **Description:**
  - The Stack Pointer (W15) is pre-decremented by 2 and the Top-of-Stack (TOS) word is written to the specified file register, which may reside anywhere in the lower 32K words of data memory.

  - The 'f' bits select the address of the file register.

- **Note 1:** This instruction operates in Word mode only.

- **Note 2:** The file register address must be word-aligned.

- **Words:** 1

- **Cycles:** 1

**Example 1:** POP 0x1230 ; Pop TOS to 0x1230

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15 1006</td>
<td>W15 1004</td>
</tr>
<tr>
<td>Data 1004</td>
<td>Data 1004</td>
</tr>
<tr>
<td>Data 1230</td>
<td>Data 1230</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Example 2:  

```
POP 0x880    ; Pop TOS to 0x880
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15</td>
<td>2000</td>
</tr>
<tr>
<td>Data 0880</td>
<td>E3E1</td>
</tr>
<tr>
<td>Data 1FFE</td>
<td>A090</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**POP**  
Pop TOS to Wd

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{\(label:\)} POP \(Wd\)  
\[Wd\]  
\[Wd++\]  
\[Wd--\]  
\[--Wd\]  
\[++Wd\]  
\[Wd+Wb\]

Operands:  
\(Wd \in [W0 ... W15]\)  
\(Wb \in [W0 ... W15]\)

Operation:  
\((W15) - 2 \rightarrow W15\)  
\((TOS) \rightarrow Wd\)

Status Affected:  
None

Encoding:  
```
0111 lwww w0hh hddd d100 1111
```

Description:  
The Stack Pointer (W15) is pre-decremented by 2 and the Top-of-Stack (TOS) word is written to \(Wd\). Either register direct or indirect addressing may be used for \(Wd\).

The ‘\(w\)’ bits define the offset register \(Wb\).  
The ‘\(h\)’ bits select the destination Address mode.  
The ‘\(d\)’ bits select the destination register.

**Note 1:**  
This instruction operates in Word mode only.

**Note 2:**  
This instruction is a specific version of the "MOV \(Ns, Wd\)" instruction (\(MOV [-W15], Wd\)). It reverse assembles as MOV.

Words:  
1

Cycles:  
1

Example 1:  

```
POP W4    ; Pop TOS to W4
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>EDA8</td>
</tr>
<tr>
<td>W15</td>
<td>1008</td>
</tr>
<tr>
<td>Data 1006</td>
<td>C45A</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:  

```
POP [++W10] ; Pre-increment W10
; Pop TOS to [W10]
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W10 0E02</td>
<td>W10 0E04</td>
</tr>
<tr>
<td>W15 1766</td>
<td>W15 1764</td>
</tr>
<tr>
<td>Data 0E04 E3E1</td>
<td>Data 0E04 C7B5</td>
</tr>
<tr>
<td>Data 1764 C7B5</td>
<td>Data 1764 C7B5</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**POP.D**  
Double Pop TOS to Wnd:Wnd+1

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

```
{label:} POP.D Wnd
```

Operands:  

Wnd ∈ [W0, W2, W4, ... W14]

Operation:  

- (W15) – 2 → W15
- (TOS) → Wnd + 1
- (W15) – 2 → W15
- (TOS) → Wnd

Status Affected:  

None

Encoding:

```
1011 1110 0000 0ddd 0100 1111
```

Description:

A double word is POPped from the Top-of-Stack (TOS) and stored to Wnd:Wnd + 1. The most significant word is stored to Wnd + 1, and the least significant word is stored to Wnd. Since a double word is POPped, the Stack Pointer (W15) gets decremented by 4.

The ‘d’ bits select the address of the destination register pair.

**Note 1:** This instruction operates on double words. See Figure 4-3 for information on how double words are aligned in memory.

**2:** Wnd must be an even working register.

**3:** This instruction is a specific version of the "MOV.D Ws, Wnd" instruction (MOV.D [--W15], Wnd). It reverse assembles as MOV.D.

Words:   1

Cycles:  2

Example 1:  

```
POP.D W6        ; Double pop TOS to W6
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6 07BB</td>
<td>W6 3210</td>
</tr>
<tr>
<td>W7 89AE</td>
<td>W7 7654</td>
</tr>
<tr>
<td>W15 0850</td>
<td>W15 084C</td>
</tr>
<tr>
<td>Data 084C 3210</td>
<td>Data 084C 3210</td>
</tr>
<tr>
<td>Data 084E 7654</td>
<td>Data 084E 7654</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Example 2: POP.D W0        ; Double pop TOS to W0

Before Instruction | After Instruction
---|---
W0 | 673E | W0 | 791C
W1 | DD23 | W1 | D400
W15 | 0BBC | W15 | 0BB8
Data 0BB8 | 791C | Data 0BB8 | 791C
Data 0BBA | D400 | Data 0BBA | D400
SR | 0000 | SR | 0000

POP.S  Pop Shadow Registers

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E
X X X X X X

Syntax: {label:} POP.S

Operands: None

Operation: POP shadow registers

Status Affected: DC, N, OV, Z, C

Encoding: 1111 1110 1000 0000 0000 0000

Description: The values in the shadow registers are copied into their respective primary registers. The following registers are affected: W0-W3, and the C, Z, OV, N and DC STATUS register flags.

Note 1: The shadow registers are not directly accessible. They may only be accessed with PUSH.S and POP.S.

2: The shadow registers are only one-level deep.

Words: 1
Cycles: 1

Example 1: POP.S     ; Pop the shadow registers
; (See PUSH.S Example 1 for contents of shadows)

Before Instruction | After Instruction
---|---
W0 | 07BB | W0 | 0000
W1 | 03FD | W1 | 1000
W2 | 9610 | W2 | 2000
W3 | 7249 | W3 | 3000
SR | 00E0 | SR | 00E1 (IPL = 7, C = 1)

Note: After instruction execution, contents of shadow registers are NOT modified.
### PUSH

**Push f to TOS**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\[
\{\text{label:}\}\quad \text{PUSH} \quad f
\]

**Operands:**

\[f \in [0 \ldots 65534]\]

**Operation:**

\[(f) \rightarrow \text{(TOS)}\]

\[(W15) + 2 \rightarrow W15\]

**Status Affected:** None

**Encoding:**

\[
\begin{array}{cccccccc}
1111 & 1000 & \text{ffff} & \text{ffff} & \text{ffff} & \text{fff0} \\
\end{array}
\]

**Description:**

The contents of the specified file register are written to the Top-of-Stack (TOS) location and then the Stack Pointer (W15) is incremented by 2. The file register may reside anywhere in the lower 32K words of data memory.

The 'f' bits select the address of the file register.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** The file register address must be word-aligned.

**Words:** 1

**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

PUSH \(0x2004\) ; Push \((0x2004)\) to TOS

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15 0B00</td>
<td>W15 0B02</td>
</tr>
<tr>
<td>Data 0B00 791C</td>
<td>Data 0B00 D400</td>
</tr>
<tr>
<td>Data 2004 D400</td>
<td>Data 2004 D400</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

PUSH \(0xC0E\) ; Push \((0xC0E)\) to TOS

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15 0920</td>
<td>W15 0922</td>
</tr>
<tr>
<td>Data 0920 0000</td>
<td>Data 0920 67AA</td>
</tr>
<tr>
<td>Data 0C0E 67AA</td>
<td>Data 0C0E 67AA</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
**PUSH**  
*Push Ws to TOS*

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} PUSH Ws

[Ws]

[Ws++]

[Ws--]

[--Ws]

[++Ws]

[Ws+Wb]
```

**Operands:**

- $Ws \in [W0 \ldots W15]$
- $Wb \in [W0 \ldots W15]$

**Operation:**

- $(Ws) \rightarrow (TOS)$
- $(W15) + 2 \rightarrow W15$

**Status Affected:** None

**Encoding:**

| 0111 | lwww | w001 | l111 | lggg | ssss |

**Description:**

The contents of $Ws$ are written to the Top-of-Stack (TOS) location and then the Stack Pointer ($W15$) is incremented by 2.

The 'w' bits define the offset register $Wb$.

The 'g' bits select the source Address mode.

The 's' bits select the source register.

**Note 1:**

This instruction operates in Word mode only.

**Note 2:**

This instruction is a specific version of the "MOV $Ws$, $Wd$" instruction ($MOV Ws$, [W15++]). It reverse assembles as MOV.

**Words:** 1

**Cycles:** 1

---

**Example 1:**

```
PUSH W2         ; Push W2 to TOS
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 6889</td>
<td>W2 6889</td>
</tr>
<tr>
<td>W15 1566</td>
<td>W15 1568</td>
</tr>
<tr>
<td>Data 1566</td>
<td>Data 1566 6889</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

---

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.
Example 2:  
**PUSH** [W5+W10]  ; Push [W5+W10] to TOS

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5 1200</td>
<td>W5 1200</td>
</tr>
<tr>
<td>W10 0044</td>
<td>W10 0044</td>
</tr>
<tr>
<td>W15 0806</td>
<td>W15 0806</td>
</tr>
<tr>
<td>Data 0806 216F</td>
<td>Data 0806 B20A</td>
</tr>
<tr>
<td>Data 1244 B20A</td>
<td>Data 1244 B20A</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**PUSH.D**  
Double Push Wns:Wns+1 to TOS

<table>
<thead>
<tr>
<th>Implemented in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIC24F</td>
</tr>
<tr>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**  
(label:) PUSH.D Wns

**Operands:**  
Wns ∈ [W0, W2, W4 ... W14]

**Operation:**  
(Wns) → (TOS)  
(W15) + 2 → W15  
(Wns + 1) → (TOS)  
(W15) + 2 → W15

**Status Affected:**  
None

**Encoding:**  
<table>
<thead>
<tr>
<th>1011</th>
<th>1110</th>
<th>1001</th>
<th>1111</th>
<th>1000</th>
<th>sss0</th>
</tr>
</thead>
</table>

**Description:**  
A double word (Wns:Wns + 1) is PUSHed to the Top-of-Stack (TOS). The least significant word (Wns) is PUSHed to the TOS first, and the most significant word (Wns + 1) is PUSHed to the TOS last. Since a double word is PUSHed, the Stack Pointer (W15) gets incremented by 4. The 's' bits select the address of the source register pair.

**Note 1:**  
This instruction operates on double words. See Figure 4-3 for information on how double words are aligned in memory.

**2:**  
Wns must be an even working register.

**3:**  
This instruction is a specific version of the “MOV.D Wns, Wd” instruction (MOV.D Wns, [W15++]). It reverse assembles as MOV.D.

**Words:**  
1

**Cycles:**  
2

Example 1:  
**PUSH.D W6**  ; Push W6:W7 to TOS

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6 C451</td>
<td>W6 C451</td>
</tr>
<tr>
<td>W7 3380</td>
<td>W7 3380</td>
</tr>
<tr>
<td>W15 1240</td>
<td>W15 1244</td>
</tr>
<tr>
<td>Data 1240 0044</td>
<td>Data 1240 C451</td>
</tr>
<tr>
<td>Data 1242 0891</td>
<td>Data 1242 3380</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Example 2: \texttt{PUSH.D W10} ; Push W10:W11 to TOS

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W10 80D3</td>
<td>W10 80D3</td>
</tr>
<tr>
<td>W11 4550</td>
<td>W11 4550</td>
</tr>
<tr>
<td>W15 0C08</td>
<td>W15 0C0C</td>
</tr>
<tr>
<td>Data 0C08 79B5</td>
<td>Data 0C08 80D3</td>
</tr>
<tr>
<td>Data 0C0A 008E</td>
<td>Data 0C0A 4550</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

PUSH.S  Push Shadow Registers

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} PUSH.S

Operands: None

Operation: PUSH shadow registers

Status Affected: None

Encoding:

| 1111 | 1110 | 1010 | 0000 | 0000 | 0000 |

Description: The contents of the primary registers are copied into their respective shadow registers. The following registers are shadowed: W0-W3, and the C, Z, OV, N and DC STATUS register flags.

**Note 1:** The shadow registers are not directly accessible. They may only be accessed with PUSH.S and POP.S.

**Note 2:** The shadow registers are only one-level deep.

Words: 1

Cycles: 1

**Example 1:**

PUSH.S ; Push primary registers into shadow registers

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0000</td>
<td>W0 0000</td>
</tr>
<tr>
<td>W1 1000</td>
<td>W1 1000</td>
</tr>
<tr>
<td>W2 2000</td>
<td>W2 2000</td>
</tr>
<tr>
<td>W3 3000</td>
<td>W3 3000</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

**Note:** After an instruction execution, contents of the shadow registers are updated.
**PWRSAV**

Enter Power Saving Mode

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} PWRSAV #lit1

**Operands:**

lit1 ∈ [0,1]

**Operation:**

0 → WDT count register
0 → WDT prescaler A count
0 → WDT prescaler B count
0 → WDTO (RCON<4>)
0 → SLEEP (RCON<3>)
0 → IDLE (RCON<2>)

If (lit1 = 0):

Enter Sleep mode

Else:

Enter Idle mode

**Status Affected:** None

**Encoding:**

<table>
<thead>
<tr>
<th>Label</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>{label:}</td>
<td>1111</td>
</tr>
<tr>
<td>PWRSAV</td>
<td>1110</td>
</tr>
<tr>
<td>#lit1</td>
<td>0100</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>000k</td>
<td></td>
</tr>
</tbody>
</table>

**Description:**

Place the processor into the specified Power Saving mode. If lit1 = '0', Sleep mode is entered. In Sleep mode, the clock to the CPU and peripherals are shutdown. If an on-chip oscillator is being used, it is also shutdown. If lit1 = '1', Idle mode is entered. In Idle mode, the clock to the CPU shuts down, but the clock source remains active and the peripherals continue to operate.

This instruction resets the Watchdog Timer Count register and the Prescaler Count registers. In addition, the WDTO, Sleep and Idle flags of the Reset System and Control register (RCON) are reset.

**Note 1:** The processor will exit from Idle or Sleep through an interrupt, processor Reset or Watchdog Time-out. See the specific device data sheet for details.

2: If awakened from Idle mode, Idle bit (RCON<2>) is set to '1' and the clock source is applied to the CPU.

3: If awakened from Sleep mode, Sleep bit (RCON<3>) is set to '1' and the clock source is started.

4: If awakened from a Watchdog Time-out, WDTO bit (RCON<4>) is set to '1'.

**Words:** 1

**Cycles:** 1

**Example 1:**

PWRSAV #0 ; Enter SLEEP mode

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR: 0040 (IPL = 2)</td>
<td>SR: 0040 (IPL = 2)</td>
</tr>
</tbody>
</table>

**Example 2:**

PWRSAV #1 ; Enter IDLE mode

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR: 0020 (IPL = 1)</td>
<td>SR: 0020 (IPL = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**RCALL** Relative Call

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} RCALL Expr

**Operands:**

Expr may be an absolute address, label or expression. Expr is resolved by the linker to a Slit16, where Slit16 ∈ [-32768 ... 32767].

**Operation:**

(PC) + 2 → PC

(PC<15:0>) → (TOS)

(W15) + 2 → W15

(PC<22:16>) → (TOS)

(W15) + 2 → W15

(PC) + (2 * Slit16) → PC

NOP → Instruction Register

**Status Affected:** None

**Encoding:**

| 0000 | 0111 | nnnn | nnnn | nnnn | nnnn | nnnn |

**Description:**

Relative subroutine call with a range of 32K program words forward or back from the current PC. Before the call is made, the return address (PC + 2) is PUSHed onto the stack. After the return address is stacked, the sign-extended 17-bit value (2 * Slit16) is added to the contents of the PC and the result is stored in the PC.

The ‘n’ bits are a signed literal that specifies the size of the relative call (in program words) from (PC + 2).

**Note:** When possible, this instruction should be used instead of CALL, since it only consumes one word of program memory.

**Words:** 1

**Cycles:** 2

**Example 1:**

```
012004       RCALL _Task1 ; Call _Task1
012006       ADD W0, W1, W2
              ...
              ...
012458 _Task1: SUB W0, W2, W3 ; _Task1 subroutine
01245A       ...
```

**Table:**

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>01 2004</td>
<td>01 2458</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>0810</td>
<td>0814</td>
</tr>
<tr>
<td>Data 0810</td>
<td>Data 0810</td>
</tr>
<tr>
<td>FFFF</td>
<td>2006</td>
</tr>
<tr>
<td>Data 0812</td>
<td>Data 0812</td>
</tr>
<tr>
<td>FFFF</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:

```
00620E   RCALL _Init ; Call _Init
006210   MOV W0, [W4++]
  ...
  ...
007000 _Init:  CLR W2 ; _Init subroutine
007002  ...
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 620E</td>
<td>PC 00 7000</td>
</tr>
<tr>
<td>W15 0C50</td>
<td>W15 0C54</td>
</tr>
<tr>
<td>Data 0C50</td>
<td>Data 0C50 6210</td>
</tr>
<tr>
<td>Data 0C52</td>
<td>Data 0C52 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

RCALL  Relative Call

Implemented in:  

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  {label:} RCALL  Expr

Operands:  Expr may be an absolute address, label or expression. Expr is resolved by the linker to a Slit16, where Slit16 \( \in [-32768 \ldots 32767] \).

Operation:

\[
\begin{align*}
(PC) + 2 & \rightarrow \text{PC} \\
(PC<15:1>) & \rightarrow \text{TOS}<15:1>, \text{SFA bit} \rightarrow \text{TOS}<0> \\
(W15) + 2 & \rightarrow W15 \\
(PC<22:16>) & \rightarrow (\text{TOS}) \\
(W15) + 2 & \rightarrow W15 \\
0 & \rightarrow \text{SFA bit} \\
(PC) + (2 \times \text{Slit16}) & \rightarrow \text{PC} \\
\text{NOP} & \rightarrow \text{Instruction Register}
\end{align*}
\]

Status Affected:  SFA

Encoding:

\[
\begin{array}{cccccccc}
0000 & 0111 & nnnn & nnnn & nnnn & nnnn & nnnn \\
\end{array}
\]

Description:  Relative subroutine call with a range of 32K program words forward or back from the current PC. Before the call is made, the return address (PC + 2) is PUSHed onto the stack. After the return address is stacked, the sign-extended 17-bit value (2 \times \text{Slit16}) is added to the contents of the PC and the result is stored in the PC.

The 'n' bits are a signed literal that specifies the size of the relative call (in program words) from (PC + 2).

**Note:**  When possible, this instruction should be used instead of CALL, since it only consumes one word of program memory.

Words:  1
Cycles:  4

**Example 1:**

012004  RCALL  _Task1  ; Call _Task1
012006  ADD    W0, W1, W2
.        ...
.        ...
012458  _Task1:  SUB    W0, W2, W3  ; _Task1 subroutine
01245A  ...

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>Instruction</td>
</tr>
<tr>
<td>PC</td>
<td>01 2004</td>
</tr>
<tr>
<td>W15</td>
<td>0810</td>
</tr>
<tr>
<td>Data 0810</td>
<td>FFFF</td>
</tr>
<tr>
<td>Data 0812</td>
<td>FFFF</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2:

Example 2:

```
00620E          RCALL  _Init       ; Call _Init
006210          MOV     W0, [W4++]
...            ...
007000 _Init:   CLR     W2        ; _Init subroutine
007002          ...
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 620E</td>
<td>PC 00 7000</td>
</tr>
<tr>
<td>W15 0C50</td>
<td>W15 0C54</td>
</tr>
<tr>
<td>Data 0C50 FFFF</td>
<td>Data 0C50 6210</td>
</tr>
<tr>
<td>Data 0C52 FFFF</td>
<td>Data 0C52 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
RCALL  

**Section 5. Instruction Descriptions**

**RCALL**  

**Computed Relative Call**

**Implemented in:**  
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**  
{label:} RCALL Wn

**Operands:**  
Wn ∈ [W0 ... W15]

**Operation:**
(PC) + 2 → PC  
(PC<15:0>) → (TOS)  
(W15) + 2 → W15  
(PC<22:16>) → (TOS)  
(W15) + 2 → W15  
(PC) + (2 * (Wn)) → PC

**NOP → Instruction Register**

**Status Affected:**  
None

**Encoding:**

```
0000 0001 0010 0000 0000 ssss
```

**Description:** Computed, relative subroutine call specified by the working register Wn. The range of the call is 32K program words forward or back from the current PC. Before the call is made, the return address (PC + 2) is PUSHed onto the stack. After the return address is stacked, the sign-extended 17-bit value (2 * (Wn)) is added to the contents of the PC and the result is stored in the PC. Register direct addressing must be used for Wn.

The ‘s’ bits select the source register.

**Words:** 1

**Cycles:** 2

**Example 1:**

00FF8C  EX1: INC  W2, W3  ; Destination of RCALL
00FF8E          ...
          ...
          ...
010008
01000A RCALL W6  ; RCALL with W6
01000C MOVE  W4, [W10]

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>01 000A</td>
<td>00 FF8C</td>
</tr>
<tr>
<td>W6</td>
<td>W6</td>
</tr>
<tr>
<td>FFC0</td>
<td>FFC0</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>1004</td>
<td>1008</td>
</tr>
<tr>
<td>Data 1004</td>
<td>Data 1004</td>
</tr>
<tr>
<td>98FF</td>
<td>000C</td>
</tr>
<tr>
<td>Data 1006</td>
<td>Data 1006</td>
</tr>
<tr>
<td>2310</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2: 

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 0302</td>
<td>00 0450</td>
</tr>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>00A6</td>
<td>00A6</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>1004</td>
<td>1008</td>
</tr>
<tr>
<td>Data 1004</td>
<td>Data 1004</td>
</tr>
<tr>
<td>32BB</td>
<td>0304</td>
</tr>
<tr>
<td>Data 1006</td>
<td>Data 1006</td>
</tr>
<tr>
<td>901A</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2:

000302          RCALL W2 ; RCALL with W2
000304          FF1L W0, W1

... 

000450  EX2: CLR W2 ; Destination of RCALL
000452       ...

Before Instruction:

- PC: 00 0302
- W2: 00A6
- W15: 1004
- Data 1004: 32BB
- Data 1006: 901A
- SR: 0000

After Instruction:

- PC: 00 0450
- W2: 00A6
- W15: 1008
- Data 1004: 0304
- Data 1006: 0000
- SR: 0000
RCALL Computed Relative Call

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: \{label:\} RCALL Wn

Operands: Wn ∈ [W0 ... W15]

Operation:
- (PC) + 2 \rightarrow PC
- (PC<15:1>) \rightarrow TOS<15:1>, SFA bit \rightarrow TOS<0>
- (W15) + 2 \rightarrow W15
- (PC<22:16>) \rightarrow TOS
- (W15) + 2 \rightarrow W15
- 0 \rightarrow SFA bit
- (PC) + (2 \times (Wn)) \rightarrow PC
- NOP \rightarrow Instruction Register

Status Affected: SFA

Encoding:

| 0000 | 0001 | 0000 | 0010 | 0000 | ssss |

Description:
Composed, relative subroutine call specified by the working register Wn. The range of the call is 32K program words forward or back from the current PC. Before the call is made, the return address (PC + 2) is PUSHed onto the stack. After the return address is stacked, the sign-extended 17-bit value (2 \times (Wn)) is added to the contents of the PC and the result is stored in the PC. Register direct addressing must be used for Wn.

The ‘s’ bits select the source register.

Words: 1
Cycles: 4

Example 1:

```
00FF8C  EX1: INC  W2, W3 ; Destination of RCALL
00FF8E   ...                          
.   ...                          
.   ...                          
010008
01000A  RCALL W6 ; RCALL with W6
01000C  MOVE W4, [W10]
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>01 000A</td>
<td>00 FF8C</td>
</tr>
<tr>
<td>W6</td>
<td>FFC0</td>
</tr>
<tr>
<td>W15</td>
<td>1004</td>
</tr>
<tr>
<td>Data 1004</td>
<td>98FF</td>
</tr>
<tr>
<td>Data 1006</td>
<td>2310</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© 2005-2011 Microchip Technology Inc.
Example 2:

```
000302   RCALL  W2       ; RCALL with W2
000304   FF1L  W0, W1
...
000450   EX2:  CLR  W2     ; Destination of RCALL
000452   ...  
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>00 0302</td>
</tr>
<tr>
<td>W2</td>
<td>00A6</td>
</tr>
<tr>
<td>W15</td>
<td>1004</td>
</tr>
<tr>
<td>Data 1004</td>
<td>32BB</td>
</tr>
<tr>
<td>Data 1006</td>
<td>901A</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>PC</td>
<td>00 0450</td>
</tr>
<tr>
<td>W2</td>
<td>00A6</td>
</tr>
<tr>
<td>W15</td>
<td>1008</td>
</tr>
<tr>
<td>Data 1004</td>
<td>0304</td>
</tr>
<tr>
<td>Data 1006</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
REPEAT

Repeat Next Instruction ‘lit14 + 1’ Times

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} REPEAT #lit14

Operands:  
lit14 $\in [0 \ldots 16383]$

Operation:  
\((\text{lit14}) \rightarrow \text{RCOUNT}\)
\((\text{PC}) + 2 \rightarrow \text{PC}\)

Enable Code Looping

Status Affected:  
RA

Encoding:

```
0000 1001 00kk kkkk kkkk kkkk
```

Description:  
Repeat the instruction immediately following the REPEAT instruction \((\text{lit14} + 1)\) times. The repeated instruction (or target instruction) is held in the instruction register for all iterations and is only fetched once. When this instruction executes, the RCOUNT register is loaded with the repeat count value specified in the instruction. RCOUNT is decremented with each execution of the target instruction. When RCOUNT equals zero, the target instruction is executed one more time, and then normal instruction execution continues with the instruction following the target instruction.

The ‘k’ bits are an unsigned literal that specifies the loop count.

Special Features, Restrictions:

1. When the repeat literal is ‘0’, REPEAT has the effect of a NOP and the RA bit is not set.
2. The target REPEAT instruction cannot be:
   - an instruction that changes program flow
   - a DO, DISI, LNK, MOV.D, PWRSAV, REPEAT or UNLK instruction
   - a 2-word instruction

Unexpected results may occur if these target instructions are used.

Note:  
The REPEAT and target instruction are interruptible.

Words:  
1

Cycles:  
1

Example 1:  
000452 REPEAT #9 ; Execute ADD 10 times
000454 ADD [W0++], W1, [W2++] ; Vector update

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 0452</td>
<td>PC 00 0454</td>
</tr>
<tr>
<td>RCOUNT 0000</td>
<td>RCOUNT 0009</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0010 (RA = 1)</td>
</tr>
</tbody>
</table>
Example 2: 00089E REPEAT #0x3FF ; Execute CLR 1024 times
0008A0 CLR [W6++] ; Clear the scratch space

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 089E</td>
<td>PC 00 08A0</td>
</tr>
<tr>
<td>RCOUNT 0000</td>
<td>RCOUNT 03FF</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0010 (RA = 1)</td>
</tr>
</tbody>
</table>
**REPEAT**

Repeat Next Instruction ‘lit15 + 1’ Times

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} REPEAT #lit15

**Operands:**

lit15 ∈ [0 ... 32767]

**Operation:**

(lit15) → RCOUNT

(PC) + 2 → PC

Enable Code Looping

**Status Affected:** RA

**Encoding:**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1001</td>
<td>0kkk</td>
<td>kkkk</td>
<td>kkkk</td>
<td>kkkk</td>
</tr>
</tbody>
</table>

**Description:**

Repeat the instruction immediately following the REPEAT instruction (lit15 + 1) times. The repeated instruction (or target instruction) is held in the instruction register for all iterations and is only fetched once.

When this instruction executes, the RCOUNT register is loaded with the repeat count value specified in the instruction. RCOUNT is decremented with each execution of the target instruction. When RCOUNT equals zero, the target instruction is executed one more time, and then normal instruction execution continues with the instruction following the target instruction.

The ‘k’ bits are an unsigned literal that specifies the loop count.

**Special Features, Restrictions:**

1. When the repeat literal is ‘0’, REPEAT has the effect of a NOP and the RA bit is not set.

2. The target REPEAT instruction cannot be:
   - an instruction that changes program flow
   - a DISI, LNK, MOV.D, PWRSAV, REPEAT or UNLK instruction
   - a 2-word instruction

Unexpected results may occur if these target instructions are used.

**Note:** The REPEAT and target instruction are interruptible.

**Words:** 1

**Cycles:** 1

**Example 1:**

000452 REPEAT #9 ; Execute ADD 10 times
000454 ADD [W0++], [W1], [W2++] ; Vector update

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>RCOUNT</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0452</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>RCOUNT</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 0454</td>
<td>0009</td>
<td>0010 (RA = 1)</td>
</tr>
</tbody>
</table>
**Example 2:**

00089E  REPEAT  #0x3FF  ; Execute CLR 1024 times
0008A0  CLR  [W6++]  ; Clear the scratch space

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 089E</td>
<td>00 08A0</td>
</tr>
<tr>
<td>RCOUNT</td>
<td>RCOUNT</td>
</tr>
<tr>
<td>0000</td>
<td>03FF</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0010 (RA = 1)</td>
</tr>
</tbody>
</table>

Before Instruction:
- PC: 00 089E
- RCOUNT: 0000
- SR: 0000

After Instruction:
- PC: 00 08A0
- RCOUNT: 03FF
- SR: 0010 (RA = 1)
Section 5. Instruction Descriptions

REPEAT

Repeat Next Instruction Wn+1 Times

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

X X X X X

Syntax: {label:} REPEAT Wn

Operands: Wn ∈ [W0 ... W15]

Operation:

(Wn<13:0>) → RCOUNT

(PS) + 2 → PC

Enable Code Looping

Status Affected: RA

Encoding:

<table>
<thead>
<tr>
<th>0000</th>
<th>1001</th>
<th>1000</th>
<th>0000</th>
<th>0000</th>
<th>ssss</th>
</tr>
</thead>
</table>

Description:
Repeat the instruction immediately following the REPEAT instruction (Wn<13:0>) times. The instruction to be repeated (or target instruction) is held in the instruction register for all iterations and is only fetched once.

When this instruction executes, the RCOUNT register is loaded with the lower 14 bits of Wn. RCOUNT is decremented with each execution of the target instruction. When RCOUNT equals zero, the target instruction is executed one more time, and then normal instruction execution continues with the instruction following the target instruction.

The 's' bits specify the Wn register that contains the repeat count.

Special Features, Restrictions:
1. When (Wn) = 0, REPEAT has the effect of a NOP and the RA bit is not set.
2. The target REPEAT instruction cannot be:
   • an instruction that changes program flow
   • a DO, DISI, LNK, MOV.D, PWRSAV, REPEAT or ULNK instruction
   • a 2-word instruction

Unexpected results may occur if these target instructions are used.

Note: The REPEAT and target instruction are interruptible.

Words: 1
Cycles: 1

Example 1:

000A26 REPEAT W4 ; Execute COM (W4+1) times
000A28 COM [W0++], [W2++] ; Vector complement

Before Instruction                  After Instruction

<table>
<thead>
<tr>
<th>PC</th>
<th>00 0A26</th>
<th>PC</th>
<th>00 0A28</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>0023</td>
<td>W4</td>
<td>0023</td>
</tr>
<tr>
<td>RCOUNT</td>
<td>0000</td>
<td>RCOUNT</td>
<td>0023</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td>SR</td>
<td>0010</td>
</tr>
</tbody>
</table>

(RA = 1)
Example 2:

```
00089E  REPEAT W10 ; Execute TBLRD (W10+1) times
0008A0  TBLRDL [W2++], [W3++] ; Decrement (0x840)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 089E</td>
<td>PC 00 08A0</td>
</tr>
<tr>
<td>W10 00FF</td>
<td>W10 00FF</td>
</tr>
<tr>
<td>RCOUNT 0000</td>
<td>RCOUNT 00FF</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0010 (RA = 1)</td>
</tr>
</tbody>
</table>
**REPEAT**

Repeat Next Instruction Wn+1 Times

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} REPEAT Wn`

Operands: Wn ∈ [W0 ... W15]

Operation:

(Wn) → RCOUNT

(PC) + 2 → PC

Enable Code Looping

Status Affected: RA

Encoding:

0000 1001 1000 0000 0000 ssss

Description:

Repeat the instruction immediately following the REPEAT instruction (Wn) times. The instruction to be repeated (or target instruction) is held in the instruction register for all iterations and is only fetched once.

When this instruction executes, the RCOUNT register is loaded with Wn. RCOUNT is decremented with each execution of the target instruction. When RCOUNT equals zero, the target instruction is executed one more time, and then normal instruction execution continues with the instruction following the target instruction.

The ‘s’ bits specify the Wn register that contains the repeat count.

**Special Features, Restrictions:**

1. When (Wn) = 0, REPEAT has the effect of a NOP and the RA bit is not set.

2. The target REPEAT instruction cannot be:
   - an instruction that changes program flow
   - a DO, DISI, LNK, MOV.D, PWRSAV, REPEAT or ULNK instruction
   - a 2-word instruction

   Unexpected results may occur if these target instructions are used.

**Note:** The REPEAT and target instruction are interruptible.

Words: 1

Cycles: 1

**Example 1:**

000A26 REPEAT W4 ; Execute COM (W4+1) times
000A28 COM [W0++], [W2++] ; Vector complement
**Example 2:**

```
00089E  REPEAT W10 ; Execute TBLRD (W10+1) times
0008A0  TBLRDL [W2++], [W3++] ; Decrement (0x840)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 089E</td>
<td>PC 00 08A0</td>
</tr>
<tr>
<td>W10 00FF</td>
<td>W10 00FF</td>
</tr>
<tr>
<td>RCOUNT 0000</td>
<td>RCOUNT 00FF</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0010 (RA = 1)</td>
</tr>
</tbody>
</table>
## RESET

### Reset

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} RESET`

Operands: None

Operation: Force all registers that are affected by a MCLR Reset to their Reset condition.

1 → SWR (RCON<6>)
0 → PC

Status Affected: OA, OB, OAB, SA, SB, SAB, DA, DC, IPL<2:0>, RA, N, OV, Z, C, SFA

Encoding: 1111 1110 0000 0000 0000 0000

Description: This instruction provides a way to execute a software Reset. All core and peripheral registers will take their power-on value. The PC will be set to '0', the location of the `RESET GOTO` instruction. The SWR bit (RCON<6>), will be set to '1' to indicate that the `RESET` instruction was executed.

**Note:** Refer to the specific device family reference manual for the power-on value of all registers.

Words: 1

Cycles: 1
Example 1: 00202A  RESET  ; Execute software RESET on dsPIC33F

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 202A</td>
<td>PC 00 0000</td>
</tr>
<tr>
<td>W0 8901</td>
<td>W0 0000</td>
</tr>
<tr>
<td>W1 08BB</td>
<td>W1 0000</td>
</tr>
<tr>
<td>W2 B87A</td>
<td>W2 0000</td>
</tr>
<tr>
<td>W3 872F</td>
<td>W3 0000</td>
</tr>
<tr>
<td>W4 C98A</td>
<td>W4 0000</td>
</tr>
<tr>
<td>W5 AAD4</td>
<td>W5 0000</td>
</tr>
<tr>
<td>W6 981E</td>
<td>W6 0000</td>
</tr>
<tr>
<td>W7 1809</td>
<td>W7 0000</td>
</tr>
<tr>
<td>W8 C341</td>
<td>W8 0000</td>
</tr>
<tr>
<td>W9 90F4</td>
<td>W9 0000</td>
</tr>
<tr>
<td>W10 F409</td>
<td>W10 0000</td>
</tr>
<tr>
<td>W11 1700</td>
<td>W11 0000</td>
</tr>
<tr>
<td>W12 1008</td>
<td>W12 0000</td>
</tr>
<tr>
<td>W13 6556</td>
<td>W13 0000</td>
</tr>
<tr>
<td>W14 231D</td>
<td>W14 0000</td>
</tr>
<tr>
<td>W15 1704</td>
<td>W15 0800</td>
</tr>
<tr>
<td>SPLIM 1800</td>
<td>SPLIM 0000</td>
</tr>
<tr>
<td>TBLPAG 007F</td>
<td>TBLPAG 0000</td>
</tr>
<tr>
<td>PSVPAG 0001</td>
<td>PSVPAG 0000</td>
</tr>
<tr>
<td>CORCON 00F0</td>
<td>CORCON 0020 (SATDW = 1)</td>
</tr>
<tr>
<td>RCON 0000</td>
<td>RCON 0040 (SWR = 1)</td>
</tr>
<tr>
<td>SR 0021 (IPL, C = 1)</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
**RETFIE**  
Return from Interrupt

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} RETFIE

Operands:  
None

Operation:  
(W15) - 2 → W15  
(TOS<15:8>) → (SR<7:0>)  
(TOS<7>) → (IPL3, CORCON<3>)  
(TOS<6:0>) → (PC<22:16>)  
(W15) - 2 → W15  
(TOS<15:0>) → (PC<15:0>)  
NOP → Instruction Register

Status Affected:  
IPL<3:0>, RA, N, OV, Z, C

Encoding:  
0000 0110 0100 0000 0000 0000

Description:  
Return from Interrupt Service Routine. The stack is POPped, which loads the low byte of the STATUS register, IPL<3> (CORCON<3>) and the Most Significant Byte of the PC. The stack is POPped again, which loads the lower 16 bits of the PC.

**Note 1:** Restoring IPL<3> and the low byte of the STATUS register restores the Interrupt Priority Level to the level before the execution was processed.

**Note 2:** Before RETFIE is executed, the appropriate interrupt flag must be cleared in software to avoid recursive interrupts.

Words:  
1

Cycles:  
3 (2 if exception pending)

**Example 1:**  
000A26 RETFIE ; Return from ISR

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 0A26</td>
<td>01 0230</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>0834</td>
<td>0830</td>
</tr>
<tr>
<td>Data 0830</td>
<td>Data 0830</td>
</tr>
<tr>
<td>0230</td>
<td>0230</td>
</tr>
<tr>
<td>Data 0832</td>
<td>Data 0832</td>
</tr>
<tr>
<td>8101</td>
<td>8101</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0081</td>
</tr>
</tbody>
</table>

(IPL = 4, C = 1)

**Example 2:**  
008050 RETFIE ; Return from ISR

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>00 8050</td>
<td>00 7008</td>
</tr>
<tr>
<td>W15</td>
<td>W15</td>
</tr>
<tr>
<td>0926</td>
<td>0922</td>
</tr>
<tr>
<td>Data 0922</td>
<td>Data 0922</td>
</tr>
<tr>
<td>7008</td>
<td>7008</td>
</tr>
<tr>
<td>Data 0924</td>
<td>Data 0924</td>
</tr>
<tr>
<td>0300</td>
<td>0300</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0003</td>
</tr>
</tbody>
</table>

(Z, C = 1)
RETFIE  Return from Interrupt

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E  X  X

Syntax:  {label:} RETFIE

Operands: None

Operation:

(W15) - 2 → W15
(TOS<15:8>) → (SR<7:0>)
(TOS<7>) → (IPL3, CORCON<3>)
(TOS<6:0>) → (PC<22:16>)
(W15) - 2 → W15
(TOS<15:1>) → (PC<15:1>)
TOS<0> → SFA bit
N O P → Instruction Register

Status Affected: IPL<3:0>, RA, N, OV, Z, C, SFA

Encoding: 0000 0110 0100 0000 0000 0000 0000

Description: Return from Interrupt Service Routine. The stack is POPped, which loads the low byte of the STATUS register, IPL<3> (CORCON<3>) and the Most Significant Byte of the PC. The stack is POPped again, which loads the lower 16 bits of the PC.

Note 1: Restoring IPL<3> and the low byte of the STATUS register restores the Interrupt Priority Level to the level before the execution was processed.

2: Before RETFIE is executed, the appropriate interrupt flag must be cleared in software to avoid recursive interrupts.

Words: 1
Cycles: 6 (5 if exception pending)

Example 1: 000A26  RETFIE  ; Return from ISR

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 0A26</td>
<td>PC 01 0230</td>
</tr>
<tr>
<td>W15 0834</td>
<td>W15 0830</td>
</tr>
<tr>
<td>Data 0830</td>
<td>Data 0830</td>
</tr>
<tr>
<td>Data 0832</td>
<td>Data 0832</td>
</tr>
<tr>
<td>CORCON 0001</td>
<td>CORCON 0001</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0081</td>
</tr>
</tbody>
</table>

(IPL = 4, C = 1)

Example 2: 008050  RETFIE  ; Return from ISR

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 8050</td>
<td>PC 00 7008</td>
</tr>
<tr>
<td>W15 0926</td>
<td>W15 0922</td>
</tr>
<tr>
<td>Data 0922</td>
<td>Data 0922</td>
</tr>
<tr>
<td>Data 0924</td>
<td>Data 0924</td>
</tr>
<tr>
<td>CORCON 0000</td>
<td>CORCON 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0003</td>
</tr>
</tbody>
</table>

(Z, C = 1)
**Section 5. Instruction Descriptions**

---

**RETLW**

**Return with Literal in Wn**

**Implemented in:**
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} RETLW{.B} #lit10, Wn

**Operands:**

lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

**Operation:**

(W15) – 2 → W15
TOS<15:8> → SR<7:0>
TOS<7:0> → PL<3> : PC<22:16>
(W15) – 2 → W15
(TOS) → (PC<15:0>)
lit10 → Wn

NOP → Instruction register

**Status Affected:** None

**Encoding:**

| 0000 | 0101 | 0Bkk | kkkk | kkkk | dddd |

**Description:**

Return from subroutine with the specified, unsigned 10-bit literal stored in Wn. The software stack is POPped twice to restore the PC and the signed literal is stored in Wn. Since two POPs are made, the Stack Pointer (W15) is decremented by 4.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘k’ bits specify the value of the literal.

The ‘d’ bits select the destination register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

**Words:** 1

**Cycles:** 3 (2 if exception pending)

**Example 1:**

000440 RETLW.B #0xA, W0 ; Return with 0xA in W0
Example 2:  00050A   RETLW  #0x230, W2 ; Return with 0x230 in W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC 00 050A</td>
<td>PC 01 7008</td>
</tr>
<tr>
<td>W2 0993</td>
<td>W2 0230</td>
</tr>
<tr>
<td>W15 1200</td>
<td>W15 11FC</td>
</tr>
<tr>
<td>Data 11FC 7008</td>
<td>Data 11FC 7008</td>
</tr>
<tr>
<td>Data 11FE 0001</td>
<td>Data 11FE 0001</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

**RETLW**

Return with Literal in Wn

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Syntax:** (label:)  RETLW(.B)  #lit10,  Wn

**Operands:**
- lit10 ∈ [0 ... 255] for byte operation
- lit10 ∈ [0 ... 1023] for word operation
- Wn ∈ [W0 ... W15]

**Operation:**
- (W15) – 2 → W15
- TOS<15:8> → SR<7:0>
- TOS<7:0> → IPL<3>: PC<22:16>
- (W15) – 2 → W15
- (TOS<15:1>) → (PC<15:1>)
- TOS<0> → SFA bit
- lit10 → Wn
- NOP → Instruction register

**Status Affected:** SFA

**Encoding:**

```
0000 0101 0Bkk kkkk kkkk dddd
```

**Description:**
Return from subroutine with the specified, unsigned 10-bit literal stored in Wn. The software stack is POPped twice to restore the PC and the signed literal is stored in Wn. Since two POPs are made, the Stack Pointer (W15) is decremented by 4.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘k’ bits specify the value of the literal.
The ‘d’ bits select the destination register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

**Words:** 1
**Cycles:** 6 (5 if exception pending)

**Example 1:**
```
000440 RETLW.B #0xA, W0 ; Return with 0xA in W0
```

**Before**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 0440</th>
<th>PC</th>
<th>00 7006</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>9846</td>
<td>W0</td>
<td>980A</td>
</tr>
<tr>
<td>W15</td>
<td>1988</td>
<td>W15</td>
<td>1984</td>
</tr>
<tr>
<td>Data 1984</td>
<td>7006</td>
<td>Data 1984</td>
<td>7006</td>
</tr>
<tr>
<td>Data 1986</td>
<td>0000</td>
<td>Data 1986</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After**
Example 2: 00050A RETLW #0x230, W2 ; Return with 0x230 in W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>00 050A</td>
</tr>
<tr>
<td>W2</td>
<td>0993</td>
</tr>
<tr>
<td>W15</td>
<td>1200</td>
</tr>
<tr>
<td>Data 11FC</td>
<td>7008</td>
</tr>
<tr>
<td>Data 11FE</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### RETURN

**Return**

**Implemented in:**
- PIC24F: X
- PIC24H: X
- PIC24E: X
- dsPIC30F: X
- dsPIC33F: X
- dsPIC33E: X

**Syntax:**
```
{label:} RETURN
```

**Operands:** None

**Operation:**
- `(W15) – 2 → W15`
- `(TOS) → (PC<22:16>)`
- `(W15) – 2 → W15`
- `(TOS) → (PC<15:0>)`
- NOP → Instruction Register

**Status Affected:** None

**Encoding:**
```
0000 0110 0000 0000 0000 0000
```

**Description:**
Return from subroutine. The software stack is POPped twice to restore the PC. Since two POPs are made, the Stack Pointer (W15) is decremented by 4.

**Words:** 1

**Cycles:** 3 (2 if exception pending)

**Example 1:**
```
001A06   RETURN     ; Return from subroutine
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 1A06</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15</td>
<td>1248</td>
</tr>
<tr>
<td>Data</td>
<td>1244</td>
</tr>
<tr>
<td>Data</td>
<td>1246</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>01 0004</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15</td>
<td>1244</td>
</tr>
<tr>
<td>Data</td>
<td>1244</td>
</tr>
<tr>
<td>Data</td>
<td>1246</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**
```
005404   RETURN     ; Return from subroutine
```

**Before Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 5404</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15</td>
<td>090A</td>
</tr>
<tr>
<td>Data</td>
<td>0906</td>
</tr>
<tr>
<td>Data</td>
<td>0908</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**After Instruction**

<table>
<thead>
<tr>
<th>PC</th>
<th>00 0966</th>
</tr>
</thead>
<tbody>
<tr>
<td>W15</td>
<td>0906</td>
</tr>
<tr>
<td>Data</td>
<td>0906</td>
</tr>
<tr>
<td>Data</td>
<td>0908</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
RETURN

Return

Implemented in: PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E
X | X

Syntax: {label:} RETURN

Operands: None

Operation:
(W15) – 2 → W15
(TOS) → (PC<22:16>)
(W15) – 2 → W15
(TOS<15:1) → (PC<15:1>)
TOS<0> → SFA bit
NOP → Instruction Register

Status Affected: SFA

Encoding:

Description: Return from subroutine. The software stack is POPped twice to restore
the PC. Since two POPs are made, the Stack Pointer (W15) is
decremented by 4.

Words: 1
Cycles: 6 (5 if exception pending)

Example 1: 001A06 RETURN ; Return from subroutine

Before Instruction | After Instruction
PC 00 1A06 | PC 01 0004
W15 1248 | W15 1244
Data 1244 0004 | Data 1244 0004
Data 1246 0001 | Data 1246 0001
SR 0000 | SR 0000

Example 2: 005404 RETURN ; Return from subroutine

Before Instruction | After Instruction
PC 00 5404 | PC 00 0966
W15 090A | W15 0906
Data 0906 0966 | Data 0906 0966
Data 0908 0000 | Data 0908 0000
SR 0000 | SR 0000
Section 5. Instruction Descriptions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Rotate Left f through Carry</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} RLC{.B} f {,WREG}`

Operands: \( f \in [0 \ldots 8191] \)

Operation:
For byte operation:
- \((C) \rightarrow \text{Dest}<0>\)
- \( (f<6:0>) \rightarrow \text{Dest}<7:1> \)
- \( (f<7>) \rightarrow C \)

For word operation:
- \((C) \rightarrow \text{Dest}<0>\)
- \( (f<14:0>) \rightarrow \text{Dest}<15:1> \)
- \( (f<15>) \rightarrow C \)

Status Affected: \( N, Z, C \)

Encoding:
```
1101 0110 1BDf ffff ffff ffff
```

Description:
Rotate the contents of the file register \( f \) one bit to the left through the Carry flag and place the result in the destination register. The Carry flag of the STATUS Register is shifted into the Least Significant bit of the destination, and it is then overwritten with the Most Significant bit of Ws.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for f, ‘1’ for WREG).

The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Note 2: The WREG is set to working register W0.

Words: 1
Cycles: 1

Example 1: RLC.B 0x1233 ; Rotate Left w/ C (0x1233) (Byte mode)

<table>
<thead>
<tr>
<th>Data 1232</th>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1232</td>
<td>E807</td>
<td>D007</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0009</td>
<td>(N, C = 1)</td>
</tr>
</tbody>
</table>

© 2005-2011 Microchip Technology Inc.
### Example 2:

```
RLC 0x820, WREG ; Rotate Left w/ C (0x820) (Word mode)
; Store result in WREG
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>5601</td>
<td>42DD</td>
</tr>
<tr>
<td>Data 0820</td>
<td>Data 0820</td>
</tr>
<tr>
<td>216E</td>
<td>216E</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0001 (C = 1)</td>
<td>0000 (C = 0)</td>
</tr>
</tbody>
</table>
### RLC
**Rotate Left Ws through Carry**

#### Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

- `{label:} RLC{.B} Ws, Wd
- `[Ws], [Wd]
- `[Ws++], [Wd++]
- `[Ws--], [Wd--]
- `[++Ws], [++Wd]
- `[--Ws], [--Wd]

#### Operands:
- `Ws ∈ [W0 ... W15]`
- `Wd ∈ [W0 ... W15]`

#### Operation:

For byte operation:
- `(C) → Wd<0>
- (Ws<6:0>) → Wd<7:1>
- (Ws<7>) → C

For word operation:
- `(C) → Wd<0>
- (Ws<14:0>) → Wd<15:1>
- (Ws<15>) → C

#### Status Affected:
- N, Z, C

#### Encoding:

| 1101 | 0010 | 1Bqq | qddd | dppp | ssss |

#### Description:

Rotate the contents of the source register `Ws` one bit to the left through the Carry flag and place the result in the destination register `Wd`. The Carry flag of the STATUS register is shifted into the Least Significant bit of `Wd`, and it is then overwritten with the Most Significant bit of `Ws`. Either register direct or indirect addressing may be used for `Ws` and `Wd`.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

#### Words:

1

#### Cycles:

1

#### Example 1:

```
RLC.B W0, W3  ; Rotate Left w/ C (W0) (Byte mode)
; Store the result in W3
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 9976</td>
<td>W0 9976</td>
</tr>
<tr>
<td>W3 5879</td>
<td>W3 58ED</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0009 (N = 1)</td>
</tr>
</tbody>
</table>
Example 2: RLC [W2++], [W8] ; Rotate Left w/ C [W2] (Word mode)
; Post-increment W2
; Store result in [W8]

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 2008</td>
<td>W2 200A</td>
</tr>
<tr>
<td>W8 094E</td>
<td>W8 094E</td>
</tr>
<tr>
<td>Data 094E 3689</td>
<td>Data 094E 8082</td>
</tr>
<tr>
<td>Data 2008 C041</td>
<td>Data 2008 C041</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0009 (N, C = 1)</td>
</tr>
</tbody>
</table>
**Section 5. Instruction Descriptions**

### RLNC

**Rotate Left f without Carry**

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} RLNC{.B} f {,WREG}

**Operands:**

f ∈ [0 ... 8191]

**Operation:**

For byte operation:

(f<6:0>) → Dest<7:1>
(f<7>) → Dest<0>

For word operation:

(f<14:0>) → Dest<15:1>
(f<15>) → Dest<0>

**Status Affected:**

N, Z

**Encoding:**

1101 0110 0BDf ffff ffff ffff

**Description:**

Rotate the contents of the file register f one bit to the left and place the result in the destination register. The Most Significant bit of f is stored in the Least Significant bit of the destination, and the Carry flag is not affected.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

**Note 2:** The WREG is set to working register W0.
Example 1:  RLNC.B 0x1233 ; Rotate Left (0x1233) (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1232</td>
<td>Data 1233</td>
</tr>
<tr>
<td>E807</td>
<td>D107</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008</td>
</tr>
</tbody>
</table>

(N = 1)

Example 2:  RLNC 0x820, WREG ; Rotate Left (0x820) (Word mode) ; Store result in WREG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>5601</td>
<td>42DC</td>
</tr>
<tr>
<td>Data 0820</td>
<td>Data 0820</td>
</tr>
<tr>
<td>216E</td>
<td>216E</td>
</tr>
<tr>
<td>SR 0001</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

(C = 1)  (C = 0)
Section 5. Instruction Descriptions

RLNC

Rotate Left Ws without Carry

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label;} RLNC(.B) Ws, Wd

[Ws], [Wd]

[Ws++], [Wd++]

[Ws--], [Wd--]

[++Ws], [++Wd]

[--Ws], [--Wd]

Operands:

Ws ∈ [W0 ... W15]

Wd ∈ [W0 ... W15]

Operation:

For byte operation:

(Ws<6:0>) → Wd<7:1>

(Ws<7>) → Wd<0>

For word operation:

(Ws<14:0>) → Wd<15:1>

(Ws<15>) → Wd<0>

Status Affected: N, Z

Encoding:

1101 0010 0Bqq qddd dppp ssss

Description:

Rotate the contents of the source register Ws one bit to the left and place the result in the destination register Wd. The Most Significant bit of Ws is stored in the Least Significant bit of Wd, and the Carry flag is not affected. Either register direct or indirect addressing may be used for Ws and Wd.

The 'B' bit selects byte or word operation ('0' for byte, '1' for word).

The 'q' bits select the destination Address mode.

The 'd' bits select the destination register.

The 'p' bits select the source Address mode.

The 's' bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
**Example 1:**  
**RLNC.B W0, W3**  
; Rotate Left (W0) (Byte mode)  
; Store the result in W3

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 9976</td>
<td>W0 9976</td>
</tr>
<tr>
<td>W3 5879</td>
<td>W3 58EC</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0009 (N, C = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**  
**RLNC [W2++], [W8]**  
; Rotate Left [W2] (Word mode)  
; Post-increment W2  
; Store result in [W8]

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 2008</td>
<td>W2 200A</td>
</tr>
<tr>
<td>W8 094E</td>
<td>W8 094E</td>
</tr>
<tr>
<td>Data 094E</td>
<td>Data 094E</td>
</tr>
<tr>
<td>3689</td>
<td>8083</td>
</tr>
<tr>
<td>Data 2008</td>
<td>Data 2008</td>
</tr>
<tr>
<td>C041</td>
<td>C041</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0009 (N, C = 1)</td>
</tr>
</tbody>
</table>
RRC

Rotate Right f through Carry

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} RRC{.B} f {,WREG}`

Operands: `f ∈ [0 ... 8191]`

Operation:

For byte operation:

- `(C) → Dest<7>`
- `(f<7:1>) → Dest<6:0>`
- `(f<0>) → C`

For word operation:

- `(C) → Dest<15>`
- `(f<15:1>) → Dest<14:0>`
- `(f<0>) → C`

Status Affected: N, Z, C

Encoding: `1101 0111 1BDf ffff ffff ffff`

Description:

Rotate the contents of the file register `f` one bit to the right through the Carry flag and place the result in the destination register. The Carry flag of the STATUS Register is shifted into the Most Significant bit of the destination, and it is then overwritten with the Least Significant bit of Ws.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for byte, ‘1’ for word).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Note 2: The WREG is set to working register W0.

Words: 1
Cycles: \(1^1\)

Example 1:

RRC.B 0x1233  ; Rotate Right w/C (0x1233) (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1232</td>
<td>E807</td>
</tr>
<tr>
<td>SR</td>
<td>E807</td>
</tr>
<tr>
<td>Data 1232</td>
<td>7407</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
### Example 2:

```
RRC 0x820, WREG ; Rotate Right w/ C (0x820) (Word mode)
; Store result in WREG
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>5601</td>
</tr>
<tr>
<td>Data 0820</td>
<td>216E</td>
</tr>
<tr>
<td>SR</td>
<td>0001 (C = 1)</td>
</tr>
<tr>
<td>WREG (W0)</td>
<td>90B7</td>
</tr>
<tr>
<td>Data 0820</td>
<td>216E</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

### RRC

**Rotate Right Ws through Carry**

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} RRC{.B} Ws, Wd

| Ws | [Ws], [Ws++] | [Ws--], ++Ws |
| Wd | [Wd], [Wd++] | [Wd--], ++Wd |

**Operands:**

Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

**Operation:**

For byte operation:

(C) → Wd<7>
(Ws<7:1>) → Wd<6:0>
(Ws<0>) → C

For word operation:

(C) → Wd<15>
(Ws<15:1>) → Wd<14:0>
(Ws<0>) → C

**Status Affected:**

N, Z, C

**Encoding:**

1101 0011 1Bqq qddd dppp ssss

**Description:**

Rotate the contents of the source register Ws one bit to the right through the Carry flag and place the result in the destination register Wd. The Carry flag of the STATUS Register is shifted into the Most Significant bit of Wd, and it is then overwritten with the Least Significant bit of Ws. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

**Note:**

The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:**

1

**Cycles:**

1(1)

---

**Note 1:**

In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
**Example 1:**

```
RRC.B W0, W3 ; Rotate Right w/ C (W0) (Byte mode)
; Store the result in W3
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>9976</td>
<td>9976</td>
</tr>
<tr>
<td>W3</td>
<td>W3</td>
</tr>
<tr>
<td>5879</td>
<td>58BB</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0001 (C = 1)</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

```
RRC [W2++], [W8] ; Rotate Right w/ C [W2] (Word mode)
; Post-increment W2
; Store result in [W8]
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>2008</td>
<td>200A</td>
</tr>
<tr>
<td>W8</td>
<td>W8</td>
</tr>
<tr>
<td>094E</td>
<td>094E</td>
</tr>
<tr>
<td>Data 094E</td>
<td>Data 094E</td>
</tr>
<tr>
<td>3689</td>
<td>E020</td>
</tr>
<tr>
<td>Data 2008</td>
<td>Data 2008</td>
</tr>
<tr>
<td>C041</td>
<td>C041</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0001 (C = 1)</td>
<td>0009 (N, C = 1)</td>
</tr>
</tbody>
</table>
**Section 5. Instruction Descriptions**

**RRNC**

**Rotate Right f without Carry**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\[
\text{\{label:\}} \ RRNC{.B} f \ \{,WREG\}
\]

**Operands:**

\[f \in [0 \ldots 8191]\]

**Operation:**

For byte operation:

\[(f<7:1>) \rightarrow \text{Dest}<6:0>\]

\[(f<0>) \rightarrow \text{Dest}<7>\]

For word operation:

\[(f<15:1>) \rightarrow \text{Dest}<14:0>\]

\[(f<0>) \rightarrow \text{Dest}<15>\]

**Status Affected:**

N, Z

**Encoding:**

\[
\begin{array}{c}
1101 \\
0111 \\
0BDE \\
\text{ffff} \\
\text{ffff} \\
\text{ffff} \\
\end{array}
\]

**Description:**

Rotate the contents of the file register f one bit to the right and place the result in the destination register. The Least Significant bit of f is stored in the Most Significant bit of the destination, and the Carry flag is not affected.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte). The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register). The ‘f’ bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

**Words:**

1

**Cycles:**

1 (1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.
Example 1:  
RRNC.B 0x1233  ; Rotate Right (0x1233) (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 1232</td>
<td>Data 1232</td>
</tr>
<tr>
<td>E807</td>
<td>7407</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2:  
RRNC 0x820, WREG  ; Rotate Right (0x820) (Word mode)  
; Store result in WREG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0) 5601</td>
<td>WREG (W0) 10B7</td>
</tr>
<tr>
<td>Data 0820 216E</td>
<td>Data 0820 216E</td>
</tr>
<tr>
<td>SR 0001 (C = 1)</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>
RRNC Rotate Right Ws without Carry

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} RRNC{.B} Ws, Wd

- [Ws], [Wd]
- [Ws++], [Wd++]
- [Ws--], [Wd--]
- [++Ws], [++Wd]
- [--Ws], [--Wd]

Operands: Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: For byte operation:

- (Ws<7:1>) → Wd<6:0>
- (Ws<0>) → Wd<7>

For word operation:

- (Ws<15:1>) → Wd<14:0>
- (Ws<0>) → Wd<15>

Status Affected: N, Z

Encoding: 1101 0011 0Bqq qddd dppp ssss

Description: Rotate the contents of the source register Ws one bit to the right and place the result in the destination register Wd. The Least Significant bit of Ws is stored in the Most Significant bit of Wd, and the Carry flag is not affected. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

© 2005-2011 Microchip Technology Inc.
Example 1:  
```
RRNC.B  W0, W3     ; Rotate Right (W0) (Byte mode)
                 ; Store the result in W3
```

**Before Instruction** | **After Instruction**
---|---
W0 | 9976 | W0 | 9976 |
W3 | 5879 | W3 | 583B |
SR | 0001 (C = 1) | SR | 0001 (C = 1) |

Example 2:  
```
RRNC [W2++], [W8]   ; Rotate Right [W2] (Word mode)
                 ; Post-increment W2
                 ; Store result in [W8]
```

**Before Instruction** | **After Instruction**
---|---
W2 | 2008 | W2 | 200A |
W8 | 094E | W8 | 094E |
Data 094E | 3689 | Data 094E | E020 |
Data 2008 | C041 | Data 2008 | C041 |
SR | 0000 | SR | 0008 (N = 1) |
### SAC

**Store Accumulator**

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E
- X
- X
- X

**Syntax:**
{label:} SAC Acc, {#Slit4,} Wd  
\[\text{Wd}\]
\[\text{Wd}++\]
\[\text{Wd}--\]
\[--\text{Wd}\]
\[++\text{Wd}\]
\[\text{Wd} + \text{Wb}\]

**Operands:**
- Acc ∈ [A,B]
- Slit4 ∈ [-8 ... +7]
- Wb, Wd ∈ [W0 ... W15]

**Operation:**
- Shift\_Slit4\_Acc (optional)
- \(\text{Acc}[31:16] \rightarrow \text{Wd}\)

**Status Affected:**
None

**Encoding:**
\[\begin{array}{cccc}
1100 & 1100 & Awww & wrrr \\
 & & rhhh & dddd
\end{array}\]

**Description:**
Perform an optional, signed 4-bit shift of the specified accumulator, then store the shifted contents of ACCxH (Acc[31:16]) to Wd. The shift range is -8:7, where a negative operand indicates an arithmetic left shift and a positive operand indicates an arithmetic right shift. Either register direct or indirect addressing may be used for Wd.

The ‘A’ bit specifies the source accumulator.  
The ‘w’ bits specify the offset register Wb.  
The ‘r’ bits encode the optional accumulator pre-shift.  
The ‘h’ bits select the destination Address mode.  
The ‘d’ bits specify the destination register Wd.

**Note 1:** This instruction does not modify the contents of Acc.
**Note 2:** This instruction stores the truncated contents of Acc. The instruction SAC.R may be used to store the rounded accumulator contents.
**Note 3:** If Data Write saturation is enabled (SATDW, CORCON<5>, = 1), the value stored to Wd is subject to saturation after the optional shift is performed.

**Words:**
1

**Cycles:**
1

**Example 1:**
SAC A, #4, W5  
; Right shift ACCA by 4  
; Store result to W5  
; CORCON = 0x0010 (SATDW = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>B900</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 120F FF00</td>
</tr>
<tr>
<td>CORCON</td>
<td>0010</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Example 2: SAC B, #4, [W5++]
; Left shift ACCB by 4
; Store result to [W5], Post-increment W5
; CORCON = 0x0010 (SATDW = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>2000</td>
</tr>
<tr>
<td>ACCB</td>
<td>FF C891 8F4C</td>
</tr>
<tr>
<td>Data 2000</td>
<td>5BBE</td>
</tr>
<tr>
<td>CORCON</td>
<td>0010</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
**SAC.R**

**Store Rounded Accumulator**

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

**Syntax:**
\{label:\} SAC.R Acc, \{#Slit4,\} Wd

**Operands:**
- Acc ∈ [A,B]
- Slit4 ∈ [-8 ... +7]
- Wb ∈ [W0 ... W15]
- Wd ∈ [W0 ... W15]

**Operation:**
- Shift\_Slit4\(\text{(Acc)}\) (optional)
- Round(Acc)
- \(\text{(Acc}[31:16])\) → Wd

**Status Affected:**
None

**Encoding:**
```
1100 1101 Awww wrrr rhhh dddd
```

**Description:**
Perform an optional, signed 4-bit shift of the specified accumulator, then store the rounded contents of ACCxH (Acc[31:16]) to Wd. The shift range is -8:7, where a negative operand indicates an arithmetic left shift and a positive operand indicates an arithmetic right shift. The Rounding mode (Conventional or Convergent) is set by the RND bit, CORCON<1>. Either register direct or indirect addressing may be used for Wd.

The ‘A’ bit specifies the source accumulator.  
The ‘w’ bits specify the offset register Wb.  
The ‘r’ bits encode the optional accumulator pre-shift.  
The ‘h’ bits select the destination Address mode.  
The ‘d’ bits specify the destination register Wd.

**Note 1:** This instruction does not modify the contents of the Acc.
**Note 2:** This instruction stores the rounded contents of Acc. The instruction SAC may be used to store the truncated accumulator contents.
**Note 3:** If Data Write saturation is enabled (SATDW, CORCON<5>, = 1), the value stored to Wd is subject to saturation after the optional shift is performed.

**Words:** 1
**Cycles:** 1
Example 1: SAC.R A, #4, W5
; Right shift ACCA by 4
; Store rounded result to W5
; CORCON = 0x0010 (SATDW = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>W5</td>
</tr>
<tr>
<td>ACCA</td>
<td>ACCA</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0B900</td>
<td>0121</td>
</tr>
<tr>
<td>00 120F FF00</td>
<td>00 120F FF00</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2: SAC.R B, #-4, [W5++]
; Left shift ACCB by 4
; Store rounded result to [W5], Post-increment W5
; CORCON = 0x0010 (SATDW = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W5</td>
<td>W5</td>
</tr>
<tr>
<td>ACCB</td>
<td>ACCB</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>2000</td>
<td>2002</td>
</tr>
<tr>
<td>FF F891 8F4C</td>
<td>FF F891 8F4C</td>
</tr>
<tr>
<td>5BBE</td>
<td>8919</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
</tr>
<tr>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

SE

**Sign-Extend Ws**

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

{label:} SE Ws, Wnd

- [Ws],
- [Ws++],
- [Ws--],
- [++Ws],
- [--Ws].

Operands:

- Ws ∈ [W0 ... W15]
- Wnd ∈ [W0 ... W15]

Operation:

- Ws<7:0> → Wnd<7:0>

  If (Ws<7> = 1):
  
  - 0xFF → Wnd<15:8>

  Else:
  
  - 0 → Wnd<15:8>

Status Affected:

- N, Z, C

Encoding:

| 1111 | 1011 | 0000 | 0ddd | dppp | ssss |

Description:

Sign-extend the byte in Ws and store the 16-bit result in Wnd. Either register direct or indirect addressing may be used for Ws, and register direct addressing must be used for Wnd. The C flag is set to the complement of the N flag.

The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

**Note 1:** This operation converts a byte to a word, and it uses no .B or .W extension.

**Note 2:** The source Ws is addressed as a byte operand, so any address modification is by ‘1’.

Words: 1

Cycles: 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in **Section 3.2.1 “Multi-Cycle Instructions”**.

**Example 1:** SE W3, W4 ; Sign-extend W3 and store to W4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3</td>
<td>7839</td>
</tr>
<tr>
<td>W4</td>
<td>1005</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### Example 2

```
SE [W2++], W12 ; Sign-extend [W2] and store to W12
; Post-increment W2
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>0900</td>
<td>0901</td>
</tr>
<tr>
<td>W12</td>
<td>W12</td>
</tr>
<tr>
<td>1002</td>
<td>FF8F</td>
</tr>
<tr>
<td>Data 0900</td>
<td>Data 0900</td>
</tr>
<tr>
<td>008F</td>
<td>008F</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0008</td>
</tr>
</tbody>
</table>

(N = 1)
SETM

Set f or WREG

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implemented in:</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} SETM{.B} f

Operands: f ∈ [0 ... 8191]

Operation:

For byte operation:

0xFF → destination designated by D

For word operation:

0xFFFF → destination designated by D

Status Affected: None

Encoding:

| 1110 | 1111 | 1Bdf | ffff | ffff | fff |

Description:

All the bits of the specified register are set to '1'. If WREG is specified, the bits of WREG are set. Otherwise, the bits of the specified file register are set.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘D’ bit selects the destination (‘0’ for WREG, ‘1’ for file register).

The ‘f’ bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Note 2: The WREG is set to working register W0.

Words: 1

Cycles: 1

Example 1: SETM.B 0x891 ; Set 0x891 (Byte mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 0890</td>
<td>Data 0890</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
<tr>
<td>0890 FF39</td>
<td>0890 FF39</td>
</tr>
</tbody>
</table>

Example 2: SETM WREG ; Set WREG (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
<tr>
<td>0900 FFFF</td>
<td>0900 FFFF</td>
</tr>
</tbody>
</table>

© 2005-2011 Microchip Technology Inc.
### SETM

**Set Ws**

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```
{label:} SETM{{.B} Wd
[Wd]
[Wd++]
[Wd--]
[++Wd]
[--Wd]
```

**Operands:**

\[Wd \in [W0 ... W15]\]

**Operation:**

- **For byte operation:**
  - \(0xFF \rightarrow Wd\) for byte operation

- **For word operation:**
  - \(0xFFFF \rightarrow Wd\) for word operation

**Status Affected:**

None

**Encoding:**

```
1110 1011 1Bqq qddd d000 0000
```

**Description:**

All the bits of the specified register are set to ‘1’. Either register direct or indirect addressing may be used for Wd.

The ‘B’ bits selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

**Note:**

The extension ‘.B’ in the instruction denotes a byte operation rather than a word operation. You may use a ‘.W’ extension to denote a word operation, but it is not required.

**Words:**

1

**Cycles:**

1

**Example 1:**

```
SETM .B  W13 ; Set W13 (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W13</td>
<td>2739</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
SETM [--W6] ; Pre-decrement W6 (Word mode)
```

```
; Set [--W6]
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>1250</td>
</tr>
<tr>
<td>Data 124E</td>
<td>3CD9</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
SFTAC  
Arithmetic Shift Accumulator by Slit6

Implemented in:  
<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Syntax:  
{label:} SFTAC Acc, #Slit6

Operands:  
Acc ∈ [A,B]  
Slit6 ∈ [-16 ... 16]

Operation:  
Shift<6>(Acc) → Acc

Status Affected:  
OA, OB, OAB, SA, SB, SAB

Encoding:  
\[
\begin{array}{cccc}
1100 & 1000 & A000 & 0000 & 01kk & kkkk \\
\end{array}
\]

Description:  
Arithmetic shift the 40-bit contents of the specified accumulator by the signed, 6-bit literal and store the result back into the accumulator. The shift range is -16:16, where a negative operand indicates a left shift and a positive operand indicates a right shift. Any bits which are shifted out of the accumulator are lost.

The 'A' bit selects the accumulator for the result. The 'k' bits determine the number of bits to be shifted.

**Note 1:** If saturation is enabled for the target accumulator (SATA, CORCON<7> or SATB, CORCON<6>), the value stored to the accumulator is subject to saturation.

**Note 2:** If the shift amount is greater than 16 or less than -16, no modification will be made to the accumulator, and an arithmetic trap will occur.

Words:  
1

Cycles:  
1

**Example 1:**  
SFTAC A, #12  
; Arithmetic right shift ACCA by 12  
; Store result to ACCA  
; CORCON = 0x0080 (SATA = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCA 00 120F FF00</td>
<td>ACCA 00 0001 20FF</td>
</tr>
<tr>
<td>CORCON 0080</td>
<td>CORCON 0080</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**  
SFTAC B, #-10  
; Arithmetic left shift ACCB by 10  
; Store result to ACCB  
; CORCON = 0x0040 (SATB = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCB FF FFF1 8F4C</td>
<td>ACCB FF C63D 3000</td>
</tr>
<tr>
<td>CORCON 0040</td>
<td>CORCON 0040</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
## SFTAC

### Arithmetic Shift Accumulator by Wb

**Implemented in:**

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} SFTAC Acc, Wb

**Operands:**

Acc ∈ [A,B]

Wb ∈ [W0 ... W15]

**Operation:**

\( \text{Shift}_{Wb}(\text{Acc}) \rightarrow \text{Acc} \)

**Status Affected:** OA, OB, OAB, SA, SB, SAB

**Encoding:**

| 1100 | 1000 | A000 | 0000 | 0000 | ssss |

**Description:**

Arithmetic shift the 40-bit contents of the specified accumulator and store the result back into the accumulator. The Least Significant 6 bits of Wb are used to specify the shift amount. The shift range is -16:16, where a negative value indicates a left shift and a positive value indicates a right shift. Any bits which are shifted out of the accumulator are lost.

The 'A' bit selects the accumulator for the source/destination. The 's' bits select the address of the shift count register.

**Note 1:** If saturation is enabled for the target accumulator (SATA, CORCON<7> or SATB, CORCON<6>), the value stored to the accumulator is subject to saturation.

**Note 2:** If the shift amount is greater than 16 or less than -16, no modification will be made to the accumulator, and an arithmetic trap will occur.

**Words:** 1

**Cycles:** 1

### Example 1:

SFTAC A, W0

; Arithmetic shift ACCA by (W0)

; Store result to ACCA

; CORCON = 0x0000 (saturation disabled)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>ACCA</td>
<td>ACCA</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>FFFC</td>
</tr>
<tr>
<td>ACCA</td>
<td>00 320F AB09</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

### Example 2:

SFTAC B, W12

; Arithmetic shift ACCB by (W12)

; Store result to ACCB

; CORCON = 0x0040 (SATB = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W12</td>
<td>W12</td>
</tr>
<tr>
<td>ACCB</td>
<td>ACCB</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W12</td>
<td>000F</td>
</tr>
<tr>
<td>ACCB</td>
<td>FF FFF1 8F4C</td>
</tr>
<tr>
<td>CORCON</td>
<td>0040</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W12</td>
<td>W12</td>
</tr>
<tr>
<td>ACCB</td>
<td>ACCB</td>
</tr>
<tr>
<td>CORCON</td>
<td>CORCON</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W12</td>
<td>FFFFC</td>
</tr>
<tr>
<td>ACCB</td>
<td>FF FFFF FFE3</td>
</tr>
<tr>
<td>CORCON</td>
<td>0040</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

### SL

**Shift Left f**

**Implemented in:**
- PIC24F
- PIC24H
- PIC24E
- dsPIC30F
- dsPIC33F
- dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**
{label:} SL{.B} f {,WREG}

**Operands:**
- f ∈ [0... 8191]

**Operation:**

- **For byte operation:**
  - (f<7>) → (C)
  - (f<6:0>) → Dest<7:1>
  - 0 → Dest<0>

- **For word operation:**
  - (f<15>) → (C)
  - (f<14:0>) → Dest<15:1>
  - 0 → Dest<0>

**Status Affected:** N, Z, C

**Encoding:**

1101 0100 0BDf ffff ffff ffff

**Description:**
Shift the contents of the file register one bit to the left and place the result in the destination register. The Most Significant bit of the file register is shifted into the Carry bit of the STATUS register, and zero is shifted into the Least Significant bit of the destination register.

The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

**Words:** 1

**Cycles:** 1 (1)

**Example 1:**

```
SL.B 0x909   ; Shift left (0x909) (Byte mode)
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data 0908</td>
<td>0839</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

© 2005-2011 Microchip Technology Inc. DS70157F-page 399
### Example 2:

```
SL 0x1650, WREG ; Shift left (0x1650) (Word mode)
; Store result in WREG
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WREG (W0)</strong></td>
<td><strong>WREG (W0)</strong></td>
</tr>
<tr>
<td>0900</td>
<td>80CA</td>
</tr>
<tr>
<td><strong>Data 1650</strong></td>
<td><strong>Data 1650</strong></td>
</tr>
<tr>
<td>4065</td>
<td>4065</td>
</tr>
<tr>
<td><strong>SR</strong></td>
<td><strong>SR</strong></td>
</tr>
<tr>
<td>0000</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
**SL**  
Shift Left Ws

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
```
{label:} SL{.B} Ws, Wd  
[Ws], [Wd]  
[Ws++], [Wd++]  
[Ws--], [Wd--]  
[++Ws], [++Wd]  
[--Ws], [--Wd]
```

Operands:  
Ws ∈ [W0 ... W15]  
Wd ∈ [W0 ... W15]

Operation:  
For byte operation:  
\[(Ws<7>) \rightarrow C\]  
\[(Ws<6:0>) \rightarrow Wd<7:1>\]  
0 \rightarrow Wd<0>  
For word operation:  
\[(Ws<15>) \rightarrow C\]  
\[(Ws<14:0>) \rightarrow Wd<15:1>\]  
0 \rightarrow Wd<0>

Status Affected:  
N, Z, C

Encoding:  
```
1101 0000 0Bqq qddd dppp ssss
```

Description:  
Shift the contents of the source register Ws one bit to the left and place the result in the destination register Wd. The Most Significant bit of Ws is shifted into the Carry bit of the STATUS register, and ‘0’ is shifted into the Least Significant bit of Wd. Either register direct or indirect addressing may be used for Ws and Wd.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words:  
1

Cycles:  
1(1)

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.
Example 1: SL.B W3, W4 ; Shift left W3 (Byte mode)
; Store result to W4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3 78A9</td>
<td>W3 78A9</td>
</tr>
<tr>
<td>W4 1005</td>
<td>W4 1052</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>

; Store result to [W12]
; Post-increment W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 0900</td>
<td>W2 0902</td>
</tr>
<tr>
<td>W12 1002</td>
<td>W12 1002</td>
</tr>
<tr>
<td>Data 0900 800F</td>
<td>Data 0900 800F</td>
</tr>
<tr>
<td>Data 1002 6722</td>
<td>Data 1002 001E</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>
**SL**  
**Shift Left by Short Literal**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**  
{label:} SL Wb, #lit4, Wnd

**Operands:**  
Wb ∈ [W0 ... W15]  
lit4 ∈ [0...15]  
Wnd ∈ [W0 ... W15]

**Operation:**  
lit4<3:0> → Shift_Val  
Wnd<15:Shift_Val> = Wb<15-Shift_Val:0>  
Wd<Shift_Val – 1:0> = 0

**Status Affected:** N, Z

**Encoding:**  
| 1101 | 1101 | 0www | wddd | d100 | kkkk |

**Description:** Shift left the contents of the source register Wb by the 4-bit unsigned literal and store the result in the destination register Wnd. Any bits shifted out of the source register are lost. Direct addressing must be used for Wb and Wnd.

The 'w' bits select the address of the base register.  
The 'd' bits select the destination register.  
The 'k' bits provide the literal operand, a five-bit integer number.  

**Note:** This instruction operates in Word mode only.

**Words:** 1  
**Cycles:** 1

**Example 1:**  
SL W2, #4, W2 ; Shift left W2 by 4  
; Store result to W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2 78A9</td>
<td>W2 8A90</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008</td>
</tr>
</tbody>
</table>

(N = 1)

**Example 2:**  
SL W3, #12, W8 ; Shift left W3 by 12  
; Store result to W8

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3 0912</td>
<td>W3 0912</td>
</tr>
<tr>
<td>W8 1002</td>
<td>W8 2000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
### SL  
**Shift Left by Wns**

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} SL Wb, Wns, Wnd

**Operands:**

- Wb ∈ [W0 ... W15]
- Wns ∈ [W0 ... W15]
- Wnd ∈ [W0 ... W15]

**Operation:**

- Wns<4:0> → Shift_Val
- Wnd<15:Shift_Val> = Wb<15 – Shift_Val:0>
- Wd<Shift_Val – 1:0> = 0

**Status Affected:** N, Z

**Encoding:**

| 1101 | 1101 | 0www | wddd | d000 | ssss |

**Description:**

Shift left the contents of the source register Wb by the 5 Least Significant bits of Wns (only up to 15 positions) and store the result in the destination register Wnd. Any bits shifted out of the source register are lost. Register direct addressing must be used for Wb, Wns and Wnd.

The 'w' bits select the address of the base register.
The 'd' bits select the destination register.
The 's' bits select the source register.

**Note 1:** This instruction operates in Word mode only.

**Note 2:** If Wns is greater than 15, Wnd will be loaded with 0x0.

**Words:** 1  
**Cycles:** 1

**Example 1:**

```assembly
SL W0, W1, W2 ; Shift left W0 by W1<0:4>
; Store result to W2
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 09A4</td>
<td>W0 09A4</td>
</tr>
<tr>
<td>W1 8903</td>
<td>W1 8903</td>
</tr>
<tr>
<td>W2 78A9</td>
<td>W2 4D20</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```assembly
SL W4, W5, W6 ; Shift left W4 by W5<0:4>
; Store result to W6
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 A409</td>
<td>W4 A409</td>
</tr>
<tr>
<td>W5 FF01</td>
<td>W5 FF01</td>
</tr>
<tr>
<td>W6 0883</td>
<td>W6 4812</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
### Section 5. Instruction Descriptions

#### SUB

**Subtract WREG from f**

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label:\} SUB{.B} f \{.,WREG\}

**Operands:**  

\( f \in [0 \ldots 8191] \)

**Operation:**  

\((f) – (\text{WREG}) \rightarrow \text{destination designated by D}\)

**Status Affected:**  

DC, N, OV, Z, C

**Encoding:**

| 1011 | 0101 | 0BDF | ffff | ffff | ffff |

**Description:** Subtract the contents of the default working register WREG from the contents of the specified file register, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).  
The 'D' bit selects the destination ('0' for WREG, '1' for file register).  
The 'f' bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

**Words:** 1  

**Cycles:** 1

**Example 1:**

SUB.B 0x1FFF ; Sub. WREG from (0x1FFF) (Byte mode)  
; Store result to 0x1FFF

Before Instruction  

<table>
<thead>
<tr>
<th>WREG (W0)</th>
<th>Data 1FFE</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7804</td>
<td>9439</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction  

<table>
<thead>
<tr>
<th>WREG (W0)</th>
<th>Data 1FFE</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7804</td>
<td>9039</td>
<td>0001</td>
</tr>
</tbody>
</table>

\(C = 1\)

**Example 2:**

SUB 0xA04, WREG ; Sub. WREG from (0xA04) (Word mode)  
; Store result to WREG

Before Instruction  

<table>
<thead>
<tr>
<th>WREG (W0)</th>
<th>Data 0A04</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>6234</td>
<td>4523</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction  

<table>
<thead>
<tr>
<th>WREG (W0)</th>
<th>Data 0A04</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2EF</td>
<td>4523</td>
<td>0008</td>
</tr>
</tbody>
</table>

\(N = 1\)
SUB
Subtract Literal from Wn

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

\{label:\} SUB{.B} #lit10, Wn

Operands:

lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation:

(Wn) – lit10 → Wn

Status Affected:
DC, N, OV, Z, C

Encoding:

```
1011 0001 0Bkk kkkk kkkk dddd
```

Description:

Subtract the 10-bit unsigned literal operand from the contents of the working register Wn, and store the result back in the working register Wn. Register direct addressing must be used for Wn.

The ‘B’ bit selects byte or word operation.
The ‘k’ bits specify the literal operand.
The ‘d’ bits select the address of the working register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Note 2: For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

Words: 1
Cycles: 1

Example 1:

```
SUB.B #0x23, W0    ; Sub. 0x23 from W0 (Byte mode)
; Store result to W0
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 7804</td>
<td>W0 78E1</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0008 (N = 1)</td>
</tr>
</tbody>
</table>

Example 2:

```
SUB #0x108, W4   ; Sub. 0x108 from W4 (Word mode)
; Store result to W4
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 6234</td>
<td>W4 612C</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001 (C = 1)</td>
</tr>
</tbody>
</table>
### SUB

**Subtract Short Literal from Wb**

*Implemented in:* PIC24F  |  PIC24H  |  PIC24E  | dsPIC30F | dsPIC33F | dsPIC33E  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label:} SUB{B} Wb, #lit5, Wd

- [Wd]
- [Wd++]
- [Wd--]
- [++Wd]
- [--Wd]

**Operands:**

- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wd ∈ [W0 ... W15]

**Operation:**

(Wb) – lit5 → Wd

**Status Affected:** DC, N, OV, Z, C

**Encoding:**

0101 0www wBqq qddd d11k kkkk

**Description:** Subtract the 5-bit unsigned literal operand from the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing must be used for Wd.

The ‘w’ bits select the address of the base register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘k’ bits provide the literal operand, a five-bit integer number.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1  
**Cycles:** 1

**Example 1:**

SUB.B W4, #0x10, W5  
; Sub. 0x10 from W4 (Byte mode)  
; Store result to W5

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 1782</td>
<td>W4 1782</td>
</tr>
<tr>
<td>W5 7804</td>
<td>W5 7872</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0005 (OV, C = 1)</td>
</tr>
</tbody>
</table>

**Example 2:**

SUB W0, #0x8, [W2++]  
; Sub. 0x8 from W0 (Word mode)  
; Store result to [W2]  
; Post-increment W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 F230</td>
<td>W0 F230</td>
</tr>
<tr>
<td>W2 2004</td>
<td>W2 2006</td>
</tr>
<tr>
<td>Data 2004 A557</td>
<td>Data 2004 F228</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0009 (N, C = 1)</td>
</tr>
</tbody>
</table>
### SUB

**Subtract Ws from Wb**

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\[
\{\text{label:}\} \text{SUB}\{.B\} \ Wb, \ Ws, \ Wd \\
\quad [Ws], \ [Wd] \\
\quad [Ws+], \ [Wd+] \\
\quad [Ws-], \ [Wd-] \\
\quad [++Ws], \ [++Wd] \\
\quad [-Ws], \ [--Wd]
\]

**Operands:**

- \(Wb\) ∈ [W0 ... W15]
- \(Ws\) ∈ [W0 ... W15]
- \(Wd\) ∈ [W0 ... W15]

**Operation:**

\((Wb) - (Ws) \rightarrow Wd\)

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

| 0101 | 0www | wBqq | qddd | dppp | ssss |

**Description:**

Subtract the contents of the source register \(Ws\) from the contents of the base register \(Wb\) and place the result in the destination register \(Wd\). Register direct addressing must be used for \(Wb\). Either register direct or indirect addressing may be used for \(Ws\) and \(Wd\).

The ‘w’ bits select the address of the base register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note:** The extension \(.B\) in the instruction denotes a byte operation rather than a word operation. You may use a \(.W\) extension to denote a word operation, but it is not required.

**Words:**

1

**Cycles:**

1\(^{(1)}\)

---

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:**

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(W0) 1732</td>
<td>(W0) 17EE</td>
</tr>
<tr>
<td>(W1) 7844</td>
<td>(W1) 7844</td>
</tr>
<tr>
<td>(SR) 0000</td>
<td>(SR) 0108 (DC, N = 1)</td>
</tr>
</tbody>
</table>
Example 2: SUB W7, [W8++], [W9++] ; Sub. [W8] from W7 (Word mode)
; Store result to [W9]
; Post-increment W8
; Post-increment W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7 2450</td>
<td>W7 2450</td>
</tr>
<tr>
<td>W8 1808</td>
<td>W8 180A</td>
</tr>
<tr>
<td>W9 2020</td>
<td>W9 2022</td>
</tr>
<tr>
<td>Data 1808 92E4</td>
<td>Data 1808 92E4</td>
</tr>
<tr>
<td>Data 2020 A557</td>
<td>Data 2020 916C</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 010C (DC, N, OV = 1)</td>
</tr>
</tbody>
</table>
### SUB

**Subtract Accumulators**

**Implemented in:**

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} SUB Acc

**Operands:**

Acc ∈ [A,B]

**Operation:**

If (Acc = A):

ACCA – ACCB → ACCA

Else:

ACCB – ACCA → ACCB

**Status Affected:**

OA, OB, OAB, SA, SB, SAB

**Encoding:**

| 1100 | 1011 | A011 | 0000 | 0000 | 0000 |

**Description:**

Subtract the contents of the unspecified accumulator from the contents of Acc, and store the result back into Acc. This instruction performs a 40-bit subtraction.

The 'A' bit specifies the destination accumulator.

**Words:**

1

**Cycles:**

1

**Example 1:**

SUB A ; Subtract ACCB from ACCA
; Store the result to ACCA
; CORCON = 0x0000 (no saturation)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCA</td>
<td>76 120F 098A</td>
</tr>
<tr>
<td>ACCB</td>
<td>23 F312 BC17</td>
</tr>
<tr>
<td>CORCON</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

SUB B ; Subtract ACCA from ACCB
; Store the result to ACCB
; CORCON = 0x0040 (SATB = 1)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCA</td>
<td>FF 9022 2EE1</td>
</tr>
<tr>
<td>ACCB</td>
<td>00 2456 8F4C</td>
</tr>
<tr>
<td>CORCON</td>
<td>0040</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
SUBB Subtract WREG and Carry bit from f

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: `{label:} SUBB{.B} f {,WREG}`

Operands: $f \in [0 ... 8191]$

Operation: $(f) - (\text{WREG}) - (\overline{C}) \rightarrow \text{destination designated by D}$

Status Affected: DC, N, OV, Z, C

Encoding:

| 1011 | 0101 | 1BDf | ffff | ffff | ffff |

Description:

Subtract the contents of the default working register WREG and the Borrow flag (Carry flag inverse, $\overline{C}$) from the contents of the specified file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

3: The Z flag is “sticky” for `ADD`, `CPB`, `SUBB` and `SUBBR`. These instructions can only clear Z.

Words: 1

Cycles: 1

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

```
SUBB.B 0x1FFF ; Sub. WREG and C from (0x1FFF) (Byte mode)
; Store result to 0x1FFF
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0) 7804</td>
<td>WREG (W0) 7804</td>
</tr>
<tr>
<td>Data 1FFE</td>
<td>Data 1FFE</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0011</td>
</tr>
</tbody>
</table>

Example 2:

```
SUBB 0xA04, WREG ; Sub. WREG and C from (0xA04) (Word mode)
; Store result to WREG
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0) 6234</td>
<td>WREG (W0) 0000</td>
</tr>
<tr>
<td>Data 0A04</td>
<td>Data 0A04</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0001</td>
</tr>
</tbody>
</table>

(C = 1)
**SUBB**  Subtract Wn from Literal with Borrow

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

{label:} SUBB(.B) #lit10, Wn

Operands:

lit10 ∈ [0 ... 255] for byte operation
lit10 ∈ [0 ... 1023] for word operation
Wn ∈ [W0 ... W15]

Operation:

(Wn) – lit10 – (C \_1) → Wn

Status Affected:

DC, N, OV, Z, C

Encoding:

| 1011 | 0001 | 1Bkk | kkkk | kkkk | dddd |

Description:

Subtract the unsigned 10-bit literal operand and the Borrow flag (Carry flag inverse, C\_1) from the contents of the working register Wn, and store the result back in the working register Wn. Register direct addressing must be used for Wn.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘k’ bits specify the literal operand.

The ‘d’ bits select the address of the working register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .w extension to denote a word operation, but it is not required.

2: For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

3: The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1
Cycles: 1

Example 1: SUBB.B #0x23, W0 ; Sub. 0x23 and C\_1 from W0 (Byte mode) ; Store result to W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>SR</td>
</tr>
<tr>
<td>78E0</td>
<td>0108</td>
</tr>
</tbody>
</table>

Example 2: SUBB #0x108, W4 ; Sub. 0x108 and C\_1 from W4 (Word mode) ; Store result to W4

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>SR</td>
</tr>
<tr>
<td>61DC</td>
<td>0001</td>
</tr>
</tbody>
</table>
### SUBB

**Subtract Short Literal from Wb with Borrow**

#### Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

```
{label;} SUBB{.B} Wb, #lit5, Wd
```

- [Wd]
- [Wd++]
- [Wd--]
- [++Wd]
- [--Wd]

#### Operands:

- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wd ∈ [W0 ... W15]

#### Operation:

\[(Wb) - \text{lit5} - (\overline{C}) \rightarrow Wd\]

#### Status Affected:

DC, N, OV, Z, C

#### Encoding:

```
0101 lwww wBqq qddd d11k kkkk
```

#### Description:

Subtract the 5-bit unsigned literal operand and the Borrow flag (Carry flag inverse, \(\overline{C}\)) from the contents of the base register Wb and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

- The ‘w’ bits select the address of the base register.
- The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
- The ‘q’ bits select the destination Address mode.
- The ‘d’ bits select the destination register.
- The ‘k’ bits provide the literal operand, a five-bit integer number.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

#### Words:

1

#### Cycles:

1

#### Example 1:

```assembly
SUBB.B W4, #0x10, W5 ; Sub. 0x10 and \(\overline{C}\) from W4 (Byte mode)
; Store result to W5
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 1782</td>
<td>W4 1782</td>
</tr>
<tr>
<td>W5 7804</td>
<td>W5 7871</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0005</td>
</tr>
</tbody>
</table>

\(\text{OV, C = 1}\)
Example 2: SUBB W0, #0x8, [W2++] ; Sub. 0x8 and C from W0 (Word mode)
; Store result to [W2]
; Post-increment W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0009</td>
<td>W0 0009</td>
</tr>
<tr>
<td>W2 2004</td>
<td>W2 2006</td>
</tr>
<tr>
<td>Data 2004 A557</td>
<td>Data 2004 0000</td>
</tr>
<tr>
<td>SR 0002 (Z = 1)</td>
<td>SR 0103 (DC, Z, C = 1)</td>
</tr>
</tbody>
</table>
Section 5. Instruction Descriptions

SUBB

Subtract Ws from Wb with Borrow

Implemented in: PIC24F PIC24H PIC24E dsPIC30F dsPIC33F dsPIC33E

X X X X X X

Syntax: {label;} SUBB{.B} Wb, Ws, Wd

[Ws], [Wd]
[Ws++], [Wd++]
[Ws--], [Wd--]
[++Ws], [++Wd]
[--Ws], [--Wd]

Operands: Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation: (Wb) – (Ws) – (C) → Wd

Status Affected: DC, N, OV, Z, C

Encoding: 0101 1www wBqq qddd dppp ssss

Description: Subtract the contents of the source register Ws and the Borrow flag (Carry flag inverse, C) from the contents of the base register Wb, and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The Z flag is “sticky” for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: SUBB.B W0, W1, W0 ; Sub. W1 and C from W0 (Byte mode) ; Store result to W0

Before Instruction | After Instruction
--- | ---
W0 1732 | W0 17ED
W1 7844 | W1 7844
SR 0000 | SR 0108 (DC, N = 1)
Example 2: \texttt{SUBB W7,\[W8++\],\[W9++\]} ; Sub. \([W8]\) and \(?\) from \(W7\) (Word mode) 
; Store result to \([W9]\) 
; Post-increment \(W8\) 
; Post-increment \(W9\)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7 2450</td>
<td>W7 2450</td>
</tr>
<tr>
<td>W8 1808</td>
<td>W8 180A</td>
</tr>
<tr>
<td>W9 2022</td>
<td>W9 2024</td>
</tr>
<tr>
<td>Data 1808 92E4</td>
<td>Data 1808 92E4</td>
</tr>
<tr>
<td>Data 2022 A557</td>
<td>Data 2022 916B</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 010C ((DC, N, OV = 1))</td>
</tr>
</tbody>
</table>
SUBBR

Subtract f from WREG with Borrow

Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: {label:} SUBBR{.B} f {,WREG}

Operands: f ∈ [0 ... 8191]

Operation: (WREG) – (f) – (C) → destination designated by D

Status Affected: DC, N, OV, Z, C

Encoding:

```
1011 1101 1BDf ffff ffff ffff
```

Description: Subtract the contents of the specified file register f and the Borrow flag (Carry flag inverse, C) from the contents of WREG, and place the result in the destination register. The optional WREG operand designates the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

3: The Z flag is "sticky" for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1

Cycles: 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.

**Example 1:** SUBBR.B 0x803 ; Sub. (0x803) and C from WREG (Byte mode)

; Store result to 0x803

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>7804</td>
</tr>
<tr>
<td>Data 0802</td>
<td>9439</td>
</tr>
<tr>
<td>SR</td>
<td>0002 (Z = 1)</td>
</tr>
<tr>
<td>Data 0802</td>
<td>6F39</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:** SUBBR 0xA04, WREG ; Sub. (0xA04) and C from WREG (Word mode)

; Store result to WREG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>6234</td>
</tr>
<tr>
<td>Data 0A04</td>
<td>6235</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>WREG (W0)</td>
<td>FFFE</td>
</tr>
<tr>
<td>Data 0A04</td>
<td>6235</td>
</tr>
<tr>
<td>SR</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
**SUBBR**

**Subtract Wb from Short Literal with Borrow**

- **Implemented in:** PIC24F, PIC24H, PIC24E, dsPIC30F, dsPIC33F, dsPIC33E
- **Syntax:** `{label:} SUBBR{.B} Wb, #lit5, Wd
  - [Wd]
  - [Wd++]
  - [Wd--]
  - [++Wd]
  - [--Wd]

- **Operands:**
  - Wb ∈ [W0 ... W15]
  - lit5 ∈ [0 ... 31]
  - Wd ∈ [W0 ... W15]

- **Operation:**
  - lit5 – (Wb) – (C) → Wd

- **Status Affected:** DC, N, OV, Z, C

- **Encoding:**
  - 0001 lwww wBqq qddd d1lk kkkk

- **Description:** Subtract the contents of the base register Wb and the Borrow flag (Carry flag inverse, Č) from the 5-bit unsigned literal and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing must be used for Wd.

  The 'w' bits select the address of the base register.
  The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
  The 'q' bits select the destination Address mode.
  The 'd' bits select the destination register.
  The 'k' bits provide the literal operand, a five-bit integer number.

**Note 1:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Note 2:** The Z flag is "sticky" for ADC, CPB, SUBB and SUBBR. These instructions can only clear Z.

- **Words:** 1
- **Cycles:** 1

**Example 1:** SUBBR.B W0, #0x10, W1 ; Sub. W0 and Č from 0x10 (Byte mode)
  ; Store result to W1

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>W0</td>
</tr>
<tr>
<td>W1</td>
<td>786A</td>
</tr>
<tr>
<td>SR</td>
<td>0003</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Status Register</th>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0003</td>
<td>0103</td>
</tr>
</tbody>
</table>

(With Z sticky after SUBB)
### Example 2: 
```
SUBBR W0, #0x8, [W2++] ; Sub. W0 and C from 0x8 (Word mode)
; Store result to [W2]
; Post-increment W2
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0009</td>
</tr>
<tr>
<td>W2</td>
<td>2004</td>
</tr>
<tr>
<td>Data 2004</td>
<td>A557</td>
</tr>
<tr>
<td>SR</td>
<td>0020 (Z = 1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0009</td>
</tr>
<tr>
<td>W2</td>
<td>2004</td>
</tr>
<tr>
<td>Data 2004</td>
<td>FFFE</td>
</tr>
<tr>
<td>SR</td>
<td>0108 (DC, N = 1)</td>
</tr>
</tbody>
</table>
SUBBR

Subtract Wb from Ws with Borrow

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} SUBBR{.B} Wb, Ws, Wd

- [Ws], [Wd]
- [Ws++], [Wd++]
- [Ws--], [Wd--]
- [++Ws], [++Wd]
- [--Ws], [--Wd]

Operands:

Wb ∈ [W0 ... W15]
Ws ∈ [W0 ... W15]
Wd ∈ [W0 ... W15]

Operation:

(Ws) – (Wb) – (C) → Wd

Status Affected:

DC, N, OV, Z, C

Encoding:

0001 lwww wBqq qddd dppp ssss

Description:

Subtract the contents of the base register Wb and the Borrow flag (Carry flag inverse, C) from the contents of the source register Ws and place the result in the destination register Wd. Register direct addressing must be used for Wb. Register direct or indirect addressing may be used for Ws and Wd.

The ‘w’ bits select the address of the base register.
The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).
The ‘q’ bits select the destination Address mode.
The ‘d’ bits select the destination register.
The ‘p’ bits select the source Address mode.
The ‘s’ bits select the source register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The Z flag is "sticky" for ADDC, CPB, SUBB and SUBBR. These instructions can only clear Z.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1:

SUBBR.B W0, W1, W0 ; Sub. W0 and C from W1 (Byte mode)
; Store result to W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>1732</td>
</tr>
<tr>
<td>W1</td>
<td>7844</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>1711</td>
</tr>
<tr>
<td></td>
<td>7844</td>
</tr>
<tr>
<td></td>
<td>0001</td>
</tr>
</tbody>
</table>

(C = 1)
### Example 2: SUBBR W7,[W8++],[W9++]

Sub. W7 and \( \overline{C} \) from [W8] (Word mode)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7 2450</td>
<td>W7 2450</td>
</tr>
<tr>
<td>W8 1808</td>
<td>W8 180A</td>
</tr>
<tr>
<td>W9 2022</td>
<td>W9 2024</td>
</tr>
</tbody>
</table>

Data 1808 92E4
Data 2022 A557
SR 0000

Data 1808 92E4
Data 2022 6E93
SR 0005 (OV, C = 1)
**SUBR Subtract f from WREG**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\{label: \} SUBR{.B} f \{.WREG\}

**Operands:**

\( f \in [0 ... 8191] \)

**Operation:**

\((WREG) - (f) \rightarrow \text{destination designated by D}\)

**Status Affected:**

DC, N, OV, Z, C

**Encoding:**

```
1011 1101 0BDf ffff ffff ffff
```

**Description:** Subtract the contents of the specified file register from the contents of the default working register WREG, and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'D' bit selects the destination ('0' for WREG, '1' for file register).

The 'f' bits select the address of the file register.

**Note 1:** The extension \(.B\) in the instruction denotes a byte operation rather than a word operation. You may use a \(.W\) extension to denote a word operation, but it is not required.

**Note 2:** The WREG is set to working register W0.

**Words:** 1

**Cycles:** 1\(^{(1)}\)

---

**Example 1:**

SUBR.B 0x1FFF ; Sub. (0x1FFF) from WREG (Byte mode)

; Store result to 0x1FFF

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>7804</td>
</tr>
<tr>
<td>Data 1FFE</td>
<td>9439</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>7804</td>
</tr>
<tr>
<td></td>
<td>7039</td>
</tr>
<tr>
<td></td>
<td>0000</td>
</tr>
</tbody>
</table>

---

**Example 2:**

SUBR 0xA04, WREG ; Sub. (0xA04) from WREG (Word mode)

; Store result to WREG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>6234</td>
</tr>
<tr>
<td>Data 0A04</td>
<td>6235</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>FFFF</td>
</tr>
<tr>
<td></td>
<td>6235</td>
</tr>
<tr>
<td></td>
<td>0000 (N = 1)</td>
</tr>
</tbody>
</table>

---

\(^{(1)}\) In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in Section 3.2.1 “Multi-Cycle Instructions”.
SUBR

Subtract Wb from Short Literal

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

{label:} SUBR(B) Wb, #lit5 Wd

Operands:

- Wb ∈ [W0 ... W15]
- lit5 ∈ [0 ... 31]
- Wd ∈ [W0 ... W15]

Operation:

lit5 – (Wb) → Wd

Status Affected:
DC, N, OV, Z, C

Encoding:

| 0001 | 0www | wBqq | qddd | d11k | kkkk |

Description:

Subtract the contents of the base register Wb from the unsigned 5-bit literal operand, and place the result in the destination register Wd.

Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the destination register.
The 'k' bits provide the literal operand, a five-bit integer number.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1

Example 1:

SUBR.B W0, #0x10, W1 ; Sub. W0 from 0x10 (Byte mode)
; Store result to W1

Before Instruction

<table>
<thead>
<tr>
<th>W0</th>
<th>W1</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>F310</td>
<td>786A</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>W0</th>
<th>W1</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>F310</td>
<td>7800</td>
<td>0108</td>
</tr>
</tbody>
</table>

Example 2:

SUBR W0, #0x8, [W2++] ; Sub. W0 from 0x8 (Word mode)
; Store result to [W2]
; Post-increment W2

Before Instruction

<table>
<thead>
<tr>
<th>W0</th>
<th>W2</th>
<th>Data 2004</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0009</td>
<td>2004</td>
<td>A557</td>
<td>0000</td>
</tr>
</tbody>
</table>

After Instruction

<table>
<thead>
<tr>
<th>W0</th>
<th>W2</th>
<th>Data 2004</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0009</td>
<td>2006</td>
<td>FFFF</td>
<td>0108</td>
</tr>
</tbody>
</table>

© 2005-2011 Microchip Technology Inc.
DS70157F-page 423
SUBR Subtract Wb from Ws

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
| X | X | X | X | X | X |

Syntax: {label:} SUBR{.B} Wb, Ws, Wd

- [Ws], [Wd]
- [Ws++], [Wd++]
- [Ws--], [Wd--]
- [++Ws], [++Wd]
- [--Ws], [--Wd]

Operands:
- Wb ∈ [W0 ... W15]
- Ws ∈ [W0 ... W15]
- Wd ∈ [W0 ... W15]

Operation: (Ws) – (Wb) → Wd

Status Affected: DC, N, OV, Z, C

Encoding:

| 0001 | 0www | wBqq | qddd | dppp | ssss |

Description: Subtract the contents of the base register Wb from the contents of the source register Ws and place the result in the destination register Wd. Register direct addressing must be used for Wb. Either register direct or indirect addressing may be used for Ws and Wd.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the source register.

Note: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 "Multi-Cycle Instructions".

Example 1: SUBR.B W0, W1, W0 ; Sub. W0 from W1 (Byte mode)
; Store result to W0

Before Instruction | After Instruction
|---|---|
W0 | 1732 | W0 | 1712 |
W1 | 7844 | W1 | 7844 |
SR | 0000 | SR | 0001 (C = 1)
Example 2: SUBR W7, [W8++], [W9++] ; Sub. W7 from [W8] (Word mode)
; Store result to [W9]
; Post-increment W8
; Post-increment W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W7</td>
<td>W7 2450</td>
</tr>
<tr>
<td>W8</td>
<td>W8 1808</td>
</tr>
<tr>
<td>W9</td>
<td>W9 2022</td>
</tr>
<tr>
<td>Data 1808</td>
<td>Data 1808 92E4</td>
</tr>
<tr>
<td>Data 2022</td>
<td>Data 2022 A557</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0005 (OV, C = 1)</td>
</tr>
</tbody>
</table>
**SWAP**

**Byte or Nibble Swap Wn**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

{label:} SWAP{.B} Wn

**Operands:**

Wn ∈ [W0 ... W15]

**Operation:**

*For byte operation:*

(Wn)<7:4> ↔ (Wn)<3:0>

*For word operation:*

(Wn)<15:8> ↔ (Wn)<7:0>

**Status Affected:** None

**Encoding:**

| 1111 | 1101 | 1B00 | 0000 | 0000 | ssss |

**Description:** Swap the contents of the working register Wn. In Word mode, the two bytes of Wn are swapped. In Byte mode, the two nibbles of the Least Significant Byte of Wn are swapped, and the Most Significant Byte of Wn is unchanged. Register direct addressing must be used for Wn.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte). The 's' bits select the address of the working register.

**Note:** The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1

**Example 1:** SWAP.B W0 ; Nibble swap (W0)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>AB87</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:** SWAP W0 ; Byte swap (W0)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>8095</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
TBLRDH  
Table Read High

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} TBLRDH{.B} [Ws], Wd 
[Ws++] , [Wd] 
[Ws--] , [Wd++] 
[++Ws] , [Wd--] 
[--Ws] , [++Wd] 
[--Wd] 

Operands:  
Ws ∈ [W0 ... W15] 
Wd ∈ [W0 ... W15] 

Operation:  
For byte operation:  
If (LSB(Ws) = 1)  
0 → Wd 
Else  
Program Mem [(TBLPAG),(Ws)] <23:16> → Wd 
For word operation:  
Program Mem [(TBLPAG),(Ws)] <23:16> → Wd <7:0>  
0 → Wd <15:8> 

Status Affected: None 

Encoding: 

|        | 1011 | 1010 | 1bbq | qddd | dppp | ssss |

Description:  
Read the contents of the most significant word of program memory and store it to the destination register Wd. The target word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Ws. Indirect addressing must be used for Ws, and either register direct or indirect addressing may be used for Wd.

In Word mode, zero is stored to the Most Significant Byte of the destination register (due to non-existent program memory) and the third program memory byte (PM<23:16>) at the specified program memory address is stored to the Least Significant Byte of the destination register.

In Byte mode, the source address depends on the contents of Ws. If Ws is not word-aligned, zero is stored to the destination register (due to non-existent program memory). If Ws is word-aligned, the third program memory byte (PM<23:16>) at the specified program memory address is stored to the destination register.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).  
The ‘q’ bits select the destination Address mode.  
The ‘d’ bits select the destination register.  
The ‘p’ bits select the source Address mode.  
The ‘s’ bits select the source register.  

Note: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

Words: 1
Cycles: 2 (PIC24F, PIC24H, dsPIC30F, dsPIC33F)  
5 (PIC24E, dsPIC33E)
Example 1: \( \text{TBLRDH.B \, [W0], \, [W1++] \, \text{Read PM \{TBLPAG:[W0]\} (Byte mode)} \)
\( \text{; Store to [W1]} \)
\( \text{; Post-increment W1} \)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0812</td>
</tr>
<tr>
<td>W1</td>
<td>0F71</td>
</tr>
<tr>
<td>Data 0F70</td>
<td>0944</td>
</tr>
<tr>
<td>Program 01 0812</td>
<td>EF 2042</td>
</tr>
<tr>
<td>TBLPAG</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

Example 2: \( \text{TBLRDH \, [W6++], \, W8 \, \text{Read PM \{TBLPAG:[W6]\} (Word mode)} \)
\( \text{; Store to W8} \)
\( \text{; Post-increment W6} \)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>3406</td>
</tr>
<tr>
<td>W8</td>
<td>65B1</td>
</tr>
<tr>
<td>Program 00 3406</td>
<td>29 2E40</td>
</tr>
<tr>
<td>TBLPAG</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
TBLRDL  
Table Read Low

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  
{label:} TBLRDL[..B] [Ws], Wd  
[Ws++], [Wd]  
[Ws--], [Wd++]  
[++Ws], [Wd--]  
[--Ws], [++Wd]  
[−Wd]

Operands:  
Ws ∈ [W0 ... W15]  
Wd ∈ [W0 ... W15]

Operation:  
For byte operation:  
If (LSB(Ws) = 1)  
Program Mem [(TBLPAG),(Ws)] <15:8> → Wd  
Else  
Program Mem [(TBLPAG),(Ws)] <7:0> → Wd  
For word operation:  
Program Mem [(TBLPAG),(Ws)] <15:0> → Wd

Status Affected: None

Encoding:  
1011 1010 0Bqq qddd dppp ssss

Description:  
Read the contents of the least significant word of program memory and store it to the destination register Wd. The target word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Ws. Indirect addressing must be used for Ws, and either register direct or indirect addressing may be used for Wd.

In Word mode, the lower 2 bytes of program memory are stored to the destination register. In Byte mode, the source address depends on the contents of Ws. If Ws is not word-aligned, the second byte of the program memory word (PM<15:7>) is stored to the destination register. If Ws is word-aligned, the first byte of the program memory word (PM<7:0>) is stored to the destination register.

The 'B' bit selects byte or word operation ('0' for word mode, '1' for byte). The 'q' bits select the destination Address mode. The 'd' bits select the destination register. The 'p' bits select the source Address mode. The 's' bits select the source register.

Note: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

Words: 1
Cycles: 2 (PIC24F, PIC24H, dsPIC30F, dsPIC33F)  
5 (PIC24E, dsPIC33E)
Example 1: TBLRDL.B [W0++] , W1 ; Read PM (TBLPAG:[W0]) (Byte mode)
; Store to W1
; Post-increment W0

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0813</td>
<td>W0 0814</td>
</tr>
<tr>
<td>W1 0F71</td>
<td>W1 0F20</td>
</tr>
<tr>
<td>Data 0F70</td>
<td>Data 0F70</td>
</tr>
<tr>
<td>Program 01 0812</td>
<td>Program 01 0812</td>
</tr>
<tr>
<td>TBLPAG 0001</td>
<td>TBLPAG 0001</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2: TBLRDL [W6] , [W8++] ; Read PM (TBLPAG:[W6]) (Word mode)
; Store to W8
; Post-increment W8

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6 3406</td>
<td>W6 3406</td>
</tr>
<tr>
<td>W8 1202</td>
<td>W8 1204</td>
</tr>
<tr>
<td>Data 1202</td>
<td>Data 1202</td>
</tr>
<tr>
<td>Program 00 3406</td>
<td>Program 00 3406</td>
</tr>
<tr>
<td>TBLPAG 0000</td>
<td>TBLPAG 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
### TBLWTH

**Table Write High**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

{\{label:\} TBLWTH{.B} Ws, [Wd]

[Ws], [Wd++]

[Ws++], [Wd--]

[Ws--], [+Wd]

[++Ws], [--Wd]

[--Ws], [+Wd]

#### Operands:

Ws ∈ [W0 ... W15]

Wd ∈ [W0 ... W15]

#### Operation:

**For byte operation:**

If (LSB(Wd) = 1)

NOP

Else

(Ws) → Program Mem [(TBLPAG), (Wd)]<23:16>

**For word operation:**

(Ws)<7:0> → Program Mem [(TBLPAG), (Wd)]<23:16>

#### Status Affected:

None

#### Encoding:

| 1011 | 1011 | qddd | dppp | ssss |

#### Description:

Store the contents of the working source register Ws to the most significant word of program memory. The destination word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Wd. Either direct or indirect addressing may be used for Ws, and indirect addressing must be used for Wd.

Since program memory is 24 bits wide, this instruction can only write to the upper byte of program memory (PM<23:16>). This may be performed using a Wd that is word-aligned in Byte mode or Word mode. If Byte mode is used with a Wd that is not word-aligned, no operation is performed.

The 'B' bit selects byte or word operation (’0’ for word, ’1’ for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the source register.

**Note:** The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

#### Words:

1

#### Cycles:

2

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 "Multi-Cycle Instructions".
Example 1:  

```
TBLWTH.B [W0++], [W1] ; Write [W0]... (Byte mode)  
; to PM Latch High (TBLPAG:[W1])  
; Post-increment W0
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0812</td>
</tr>
<tr>
<td>W1</td>
<td>0F70</td>
</tr>
<tr>
<td>Data 0812</td>
<td>0944</td>
</tr>
<tr>
<td>Program 01 0F70</td>
<td>EF 2042</td>
</tr>
<tr>
<td>TBLPAG</td>
<td>0001</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Note:** Only the Program Latch is written to. The contents of program memory are not updated until the Flash memory is programmed using the procedure described in the specific device family reference manual.

Example 2:  

```
TBLWTH W6, [W8++] ; Write W6... (Word mode)  
; to PM Latch High (TBLPAG:[W8])  
; Post-increment W8
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6</td>
<td>0026</td>
</tr>
<tr>
<td>W8</td>
<td>0870</td>
</tr>
<tr>
<td>Program 00 0870</td>
<td>22 3551</td>
</tr>
<tr>
<td>TBLPAG</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Note:** Only the Program Latch is written to. The contents of program memory are not updated until the Flash memory is programmed using the procedure described in the specific device family reference manual.
Section 5. Instruction Descriptions

TBLWTL  Table Write Low

Implemented in: PIC24F  PIC24H  PIC24E  dsPIC30F  dsPIC33F  dsPIC33E

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax: 

{label:} TBLWTL(.B) Ws, [Wd]

[Ws], [Wd++]

[Ws++], [Wd--]

[Ws--], [++Wd]

[++Ws], [--Wd]

[--Wd],

Operands: Ws ∈ [W0 ... W15]

Wd ∈ [W0 ... W15]

Operation: For byte operation:

If (LSB(Wd)=1)

(Ws) → Program Mem [(TBLPAG),(Wd)] <15:8>

Else

(Ws) → Program Mem [(TBLPAG),(Wd)] <7:0>

For word operation:

(Ws) → Program Mem [(TBLPAG),(Wd)] <15:0>

Status Affected: None

Encoding:

| 1011 | 1011 | 0Bqq | qddd | dppp | ssss |

Description:

Store the contents of the working source register Ws to the least significant word of program memory. The destination word address of program memory is formed by concatenating the 8-bit Table Pointer register, TBLPAG<7:0>, with the effective address specified by Wd. Either direct or indirect addressing may be used for Ws, and indirect addressing must be used for Wd.

In Word mode, Ws is stored to the lower 2 bytes of program memory. In Byte mode, the Least Significant bit of Wd determines the destination byte.

If Wd is not word-aligned, Ws is stored to the second byte of program memory (PM<15:8>). If Wd is word-aligned, Ws is stored to the first byte of program memory (PM<7:0>).

The 'B' bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘q’ bits select the destination Address mode.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

Note: The extension .B in the instruction denotes a byte move rather than a word move. You may use a .W extension to denote a word move, but it is not required.

Words: 1

Cycles: 2\(^{(1)}\)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

© 2005-2011 Microchip Technology Inc.
Example 1:  

```
TBLWTL.B W0, [W1++] ; Write W0... (Byte mode)
; to PM Latch Low (TBLPAG:[W1])
; Post-increment W1
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 6628</td>
<td>W0 6628</td>
</tr>
<tr>
<td>W1 1225</td>
<td>W1 1226</td>
</tr>
<tr>
<td>Program 00 1224</td>
<td>Program 01 1224</td>
</tr>
<tr>
<td>78 0080</td>
<td>78 2880</td>
</tr>
<tr>
<td>TBLPAG 0000</td>
<td>TBLPAG 0000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Note:** Only the Program Latch is written to. The contents of program memory are not updated until the Flash memory is programmed using the procedure described in the specific device family reference manual.

Example 2:  

```
TBLWTL [W6], [W8] ; Write [W6]... (Word mode)
; to PM Latch Low (TBLPAG:[W8])
; Post-increment W8
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W6 1600</td>
<td>W6 1600</td>
</tr>
<tr>
<td>W8 7208</td>
<td>W8 7208</td>
</tr>
<tr>
<td>Data 1600</td>
<td>Data 1600</td>
</tr>
<tr>
<td>0130</td>
<td>0130</td>
</tr>
<tr>
<td>Program 01 7208</td>
<td>Program 01 7208</td>
</tr>
<tr>
<td>09 0002</td>
<td>09 0130</td>
</tr>
<tr>
<td>TBLPAG 0001</td>
<td>TBLPAG 0001</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Note:** Only the Program Latch is written to. The contents of program memory are not updated until the Flash memory is programmed using the procedure described in the specific device family reference manual.
Section 5. Instruction Descriptions

ULNK  De-allocate Stack Frame

Implemented in: | PIC24F | PIC24H | PIC24E | dsPIC30F | dsPIC33F | dsPIC33E |
---|---|---|---|---|---|---|
   | X | X | | X | X | |

Syntax: \{label:} ULNK

Operands: None

Operation: W14 \rightarrow W15
(W15) – 2 \rightarrow W15
(TOS) \rightarrow W14

Status Affected: None

Encoding: 1111 1010 1000 0000 0000 0000

Description: This instruction de-allocates a Stack Frame for a subroutine calling sequence. The Stack Frame is de-allocated by setting the Stack Pointer (W15) equal to the Frame Pointer (W14), and then POPping the stack to reset the Frame Pointer (W14).

Words: 1

Cycles: 1

Example 1: ULNK \; Unlink the stack frame

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W14 2002</td>
<td>W14 2000</td>
</tr>
<tr>
<td>W15 20A2</td>
<td>W15 2000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2: ULNK \; Unlink the stack frame

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W14 0802</td>
<td>W14 0800</td>
</tr>
<tr>
<td>W15 0812</td>
<td>W15 0800</td>
</tr>
<tr>
<td>Data 0800</td>
<td>Data 0800</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
ULNK  De-allocate Stack Frame

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>XX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:  

{label:} ULNK

Operands:  
None

Operation:

W14 → W15
(W15) – 2 → W15
(TOS) → W14
0 → SFA bit

Status Affected:  
SFA

Encoding:

```
1111 1010 1000 0000 0000 0000
```

Description:  
This instruction de-allocates a Stack Frame for a subroutine calling sequence. The Stack Frame is de-allocated by setting the Stack Pointer (W15) equal to the Frame Pointer (W14), and then POPping the stack to reset the Frame Pointer (W14).

Words:  
1

Cycles:  
1

Example 1:  
ULNK ; Unlink the stack frame

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W14 2002</td>
<td>W14 2000</td>
</tr>
<tr>
<td>W15 20A2</td>
<td>W15 2000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>Data 2000</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

Example 2:  
ULNK ; Unlink the stack frame

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W14 0802</td>
<td>W14 0800</td>
</tr>
<tr>
<td>W15 0812</td>
<td>W15 0800</td>
</tr>
<tr>
<td>Data 0800</td>
<td>Data 0800</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
XOR

Exclusive OR f and WREG

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Syntax: {label:} XOR{.B} f {,WREG}

Operands: f ∈ [0 ... 8191]
Operation: (f).XOR.(WREG) → destination designated by D
Status Affected: N, Z
Encoding:

| 1011 | 0110 | 1Bdf | efff | efff | efff |

Description:

Compute the logical exclusive OR operation of the contents of the default working register WREG and the contents of the specified file register and place the result in the destination register. The optional WREG operand determines the destination register. If WREG is specified, the result is stored in WREG. If WREG is not specified, the result is stored in the file register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'D' bit selects the destination ('0' for WREG, '1' for file register).
The 'f' bits select the address of the file register.

Note 1: The extension .B in the instruction denotes a byte operation rather than a word operation. You may use a .W extension to denote a word operation, but it is not required.

2: The WREG is set to working register W0.

Words: 1
Cycles: 1(1)

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

Example 1: XOR.B 0x1FFF ; XOR (0x1FFF) and WREG (Byte mode) ; Store result to 0x1FFF

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>7804</td>
<td>7804</td>
</tr>
<tr>
<td>Data 1FFE</td>
<td>Data 1FFE</td>
</tr>
<tr>
<td>9439</td>
<td>9039</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>

Example 2: XOR 0xA04, WREG ; XOR (0xA04) and WREG (Word mode) ; Store result to WREG

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>WREG (W0)</td>
<td>WREG (W0)</td>
</tr>
<tr>
<td>6234</td>
<td>C267</td>
</tr>
<tr>
<td>Data 0A04</td>
<td>Data 0A04</td>
</tr>
<tr>
<td>A053</td>
<td>A053</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0008 (N = 1)</td>
</tr>
</tbody>
</table>
### XOR

**Exclusive OR Literal and Wn**

#### Implemented in:

<table>
<thead>
<tr>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

#### Syntax:

```
{label:} XOR{.B} #lit10, Wn
```

#### Operands:

- `lit10` ∈ [0 ... 255] for byte operation
- `lit10` ∈ [0 ... 1023] for word operation
- `Wn` ∈ [W0 ... W15]

#### Operation:

`lit10.XOR(Wn) → Wn`

#### Status Affected:

N, Z

#### Encoding:

```
1011 0010 1Bkk kkkk kkkk dddd
```

#### Description:

Compute the logical exclusive OR operation of the unsigned 10-bit literal operand and the contents of the working register Wn and store the result back in the working register Wn. Register direct addressing must be used for Wn.

The ‘B’ bit selects byte or word operation (‘0’ for word, ‘1’ for byte).

The ‘k’ bits specify the literal operand.

The ‘d’ bits select the address of the working register.

**Note 1:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**Note 2:** For byte operations, the literal must be specified as an unsigned value [0:255]. See Section 4.6 “Using 10-bit Literal Operands” for information on using 10-bit literal operands in Byte mode.

#### Words: 1

#### Cycles: 1

**Example 1:**

```
XOR.B #0x23, W0 ; XOR 0x23 and W0 (Byte mode)
; Store result to W0
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 7804</td>
<td>W0 7827</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
XOR   #0x108, W4 ; XOR 0x108 and W4 (Word mode)
; Store result to W4
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4 6134</td>
<td>W4 603C</td>
</tr>
<tr>
<td>SR 0000</td>
<td>SR 0000</td>
</tr>
</tbody>
</table>
XOR

Exclusive OR Wb and Short Literal

Implemented in:

<table>
<thead>
<tr>
<th></th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Syntax:

```
{label:} XOR{.B} Wb, #lit5, Wd
```

- [Wd]               
- [Wd++]              
- [Wd--]             
- [++Wd]             
- [−Wd]

Operands:

- \( Wb \in [W0 ... W15] \)
- \( \text{lit5} \in [0 ... 31] \)
- \( Wd \in [W0 ... W15] \)

Operation:

\((Wb).\text{XOR.}\text{lit5} \rightarrow Wd\)

Status Affected:

- \(N, Z\)

Encoding:

```
0110 1www wBqq qddd d11k kkkk
```

Description:

Compute the logical exclusive OR operation of the contents of the base register \( Wb \) and the unsigned 5-bit literal operand and place the result in the destination register \( Wd \). Register direct addressing must be used for \( Wb \). Either register direct or indirect addressing may be used for \( Wd \).

The 'w' bits select the address of the base register.

The 'B' bit selects byte or word operation ('0' for word, '1' for byte).

The 'q' bits select the destination Address mode.

The 'd' bits select the destination register.

The 'k' bits provide the literal operand, a 5-bit integer number.

**Note:** The extension \(.B\) in the instruction denotes a byte operation rather than a word operation. You may use a \(.W\) extension to denote a word operation, but it is not required.

Words: 1

Cycles: 1

**Example 1:**

```
XOR.B W4, #0x14, W5     ; XOR W4 and 0x14 (Byte mode)
; Store result to W5
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W4</td>
<td>C822</td>
</tr>
<tr>
<td>W5</td>
<td>1200</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>

**Example 2:**

```
XOR W2, #0x1F, [W8++] ; XOR W2 by 0x1F (Word mode)
; Store result to [W8]
; Post-increment W8
```

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>8505</td>
</tr>
<tr>
<td>W8</td>
<td>1004</td>
</tr>
<tr>
<td>Data 1004</td>
<td>6628</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
</tbody>
</table>
### XOR

**Exclusive OR Wb and Ws**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

```c
{label:} XOR{.B} Wb, Ws, Wd

[Wb], [Ws], [Ws++], [Ws--], [+Ws], [-Ws], [Wd], [Wd++], [Wd--], [+Wd], [--Wd]
```

**Operands:**

- `Wb ∈ [W0 ... W15]`
- `Ws ∈ [W0 ... W15]`
- `Wd ∈ [W0 ... W15]`

**Operation:**

`(Wb).XOR.(Ws) → Wd`

**Status Affected:**

- `N`
- `Z`

**Encoding:**

```
0110 lwww wBqq qddd dppp ssss
```

**Description:**

Compute the logical exclusive OR operation of the contents of the source register `Ws` and the contents of the base register `Wb`, and place the result in the destination register `Wd`. Register direct addressing must be used for `Wb`. Either register direct or indirect addressing may be used for `Ws` and `Wd`.

The 'w' bits select the address of the base register.
The 'B' bit selects byte or word operation ('0' for word, '1' for byte).
The 'q' bits select the destination Address mode.
The 'd' bits select the destination register.
The 'p' bits select the source Address mode.
The 's' bits select the source register.

**Note:** The extension `.B` in the instruction denotes a byte operation rather than a word operation. You may use a `.W` extension to denote a word operation, but it is not required.

**Words:** 1

**Cycles:** 1

**Note 1:** In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see Note 3 in Section 3.2.1 “Multi-Cycle Instructions”.

---


© 2005-2011 Microchip Technology Inc.
### Example 1:

XOR.B W1, [W5++], [W9++] ; XOR W1 and [W5] (Byte mode)
; Store result to [W9]
; Post-increment W5 and W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>AAAA</td>
</tr>
<tr>
<td>W5</td>
<td>2000</td>
</tr>
<tr>
<td>W9</td>
<td>2600</td>
</tr>
<tr>
<td>Data 2000</td>
<td>115A</td>
</tr>
<tr>
<td>Data 2600</td>
<td>0000</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>Data 2000</td>
<td>115A</td>
</tr>
<tr>
<td>Data 2600</td>
<td>00F0</td>
</tr>
<tr>
<td>SR</td>
<td>0008</td>
</tr>
</tbody>
</table>

### Example 2:

XOR W1, W5, W9 ; XOR W1 and W5 (Word mode)
; Store the result to W9

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>FEDC</td>
</tr>
<tr>
<td>W5</td>
<td>1234</td>
</tr>
<tr>
<td>W9</td>
<td>A34D</td>
</tr>
<tr>
<td>SR</td>
<td>0000</td>
</tr>
<tr>
<td>W1</td>
<td>FEDC</td>
</tr>
<tr>
<td>W5</td>
<td>1234</td>
</tr>
<tr>
<td>W9</td>
<td>ECE8</td>
</tr>
<tr>
<td>SR</td>
<td>0008</td>
</tr>
</tbody>
</table>
### ZE

**Zero-Extend Ws**

<table>
<thead>
<tr>
<th>Implemented in:</th>
<th>PIC24F</th>
<th>PIC24H</th>
<th>PIC24E</th>
<th>dsPIC30F</th>
<th>dsPIC33F</th>
<th>dsPIC33E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Syntax:**

\[
\{\text{label:}\} \text{ZE} \quad \text{Ws,} \quad \text{Wnd}
\]

- \([\text{Ws}]\)
- \([\text{Ws}++]\)
- \([\text{Ws}--]\)
- \([++\text{Ws}]\)
- \([-\text{Ws}]\)

**Operands:**

- \(\text{Ws} \in [\text{W0} \ldots \text{W15}]\)
- \(\text{Wnd} \in [\text{W0} \ldots \text{W15}]\)

**Operation:**

- \(\text{Ws}<7:0> \rightarrow \text{Wnd}<7:0>\)
- \(0 \rightarrow \text{Wnd}<15:8>\)

**Status Affected:**

- \(\text{N, Z, C}\)

**Encoding:**

\[
\begin{array}{cccccc}
\text{l111} & \text{l101} & \text{l000} & \text{0ddd} & \text{dp\text{pp}} & \text{s\text{sss}}
\end{array}
\]

**Description:**

Zero-extend the Least Significant Byte in source working register \(\text{Ws}\) to a 16-bit value and store the result in the destination working register \(\text{Wnd}\). Either register direct or indirect addressing may be used for \(\text{Ws}\), and register direct addressing must be used for \(\text{Wnd}\). The \(\text{N}\) flag is cleared and the \(\text{C}\) flag is set, because the zero-extended word is always positive.

The ‘d’ bits select the destination register.

The ‘p’ bits select the source Address mode.

The ‘s’ bits select the source register.

**Note 1:** This operation converts a byte to a word, and it uses no .B or .W extension.

**Note 2:** The source \(\text{Ws}\) is addressed as a byte operand, so any address modification is by ‘1’.

**Words:** 1

**Cycles:** 1

**Example 1:**

ZE \(\text{W3, W4}\); zero-extend \(\text{W3}\)

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{W3})</td>
<td>(\text{W3})</td>
</tr>
<tr>
<td>(\text{W4})</td>
<td>(\text{W4})</td>
</tr>
<tr>
<td>(\text{SR})</td>
<td>(\text{SR}) (C = 1)</td>
</tr>
</tbody>
</table>

Note 1: In dsPIC33E and PIC24E devices, the listed cycle count does not apply to read and read-modify-write operations on non-CPU Special Function Registers. For more details, see **Note 3** in [Section 3.2.1 “Multi-Cycle Instructions”].
Section 5. Instruction Descriptions

Example 2:  

ZE [W2+], W12 ; Zero-extend [W2]  
; Store to W12  
; Post-increment W2

<table>
<thead>
<tr>
<th>Before Instruction</th>
<th>After Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2</td>
<td>W2</td>
</tr>
<tr>
<td>0900</td>
<td>0901</td>
</tr>
<tr>
<td>W12</td>
<td>W12</td>
</tr>
<tr>
<td>1002</td>
<td>008F</td>
</tr>
<tr>
<td>Data 0900</td>
<td>Data 0900</td>
</tr>
<tr>
<td>268F</td>
<td>268F</td>
</tr>
<tr>
<td>SR</td>
<td>SR</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
</tr>
</tbody>
</table>

(C = 1)
Section 6. Built-in Functions

HIGHLIGHTS

This section of the manual contains the following major topics:

6.1 Introduction .................................................................................................................... 446
6.2 Built-in Function List.................................................................................................... 447
6.1  INTRODUCTION

This section describes the built-in functions that are specific to the MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs (formerly MPLAB C30).

Built-in functions give the C programmer access to assembler operators or machine instructions that are currently only accessible using in-line assembly, but are sufficiently useful that they are applicable to a broad range of applications. Built-in functions are coded in C source files syntactically like function calls, but they are compiled to assembly code that directly implements the function, and do not involve function calls or library routines.

There are a number of reasons why providing built-in functions is preferable to requiring programmers to use in-line assembly. They include the following:

1. Providing built-in functions for specific purposes simplifies coding.
2. Certain optimizations are disabled when in-line assembly is used. This is not the case for built-in functions.
3. For machine instructions that use dedicated registers, coding in-line assembly while avoiding register allocation errors can require considerable care. The built-in functions make this process simpler as you do not need to be concerned with the particular register requirements for each individual machine instruction.

The built-in functions are listed below followed by their individual detailed descriptions.

- __builtin_addab
- __builtin_add
- __builtin_btg
- __builtin_clr
- __builtin_clr_prefetch
- __builtin_divf
- __builtin_divmodsd
- __builtin_divmodud
- __builtin_divsd
- __builtin_divud
- __builtin_dmaoffset
- __builtin_ed
- __builtin_edac
- __builtin_edsoffset
- __builtin_edspage
- __builtin_fblcl
- __builtin_lac
- __builtin_mac
- __builtin_modsd
- __builtin_modud
- __builtin_movsac
- __builtin_mpy
- __builtin_mpy
- __builtin_mpy
- __builtin_mpyn
- __builtin_msc
- __builtin_mulss
- __builtin_mulsoffset
- __builtin_mulspage
- __builtin_mulsp
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplication
- __builtin_multiplica

This section describes only the built-in functions related to the CPU operations. The compiler provides additional built-in functions for operations such as writing to Flash program memory and changing the oscillator settings. Refer to the “MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide” (DS51284) for a complete list of compiler built-in functions.
6.2  BUILT-IN FUNCTION LIST

This section describes the programmer interface to the compiler built-in functions. Since the functions are “built-in”, there are no header files associated with them. Similarly, there are no command-line switches associated with the built-in functions – they are always available. The built-in function names are chosen such that they belong to the compiler’s namespace (they all have the prefix __builtin_), so they will not conflict with function or variable names in the programmer’s namespace.

__builtin_addab

Description:
Add accumulators A and B with the result written back to the specified accumulator. For example:

```
register int result asm("A");
register int B asm("A");

result = __builtin_addab(result,B);
```
will generate:
```
add A
```

Prototype:
```
int __builtin_addab(int Accum_a, int Accum_b);
```

Argument:
- **Accum_a** First accumulator to add.
- **Accum_b** Second accumulator to add.

Return Value:
Returns the addition result to an accumulator.

Assembler Operator / Machine Instruction:
```
add
```

Error Messages:
An error message appears if the result is not an accumulator register.
__builtin_add

Description:
Add value to the accumulator specified by result with a shift specified by literal shift. For example:

```
register int result asm("A");
int value;
result = __builtin_add(result,value,0);
```

If value is held in w0, the following will be generated:
add w0, #0, A

Prototype:
```
int __builtin_add(int Accum,int value, const int shift);
```

Argument:
Accum Accumulator to add.
value Integer number to add to accumulator value.
shift Amount to shift resultant accumulator value.

Return Value:
Returns the shifted addition result to an accumulator.

Assembler Operator / Machine Instruction:
add

Error Messages:
An error message appears if:
• the result is not an accumulator register
• argument 0 is not an accumulator
• the shift value is not a literal within range
__builtin_btg

**Description:**
This function will generate a `btg` machine instruction. Some examples include:
```c
int i;  /* near by default */
int l __attribute__((far));
struct foo {
    int bit1:1;
} barbits;

int bar;

void some_bittoggles() {
    register int j asm("w9");
    int k;
    k = i;
    __builtin_btg(&i,1);
    __builtin_btg(&j,3);
    __builtin_btg(&k,4);
    __builtin_btg(&l,11);
    return j+k;
}
```

Note that taking the address of a variable in a register will produce warning by the compiler and cause the register to be saved onto the stack (so that its address may be taken); this form is not recommended. This caution only applies to variables explicitly placed in registers by the programmer.

**Prototype:**
```c
void __builtin_btg(unsigned int *, unsigned int 0xn);
```

**Argument:**
- `*` A pointer to the data item for which a bit should be toggled.
- `0xn` A literal value in the range of 0 to 15.

**Return Value:**
Returns a `btg` machine instruction.

**Assembler Operator / Machine Instruction:**
`btg`

**Error Messages:**
An error message appears if the parameter values are not within range.
__builtin_clr

Description:
Clear the specified accumulator. For example:

register int result asm("A");
result = __builtin_clr();

will generate:
clr A

Prototype:
int __builtin_clr(void);

Argument:
None

Return Value:
Returns the cleared value result to an accumulator.

Assembler Operator / Machine Instruction:
clr

Error Messages:
An error message appears if the result is not an accumulator register.
__builtin_clr_prefetch

Description:
Clear an accumulator and prefetch data ready for a future MAC operation.

`xptr` may be null to signify no X prefetch to be performed, in which case the values of `xincr` and `xval` are ignored, but required.

`yptr` may be null to signify no Y prefetch to be performed, in which case the values of `yincr` and `yval` are ignored, but required.

`xval` and `yval` nominate the address of a C variable where the prefetched value will be stored.

`xincr` and `yincr` may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

If `AWB` is non null, the other accumulator will be written back into the referenced variable.

For example:
```c
register int result asm("A");
register int B asm("B");
int x_memory_buffer[256] __attribute__((space(xmemory)));
int y_memory_buffer[256] __attribute__((space(ymemory)));
int *xmemory;
int *ymemory;
int awb;
int xVal, yVal;

xmemory = x_memory_buffer;
ymemory = y_memory_buffer;
result = __builtin_clr(&xmemory, &xVal, 2,
                      &ymemory, &yVal, 2, &awb, B);
```

May generate:
```
clr A, [w8]+=2, w4, [w10]+=2, w5, w13
```

The compiler may need to spill w13 to ensure that it is available for the write-back. It may be recommended to users that the register be claimed for this purpose.

After this instruction:
- result will be cleared
- `xVal` will contain `x_memory_buffer[0]`
- `yVal` will contain `y_memory_buffer[0]`
- `xmemory` and `ymemory` will be incremented by 2, ready for the next mac operation

Prototype:
```c
int __builtin_clr_prefetch(
    int **xptr, int *xval, int xincr,
    int **yptr, int *yval, int yincr, int *AWB,
    int AWB_accum);
```
_builtin_clr_prefetch (Continued)

Argument:

- **xptr**: Integer pointer to x prefetch.
- **xval**: Integer value of x prefetch.
- **xincr**: Integer increment value of x prefetch.
- **y.ptr**: Integer pointer to y prefetch.
- **y.val**: Integer value of y prefetch.
- **y.incr**: Integer increment value of y prefetch.
- **AWB**: Accumulator write back location.
- **AWB_accum**: Accumulator to write back.

**Note:** The arguments **xptr** and **y.ptr** must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:

Returns the cleared value result to an accumulator.

Assembler Operator / Machine Instruction:

- **clr**

Error Messages:

An error message appears if:

- the result is not an accumulator register
- **x.val** is a null value but **xptr** is not null
- **y.val** is a null value but **y.ptr** is not null
- **AWB_accum** is not an accumulator and **AWB** is not null
Section 6. Built-in Functions

__builtin_divf

Description:
Computes the quotient \( \frac{num}{den} \). A math error exception occurs if \( den \) is zero. Function arguments are unsigned, as is the function result.

Prototype:
unsigned int __builtin_divf(unsigned int num,
unsigned int den);

Argument:
num  numerator
den  denominator

Return Value:
Returns the unsigned integer value of the quotient \( \frac{num}{den} \).

Assembler Operator / Machine Instruction:
div.f

__builtin_divmodsd

Description:
Issues the 16-bit architecture's native signed divide support. Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture both the quotient and remainder.

Prototype:
signed int __builtin_divmodsd(
signed long dividend, signed int divisor,
signed int *remainder);

Argument:

 dividend  number to be divided
divisor    number to divide by
remainder  pointer to remainder

Return Value:
Quotient and remainder.

Assembler Operator / Machine Instruction:
divmodsd

Error Messages:
None.
__builtin_divmodud

Description:
Issues the 16-bit architecture’s native unsigned divide support. Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture both the quotient and remainder.

Prototype:
unsigned int __builtin_divmodud(
unsigned long dividend, unsigned int divisor,
unsigned int *remainder);

Argument:
- dividend: number to be divided
- divisor: number to divide by
- remainder: pointer to remainder

Return Value:
Quotient and remainder.

Assembler Operator / Machine Instruction:
divmodud

Error Messages:
None.

__builtin_divsd

Description:
Computes the quotient \( \frac{num}{den} \). A math error exception occurs if \( den \) is zero. Function arguments are signed, as is the function result. The command-line option -Wconversions can be used to detect unexpected sign conversions.

Prototype:
int __builtin_divsd(const long num, const int den);

Argument:
- num: numerator
- den: denominator

Return Value:
Returns the signed integer value of the quotient \( \frac{num}{den} \).

Assembler Operator / Machine Instruction:
div.sd
__builtin_divud

Description:
Computes the quotient num / den. A math error exception occurs if den is zero. Function arguments are unsigned, as is the function result. The command-line option -Wconversions can be used to detect unexpected sign conversions.

Prototype:

unsigned int __builtin_divud(const unsigned long num, const unsigned int den);

Argument:
num numerator
den denominator

Return Value:
Returns the unsigned integer value of the quotient num / den.

Assembler Operator / Machine Instruction:

div.ud

__builtin_dmaoffset

Description:
Obtains the offset of a symbol within DMA memory. For example:

unsigned int result;
char buffer[256] __attribute__((space(dma)));

result = __builtin_dmaoffset(&buffer);

May generate:

mov #dmaoffset(buffer), w0

Prototype:

unsigned int __builtin_dmaoffset(const void *p);

Argument:
*p pointer to DMA address value

Return Value:
Returns the offset to a variable located in DMA memory.

Assembler Operator / Machine Instruction:

dmaoffset

Error Messages:
An error message appears if the parameter is not the address of a global symbol.
__builtin_ed

Description:
Squares \( \text{sqr} \), returning it as the result. Also prefetches data for future square operation by computing \( xptr - yptr \) and storing the result in \( \text{distance} \).
\( xincr \) and \( yincr \) may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

For example:
```c
register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(distance,
    &xmemory, 2,
    &ymemory, 2,
    &distance);
```

May generate:
```assembly
ed w4*w4, A, [w8]+=2, [W10]+=2, w4
```

Prototype:
```c
int __builtin_ed(int sqr, int **xptr, int xincr,
    int **yptr, int yincr, int *distance);
```

Argument:
- \( \text{sqr} \) Integer squared value.
- \( \text{xptr} \) Integer pointer to pointer to x prefetch.
- \( \text{xincr} \) Integer increment value of x prefetch.
- \( \text{yptr} \) Integer pointer to pointer to y prefetch.
- \( \text{yincr} \) Integer increment value of y prefetch.
- \( \text{distance} \) Integer pointer to distance.

Note: The arguments \( \text{xptr} \) and \( \text{yptr} \) must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:
Returns the squared result to an accumulator.

Assembler Operator / Machine Instruction:
```assembly
ed
```

Error Messages:
An error message appears if:
- the result is not an accumulator register
- \( \text{xptr} \) is null
- \( \text{yptr} \) is null
- \( \text{distance} \) is null
__builtin_edac

Description:
Squares `sqr` and sums with the nominated accumulator register, returning it as the result. Also prefetches data for future square operation by computing `**xptr - **yptr` and storing the result in `*distance`.

`xincr` and `yincr` may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

For example:

```c
register int result asm("A");
int *xmemory, *ymemory;
int distance;

result = __builtin_ed(result, distance,
                     &xmemory, 2,
                     &ymemory, 2,
                     &distance);
```

May generate:

```
edac w4*w4, A, [w8]+=2, [W10]+=2, w4
```

Prototype:

```c
int __builtin_edac(int Accum, int sqr,
                   int **xptr, int xincr, int **yptr, int yincr,
                   int *distance);
```

Argument:

- `Accum`  Accumulator to sum.
- `sqr`   Integer squared value.
- `xptr` Integer pointer to pointer to x prefetch.
- `xincr` Integer increment value of x prefetch.
- `yptr` Integer pointer to pointer to y prefetch.
- `yincr` Integer increment value of y prefetch.
- `distance` Integer pointer to distance.

Note: The arguments `xptr` and `yptr` must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:
Returns the squared result to specified accumulator.

Assembler Operator / Machine Instruction:
```
edac
```

Error Messages:
An error message appears if:

- the result is not an accumulator register
- `Accum` is not an accumulator register
- `xptr` is null
- `yptr` is null
- `distance` is null
__builtin_edsoffset

Description:
Returns the eds page offset of the object whose address is given as a parameter. The argument $p$ must be the address of an object in extended data space; otherwise an error message is produced and the compilation fails. See the space attribute in Section 2.3.1 “Specifying Attributes of Variables” of the “MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide” (DS51284).

Prototype:

unsigned int __builtin_edsoffset(int *p);

Argument:
$p$ object address

Return Value:
Returns the eds page number of the object whose address is given as a parameter.

Assembler Operator / Machine Instruction:
edsoffset

__builtin_edspage

Description:
Returns the eds page number of the object whose address is given as a parameter. The argument $p$ must be the address of an object in extended data space; otherwise an error message is produced and the compilation fails. See the space attribute in Section 2.3.1 “Specifying Attributes of Variables” of the “MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide” (DS51284).

Prototype:

unsigned int __builtin_edspage(int *p);

Argument:
$p$ object address

Return Value:
Returns the eds page number of the object whose address is given as a parameter.

Assembler Operator / Machine Instruction:
edspage
__builtln_fbcl

**Description:**
Finds the first bit change from left in value. This is useful for dynamic scaling of fixed-point data.
For example:

```c
int result, value;
result = __builtin_fbcl(value);
```
May generate:
```
fbl w4, w5
```

**Prototype:**
```
int __builtin_fbcl(int value);
```

**Argument:**
```
value  Integer number of first bit change.
```

**Return Value:**
Returns the shifted addition result to an accumulator.

**Assembler Operator / Machine Instruction:**
```
fbl
```

**Error Messages:**
An error message appears if the result is not an accumulator register.

__builtln_lac

**Description:**
Shifts value by `shift` (a literal between -8 and 7) and returns the value to be stored into the accumulator register. For example:

```c
register int result asm("A");
int value;
result = __builtin_lac(value, 3);
```
May generate:
```
lac w4, #3, A
```

**Prototype:**
```
int __builtin_lac(int value, int shift);
```

**Argument:**
```
value  Integer number to be shifted.
shift  Literal amount to shift.
```

**Return Value:**
Returns the shifted addition result to an accumulator.

**Assembler Operator / Machine Instruction:**
```
lac
```

**Error Messages:**
An error message appears if:
- the result is not an accumulator register
- the shift value is not a literal within range
__builtin_mac

Description:
Computes \(a \times b\) and sums with accumulator; also prefetches data ready for a future MAC operation.

\(xptr\) may be null to signify no X prefetch to be performed, in which case the values of \(xincr\) and \(xval\) are ignored, but required.

\(yptr\) may be null to signify no Y prefetch to be performed, in which case the values of \(yincr\) and \(yval\) are ignored, but required.

\(xval\) and \(yval\) nominate the address of a C variable where the prefetched value will be stored.
\(xincr\) and \(yincr\) may be the literal values: \(-6, -4, -2, 0, 2, 4, 6\) or an integer value.
If \(AWB\) is non null, the other accumulator will be written back into the referenced variable.

For example:
```c
register int result asm("A");
register int B asm("B");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mac(result, xVal, yVal, &xmemory, &xVal, 2, &ymemory, &yVal, 2, 0, B);
```

May generate:
```
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:
```c
int __builtin_mac(int Accum, int a, int b,
int **xptr, int *xval, int xincr,
int **yptr, int *yval, int yincr, int *AWB,
int AWB_accum);
```

Argument:
- **Accum**: Accumulator to sum.
- **a**: Integer multiplicand.
- **b**: Integer multiplier.
- **xptr**: Integer pointer to pointer to x prefetch.
- **xval**: Integer pointer to value of x prefetch.
- **xincr**: Integer increment value of x prefetch.
- **yptr**: Integer pointer to pointer to y prefetch.
- **yval**: Integer pointer to value of y prefetch.
- **yincr**: Integer increment value of y prefetch.
- **AWB**: Accumulator write-back location.
- **AWB_accum**: Accumulator to write-back.

**Note:** The arguments \(xptr\) and \(yptr\) must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:
Returns the cleared value result to an accumulator.

**Assembler Operator / Machine Instruction:**
```
mac
```
__builtin_mac (Continued)

Error Messages:
An error message appears if:
• the result is not an accumulator register
• _Accum_ is not an accumulator register
• _xval_ is a null value but _xptr_ is not null
• _yval_ is a null value but _yptr_ is not null
• _AWB_accum_ is not an accumulator register and _AWB_ is not null
__builtin_modsd

Description:
Issues the 16-bit architecture’s native signed divide support. Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture only the remainder.

Prototype:
signed int __builtin_modsd(signed long dividend, signed int divisor);

Argument:
dividend  number to be divided
divisor    number to divide by

Return Value:
Remainder.

Assembler Operator / Machine Instruction:
modsd

Error Messages:
None.

__builtin_modud

Description:
Issues the 16-bit architecture’s native unsigned divide support. Notably, if the quotient does not fit into a 16-bit result, the results (including remainder) are unexpected. This form of the built-in function will capture only the remainder.

Prototype:
unsigned int __builtin_modud(unsigned long dividend, unsigned int divisor);

Argument:
dividend  number to be divided
divisor    number to divide by

Return Value:
Remainder.

Assembler Operator / Machine Instruction:
modud

Error Messages:
None.
__builtin_movsac

Description:
Computes nothing, but prefetches data ready for a future MAC operation.

\(xptr\) may be null to signify no X prefetch to be performed, in which case the values of \(xval\) and \(xincr\) are ignored, but required.

\(yptr\) may be null to signify no Y prefetch to be performed, in which case the values of \(yval\) and \(yincr\) are ignored, but required.

\(xval\) and \(yval\) nominate the address of a C variable where the prefetched value will be stored.

\(xincr\) and \(yincr\) may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

If \(AWB\) is not null, the other accumulator will be written back into the referenced variable.

For example:

```c
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;
result = __builtin_movsac(&xmemory, &xVal, 2,
                          &ymemory, &yVal, 2, 0, 0);
```

May generate:

```
movsac A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```c
int __builtin_movsac(
    int **xptr, int *xval, int xincr,
    int **yptr, int *yval, int yincr, int *AWB
    int AWB_accum);
```

Argument:

- \(xptr\) Integer pointer to pointer to x prefetch.
- \(xval\) Integer pointer to value of x prefetch.
- \(xincr\) Integer increment value of x prefetch.
- \(yptr\) Integer pointer to pointer to y prefetch.
- \(yval\) Integer pointer to value of y prefetch.
- \(yincr\) Integer increment value of y prefetch.
- \(AWB\) Accumulator write back location.
- \(AWB\_accum\) Accumulator to write back.

Note: The arguments \(xptr\) and \(yptr\) must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:

Returns prefetch data.

Assembler Operator / Machine Instruction:

```
movsac
```

Error Messages:

An error message appears if:

- the result is not an accumulator register
- \(xval\) is a null value but \(xptr\) is not null
- \(yval\) is a null value but \(yptr\) is not null
- \(AWB\_accum\) is not an accumulator register and \(AWB\) is not null
__builtin_mpy

Description:
Computes $a \times b$; also prefetches data ready for a future MAC operation.

*xptr* may be null to signify no X prefetch to be performed, in which case the values of *xincr* and *xval* are ignored, but required.

*yptr* may be null to signify no Y prefetch to be performed, in which case the values of *yincr* and *yval* are ignored, but required.

*xval* and *yval* nominate the address of a C variable where the prefetched value will be stored.

*xincr* and *yincr* may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

For example:

```c
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                       &xmemory, &xVal, 2,
                       &ymemory, &yVal, 2);
```

May generate:

```c
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```c
int __builtin_mpy(int a, int b,
                   int **xptr, int *xval, int xincr,
                   int **yptr, int *yval, int yincr);
```

Argument:

- **a**: Integer multiplicand.
- **b**: Integer multiplier.
- **xptr**: Integer pointer to pointer to x prefetch.
- **xval**: Integer pointer to value of x prefetch.
- **xincr**: Integer increment value of x prefetch.
- **yptr**: Integer pointer to pointer to y prefetch.
- **yval**: Integer pointer to value of y prefetch.
- **yincr**: Integer increment value of y prefetch.
- **AWB**: Integer pointer to accumulator selection.

Note: The arguments *xptr* and *yptr* must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:

Returns the cleared value result to an accumulator.

Assembler Operator / Machine Instruction:

```c
mpy
```

Error Messages:

An error message appears if:

- the result is not an accumulator register
- *xval* is a null value but *xptr* is not null
- *yval* is a null value but *yptr* is not null
__builtin_mpyn

Description:
Computes \(-a \times b\); also prefetches data ready for a future MAC operation.

*xptr may be null to signify no X prefetch to be performed, in which case the values of xincr and xval are ignored, but required.
*yptr may be null to signify no Y prefetch to be performed, in which case the values of yincr and yval are ignored, but required.

xval and yval nominate the address of a C variable where the prefetched value will be stored.

xincr and yincr may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

For example:

```c
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_mpy(xVal, yVal,
                        &xmemory, &xVal, 2,
                        &ymemory, &yVal, 2);
```

May generate:
```
mac w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:

```c
int __builtin_mpyn(int a, int b,
                    int **xptr, int *xval, int xincr,
                    int **yptr, int *yval, int yincr);
```

Argument:

- **a**: Integer multiplicand.
- **b**: Integer multiplier.
- **xptr**: Integer pointer to pointer to x prefetch.
- **xval**: Integer pointer to value of x prefetch.
- **xincr**: Integer increment value of x prefetch.
- **yptr**: Integer pointer to pointer to y prefetch.
- **yval**: Integer pointer to value of y prefetch.
- **yincr**: Integer increment value of y prefetch.
- **AWS**: Integer pointer to accumulator selection.

**Note:** The arguments xptr and yptr must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:

Returns the cleared value result to an accumulator.

Assembler Operator / Machine Instruction:

```c
mpyn
```

Error Messages:

An error message appears if:
- the result is not an accumulator register
- **xval** is a null value but **xptr** is not null
- **yval** is a null value but **yptr** is not null
__builtin_msc

Description:
Computes \( a \times b \) and subtracts from accumulator; also prefetches data ready for a future MAC operation.

\( xptr \) may be null to signify no X prefetch to be performed, in which case the values of \( xincr \) and \( xval \) are ignored, but required.

\( yptr \) may be null to signify no Y prefetch to be performed, in which case the values of \( yincr \) and \( yval \) are ignored, but required.

\( xval \) and \( yval \) nominate the address of a C variable where the prefetched value will be stored.

\( xincr \) and \( yincr \) may be the literal values: -6, -4, -2, 0, 2, 4, 6 or an integer value.

If \( AWB \) is non null, the other accumulator will be written back into the referenced variable.

For example:
```c
register int result asm("A");
int *xmemory;
int *ymemory;
int xVal, yVal;

result = __builtin_msc(result, xVal, yVal,
                        &xmemory, &xVal, 2,
                        &ymemory, &yVal, 2, 0, 0);
```

May generate:
```
msc w4*w5, A, [w8]+=2, w4, [w10]+=2, w5
```

Prototype:
```c
int __builtin_msc(int Accum, int a, int b,
                   int **xptr, int *xval, int xincr,
                   int **yptr, int *yval, int yincr, int *AWB,
                   int AWB_accum);
```

Argument:

- **Accum**: Accumulator to sum.
- **a**: Integer multiplicand.
- **b**: Integer multiplier.
- **xptr**: Integer pointer to pointer to x prefetch.
- **xval**: Integer pointer to value of x prefetch.
- **xincr**: Integer increment value of x prefetch.
- **yptr**: Integer pointer to pointer to y prefetch.
- **yval**: Integer pointer to value of y prefetch.
- **yincr**: Integer increment value of y prefetch.
- **AWB**: Accumulator write back location.
- **AWB_accum**: Accumulator to write back.

Note: The arguments \( xptr \) and \( yptr \) must point to the arrays located in the x data memory and y data memory, respectively.

Return Value:
Returns the cleared value result to an accumulator.

Assembler Operator / Machine Instruction:
```
msc
```
Error Messages:

An error message appears if:

- the result is not an accumulator register
- Accum is not an accumulator register
- xval is a null value but xptr is not null
- yval is a null value but yptr is not null
- AWB_accum is not an accumulator register and AWB is not null
__builtin_mulss

Description:
Computes the product $p0 \times p1$. Function arguments are signed integers, and the function result is a signed long integer. The command-line option `-Wconversions` can be used to detect unexpected sign conversions.

Prototype:
signed long __builtin_mulss(const signed int p0, const signed int p1);

Argument:
- $p0$ multiplicand
- $p1$ multiplier

Return Value:
Returns the signed long integer value of the product $p0 \times p1$.

Assembler Operator / Machine Instruction:
mul.ss

__builtin_mulsu

Description:
Computes the product $p0 \times p1$. Function arguments are integers with mixed signs, and the function result is a signed long integer. The command-line option `-Wconversions` can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction, including immediate mode for operand $p1$.

Prototype:
signed long __builtin_mulsu(const signed int p0, const unsigned int p1);

Argument:
- $p0$ multiplicand
- $p1$ multiplier

Return Value:
Returns the signed long integer value of the product $p0 \times p1$.

Assembler Operator / Machine Instruction:
mul.su
__builtin_mulus

Description:
Computes the product \( p_0 \times p_1 \). Function arguments are integers with mixed signs, and the function result is a signed long integer. The command-line option -Wconversions can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction.

Prototype:

```c
signed long __builtin_mulus(const unsigned int p0, const signed int p1);
```

Argument:
- \( p_0 \) multiplicand
- \( p_1 \) multiplier

Return Value:
Returns the signed long integer value of the product \( p_0 \times p_1 \).

Assembler Operator / Machine Instruction:

```
mul.us
```

__builtin_muluu

Description:
Computes the product \( p_0 \times p_1 \). Function arguments are unsigned integers, and the function result is an unsigned long integer. The command-line option -Wconversions can be used to detect unexpected sign conversions. This function supports the full range of addressing modes of the instruction, including immediate mode for operand \( p_1 \).

Prototype:

```c
unsigned long __builtin_muluu(const unsigned int p0, const unsigned int p1);
```

Argument:
- \( p_0 \) multiplicand
- \( p_1 \) multiplier

Return Value:
Returns the signed long integer value of the product \( p_0 \times p_1 \).

Assembler Operator / Machine Instruction:

```
mul.uu
```
__builtin_nop

Description:
Generates a nop instruction.
Prototype:
void __builtin_nop(void);
Argument:
None.
Return Value:
Returns a no operation (nop).
Assembler Operator / Machine Instruction:
nop

__builtin_psvoffset

Description:
Returns the psv page offset of the object whose address is given as a parameter. The argument
p must be the address of an object in an EE data, PSV or executable memory space; otherwise
an error message is produced and the compilation fails. See the space attribute in
Section 2.3.1 “Specifying Attributes of Variables” of the “MPLAB® C Compiler for PIC24
MCUs and dsPIC® DSCs User's Guide” (DS51284).
Prototype:
unsigned int __builtin_psvoffset(const void *p);
Argument:
p object address
Return Value:
Returns the psv page number offset of the object whose address is given as a parameter.
Assembler Operator / Machine Instruction:
psvoffset
Error Messages:
The following error message is produced when this function is used incorrectly:
“Argument to __builtin_psvoffset() is not the address of an object in code, psv, or
eedata section”.
The argument must be an explicit object address.
For example, if obj is object in an executable or read-only section, the following syntax is valid:
unsigned page = __builtin_psvoffset(&obj);
Section 6. Built-in Functions

__builtin_psvpage

Description:
Returns the psv page number of the object whose address is given as a parameter. The argument p must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the space attribute in Section 2.3.1 "Specifying Attributes of Variables" of the "MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide" (DS51284).

Prototype:
unsigned int __builtin_psvpage(const void *p);

Argument:
p object address

Return Value:
Returns the psv page number of the object whose address is given as a parameter.

Assembler Operator / Machine Instruction:
psvpage

Error Messages:
The following error message is produced when this function is used incorrectly:
"Argument to __builtin_psvpage() is not the address of an object in code, psv, or eedata section".
The argument must be an explicit object address.
For example, if obj is object in an executable or read-only section, the following syntax is valid:
unsigned page = __builtin_psvpage(&obj);

__builtin_readsfr

Description:
Reads the SFR.

Prototype:
unsigned int __builtin_readsfr(const void *p);

Argument:
p object address

Return Value:
Returns the SFR.

Assembler Operator / Machine Instruction:
readsfr

Error Messages:
The following error message is produced when this function is used incorrectly:
__builtin_return_address

Description:
Returns the return address of the current function, or of one of its callers. For the level argument, a value of 0 yields the return address of the current function, a value of 1 yields the return address of the caller of the current function, and so forth. When level exceeds the current stack depth, 0 will be returned. This function should only be used with a non-zero argument for debugging purposes.

Prototype:
int __builtin_return_address (const int level);

Argument:
level Number of frames to scan up the call stack.

Return Value:
Returns the return address of the current function, or of one of its callers.

Assembler Operator / Machine Instruction:
return_address

__builtin_sac

Description:
Shifts value by shift (a literal between -8 and 7) and returns the value.

For example:
register int value asm("A");
int result;

result = __builtin_sac(value,3);

May generate:
sac A, #3, w0

Prototype:
int __builtin_sac(int value, int shift);

Argument:
value Integer number to be shifted.
shift Literal amount to shift.

Return Value:
Returns the shifted result to an accumulator.

Assembler Operator / Machine Instruction:
sac

Error Messages:
An error message appears if:
• the result is not an accumulator register
• the shift value is not a literal within range
__builtin_sacr

Description:
Shifts value by \textit{shift} (a literal between -8 and 7) and returns the value which is rounded using the rounding mode determined by the CORCONbits.RND control bit.

For example:

```c
register int value asm("A");
int result;

result = __builtin_sac(value,3);
```

May generate:

```assembly
sac.r A, #3, w0
```

Prototype:

```c
int __builtin_sacr(int value, int shift);
```

Argument:

- \texttt{value}  Integer number to be shifted.
- \texttt{shift}  Literal amount to shift.

Return Value:

Returns the shifted result to the CORCON register.

Assembler Operator / Machine Instruction:

```assembly
sacr
```

Error Messages:

An error message appears if:

- the result is not an accumulator register
- the shift value is not a literal within range
__builtin_sftac

Description:
Shifts accumulator by \textit{shift}. The valid shift range is -16 to 16.

For example:

\begin{verbatim}
register int result asm("A");
int i;

result = __builtin_sftac(result,i);
\end{verbatim}

May generate:

\texttt{sftac A, w0}

Prototype:

\begin{verbatim}
int __builtin_sftac(int Accum, int shift);
\end{verbatim}

Argument:

\begin{itemize}
  \item \textit{Accum} \hspace{1cm} \textit{Accumulator to shift.}
  \item \textit{shift} \hspace{1cm} \textit{Amount to shift.}
\end{itemize}

Return Value:

Returns the shifted result to an accumulator.

Assembler Operator / Machine Instruction:

\texttt{sftac}

Error Messages:

An error message appears if:
\begin{itemize}
  \item the result is not an accumulator register
  \item \textit{Accum} is not an accumulator register
  \item the shift value is not a literal within range
\end{itemize}
__builtin_subab

Description:
Subtracts accumulators A and B with the result written back to the specified accumulator. For example:

```c
register int result asm("A");
register int B asm("B");
result = __builtin_subab(result,B);
```

will generate:

```assembly
sub A
```

Prototype:
```c
int __builtin_subab(int Accum_a, int Accum_b);
```

Argument:
- `Accum_a`: Accumulator from which to subtract.
- `Accum_b`: Accumulator to subtract.

Return Value:
Returns the subtraction result to an accumulator.

Assembler Operator / Machine Instruction:
`sub`

Error Messages:
An error message appears if the result is not an accumulator register.

__builtin_tbladdress

Description:
Returns a value that represents the address of an object in program memory. The argument `p` must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the `space` attribute in Section 2.3.1 “Specifying Attributes of Variables” of the “MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide” (DS51284).

Prototype:
```c
unsigned long __builtin_tbladdress(const void *p);
```

Argument:
- `p`: object address

Return Value:
Returns an `unsigned long` value that represents the address of an object in program memory.

Assembler Operator / Machine Instruction:
`tbladdress`
__builtin_tbladdress

Error Messages:
The following error message is produced when this function is used incorrectly:

"Argument to __builtin_tbladdress() is not the address of an object in code, psv, or eedata section".

The argument must be an explicit object address.
For example, if obj is object in an executable or read-only section, the following syntax is valid:

unsigned long page = __builtin_tbladdress(&obj);

__builtin_tbloffset

Description:
Returns the table page offset of the object whose address is given as a parameter. The argument p must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the space attribute in Section 2.3.1 “Specifying Attributes of Variables” of the "MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide" (DS51284).

Prototype:

unsigned int __builtin_tbloffset(const void *p);

Argument:
p object address

Return Value:
Returns the table page number offset of the object whose address is given as a parameter.

Assembler Operator / Machine Instruction:
tbloffset

Error Messages:
The following error message is produced when this function is used incorrectly:

"Argument to __builtin_tbloffset() is not the address of an object in code, psv, or eedata section".

The argument must be an explicit object address.
For example, if obj is object in an executable or read-only section, the following syntax is valid:

unsigned page = __builtin_tbloffset(&obj);
__builtin_tblpage

Description:
Returns the table page number of the object whose address is given as a parameter. The argument \( p \) must be the address of an object in an EE data, PSV or executable memory space; otherwise an error message is produced and the compilation fails. See the space attribute in Section 2.3.1 “Specifying Attributes of Variables” of the “MPLAB® C Compiler for PIC24 MCUs and dsPIC® DSCs User’s Guide” (DS51284).

Prototype:
unsigned int __builtin_tblpage(const void *p);

Argument:
\( p \) object address

Return Value:
Returns the table page number of the object whose address is given as a parameter.

Assembler Operator / Machine Instruction:
tblpage

Error Messages:
The following error message is produced when this function is used incorrectly:
“Argument to __builtin_tblpage() is not the address of an object in code, psv, or eedata section”.
The argument must be an explicit object address.
For example, if \( \text{obj} \) is object in an executable or read-only section, the following syntax is valid:
unsigned page = __builtin_tblpage(&\text{obj});

__builtin_tblrdh

Description:
Issues the tblrdh.w instruction to read a word from Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of __builtin_tbloffset() and __builtin_tblpage().

Please refer to the specific device data sheet or the appropriate family reference manual for complete details regarding reading and writing program Flash.

Prototype:
unsigned int __builtin_tblrdh(unsigned int offset);

Argument:
\( offset \) desired memory offset

Return Value:
None.

Assembler Operator / Machine Instruction:
tblrdh

Error Messages:
None.
__builtin_tblrdl

Description:
Issues the tblrdl.w instruction to read a word from Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of __builtin_tbloffset() and __builtin_tblpage().
Please refer to the specific device data sheet or the appropriate family reference manual for complete details regarding reading and writing program Flash.

Prototype:
unsigned int __builtin_tblrdl(unsigned int offset);

Argument:
offset  desired memory offset

Return Value:
None.

Assembler Operator / Machine Instruction:
tblrdl

Error Messages:
None.

__builtin_tblwth

Description:
Issues the tblwth.w instruction to write a word to Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of __builtin_tbloffset() and __builtin_tblpage().
Please refer to the specific device data sheet or the appropriate family reference manual for complete details regarding reading and writing program Flash.

Prototype:
void __builtin_tblwth(unsigned int offset unsigned int data);

Argument:
offset  desired memory offset
data  data to be written

Return Value:
None.

Assembler Operator / Machine Instruction:
tblwth

Error Messages:
None.
**__builtin_tblwtl**

**Description:**
Issues the `tblrdl.w` instruction to write a word to Flash or EEDATA memory. You must set up the TBLPAG to point to the appropriate page. To do this, you may make use of `__builtin_tbloffset()` and `__builtin_tblpage()`.

Please refer to the specific device data sheet or the appropriate family reference manual for complete details regarding reading and writing program Flash.

**Prototype:**
```c
void __builtin_tblwtl(unsigned int offset, unsigned int data);
```

**Argument:**
- `offset`: desired memory offset
- `data`: data to be written

**Return Value:**
None.

**Assembler Operator / Machine Instruction:**
- `tblwtl`

**Error Messages:**
None.
Example 6-1: Additional Inline Functions

```c
#include "p33fxxxx.h"

volatile long Result_mpy1616;
volatile long Result_addab;
volatile long Result_subab;
volatile long Result_mpy3216;
volatile long Result_div3216;

register int Accu_A asm("A");
register int Accu_B asm("B");

inline static long mpy_32_16 (long, int);

inline static long mpy_32_16 (long x, int y)
{
    long result;
    int temp1, temp2;
    temp1 = (x>>1)&0x7FFF;
    temp2 = x>>16;
    Accu_A = __builtin_mpy (temp1, y, 0,0,0,0,0,0);
    Accu_A = __builtin_sftac (15);
    Accu_A = __builtin_mac (temp2, y, 0,0,0,0,0,0);
    asm("mov _ACCAL,%0
        "mov _ACCAH,%d0": "r"(result): "w"(Accu_A));
    return result;
}

int main (void)
{
    // Variable declarations
    int Input1;
    int Input2;
    int Input3;
    int Input4;
    long Input5;
    int Input6;
    long Input7;
    int Input8;

    // Enable 32-bit saturation, signed and fractional modes for both ACCA
    // and ACCB
    CORCON = 0x00C0;

    // Example of 16*16-bit fractional multiplication using ACCA
    Input1 = 32767;
    Input2 = 32767;
    Accu_A = __builtin_mpy (Input1, Input2, 0,0,0,0,0,0);
    asm("mov _ACCAL,%0
        "mov _ACCAH,%d0": "r"(Result_mpy1616): "w"(Accu_A));

    // Example of 16*16-bit fractional multiplication using ACCB
    Input3 = 16384;
    Input4 = 16384;
    Accu_B = __builtin_mpy (Input3, Input4, 0,0,0,0,0,0);
    asm("mov _ACCBBL,%0
        "mov _ACCBBH,%d0": "r"(Result_mpy1616): "w"(Accu_B));

    // Example of 32-bit addition using ACCA (ACCA = ACCA + ACCB)
    Accu_A = __builtin_addab();
    asm("mov _ACCAH,%d0": "r"(Result_addab): "w"(Accu_A));

    // Example of 32-bit subtraction using ACCB (ACCB = ACCB - ACCA)
    Accu_B = __builtin_subab();
    asm("mov _ACCBH,%d0": "r"(Result_subab): "w"(Accu_B));

    // Example of 32*16-bit fractional multiplication using ACCA
    Input5 = 0x7FFFFFFF;
    Input6 = 32767;
    Result_mpy3216 = mpy_32_16 (Input5, Input6);
    while(1);
}
```
Example 6-2: Divide_32_by_16

```c
#include <p33Fxxxx.h>
#include "divide.h"

_FOSCSEL(FNOSC_FRC);
_FOSC(FCCKSM_CSDCMD & OSCIOFNC_OFF & POSCMD_NONE);
_FWDT(FWDTEN_OFF);

unsigned int divide_(long a, int b) {
    union convert {
        unsigned long l;
        unsigned int i[2];
    } c;

    int sign;
    unsigned int result;

    c.l = a;
    sign = c.i[1] ^ b;

    if (a < 0) a = (-a);
    if (b < 0) b = -b;
    result = __builtin_divud(a, b);
    result >>= 1;
    if (sign < 0) result = -result;
    return result;
}

int main(void) {
    unsigned long dividend;
    unsigned int divisor;
    unsigned int quotient;

    dividend = 0x3FFFFFFF;
    divisor = 0x7FFF;

    quotient = divide_((long)dividend, (int)divisor);
    while(1);}
```
Section 7. Reference

HIGHLIGHTS

This section of the manual contains the following major topics:

7.1 Instruction Bit Map ................................................................. 484
7.2 Instruction Set Summary Table .................................................. 486
7.3 Revision History ....................................................................... 496
7.1 INSTRUCTION BIT MAP

Instruction encoding for the 16-bit MCU and DSC family devices is summarized in Table 7-1. This table contains the encoding for the MSB of each instruction. The first column in the table represents bits 23:20 of the opcode, and the first row of the table represents bits 19:16 of the opcode. The first byte of the opcode is formed by taking the first column bit value and appending the first row bit value. For instance, the MSB of the **PUSH** instruction (last row, ninth column) is encoded with \( \text{11111000b (0xF8)} \).

**Note:** The complete opcode for each instruction may be determined by the instruction descriptions in Section 5. "Instruction Descriptions", using Table 5-1 through Table 5-12.
### Table 7-1: Instruction Encoding

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>NOP</td>
<td>BRA</td>
<td>CALL</td>
<td>GOTO</td>
</tr>
<tr>
<td>0001</td>
<td>SUBR</td>
<td>SUBBR</td>
<td>BLED</td>
</tr>
<tr>
<td>0010</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>0011</td>
<td>ADD</td>
<td>ADD</td>
<td>ADD</td>
</tr>
<tr>
<td>0100</td>
<td>SUB</td>
<td>SUBB</td>
<td>SUB</td>
</tr>
<tr>
<td>0110</td>
<td>AND</td>
<td>XOR</td>
<td>XOR</td>
</tr>
<tr>
<td>0111</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>1000</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>1001</td>
<td>MOV</td>
<td>MOV</td>
<td>MOV</td>
</tr>
<tr>
<td>1010</td>
<td>BSET</td>
<td>BCLR</td>
<td>BTG</td>
</tr>
<tr>
<td>1011</td>
<td>ADD</td>
<td>SUB</td>
<td>SUBB</td>
</tr>
<tr>
<td>1100</td>
<td>MAC</td>
<td>MPY</td>
<td>MPY</td>
</tr>
<tr>
<td>1101</td>
<td>SL</td>
<td>ASR</td>
<td>LSR</td>
</tr>
<tr>
<td>1110</td>
<td>CP0</td>
<td>CP</td>
<td>CPB</td>
</tr>
<tr>
<td>1111</td>
<td>EDAC</td>
<td>MAC</td>
<td>MPY</td>
</tr>
</tbody>
</table>

**Note:**
1. This instruction is only available in dsPIC30F, dsPIC33F, and dsPIC33E family devices.
2. This instruction is only available in PIC24E and dsPIC33E family devices.
7.2 INSTRUCTION SET SUMMARY TABLE

The complete 16-bit MCU and DSC device instruction set is summarized in Table 7-2. This table contains an alphabetized listing of the instruction set. It includes instruction assembly syntax, description, size (in 24-bit words), execution time (in instruction cycles), affected Status bits, and the page number in which the detailed description can be found. Table 1-2 identifies the symbols that are used in the Instruction Set Summary Table.

Table 7-2: Instruction Set Summary Table

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Mnemonic, Operands</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>f {,WREG}</td>
<td>Destination = f + WREG</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ADD</td>
<td>#lit10,Wn</td>
<td>Wn = lit10 + Wn</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>100</td>
</tr>
<tr>
<td>ADD</td>
<td>Wb,#lit5,Wd</td>
<td>Wd = Wb + lit5</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>101</td>
</tr>
<tr>
<td>ADD</td>
<td>Wb,Ws,Wd</td>
<td>Wd = Wb + Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>102</td>
</tr>
<tr>
<td>ADD</td>
<td>Acc(2)</td>
<td>Add accumulators</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>103</td>
</tr>
<tr>
<td>ADD</td>
<td>Wso,#Slit4,Acc</td>
<td>16-bit signed add to accumulator</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>104</td>
</tr>
<tr>
<td>ADDC</td>
<td>f {,WREG}</td>
<td>Destination = f + WREG + (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>106</td>
</tr>
<tr>
<td>ADDC</td>
<td>#lit10,Wn</td>
<td>Wn = lit10 + Wn + (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>107</td>
</tr>
<tr>
<td>ADDC</td>
<td>Wb,#lit5,Wd</td>
<td>Wd = Wb + lit5 + (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>108</td>
</tr>
<tr>
<td>ADDC</td>
<td>Wb,Ws,Wd</td>
<td>Wd = Wb + Ws + (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>110</td>
</tr>
<tr>
<td>AND</td>
<td>f {,WREG}</td>
<td>Destination = f .AND. WREG</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>112</td>
</tr>
<tr>
<td>AND</td>
<td>#lit10,Wn</td>
<td>Wn = lit10 .AND. Wn</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>113</td>
</tr>
<tr>
<td>AND</td>
<td>Wb,#lit5,Wd</td>
<td>Wd = Wb .AND. lit5</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>114</td>
</tr>
<tr>
<td>AND</td>
<td>Wb,Ws,Wd</td>
<td>Wd = Wb .AND. Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>115</td>
</tr>
<tr>
<td>ASR</td>
<td>f {,WREG}</td>
<td>Destination = arithmetic right shift f, Lsb →C</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>117</td>
</tr>
<tr>
<td>ASR</td>
<td>Ws,Wd</td>
<td>Wd = arithmetic right shift Ws, Lsb →C</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>119</td>
</tr>
<tr>
<td>ASR</td>
<td>Wb,#lit4,Wnd</td>
<td>Wnd = arithmetic right shift Wb by lit4, Lsb →C</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>121</td>
</tr>
</tbody>
</table>

Legend:  Ø set or cleared;  ø may be cleared, but never set;  ⊖ may be set, but never cleared;  ⊖ always set;  ⊖ always cleared; — unchanged

Note 1: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
Note 2: This instruction/operand is only available in PIC24E and dsPIC33E devices.
Note 3: This instruction/operand is only available in PIC24E devices.
Note 4: This instruction/operand is only available in dsPIC33E devices.
Note 5: This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
Note 6: This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### ASR

\[ Wb, Wns, Wnd \]

Wnd = arithmetic right shift Wb by Wns, LSb — IC

### BCLR

<table>
<thead>
<tr>
<th>f, #bit4</th>
<th>Bit clear f</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>Expr</th>
<th>Branch unconditionally</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>Wn</th>
<th>Computed branch</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>C, Expr</th>
<th>Branch if Carry</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>GE, Expr</th>
<th>Branch if greater than or equal</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>GEU, Expr</th>
<th>Branch if Carry</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>GT, Expr</th>
<th>Branch if greater than</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>GTU, Expr</th>
<th>Branch if unsigned greater than</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>LE, Expr</th>
<th>Branch if less than or equal</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>LEU, Expr</th>
<th>Branch if unsigned less than or equal</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>LT, Expr</th>
<th>Branch if less than</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>LTU, Expr</th>
<th>Branch if Carry</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>N, Expr</th>
<th>Branch if Negative</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>NC, Expr</th>
<th>Branch if not Carry</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>NN, Expr</th>
<th>Branch if not Negative</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>NO, Expr</th>
<th>Branch if Overflow</th>
</tr>
</thead>
</table>

### BRA

<table>
<thead>
<tr>
<th>NZ, Expr</th>
<th>Branch if not Zero</th>
</tr>
</thead>
</table>

### BSBC

<table>
<thead>
<tr>
<th>OA, Expr(2)</th>
<th>Branch if Accumulator A overflow</th>
</tr>
</thead>
</table>

### BSBC

<table>
<thead>
<tr>
<th>OB, Expr(2)</th>
<th>Branch if Accumulator B overflow</th>
</tr>
</thead>
</table>

### BSBC

<table>
<thead>
<tr>
<th>OV, Expr</th>
<th>Branch if Overflow</th>
</tr>
</thead>
</table>

### BSBC

<table>
<thead>
<tr>
<th>SA, Expr(2)</th>
<th>Branch if Accumulator A saturated</th>
</tr>
</thead>
</table>

### BSBC

<table>
<thead>
<tr>
<th>SB, Expr(2)</th>
<th>Branch if Accumulator B saturated</th>
</tr>
</thead>
</table>

### BSBC

<table>
<thead>
<tr>
<th>Z, Expr</th>
<th>Branch if Zero</th>
</tr>
</thead>
</table>

### BSET

<table>
<thead>
<tr>
<th>f, #bit4</th>
<th>Bit set f</th>
</tr>
</thead>
</table>

### BSET

<table>
<thead>
<tr>
<th>Ws, #bit4</th>
<th>Bit set Ws</th>
</tr>
</thead>
</table>

### BSET

<table>
<thead>
<tr>
<th>Ws, Wb</th>
<th>Write C bit to Ws &lt; Wb</th>
</tr>
</thead>
</table>

---

**Legend:**

- Set or cleared; may be cleared, but never set; may be set, but never cleared; always set; always cleared; unchanged

**Note:**

1. SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3. This instruction/operand is only available in PIC24E and dsPIC33E devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in dsPIC30F, PIC24H, dsPIC33F, and dsPIC33E devices.
6. This instruction/operand is only available in dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax Mnemonic, Operands</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSN.Z ws, Wb</td>
<td>Write Z bit to Ws &lt; Wb&gt;</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTG f, #bit4</td>
<td>Bit toggle f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTG ws, #bit4</td>
<td>Bit toggle Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSC f, #bit4</td>
<td>Bit test f, skip if clear</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSC ws, #bit4</td>
<td>Bit test Ws, skip if clear</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSS f, #bit4</td>
<td>Bit test f, skip if set</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSS ws, #bit4</td>
<td>Bit test Ws, skip if set</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTST f, #bit4</td>
<td>Bit test f to Z</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Ø</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTST.C ws, #bit4</td>
<td>Bit test Ws to C</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTST.Z ws, #bit4</td>
<td>Bit test Ws to Z</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTST.C ws, wb</td>
<td>Bit test Ws &lt; Wb&gt; to C</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTST.Z ws, wb</td>
<td>Bit test Ws &lt; Wb&gt; to Z</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSTS f, #bit4</td>
<td>Bit test f to Z, then set f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSTS.C ws, #bit4</td>
<td>Bit test Ws to C then set</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BTSTS.Z ws, #bit4</td>
<td>Bit test Ws to Z then set</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CALL Expr</td>
<td>Call subroutine</td>
<td>2</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CALL Wn</td>
<td>Call indirect subroutine</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CALL.L Wn(3)</td>
<td>Call indirect subroutine (long address)</td>
<td>1</td>
<td>4</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLR f</td>
<td>f = 0x0000</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLR WREG</td>
<td>WREG = 0x0000</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLR Wd</td>
<td>Wd = 0</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLR Acc, [Wx], [Wxd], [Wy], <a href="5">Wyd</a></td>
<td>Clear accumulator</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>CLRWDWT</td>
<td>Clear Watchdog Timer</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>COM f (, WREG)</td>
<td>Destination = f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>COM ws, Wd</td>
<td>Wd = ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

**Legend:**
- Ø set or cleared; Ø may be cleared, but never set; Ø may be set, but never cleared; Ø always set; Ø always cleared; — unchanged

**Note:**
1. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
2. This instruction/operand is only available in PIC24E and dsPIC33E devices.
3. This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
6. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax Mnemonic, Operands</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP f</td>
<td>Compare (f – WREG)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>191</td>
</tr>
<tr>
<td>CP Wb,#lit5</td>
<td>Compare (Wb – #lit5)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>192</td>
</tr>
<tr>
<td>CP Wb,#lit8</td>
<td>Compare (Wb – #lit8)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>193</td>
</tr>
<tr>
<td>CP Wb,Ws</td>
<td>Compare (Wb – Ws)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>194</td>
</tr>
<tr>
<td>CP0 f</td>
<td>Compare (f – 0x0000)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>196</td>
</tr>
<tr>
<td>CP0 Ws</td>
<td>Compare (Ws – 0x0000)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>197</td>
</tr>
<tr>
<td>CPB f</td>
<td>Compare with borrow (f – WREG – C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>198</td>
</tr>
<tr>
<td>CPB Wb,#lit5</td>
<td>Compare with borrow (Wb – #lit5 – C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>199</td>
</tr>
<tr>
<td>CPB Wb,#lit8</td>
<td>Compare with borrow (Wb – #lit8 – C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>200</td>
</tr>
<tr>
<td>CPB Wb,Ws</td>
<td>Compare with borrow (Wb – Ws – C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>201</td>
</tr>
<tr>
<td>CPBEQ Wb,Wn,Expr(3)</td>
<td>Compare Wb with Wn, branch if =</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>203</td>
</tr>
<tr>
<td>CPBGTE Wb,Wn,Expr(3)</td>
<td>Signed Compare Wb with Wn, branch if &gt;</td>
<td>1</td>
<td>1</td>
<td>(5)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>204</td>
</tr>
<tr>
<td>CPBLT Wb,Wn,Expr(3)</td>
<td>Signed Compare Wb with Wn, branch if &lt;</td>
<td>1</td>
<td>1</td>
<td>(5)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>205</td>
</tr>
<tr>
<td>CPBNE Wb,Wn,Expr(3)</td>
<td>Compare Wb with Wn, branch if ≠</td>
<td>1</td>
<td>1</td>
<td>(5)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>206</td>
</tr>
<tr>
<td>CPSEQ Wb,Wn</td>
<td>Compare (Wb with Wn), skip if =</td>
<td>1</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>207</td>
</tr>
<tr>
<td>CPSEQ Wb,Wn</td>
<td>Signed Compare (Wb with Wn), skip if &gt;</td>
<td>1</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>211</td>
</tr>
<tr>
<td>CPSEQ Wb,Wn</td>
<td>Signed Compare (Wb with Wn), skip if &lt;</td>
<td>1</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>212</td>
</tr>
<tr>
<td>CPSEQ Wb,Wn</td>
<td>Compare (Wb with Wn), skip if ≠</td>
<td>1</td>
<td>1</td>
<td>(2 or 3)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>214</td>
</tr>
<tr>
<td>DAW.W Wn</td>
<td>Wn = decimal adjust Wn</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>216</td>
</tr>
<tr>
<td>DEC f (,WREG)</td>
<td>Destination = f – 1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>217</td>
</tr>
<tr>
<td>DEC Ws,Wd</td>
<td>Wd = Ws – 1</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>DEC2 f (,WREG)</td>
<td>Destination = f – 2</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>220</td>
</tr>
<tr>
<td>DEC2 Ws,Wd</td>
<td>Wd = Ws – 2</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>221</td>
</tr>
</tbody>
</table>

**Legend:**
- $\diamond$ may be set, but never cleared;
- $\checkmark$ may be cleared, but never set;
- ‘1’ always set;
- ‘0’ always cleared;
- — unchanged

**Note:**
1. SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3. This instruction/operand is only available in PIC24E and dsPIC33E devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
6. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DISI #lit14</code></td>
<td>Disable interrupts for lit14 instruction cycles</td>
<td>223</td>
</tr>
<tr>
<td><code>DIV.S Wn,Wn</code></td>
<td>Signed 16/16-bit integer divide, Q →Wo, R →W1</td>
<td>224</td>
</tr>
<tr>
<td><code>DIV.SD Wn,Wn</code></td>
<td>Signed 32/32-bit integer divide, Q →Wo, R →W1</td>
<td>224</td>
</tr>
<tr>
<td><code>DIV.U Wn,Wn</code></td>
<td>Unsigned 16/16-bit integer divide, Q →Wo, R →W1</td>
<td>226</td>
</tr>
<tr>
<td><code>DIV.UD Wn,Wn</code></td>
<td>Unsigned 32/32-bit integer divide, Q →Wo, R →W1</td>
<td>226</td>
</tr>
<tr>
<td><code>DIVF Wn,Wn(2)</code></td>
<td>Signed 16/16-bit fractional divide, Q →Wo, R →W1</td>
<td>228</td>
</tr>
<tr>
<td><code>DO #lit14,Expr(2)</code></td>
<td>Do code to PC + Expr, (lit14 + i) times</td>
<td>230</td>
</tr>
<tr>
<td><code>DO #lit15,Expr(2)</code></td>
<td>Do code to PC + Expr, (lit15 + i) times</td>
<td>233</td>
</tr>
<tr>
<td><code>DO Wn,Expr(2)</code></td>
<td>Do code to PC + Expr, (Wn + i) times</td>
<td>235</td>
</tr>
<tr>
<td><code>ED Wm,Wm,Acc,[Wx],[Wy],[Wd(2)]</code></td>
<td>Euclidean distance (no accumulate)</td>
<td>239</td>
</tr>
<tr>
<td><code>EDAC Wm,Wm,Acc,[Wx],[Wy],[Wd(2)]</code></td>
<td>Euclidean distance</td>
<td>241</td>
</tr>
<tr>
<td><code>EXCH Wn,Wn</code></td>
<td>Swap Wns and Wnd</td>
<td>243</td>
</tr>
<tr>
<td><code>FBCL Wn,Wn</code></td>
<td>Find bit change from left (MSb) side</td>
<td>244</td>
</tr>
<tr>
<td><code>FFIL Wn,Wn</code></td>
<td>Find first one from left (MSb) side</td>
<td>246</td>
</tr>
<tr>
<td><code>FFIR Wn,Wn</code></td>
<td>Find first one from right (LSb) side</td>
<td>248</td>
</tr>
<tr>
<td><code>GOTO Expr</code></td>
<td>Go to address</td>
<td>250</td>
</tr>
<tr>
<td><code>GOTO Wn</code></td>
<td>Go to address indirectly</td>
<td>251</td>
</tr>
<tr>
<td><code>GOTO.L Wn(3)</code></td>
<td>Go to address indirectly (long address)</td>
<td>253</td>
</tr>
<tr>
<td><code>INC f(WREG)</code></td>
<td>Destination = f + 1</td>
<td>254</td>
</tr>
<tr>
<td><code>INC Ws,Wd</code></td>
<td>Wd = Ws + 1</td>
<td>255</td>
</tr>
<tr>
<td><code>INC2 f(WREG)</code></td>
<td>Destination = f + 2</td>
<td>257</td>
</tr>
<tr>
<td><code>INC2 Ws,Wd</code></td>
<td>Wd = Ws + 2</td>
<td>258</td>
</tr>
<tr>
<td><code>IOR f(WREG)</code></td>
<td>Destination = f .IOR. WREG</td>
<td>260</td>
</tr>
<tr>
<td><code>IOR #lit0,Wn</code></td>
<td>Wn = #lit0 .JOR. Wn</td>
<td>261</td>
</tr>
<tr>
<td><code>IOR Wb,#lit5,Wd</code></td>
<td>Wd = Wb .JOR. lit5</td>
<td>262</td>
</tr>
<tr>
<td><code>IOR Wb,Wb,Wr</code></td>
<td>Wr = Wb .JOR. Ws</td>
<td>263</td>
</tr>
<tr>
<td><code>LAC Wso,#lit4,Acc(2)</code></td>
<td>Load accumulator</td>
<td>265</td>
</tr>
</tbody>
</table>

**Legend:**
- `;` set or cleared; `&` may be cleared, but never set; `@` may be set, but never cleared; `!` always set; `*` always cleared; `—` unchanged

**Note:**
1. SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3. This instruction/operand is only available in PIC24E and dsPIC33E devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in dsPIC33E devices.
6. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNK #lit14</td>
<td>Link Frame Pointer</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>267</td>
</tr>
<tr>
<td>LSR f (),WREG</td>
<td>Destination = logical right shift f, MSb → C</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>269</td>
</tr>
<tr>
<td>LSR Ws,Wd</td>
<td>Wd = logical right shift Ws, MSb → C</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>271</td>
</tr>
<tr>
<td>LSR Wb,#lit4,Wnd</td>
<td>Wnd = logical right shift Wb by lit4, MSb → C</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>273</td>
</tr>
<tr>
<td>MAC Wm*Wn,Acc,[Wx],[Wd],[Wyd],AWB</td>
<td>Multiply and accumulate</td>
<td>1</td>
<td>1</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>275</td>
</tr>
<tr>
<td>MAC Wm*Wn,Acc,[Wx],[Wd],[Wyd]</td>
<td>Square and accumulate</td>
<td>1</td>
<td>1</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>277</td>
</tr>
<tr>
<td>MOV f (),WREG</td>
<td>Move f to destination</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>279</td>
</tr>
<tr>
<td>MOV WREG,f</td>
<td>Move WREG to f</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>280</td>
</tr>
<tr>
<td>MOV f,Wd</td>
<td>Move f to Wd</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>281</td>
</tr>
<tr>
<td>MOV Wns,f</td>
<td>Move Wns to f</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>282</td>
</tr>
<tr>
<td>MOV.B #lit8,Wnd</td>
<td>Move 8-bit unsigned literal to Wnd</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>283</td>
</tr>
<tr>
<td>MOV #lit16,Wnd</td>
<td>Move 16-bit literal to Wnd</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>284</td>
</tr>
<tr>
<td>MOV [Ws+Slit10],Wnd</td>
<td>Move [Ws + Slit10] to Wnd</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>285</td>
</tr>
<tr>
<td>MOV Wns,[Wd+Slit10]</td>
<td>Move Wns to [Wd + Slit10]</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>286</td>
</tr>
<tr>
<td>MOV Wso,Wdo</td>
<td>Move Wso to Wdo</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>287</td>
</tr>
<tr>
<td>MOV.D Wns,Wnd</td>
<td>Move double Wns to Wnd:Wnd + 1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>MOV.D Wns,Wnd</td>
<td>Move double Wns:Wns + 1 to Wnd</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>289</td>
</tr>
<tr>
<td>MOVPAG #lit10,DSRPAG</td>
<td>Move 10-bit literal to DSRPAG</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>MOVPAG #lit9,DSWPAG</td>
<td>Move 9-bit literal to DSWPAG</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>MOVPAG #lit8,TBLPAG</td>
<td>Move 8-bit literal to TBLPAG</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>291</td>
</tr>
<tr>
<td>MOVPAG Wn,DSRPAG</td>
<td>Move Wn to DSRPAG</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>292</td>
</tr>
<tr>
<td>MOVPAG Wn,DSWPAG</td>
<td>Move Wn to DSWPAG</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>292</td>
</tr>
<tr>
<td>MOVPAG Wn,TBLPAG</td>
<td>Move Wn to TBLPAG</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>292</td>
</tr>
<tr>
<td>MOVSAC Acc,[Wx],Wxd,[Wy],Wyd,AWB</td>
<td>Move [Wx] to Wxd, and [Wy] to Wyd</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>293</td>
</tr>
<tr>
<td>MPY Wm*Wn,Acc,[Wx],Wxd,[Wy],Wyd</td>
<td>Multiply Wn by Wm to accumulator</td>
<td>1</td>
<td>1</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>295</td>
</tr>
</tbody>
</table>

**Legend:** ☐ set or cleared; ☐ may be cleared, but never set; ☐ may be set, but never cleared; ’1’ always set; ’0’ always cleared; — unchanged

**Note:**
1. SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3. This instruction/operand is only available in PIC24E and dsPIC33E devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33E devices.
6. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPY Wm,Wn,Acc, [Wx],Wxd, [Wy], Wyd(2)</td>
<td>Square to accumulator</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>297</td>
</tr>
<tr>
<td>MPY.N Wm,Wn,Acc, [Wx],Wxd, [Wy], Wyd(2)</td>
<td>(Multiply Wn by Wm) to accumulator</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>299</td>
</tr>
<tr>
<td>MSC Wm,Wn,Acc, [Wx],Wxd, [Wy], Wyd, ARB(2)</td>
<td>Multiply and subtract from accumulator</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>301</td>
</tr>
<tr>
<td>MUL f</td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>303</td>
</tr>
<tr>
<td>MUL.SS Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>305</td>
</tr>
<tr>
<td>MUL.SS Wb,Ws,Acc(4)</td>
<td>Accumulator = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>307</td>
</tr>
<tr>
<td>MUL.SU Wb,#lit5,Wnd</td>
<td>(Wnd + 1,Wnd) = signed(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>308</td>
</tr>
<tr>
<td>MUL.SU Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = signed(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>310</td>
</tr>
<tr>
<td>MUL.SU Wb,Ws,Acc(4)</td>
<td>Accumulator = signed(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>312</td>
</tr>
<tr>
<td>MUL.SU Wb,#lit5,Acc(4)</td>
<td>Accumulator = signed(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>314</td>
</tr>
<tr>
<td>MUL.SS Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = unsigned(Wb) * signed(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>315</td>
</tr>
<tr>
<td>MUL.SS Wb,Ws,Acc(4)</td>
<td>Accumulator = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>317</td>
</tr>
<tr>
<td>MUL.UU Wb,#lit5,Wnd</td>
<td>(Wnd + 1,Wnd) = unsigned(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>319</td>
</tr>
<tr>
<td>MUL.UU Wb,Ws,Wnd</td>
<td>(Wnd + 1,Wnd) = unsigned(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>320</td>
</tr>
<tr>
<td>MUL.UU Wb,Ws,Acc(4)</td>
<td>Accumulator = unsigned(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>322</td>
</tr>
<tr>
<td>MUL.UU Wb,#lit5,Acc(4)</td>
<td>Accumulator = unsigned(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>323</td>
</tr>
<tr>
<td>MUL.WS Wb,Ws,Wnd(3)</td>
<td>Wnd = signed(Wb) * signed(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>324</td>
</tr>
<tr>
<td>MUL.WS Wb,Ws,Wnd(3)</td>
<td>Wnd = signed(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>326</td>
</tr>
<tr>
<td>MUL.WS Wb,#lit5,Wnd(3)</td>
<td>Wnd = signed(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>328</td>
</tr>
<tr>
<td>MUL.WS Wb,Ws,Wnd(3)</td>
<td>Wnd = unsigned(Wb) * signed(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>329</td>
</tr>
<tr>
<td>MUL.WS Wb,Ws,Wnd(3)</td>
<td>Wnd = unsigned(Wb) * unsigned(Ws)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>331</td>
</tr>
<tr>
<td>MUL.WS Wb,#lit5,Wnd(3)</td>
<td>Wnd = unsigned(Wb) * unsigned(lit5)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>332</td>
</tr>
<tr>
<td>NEG f (WREG)</td>
<td>Destination = f + 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>333</td>
</tr>
<tr>
<td>NEG Ws,Wd</td>
<td>Wd = Ws + 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>333</td>
</tr>
<tr>
<td>NEG Acc(2)</td>
<td>Negate accumulator</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>335</td>
</tr>
<tr>
<td>NOP</td>
<td>No operation</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>336</td>
</tr>
</tbody>
</table>

Legend: * set or cleared; Ø may be cleared, but never set; Ø may be set, but never cleared; 1 always set; 0 always cleared; — unchanged

Note
1: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2: This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3: This instruction/operand is only available in PIC24E and dsPIC33E devices.
4: This instruction/operand is only available in dsPIC33E devices.
5: This instruction/operand is only available in dsPIC33F devices.
6: This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax Mnemonic, Operands</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA</th>
<th>OB</th>
<th>SA</th>
<th>SB</th>
<th>OAB</th>
<th>SAB</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOPR</td>
<td>No operation</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>336</td>
</tr>
<tr>
<td>POP f</td>
<td>POP TOS to f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>337</td>
</tr>
<tr>
<td>POP Wdo</td>
<td>POP TOS to Wdo</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>338</td>
</tr>
<tr>
<td>POP.D Wnd</td>
<td>POP double from TOS to Wnd:Wnd + 1</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>339</td>
</tr>
<tr>
<td>POP.S</td>
<td>POP shadow registers</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>340</td>
</tr>
<tr>
<td>PUSH f</td>
<td>PUSH f to TOS</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>341</td>
</tr>
<tr>
<td>PUSH Wso</td>
<td>PUSH Wso to TOS</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>342</td>
</tr>
<tr>
<td>PUSH.D Wns</td>
<td>PUSH double Wns:Wns + 1 to TOS</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>343</td>
</tr>
<tr>
<td>PUSH.S</td>
<td>PUSH shadow registers</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>345</td>
</tr>
<tr>
<td>PWRSAV #lit1</td>
<td>Enter Power-saving mode</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>346</td>
</tr>
<tr>
<td>RCALL Expr</td>
<td>Relative call</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>347</td>
</tr>
<tr>
<td>RCALL Wn</td>
<td>Computed call</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>351</td>
</tr>
<tr>
<td>REPEAT #lit14(5)</td>
<td>Repeat next instruction (#lit14 + 1) times</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>355</td>
</tr>
<tr>
<td>REPEAT #lit15(5)</td>
<td>Repeat next instruction (#lit15 + 1) times</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>357</td>
</tr>
<tr>
<td>REPEAT Wn</td>
<td>Repeat next instruction (Wn + 1) times</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>359</td>
</tr>
<tr>
<td>RESET</td>
<td>Software device Reset</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>363</td>
</tr>
<tr>
<td>RETFIE</td>
<td>Return from interrupt enable</td>
<td>1</td>
<td>3 (2)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>365</td>
</tr>
<tr>
<td>RETILW #lit10.Wn</td>
<td>Return with lit10 in Wn</td>
<td>1</td>
<td>3 (2)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>367</td>
</tr>
<tr>
<td>RETURN</td>
<td>Return from subroutine</td>
<td>1</td>
<td>3 (2)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>371</td>
</tr>
<tr>
<td>RLC f {,WREG}</td>
<td>Destination = rotate left through Carry f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>373</td>
</tr>
<tr>
<td>RLC Ws,Wd</td>
<td>Wd = rotate left through Carry Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>375</td>
</tr>
<tr>
<td>RLNC f {,WREG}</td>
<td>Destination = rotate left (no Carry) f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>377</td>
</tr>
<tr>
<td>RLNC Ws,Wd</td>
<td>Wd = rotate left (no Carry) Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>379</td>
</tr>
<tr>
<td>RRC f {,WREG}</td>
<td>Destination = rotate right through Carry f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>381</td>
</tr>
<tr>
<td>RRC Ws,Wd</td>
<td>Wd = rotate right through Carry Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>383</td>
</tr>
<tr>
<td>RRNC f {,WREG}</td>
<td>Destination = rotate right (no Carry) f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>385</td>
</tr>
<tr>
<td>RRNC Ws,Wd</td>
<td>Wd = rotate right (no Carry) Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>387</td>
</tr>
<tr>
<td>SAC Acc,#Slit4,Wdo(2)</td>
<td>Store accumulator</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>389</td>
</tr>
</tbody>
</table>

**Legend:** 'O' set or cleared; 'D' may be cleared, but never set; 'U' may be set, but never cleared; '1' always set; '0' always cleared; — unchanged

**Note:**
1. SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
6. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax Mnemonic</th>
<th>Operands</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAC.R</td>
<td>Acc,#Slit4,Wd(2)</td>
<td>Store rounded Accumulator</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>391</td>
</tr>
<tr>
<td>SE</td>
<td>Ws,Wd</td>
<td>Wd = sign-extended Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>d</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>393</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SETM</td>
<td>f</td>
<td>f = 0xFFFF</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>395</td>
</tr>
<tr>
<td>SETM</td>
<td>WREG</td>
<td>WREG = 0xFFFF</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>395</td>
</tr>
<tr>
<td>SETM</td>
<td>Wd</td>
<td>Wd = 0xFFFF</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>396</td>
</tr>
<tr>
<td>SFTAC</td>
<td>Acc,#Slit6(2)</td>
<td>Arithmetic shift accumulator by Slat6</td>
<td>1</td>
<td>1</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>397</td>
</tr>
<tr>
<td>SFTAC</td>
<td>Acc,Wb(2)</td>
<td>Arithmetic shift accumulator by (Wb)</td>
<td>1</td>
<td>1</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>398</td>
</tr>
<tr>
<td>SL</td>
<td>f,(WREG)</td>
<td>Destination = arithmetic left shift f</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>399</td>
</tr>
<tr>
<td>SL</td>
<td>Ws,Wd</td>
<td>Wd = arithmetic left shift Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>401</td>
</tr>
<tr>
<td>SL</td>
<td>Wb,#lit4,Wnd</td>
<td>Wnd = left shift Wb by lit4</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>403</td>
</tr>
<tr>
<td>SL</td>
<td>Wb,Wns,Wnd</td>
<td>Wnd = left shift Wb by Wns</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>404</td>
</tr>
<tr>
<td>SUB</td>
<td>f,(WREG)</td>
<td>Destination = f – WREG</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>405</td>
</tr>
<tr>
<td>SUB</td>
<td>#lit10,Wh</td>
<td>Wh = Wh – lit10</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>406</td>
</tr>
<tr>
<td>SUB</td>
<td>Wb,#lit5,Wd</td>
<td>Wd = Wb – lit5</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>407</td>
</tr>
<tr>
<td>SUB</td>
<td>Wb,Ws,Wd</td>
<td>Wd = Wb – Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>408</td>
</tr>
<tr>
<td>SUB</td>
<td>Acc(2)</td>
<td>Subtract accumulators</td>
<td>1</td>
<td>1</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>410</td>
</tr>
<tr>
<td>SUBBB</td>
<td>f,(WREG)</td>
<td>destination = f – WREG – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>411</td>
</tr>
<tr>
<td>SUBBB</td>
<td>#lit10,Wh</td>
<td>Wh = Wh – lit10 – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>412</td>
</tr>
<tr>
<td>SUBBB</td>
<td>Wb,#lit5,Wd</td>
<td>Wd = Wb – lit5 – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>413</td>
</tr>
<tr>
<td>SUBBB</td>
<td>Wb,Ws,Wd</td>
<td>Wd = Wb – Ws – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>415</td>
</tr>
<tr>
<td>SUBBR</td>
<td>f,(WREG)</td>
<td>Destination = WREG – f – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>417</td>
</tr>
<tr>
<td>SUBBR</td>
<td>Wb,#lit5,Wd</td>
<td>Wd = lit5 – Wb – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>418</td>
</tr>
<tr>
<td>SUBBR</td>
<td>Wb,Ws,Wd</td>
<td>Wd = Ws – Wb – (C)</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>420</td>
</tr>
<tr>
<td>SWAP</td>
<td>Wh</td>
<td>Wh = byte or nibble swap Wh</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>÷</td>
<td>—</td>
<td>426</td>
</tr>
</tbody>
</table>

**Legend:**
- ÷ set or cleared; ÷ may be cleared, but never set; ÷ may be set, but never cleared;
- always set; ÷ always cleared; — unchanged

**Note:**
1: SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2: This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3: This instruction/operand is only available in PIC24E and dsPIC33E devices.
4: This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
5: This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
6: This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
### Table 7-2: Instruction Set Summary Table (Continued)

<table>
<thead>
<tr>
<th>Assembly Syntax Mnemonic, Operands</th>
<th>Description</th>
<th>Words</th>
<th>Cycles</th>
<th>OA(2)</th>
<th>OB(2)</th>
<th>SA(1,2)</th>
<th>SB(1,2)</th>
<th>OAB(2)</th>
<th>SAB(1,2)</th>
<th>DC</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBLRDH [Ws],Wd</td>
<td>Read high program word to Wd</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>427</td>
</tr>
<tr>
<td>TBLRDL [Ws],Wd</td>
<td>Read low program word to Wd</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>429</td>
</tr>
<tr>
<td>TBLWTH Ws,(Wd)</td>
<td>Write Ws to high program word</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>431</td>
</tr>
<tr>
<td>TBLWTL Ws,(Wd)</td>
<td>Write Ws to low program word</td>
<td>1</td>
<td>2</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>433</td>
</tr>
<tr>
<td>UINLK</td>
<td>Unlink Frame Pointer</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>435</td>
</tr>
<tr>
<td>XOR f {,WREG}</td>
<td>Destination = f.XOR. WREG</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>437</td>
</tr>
<tr>
<td>XOR #lit10,Wn</td>
<td>Wn = lit10 .XOR. Wn</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>438</td>
</tr>
<tr>
<td>XOR Wb,#lit5,Wd</td>
<td>Wd = Wb .XOR. lit5</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>439</td>
</tr>
<tr>
<td>XOR Wb,Ws,Wd</td>
<td>Wd = Wb .XOR. Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>440</td>
</tr>
<tr>
<td>ZE Ws,Wd</td>
<td>Wnd = zero-extended Ws</td>
<td>1</td>
<td>1</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>442</td>
</tr>
</tbody>
</table>

**Legend:**
- Ø set or cleared; ∅ may be cleared, but never set; 1' may be set, but never cleared; 1' always set; 0' always cleared; — unchanged

**Note:**
1. SA, SB and SAB are only modified if the corresponding saturation is enabled, otherwise unchanged.
2. This instruction/operand is only available in dsPIC30F, dsPIC33F, and dsPIC33E devices.
3. This instruction/operand is only available in PIC24E and dsPIC33E devices.
4. This instruction/operand is only available in dsPIC33E devices.
5. This instruction/operand is only available in PIC24F, PIC24H, dsPIC30F, and dsPIC33F devices.
6. This instruction/operand is only available in dsPIC30F and dsPIC33F devices.
7.3 REVISION HISTORY

Revision A (May 2005)
This is the initial release of this document.

Revision B (September 2005)
This revision incorporates all known errata at the time of this document update.

Revision C (February 2008)
This revision includes the following corrections and updates:
- Instruction Updates:
  - Updated BRA instruction (see “BRA”)
  - Updated DIVF instruction (see “DIVF”)
  - Updated DO instruction (see “DO”)
  - Updated SUB instruction (see “SUB”)

Revision D (November 2009)
This revision includes the following corrections and updates:
- Document has been completely redesigned to accommodate all current 16-bit families: dsPIC30F, dsPIC33F, PIC24F and PIC24H

Revision E (June 2010)
This revision includes the following corrections and updates:
- Information specific to dsPIC33E and PIC24E devices has been added throughout the document

Revision F (July 2011)
This revision includes the following corrections and updates:
- Added a new section “Built-in Functions”
- Added and updated the cross-references throughout the document
- Updated the bit characteristics from U to U-0 in Register 2-4 and Register 2-6
- Added a note throughout the document specifying the requirement of an additional cycle for read and read-modify-write operations on non-CPU special function registers in dsPIC33E and PIC24E devices
- Updates to formatting and minor text changes were incorporated throughout the document
INDEX

Symbols

__builtin_addab .......................................................... 447
__builtin_big ............................................................ 447, 448
__builtin_divmod .......................................................... 454
__builtin_divsd ............................................................ 454
__builtin_edsoffset .......................................................... 458
__builtin_edspage .......................................................... 458
__builtin_mac .............................................................. 460
__builtin_modsd ............................................................ 462
__builtin_modud ............................................................ 462
__builtin_mpy .............................................................. 464
__builtin_mpyhn .............................................................. 465
__builtin_mul .............................................................. 465
__builtin_muluu .............................................................. 469
__builtin_mul .............................................................. 470
__builtin_psvoffset .......................................................... 470
__builtin_sac .............................................................. 472
__builtin_subab .............................................................. 475
__builtin_tbladdress .......................................................... 475
__builtin_tblwt ............................................................. 479
__builtin_tblwth ............................................................. 479
__builtin_tblwlt ............................................................. 479
_Accumulator A, Accumulator B ......................................... 19
_Accumulator Access ....................................................... 84
_Accumulator Selection ..................................................... 97
_Accumulator Usage ........................................................ 83
_Addressing Modes for Wd Destination Register .................. 95
_Addressing Modes for Ws Source Register ......................... 95
_Assigned Working Register Usage ..................................... 78

_Built-In Functions

__builtin_addab .......................................................... 447
__builtin_big ............................................................ 447, 448
__builtin_divmod .......................................................... 454
__builtin_divsd ............................................................ 454
__builtin_edsoffset .......................................................... 458
__builtin_edspage .......................................................... 458
__builtin_mac .............................................................. 460
__builtin_modsd ............................................................ 462
__builtin_modud ............................................................ 462
__builtin_mpy .............................................................. 464
__builtin_mpyhn .............................................................. 465
__builtin_mul .............................................................. 465
__builtin_muluu .............................................................. 469
__builtin_mul .............................................................. 470
__builtin_psvoffset .......................................................... 470
__builtin_sac .............................................................. 472
__builtin_subab .............................................................. 475
__builtin_tbladdress .......................................................... 475
__builtin_tblwt ............................................................. 479
__builtin_tblwth ............................................................. 479
__builtin_tblwlt ............................................................. 479

_Immediate Addressing ..................................................... 59
_Indirect Addressing with Effective Address Update .............. 55
_Indirect Addressing with Register Offset ......................... 56
_Legal Word Move Operations ........................................... 67
_MAC Accumulator WB Syntax ........................................... 87
_MAC Prefetch Syntax ..................................................... 86
_Move with Literal Offset Instructions ................................ 56
_MSC Instruction with Two Prefetches and Accumulator Write Back .......................................................... 87
_Normalizing with FBCL .................................................... 90
_Register Direct Addressing ............................................. 54
_Sample Byte Math Operations ........................................... 65
_Sample Byte Move Operations .......................................... 64
_Scaling with FBCL ........................................................ 89
_S tack Pointer Usage ...................................................... 71
_Unsigned f and WREG Multiply (Legacy MULWF Instruction) .......................................................... 80
_U sing 10-bit Literals for Byte Operands ............................. 69
_U sing the Default Working Register WREG ...................... 79
_Conditional Branch Instructions ....................................... 76
_Core Control Register .................................................... 24

_D Data Addressing Mode Tree .......................................... 59
_Data Addressing Modes ................................................ 52
_DCOUNT Register .......................................................... 20
_Default Working Register (WREG) .................................... 18, 79
_Development Support ..................................................... 8
_DOEND Register ............................................................ 21
_DOSTART Register .......................................................... 20
_DSP Accumulator Instructions ......................................... 88
_DSP Data Formats .......................................................... 81
_DSP MAC Indirect Addressing Modes ................................ 57
_DSP MAC Instructions .................................................... 84

_F File Register Addressing ............................................... 52

_I Immediate Addressing ..................................................... 58
_Operands in the Instruction Set ....................................... 58
_Implied DSP Operands .................................................... 78
_Implied Frame and Stack Pointer ..................................... 78
_Instruction Bit Map .......................................................... 464
_Instruction Description Example ...................................... 98
_Instruction Descriptions .................................................. 99

_ADD (16-bit Signed Add to Accumulator) .......................... 104
_ADD (Add Accumulators) ................................................... 103
_ADD (Add f to WREG) ...................................................... 99
_ADD (Add Literal to Wn) .................................................. 100
_ADD (Add Wb to Short Literal) ......................................... 101
_ADD (Add Wb to Ws) ....................................................... 102
_ADDC (Add f to WREG with Carry) ................................... 106
_ADDC (Add Literal to Wn with Carry) ................................ 107
_ADDC (Add Wb to Short Literal with Carry) ....................... 108
_ADDC (Add Wb to Ws with Carry) ...................................... 110
_AND (AND f and WREG) ................................................... 112
_AND (AND (AND Literal and Wn) ........................................ 113
_AND (AND (AND Literal and Wn) ........................................ 114
_AND (AND (AND Literal and Wn) ........................................ 115
_AND (AND (AND Literal and Wn) ........................................ 116
_AND (AND (AND Literal and Wn) ........................................ 117
_AND (AND (AND Literal and Wn) ........................................ 118
_AND (AND (AND Literal and Wn) ........................................ 119

© 2005-2011 Microchip Technology Inc. DS70157F-page 497
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCLR (Bit Clear in Ws)</td>
<td>124</td>
</tr>
<tr>
<td>BCLR.B (Bit Clear f)</td>
<td>123</td>
</tr>
<tr>
<td>BRA (Branch Unconditionally)</td>
<td>126</td>
</tr>
<tr>
<td>BRA (Computed Branch)</td>
<td>128, 129</td>
</tr>
<tr>
<td>BRA C (Branch if Carry)</td>
<td>130</td>
</tr>
<tr>
<td>BRA GE (Branch if Signed Greater Than or Equal)</td>
<td>132</td>
</tr>
<tr>
<td>BRA GEU (Branch if Unsigned Greater Than or Equal)</td>
<td>134</td>
</tr>
<tr>
<td>BRA GT (Branch if Signed Greater Than)</td>
<td>135</td>
</tr>
<tr>
<td>BRA GTU (Branch if Unsigned Greater Than)</td>
<td>136</td>
</tr>
<tr>
<td>BRA LE (Branch if Signed Less Than or Equal)</td>
<td>137</td>
</tr>
<tr>
<td>BRA LEU (Branch if Unsigned Less Than or Equal)</td>
<td>138</td>
</tr>
<tr>
<td>BRA LT (Branch if Signed Less Than)</td>
<td>139</td>
</tr>
<tr>
<td>BRA LTB (Branch if Not Carry)</td>
<td>142</td>
</tr>
<tr>
<td>BRA LTU (Branch if Unsigned Less Than)</td>
<td>140</td>
</tr>
<tr>
<td>BRA N (Branch if Negative)</td>
<td>141</td>
</tr>
<tr>
<td>BRA NN (Branch if Not Negative)</td>
<td>143</td>
</tr>
<tr>
<td>BRA NOV (Branch if Not Overflow)</td>
<td>144</td>
</tr>
<tr>
<td>BRA NZ (Branch if Not Zero)</td>
<td>145</td>
</tr>
<tr>
<td>BRA OA (Branch if Overflow Accumulator A)</td>
<td>146</td>
</tr>
<tr>
<td>BRA OB (Branch if Overflow Accumulator B)</td>
<td>147</td>
</tr>
<tr>
<td>BRA OV (Branch if Overflow)</td>
<td>148</td>
</tr>
<tr>
<td>BRA SA (Branch if Saturation Accumulator A)</td>
<td>149</td>
</tr>
<tr>
<td>BRA SB (Branch if Saturation Accumulator B)</td>
<td>150</td>
</tr>
<tr>
<td>BRA Z (Branch if Zero)</td>
<td>151</td>
</tr>
<tr>
<td>BSET (Bit Set f)</td>
<td>152</td>
</tr>
<tr>
<td>BSET (Bit Set in Ws)</td>
<td>153</td>
</tr>
<tr>
<td>BSW (Write Bit in Ws)</td>
<td>155</td>
</tr>
<tr>
<td>BTG (Bit Toggle f)</td>
<td>157</td>
</tr>
<tr>
<td>BTG (Bit Toggle in Ws)</td>
<td>158</td>
</tr>
<tr>
<td>BTSC (Bit Test f, Skip if Clear)</td>
<td>160</td>
</tr>
<tr>
<td>BTSS (Bit Test f, Skip if Set)</td>
<td>164</td>
</tr>
<tr>
<td>BTST (Bit Test f)</td>
<td>166</td>
</tr>
<tr>
<td>BTST (Bit Test in Ws)</td>
<td>169, 171</td>
</tr>
<tr>
<td>BTSTS (Bit Test/Set in Ws)</td>
<td>173</td>
</tr>
<tr>
<td>CALL (Call Indirect Subroutine)</td>
<td>180, 181</td>
</tr>
<tr>
<td>CALL (Call Subroutine)</td>
<td>177, 178</td>
</tr>
<tr>
<td>CALL.L (Call Indirect Subroutine Long)</td>
<td>183</td>
</tr>
<tr>
<td>CBSLT (Signed Compare Wb with Wn, Branch if Less Than)</td>
<td>205</td>
</tr>
<tr>
<td>CLR (Clear Accumulator, Prefetch Operands)</td>
<td>186</td>
</tr>
<tr>
<td>CLR (Clear f or WREG)</td>
<td>184</td>
</tr>
<tr>
<td>CLR (Clear Wd)</td>
<td>185</td>
</tr>
<tr>
<td>CLRWT (Clear Watchdog Timer)</td>
<td>188</td>
</tr>
<tr>
<td>COM (Complement f)</td>
<td>189</td>
</tr>
<tr>
<td>COM (Complement Ws)</td>
<td>190</td>
</tr>
<tr>
<td>CP (Compare f with WREG, Set Status Flags)</td>
<td>191</td>
</tr>
<tr>
<td>CP (Compare Wb with lit5, Set Status Flags)</td>
<td>192</td>
</tr>
<tr>
<td>CP (Compare Wb with lit8, Set Status Flags)</td>
<td>193</td>
</tr>
<tr>
<td>CP (Compare Wb with Ws, Set Status Flags)</td>
<td>194</td>
</tr>
<tr>
<td>CP0 (Compare f with 0x0, Set Status Flags)</td>
<td>196</td>
</tr>
<tr>
<td>CP0 (Compare Ws with 0x0, Set Status Flags)</td>
<td>197</td>
</tr>
<tr>
<td>CPB (Compare f with WREG using Borrow, Set Status Flags)</td>
<td>198</td>
</tr>
<tr>
<td>CPB (Compare Ws with Wb using Borrow)</td>
<td>199, 200</td>
</tr>
<tr>
<td>CPB (Compare Ws with Wb using Borrow, Set Status Flags)</td>
<td>201</td>
</tr>
<tr>
<td>CPB (Compare Ws with Wb using Borrow)</td>
<td>202</td>
</tr>
<tr>
<td>CPB (Compare Ws with Wb using Borrow)</td>
<td>203</td>
</tr>
<tr>
<td>CPBGT (Signed Compare Wb with Wn, Branch if Greater Than)</td>
<td>204</td>
</tr>
<tr>
<td>CPBGE (Signed Compare Wb with Wn, Branch if Not Equal)</td>
<td>206</td>
</tr>
<tr>
<td>CPSEQ (Compare Wb with Wn, Skip if Equal)</td>
<td>207, 208</td>
</tr>
<tr>
<td>CPSGT (Signed Compare Wb with Wn, Skip if Greater Than)</td>
<td>210</td>
</tr>
<tr>
<td>CPSGT (Signed Compare Wb with Wn, Skip if Greater Than)</td>
<td>211</td>
</tr>
<tr>
<td>CPSGT (Signed Compare Wb with Wn, Skip if Greater Than)</td>
<td>212, 213</td>
</tr>
<tr>
<td>CPSNE (Signed Compare Wb with Wn, Skip if Not Equal)</td>
<td>214, 215</td>
</tr>
<tr>
<td>DAW.B (Decimal Adjust Wn)</td>
<td>216</td>
</tr>
<tr>
<td>DEC (Decrement f)</td>
<td>217</td>
</tr>
<tr>
<td>DEC (Decrement Ws)</td>
<td>218</td>
</tr>
<tr>
<td>DEC2 (Decrement f by 2)</td>
<td>220</td>
</tr>
<tr>
<td>DEC2 (Decrement Ws by 2)</td>
<td>221</td>
</tr>
<tr>
<td>DIV.S (Signed Integer Divide)</td>
<td>224</td>
</tr>
<tr>
<td>DIV.U (Unsigned Integer Divide)</td>
<td>226</td>
</tr>
<tr>
<td>DIVF (Fractional Divide)</td>
<td>228</td>
</tr>
<tr>
<td>DO (Initialize Hardware Loop)</td>
<td>230, 233</td>
</tr>
<tr>
<td>DO (Initialize Hardware Loop Wn)</td>
<td>235, 237</td>
</tr>
<tr>
<td>ED (Euclidean Distance, No Accumulate)</td>
<td>239</td>
</tr>
<tr>
<td>EDAC (Euclidean Distance)</td>
<td>241</td>
</tr>
<tr>
<td>EXCH (Exchange Wns and Wnd)</td>
<td>243</td>
</tr>
<tr>
<td>FBCL (Find First Bit Change from Left)</td>
<td>244</td>
</tr>
<tr>
<td>FF1L (Find First One from Left)</td>
<td>246</td>
</tr>
<tr>
<td>FF1R (Find First One from Right)</td>
<td>248</td>
</tr>
<tr>
<td>GOTO (Unconditional Indirect Jump)</td>
<td>250, 251</td>
</tr>
<tr>
<td>GOTO (Unconditional Jump)</td>
<td>250</td>
</tr>
<tr>
<td>GOTO.L (Unconditional Indirect Jump Long)</td>
<td>253</td>
</tr>
<tr>
<td>INC (Increment f)</td>
<td>254</td>
</tr>
<tr>
<td>INC (Increment Ws)</td>
<td>255</td>
</tr>
<tr>
<td>INC2 (Increment f by 2)</td>
<td>257</td>
</tr>
<tr>
<td>INC2 (Increment Ws by 2)</td>
<td>258</td>
</tr>
<tr>
<td>IOR (Inclusive OR f and WREG)</td>
<td>260</td>
</tr>
<tr>
<td>IOR (Inclusive OR f and WREG)</td>
<td>261</td>
</tr>
<tr>
<td>IOR (Inclusive OR f and WREG)</td>
<td>262</td>
</tr>
<tr>
<td>IOR (Inclusive OR f and WREG)</td>
<td>263</td>
</tr>
<tr>
<td>LAC (Load Accumulator)</td>
<td>265</td>
</tr>
<tr>
<td>LNK (Allocate Stack Frame)</td>
<td>267, 268</td>
</tr>
<tr>
<td>LS (Logical Shift Right)</td>
<td>273</td>
</tr>
<tr>
<td>LS (Logical Shift Right)</td>
<td>274</td>
</tr>
<tr>
<td>LS (Logical Shift Right)</td>
<td>269</td>
</tr>
<tr>
<td>LS (Logical Shift Right Ws)</td>
<td>271</td>
</tr>
<tr>
<td>MAC (Multiply and Accumulate)</td>
<td>275</td>
</tr>
<tr>
<td>MAC (Multiply and Accumulate)</td>
<td>277</td>
</tr>
<tr>
<td>MOV (Move 16-bit Literal to Wn)</td>
<td>284</td>
</tr>
<tr>
<td>MOV (Move f to Destination)</td>
<td>279</td>
</tr>
<tr>
<td>MOV (Move f to Wnd)</td>
<td>281</td>
</tr>
<tr>
<td>MOV (Move Wns to [Wd with offset])</td>
<td>286</td>
</tr>
<tr>
<td>MOV (Move Wns to f)</td>
<td>282</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>287</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>285</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>285</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>283</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>289</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>291</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>292</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>293</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>295</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>297</td>
</tr>
<tr>
<td>MOV (Move Wns to Wd)</td>
<td>299</td>
</tr>
<tr>
<td>MUL (Integer Unsigned Multiply f and WREG)</td>
<td>301</td>
</tr>
<tr>
<td>MUL (Integer Unsigned Multiply f and WREG)</td>
<td>303</td>
</tr>
<tr>
<td>MUL.SS (Integer 16x8-bit Signed Multiply with Accumulator Destination)</td>
<td>307</td>
</tr>
<tr>
<td>MUL.SS (Integer 16x16-bit Signed Multiply)</td>
<td>305</td>
</tr>
</tbody>
</table>
MUL.UU (Integer 16x16-bit Signed-Unsigned Multiply) .......................... 315
MUL.UU (Integer 16x16-bit Signed-Unsigned Multiply with Accumulator Destination) ................. 323
MUL.UU (Integer 16x16-bit Signed-Unsigned Short Literal Multiply) ............................................ 329
MUL.WS (Integer 16x16-bit Signed Multiply with Accumulator Destination) ... 331
MUL.WU (Integer 16x16-bit Signed-Unsigned Multiply) .......................................................... 332
NEG (Negate Accumulator) ...................................................... 335
NEG (Negate f) ................................................................. 333
NEG (Negate Ws) ............................................................... 333
NOP (No Operation) ................................................................ 336
NOPR (No Operation) ........................................................... 336
POP (Pop TOP to f) ............................................................... 337
POP (Pop TOP to Wd) ............................................................ 338
POPD (Double Pop TOP to Wnd/ Wnd+1) .................................................................................. 339
POPS (Pop Shadow Registers) ................................................ 340
PUSH (Push f to TOP) ............................................................ 341
PUSH (Push Ws to TOP) .......................................................... 342
PUSHD (Double Push Ws/ Ws+1 to TOP) .................................................................................. 343
PUSH.S (Push Shadow Registers) ................................................ 344
PWSAV (Enter Power Saving Mode) .............................................. 346
RCALL (Computed Relative Call) .................................................. 351, 353
RCALL (Relative Call) ............................................................... 347, 349
REPEAT (Repeat Next Instruction 'lit114 + 1' Times) 355
REPEAT (Repeat Next Instruction 'lit115 + 1' Times) 357
REPEAT (Repeat Next Instruction 'lit115 + 1' Times) 359, 361
RESET (Reset) ....................................................................... 363
RETFIE (Return from Interrupt) .................................................. 365, 366
RETLLW (Return with Literal in Wn) ................................................ 367, 369
RETURN (Return) .................................................................. 371, 372
RLC (Rotate Left f through Carry) .............................................. 373
RLC (Rotate Left Ws through Carry) ......................................... 375
RLNC (Rotate Left f without Carry) ............................................. 377
RLNC (Rotate Left Ws without Carry) ......................................... 379
RRC (Rotate Right f through Carry) ........................................... 381
RRC (Rotate Right Ws through Carry) ......................................... 383
RRNC (Rotate Right f without Carry) ........................................... 385
RRNC (Rotate Right Ws without Carry) ......................................... 387
SAC (Store Accumulator) ......................................................... 389
SAC.R (Store Rounded Accumulator) ........................................... 391
SE (Sign-Extend Ws) ............................................................... 393
SE (Sign-Extend Ws) ............................................................... 393
SETM (Set f or WREG) .......................................................... 395
SETM (Set Ws) ..................................................................... 396
SFTAC (Arithmetic Shift Accumulator by lit5) .......................... 397
SFTAC (Arithmetic Shift Accumulator by Wb) .......................... 398
SL (Shift Left by Short Literal) .................................................. 403
SL (Shift Left by Wn) ............................................................. 404
SL (Shift Left f) .................................................................... 399
SL (Shift Left Ws) ................................................................. 401
SUB (Subtract Accumulators) ................................................... 410
SUB (Subtract Literal from Wn) .................................................. 406
SUB (Subtract Short Literal from Wb) ...................................... 407
SUB (Subtract WREG from f) .................................................... 405
SUB (Subtract Ws from Wb) ..................................................... 408
SUBB (Subtract Short Literal from Wb with Borrow) ............... 413
SUBB (Subtract Wn from Literal with Borrow) .......................... 412
SUBBB (Subtract WREG and Carry bit from f) ...................... 411
SUBBB (Subtract Ws from Wb with Borrow) ............................. 416
SUBBR (Subtract f from WREG with Borrow) .......... 417
SUBBR (Subtract Wb from Short Literal with Borrow) 418
SUBBR (Subtract g from WREG) .............................................. 420
SUBBR (Subtract Wb from Short Literal) ................................. 423
SUBBR (Subtract Wb from Ws) ............................................... 424
SWAP (Byte or Nibble Swap Wn) .............................................. 426
TBLRDH (Table Read High) ..................................................... 427
TBLRLD (Table Read Low) ..................................................... 429
TBLWTH (Table Write High) .................................................... 431
TBLWLW (Table Write Low) .................................................... 433
JUMP (Jump to Label) ............................................................. 435, 436
JUMP (Jump to Label) ............................................................. 437
JUMP (Jump to Label) ............................................................. 438
JUMP (Jump to Label) ............................................................. 439
JUMP (Jump to Label) ............................................................. 440
ZE (Zero-Extend Wn) ............................................................. 442
Instruction Encoding Field Descriptors Introduction ................. 94
Instruction Set Overview .......................................................... 38
Bit Instructions .................................................................... 45
Compare/Skip Instructions ....................................................... 46
Control Instructions ............................................................... 49
DSP Instructions ................................................................. 50
dsPIC30F/33F Instruction Groups ............................................. 38
Logic Instructions ................................................................. 43
Math Instructions ................................................................. 44
Move Instructions ................................................................. 40
Program Flow Instructions ..................................................... 47
Rotate/Shift Instructions ......................................................... 44
Shadow/Stack Instructions ...................................................... 49
Instruction Set Summary Table .................................................. 486
Instruction Set Symbols ........................................................... 8
(text) .................................................................................. 8
]{label:} ................................................................................. 8
} ..................................................................................... 8
(text) .................................................................................. 8

### Worldwide Sales and Service

#### AMERICAS

**Corporate Office**  
2355 West Chandler Blvd.  
Chandler, AZ 85224-6199  
Tel: 480-792-7200  
Fax: 480-792-7277  
Technical Support: [http://www.microchip.com/support](http://www.microchip.com/support)  
Web Address: www.microchip.com

**Atlanta**  
Duluth, GA  
Tel: 678-957-9614  
Fax: 678-957-1455

**Boston**  
Westborough, MA  
Tel: 774-447-0087  
Fax: 774-447-0088

**Chicago**  
Itasca, IL  
Tel: 630-285-0071  
Fax: 630-285-0075

**Cleveland**  
Independence, OH  
Tel: 216-447-0464  
Fax: 216-447-0643

**Dallas**  
Addison, TX  
Tel: 972-818-7423  
Fax: 972-818-2924

**Denver**  
Farmington Hills, MI  
Tel: 248-538-2250  
Fax: 248-538-2260

**Indianapolis**  
Noblesville, IN  
Tel: 317-773-8323  
Fax: 317-773-5453

**Los Angeles**  
Mission Viejo, CA  
Tel: 949-462-9523  
Fax: 949-462-9608

**Santa Clara**  
Santa Clara, CA  
Tel: 408-961-6444  
Fax: 408-961-6445

**Toronto**  
Mississauga, Ontario, Canada  
Tel: 905-673-0699  
Fax: 905-673-6509

#### ASIA/PACIFIC

**Asia Pacific Office**  
Suites 3707-14, 37th Floor  
Tower 6, The Gateway  
Harbour City, Kowloon  
Hong Kong  
Tel: 852-2401-1120  
Fax: 852-2401-3431

**Australia - Sydney**  
Tel: 61-2-9868-6733  
Fax: 61-2-9868-6755

**China - Beijing**  
Tel: 86-10-8569-7000  
Fax: 86-10-8524-2104

**China - Chengdu**  
Tel: 86-28-8665-5511  
Fax: 86-28-8665-7889

**China - Chongqing**  
Tel: 86-23-8980-9588  
Fax: 86-23-8980-9500

**China - Hangzhou**  
Tel: 86-571-2819-3180  
Fax: 86-571-2819-3189

**China - Hong Kong SAR**  
Tel: 852-2401-1200  
Fax: 852-2401-3431

**China - Nanjing**  
Tel: 86-25-8473-2600  
Fax: 86-25-8473-2601

**China - Qingdao**  
Tel: 86-532-8502-7355  
Fax: 86-532-8502-7205

**China - Shanghai**  
Tel: 86-21-5407-5533  
Fax: 86-21-5407-5066

**China - Shenyang**  
Tel: 86-24-2334-2829  
Fax: 86-24-2334-2393

**China - Shenzhen**  
Tel: 86-755-8203-2660  
Fax: 86-755-8203-1760

**China - Wuhan**  
Tel: 86-27-5980-5300  
Fax: 86-27-5980-5118

**China - Xi'an**  
Tel: 86-29-8833-7252  
Fax: 86-29-8833-7256

**China - Xiamen**  
Tel: 86-592-2388130  
Fax: 86-592-2388130

**China - Zhuhai**  
Tel: 86-756-3210040  
Fax: 86-756-3210049

#### ASIA/PACIFIC

**India - Bangalore**  
Tel: 91-80-3090-4444  
Fax: 91-80-3090-4123

**India - New Delhi**  
Tel: 91-11-4160-8631  
Fax: 91-11-4160-8632

**India - Pune**  
Tel: 91-20-2566-1512  
Fax: 91-20-2566-1513

**Japan - Yokohama**  
Tel: 81-45-471-6166  
Fax: 81-45-471-6122

**Korea - Daegu**  
Tel: 82-53-744-4301  
Fax: 82-53-744-4302

**Korea - Seoul**  
Tel: 82-2-554-7200  
Fax: 82-2-558-5932 or 82-2-558-5934

**Malaysia - Kuala Lumpur**  
Tel: 60-3-6201-9857  
Fax: 60-3-6201-9859

**Malaysia - Penang**  
Tel: 60-4-227-8870  
Fax: 60-4-227-4068

**Philippines - Manila**  
Tel: 63-2-634-9065  
Fax: 63-2-634-9069

**Singapore**  
Tel: 65-6334-8870  
Fax: 65-6334-8850

**Taiwan - Hsin Chu**  
Tel: 886-3-6578-300  
Fax: 886-3-6578-370

**Taiwan - Kaohsiung**  
Tel: 886-7-231-7830  
Fax: 886-7-330-9305

**Taiwan - Taipei**  
Tel: 886-2-2500-6610  
Fax: 886-2-2508-0102

**Thailand - Bangkok**  
Tel: 66-2-694-1351  
Fax: 66-2-694-1350

#### EUROPE

**Austria - Wels**  
Tel: 43-7242-2244-39  
Fax: 43-7242-2244-393

**Denmark - Copenhagen**  
Tel: 45-4450-2828  
Fax: 45-4485-2829

**France - Paris**  
Tel: 33-1-69-53-63-20  
Fax: 33-1-69-30-90-79

**Germany - Munich**  
Tel: 49-89-627-144-0  
Fax: 49-89-627-144-44

**Italy - Milan**  
Tel: 39-0331-742611  
Fax: 39-0331-466781

**Netherlands - Drunen**  
Tel: 31-416-690399  
Fax: 31-416-690340

**Spain - Madrid**  
Tel: 34-91-708-08-90  
Fax: 34-91-708-08-91

**UK - Wokingham**  
Tel: 44-118-921-5869  
Fax: 44-118-921-5820

05/02/11