Computer Organization & Microprocessors

Version 4.6b released on July 2018
First printed on March 2007

Background and Acknowledgements

This material is intended for the second course in digital systems focus on Computer Organization and Microprocessors. The content is derived from the author's educational, engineering and management career, and teaching experience.

I would like to extend special thanks to the many students and colleagues for their contributions in making this material a more effective learning tool.

Further, I would invite the reader to forward corrections, additional topics, examples and problems to me for future updates.

Thanks,
Izad Khormaee
www.EngrCS.com

Microchip material used by permission:
Excerpts from Microchip Technology Inc.'s PIC microprocessor Datasheets, application notes and other resources has been included with permission from Microchip Technology Inc., November 9, 2006. The following are registered trademarks of Microchip Technology Incorporated in the U.S.A. and other countries: Accuron, dsPIC, KEELOQ, microID, MPLAB, PIC, PICmicro, PICSTART, PowerSmart, PRO MATE, rfPIC and SmartShunt; as well as the Microchip logo, the Microchip name and logo, and the KEELOQ logo.

Table of Contents

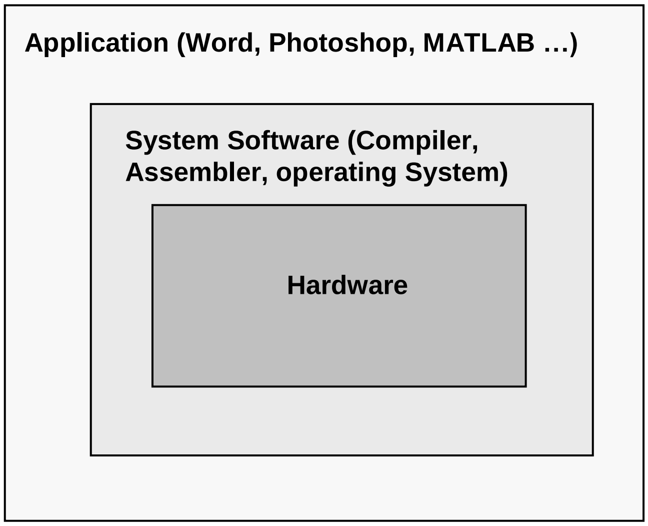
CHAPTER 1. INTRODUCTIONS

Key Concepts and Overview

- Overview of components, subsystems and interfaces
- Processor Design Considerations
- Computing systems Classification
- Historical Perspective and Trends
- What’s next…
- Integrated Development Environment (IDE)
- Additional Resources

1.1. Overview of components, subsystems and interfaces

- Computer Layers


Application (Word, Photoshop, MATLAB …)
System Software (Compiler, Assembler, operating System)
Hardware

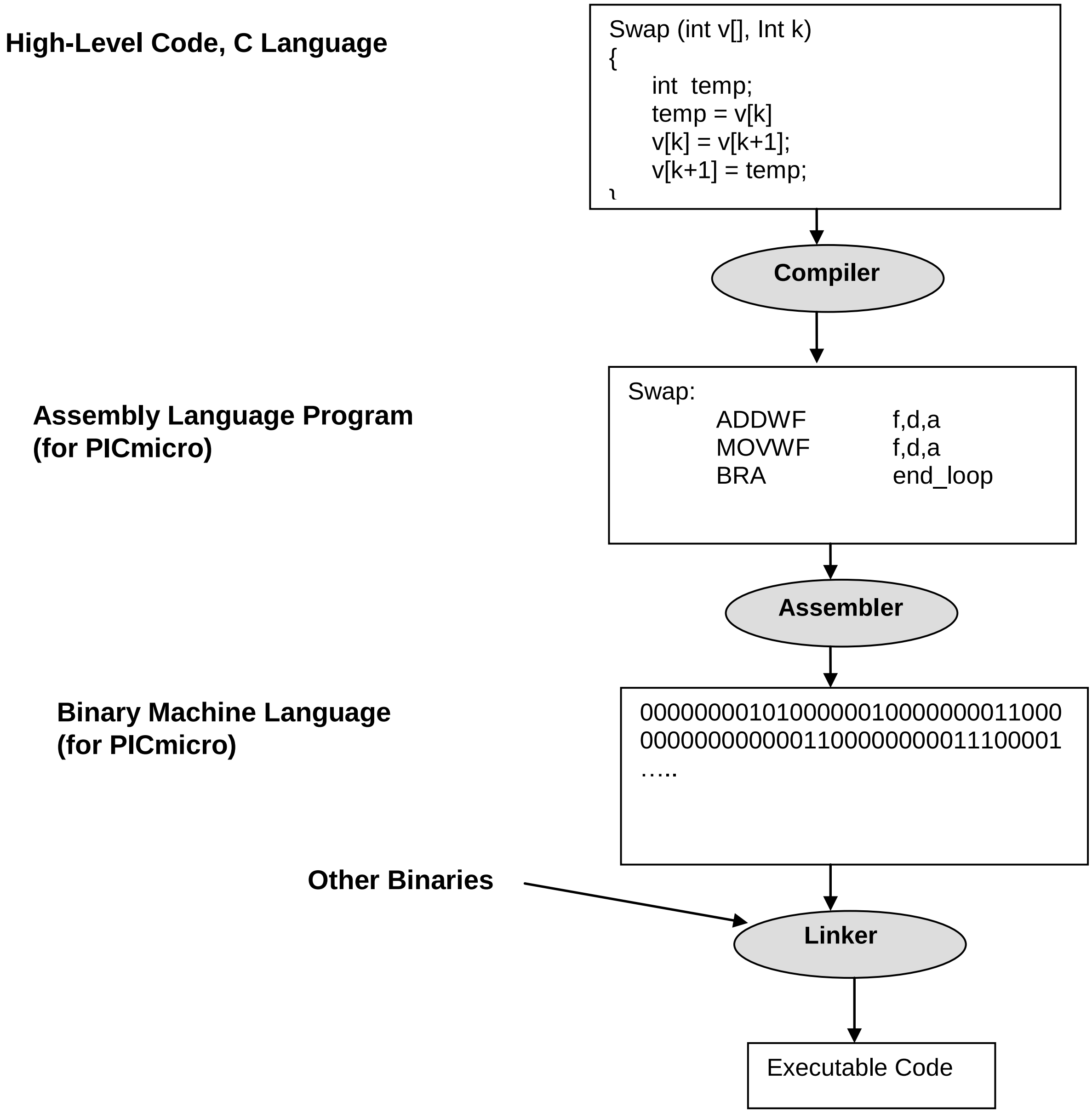
- Operating System roles

  Operating Systems are basically the system resource managers and controllers. The common Operating Systems include Windows, Linux and Mac OS.

  The Operating System roles include:

  - Handle basic Input/output
  - Start and stop applications
  - Allocate storage, memory and processor – In general, manage the use of computer resources among the applications (active processes)

- Steps from High Level Language (C, C++, C#, Java, …) to executable code



```
Swap (int v[], Int k)
{
      int  temp;
      temp = v[k]
      v[k] = v[k+1];
      v[k+1] = temp;
}
```

```
Swap:
            ADDWF          f,d,a
            MOVWF          f,d,a
            BRA            end_loop
```

```
0000000010100000010000000011000
0000000000001100000000011100001
…..
```

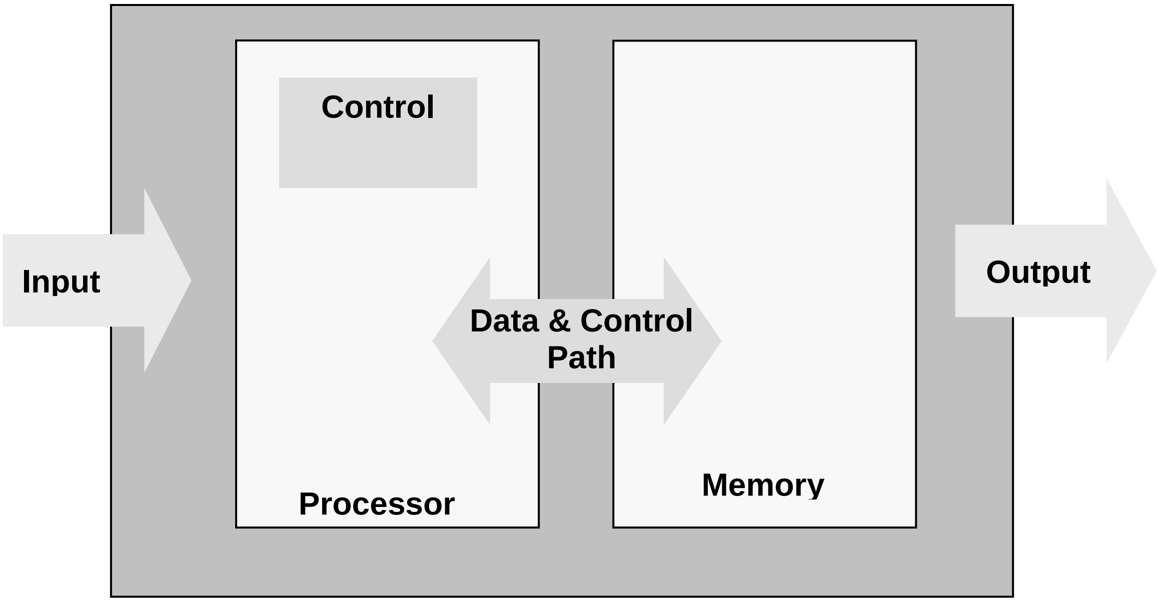
As it can be seen from the above figure, high level languages such as C make it easier for human programmers to read and write the program. Improved code readability increases the programmers' productivity which has led to popularity of high level language amongst Software engineers and businesses.

Compilers and assemblers are used to translate the high level language into Machine language which can be executed on the processor. Programmers typically use Assembly language to optimize the parts of code that have high impact on performance.

Finally, the linker allows integration of functions which are previously written or functions from available libraries.

- Computer Architecture
  Architecture defines the flow of data and patterns of the system. In general, a computer system can

be described using the following architectural diagram:



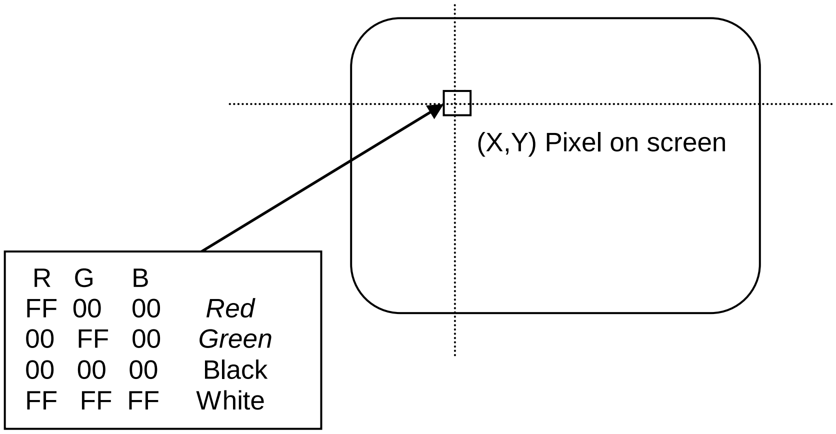
- Computer Components

  - Input
    Keyboard, Mouse, Microphone, Joy stick and Video camera are examples of input devices.

  - Output
    One of the main forms of output is presenting the information on a display. There are a large number of display types. Examples include:

    - Cathode Ray Tube-CRT
    - Liquid Crystal Display-LCD
    - Electro Luminescent-EL
    - Plasma

    Typically, displays are memory-mapped which means there is a memory location for every dot on the display. The value in the memory controls the color and intensity of the corresponding dot. Collections of dots may be used to form an image, text or other display elements. The following diagram presents a few examples which show the relationship between data and a RGB (Red, Green, and Blue) memory-mapped display:



R	G	B	
FF	00	00	*Red*
00	FF	00	*Green*
00	00	00	Black
FF	FF	FF	White

- Networking is another important type of Input/Output
  The following list categorizes networking based on the geographical coverage:

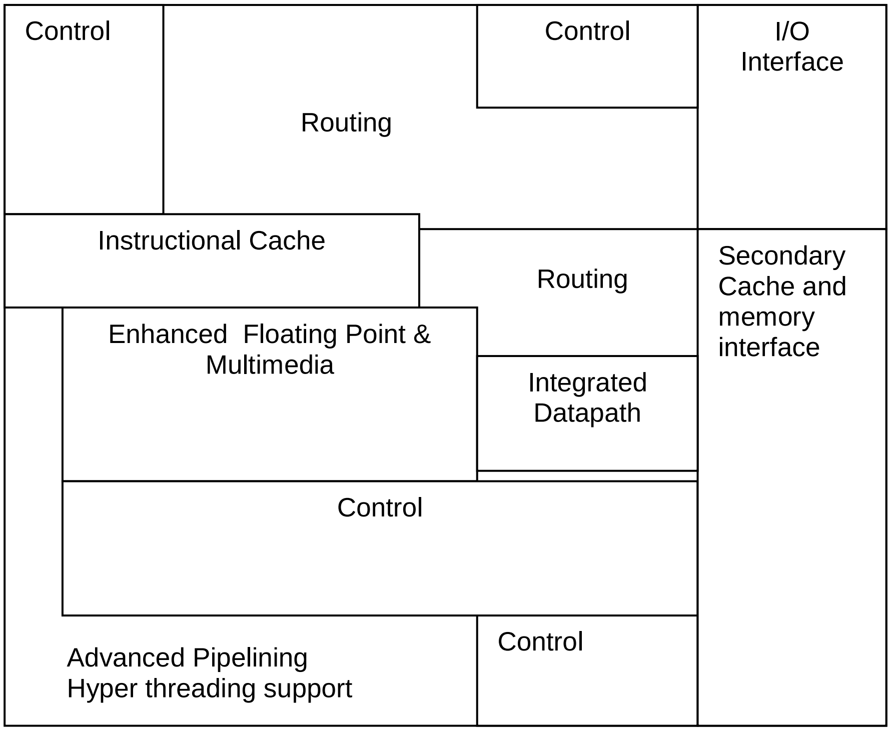
  - Personal Networking (PN)
    PN coverage is around one person, desk or room. Some examples are:

    - USB
    - Blue tooth
    - RS232 – Serial bus
    - Parallel Bus

  - Local Area Networking (LAN)
    IEEE 802.3 is the most common LAN type in use within a building or small campus. The light weight wired implementation is the most common type of Local area networking in use. The wireless implementation of this network type has also grown dramatically.

  - Wide Area networking (WAN)
    WAN coverage is across cities, countries or continents. WAN service is typically leased from a telecommunication company. One could say that the telephone system is a form of WAN.

- Processor
  Processor or Central Processing Unit (CPU) is the program execution unit of the computer and can be thought of as the brain. The following diagram shows the most common elements or functional blocks of a processor:



- Memory/Storage
  - Primary Memory

    Typically referred to as solid state memory. It is smaller in size but faster (Access time in nanoseconds) and is used during program execution. (512 Mbytes for $100 in 2005)

    - Volatile memory

      This is the most common type of memory where data is retained as long as power is applied. There most common types of volatile memory are:

      - CACHE

        Cache is the fastest memory and it is used for frequently accessed instructions and data. It is intermediate memory between processor and memory/storage.

      - DRAM

        Dynamic Random Access Memory (DRAM), is the second fastest memory type used for data and programs. Processor can execute the instruction directly from DRAM. Physically, they are available in a variety of packages depending on the application.

    - Nonvolatile memory

      This type of memory preserves the data even if power is removed. Read Only Memory (ROM), Erasable Programmable Read Only Memory (EPROM), Flash RAM and Nonvolatile RAM (NVRAM) are a few examples of nonvolatile memory.

  - Storage or Secondary Memory

    Typically, larger in size but slower access (Access time in micro to milliseconds). Also lower cost per mega bytes (250 GB for $100 in 2005)

    Some examples include Floppy Disk, Hard Disk (Magnetic Disk), CD and DVD (Optical Disk), Zip Drive, USB Jump Drive, Magnetic Tape.

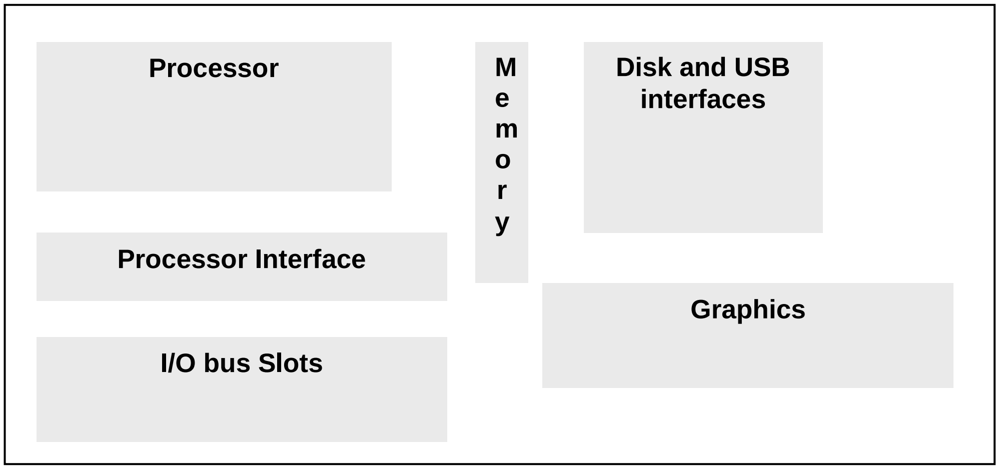
  - Selecting amongst memory types

    The selection of memory types are driven by the tradeoff between speed and price. Further, it should be noted that speed and price are inversely proportional. Most applications benefit from fastest memory, but budgets limit the speed of memory which developers can afford.

    The price and speed trade off leads to small size cache memory (fastest, typically static RAM) and medium sized main memory (typically DRAM). The largest memory or storage is typically the hard disk which is also the lowest cost per byte.

- Main or Mother-board

  A computer typically has a main board which houses the processor and other interface logic required for the operation of the computer system. The following diagram shows some of the common components found on a main board:


Processor
M
e
m
o
r
y
Disk and USB interfaces
Processor Interface
Graphics
I/O bus Slots

1.2. Processor Design Considerations

- Functionality
  Functionality is the foundation of design and as such is prominent in design consideration.

- Speed/Performance
  Speed and performance are increasingly more important considerations in computer and process design. The market is demanding higher performance computers as applications have increased in complexity. Examples include:

  - Increased demand for graphics in order to create more natural presentations
  - More types and larger data sets leading to larger and more complex database management
  - Multi-tasking and increased demand on Operating System
  - User’s expectation of instantaneous response.

- Usability
  Usability or ease-of-use continues to grow in importance as a broader range of users attempts to access more of the computer’s functionality.

- Maintainability and reliability
  As the systems become more complicated, the need for maintainability and expandability of existing software and hardware is more important than ever before. This has resulted in designers needing to use hardware modularization and its equivalent in software, Object Oriented Design.

- Memory Requirement
  As the technology advances and more memory becomes available at lower cost, minimization of memory requirement becomes less of a design issue.

  For example, a typical desktop computer in 1985 had 512 Kbytes of RAM, where the same type of computer in 2005 had 512 Mbytes of RAM. That represents a 1,000 fold increase in 20 years. The price for a 2 Gbyte in 2010 was roughly about the same as the price for a 512 Kbyte in 1985.

1.3. Computing systems Classification

- Computer Usage
  It is impossible to go through a day without interacting with computer systems in our modern society. Today, computers are integrated into many facets of living and working. In many cases, you may be benefiting from the power of a computer, but you may not be aware of its existence. The following list provides a few examples:

  - Cars
  - Home Appliances
  - Personal Computers
  - Internet
  - Cell Phones
  - Medical solutions such as Hearing-aid, pace maker and others
  - Traffic Light

- Classes of Computer Application

  - Workstations & Desktop Computers
    A computer used by one person with input and output devices. It may be used for personal, business, games, hobby, engineering, science or other activities. These systems typically have a dedicated display, keyboard and network connection.

  - Servers
    A computer used for running large programs for multiple users, often simultaneously. It is typically in a data center, accessible only through a network. A server might not have its own keyboard and display.

    Servers are available in a wide range of performance and functionality. The low-end servers and Supercomputers are the extreme ends of the spectrum:

    - Low-end Server
      This type of server may be a desktop computer running networkable version of windows, Linux or some other operating system.

    - Supercomputers
      This class of computers has the highest performance and is the most expensive. Supercomputers are typically used for specific and computationally intensive problems such as weather forecasting.

  - Embedded
    Computers embedded inside a device performing a set of predetermined functions. Embedded systems are the most pervasive type of computers and are expected continue to grow rapidly based on current trends. Embedded systems can be found in a broad range of products such as washing machines, cell phones and PDAs. A typically modern car has multiple embedded systems such as the fuel system controller and ABS breaking system.

    Based on a 2002 survey, the computer system usage for each type of computer is shown below:

    - 1122 million embedded or 89.5% of total
    - 131 million desktops or 10.4% of total
    - 1 million servers 0.1% of total

- Microprocessor Survey
  As of 2014, majority of processors are 64-bit (data is 64 bits wide). Prior to 2000, most processor designers were developing Complex Instruction Set Computers (CISC) which provide a large set of instructions. The most influential producers of CISC processor vendors were:

  - Motorola 68K - CISC

  - Intel's IA-32 (Intel's Pentium,…) - CISC

    Since 2000, the idea behind CISC has been successfully challenged by many processor designers and as of 2014, most major producer have migrated to Reduced Instruction Set Computers (RISC) which provide a selected few simple instructions, but instructions execute in a single clock cycle. This means that instruction execution is much faster than in CISC. The most influential producers of RISC processor vendors are:

    - IBM's Power PC – RISC
      * Also used in Apple PCs until 2006 when Apple moved to Intel's RISC processors.

    - Sun Microsystems' SPARC - RISC

    - Microchip's PIC processors and Microcontrollers - RISC
      "PICmicro will be used throughout this book as an example"

    - ARM Processors - RISC

    - MIPS - RISC

1.4. Historical Perspective and Trends

- Technology Trends

Year	Technology used in Computers	Relative Measure (Transistors/Device)
1951	Vacuum Tube	1
1965	Transistor	35
1975	Integrated circuits	900
1995	Very Large Scale Integrated Circuit	2,400,000
2005	Ultra Large Scale Integrated Circuit	6,200,000
2009	Dual Core Itanium 2 (596 mm^2 Die) “using 90 nm process”	1,700,000,000
2012	8-Core Itanium Poulson (544 mm^2 Die) “using 32 nm process”	3,100,000,000
2014	NVIDIA GK110 processor (551 mm^2 Die) “using 28 nm process”	7,100,000,000

- Moore’s law states that the number of transistors per square inch will double every 18-24 months. This observation has held true over the past 50 years (1965 – 2015).

- Complementary Metal Oxide Semiconductor (CMOS) is the dominant semiconductor technology for integrated circuits. The main reason is that it consumes power mainly during switching according the following formula:

  Power = Capacitive load x Voltage^2 x Frequency switched

  It is important to note that two of the main limiting factors for integrated circuits are power consumption and dissipation of resulting heat.

- Computer design trends:
  - Continued minimization of size and faster execution
  - Lower voltage (5v → 1.5V …)
  - Use of Biological solutions
  - Nano technology
  - Parallel processing
  - Large data buses 32→64→128→?

1.5. What's next…

The remainder of this book is focused on introducing key concepts in computer organization and system design. As much as possible, the general concepts will be introduced first, followed by an implementation example.

Microchip PIC 18F1220 Microcontroller will be used as the implementation example throughout the remainder of this book. Microchip PIC 18F1220 will be referred to as PICmicro.

PICmicro is a microcontroller as opposed to a microprocessor, which means, in addition to the functionality available in a typical microprocessor, PICmirco has additional functionality and circuits which are outlined below:

- Memory
  - 4K bytes of Program Flash Memory
    Flash memory used to store the program instruction set which can be reprogrammed up to 100,000 times. The programming is retained for over 40 years.

  - 256 bytes of Data Memory
    This memory is used for data. It will be referred to as the register file since all the available data memory is available to the user.

- 16 input/output ports

- Seven 10-bit Analog to Digital Converters

- One Pulse Width Modulator (PWM)
  PWM is used to control amount of power delivered by modulating (changing) duty cycles.

- One Enhanced Universal Asynchronous Receiver Transmitter (EUSART)
  Serial to parallel and parallel to serial capability with auto speed detection and wake-up capability.

- Three timers

- Priority-level interrupts

- Choice of internal or external oscillator

1.6. Integrated Development Environment (IDE)

Most processor vendors provide a full Integrated Development Environment (IDE) to support the developers using of their processors in development of new products. Typically, an IDE includes editor, compiler, assembler, linker, debugger, simulator and other useful applications/tools. Processor vendors such as Microchip are focused on providing effective IDEs to increase adoption rates resulting in the higher use of their processors.

Microchip's PICmicro family of processors has an extensive set of hardware and software development tools supporting the designers. PICmicro IDE is called MPLAB IDE and can be downloaded from www.EngrCS.com or directly from Microchip's website. MPLAB IDE is available for Widows, MAC and Linux. The MPLAB IDE offers the following core functionality:

- Code Management and Editor
- C complier is available but needs to be downloaded and installed.
- Assembler
- Linker
- Simulator
- Programmer Interface
- Debugger
- Extensive online help and tutorial

Below is a brief overview of these key components of MPLAB IDE:

- Code Management and Editor
  MPLAB IDE provides tools for managing your file as part of a project and editing your code in a context sensitive editor that provides syntax hints during programming.

- Compiler
  MPLAB's C compiler is a complete ANSI C compiler for PICmicro. This compiler is fully compatible and integrates seamlessly with MPLAB IDE. It also provides symbolic information that works with MPLAB IDE debugger and simulator.

  C code is saved in files with extension (.c) and include files are saved in files with extension (.h). Following is an example of PICmicro C code:

```
/*************************************************************
* File:        main.c
* Project:     A Simple Counter
* Author:      Great Designer
* Device:      PICmicro (PIC18F1220)
*************************************************************/

#include <p18f1220.h>
//TRISA, TRISB, PORTA, PORT are already defined in p18f1220.h

void main(void)
{
        unsigned char input;
        unsigned char lastinput = 0x00;
        unsigned char count = 0x00;

        ADCON1 = 0x7F;
        TRISA = 0x01;
        TRISB = 0x00;

        while(1)
        {
                input = PORTA;
                input = input & 0x01;
                if(input != lastinput)
                {
                        count++;
                        PORTB = count;
                }
                lastinput = input;
        }
}
```

❖ Assembler

PICmicro's assembler (MPASM) is an integral part of MPLAB IDE and MPASM, is a full-featured, universal macro assembler for all PICmicro MCUs. MPASM generates relocatable object files for the object linker (MPLINK), MAP files with detailed memory usage and symbol references, absolute LST files that contain source lines, machine code and COFF files for debugging.

Assembly code is saved in file with extension (.asm) which are part of a project such as the following code:

```
;--------------------------------------------------------------------------------------
; FILE: main.asm
; DESC: A Simple Counter
; DATE: 5-18-06
; AUTH: Great Designer
; DEVICE: PICmicro (PIC18F1220)
;--------------------------------------------------------------------------------------

       list    p=18F1220                            ; Set processor type
       radix   hex                                  ; Sets the default radix for data exp.

#define        PORTA          0xF80
#define        PORTB          0xF81
#define        TRISA          0xF92
#define        TRISB          0xF93
#define        ADCON1         0xFC1

               COUNT   equ    0x080
               LASTIN  equ    0x081
               INPUT   equ    0x082
               TEMP    equ    0x083

               org 0x000             ; Set the program origin (start) to 0x000

               ; Initialize all I/O ports
               CLRF    PORTA                ; Initialize PORTA
               CLRF    PORTB                ; Initialize PORTB
               MOVLW   0x7F                 ; Set all A\D Converter Pins as
               MOVWF   ADCON1               ; digital I/O pins
               MOVLW   0x0A                 ; Value used to initialize data direction
               MOVWF   TRISB                ; Set Port B <pins 0,2,4:7> as output
                                            ; Set Port B<pins 1,3> as input
               MOVLW   0xE2                 ; Value used to initialize data direction
               MOVWF   TRISA                ; Set Port A <Pin 7:5,1> as input
                                            ; Set Port A <Pin 0, 2:4> as output

               MOVLW   0x00                 ; W = 0
               MOVWF   COUNT                ; COUNT = WREG
               MOVWF   LASTIN               ; LASTIN = WREG

Loop:          MOVFF   PORTA, INPUT         ; INPUT = PORTA
               MOVF    INPUT, 0             ; W = PORTA
               XORWF   LASTIN, 0            ; W = W XOR LASTIN
               ANDLW   0x1                  ; W = W AND 0x1
               MOVFF   INPUT, LASTIN        ; LASTIN = PORTA
               MOVWF   TEMP                 ; TEMP = W
               BTFSC   TEMP, 0              ; If TEMP<0> = 0 Then Skip Next Command
               CALL    Increment
               GOTO    Loop

Increment:     MOVF    COUNT, 0             ; W = COUNT
               ADDLW   1                    ; W = W + 1
               MOVWF   COUNT                ; COUNT = W
               MOVWF   PORTB                ; PORTB = W
               RETURN

       end                                  ; Indicates the end of the program.
```

❖ Linker and Object Librarian
The linker is required to combine various object files generated by assembler and code libraries into an executable program. The MPLINK object linker combines relocatable objects created by the MPASM assembler and the MPLAB C compiler. It can also link relocatable objects from precompiled libraries using directives from a linker script.

The MPLIB object librarian manages the creation and modification of library files of precompiled code. When a routine from a library is called from a source file, only the modules which contain that routine will be linked in with the application. This allows large libraries to be used efficiently in many different applications.

- Simulator
  A great way to test the functionality of your code is to use a simulator before downloading the code into the processor hardware. The simulator enables the designer to test the functionality while viewing the processor's internal states and registers, as well as access to the debugging process. The only limitation is that the simulator will not test the timing requirements since the code is not running at the proper speed.

  The MPLAB SIM software simulator allows PICmicro code simulation in a PC hosted environment by simulating the PICmicro on an instruction level. For a given instruction, the data areas can be viewed or modified as stimuli are applied from either files or user key presses. The execution can be performed in different modes: Single-Step, Execute Until Break, or Trace. The MPLAB simulator supports symbolic debugging using MPLAB C Compilers and/or the MPASM assembler.

- Debugger
  The code can be debugged while simulating. The developer will have access to all the variables and memory locations as well as the ability to either single step through the code or run the code to a pre-determined breakpoint.

- Programmer Interface
  Once the code has been tested with simulation, the next step is to download the code into the PICmicro chip so that it can be installed in the circuit. From MPLAB, code can be downloaded to PIC Micro using one the PIC programmers.

1.7. Additional Resources

- Peterson. Computer Organization and Design, (2007) Elsevier Service.

- Microchip Staff. Microchip PIC 18F1220/1320 Data Sheet. (2004) Microchip Technology Incorporated.

- Microchip Staff. MPLAB IDE User’s Guide and Getting Started with MPLAB

1.8. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 2. ASSEMBLY INSTRUCTIONS AND PROCESSOR ARCHITECTURE

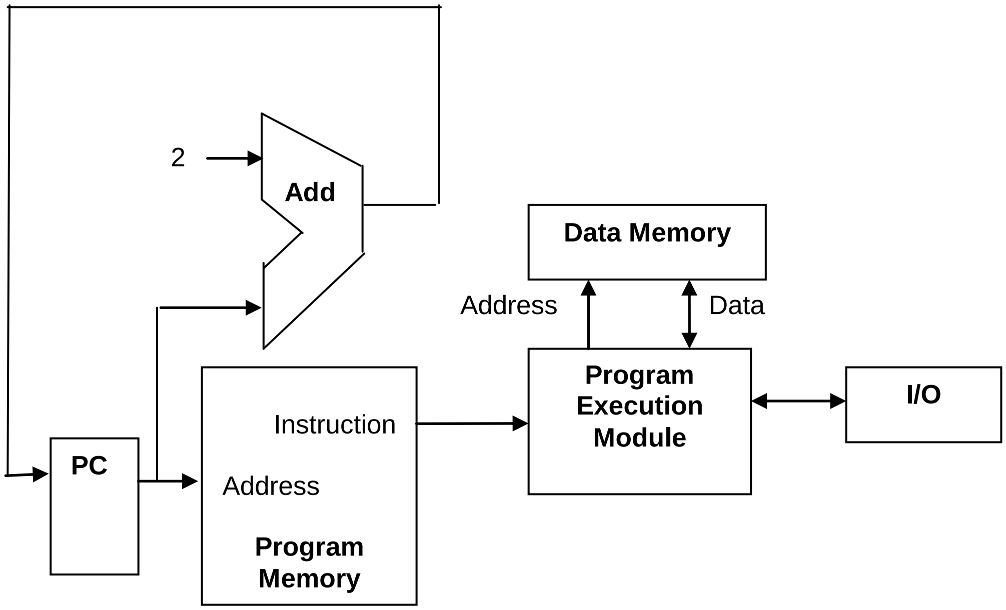
Key concepts and Overview

- Instruction Structure & Execution
- Byte-oriented Instructions
- Bit-oriented Instructions
- Literal-oriented Instructions
- Control Instructions
- Memory Layout & Definitions
- Additional Resources

2.1. Instruction Structure & Execution

At the most basic level, a processor's first step is to read an instruction (set of binary values). This step may also be referred to as fetching an instruction. In the next step, the processor will decode and execute the instruction. Finally, the processor writes any resulting data to memory. These steps are repeated until the processor is diverted.

Below is a high level view of this process where PC refers to Program Counter. PC's value is the address of the next instruction to be fetched and executed: Notice in this example, two is added to PC each time which means each instruction is 2 bytes long.



An instruction is made up of at least two fields and may use two, four or more bytes. First field is typically the opcode that identifies the desired operation. The second field is the operand for the operation. There may be additional fields as needed.

PICmicro instructions are single word (two bytes or 16 bits) long except for the three double-word instructions. All single-word instructions are executed in a single cycle. Single cycle execution is a common characteristic of Reduced Instruction Set Computer (RISC) where there are small numbers of instructions, but the instructions execute in a single clock cycle.

PICmicro has five types of instructions:

- Byte-Oriented operations
- Bit-Oriented operations
- Control Operations
- Literal Operations
- Memory-Block Operations

The Byte-Oriented, Bit-Oriented and Literal operation instructions move and manipulate data. We will be discussing these instructions in more detail later in this chapter. For these instructions, PC is incremented by 2 each time an instruction is executed so that PC will always be pointing to the next instruction.

Control operation instructions are used to change the next PC value to point to an address other than PC+2 if certain conditions are met. This set of instructions will be discussed in a later chapter. The control instructions are used to implement conditional expression such as “If-then-else” and loops such as “For loop”.

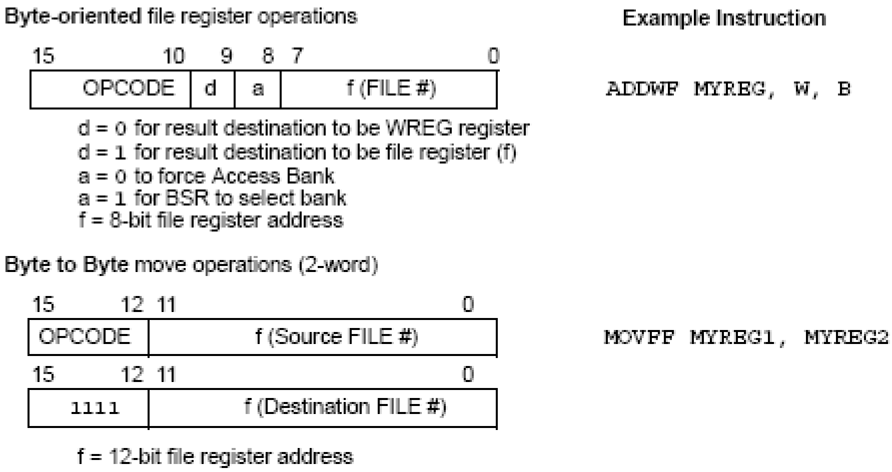
Refer to appendices for a complete listing of PICmicro instructions.

2.2. Byte-oriented Operations

Most byte-oriented instructions have three operands:

- The file register (specified by "f")
- The destination of the results (specified by "d")
- The accessed memory (specified by 'a")

The destination designator "d" specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the WREG register. If "d" is one, the result is placed in the file register specified in the instruction (default).



Byte-oriented file register operations

Example Instruction

15 … 10	9	8	7 … 0
OPCODE	d	a	f (FILE #)

`ADDWF MYREG, W, B`

d = 0 for result destination to be WREG register
d = 1 for result destination to be file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

Byte to Byte move operations (2-word)

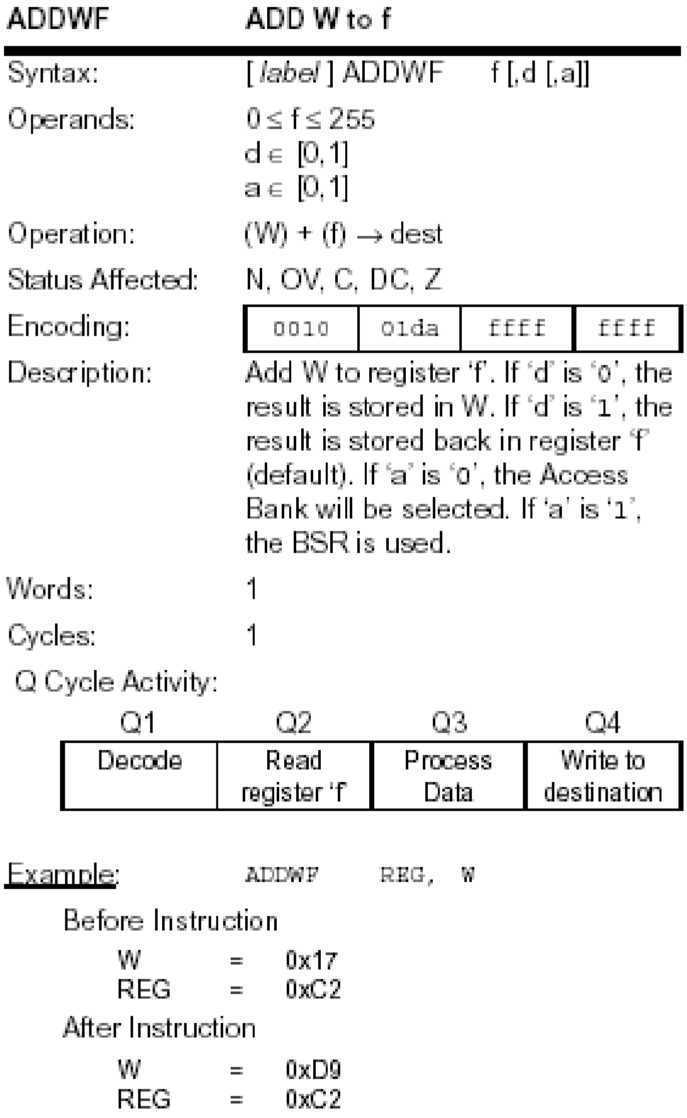
15 … 12	11 … 0
OPCODE	f (Source FILE #)
1111	f (Destination FILE #)

`MOVFF MYREG1, MYREG2`

f = 12-bit file register address

In this example ADDWF is the opcode (see appendix for PICmicro Instruction Set). Below are a more detailed description and examples of Byte-Oriented Instructions:

- ❖ Add WREG and f *"ADDWF f,d,a"*



ADDWF	**ADD W to f**
Syntax:	[*label*] ADDWF f [,d [,a]]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	$(W) + (f) \rightarrow$ dest
Status Affected:	N, OV, C, DC, Z
Encoding:	`0010` | `01da` | `ffff` | `ffff`
Description:	Add W to register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR is used.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ADDWF REG, W`

Before Instruction

W	=	0x17
REG	=	0xC2

After Instruction

W	=	0xD9
REG	=	0xC2

Notes:

- "ε" in the set of.
  d ε [0,1] means d can be 0 or 1.

- Arithmetic Logic Unit (ALU) Status Bit Definitions (Status Register – SFR)
  - "C" Carry - Set when the instruction results in a carry out of the most significant bit, clear when no carry
  - "DC" Digit Carry - Set for carry out of the 4th low order bit.
  - "N" Set for Negative result, clear for non-negative result
  - "OV" Set for overflow result, clear for non-overflow result
  - "Z" Set for zero result, clear for non-zero result

- Hexadecimal Designation
  Both "0x" prefix and "h" postfix indicate a hexadecimal number.

  For example both "0x1F" and "1Fh" are representations of 0x1F hexadecimal.

- BSR "Bank Select Register"
  By default, BSR will be set to "0" which means only the first 8 bits of the register file address are used and the upper 4-bits are set to 0 (0-FF).
  Later in this chapter, BSR will be discussed.

- "[ ]" any syntax item in the square bracket is optional. "[ ]" may be used as nested construct.

- "()" signifies that the content of the register (not the address) will be used in the operation. For example (f) refers to content of register f.

- Example – Given the following memory map, determine the value stored at memory location 33:

Address	**Data**
	.
	.
	.
0x20	12
0x21	23
0x22	34
0x23	65
	.
	.
	.

Solution:

Location 33: Value is 23. (33 in decimal is equal to 0x21 in hexadecimal)

- Example - ADDWF 0x12, 0, 0

Before Instruction

W = 0x10
REG (0x12) = 0x20

After Instruction

W = 0x30
REG (0x12) = 0x20

The value 0x10 is taken from working register WREG and added to the value 0x20 from file register 0x12. Because we have a zero for the [d] syntax item, the result is stored back in WREG.

- Example - Given W = 25 and F register (22) = 15.
  a) Determine what the values of W and register (22) will be after execution of the following assembly code statement:

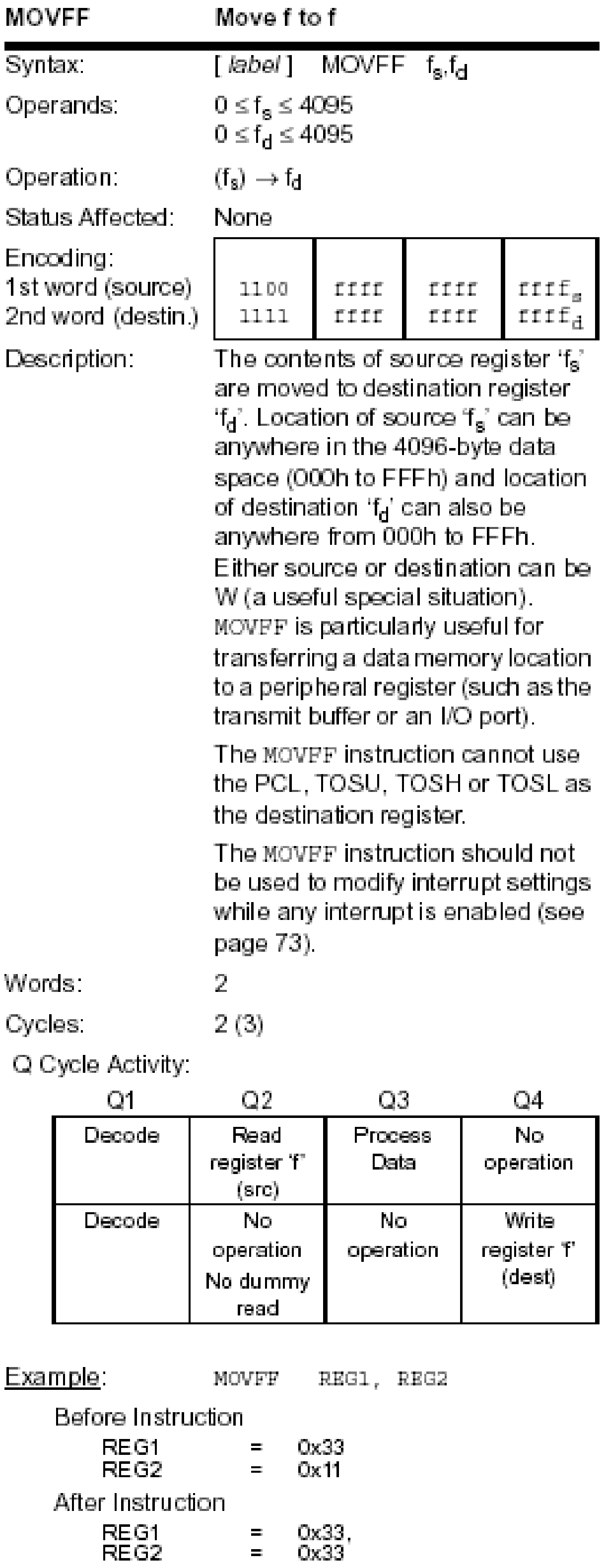
  ```
  ADDWF 22,1
  ```

  b) Determine the machine code equivalent for the above assembly code.
  Solutions:

  a) W = 25 and F register (22) = 40.

  b) Equivalent Machine code is “0010 0111 0001 0110”.

- Move f_s to f_d *"MOVFF f_s,f_d"*



MOVFF **Move f to f**

Syntax: [*label*] MOVFF f_s,f_d

Operands: $0 \le f_s \le 4095$
$0 \le f_d \le 4095$

Operation: $(f_s) \rightarrow f_d$

Status Affected: None

Encoding:

1st word (source)	1100	ffff	ffff	$ffff_s$
2nd word (destin.)	1111	ffff	ffff	$ffff_d$

Description: The contents of source register 'f_s' are moved to destination register 'f_d'. Location of source 'f_s' can be anywhere in the 4096-byte data space (000h to FFFh) and location of destination 'f_d' can also be anywhere from 000h to FFFh. Either source or destination can be W (a useful special situation). MOVFF is particularly useful for transferring a data memory location to a peripheral register (such as the transmit buffer or an I/O port).

The MOVFF instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.

The MOVFF instruction should not be used to modify interrupt settings while any interrupt is enabled (see page 73).

Words: 2

Cycles: 2 (3)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f' (src)	Process Data	No operation
Decode	No operation No dummy read	No operation	Write register 'f' (dest)

Example: MOVFF REG1, REG2

Before Instruction

REG1	=	0x33
REG2	=	0x11

After Instruction

REG1	=	0x33,
REG2	=	0x33

- Example - Given the following memory content:

Address	Data
	…
0x10	33
	…
0x15	25
	…

After execution of "MOVFF 0x10, 0x15":

a) What are the content in file registers 0x10 and 0x15?
b) What's the machine code equivalent?
c) Assuming the instructions are stored starting at program memory location 0x26, show the program memory content from 0x26 to 0x29.

Solutions:

a)

Address	Data
	…
0x10	33
	…
0x15	33
	…

b) "1100 0000 0001 0000"
"1111 0000 0001 0101"

c)

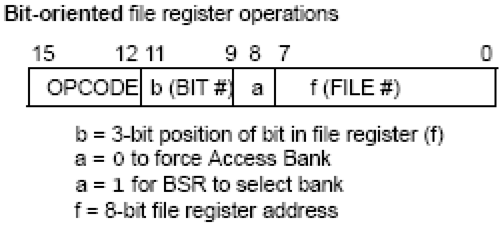
Address	Data
0x26	0x10
0x27	0xC0
0x28	0x15
0x29	0xF0

2.3. Bit-oriented Operations

A bit-oriented instruction has three operands:

- The file register (specified by “f”)
- The bit in the file register (specified by “b”)
- The accessed memory (specified by “a”)

The bit field designator ‘b’ selects the number (position) of the bit affected by the operation, while the file register designator “f”’ represents the number (address) of the file in which the bit is located.



Bit-oriented file register operations

15 12	11 9	8	7 0
OPCODE	b (BIT #)	a	f (FILE #)

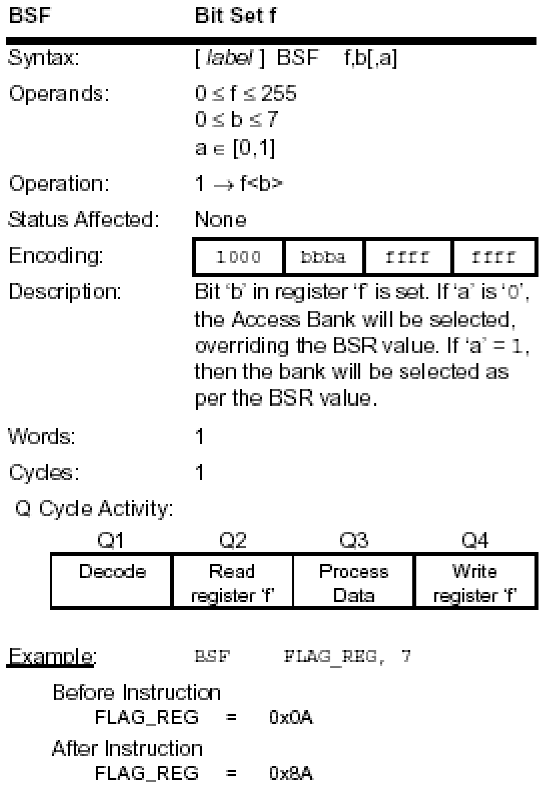
b = 3-bit position of bit in file register (f)
a = 0 to force Access Bank
a = 1 for BSR to select bank
f = 8-bit file register address

Example Instructions

```
BSF MYREG, bit
```

Below is a more detailed description of the two example instructions for the Bit-Oriented Instructions:

❖ Bit Set f “BSF f, b, a”



BSF **Bit Set f**

Syntax: [*label*] BSF f,b[,a]

Operands: 0 ≤ f ≤ 255
0 ≤ b ≤ 7
a ∈ [0,1]

Operation: 1 → f<b>

Status Affected: None

Encoding:

1000	bbba	ffff	ffff

Description: Bit ‘b’ in register ‘f’ is set. If ‘a’ is ‘0’, the Access Bank will be selected, overriding the BSR value. If ‘a’ = 1, then the bank will be selected as per the BSR value.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register ‘f’	Process Data	Write register ‘f’

Example: BSF FLAG_REG, 7

Before Instruction
FLAG_REG = 0x0A
After Instruction
FLAG_REG = 0x8A

- Example – value at location 29h is set to 0x20. What is the value at location 29h after the following code has been executed:

  ”BSF 0x29,2”

  Solution:
  value in location 29h will be 0x24

- Example – All memory locations have been cleared prior to executing the following machine code:

  “1000 1010 0111 0000”

  a) What ‘s the assembly code equivalent?
  b) Which memory location has been changed and what is the new content?

  Solution:

  a) BSF 0x70,5,0
  b) Location 0x70 changed to
  “0010 0000” or “0x20”

Example – Location 0x35 is set to 0x31 before execution of instruction “BSF 0x35,3”.

a) What is the Machine Code for the instruction in Hex?
b) What is the value in location 0x35 after the instruction execution?

Solution:

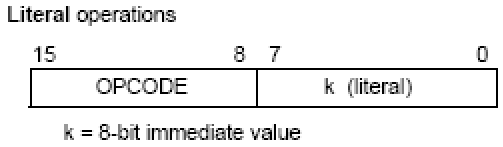
a) Machine Code: $(1000\ 0111\ 0011\ 0101)_2$
0x8735

b) Content of 0x35: $(0011\ 1001)_2$ → 0x39

2.4. Literal-oriented Operations

The **literal** instructions may use some of the following operands:

- A literal value to be loaded into a file register (specified by ‘k’)
- The desired Special Function Register (FSR) register to load with the literal value (specified by ‘f’)
- No operand required (specified by ‘—’)

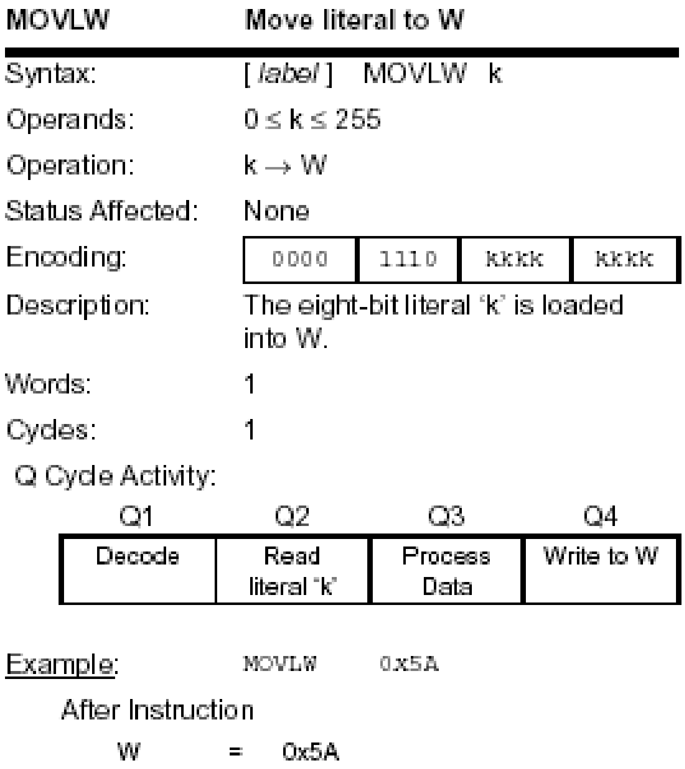


Example Instructions

```
MOVLW 0x7F
```

Below is a more detailed description of the example instructions for the Literal-Oriented Instructions:

- Move literal to WREG *“MOVLW 0x7F”*



MOVLW **Move literal to W**

Syntax: [*label*] MOVLW k

Operands: $0 \le k \le 255$

Operation: $k \rightarrow W$

Status Affected: None

Encoding:

0000	1110	kkkk	kkkk

Description: The eight-bit literal ‘k’ is loaded into W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal ‘k’	Process Data	Write to W

Example: MOVLW 0x5A

After Instruction

W = 0x5A

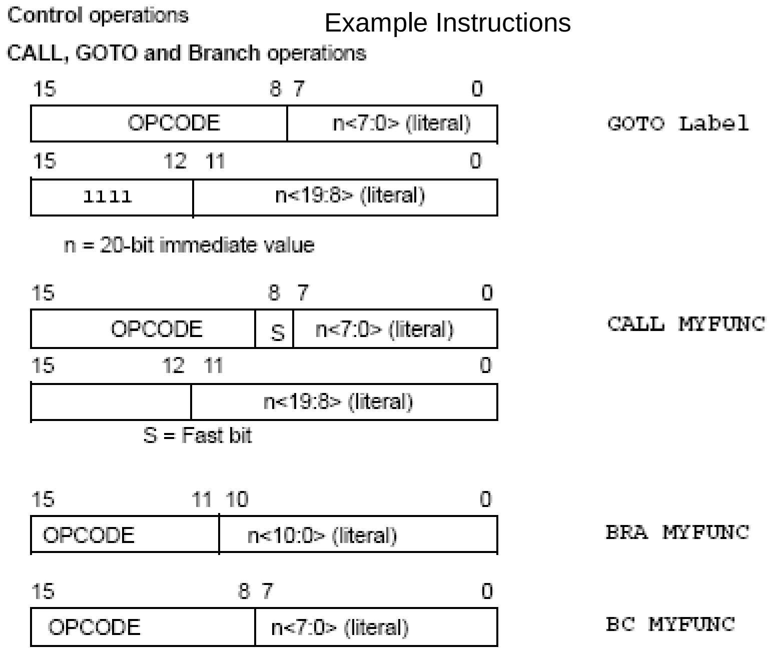
Notes:

-

2.5. Control Operations

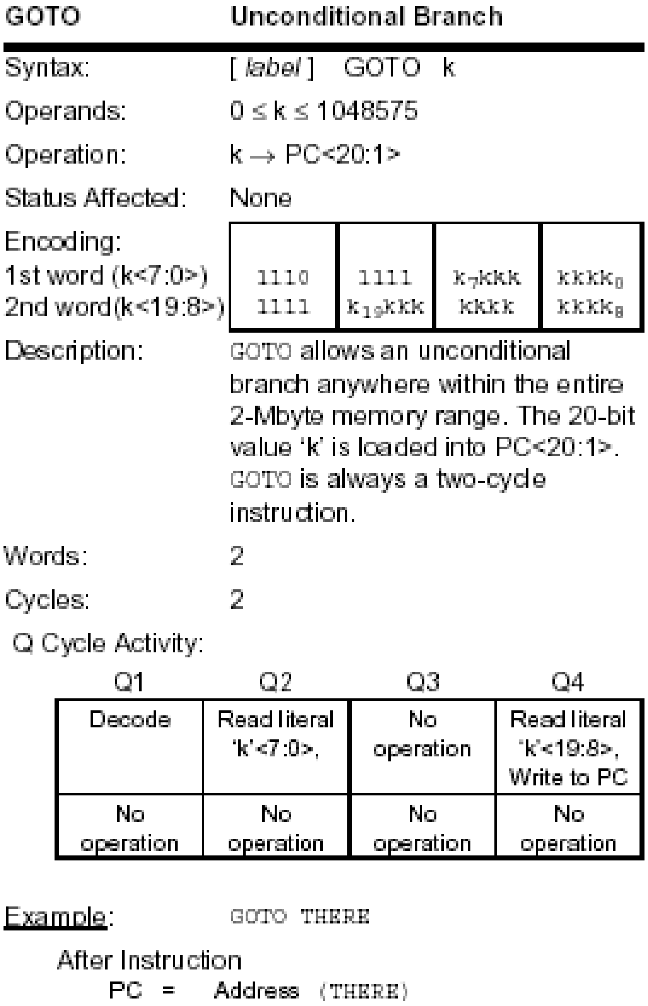
The control instructions may use some of the following operands:

- A program memory address (specified by ‘n’)
- The mode of the CALL or RETURN instructions (specified by ‘s’)
- No operand required (specified by ‘—’)



More detailed description of the Control instruction examples to follow:

❖ Go to address “k=$K_{19}K_{18}$ . . . K_1K_0” *“GOTO k”*



GOTO **Unconditional Branch**

Syntax: [*label*] GOTO k

Operands: $0 \le k \le 1048575$

Operation: k → PC<20:1>

Status Affected: None

Encoding:				
1st word (k<7:0>)	1110	1111	$k_7$kkk	kkkk_0
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: GOTO allows an unconditional branch anywhere within the entire 2-Mbyte memory range. The 20-bit value ‘k’ is loaded into PC<20:1>. GOTO is always a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal ‘k’<7:0>,	No operation	Read literal ‘k’<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: GOTO THERE

After Instruction
PC = Address (THERE)

- “k” is shifted to the left by 1 before being assigned to PC. This means that jump are always to a word boundary.

- Example – Determine the value of PC after the following machine code is executed:

  ”1110 1111 1000 0001”
  ”1111 0000 1000 0010”

  Solution:
  PC = 10502h

- Example – Determine the Machine Code equivalent for the Goto statement in the following Code Segment:

```
Address               Instruction (Assembly OpCode)
0x290                                     GOTO GreatProgram
….
0x932                 GreatProgram        ; Now what?
```

Solution:

- Example – Determine the next instruction location (PC) to be accessed after the execution of the following Machine Code:

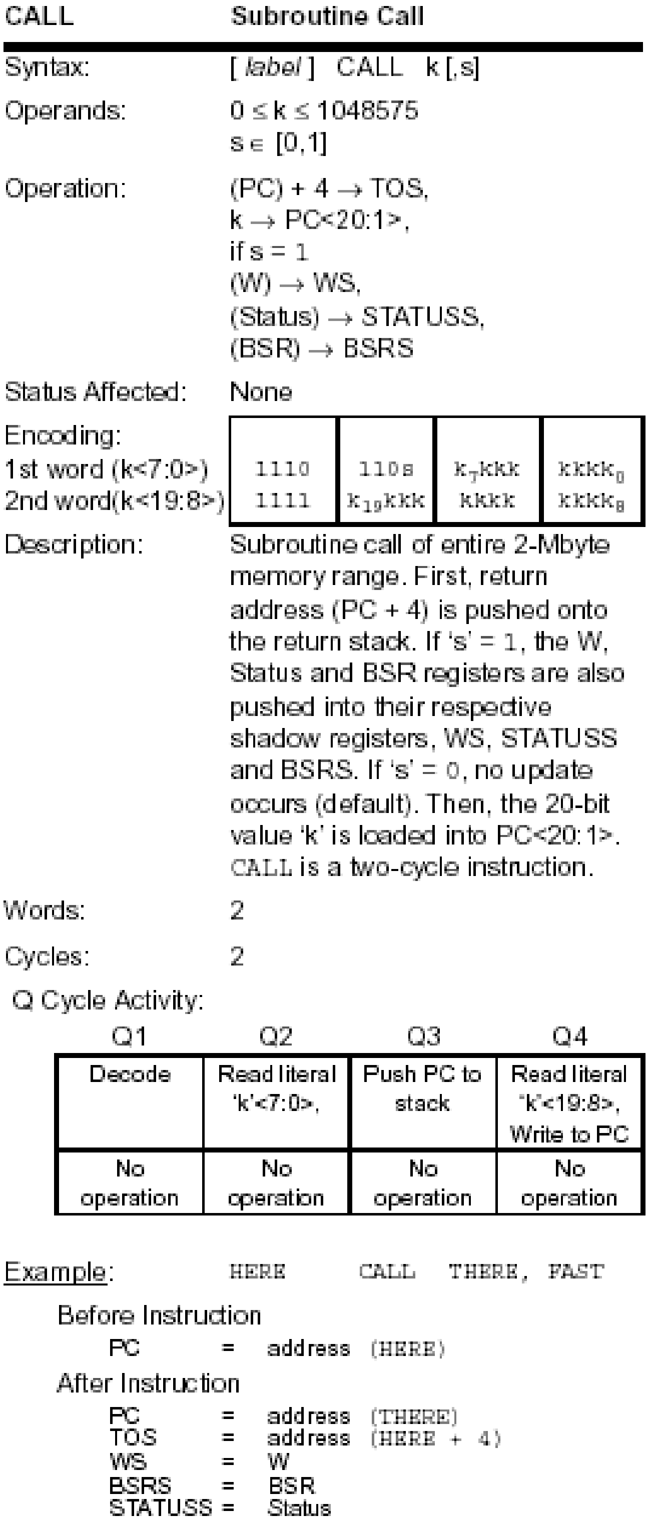
```
1110  1111  1010  1100
1111  0000  0000  0010
```

Solution:

❖ Call Subroutine at address “$k=K_{19}K_{18} \ldots K_1K_0$” *“CALL k,s”*

Notes:

▪



CALL **Subroutine Call**

Syntax: [*label*] CALL k [,s]

Operands: $0 \le k \le 1048575$
$s \in [0,1]$

Operation: (PC) + 4 → TOS,
k → PC<20:1>,
if s = 1
(W) → WS,
(Status) → STATUSS,
(BSR) → BSRS

Status Affected: None

Encoding:

1st word (k<7:0>)	1110	110s	$k_7$kkk	kkkk_0
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: Subroutine call of entire 2-Mbyte memory range. First, return address (PC + 4) is pushed onto the return stack. If ‘s’ = 1, the W, Status and BSR registers are also pushed into their respective shadow registers, WS, STATUSS and BSRS. If ‘s’ = 0, no update occurs (default). Then, the 20-bit value ‘k’ is loaded into PC<20:1>. CALL is a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal ‘k’<7:0>,	Push PC to stack	Read literal ‘k’<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: HERE CALL THERE, FAST

Before Instruction
PC = address (HERE)

After Instruction
PC = address (THERE)
TOS = address (HERE + 4)
WS = W
BSRS = BSR
STATUSS = Status

- Example – Determine the value of PC after the execution of the following Machine Code:

```
1110  1111  1010  1100
1111  0100  0001  0101
```

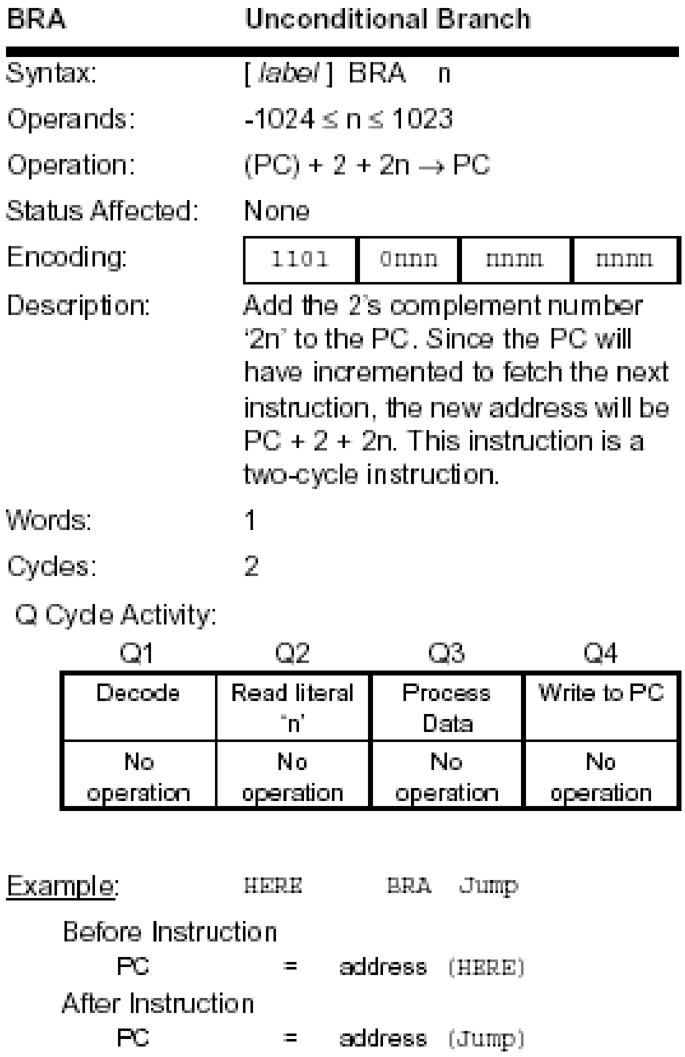
Solution:

- Example – Determine the Machine Code equivalent for the following CALL Instruction:

Address	Instruction (Assembly OpCode)	
0x24	FortyTwo:	MOVFF answer, life
….		
0x290		CALL FortyTwo

Solution:

❖ Branch Unconditionally *“BRA n”*



BRA **Unconditional Branch**

Syntax: [*label*] BRA n

Operands: -1024 ≤ n ≤ 1023

Operation: (PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1101	0nnn	nnnn	nnnn

Description: Add the 2’s complement number ‘2n’ to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal ‘n’	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: HERE BRA Jump

Before Instruction
PC = address (HERE)
After Instruction
PC = address (Jump)

Notes:

- Address Calculation

  New PC = BRA’s PC + 2 + 2*n
  “n is in 2’s Complement format”

- Example – Determine the address of the next instruction to be executed after the following BRA instruction:

Address	Memory Content
0x236	1101 0111 1000 1000

It is important to note that offset is provided in “2n” and 2’s complements format. Therefore:

```
   1111 0001 0000   (Offset=2*n) +
   0010 0011 0110   (PC)
               10   (2)
.-------------------------------
 ~~1~~  0001 0100 1000
   or
    0x148 new PC
```

As you see the overflow is ignored and the address of the next instruction after BRA will be location 0x148.

➢ Example – Determine the Machine Code equivalent for the following BRA instruction:

Address	Instruction (Assembly OpCode)	
0x110		BRA NextEvent
….		
0x230	NextEvent:	NOP

Solution:

- Example – Determine the PC after the execution of:

Address	Code
0x3210	BRA 215

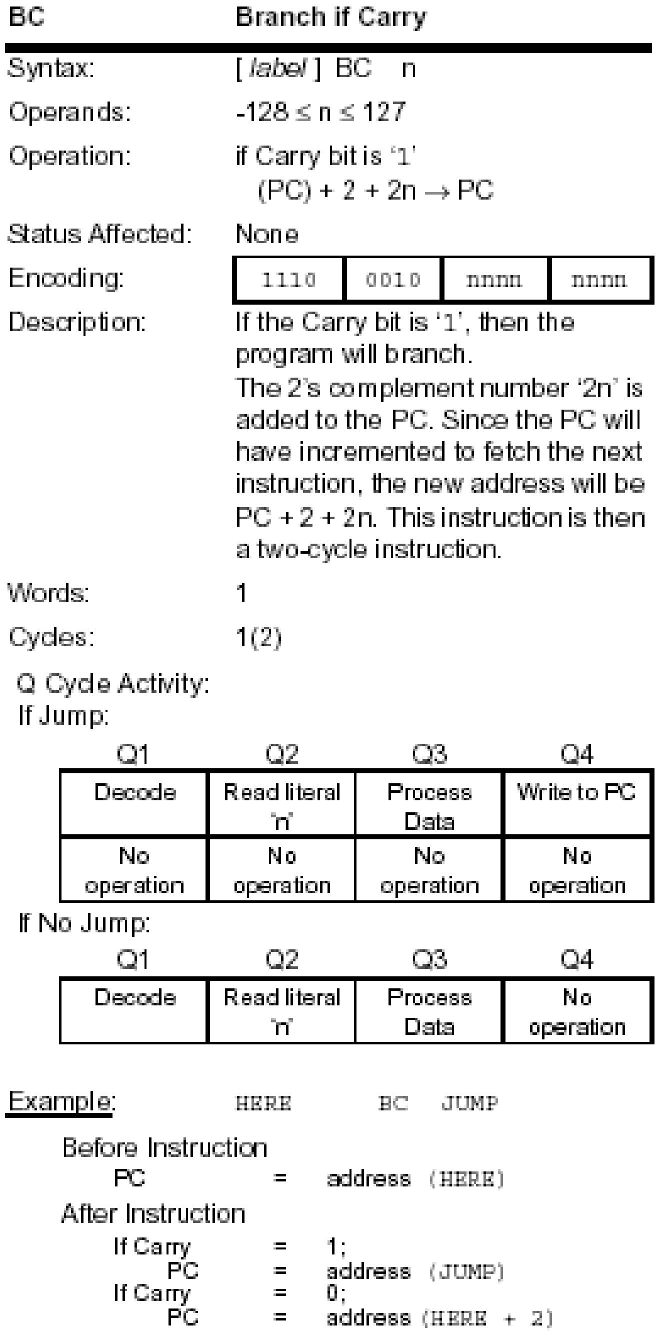
Solution:

- Example – Determine the Machine Code equivalent for the following BRA instruction:
  Hint: Negative n value.

Address	Instruction (Assembly OpCode)	
0x2110	Step:	MOVWF Dove, 0
….		
0x2140		BRA Step

Solution:

❖ Branch if Carry *"BC n"*



BC **Branch if Carry**

Syntax: [*label*] BC n

Operands: -128 ≤ n ≤ 127

Operation: if Carry bit is '1'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0010	nnnn	nnnn

Description: If the Carry bit is '1', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: HERE BC JUMP

Before Instruction
PC = address (HERE)

After Instruction
If Carry = 1;
PC = address (JUMP)
If Carry = 0;
PC = address (HERE + 2)

- Example – Assuming Carry bit is set, determine the PC after execution of the following machine code:

Address	**Data .**
0x252	1110 0010 1111 1100

Solution:
OpCode → BRA
n= $(1111\ 1100)_{RC}$ = -2

New PC = BR PC + 2 + 2n = 0x250

- Example – Implement the following C code using BC instruction:

```
high=0;
if (temp == 255) {
    high = 1;
}
```

Solution:

```
    high    equ 0x80
    temp    equ 0x81
    clrf    high
    …
```

- Example – Write the machine code for line labeled "loop2" in the following code segment:

```
          Wreg = 245
          org 0x3442
Loop:     Nop
          BNZ loop2
          ADDLW 25
Loop2:    BC Loop
```

Solution:
"1110 0010 1111 1100"

- Example – Determine the Machine Code equivalent for the following code segment:

```
Address     Instruction (Assembly OpCode)
0x220       Loop:       ADDLW     52
0x222                   MOVFF     New, Old
0x226                   BC        Loop
….
0x340       Step:       MOVWF     Dove, 0
```

Solution:

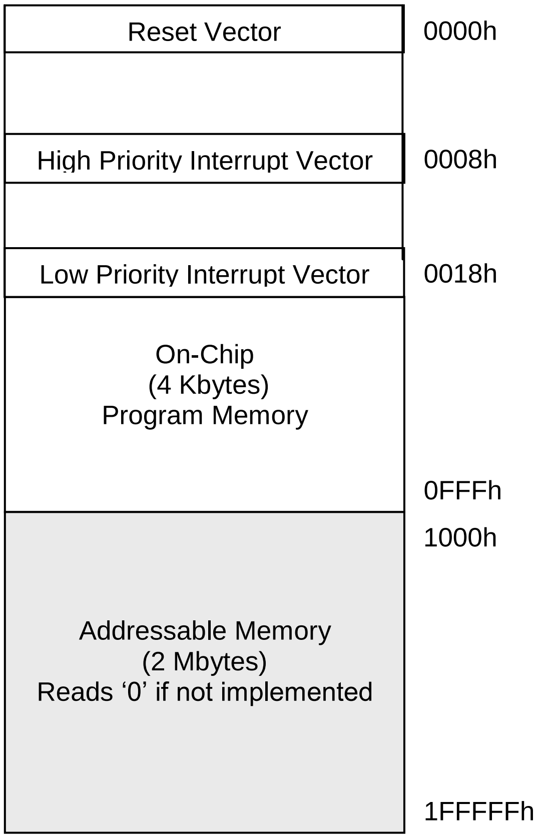
2.6. Memory Layout & Definitions

In general, computer memory is organized into two sections: data memory and program memory. The size and organization of the memory depend on the type of system and its function. For the PICmicro example used here, the memory sizes are in Kilobytes (10^3) and Megabytes(10^6). Depending on your application, memory ranges may be in Gigabytes (10^9) and Terrabytes (10^{12}).

Typical computer systems have all three types of memory (Program memory, Data memory, Nonvolatile memory). Although Program and Data memories may be the same type of hardware, Program and Data are stored in different sections of memory. This organization is required to ensure that data does not overwrite programs. Additionally, if the data and program paths are kept separate, the processor can read and write instructions and data simultaneously in order to improve performance.

As discussed earlier, PICmicro is a microcontroller which means that it will have some amount of each memory type on-chip, in addition to other functionality. Specifically, PICmicro has the following types of memory on the chip:

- Program Memory – 4 Kbytes on-chip with ability to access off-chip memory if available. The on-chip program memory is EEPROM which is non-volatile (data is not lost when power is removed). The following diagram outlines the total memory space and location of on-chip program memory from 0018h-0FFFh



Program Memory Space (2 Mbytes)

- Data RAM

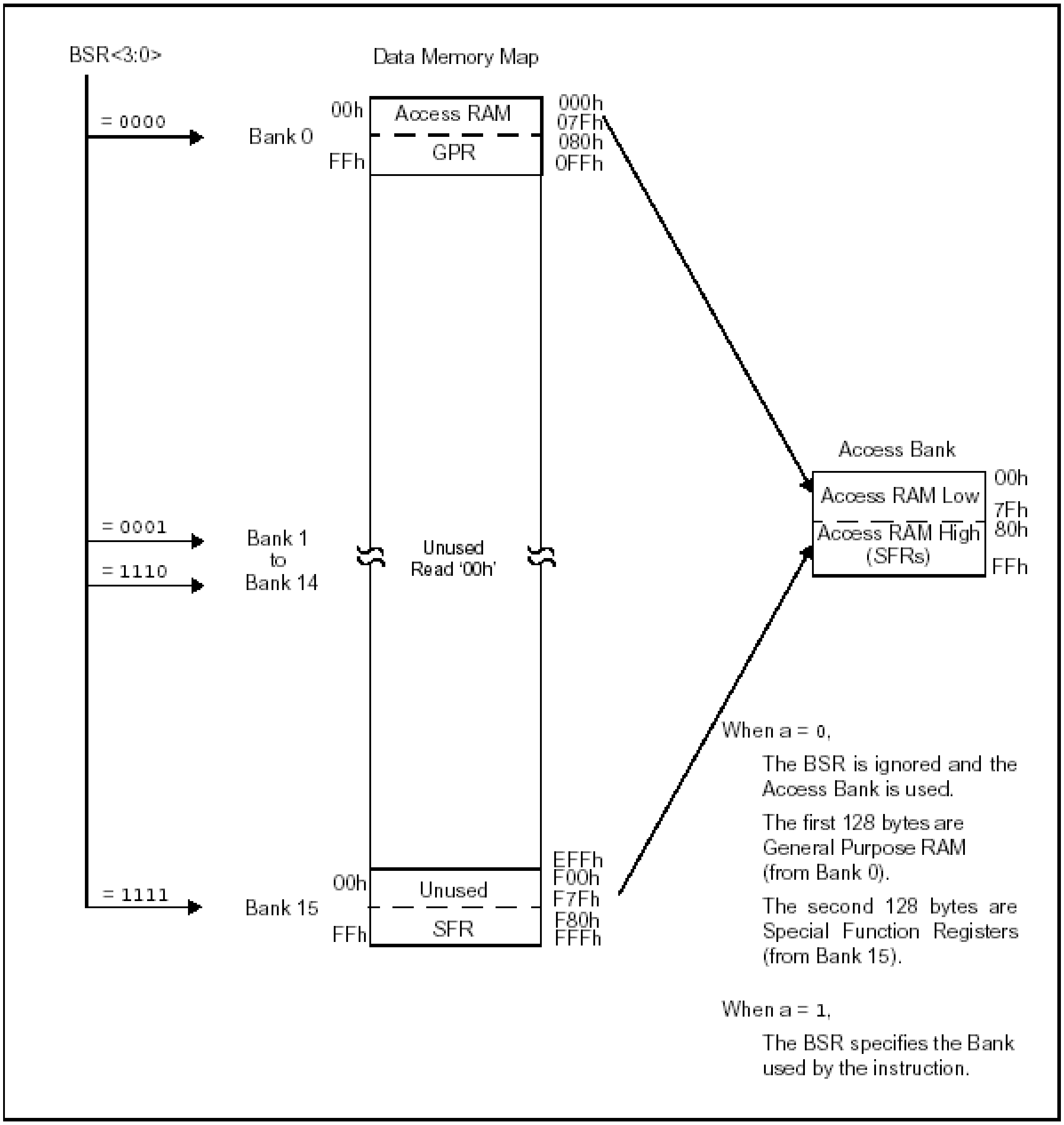
  PICmirco's Program and Data memories use separate buses. This allows for concurrent access of program and data resulting in improved performance.

  The data memory is implemented as static RAM. Each register in the data memory has a 12-bit address, allowing up to 4096 bytes of data memory. The data memory map is divided into as many as 16 banks that contain 256 bytes each. The lower 4 bits of the Bank Select Register (BSR<3:0>) select which bank will be accessed. The upper 4 bits for the BSR are not implemented.

  The data memory contains Special Function Registers (SFR) and General Purpose Registers (GPR). The SFRs are used for configuraton and status reporting of the controller and peripheral functions, while GPRs are used for data storage and temporary memory for programs. The SFRs are located in Bank 15,from F80h to FFFh. . Any remaining space in the Bank may be implemented as GPRs. GPRs start at the first location of Bank 0 (000h) and extend upwards through the rest of the banks. Any read of an unimplemented location will return '0's.

  The entire data memory may be accessed directly or indirectly. Direct addressing may require the use of the BSR register. Indirect addressing requires the use of a File Select Register (FSRn) and a corresponding Indirect File Operand (INDFn). Each FSR holds a 12-bit address value that can be used to access any location in the Data Memory map without banking.

  The instruction set and architecture allow operations across all banks. This may be accomplished by indirect addressing or by the use of the MOVFF instruction. The MOVFF instruction is a two-word/two-cycle instruction that moves a value from one register to another. To ensure that commonly used registers (SFRs and select GPRs) can be accessed in a single cycle, regardless of the current BSR values, an Access Bank is implemented. A segment of Bank 0 and a segment of Bank 15 comprise the Access RAM.



PICmicro has banked memory in the General Purpose Registers (GPRs) area. GPRs are not initialized by a Power-on Reset and are unchanged on all other Resets. Data RAM is available for use as GPRs by all instructions. The second half of Bank 15 (F80h to FFFh) contains Special Function Registers (SFRs). The SFRs are used by the Central Processing Unit (CPU) and peripheral modules for controlling the desired operation of the device.

The following registers are most commonly used:

- Addresses 0xF80 through 0xFFF. These are the Special Function Registers (SFR) such as PORTA, PORTB, TRISA, TRISB, etc.
- Addresses 0x000 through 0x0FF. These are the Access Ram and General Purpose Registers (GPR) which can be used by programs to store data.

In many instructions, the value of flag “a” may be set to define the use of BSR as shown below:

- **a=0**
  GPR address → 0x000 – 0x07F ; *GPRs are available at 0x00-0x7F*
  SFR address → 0xF80 – 0xFFF ; *SFR range is always from F80 to FFH*

- **a=1 “Default” with BSR =0**
  GPR address → 0x080 – 0x0FF ; *GPRs are available at two ranges 0x80-0x0FF or 0x00-0x7F*
  SFR address → 0xF80 – 0xFFF ; *SFR range is always from F80 to FFH*

- **a=1 “Default” with BSR =3**
  GPR address → 0x380 – 0x3FF ; *GPRs are available at two ranges 0x380-0x3FF or 0x00-0x7F*
  SFR address → 0xF80 – 0xFFF ; *SFR range is always from F80 to FFH*

2.7. Additional Resources

- Microchip Staff. Microchip PIC 18F1220/1320 Data Sheet. (2004) Microchip Technology Incorporated.

- Huang,. PIC Microcontroller: An Introduction to Software & Hardware Interfacing, (2004) Thomson.

- Reese. Microprocessor: From Assembly Language to C using the PIC18Fxxx2. (2003) Course Technology.

- Peterson. Computer Organization and Design, (2007) Elsevier Service.

2.8. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

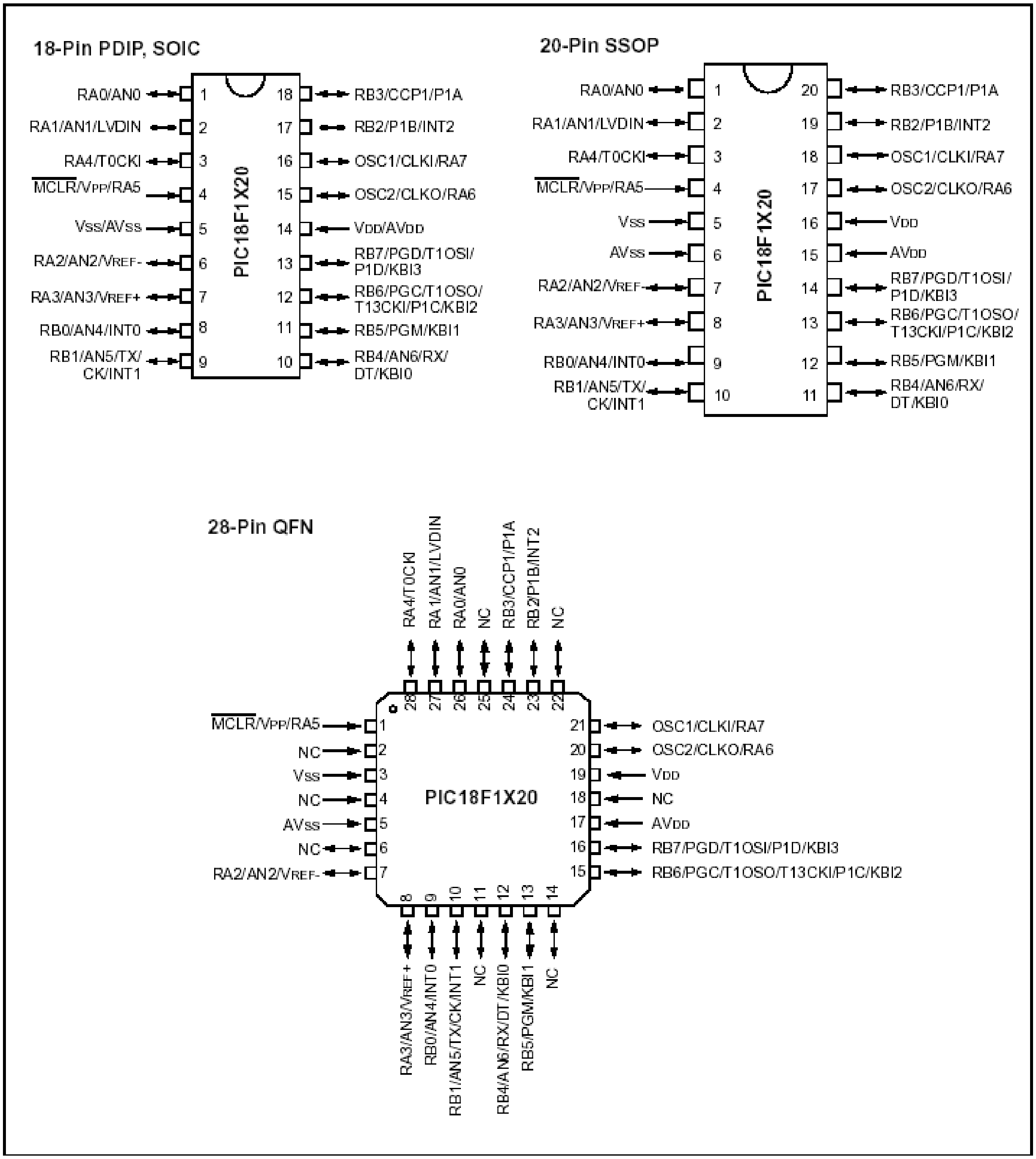
CHAPTER 3. INPUT/OUTPUT ORGANIZATIONS

Key concepts and Overview

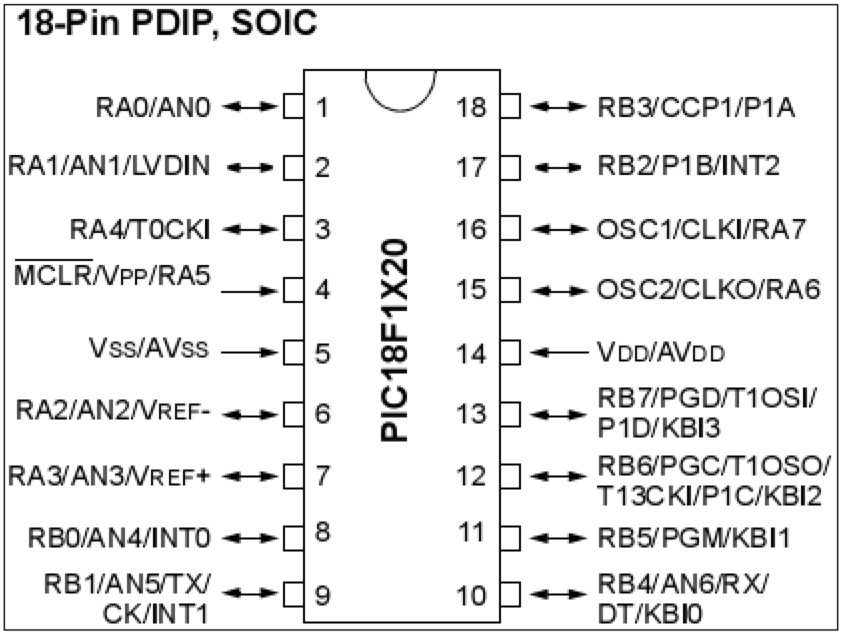
- Pinout and Packaging
- Accessing I/O Devices
- Additional Resources

3.1. Pinout and Packaging

PICmicro is available in three types of packaging. It is important to note that each package has a different pin layout. Plastic Dual In-Line Package (PDIP) is the most common type of packaging for prototyping where automated systems are not available. On the other hand, Quad Flat No-Lead (QFN) package and Shrink Small Outline Package (SSOP) are most commonly used for high volume production where automation can handle small sizes and cost is important. SSOP is able to handle a wider range of temperatures compared to QFN.



- Prototype Packaging
  As discussed earlier PDIP is the most common packaging used for prototyping. PICmicro (PIC18F1220) pin out is shown below:

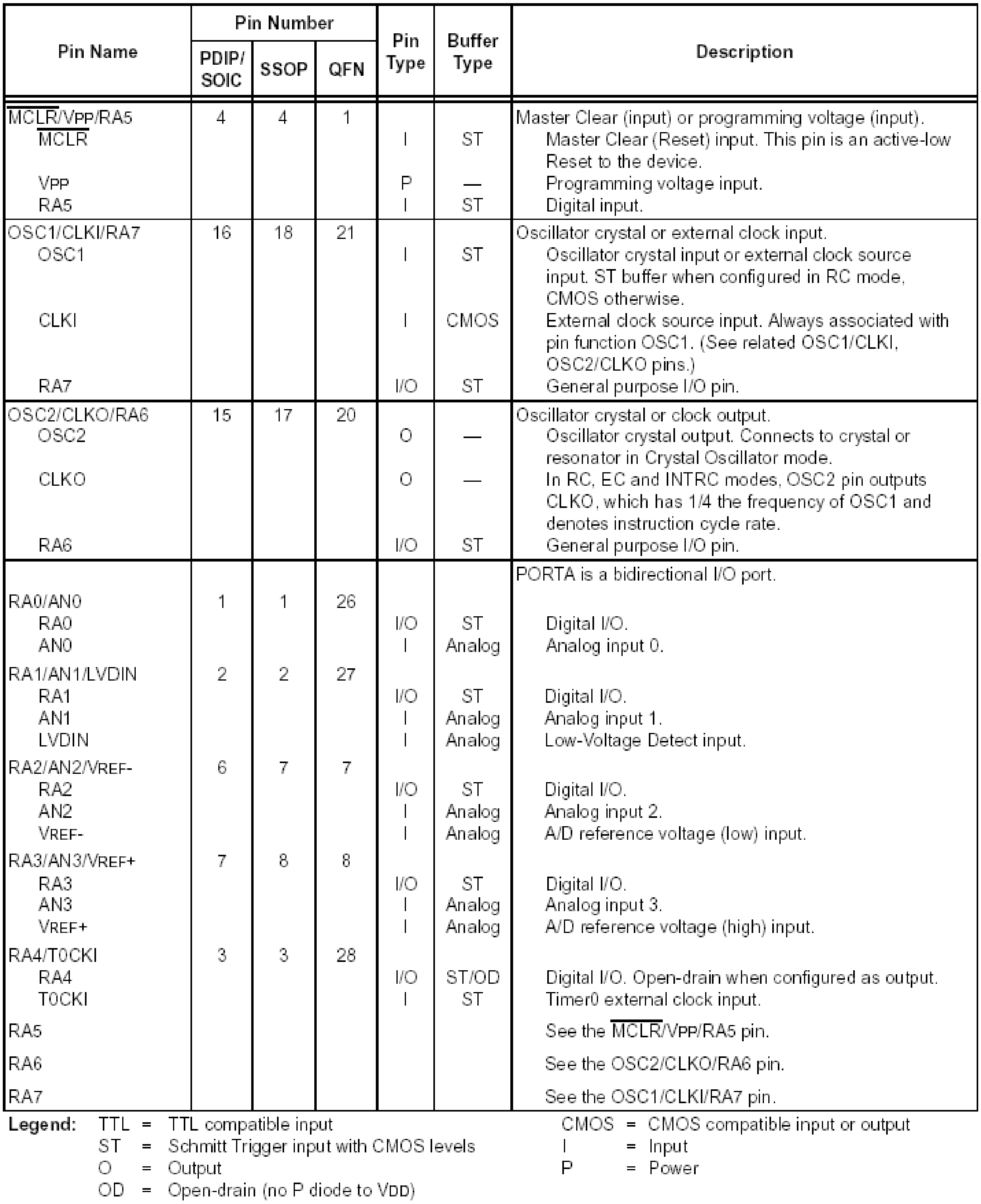


Each pin can be configured to perform a variety of functions, for example Pin 8 may be an I/O port (RB0), I/O port (AN4), or external Interrupt 0 (INT0). This type of multi-use is common in microcontroller with high level of functionality, but it is less common in general purpose microprocessors.

The two pins whose definition is constant are pins 5 and 14 which are ground and power.

Pin 5	Ground (0 V)
Pin 14	Power (2 to 5.5 V)

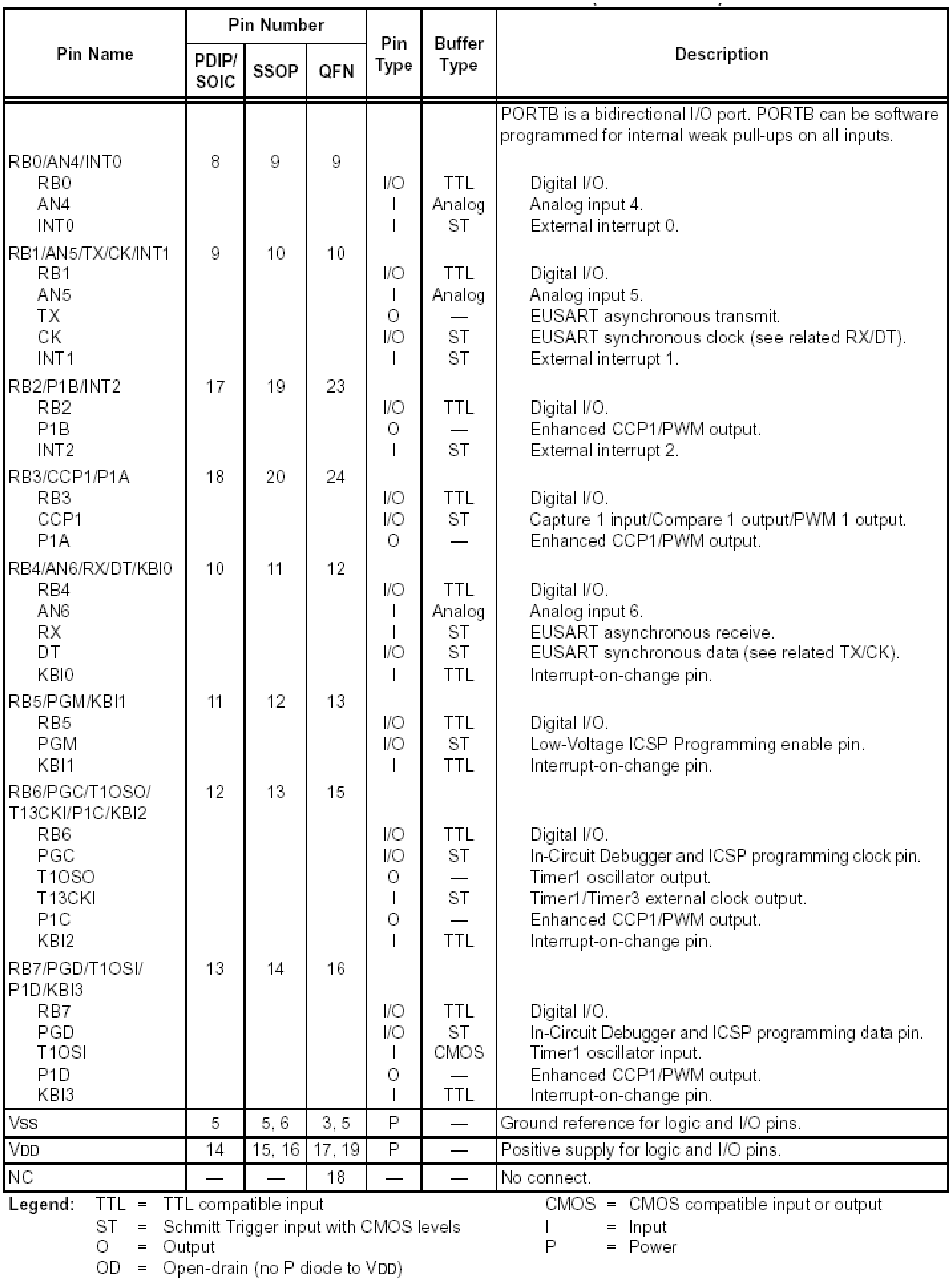
- Full PIN Descriptions
  PICmicro's pin definition is outlined in the following two tables:



Pin Name	Pin Number			Pin Type	Buffer Type	Description
	PDIP/ SOIC	SSOP	QFN			
$\overline{\text{MCLR}}$/VPP/RA5	4	4	1			Master Clear (input) or programming voltage (input).
$\overline{\text{MCLR}}$				I	ST	Master Clear (Reset) input. This pin is an active-low Reset to the device.
VPP				P	—	Programming voltage input.
RA5				I	ST	Digital input.
OSC1/CLKI/RA7	16	18	21			Oscillator crystal or external clock input.
OSC1				I	ST	Oscillator crystal input or external clock source input. ST buffer when configured in RC mode, CMOS otherwise.
CLKI				I	CMOS	External clock source input. Always associated with pin function OSC1. (See related OSC1/CLKI, OSC2/CLKO pins.)
RA7				I/O	ST	General purpose I/O pin.
OSC2/CLKO/RA6	15	17	20			Oscillator crystal or clock output.
OSC2				O	—	Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode.
CLKO				O	—	In RC, EC and INTRC modes, OSC2 pin outputs CLKO, which has 1/4 the frequency of OSC1 and denotes instruction cycle rate.
RA6				I/O	ST	General purpose I/O pin.
						PORTA is a bidirectional I/O port.
RA0/AN0	1	1	26			
RA0				I/O	ST	Digital I/O.
AN0				I	Analog	Analog input 0.
RA1/AN1/LVDIN	2	2	27			
RA1				I/O	ST	Digital I/O.
AN1				I	Analog	Analog input 1.
LVDIN				I	Analog	Low-Voltage Detect input.
RA2/AN2/VREF-	6	7	7			
RA2				I/O	ST	Digital I/O.
AN2				I	Analog	Analog input 2.
VREF-				I	Analog	A/D reference voltage (low) input.
RA3/AN3/VREF+	7	8	8			
RA3				I/O	ST	Digital I/O.
AN3				I	Analog	Analog input 3.
VREF+				I	Analog	A/D reference voltage (high) input.
RA4/T0CKI	3	3	28			
RA4				I/O	ST/OD	Digital I/O. Open-drain when configured as output.
T0CKI				I	ST	Timer0 external clock input.
RA5						See the $\overline{\text{MCLR}}$/VPP/RA5 pin.
RA6						See the OSC2/CLKO/RA6 pin.
RA7						See the OSC1/CLKI/RA7 pin.

Legend:

TTL	= TTL compatible input		CMOS	=	CMOS compatible input or output
ST	= Schmitt Trigger input with CMOS levels		I	=	Input
O	= Output		P	=	Power
OD	= Open-drain (no P diode to VDD)				



Pin Name	Pin Number			Pin Type	Buffer Type	Description
	PDIP/ SOIC	SSOP	QFN			
						PORTB is a bidirectional I/O port. PORTB can be software programmed for internal weak pull-ups on all inputs.
RB0/AN4/INT0	8	9	9			
RB0				I/O	TTL	Digital I/O.
AN4				I	Analog	Analog input 4.
INT0				I	ST	External interrupt 0.
RB1/AN5/TX/CK/INT1	9	10	10			
RB1				I/O	TTL	Digital I/O.
AN5				I	Analog	Analog input 5.
TX				O	—	EUSART asynchronous transmit.
CK				I/O	ST	EUSART synchronous clock (see related RX/DT).
INT1				I	ST	External interrupt 1.
RB2/P1B/INT2	17	19	23			
RB2				I/O	TTL	Digital I/O.
P1B				O	—	Enhanced CCP1/PWM output.
INT2				I	ST	External interrupt 2.
RB3/CCP1/P1A	18	20	24			
RB3				I/O	TTL	Digital I/O.
CCP1				I/O	ST	Capture 1 input/Compare 1 output/PWM 1 output.
P1A				O	—	Enhanced CCP1/PWM output.
RB4/AN6/RX/DT/KBI0	10	11	12			
RB4				I/O	TTL	Digital I/O.
AN6				I	Analog	Analog input 6.
RX				I	ST	EUSART asynchronous receive.
DT				I/O	ST	EUSART synchronous data (see related TX/CK).
KBI0				I	TTL	Interrupt-on-change pin.
RB5/PGM/KBI1	11	12	13			
RB5				I/O	TTL	Digital I/O.
PGM				I/O	ST	Low-Voltage ICSP Programming enable pin.
KBI1				I	TTL	Interrupt-on-change pin.
RB6/PGC/T1OSO/ T13CKI/P1C/KBI2	12	13	15			
RB6				I/O	TTL	Digital I/O.
PGC				I/O	ST	In-Circuit Debugger and ICSP programming clock pin.
T1OSO				O	—	Timer1 oscillator output.
T13CKI				I	ST	Timer1/Timer3 external clock output.
P1C				O	—	Enhanced CCP1/PWM output.
KBI2				I	TTL	Interrupt-on-change pin.
RB7/PGD/T1OSI/ P1D/KBI3	13	14	16			
RB7				I/O	TTL	Digital I/O.
PGD				I/O	ST	In-Circuit Debugger and ICSP programming data pin.
T1OSI				I	CMOS	Timer1 oscillator input.
P1D				O	—	Enhanced CCP1/PWM output.
KBI3				I	TTL	Interrupt-on-change pin.
VSS	5	5, 6	3, 5	P	—	Ground reference for logic and I/O pins.
VDD	14	15, 16	17, 19	P	—	Positive supply for logic and I/O pins.
NC	—	—	18	—	—	No connect.

Legend: TTL = TTL compatible input CMOS = CMOS compatible input or output
ST = Schmitt Trigger input with CMOS levels I = Input
O = Output P = Power
OD = Open-drain (no P diode to VDD)

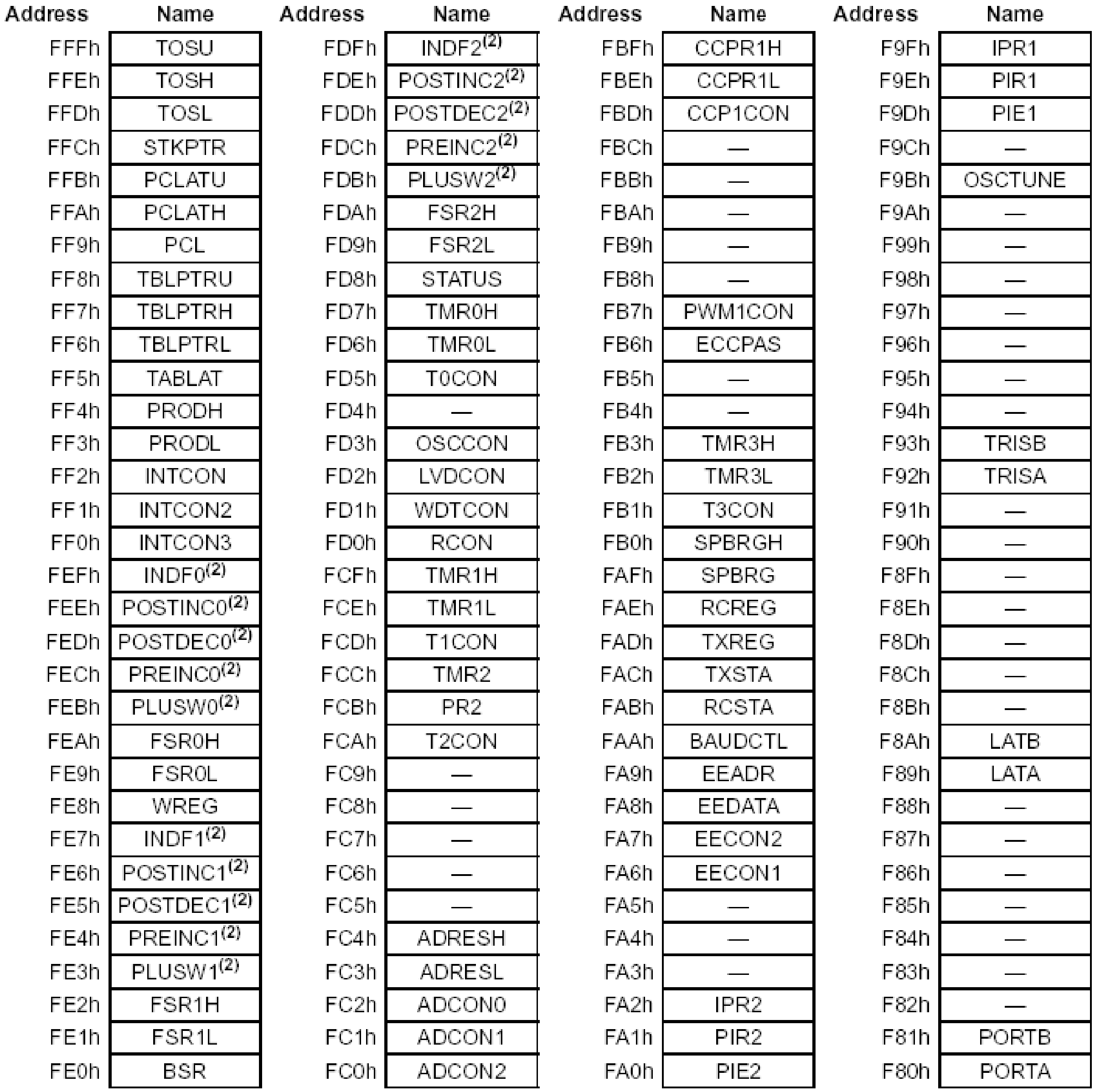
3.2. Accessing I/O Devices

PICmicro programs are able to read from and write to external devices by using the Special Function registers (SFRs). SFRs are used to configure the external pins as input/output, and the internal peripheral modules such as the Analog to Digital converter and the Pulse Width Modulator.

SFRs can be classified as relating to either the core functions or the peripheral functions. The registers related to the “core” are described in this section, and the others will be covered in the latter part of the text. Note that the unused SFR locations will be signified by “__” and are read as ‘0’s.

Hint: all the names and values have been predefined in header file “p18f1220.inc” for assembly programming and in “p18f1220.h” for C programming.

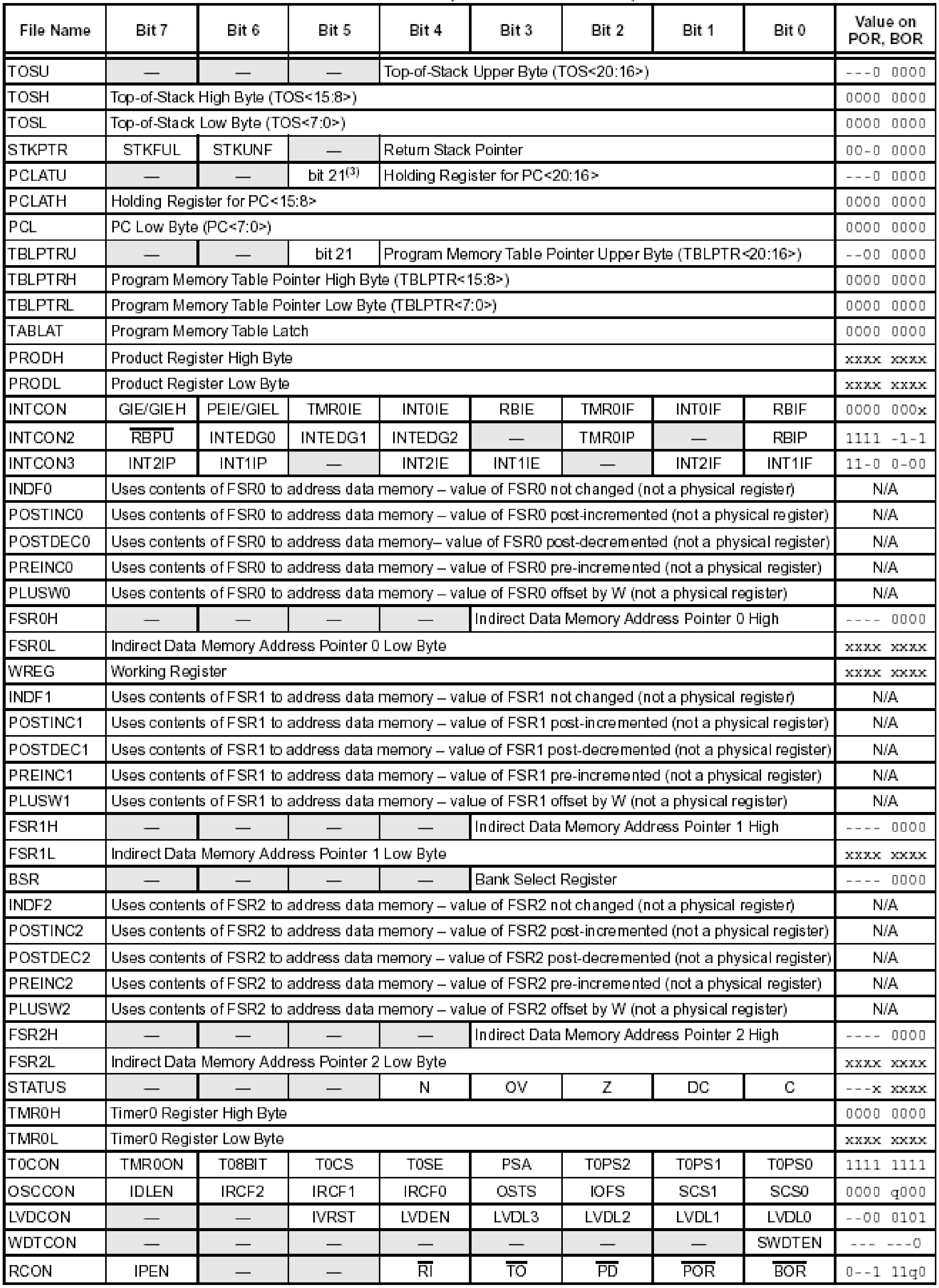
❖ Below is a list of Special Function Registers (SFR):



Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(2)	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2(2)	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2(2)	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2(2)	FBCh	—	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2(2)	FBBh	—	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	—	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	PWM1CON	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCPAS	F96h	—
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	—
FF4h	PRODH	FD4h	—	FB4h	—	F94h	—
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—
FEFh	INDF0(2)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC0(2)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0(2)	FCDh	T1CON	FADh	TXREG	F8Dh	—
FECh	PREINC0(2)	FCCh	TMR2	FACh	TXSTA	F8Ch	—
FEBh	PLUSW0(2)	FCBh	PR2	FABh	RCSTA	F8Bh	—
FEAh	FSR0H	FCAh	T2CON	FAAh	BAUDCTL	F8Ah	LATB
FE9h	FSR0L	FC9h	—	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	—	FA8h	EEDATA	F88h	—
FE7h	INDF1(2)	FC7h	—	FA7h	EECON2	F87h	—
FE6h	POSTINC1(2)	FC6h	—	FA6h	EECON1	F86h	—
FE5h	POSTDEC1(2)	FC5h	—	FA5h	—	F85h	—
FE4h	PREINC1(2)	FC4h	ADRESH	FA4h	—	F84h	—
FE3h	PLUSW1(2)	FC3h	ADRESL	FA3h	—	F83h	—
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	—
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

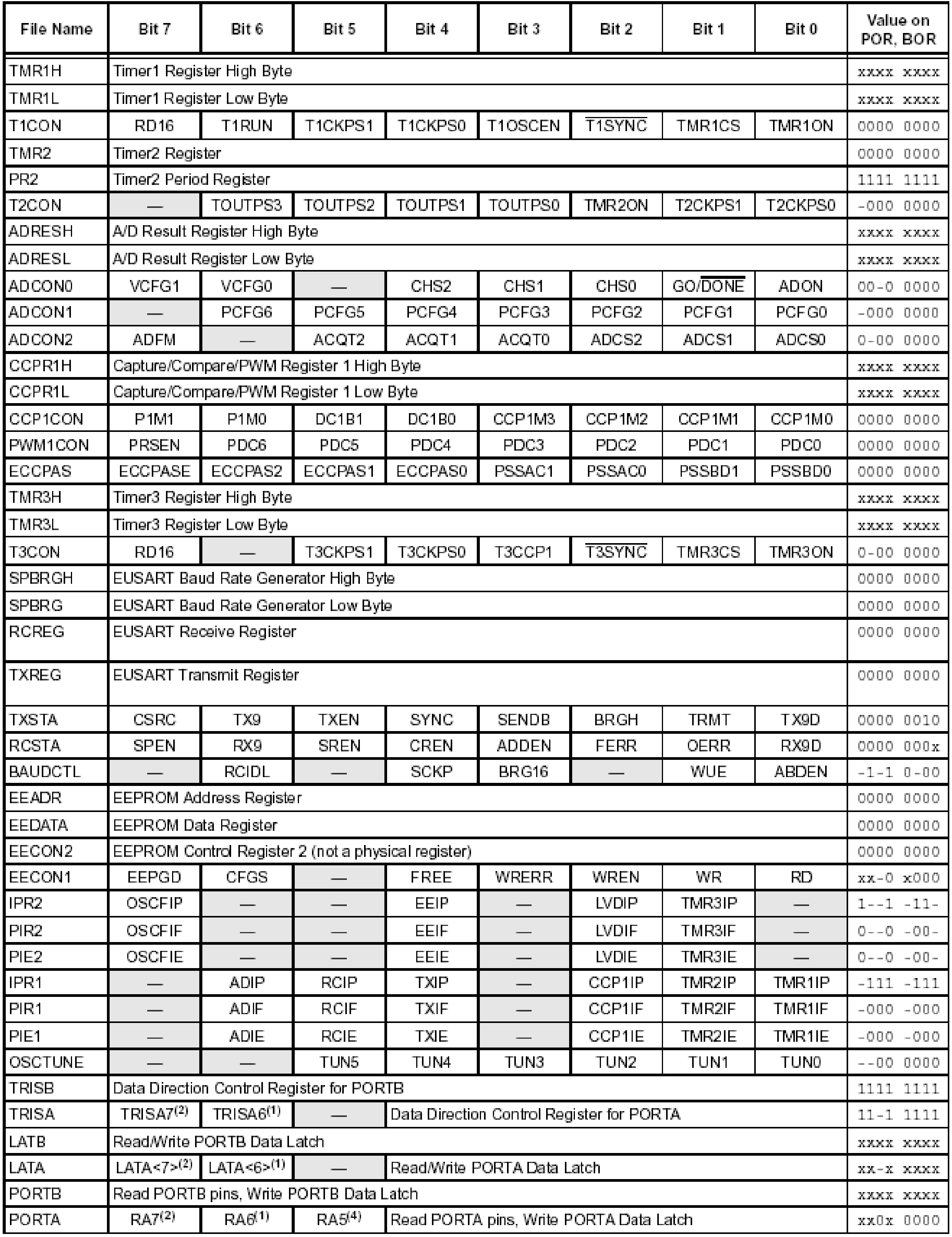
Note: 1) Unimplemented registers are read as ‘0’
2) Not a physical register

Special Function Register (SFR) Summary Table 1 of 2

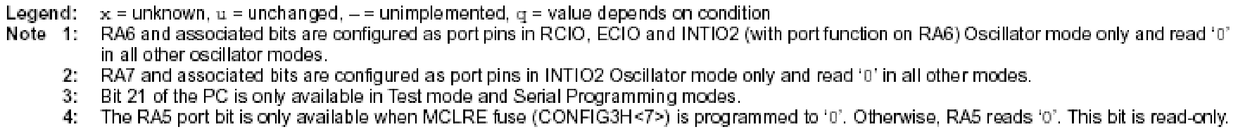


File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR
TOSU	—	—	—	Top-of-Stack Upper Byte (TOS<20:16>)					---0 0000
TOSH	Top-of-Stack High Byte (TOS<15:8>)								0000 0000
TOSL	Top-of-Stack Low Byte (TOS<7:0>)								0000 0000
STKPTR	STKFUL	STKUNF	—	Return Stack Pointer					00-0 0000
PCLATU	—	—	bit 21[3]	Holding Register for PC<20:16>					---0 0000
PCLATH	Holding Register for PC<15:8>								0000 0000
PCL	PC Low Byte (PC<7:0>)								0000 0000
TBLPTRU	—	—	bit 21	Program Memory Table Pointer Upper Byte (TBLPTR<20:16>)					--00 0000
TBLPTRH	Program Memory Table Pointer High Byte (TBLPTR<15:8>)								0000 0000
TBLPTRL	Program Memory Table Pointer Low Byte (TBLPTR<7:0>)								0000 0000
TABLAT	Program Memory Table Latch								0000 0000
PRODH	Product Register High Byte								xxxx xxxx
PRODL	Product Register Low Byte								xxxx xxxx
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x
INTCON2	$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP	1111 -1-1
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF	11-0 0-00
INDF0	Uses contents of FSR0 to address data memory – value of FSR0 not changed (not a physical register)								N/A
POSTINC0	Uses contents of FSR0 to address data memory – value of FSR0 post-incremented (not a physical register)								N/A
POSTDEC0	Uses contents of FSR0 to address data memory– value of FSR0 post-decremented (not a physical register)								N/A
PREINC0	Uses contents of FSR0 to address data memory – value of FSR0 pre-incremented (not a physical register)								N/A
PLUSW0	Uses contents of FSR0 to address data memory – value of FSR0 offset by W (not a physical register)								N/A
FSR0H	—	—	—	—	Indirect Data Memory Address Pointer 0 High				---- 0000
FSR0L	Indirect Data Memory Address Pointer 0 Low Byte								xxxx xxxx
WREG	Working Register								xxxx xxxx
INDF1	Uses contents of FSR1 to address data memory – value of FSR1 not changed (not a physical register)								N/A
POSTINC1	Uses contents of FSR1 to address data memory – value of FSR1 post-incremented (not a physical register)								N/A
POSTDEC1	Uses contents of FSR1 to address data memory – value of FSR1 post-decremented (not a physical register)								N/A
PREINC1	Uses contents of FSR1 to address data memory – value of FSR1 pre-incremented (not a physical register)								N/A
PLUSW1	Uses contents of FSR1 to address data memory – value of FSR1 offset by W (not a physical register)								N/A
FSR1H	—	—	—	—	Indirect Data Memory Address Pointer 1 High				---- 0000
FSR1L	Indirect Data Memory Address Pointer 1 Low Byte								xxxx xxxx
BSR	—	—	—	—	Bank Select Register				---- 0000
INDF2	Uses contents of FSR2 to address data memory – value of FSR2 not changed (not a physical register)								N/A
POSTINC2	Uses contents of FSR2 to address data memory – value of FSR2 post-incremented (not a physical register)								N/A
POSTDEC2	Uses contents of FSR2 to address data memory – value of FSR2 post-decremented (not a physical register)								N/A
PREINC2	Uses contents of FSR2 to address data memory – value of FSR2 pre-incremented (not a physical register)								N/A
PLUSW2	Uses contents of FSR2 to address data memory – value of FSR2 offset by W (not a physical register)								N/A
FSR2H	—	—	—	—	Indirect Data Memory Address Pointer 2 High				---- 0000
FSR2L	Indirect Data Memory Address Pointer 2 Low Byte								xxxx xxxx
STATUS	—	—	—	N	OV	Z	DC	C	---x xxxx
TMR0H	Timer0 Register High Byte								0000 0000
TMR0L	Timer0 Register Low Byte								xxxx xxxx
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111
OSCCON	IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0	0000 q000
LVDCON	—	—	IVRST	LVDEN	LVDL3	LVDL2	LVDL1	LVDL0	--00 0101
WDTCON	—	—	—	—	—	—	—	SWDTEN	--- ---0
RCON	IPEN	—	—	$\overline{\text{RI}}$	$\overline{\text{TO}}$	$\overline{\text{PD}}$	$\overline{\text{POR}}$	$\overline{\text{BOR}}$	0--1 11q0

Special Function Register (SFR) Summary Table 2 of 2



File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR
TMR1H	Timer1 Register High Byte								xxxx xxxx
TMR1L	Timer1 Register Low Byte								xxxx xxxx
T1CON	RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	0000 0000
TMR2	Timer2 Register								0000 0000
PR2	Timer2 Period Register								1111 1111
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000
ADRESH	A/D Result Register High Byte								xxxx xxxx
ADRESL	A/D Result Register Low Byte								xxxx xxxx
ADCON0	VCFG1	VCFG0	—	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON	00-0 0000
ADCON1	—	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0	-000 0000
ADCON2	ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0	0-00 0000
CCPR1H	Capture/Compare/PWM Register 1 High Byte								xxxx xxxx
CCPR1L	Capture/Compare/PWM Register 1 Low Byte								xxxx xxxx
CCP1CON	P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0	0000 0000
PWM1CON	PRSEN	PDC6	PDC5	PDC4	PDC3	PDC2	PDC1	PDC0	0000 0000
ECCPAS	ECCPASE	ECCPAS2	ECCPAS1	ECCPAS0	PSSAC1	PSSAC0	PSSBD1	PSSBD0	0000 0000
TMR3H	Timer3 Register High Byte								xxxx xxxx
TMR3L	Timer3 Register Low Byte								xxxx xxxx
T3CON	RD16	—	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON	0-00 0000
SPBRGH	EUSART Baud Rate Generator High Byte								0000 0000
SPBRG	EUSART Baud Rate Generator Low Byte								0000 0000
RCREG	EUSART Receive Register								0000 0000
TXREG	EUSART Transmit Register								0000 0000
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	0000 0010
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x
BAUDCTL	—	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	-1-1 0-00
EEADR	EEPROM Address Register								0000 0000
EEDATA	EEPROM Data Register								0000 0000
EECON2	EEPROM Control Register 2 (not a physical register)								0000 0000
EECON1	EEPGD	CFGS	—	FREE	WRERR	WREN	WR	RD	xx-0 x000
IPR2	OSCFIP	—	—	EEIP	—	LVDIP	TMR3IP	—	1--1 -11-
PIR2	OSCFIF	—	—	EEIF	—	LVDIF	TMR3IF	—	0--0 -00-
PIE2	OSCFIE	—	—	EEIE	—	LVDIE	TMR3IE	—	0--0 -00-
IPR1	—	ADIP	RCIP	TXIP	—	CCP1IP	TMR2IP	TMR1IP	-111 -111
PIR1	—	ADIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF	-000 -000
PIE1	—	ADIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE	-000 -000
OSCTUNE	—	—	TUN5	TUN4	TUN3	TUN2	TUN1	TUN0	--00 0000
TRISB	Data Direction Control Register for PORTB								1111 1111
TRISA	TRISA7(2)	TRISA6(1)	—	Data Direction Control Register for PORTA					11-1 1111
LATB	Read/Write PORTB Data Latch								xxxx xxxx
LATA	LATA<7>(2)	LATA<6>(1)	—	Read/Write PORTA Data Latch					xx-x xxxx
PORTB	Read PORTB pins, Write PORTB Data Latch								xxxx xxxx
PORTA	RA7(2)	RA6(1)	RA5(4)	Read PORTA pins, Write PORTA Data Latch					xx0x 0000



Legend: x = unknown, u = unchanged, – = unimplemented, q = value depends on condition

Note 1: RA6 and associated bits are configured as port pins in RCIO, ECIO and INTIO2 (with port function on RA6) Oscillator mode only and read '0' in all other oscillator modes.

2: RA7 and associated bits are configured as port pins in INTIO2 Oscillator mode only and read '0' in all other modes.

3: Bit 21 of the PC is only available in Test mode and Serial Programming modes.

4: The RA5 port bit is only available when MCLRE fuse (CONFIG3H<7>) is programmed to '0'. Otherwise, RA5 reads '0'. This bit is read-only.

- I/O Port Set up

  As discussed earlier, microprocessors typically consist only of a Central Processing Unit (CPU) while all other functionality is implemented externally through specialized integrated circuits. All of these modules are accessed and controlled as if they were memory locations by reading and writing to their respective locations.

  PICmicro is a microcontroller which means that, in addition to the CPU, a number of additional functional modules are contained onboard the chip. These additional functional modules include:

  - As many as 16 external PICmicro pins that can be configured as input or output ports.
  - Analog to digital converter module
  - Pulse Width Modulator (PWM) which is used to control the speed of DC motors and other devices that may be controlled with amount of energy delivered.

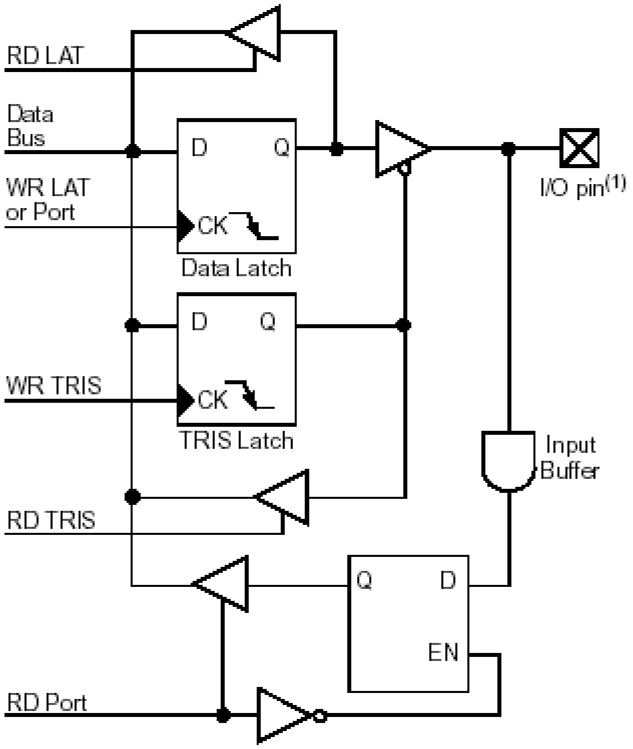
  Later in this chapter, all three of these modules will be discussed.

- External Pin Set up as general purpose I/O Ports

  Some pins of the I/O ports are multiplexed with an alternate function from other modules on the PICmicro. In general, when a peripheral is enabled, the pins used by the peripheral may not be used as general purpose I/O pins. Each port has three registers for its operation. These registers are:

  - TRIS register (data direction register)
  - PORT register (reads the levels on the pins of the device)
  - LAT register (output latch)

A simplified model of a generic I/O port without the interfaces to other peripherals is shown below:



Note 1: I/O pins have diode protection to V_{DD} and V_{SS}.

- PORTA, TRISA and LATA Registers

  PORTA is an 8-bit wide, bidirectional port. Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch.

	RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0
Port A Register:	b7	b6	b5	b4	b3	b2	b1	b0
I/O Pins:	p16	p15	p4	p3	p7	p6	p2	p1
Alternative Uses:	Each I/O pin may be configured for multiple uses, refer to pin definitions earlier in the chapter for a list of alternative uses for each pin.							

The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input. When the pin is set to input it will be in a high-impedance mode. Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output. In this mode the contents of the corresponding bit in the output latch (LATA) will be available on the selected external I/O pin.

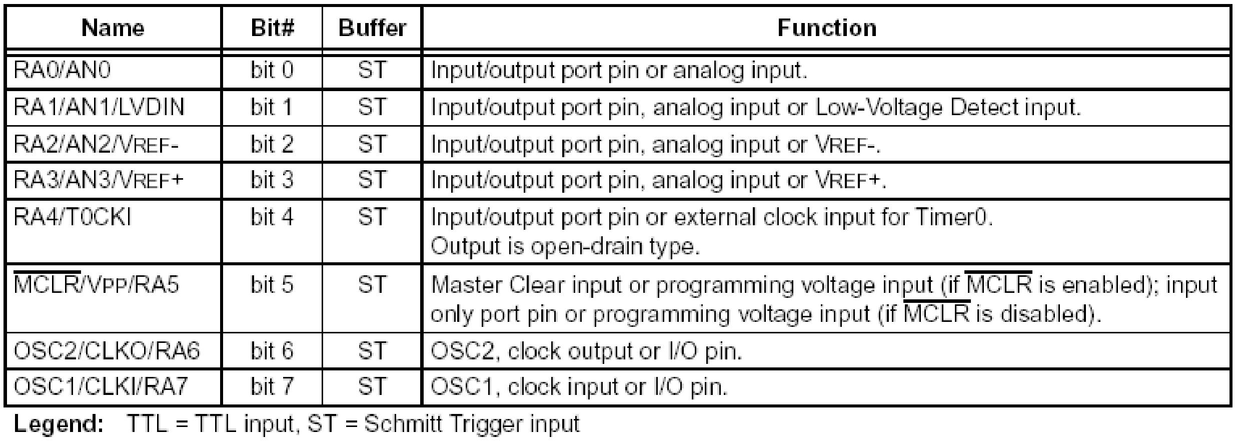
The Data Latch register (LATA) is also memory mapped. Read-modify-write operations on the LATA register read and write the latched output value from and to PORTA.

Any instruction that specifies a file register as part of the instruction performs a Read-Modify-Write (R-M-W) operation. The register is read, the data is modified, and the result is stored according to either the instruction or the destination designator ‘d’. A read operation is performed on a register even if the instruction writes to that register. It is important to consider the impact of a read on the configuration before using read-modify-write instructions.

- Example of initializing PortA

```
CLRF      PORTA       ; Initialize PORTA by clearing output data latches
CLRF      LATA        ; Alternate method to clear output data latches
MOVLW     0x7F        ; Configure A/D
MOVWF     ADCON1       ; for digital inputs
MOVLW     0xF0        ; Value used to initialize data direction
MOVWF     TRISA       ; Set RA<3:0> as outputs and RA<7:4> as inputs
```

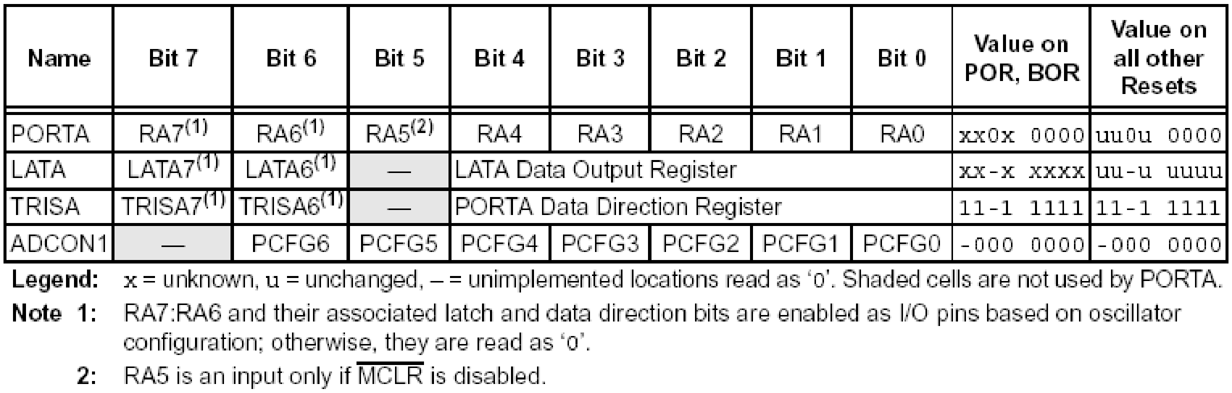
- Port A Functions Summary



Name	Bit#	Buffer	Function
RA0/AN0	bit 0	ST	Input/output port pin or analog input.
RA1/AN1/LVDIN	bit 1	ST	Input/output port pin, analog input or Low-Voltage Detect input.
RA2/AN2/VREF-	bit 2	ST	Input/output port pin, analog input or VREF-.
RA3/AN3/VREF+	bit 3	ST	Input/output port pin, analog input or VREF+.
RA4/T0CKI	bit 4	ST	Input/output port pin or external clock input for Timer0. Output is open-drain type.
$\overline{\text{MCLR}}$/VPP/RA5	bit 5	ST	Master Clear input or programming voltage input (if $\overline{\text{MCLR}}$ is enabled); input only port pin or programming voltage input (if $\overline{\text{MCLR}}$ is disabled).
OSC2/CLKO/RA6	bit 6	ST	OSC2, clock output or I/O pin.
OSC1/CLKI/RA7	bit 7	ST	OSC1, clock input or I/O pin.

Legend: TTL = TTL input, ST = Schmitt Trigger input

- Port A Associated Registers Summary



Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
PORTA	RA7[1]	RA6[1]	RA5[2]	RA4	RA3	RA2	RA1	RA0	xx0x 0000	uu0u 0000
LATA	LATA7[1]	LATA6[1]	—	LATA Data Output Register					xx-x xxxx	uu-u uuuu
TRISA	TRISA7[1]	TRISA6[1]	—	PORTA Data Direction Register					11-1 1111	11-1 1111
ADCON1	—	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0	-000 0000	-000 0000

Legend: x = unknown, u = unchanged, – = unimplemented locations read as ‘0’. Shaded cells are not used by PORTA.

Note 1: RA7:RA6 and their associated latch and data direction bits are enabled as I/O pins based on oscillator configuration; otherwise, they are read as ‘0’.

2: RA5 is an input only if $\overline{\text{MCLR}}$ is disabled.

- PORTB, TRISB and LATB Registers
  PORTB is an 8-bit wide, bidirectional port. Reading the PORTB register reads the status of the pins, whereas writing to it will write to the port latch.

	RB7	**RB6**	**RB5**	**RB4**	**RB3**	**RB2**	**RB1**	**RB0**
Port B Register:	b7	b6	b5	b4	b3	b2	b1	b0
I/O Pins:	p13	p12	p11	p10	p18	p17	p9	p8
Alternating Uses:	Each I/O pin may be configured for multiple uses, refer to pin definitions earlier in the chapter for a list of alternative uses for each pin.							

The corresponding data direction register is TRISB. Setting a TRISB bit (= 1) will make the corresponding PORTB pin an input. When the pin is set to input it will be in a high-impedance mode. Clearing a TRISB bit (= 0) will make the corresponding PORTB pin an output. In this mode the contents of the corresponding bit in the output latch (LATB) on the selected pin.

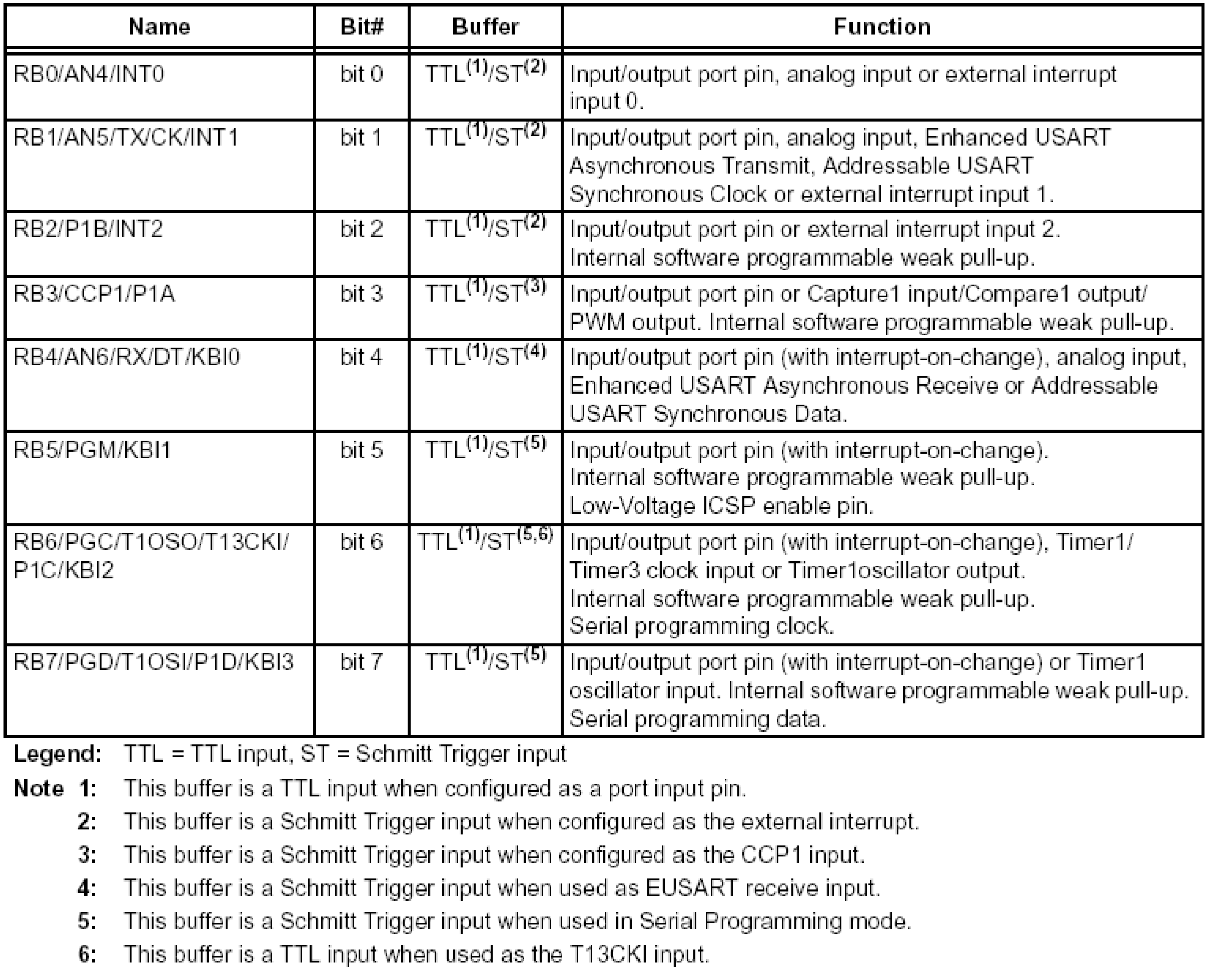
The Data Latch register (LATB) is also memory mapped. Read-modify-write operations on the LATB register read and write the latched output value from and to PORTB.

Any instruction that specifies a file register as part of the instruction performs a Read-Modify-Write (R-M-W) operation. The register is read, the data is modified, and the result is stored according to either the instruction or the destination designator ‘d’. A read operation is performed on a register even if the instruction writes to that register. It is important to consider the impact of a read on the configuration before using read-modify-write instructions.

- Example of initializing PortB

```
CLRF      PORTB       ; Initialize PORTB by clearing output data latches
CLRF      LATB        ; Alternate method to clear output data latches
MOVLW     0x70        ; Set RB0 , RB1, RB4 (Pins 8, 9  &10) as
MOVWF     ADCON1       ; digital I/O pins
MOVLW     0xCF        ; Value used to initialize data direction
MOVWF     TRISB       ; Set RB<3:0> as inputs, RB<5:4> as outputs and
                      ; RB<7:6> as inputs
```

- Port B Functions Summary



Name	Bit#	Buffer	Function
RB0/AN4/INT0	bit 0	TTL[1]/ST[2]	Input/output port pin, analog input or external interrupt input 0.
RB1/AN5/TX/CK/INT1	bit 1	TTL[1]/ST[2]	Input/output port pin, analog input, Enhanced USART Asynchronous Transmit, Addressable USART Synchronous Clock or external interrupt input 1.
RB2/P1B/INT2	bit 2	TTL[1]/ST[2]	Input/output port pin or external interrupt input 2. Internal software programmable weak pull-up.
RB3/CCP1/P1A	bit 3	TTL[1]/ST[3]	Input/output port pin or Capture1 input/Compare1 output/ PWM output. Internal software programmable weak pull-up.
RB4/AN6/RX/DT/KBI0	bit 4	TTL[1]/ST[4]	Input/output port pin (with interrupt-on-change), analog input, Enhanced USART Asynchronous Receive or Addressable USART Synchronous Data.
RB5/PGM/KBI1	bit 5	TTL[1]/ST[5]	Input/output port pin (with interrupt-on-change). Internal software programmable weak pull-up. Low-Voltage ICSP enable pin.
RB6/PGC/T1OSO/T13CKI/ P1C/KBI2	bit 6	TTL[1]/ST[5,6]	Input/output port pin (with interrupt-on-change), Timer1/ Timer3 clock input or Timer1oscillator output. Internal software programmable weak pull-up. Serial programming clock.
RB7/PGD/T1OSI/P1D/KBI3	bit 7	TTL[1]/ST[5]	Input/output port pin (with interrupt-on-change) or Timer1 oscillator input. Internal software programmable weak pull-up. Serial programming data.

Legend: TTL = TTL input, ST = Schmitt Trigger input

Note 1: This buffer is a TTL input when configured as a port input pin.

2: This buffer is a Schmitt Trigger input when configured as the external interrupt.

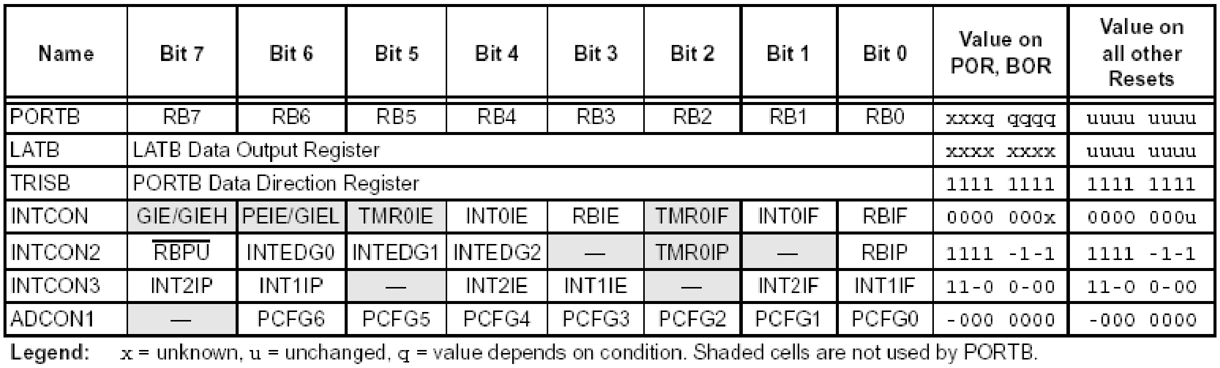
3: This buffer is a Schmitt Trigger input when configured as the CCP1 input.

4: This buffer is a Schmitt Trigger input when used as EUSART receive input.

5: This buffer is a Schmitt Trigger input when used in Serial Programming mode.

6: This buffer is a TTL input when used as the T13CKI input.

- Port B Associated Registers Summary



Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxq qqqq	uuuu uuuu
LATB	LATB Data Output Register								xxxx xxxx	uuuu uuuu
TRISB	PORTB Data Direction Register								1111 1111	1111 1111
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x	0000 000u
INTCON2	$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP	1111 -1-1	1111 -1-1
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF	11-0 0-00	11-0 0-00
ADCON1	—	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0	-000 0000	-000 0000

Legend: x = unknown, u = unchanged, q = value depends on condition. Shaded cells are not used by PORTB.

- Example of Basic Input/Output Configuration
  As described earlier, configuring external PICmicro pins as input/output is as simple as writing to SFR registers ADCON1, TRISA and TRISB. Below is a sample pin configuration code from a counter program:

The following lines clear the data in PORTA and PORTB.

```
CLRF        PORTA
CLRF        PORTB
```

The following line sets the W register (accumulator) to value 0x7F = 01111111

```
MOVLW       0x7F
```

The W register is used as a temporary location for values. In this case the value 0x7F has been placed in W register first. The following line copies the value of W register to ADCON1 register. The ADCON1 register is one of three registers that control the operation of the PICmicro's built in Analog to Digital Converter (ADC). Setting the value of ADCON1 to 0x7F tells the PIC that pins 1, 2, 6, 7, 8, 9 and 10 will be used for input/output rather than for the ADC.

```
MOVWF       ADCON1
```

The following line sets the W register (accumulator) to value 0x00 = 00000000

```
MOVLW       0x00
```

The following line copies the contents of the W register to the TRISB register.
The TRISB register is the control register for PORTB. The bits in TRISB signify which direction the data is flowing in PORTB (1 = Input, 0 = Output).

```
MOVWF       TRISB
```

The following line sets the W register (accumulator) to value 0x01 = 00000001

```
MOVLW       0x01
```

The following line copies the contents of the W register to the TRISA register.
The TRISA register is the control register for PORTA. The bits in TRISA signify which direction data is flowing in PORTA (1 = Input, 0 = Output).

```
MOVWF       TRISA
```

Now that the input/output pins have been configured, the user can write to or read from these input/output pins by writing and reading from PORTA and PORTB registers. For example, the following code writes data (0xAB) to the 8 pins of PortA:

```
      MOVLW       0xAB
      MOVWF       PORTA
```

3.3. Additional Resources

- Staff. Microchip PIC 18F1220/1320 Data Sheet. (2004) Microchip Technology Incorporated.

- Huang,. PIC Microcontroller: An Introduction to Software & Hardware Interfacing, (2004) Thomson.

- Reese. Microprocessor: From Assembly Language to C using the PIC18Fxxx2. (2003) Course Technology.

- Peterson. Computer Organization and Design, (2007) Elsevier Service.

3.4. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 4. PROGRAM FLOW, EVENT HANDLING AND CONTROL

Key concepts and Overview

- Overview
- Stack Operations
- Procedure Call and Return Instructions
- Interrupt/exception handling
- Clock and Oscillator
- Timers
- Power Management
- Reset
- Analog-to-Digital Converter
- Pulse Width Modulation (PWM)
- Additional Resources

4.1. Overview

As discussed earlier, the CPU executes instructions in a sequential fashion. PICmicro will execute the instruction in the word (2 bytes) that is pointed to by the Program Counter (PC). Upon completion of the current instruction, PC is incremented by 2 and executes the next instruction word pointed to by the PC. There are a number of instructions and events that are designed to move PC to another location other than PC+2. The following are the most common of these instructions and events:

- Branch instructions
  As seen in earlier chapters, branch instructions redirect the PC to a location in memory other than (PC + 2). Each branch instruction may test a specific condition. For example: "BC n" will cause the PC to move to n if the Carry flag is set, otherwise it will continue executing the next instruction word pointed to by PC+2. Branch instructions are used to develop high-level language "If-then-else" statements, other conditional statements, and loop constructs.
- Procedure Call and Return Instructions
  The Call instruction directs the PC to a new location similar to the Branch instructions. Additionally, it keeps tracks of the original (PC+4) so that it may return to this location after completing a set of instructions at the new location. The Return instruction is used to return to the location of the instruction immediately following the Call instruction. The implementation of high-level language functions and procedures rely on these types of instructions.

- Interrupts and exceptions
  Interrupts and exceptions are required for implementation of event detection and handling. Exception refers to a software-initiated interrupt. We will use interrupt to refer to both exceptions and interrupts. Interrupts may occur at any time during the program execution. Once an Interrupt is detected, the PC will be changed to point to a pre-determined location in the memory corresponding to the detected interrupt. The code that starts at this new location is written to handle the interrupt or exception.

- Timers
  There are situations where the CPU has to wait for a specified amount of time. Although the processor may be placed in a wait loop by executing NOP instructions, this approach has a number of disadvantages:

  - The actual time has to be experimentally determined since calculations based on instruction cycle time may be difficult to make.
  - In a multiprocessing system, the loop only counts the time allocated to the process executing the wait loop and not the time used by other processes.
  - The processor is not able to perform other tasks while it is in the wait loop.

  - Timers solve these issues by allowing the CPU to continue with normal task execution until a timer timeout interrupt has occurred. The Timer timeout is able to generate a timer interrupt after a specified time duration which would result in redirecting the PC to a new location in the memory to execute the code that handles the timed event.

- Power managed Modes
  Increasingly, most processors have the power management capability in order to save power. The key feature is the ability to transition from sleep to active mode driven by an external event. For example: When the user presses a key on a computer, or upon detection of network activity. Using this functionality, the CPU may be put into a standby or low power mode until it is needed.

4.2. Stack Operations

Stacks are special memory locations used to store return addresses and other information that will need to be retrieved later. This space is called a stack since one can visualize data being stacked on the top of each other. There are specific operations for adding and removing words from stack.

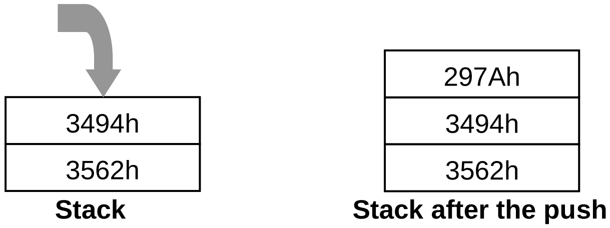
STKPTR Special Function Register contains information about the stack status (empty or full) and stack point as shown below:

Bit 7	Bit 6	Bit5	Bit4	
STKFUL {1: Stack is Full}	STKUNF {1: Stack is Empty}	—	Stack Points 1-31 {0 is not valid}	STKPTR

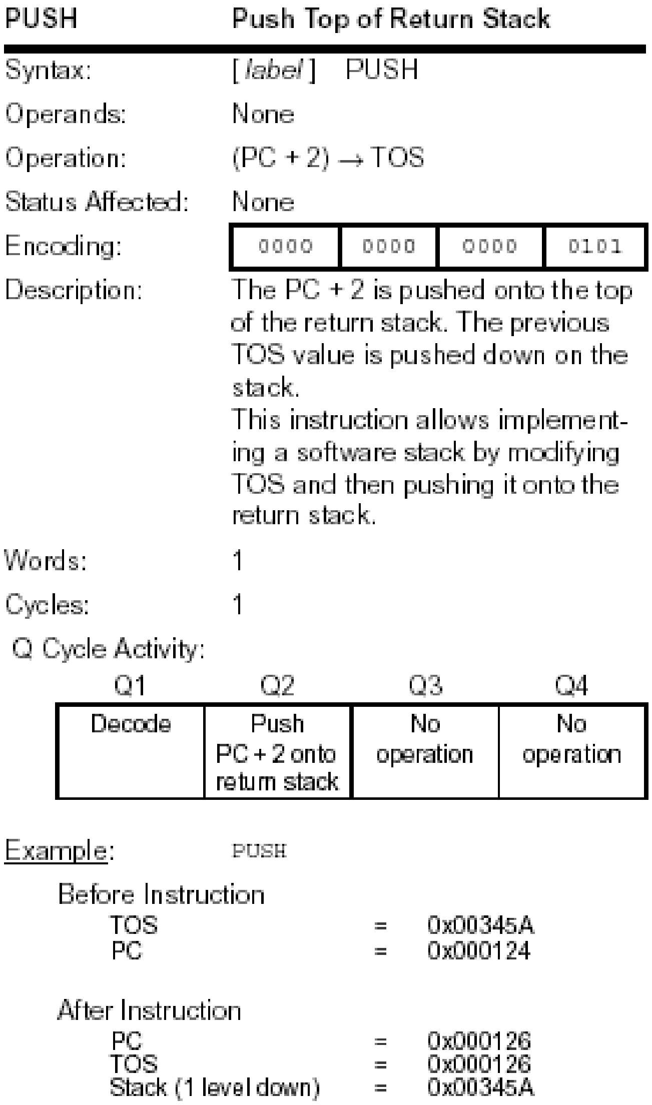
The most common use of the stack is for subroutines. When a CALL instruction is executed, the current value of (PC+4) is added (or "pushed") on to the stack so that it can be recovered (or "popped") during a RETURN instruction. The value recovered by RETURN is used as the location of the next instruction.

- ❖ PUSH Instruction
  Adding a word to the stack is called pushing a word onto the stack which is shown below:

<u>Address</u>	<u>Code/ Program</u>
2978h	PUSH



Below are the specifications for the PUSH instruction:



PUSH **Push Top of Return Stack**

Syntax:	[*label*] PUSH
Operands:	None
Operation:	(PC + 2) → TOS
Status Affected:	None
Encoding:	0000 0000 0000 0101
Description:	The PC + 2 is pushed onto the top of the return stack. The previous TOS value is pushed down on the stack. This instruction allows implementing a software stack by modifying TOS and then pushing it onto the return stack.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Push PC + 2 onto return stack	No operation	No operation

<u>Example</u>: PUSH

Before Instruction

TOS	=	0x00345A
PC	=	0x000124

After Instruction

PC	=	0x000126
TOS	=	0x000126
Stack (1 level down)	=	0x00345A

<u>Note</u>

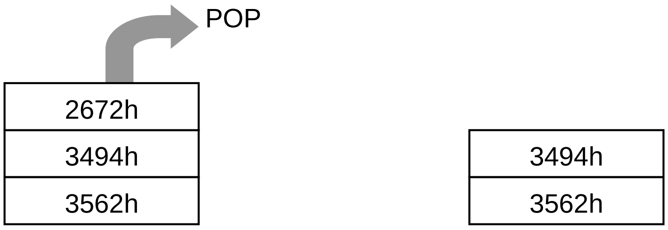
- 21-bit value for the content of the top of stack (TOS) is located at TOSU, TOSH and TOSL Special Function Registers (Top Of Stack Upper, High, Low).
- Bits 6 and 7 of STKPTR Special Function Register indicate whether stack is empty and full, respectively.

  Before attempting to add data to the stack, it is important to check bit 7 of STKPTR to ensure that the stack is not full.

  Similarly, before attempting to remove data from the stack, it is important to check bit 6 of STKPTR to ensure that the stack is not empty.
- Program Counter (PC) is the address of the next instruction to be executed.

- ❖ Pop Instruction

The removal operation is called popping a word from the stack which is shown below:



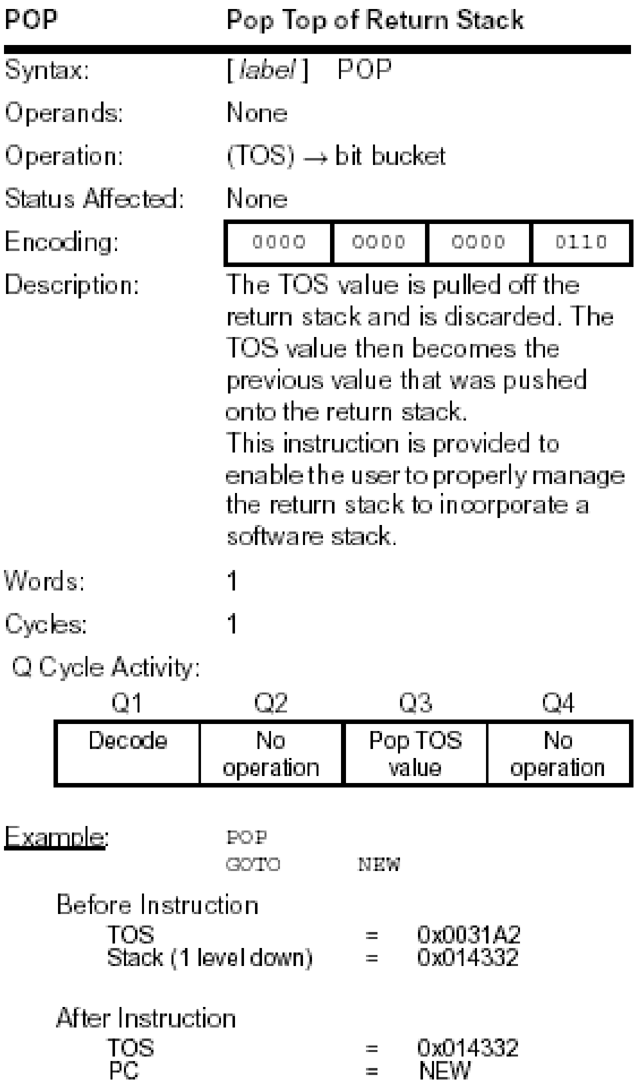
Stack
2672h
3494h
3562h

Stack

Stack after the pop
3494h
3562h

Stack after the pop

Below is the specification for the pop instruction:



POP **Pop Top of Return Stack**

Syntax:	[*label*] POP
Operands:	None
Operation:	(TOS) → bit bucket
Status Affected:	None
Encoding:	0000 0000 0000 0110
Description:	The TOS value is pulled off the return stack and is discarded. The TOS value then becomes the previous value that was pushed onto the return stack. This instruction is provided to enable the user to properly manage the return stack to incorporate a software stack.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Pop TOS value	No operation

Example:
```
POP
GOTO     NEW
```

Before Instruction

TOS = 0x0031A2
Stack (1 level down) = 0x014332

After Instruction

TOS = 0x014332
PC = NEW

Note

- 21-bit value for the content of the top of stack (TOS) is located at TOSU, TOSH and TOSL Special Function Registers (Top Of Stack Upper, High, Low).
- Bits 6 and 7 of STKPTR Special Function Register indicate whether stack is empty and full, respectively.

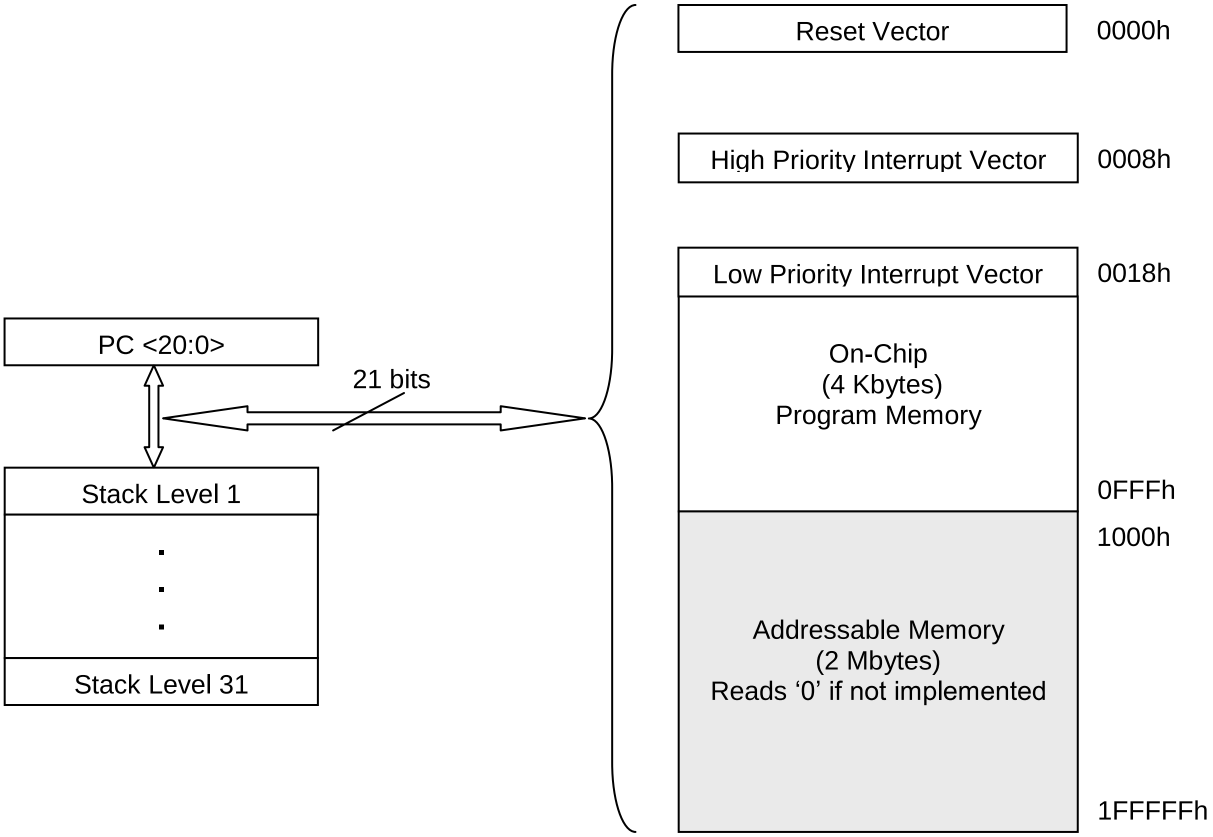
  Before attempting to add data to the stack, it is important to check bit 7 of STKPTR to ensure that the stack is not full.

  Similarly, before attempting to remove data from the stack, it is important to check bit 6 of STKPTR to ensure that the stack is not empty.
- Program Counter (PC) is the address of the next instruction to be executed.

- Stack Usage
  PICmicro has 31 stack levels (Level 1 – 31) which are most commonly used for saving data between procedure calls or interrupts. In most cases, stack stores the value of key registers or PC locations that may be needed later.

  Stack memory space does not occupy any of the available program or data memory. However, the designer may decide to use specific memory to extend stack space beyond the 31 levels.

  The following diagram depicts an overview of PICmicro stack and user memory space:



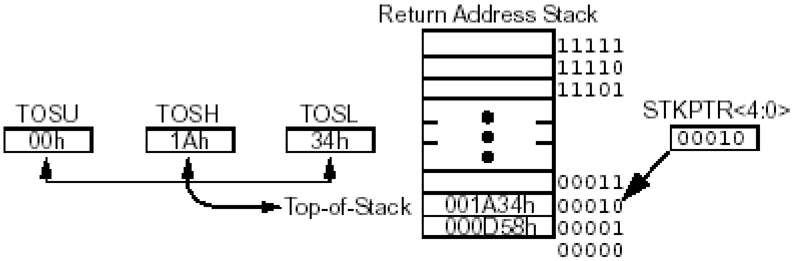
User Memory Space (2 Mbytes)

- Return Address Stack
  The return address stack allows any combination of up to 31 program calls and interrupts to occur before a RETURN is needed. The Program Counter (PC) for next instruction is pushed onto the stack when a CALL or RCALL instruction is executed, or an interrupt is acknowledged. The PC value for next instruction is pulled off the stack on a RETURN, a RETLW or a RETFIE instruction. PCLATU and PCLATH registers are not affected by any of the RETURN or CALL instructions.

  The stack operates as a 31-word by 21-bit memory and a 5-bit stack pointer, with the Stack Pointer initialized to 00000b after all Resets. There is no memory location is associated with Stack Pointer, 00000b. This is only a Reset value. During a CALL type instruction, the Stack Pointer is first incremented, and then the PC value of the next instruction is written to the memory location pointed to by the Stack Pointer (STKPTR) register.

During a RETURN type instruction, the contents of the memory location pointed to by the Stack Pointer are written to the PC, and then the Stack Pointer is decremented. RETURN type instruction causes the contents of the memory location pointed to by the Stack Pointer to be transferred to the PC followed by Stack Pointer decrement (which is the same as a pop from the stack). The Stack Pointer is readable and writable, and the address on the top of the stack is readable and writable through the Top-Of-Stack (TOS) Special File Registers. Data can also be pushed to or popped from the stack using the TOS Special Function Registers. As mentioned earlier, the STKPTR register also contains status bits indicating if the stack is full or empty.

- Top-Of-Stack Access
  The top of the stack is readable and writable. Three register locations, TOSU, TOSH and TOSL (Top-Of-Stack Upper, High, and Low), hold the contents of the stack location pointed to by the STKPTR register as shown below:



Access to top of stack allows users to implement a software stack if necessary. After a CALL, RCALL or interrupt, the software can read the pushed value by reading the TOSU, TOSH and TOSL registers. These values can be copied to a user-defined software stack. At return time, the software can replace TOSU, TOSH and TOSL with the values saved on the software stack, and then do a return.

The user must disable the global interrupt enable bits while accessing the stack to prevent inadvertent stack corruption (refer to the interrupt section for more detail).

- Example – Determine the value of TOSU, TOSH and TOSL after the following instruction has been executed:

Address	**Instruction**
07FEh	PUSH

Solution:
Pushed on stack (PC + 2 = 0x800)
TOSU=0x00, TOSH=0x08, TOSL=0x00

- Example – Determine the value of TOSU, TOSH and TOSL after the following instruction has been executed:

Address	**Instruction**
001890h	POP

Solution:
Unknown (insufficient information)

- Example – Determine the value of data memory locations 0xFFF, 0xFFE and 0xFFD after the execution of:

Address	**Instruction**
0x292	PUSH

Solution:

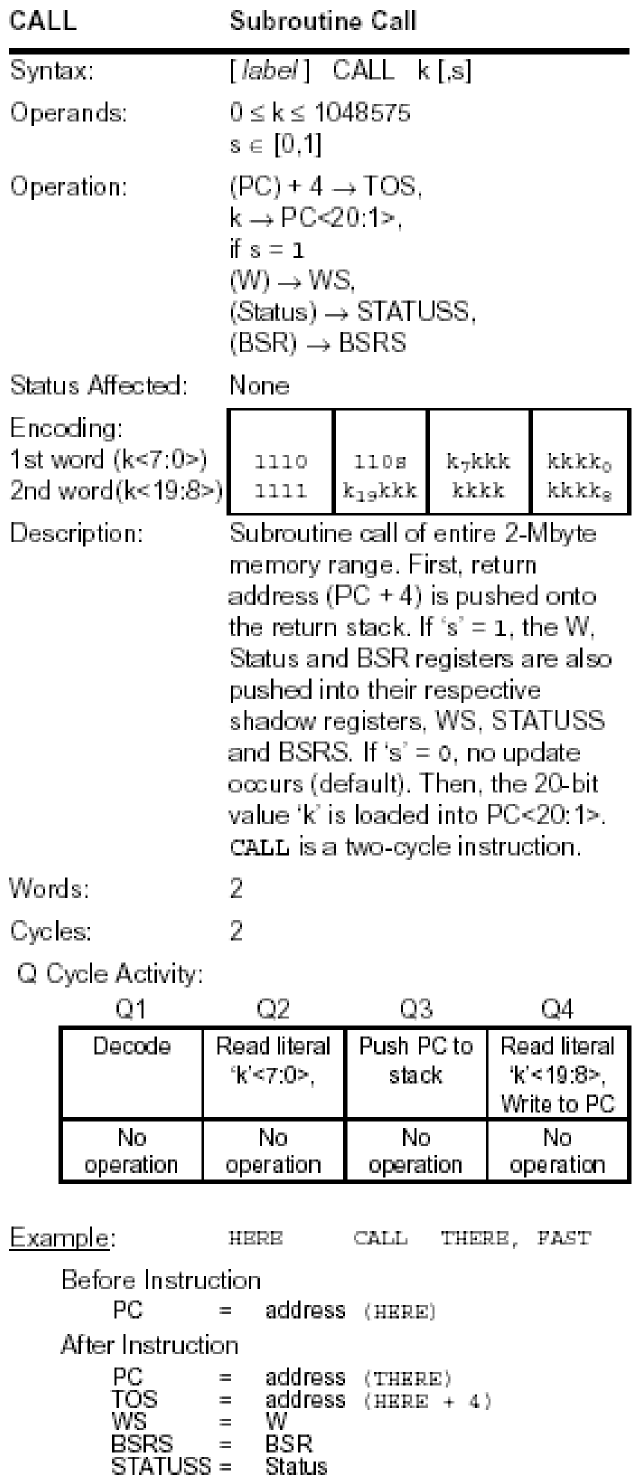
4.3. Procedure Call and Return Instructions

Procedure call and return instructions are important in a programmer's ability to create blocks of codes that could be shared by multiple parts of one program or multiple programs, eliminating the need to rewrite the same code multiple times. The major benefits of this type of reuse are reduction in code size and ease of maintenance since any fix only requires change to one code segment.

Procedure call and return instructions have this advantage over Branch and Goto instructions because of their ability to return the PC to the code immediately following the CALL instruction. Again, the high level language functions and procedures are implemented using Procedure CALL and RETURN instructions.

PICmicro provides CALL, RCALL, RETURN and RETLW in support of procedures as described below:

- ❖ CALL n,s



CALL **Subroutine Call**

Syntax: [*label*] CALL k [,s]

Operands: 0 ≤ k ≤ 1048575
s ∈ [0,1]

Operation: (PC) + 4 → TOS,
k → PC<20:1>,
if s = 1
(W) → WS,
(Status) → STATUSS,
(BSR) → BSRS

Status Affected: None

Encoding:

1st word (k<7:0>)	1110	110s	$k_7$kkk	kkkk_0
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: Subroutine call of entire 2-Mbyte memory range. First, return address (PC + 4) is pushed onto the return stack. If 's' = 1, the W, Status and BSR registers are also pushed into their respective shadow registers, WS, STATUSS and BSRS. If 's' = 0, no update occurs (default). Then, the 20-bit value 'k' is loaded into PC<20:1>. CALL is a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>,	Push PC to stack	Read literal 'k'<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: HERE CALL THERE, FAST

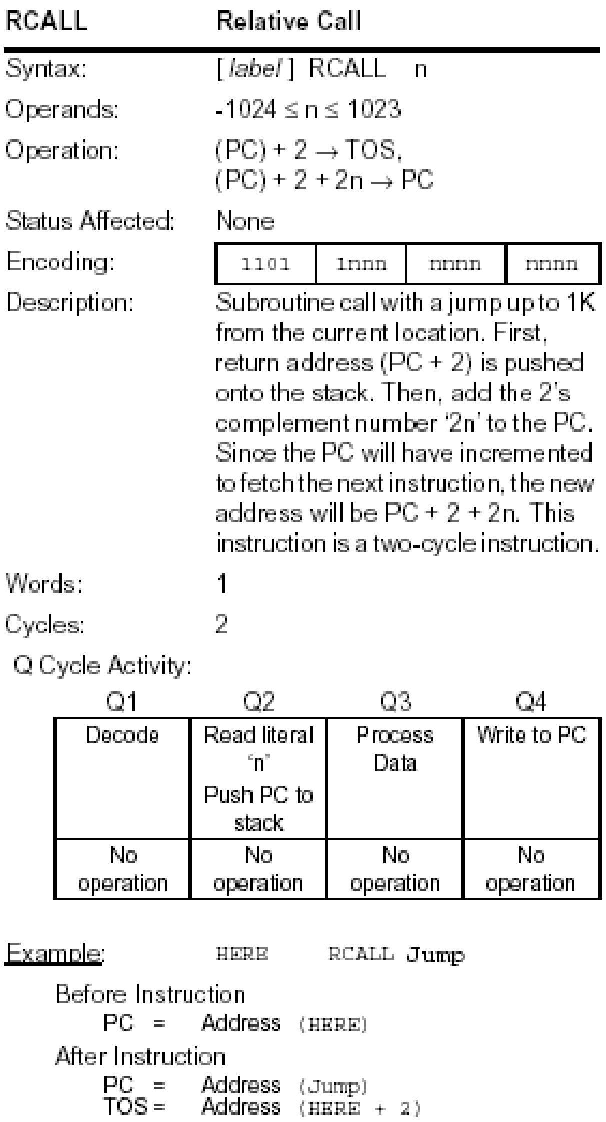
Before Instruction
PC = address (HERE)

After Instruction
PC = address (THERE)
TOS = address (HERE + 4)
WS = W
BSRS = BSR
STATUSS = Status

Notes:

-

- RCALL n



RCALL **Relative Call**

Syntax: [*label*] RCALL n

Operands: -1024 ≤ n ≤ 1023

Operation: (PC) + 2 → TOS,
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1101	1nnn	nnnn	nnnn

Description: Subroutine call with a jump up to 1K from the current location. First, return address (PC + 2) is pushed onto the stack. Then, add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'n' Push PC to stack	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: HERE RCALL Jump

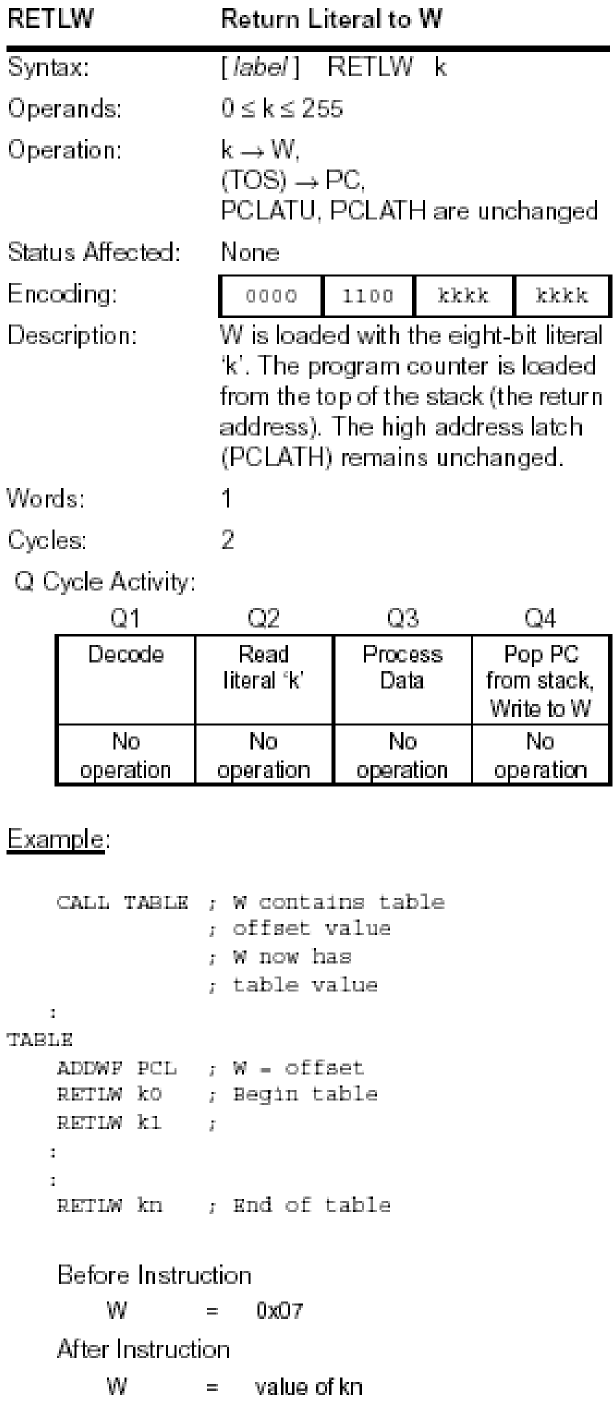
Before Instruction
PC = Address (HERE)

After Instruction
PC = Address (Jump)
TOS = Address (HERE + 2)

Notes:

-

❖ RETLW



RETLW	Return Literal to W
Syntax:	[*label*] RETLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → W, (TOS) → PC, PCLATU, PCLATH are unchanged
Status Affected:	None
Encoding:	0000 1100 kkkk kkkk
Description:	W is loaded with the eight-bit literal 'k'. The program counter is loaded from the top of the stack (the return address). The high address latch (PCLATH) remains unchanged.
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Pop PC from stack, Write to W
No operation	No operation	No operation	No operation

Example:

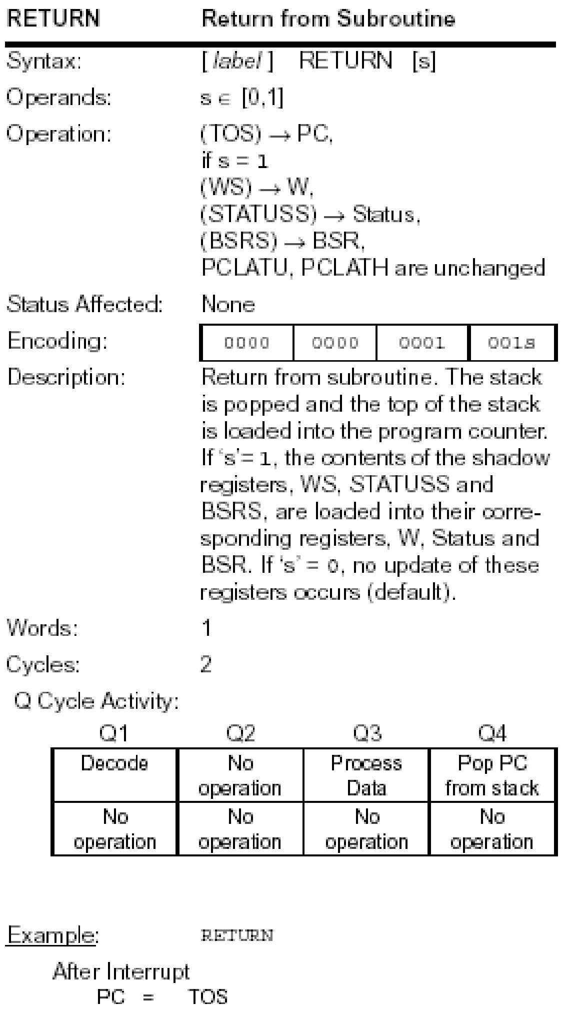
```
    CALL TABLE  ; W contains table
                ; offset value
                ; W now has
                ; table value
   :
TABLE
    ADDWF PCL   ; W = offset
    RETLW k0    ; Begin table
    RETLW k1    ;
   :
   :
    RETLW kn    ; End of table
```

Before Instruction
W = 0x07
After Instruction
W = value of kn

Notes:

▪

- RETURN



RETURN **Return from Subroutine**

Syntax:	[*label*] RETURN [s]
Operands:	s ∈ [0,1]
Operation:	(TOS) → PC, if s = 1 (WS) → W, (STATUSS) → Status, (BSRS) → BSR, PCLATU, PCLATH are unchanged
Status Affected:	None
Encoding:	0000 0000 0001 001s
Description:	Return from subroutine. The stack is popped and the top of the stack is loaded into the program counter. If 's'= 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, W, Status and BSR. If 's' = 0, no update of these registers occurs (default).
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	Pop PC from stack
No operation	No operation	No operation	No operation

Example: RETURN

After Interrupt
PC = TOS

- Example - For the following code segment:

```
Address    Instruction                    .
52h                    CALL    add_one
                       DECF    0x81

           ….

75h        add_one:    INCF    0x81

                       …

                       RETURN
```

a) Are all the shown addresses valid? If not, what is the next valid address for any invalid address?

b) Determine location of instruction "DECF".

c) Determine machine Code for each shown instruction.

d) Determine top of stack value after each instruction. Assuming TOS is 0x1298 before code execution starts.

Solution:

- **Example** - Function CALL and Return
  a) Write an assembly code function “diff” that subtracts two 8–bit operands and returns the result.
  b) Write the equivalent machine code.
  c) Call Diff from location 0x128 when Top of Stack (TOS) is set to 0x1232. Show value of PC and TOS immediately before CALL, before Return, and after Return.

Solution:

a) diff function

```
            op1       equ       0x80
            op2       equ       0x81
            result    equ       0x82

            org       0x200
            ; diff function returns result=op1 – op2
   diff:    MOVF      op2, W
            SUBWF     op1, W
            MOVWF     result
            RETURN    1
```

b)

c)

4.4. Interrupt/exception handling

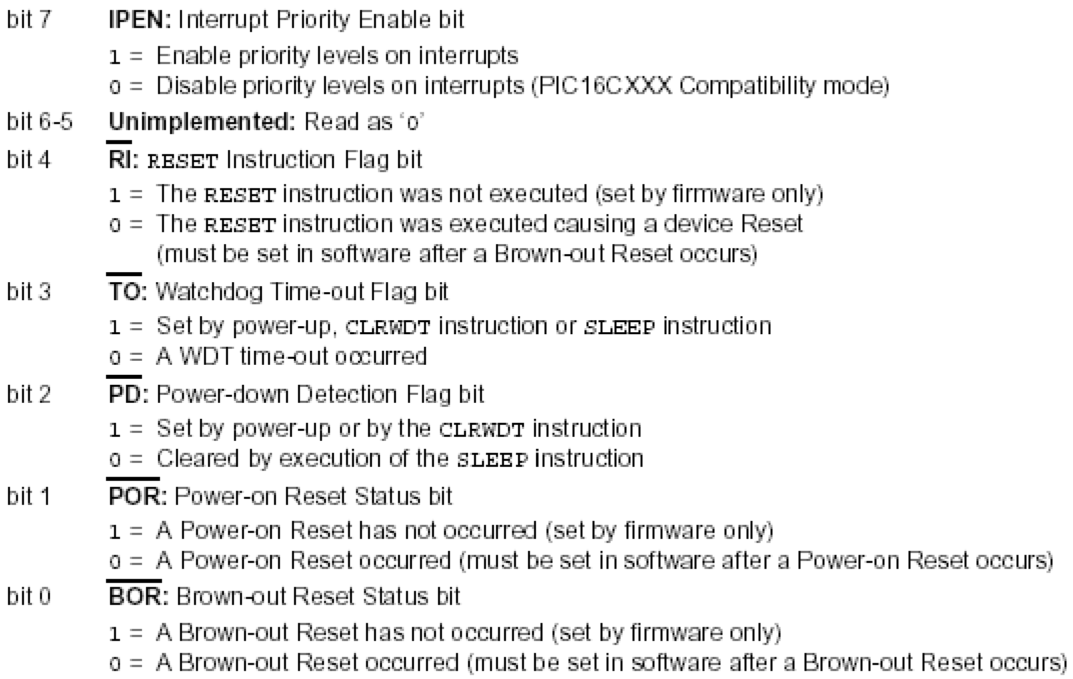
As discussed earlier, interrupts are required for event detection and handling. Interrupts may occur at anytime. When they do, the location of the next instruction is pushed onto the stack, and the PC is changed to point to a pre-determined location in the memory. The code that starts at this new location is written to handle the interrupt or exception.

PICmicro handles interrupts from multiple sources. Additionally, the ability to assign interrupt (INT1 & INT2) priority enables the designer to assign a low or high priority to each interrupt source. The high priority interrupt events can override any low priority interrupts that may be in progress. The high priority interrupt vector is at 000008h program memory location, and the low priority interrupt vector is at 000018h program memory location. Interrupt vector is the location that PC will be set to after an interrupt has occurred and has been acknowledged.

The following four SFR registers are used to control interrupt operations (there are another six SFR registers that will be discussed later):

❖ RCON Register

Bit 7							Bit0	
IPEN	—	—	RI’	TO’	PD’	POR’	BOR’	RCON



bit 7 **IPEN:** Interrupt Priority Enable bit
1 = Enable priority levels on interrupts
0 = Disable priority levels on interrupts (PIC16CXXX Compatibility mode)

bit 6-5 **Unimplemented:** Read as ‘0’

bit 4 **$\overline{\text{RI}}$:** RESET Instruction Flag bit
1 = The RESET instruction was not executed (set by firmware only)
0 = The RESET instruction was executed causing a device Reset
(must be set in software after a Brown-out Reset occurs)

bit 3 **$\overline{\text{TO}}$:** Watchdog Time-out Flag bit
1 = Set by power-up, CLRWDT instruction or SLEEP instruction
0 = A WDT time-out occurred

bit 2 **$\overline{\text{PD}}$:** Power-down Detection Flag bit
1 = Set by power-up or by the CLRWDT instruction
0 = Cleared by execution of the SLEEP instruction

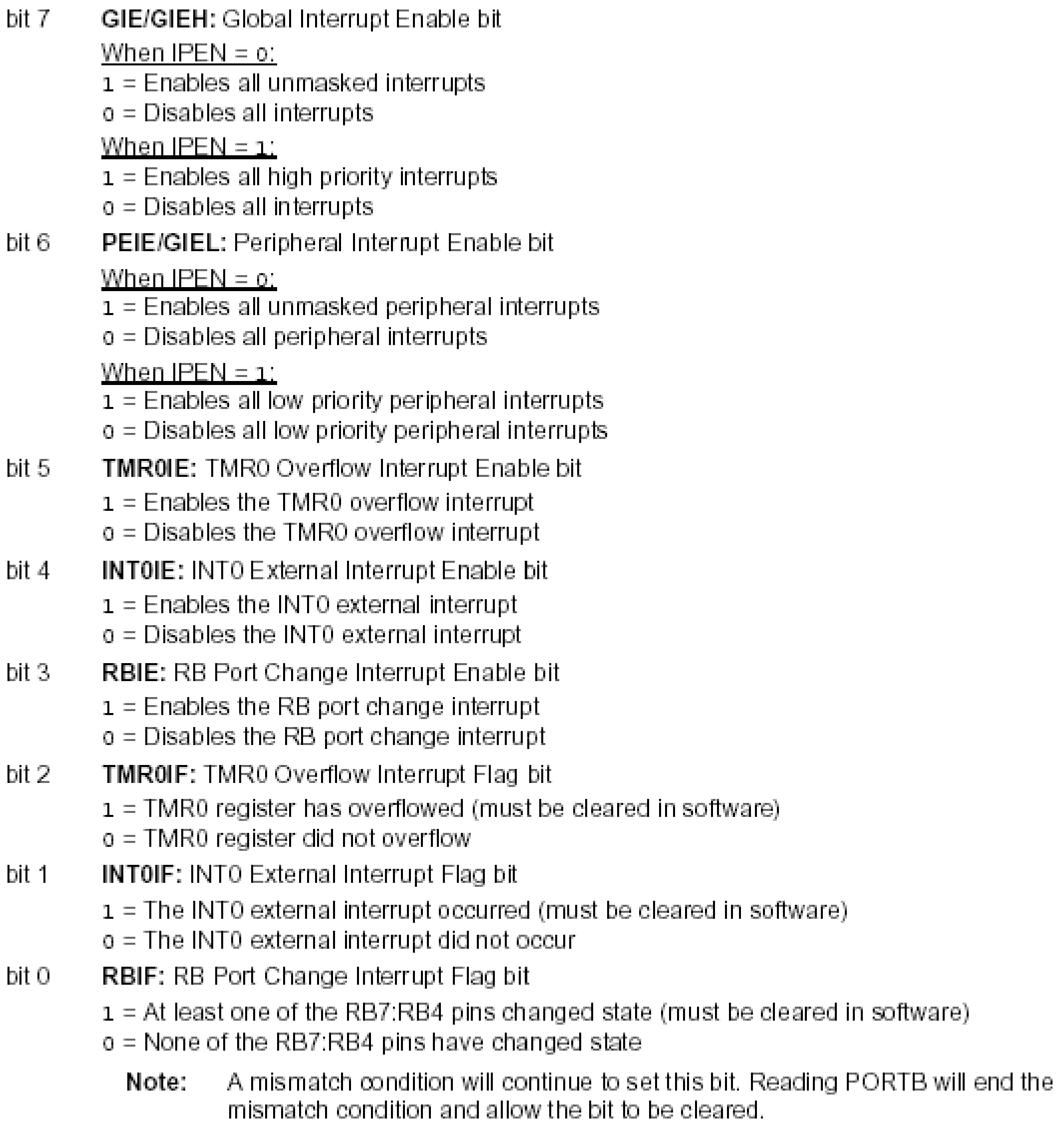
bit 1 **$\overline{\text{POR}}$:** Power-on Reset Status bit
1 = A Power-on Reset has not occurred (set by firmware only)
0 = A Power-on Reset occurred (must be set in software after a Power-on Reset occurs)

bit 0 **$\overline{\text{BOR}}$:** Brown-out Reset Status bit
1 = A Brown-out Reset has not occurred (set by firmware only)
0 = A Brown-out Reset occurred (must be set in software after a Brown-out Reset occurs)

- INTCON Register

Bit 7 Bit0

GIE/ GIEH	PEIE/ GIEL	TMR0 IE	INT0 IE	RBIE	TMR0 IF	INT0 IF	RB IF	INTCON



bit 7 **GIE/GIEH:** Global Interrupt Enable bit

When IPEN = 0:
1 = Enables all unmasked interrupts
0 = Disables all interrupts

When IPEN = 1:
1 = Enables all high priority interrupts
0 = Disables all interrupts

bit 6 **PEIE/GIEL:** Peripheral Interrupt Enable bit

When IPEN = 0:
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts

When IPEN = 1:
1 = Enables all low priority peripheral interrupts
0 = Disables all low priority peripheral interrupts

bit 5 **TMR0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE:** INT0 External Interrupt Enable bit
1 = Enables the INT0 external interrupt
0 = Disables the INT0 external interrupt

bit 3 **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **TMR0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INT0IF:** INT0 External Interrupt Flag bit
1 = The INT0 external interrupt occurred (must be cleared in software)
0 = The INT0 external interrupt did not occur

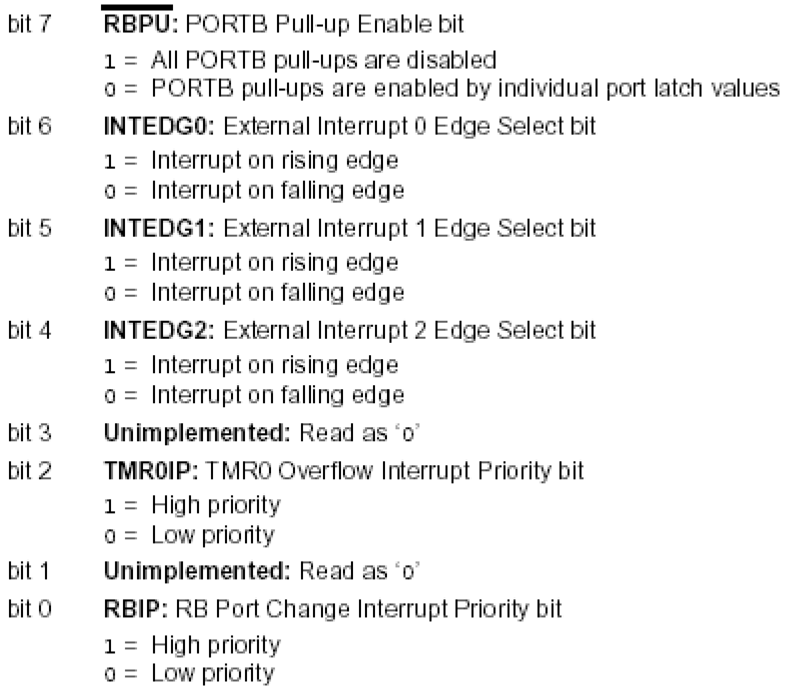
bit 0 **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Note: A mismatch condition will continue to set this bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared.

- INTCON2 Register

Bit 7 Bit0

RBPU’	INTE DG0	INTE DG1	INTE DG2	—	TMR0 IP	—	RBIP	INTCON2



bit 7 **$\overline{\text{RBPU}}$:** PORTB Pull-up Enable bit
1 = All PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG0:** External Interrupt 0 Edge Select bit
1 = Interrupt on rising edge
0 = Interrupt on falling edge

bit 5 **INTEDG1:** External Interrupt 1 Edge Select bit
1 = Interrupt on rising edge
0 = Interrupt on falling edge

bit 4 **INTEDG2:** External Interrupt 2 Edge Select bit
1 = Interrupt on rising edge
0 = Interrupt on falling edge

bit 3 **Unimplemented:** Read as ‘0’

bit 2 **TMR0IP:** TMR0 Overflow Interrupt Priority bit
1 = High priority
0 = Low priority

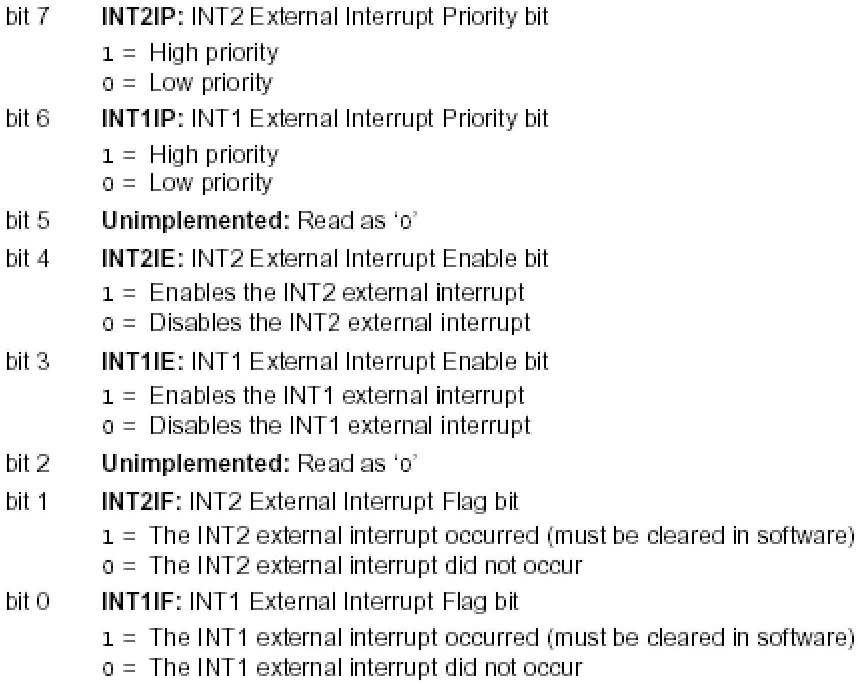
bit 1 **Unimplemented:** Read as ‘0’

bit 0 **RBIP:** RB Port Change Interrupt Priority bit
1 = High priority
0 = Low priority

- INTCON3

Bit 7 Bit0

INT2 IP	INT1 IP	—	INT2 IE	INT1 IE	—	INT2 IF	INT1 IF	INTCON3



bit 7 **INT2IP:** INT2 External Interrupt Priority bit
1 = High priority
0 = Low priority

bit 6 **INT1IP:** INT1 External Interrupt Priority bit
1 = High priority
0 = Low priority

bit 5 **Unimplemented:** Read as '0'

bit 4 **INT2IE:** INT2 External Interrupt Enable bit
1 = Enables the INT2 external interrupt
0 = Disables the INT2 external interrupt

bit 3 **INT1IE:** INT1 External Interrupt Enable bit
1 = Enables the INT1 external interrupt
0 = Disables the INT1 external interrupt

bit 2 **Unimplemented:** Read as '0'

bit 1 **INT2IF:** INT2 External Interrupt Flag bit
1 = The INT2 external interrupt occurred (must be cleared in software)
0 = The INT2 external interrupt did not occur

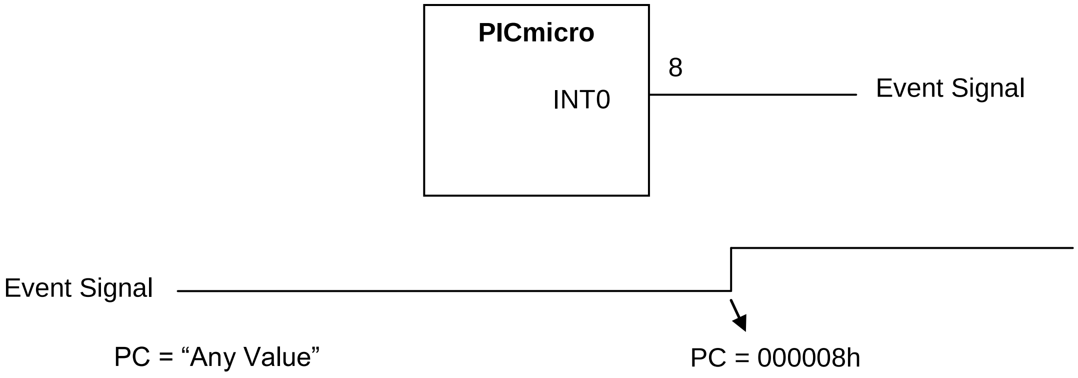
bit 0 **INT1IF:** INT1 External Interrupt Flag bit
1 = The INT1 external interrupt occurred (must be cleared in software)
0 = The INT1 external interrupt did not occur

It is also recommended that the Microchip header files supplied with MPLAB® IDE be used for the symbolic bit names in these and other registers. This allows the assembler/compiler to automatically take care of the placement of these bits within the specified register. For Assembly code, use the following statement at the top of your assembly code to include all SFR addresses and bit names as specified in the appendix:

```
# include p18f1220.inc
```

There are three external interrupts available on PICmicro (INT0-Pin 8, INT1-Pin9 and INT2-Pin 17 on the PDIP package), three timers capable of generating interrupt and others to be discussed in the future.

Below is an example of connecting interrupt INT0 to Event Signal. Anytime Event Signal goes from low to high, a high priority interrupt is caused and PC is set to 000008h.



In general, each interrupt source has three bits to control its operation. The functions of these bits are:

- Flag bit to indicate that an interrupt event occurred.
- Enable bit that allows program execution to branch to the interrupt vector address when the flag bit is set.
- Priority bit to select high priority or low priority (INT0 has no priority bit and is always high priority)

The following 6 registers are used to configure Interrupt Enable, Flag and Priority:

- IPR1, PIE1, PIR1

Bit 7							Bit0	
—	ADIP	RCIP	TXIP	—	CCP1 IP	TMR2 IP	TMR1 IP	IPR1
—	ADIE	RCIE	TXIE	—	CCP1 IE	TMR2 IE	TMR1 IE	PIE1
—	ADIF	RCIF	TXIF	—	CCP1 IF	TMR2 IF	TMR1 IF	PIR1

- IPR2, PIE2, PIR2

Bit 7							Bit0	
OSCF IP	—	—	EEIP	—	LVD IP	TMR3 IP	—	IPR2
OSCF IE	—	—	EEIE	—	LVD IE	TMR3 IE	—	PIE2
OSCF IF	—	—	EEIF	—	LVD IF	TMR3 IF	—	PIR2

The interrupt priority feature is enabled by setting the IPEN bit (RCON<7>). When interrupt priority is enabled, there are two bits which enable interrupts globally. Setting the GIEH bit (INTCON<7>) enables all interrupts that have the priority bit set (high priority). Setting the GIEL bit (INTCON<6>) enables all interrupts that have the priority bit cleared (low priority). When the interrupt flag, enable bit and appropriate global interrupt enable bit are set, the interrupt will vector immediately to address 000008h or 000018h, depending on the priority bit setting. Individual interrupts can be disabled through their corresponding enable bits.

When the IPEN bit is cleared (default state), the interrupt priority feature is disabled and interrupts are compatible with PICmicro mid-range devices. In Compatibility mode, the interrupt priority bits for each source have no effect. INTCON<6> is the PEIE bit, which enables/disables all peripheral interrupt sources. INTCON<7> is the GIE bit, which enables/disables all interrupt sources. All interrupts branch to address 000008h in Compatibility mode.

When an interrupt is responded to, the global interrupt enable bit is cleared to disable further interrupts. If the IPEN bit is cleared, this is the GIE bit. If interrupt priority levels are used, this will be either the GIEH or GIEL bit. High priority interrupt sources can interrupt a low priority interrupt. Low priority interrupts are not processed while high priority interrupts are in progress.

The return address is pushed onto the stack and the PC is loaded with the interrupt vector address (000008h or 000018h). Once in the Interrupt Service Routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits. The interrupt flag bits must be cleared in software before re-enabling interrupts to avoid recursive interrupts.

The “return from interrupt” instruction, RETFIE, exits the interrupt routine and sets the GIE bit (GIEH or GIEL, if priority levels are used), which re-enables interrupts.

For external interrupt events, such as the INT pins or the PORTB input change interrupt, the interrupt latency will be three to four instruction cycles. The exact latency is the same for one or two-cycle instructions. Individual interrupt flag bits are set, regardless of the status of their corresponding enable bit or the GIE bit.

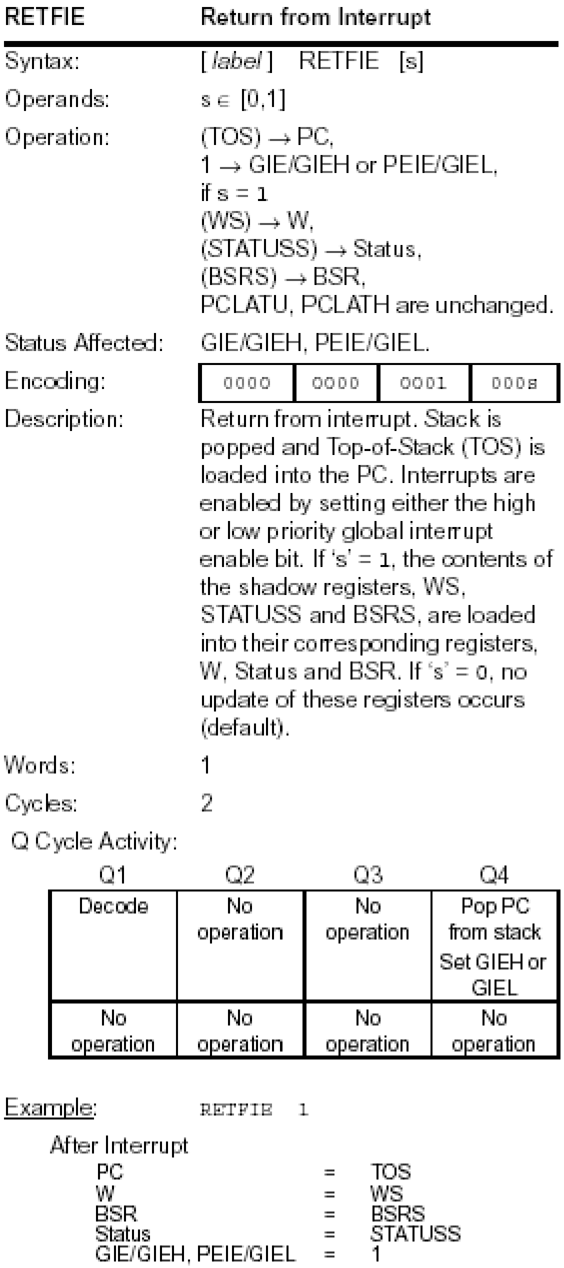
Note: Do not use the MOVFF instruction to modify any of the interrupt control registers while any interrupt is enabled. Doing so may cause erratic microcontroller behavior.

❖ Returning from interrupt handling code

At the time of interrupt, the value PC+2 (pointer to the next instruction) is pushed on the stack. Once the interrupt handling code has finished, it can return to the instruction the program was at before the interrupt occurred by popping the stack and using the top of stack value as the PC.

The instruction RETFIE, when executed, will automatically enable all interrupts and return the program back to the location of the next instruction before the interrupt.

❖ RETFIE Instruction



RETFIE **Return from Interrupt**

Syntax:	[*label*] RETFIE [s]
Operands:	s ∈ [0,1]
Operation:	(TOS) → PC, 1 → GIE/GIEH or PEIE/GIEL, if s = 1 (WS) → W, (STATUSS) → Status, (BSRS) → BSR, PCLATU, PCLATH are unchanged.
Status Affected:	GIE/GIEH, PEIE/GIEL.
Encoding:	0000 0000 0001 000s
Description:	Return from interrupt. Stack is popped and Top-of-Stack (TOS) is loaded into the PC. Interrupts are enabled by setting either the high or low priority global interrupt enable bit. If 's' = 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, W, Status and BSR. If 's' = 0, no update of these registers occurs (default).
Words:	1
Cycles:	2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	Pop PC from stack Set GIEH or GIEL
No operation	No operation	No operation	No operation

Example: RETFIE 1

After Interrupt

PC	=	TOS
W	=	WS
BSR	=	BSRS
Status	=	STATUSS
GIE/GIEH, PEIE/GIEL	=	1

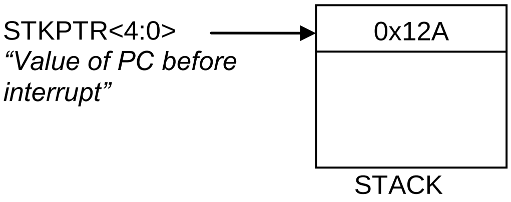
Notes:

- Example – High priority interrupt and return code.

  Solution:

Address	Content .
0x008	MOVLW 23
0x00A	ADDWF 0x90, 1, 0
0x00C	CLRF 0x89
0x00E	RETFIE
…	
0x126	MOVLW 23
0x128	ADDWF 0x90, 1, 0
0x12A	CLRF 0x89

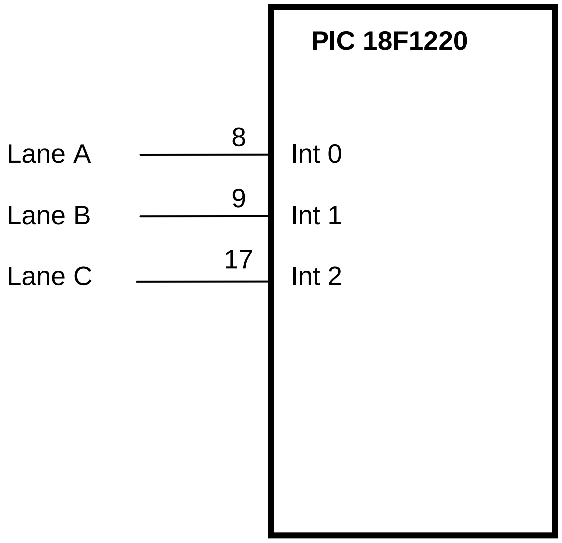
A high Priority Interrupt occurs when instruction at location 0x128 is being executed, so PC+2 is equal to 0x12A.



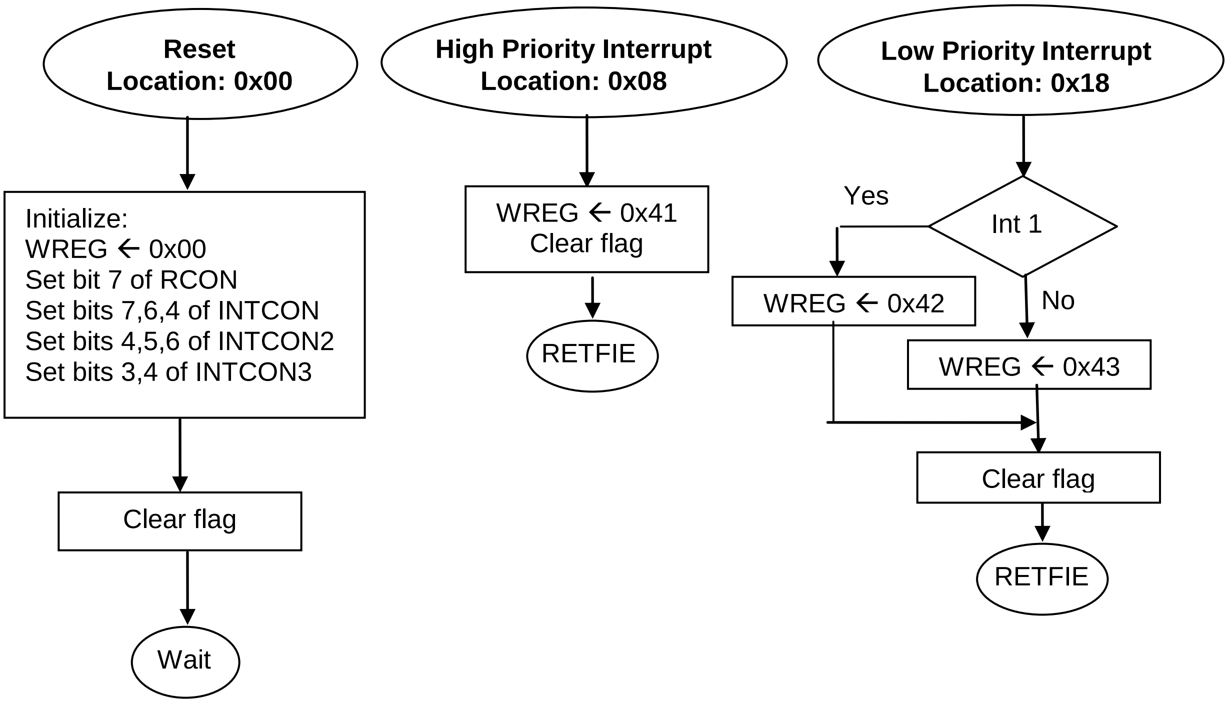
- Example – Event Handling using Interrupts
  Write an interrupt handling code to implement a 3-way intersection traffic light controller. Inputs are Lane A (highest priority), Lane B, and Lane C (lowest priority) where "1" indicates presence of a car in the lane... Set WREG to ASCII A (0x41), ASCII B (0x42) or ASCII C (0x43) indicating the highest Priority Lane that is occupied.

  Solutions:

  Partial Schematic



Flow Chart for Reset, Int0, Int 1 and Int 2 handling



Sample code for Reset, Int0, Int 1 and Int 2 handling

Refer to Lab documentation.

4.5. Clock and Oscillator

PICmicro is much more flexible than typical microprocessors when it comes to selecting the system clock. It provides over 10 different options. Most processors have a range of speed, and an external oscillator will be required for proper operation and generation of system clock. PICmicro allows for external crystal, RC or internal oscillators.

PICmicro's internal oscillator block can generate two different clock signals; either one can be used as the system clock.

The main output (INTOSC) is an 8 MHz clock source, which can be used to directly drive the system clock. It also drives a post scalar, which can provide a range of clock frequencies from 125 kHz to 4 MHz. The INTOSC output is enabled when a system clock frequency from 125 kHz to 8 MHz is selected.

The other clock source is the internal RC oscillator (INTRC), which provides a 31 kHz output. The INTRC oscillator is enabled by selecting the internal oscillator block as the system clock source, or when one of the following is enabled: Power-up Timer, Fail-Safe Clock Monitor, Timer or Two-Speed Start-up.

❖ INTIO Modes

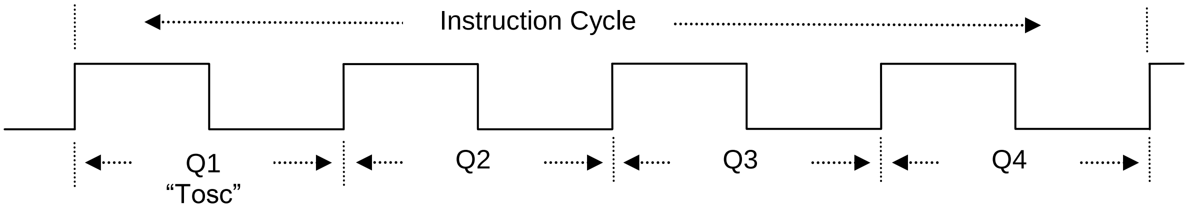
Using the internal oscillator as the clock source can eliminate the need for up to two external oscillator pins, which can then be used for digital I/O. Two distinct configurations are available:

➢ In INTIO1 mode (Default Setting), the OSC2 pin outputs FOSC/4, while OSC1 functions as RA7 for digital input and output.

➢ In INTIO2 mode, OSC1 functions as RA7 and OSC2 functions as RA6, both for digital input and output.

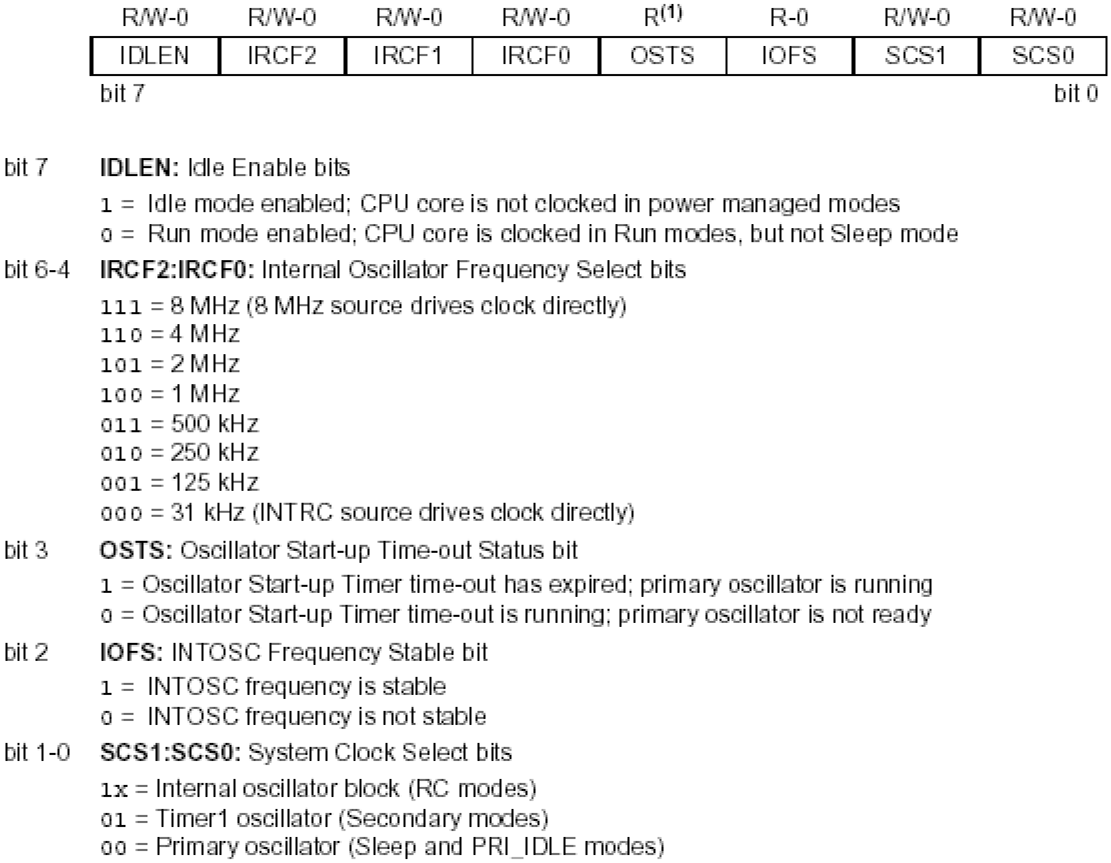
Default oscillator setting, INTIO1 mode, will be assumed throughout the remainder of this document. In this mode, the internal oscillator is used as the system clock. Additionally, the clock (FOSC/4) is accessible via OSC2 pin (pin# 15 on PDIP package). It is important to note that this pin will not be available for other uses such as RA6.

Another point to consider is that each instruction cycle is made up of 4 system clock or Oscillator cycles (Tosc) as shown below:



As discussed earlier, the internal clock frequency is set at 31 kHz which means each clock period is Tosc = 1/f = 32 µsec. Therefore, an instruction cycle is 4*Tosc = 128 µsec.

- OSCCON Register
  This SFR register is used to configure the oscillator and the system clock.



R/W-0	R/W-0	R/W-0	R/W-0	R[1]	R-0	R/W-0	R/W-0
IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0
bit 7							bit 0

bit 7 **IDLEN:** Idle Enable bits
1 = Idle mode enabled; CPU core is not clocked in power managed modes
0 = Run mode enabled; CPU core is clocked in Run modes, but not Sleep mode

bit 6-4 **IRCF2:IRCF0:** Internal Oscillator Frequency Select bits
111 = 8 MHz (8 MHz source drives clock directly)
110 = 4 MHz
101 = 2 MHz
100 = 1 MHz
011 = 500 kHz
010 = 250 kHz
001 = 125 kHz
000 = 31 kHz (INTRC source drives clock directly)

bit 3 **OSTS:** Oscillator Start-up Time-out Status bit
1 = Oscillator Start-up Timer time-out has expired; primary oscillator is running
0 = Oscillator Start-up Timer time-out is running; primary oscillator is not ready

bit 2 **IOFS:** INTOSC Frequency Stable bit
1 = INTOSC frequency is stable
0 = INTOSC frequency is not stable

bit 1-0 **SCS1:SCS0:** System Clock Select bits
1x = Internal oscillator block (RC modes)
01 = Timer1 oscillator (Secondary modes)
00 = Primary oscillator (Sleep and PRI_IDLE modes)

- Example - PIC micro is running with a 32 µsec internal clock. How would it take to execute “CALL Delay”? Given:

```
Delay:
        CLRF    Wreg
intL:   INCF    Wreg
        BNZ     intL
        RETURN
```

Solution:

```
                                    Cycle
Delay:
        CLRF    Wreg                1
intL:   INCF    Wreg                1
        BNZ     intL                2 when branching, 1 otherwise
        RETURN                      2
```

Total Time = { (4*32) * [1 + 255*(1+2) + (1+1) + 2] } = 98,560 usec → Approx 0.1 Sec.

- Example - PIC micro is running with a 32 µsec internal clock. How long would it take to execute the following function?

```
Delay:  MOVLW    0x00
        MOVWF    0x80
 Loop:  MOVFF    PORTB, PORTA
        NOP
        INCF     0x80
        BNC      Loop
        RETURN
```

Solution:

4.6. Timers

Timers allow the designer to set a duration which, at its end, the timer will set a flag and cause an interrupt if configured. PICmicro has four timer modules (Timer0 through Timer3). Each with a set of unique features which are outlined in the next few pages.

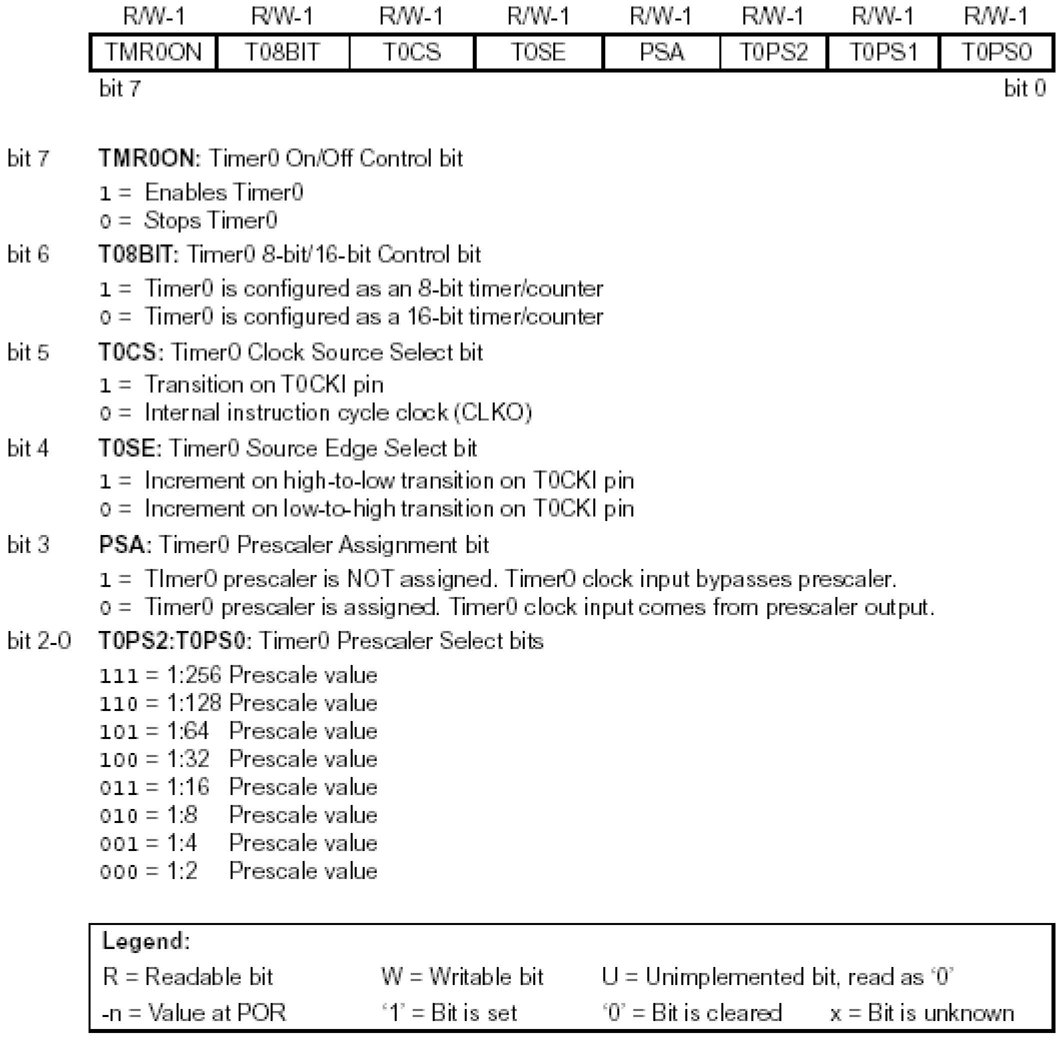
Timers may be configured to generate either low or high priority interrupt. In general, configuring a timer is a three-step process:

1) Configure the interrupt registers
2) Set the value of Timer’s Low and High registers
3) Configure the timer control registered.

The following pages outline each timer module and associated registers.

- Timer0 Module
  Timer0 may be set to 8-bit or 16-bit mode. In 8-bit mode, interrupt is generated on overflow from FFh to 00h. In 16-bit mode, interrupt is generated on overflow from FFFFh to 0000h.

  T0CON register controls all aspects of Timer0. T0CON is readable and writable.



R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0
bit 7							bit 0

bit 7 **TMR0ON:** Timer0 On/Off Control bit
1 = Enables Timer0
0 = Stops Timer0

bit 6 **T08BIT:** Timer0 8-bit/16-bit Control bit
1 = Timer0 is configured as an 8-bit timer/counter
0 = Timer0 is configured as a 16-bit timer/counter

bit 5 **T0CS:** Timer0 Clock Source Select bit
1 = Transition on T0CKI pin
0 = Internal instruction cycle clock (CLKO)

bit 4 **T0SE:** Timer0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

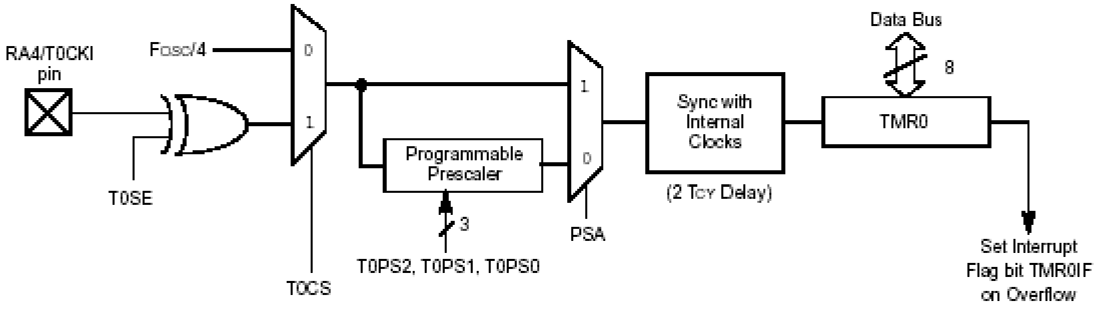
bit 3 **PSA:** Timer0 Prescaler Assignment bit
1 = TImer0 prescaler is NOT assigned. Timer0 clock input bypasses prescaler.
0 = Timer0 prescaler is assigned. Timer0 clock input comes from prescaler output.

bit 2-0 **T0PS2:T0PS0:** Timer0 Prescaler Select bits
111 = 1:256 Prescale value
110 = 1:128 Prescale value
101 = 1:64 Prescale value
100 = 1:32 Prescale value
011 = 1:16 Prescale value
010 = 1:8 Prescale value
001 = 1:4 Prescale value
000 = 1:2 Prescale value

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

Timer0 Block Diagram in 8-bit:



Note: Upon Reset, Timer0 is enabled in 8-bit mode with clock input from T0CKI maximum prescale.

It is important to write TMRnH before writing TMRnL when configuring Timers in 16-bit mode

- Example - In a PICmirco system, TIMER0 is set to 8-bit mode with pre-scale 1:256 using internal RC clock. What values of T0CON, TMR0L & TMR0H results in approximately one second to next TIMER 0 interrupt?

  Solution:

  T0CON = “110x 0111”

  Time/Count = 32 µsec/ Tosc X 4 cycle/Tins X 256 = 32,768 µsec

  # counts for one second = 1,000,000 µsec / 32,768 µsec = 30.5 → 31 counts

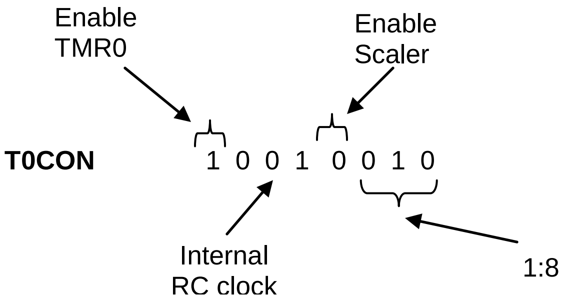
  TMR0L = (256 – 31) = 225 → “1110 0001”
  TMR0H → “xxxx xxxx”

  - Extension – How would the value of T0CON and TMR0 change if we want to use TIMER 0 in 16-bit mode.

    Solutions:

- Example – In a PICmicro system, register T0CON is set to 0x92, TMR0H is set to 0xFF and TMR0L is set to 0xF0. How long in seconds before Timer 0 interrupt occurs?

  Solution:



  Timer is set to “0xFFF0” → 16 count to interrupt
  Which means that the values of high timer 0 register (TMR0H) is set to 0xFF and low timer 0 register (TMR0L) is set to 0xF0.

  Each Count = 32 µsec/Tosc X 4 cycle/Tins X 8 = 1024 µsec.

  Duration to next interrupt = 16 count x 1,024 µsec / count = 16,384 µsec. = 0.016384 seconds

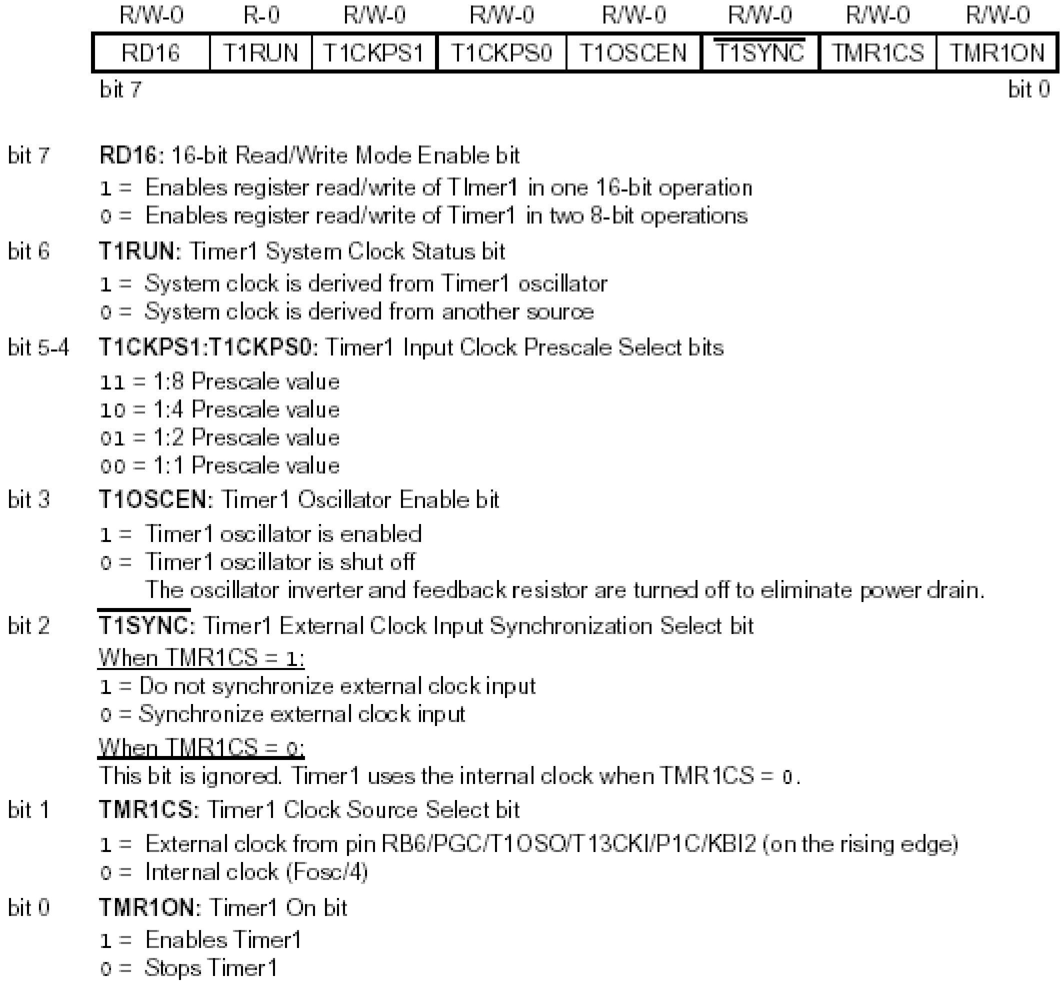
- Example –Given that T0CON is set to “0x87”, how long does it take to increment TMR0 by 1 (a single tick) and what value should be loaded in TMR0L and TMR0H if interrupt is expected in 640 msec?

  Solutions:

- Given that Timer0 is enabled and T0CON is set to “10010100”, TMR0L is set to “10000000” and TMR0H is set to “11110000”, Find:
  a) How long does each tick (TMR0L increment) takes in seconds?
  b) How many ticks before TMR0 interrupts?
  c) How many seconds before TMR0 Interrupt?

- Timer1 Module
  Timer1 is a 16-bit timer/count using two 8-bit registers (TMR1H and TMR1L). Both registers are readable and writable.

  T1CON register controls all aspects of Timer1. T1CON is readable and writable.



R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7 **RD16:** 16-bit Read/Write Mode Enable bit
1 = Enables register read/write of TImer1 in one 16-bit operation
0 = Enables register read/write of Timer1 in two 8-bit operations

bit 6 **T1RUN:** Timer1 System Clock Status bit
1 = System clock is derived from Timer1 oscillator
0 = System clock is derived from another source

bit 5-4 **T1CKPS1:T1CKPS0:** Timer1 Input Clock Prescale Select bits
11 = 1:8 Prescale value
10 = 1:4 Prescale value
01 = 1:2 Prescale value
00 = 1:1 Prescale value

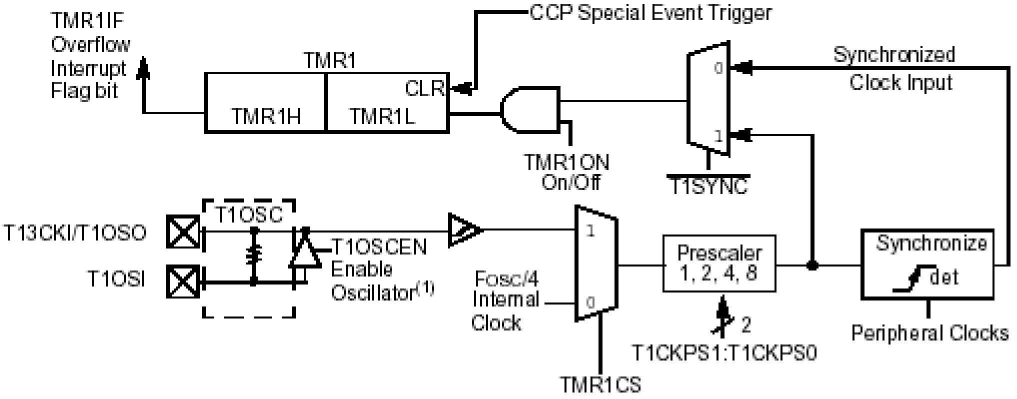
bit 3 **T1OSCEN:** Timer1 Oscillator Enable bit
1 = Timer1 oscillator is enabled
0 = Timer1 oscillator is shut off
The oscillator inverter and feedback resistor are turned off to eliminate power drain.

bit 2 **$\overline{\text{T1SYNC}}$:** Timer1 External Clock Input Synchronization Select bit
When TMR1CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input
When TMR1CS = 0:
This bit is ignored. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS:** Timer1 Clock Source Select bit
1 = External clock from pin RB6/PGC/T1OSO/T13CKI/P1C/KBI2 (on the rising edge)
0 = Internal clock (Fosc/4)

bit 0 **TMR1ON:** Timer1 On bit
1 = Enables Timer1
0 = Stops Timer1

Timer1 Block Diagram:



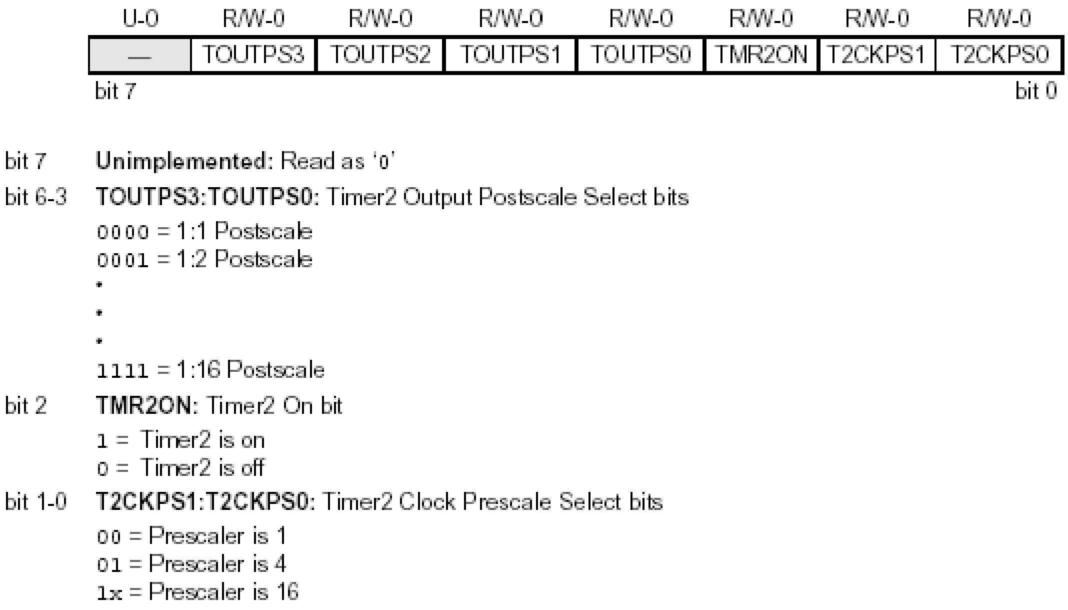
- Timer2 Module

Timer2 can be used as the Pulse Width Modulator (PWM) time base in the PWM mode of the CCP module. The TMR2 register is readable and writable and is cleared by any device Reset. The input clock (FOSC/4) has pre-scale options of 1:1, 1:4 or 1:16, selected by control bits, T2CKPS1:T2CKPS0 (T2CON<1:0>). Additionally, there are post scalar options of 1:1 to 1:16 selected by bits, TOUTPS3:TOUTPS0 (T2CON<6:3>), that are applied to input clock in order to increment TMR2 register content. Once TMR2 register reaches its maximum count, a Timer 2 interrupt (latched in flag bit, TMR2IF (PIR1<1>)) is generated..

The pre-scalar and post-scalar counters are cleared when any of the following occurs:

- A write to the TMR2 register
- A write to the T2CON register
- Any device Reset (Power-on Reset, MCLR Reset, Watchdog Timer Reset or Brown-out Reset)

TMR2 is not cleared when T2CON is written. T2CON is the Timer2 Control Register and is described below:



U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7							bit 0

bit 7 **Unimplemented:** Read as '0'

bit 6-3 **TOUTPS3:TOUTPS0:** Timer2 Output Postscale Select bits
0000 = 1:1 Postscale
0001 = 1:2 Postscale
•
•
•
1111 = 1:16 Postscale

bit 2 **TMR2ON:** Timer2 On bit
1 = Timer2 is on
0 = Timer2 is off

bit 1-0 **T2CKPS1:T2CKPS0:** Timer2 Clock Prescale Select bits
00 = Prescaler is 1
01 = Prescaler is 4
1x = Prescaler is 16

- Example – Timers

T2CON is set to 45 hex. and is using internal RC clock.

a) How long does it take for a single timer tick, or to increment TMR2 by 1?
b) What value TMR2 (Low & High bytes) will cause Timer 2 interrupt after 60 msec?

Solution:

- Timer3 Module
  Timer3 is a 16-bit timer/counter using two 8-bit registers (TMR3H and TMR3L). Both registers are readable and writable.

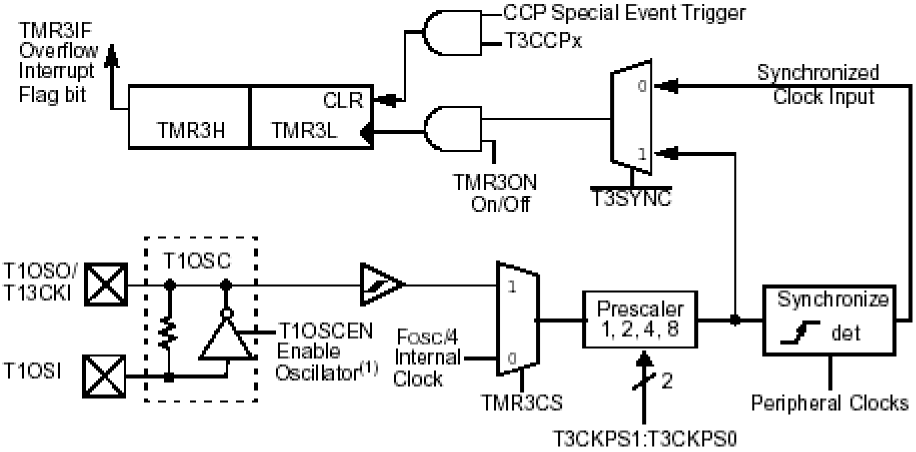
  Timer3 can operate in one of these modes:
  - As a timer
  - As a synchronous counter
  - As an asynchronous counter

  The operating mode is determined by the clock select bit, TMR3CS (T3CON<1>).

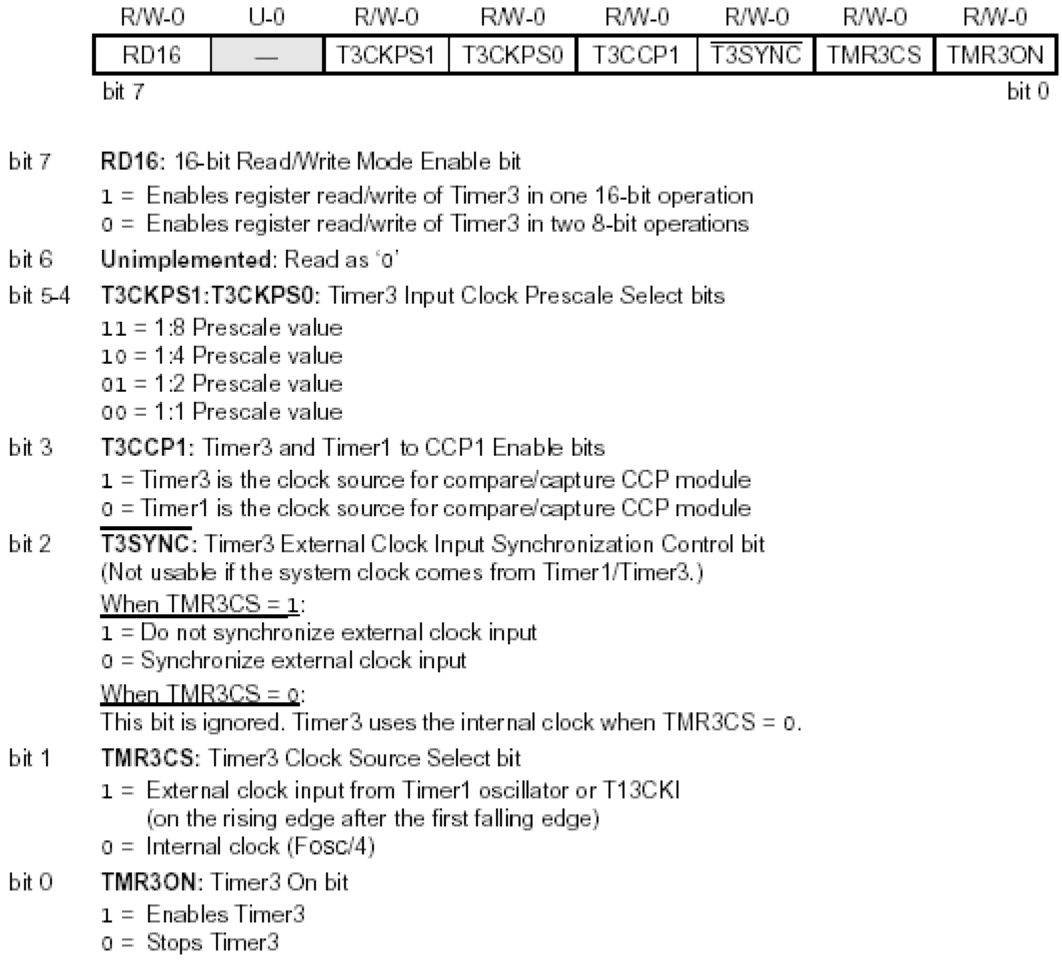
  When TMR3CS = 0, Timer3 increments every instruction cycle. When TMR3CS = 1, Timer3 increments on every rising edge of the Timer1 external clock input or the Timer1 oscillator, if enabled.

  When the Timer1 oscillator is enabled (T1OSCEN is set), the RB7/PGD/T1OSI/P1D/KBI3 and RB6/PGC/ T1OSO/T13CKI/P1C/KBI2 pins become inputs. That is, the TRISB7:TRISB6 value is ignored and the pins are read as ‘0’.

  Below is a block diagram of Timer3:



T3CON register controls all aspects of Timer3. T3CON is readable and writable.



R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	—	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON
bit 7							bit 0

bit 7 **RD16:** 16-bit Read/Write Mode Enable bit

1 = Enables register read/write of Timer3 in one 16-bit operation
0 = Enables register read/write of Timer3 in two 8-bit operations

bit 6 **Unimplemented**: Read as '0'

bit 5-4 **T3CKPS1:T3CKPS0:** Timer3 Input Clock Prescale Select bits

11 = 1:8 Prescale value
10 = 1:4 Prescale value
01 = 1:2 Prescale value
00 = 1:1 Prescale value

bit 3 **T3CCP1:** Timer3 and Timer1 to CCP1 Enable bits

1 = Timer3 is the clock source for compare/capture CCP module
0 = Timer1 is the clock source for compare/capture CCP module

bit 2 **$\overline{\text{T3SYNC}}$:** Timer3 External Clock Input Synchronization Control bit
(Not usable if the system clock comes from Timer1/Timer3.)

When TMR3CS = 1:
1 = Do not synchronize external clock input
0 = Synchronize external clock input

When TMR3CS = 0:
This bit is ignored. Timer3 uses the internal clock when TMR3CS = 0.

bit 1 **TMR3CS:** Timer3 Clock Source Select bit

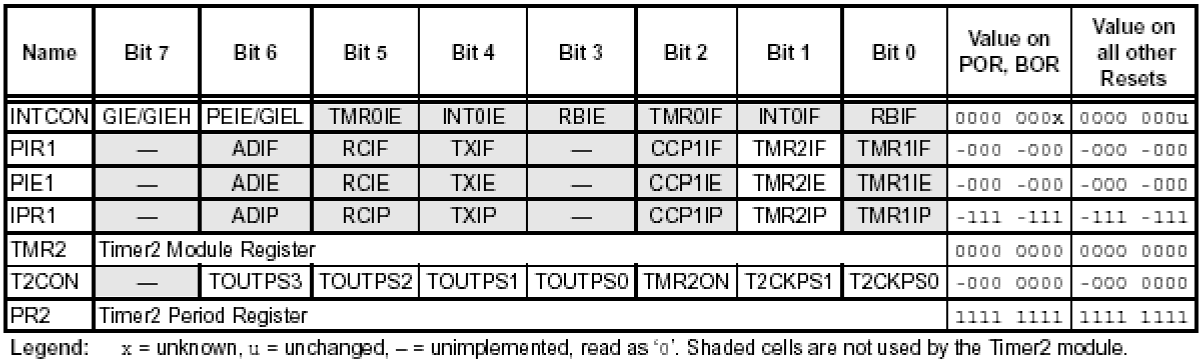
1 = External clock input from Timer1 oscillator or T13CKI
(on the rising edge after the first falling edge)
0 = Internal clock (Fosc/4)

bit 0 **TMR3ON:** Timer3 On bit

1 = Enables Timer3
0 = Stops Timer3

- Other Timer Related Registers

  Other Registers that affect the performance of Timers are shown below (See Interrupt Section for more detail):



Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x	0000 000u
PIR1	—	ADIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF	-000 -000	-000 -000
PIE1	—	ADIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE	-000 -000	-000 -000
IPR1	—	ADIP	RCIP	TXIP	—	CCP1IP	TMR2IP	TMR1IP	-111 -111	-111 -111
TMR2	Timer2 Module Register								0000 0000	0000 0000
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
PR2	Timer2 Period Register								1111 1111	1111 1111

Legend: x = unknown, u = unchanged, – = unimplemented, read as '0'. Shaded cells are not used by the Timer2 module.

- Example – Timers

  Use Timer 0 to flash an LED once every 2 seconds (1 second on, 1 second off).

  Solution:

- Example – Timers

  Write pseudo code and assembly code to set up Timer 0 to interrupt after 256 msec & upon interrupt, set Wreg to 35 and disable timers. (set Timer 0 for highest possible precision).

  Solution:

4.7. Power Management

In addition to normal operation, PICmicro, like most of today’s processors, has low-power modes to save power. Below are the three categories of power management:

- Sleep mode
- Idle modes
- Run modes

Each of these modes disables or reduces the clock speed for a given portion of the processor circuits in order to reduce power. The Run and Idle modes may use any of the three available clock sources (primary, secondary or INTOSC multiplexer); the Sleep mode does not use a clock source.

4.8. Reset

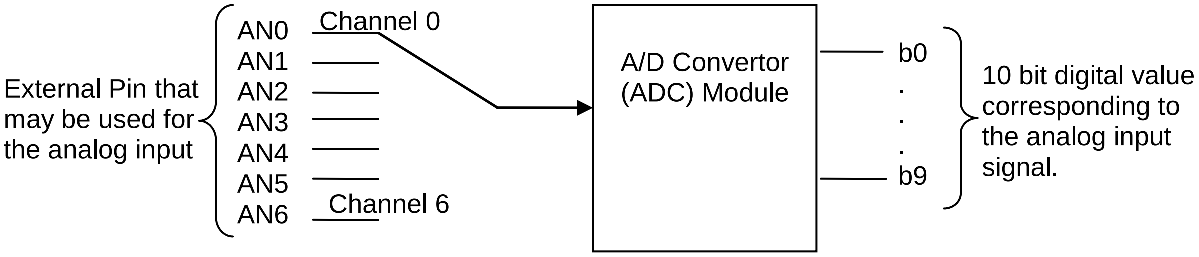
Reset is required to start the processor into a known state. You can use the hardware or software reset to return the processor to a known state. PICmicro handles resets from various sources which are listed below:

- Power-on Reset (POR)
- MCLR Reset during normal operation
- MCLR Reset during Sleep
- Watchdog Timer (WDT) Reset (during execution)
- Programmable Brown-out Reset (BOR)
- RESET Instruction
- Stack Full Reset
- Stack Underflow Reset

Most registers are unaffected by a Reset which means the status of registers are unknown during Power On Reset (POR). Therefore, it is important to initialize registers during the reset handling section of the start up to ensure known starting values. Some registers are forced to a “Reset state”, depending on the type of Reset that occurred.

4.9. Analog-to-Digital Converter

PICmicro has one 10-bit Analog-to-Digital (A/D) converter (ADC) module. The ADC will accept an analog input signal and convert the value of the input signal voltage to a 10-bit number. This functionality allows the user to relate analog signal to a digital value. The following figure is a graphical representation of the ADC operation:


External Pin that may be used for the analog input
AN0
AN1
AN2
AN3
AN4
AN5
AN6
Channel 0
Channel 6
A/D Convertor (ADC) Module
b0
.
.
.
b9
10 bit digital value corresponding to the analog input signal.

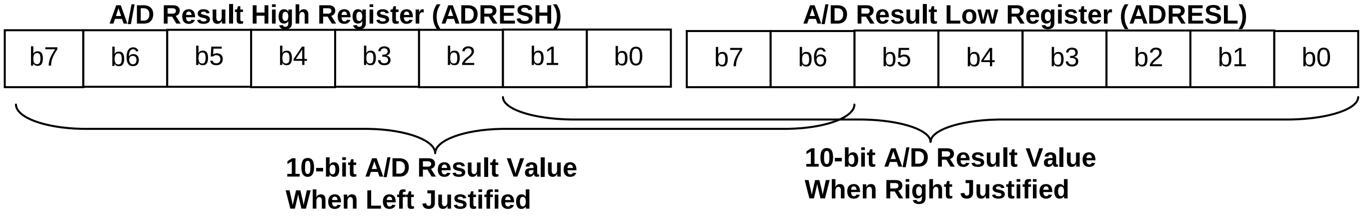
Notice that there is only one ADC, but the user may acquire an analog input signal from one of seven different pins (AN0 through AN6).

PICmicro ADC has a programmable acquisition time which means that the amount of time required to convert from analog to digital value can be changed.

GO/DONE bit (bit 1 of ADCON0) is used to indicate whether ADC is in the process of conversion or it has completed the conversion. So the user has to wait until the conversion is completed as indicated by GO/DONE before reading the value of the results.

The ADC module is controlled and monitored through five SFR registers as shown below:

- A/D Result High Register (ADRESH) & A/D Result Low Register (ADRESL)
  Hold the value resulting from the A/D conversion.


A/D Result High Register (ADRESH)
b7
b6
b5
b4
b3
b2
b1
b0
A/D Result Low Register (ADRESL)
b7
b6
b5
b4
b3
b2
b1
b0
10-bit A/D Result Value When Left Justified
10-bit A/D Result Value When Right Justified

Note that the contents of ADRESH/ADRESL registers are not modified during Power-on Reset which means the contents of ADRESH and ADRESL are unknown after a Power-on Reset. The user has to ensure that an acquisition has been completed before reading the contents of these registers.

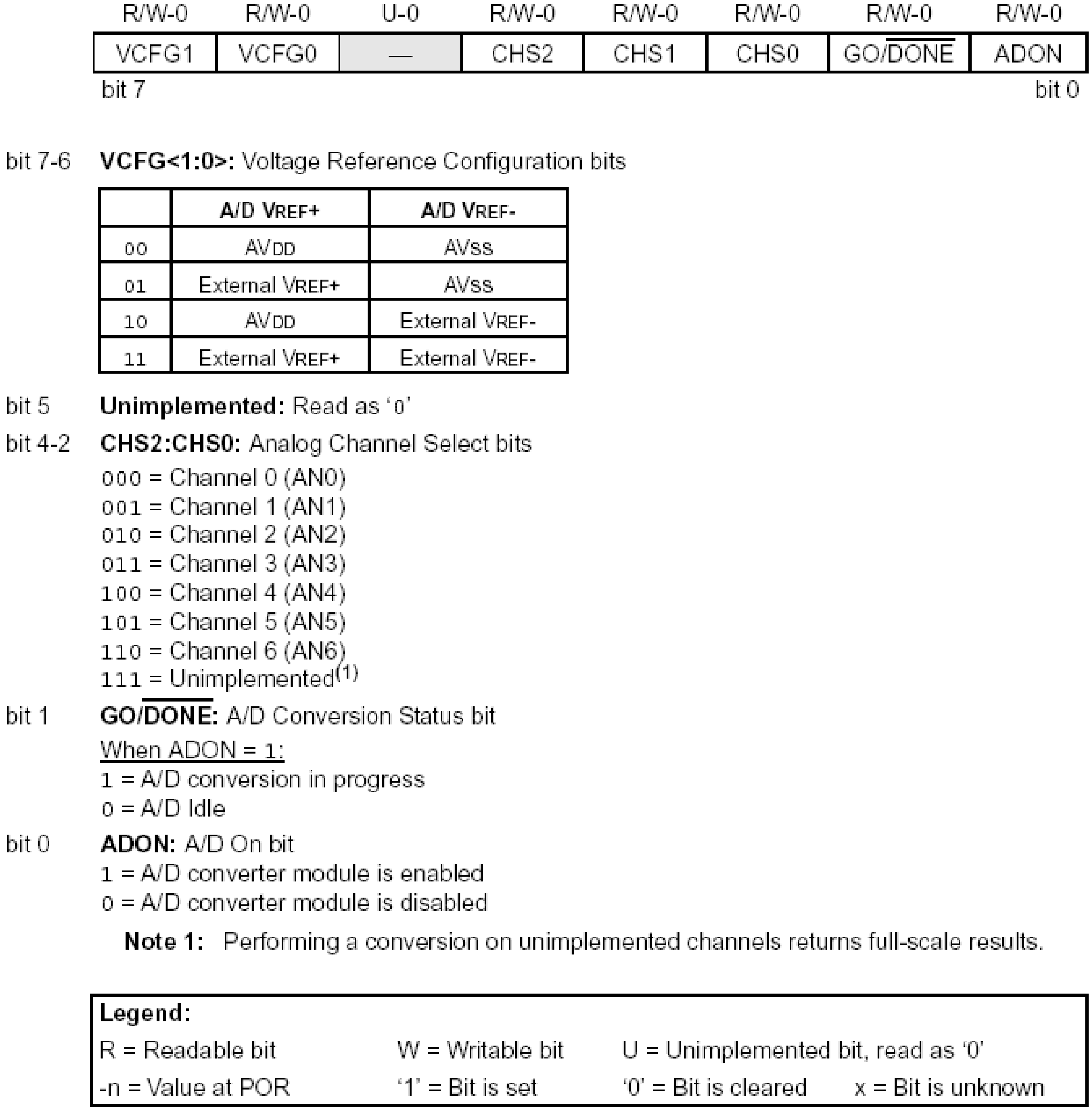
- A/D Control Register 0 (ADCON0)
  ADCON0 controls the operation of the A/D module.

- A/D Control Register 1 (ADCON1)
  ADCON1 configures the functions of the external port pins.

- A/D Control Register 2 (ADCON2)
  ADCON2 configures the A/D clock source, programmed acquisition time and justification.

The following pages offer more detailed descriptions of the three control registers:

- ADCON0 “A/D Control Register 0”



R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
VCFG1	VCFG0	—	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

bit 7-6 **VCFG<1:0>:** Voltage Reference Configuration bits

	A/D VREF+	A/D VREF-
00	AVDD	AVSS
01	External VREF+	AVSS
10	AVDD	External VREF-
11	External VREF+	External VREF-

bit 5 **Unimplemented:** Read as ‘0’

bit 4-2 **CHS2:CHS0:** Analog Channel Select bits
000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Unimplemented[1]

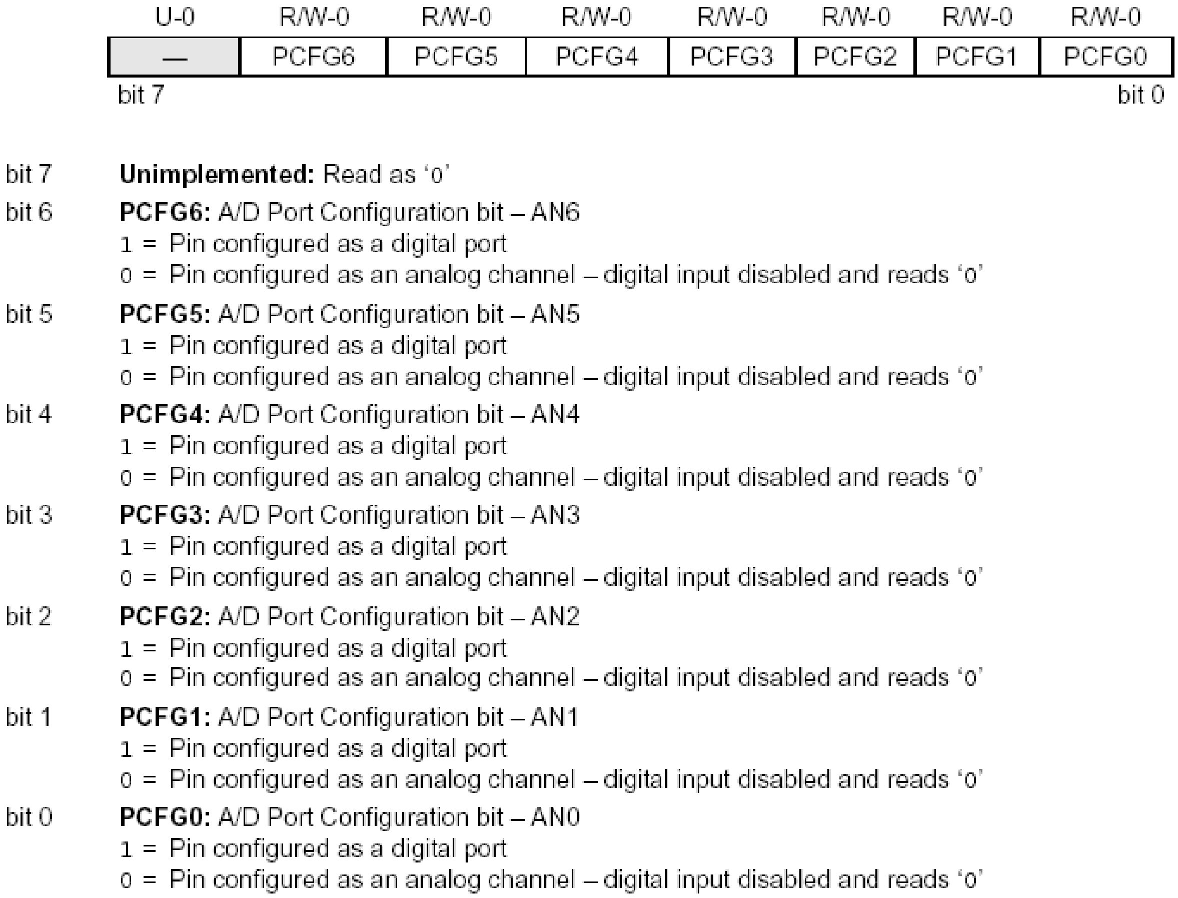
bit 1 **GO/$\overline{\text{DONE}}$:** A/D Conversion Status bit
When ADON = 1:
1 = A/D conversion in progress
0 = A/D Idle

bit 0 **ADON:** A/D On bit
1 = A/D converter module is enabled
0 = A/D converter module is disabled

Note 1: Performing a conversion on unimplemented channels returns full-scale results.

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

- ADCON1 “A/D Control Register 1”



U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **Unimplemented:** Read as ‘0’

bit 6 **PCFG6:** A/D Port Configuration bit – AN6
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

bit 5 **PCFG5:** A/D Port Configuration bit – AN5
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

bit 4 **PCFG4:** A/D Port Configuration bit – AN4
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

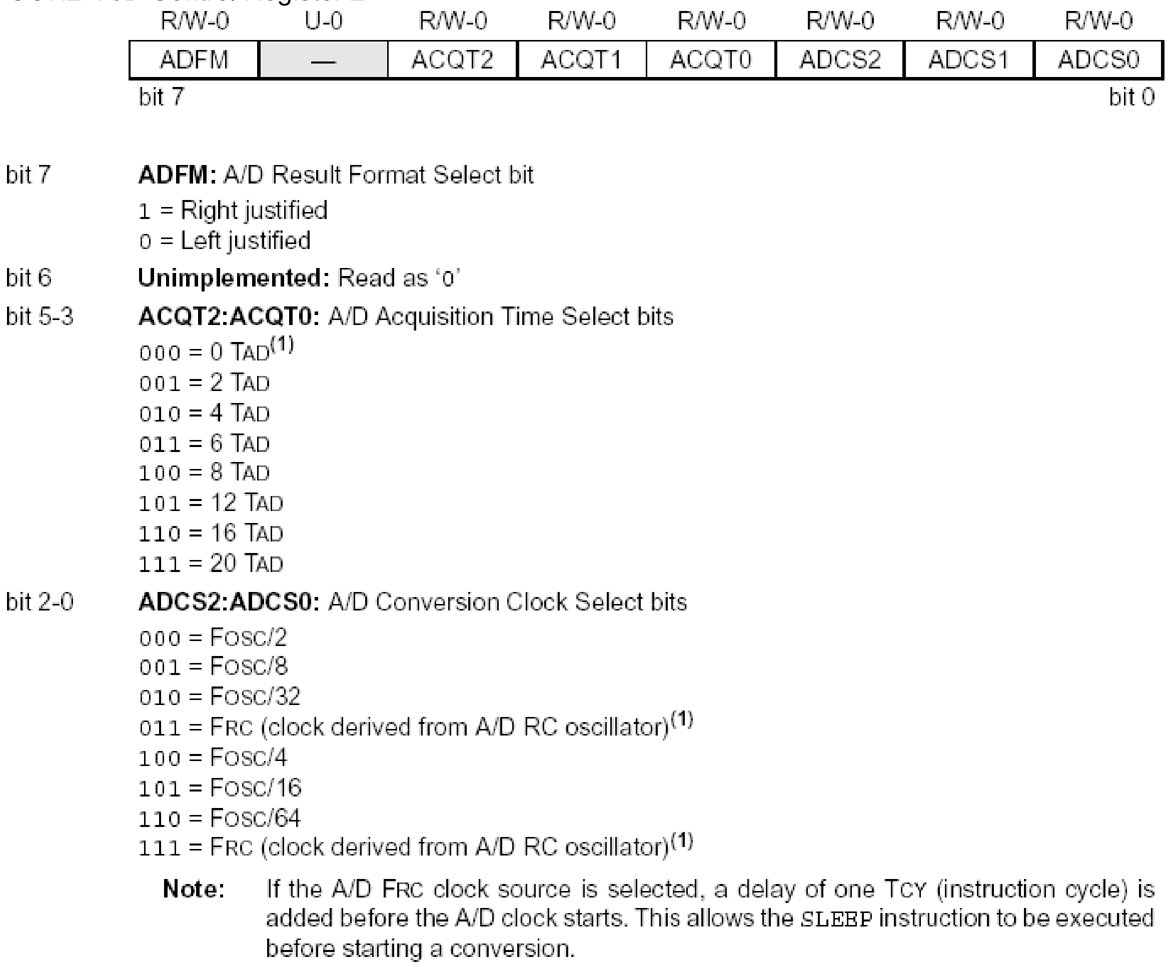
bit 3 **PCFG3:** A/D Port Configuration bit – AN3
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

bit 2 **PCFG2:** A/D Port Configuration bit – AN2
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

bit 1 **PCFG1:** A/D Port Configuration bit – AN1
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

bit 0 **PCFG0:** A/D Port Configuration bit – AN0
1 = Pin configured as a digital port
0 = Pin configured as an analog channel – digital input disabled and reads ‘0’

- ADCON2 “A/D Control Register 2”



R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7							bit 0

bit 7 **ADFM:** A/D Result Format Select bit
1 = Right justified
0 = Left justified

bit 6 **Unimplemented:** Read as ‘0’

bit 5-3 **ACQT2:ACQT0:** A/D Acquisition Time Select bits
000 = 0 TAD[1]
001 = 2 TAD
010 = 4 TAD
011 = 6 TAD
100 = 8 TAD
101 = 12 TAD
110 = 16 TAD
111 = 20 TAD

bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Select bits
000 = FOSC/2
001 = FOSC/8
010 = FOSC/32
011 = FRC (clock derived from A/D RC oscillator)[1]
100 = FOSC/4
101 = FOSC/16
110 = FOSC/64
111 = FRC (clock derived from A/D RC oscillator)[1]

Note: If the A/D FRC clock source is selected, a delay of one TCY (instruction cycle) is added before the A/D clock starts. This allows the SLEEP instruction to be executed before starting a conversion.

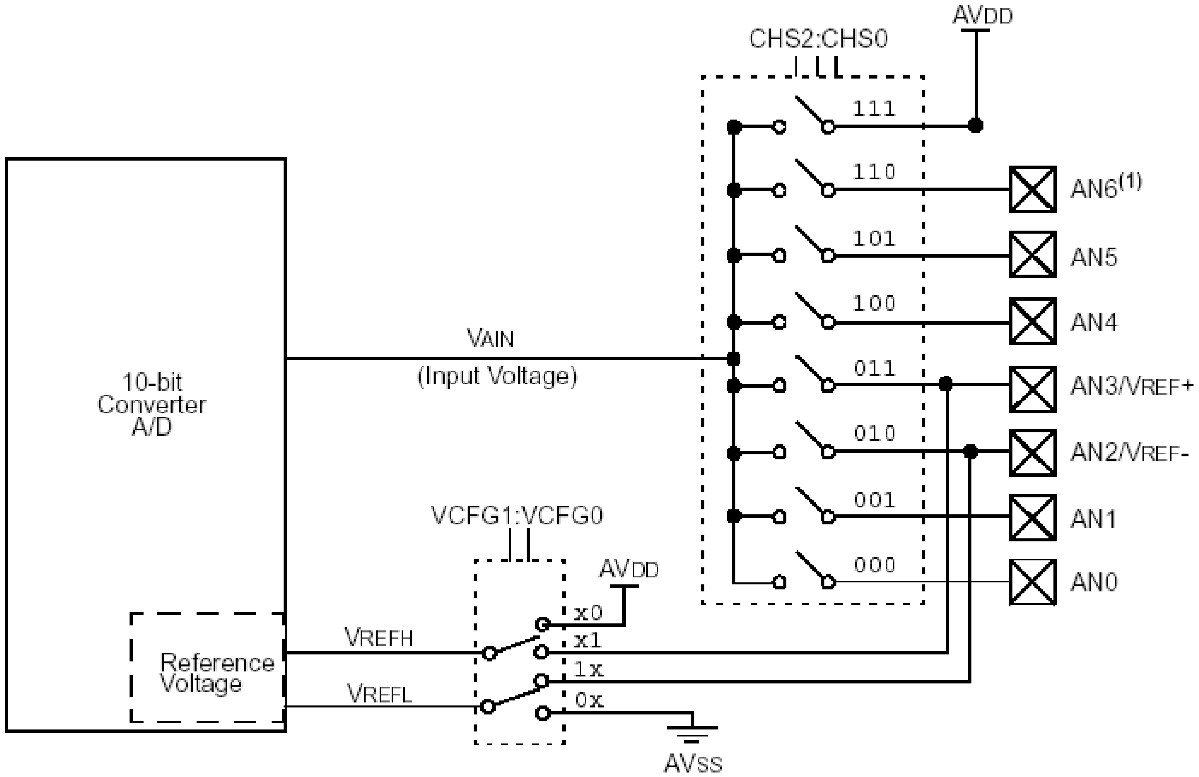
- Configuring ADC Module for Conversion

The analog reference voltage is software selectable. Meaning the user can set the high and low voltage level range to either the supply voltage (AVDD and AVSS), or the voltage levels on the RA3/AN3/VREF+ and RA2/AN2/VREF- pins. See the figure on the next page for a graphical representation.

The ADC has a unique feature of being able to operate while the device is in Sleep mode. To operate in Sleep, the ADC clock must be derived from the ADC’s internal RC oscillator.

ADC module operates by sampling the analog input and holding that value during the conversion time. This is referred to as “sample and hold”. The output of the “sample and hold” is the input to the converter, which generates the digital results by successive approximation. Note that a power-on reset will abort the conversion which means after a power-on reset the digital value in the results registers is not valid.

Each external pin associated with the ADC can be configured as an analog input, or as a digital I/O. The ADRESH and ADRESL registers contain the result of the A/D conversion. When the A/D conversion is complete, the result is loaded into the ADRESH/ADRESL registers, the GO/DONE bit (ADCON0 register) is cleared and A/D Interrupt Flag bit, ADIF, is set. The following figure shows the block diagram of the A/D module:



After the A/D module has been configured as desired, the selected channel must be acquired before the conversion is started. The analog input channels must have their corresponding TRIS bits selected as inputs. After this acquisition time has elapsed, the A/D conversion can be started. An acquisition time can be programmed to occur between setting the GO/DONE’ bit and the actual start of the conversion.

Below are the steps to configure the A/D Converter:

1. Configure the A/D module:
   - Select external pin (channel) that will be used as the analog (use bits 2 to 4 of ADCON0)
   - Configure analog pins as input (Bits 0-6 of TRISA)
   - Set the voltage reference (Bits 6 and 7 of ADCON0)
   - Select A/D acquisition time (Bits 3-5 of ADCON2)
   - Select A/D conversion clock (Bits 0-2 of ADCON2)
   - Turn on A/D module (Bit 0 of ADCON0)

2. Configure A/D interrupts (if desired):
   In registers INTCON, IPR1, PIR1 and PIE1, perform the following modifications:
   - Set GIE bit
   - Clear ADIF bit
   - Set ADIE bit
   - Set ADIP bit

3. Wait the required acquisition time (refer to next section “A/D Acquisition Requirements” for more detail).

4. Start conversion:
   - Set GO/DONE bit (ADCON0 register)

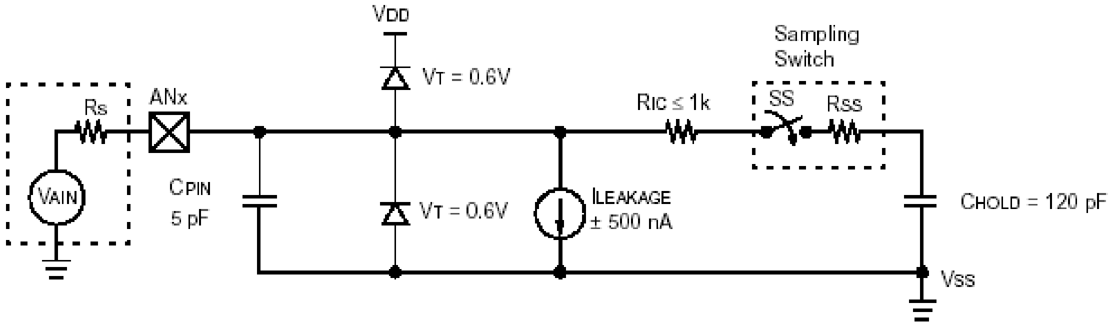
5. Wait for A/D conversion to complete, by either:
   - Polling for the GO/DONE’ bit to be cleared
   - Waiting for the A/D interrupt

6. Read A/D Result registers (ADRESH:ADRESL); clear bit ADIF if required.

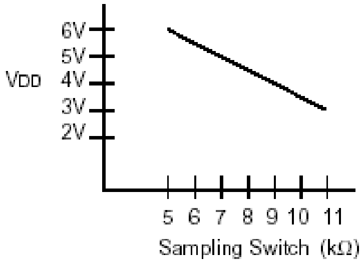
7. For the next conversion, go to step 1 or step 2, as required. The A/D conversion time per bit is defined as TAD. A minimum wait of 2 TAD is required before the next acquisition starts.

➢ A/D Acquisition Requirements

For the A/D converter to meet its specified accuracy, the charge holding capacitor (CHOLD) must be allowed to fully charge to the input channel voltage level. The analog input model is shown below:



Legend:		
	CPIN	= input capacitance
	VT	= threshold voltage
	ILEAKAGE	= leakage current at the pin due to various junctions
	RIC	= interconnect resistance
	SS	= sampling switch
	CHOLD	= sample/hold capacitance (from DAC)
	RSS	= sampling switch resistance



Note: When the conversion is started, the holding capacitor is disconnected from the input pin.

The source impedance (R_S) and the internal sampling switch (R_{SS}) impedance directly affect the time required to charge the capacitor C_{HOLD}. The sampling switch (R_{SS}) impedance varies over the device voltage (V_{DD}). The source impedance affects the offset voltage at the analog input (due to pin leakage current). **The maximum recommended impedance for analog sources is R_S = 2.5 k**. After the analog input channel is selected (changed), the channel must be sampled for at least the minimum acquisition time before starting a conversion.

A/D acquisition time and minimum charging time are calculated as shown on the following page:

- System configuration for the calculation:

C_{HOLD}	=	120 pF
R_s	=	2.5 kΩ
Conversion Error	≤	1/2 LSb
V_{DD}	=	5V → RSS = 7 kΩ
Temperature	=	50°C (system max.)
V_{HOLD}	=	0V @ time = 0

- Acquisition Time (T_{ACQ})
  T_{ACQ} = Amplifier Settling Time + Holding Cap. Charging Time + Temp. Coefficient
  $T_{ACQ} = T_{AMP} + T_C + T_{COFF}$

  Note: This equation assumes that 1/2 LSb error is used (10-bits or 1024 steps for the A/D). The 1/2 LSb error is the maximum error allowed for the A/D to meet its specified resolution

- A/D Minimum Charging Time
  $V_{HOLD} = (\Delta V_{REF} – (\Delta V_{REF}/2048)) \bullet (1 – e^{(-TC/CHOLD(RIC + RSS + RS))})$
  or
  $T_C = -(C_{HOLD})(RIC + RSS + RS)\ \ln(1/2048)$

- Calculating Minimum Required Acquisition Time
  $T_{ACQ} = T_{AMP} + T_C + T_{COFF}$
  $T_{AMP} = 5\ \mu s$
  $T_{COFF} = (Temp – 25^oC)(0.05\ \mu s/^oC) = (50^oC – 25^oC)(0.05\ \mu s/^oC) = 1.25\ \mu s$

  Temperature coefficient is only required for temp. > 25ºC. Below 25ºC, $T_{COFF} = 0\ \mu s$.
  $T_C = -(C_{HOLD})(RIC + RSS + RS)\ \ln(1/2047)\ \mu s$
  $-(120\ pF)\ (1\ k\Omega + 7\ k\Omega + 2.5\ k\Omega)\ \ln(0.0004883)\ \mu s$
  $9.61\ \mu s$

  $T_{ACQ} = 5\ \mu s + 1.25\ \mu s + 9.61\ \mu s = 12.86\ \mu s$ “minimum acquisition time”

- A/D V_{REF+} and V_{REF-} References
  PICmicro may be configured to use external voltage references instead of the internal AVDD and AVSS sources. If external sources are used, the source impedance of the VREF+ and VREF- voltage sources must be considered. **The maximum recommended impedance of the VREF+ and VREF- external reference voltage sources is 250Ω..**

- Automatic Acquisition Time
  The ADCON2 register allows the user to select an acquisition time that occurs each time the ‘GO/DONE’ bit is set. When the ‘GO/DONE’ bit is set, sampling is stopped and a conversion begins. The user is responsible for ensuring the required acquisition time has passed between selecting the desired input channel and setting the GO/DONE’ bit. This occurs when the ACQT2:ACQT0 bits (ADCON2<5:3>) remain in their Reset state (‘000’) and is compatible with devices that do not offer programmable acquisition times.

  If desired, the ACQT bits can be set to select a programmable acquisition time for the A/D module. When the GO/DONE’ bit is set, the A/D module continues to sample the input for the selected acquisition time, then automatically begins a conversion. Since the acquisition time is programmed, there may be no need to wait for an acquisition time between selecting a channel and setting the GO/DONE’ bit. For cases in which the GO/DONE’ bit is cleared when the conversion is completed, the ADIF flag is set and the A/D begins sampling the currently selected channel again. If an acquisition time is programmed, there is nothing to indicate if the acquisition time has ended or if the conversion has begun.

- Selecting A/D Conversion Clock
  The A/D conversion time per bit is defined as TAD. The A/D conversion requires 11 TAD per 10-bit conversion. The source of the A/D conversion clock is software selectable. There are seven possible options for TAD:

A/D Clock (T_{AD})	ADCS2:ADSC0 Bits	Max. F_{osc}=1/T_{osc}
2 * Tosc	000	1.25 Mhz
4 * Tosc	100	2.50 Mhz
8 * Tosc	001	5.00 Mhz
16 * Tosc	101	10.0 Mhz
32 * Tosc	010	20.0 Mhz
64 * Tosc	110	40.0 Mhz
RC(1)	x11	1.00 Mhz

Note: (1)The internal RC source has a typical TAD time of 4 μs.

- Operation in Low-Power Modes
  The selection of the automatic acquisition time and the A/D conversion clock is determined, in part, by the low-power mode clock source and frequency while in a low-power mode.

  If the A/D is expected to operate while the device is in a low-power mode, the ACQT2:ACQT0 and ADCS2:ADCS0 bits in ADCON2 should be updated in accordance with the low-power mode clock that will be used. After the low-power mode is entered (from either of the Run modes), an A/D acquisition or conversion may be started. Once an acquisition or conversion is started, the device should continue to be clocked by the same low-power mode clock source until the conversion has been completed. If desired, the device may be placed into the corresponding low-power (ANY)_IDLE mode during the conversion.

  If the low-power mode clock frequency is less than 1 MHz, the A/D RC clock source should be selected. Operation in the Low-Power Sleep mode requires the A/D RC clock to be selected. If bits ACQT2:ACQT0 are set to '000' and a conversion is started, the conversion will be delayed one instruction cycle to allow execution of the SLEEP instruction and entry to Low-Power Sleep mode. The IDLEN and SCS bits in the OSCCON register must have already been cleared prior to starting the conversion.

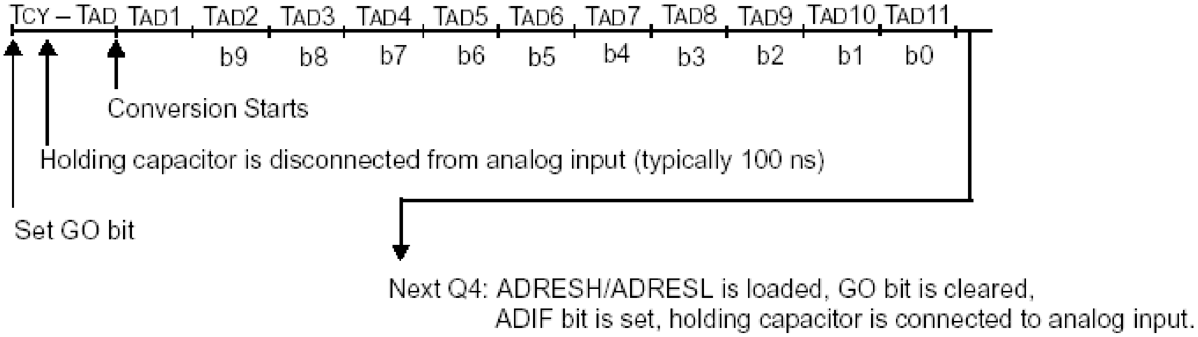
- Configuring Analog Port Pins
  The ADCON1, TRISA and TRISB registers are used to configure the A/D port pins. The port pins needed as analog inputs must have their corresponding TRIS bits set (input). If the TRIS bit is cleared (output), the digital output level (VOH or VOL) will be converted.

  The A/D operation is independent of the state of the CHS2:CHS0 bits and the TRIS bits.

  Notes:
  1) When reading the Port register, all pins configured as analog input channels will read as cleared (a low level). Pins configured as digital inputs will convert an analog input to a high or a low level.
  2) Analog levels on any pin defined as a digital input may cause the digital input buffer to consume current out of the device's specification limits.

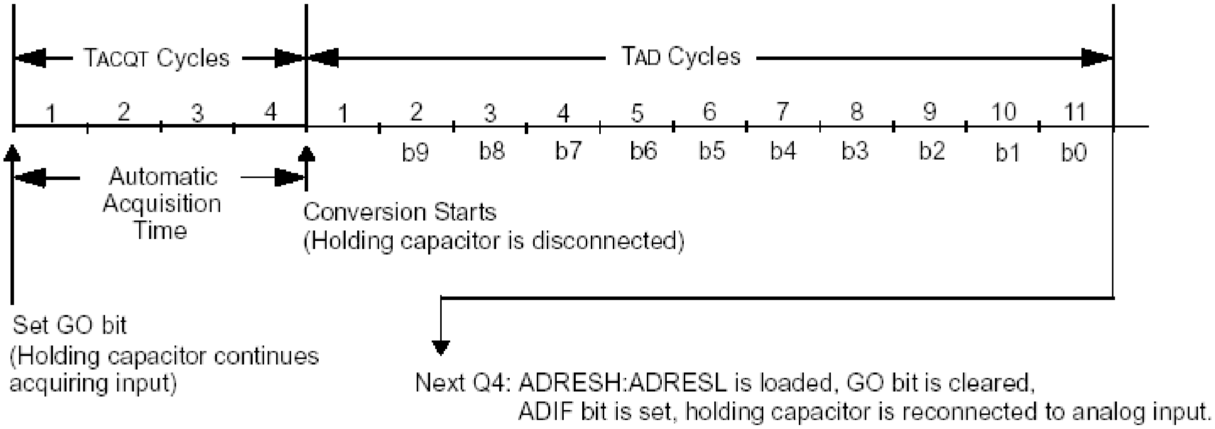
- A/D Conversion timing
  The following diagram shows the operation of the A/D converter after the GO bit has been set and the ACQT2:ACQT0 bits are cleared:

A/D CONVERSION T_{AD} Cycle (ACQT<2:0>=000, T_{ACQ}=0)


TCY – TAD
TAD1
TAD2
TAD3
TAD4
TAD5
TAD6
TAD7
TAD8
TAD9
TAD10
TAD11
b9
b8
b7
b6
b5
b4
b3
b2
b1
b0
Conversion Starts
Holding capacitor is disconnected from analog input (typically 100 ns)
Set GO bit
Next Q4: ADRESH/ADRESL is loaded, GO bit is cleared,
ADIF bit is set, holding capacitor is connected to analog input.

The following diagram shows the operation of the A/D converter after the GO bit has been set, the ACQT2:ACQT0 bits have been set to ‘010’ and a 4 TAD acquisition time has been selected before the conversion starts:

A/D CONVERSION T_{AD} Cycle (ACQT<2:0>=010, T_{ACQ}=4T_{AD})


TACQT Cycles
TAD Cycles
1
2
3
4
1
2
3
4
5
6
7
8
9
10
11
b9
b8
b7
b6
b5
b4
b3
b2
b1
b0
Automatic Acquisition Time
Conversion Starts
(Holding capacitor is disconnected)
Set GO bit
(Holding capacitor continues acquiring input)
Next Q4: ADRESH:ADRESL is loaded, GO bit is cleared,
ADIF bit is set, holding capacitor is reconnected to analog input.

Clearing the GO/DONE’ bit during a conversion will abort the current conversion. The A/D Result register pair will NOT be updated with the partially completed A/D conversion sample. This means the ADRESH:ADRESL registers will continue to contain the value of the last completed conversion (or the last value written to the ADRESH:ADRESL registers).

After the A/D conversion is completed or aborted, a 2 TAD wait is required before the next acquisition can be started. After this wait, acquisition on the selected channel is automatically started.

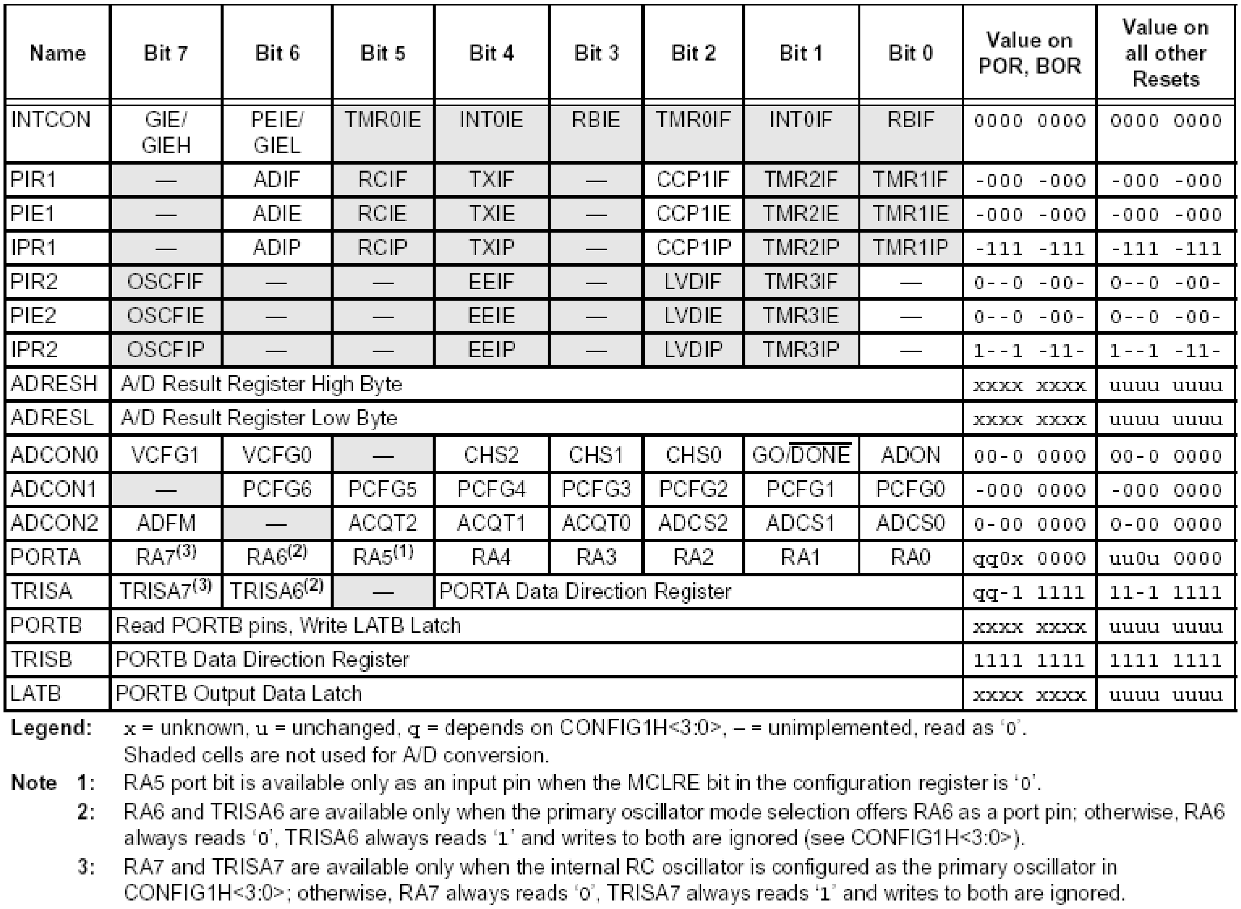
Note: The GO/DONE’ bit should NOT be set in the same instruction that turns on the A/D.

- Use of the CCP1 Trigger
  An A/D conversion can be started by the “special event trigger” of the CCP1 module. This requires that the CCP1M3:CCP1M0 bits (CCP1CON<3:0>) be programmed as ‘1011’ and that the A/D module is enabled (ADON bit is set). When the trigger occurs, the GO/DONE’ bit will be set, starting the A/D acquisition and conversion, and the Timer1 (or Timer3) counter will be reset

to zero. Timer1 (or Timer3) is reset to automatically repeat the A/D acquisition period with minimal software overhead (moving ADRESH/ADRESL to the desired location). The appropriate analog input channel must be selected, and the minimum acquisition period is either timed by the user, or an appropriate T_{ACQ} time selected before the “special event trigger” sets the GO/DONE bit (starts a conversion).

If the A/D module is not enabled (ADON is cleared), the “special event trigger” will be ignored by the A/D module, but will still reset the Timer1 (or Timer3) counter.

- Summary of A/D Registers



Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
INTCON	GIE/ GIEH	PEIE/ GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 0000	0000 0000
PIR1	—	ADIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF	-000 -000	-000 -000
PIE1	—	ADIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE	-000 -000	-000 -000
IPR1	—	ADIP	RCIP	TXIP	—	CCP1IP	TMR2IP	TMR1IP	-111 -111	-111 -111
PIR2	OSCFIF	—	—	EEIF	—	LVDIF	TMR3IF	—	0--0 -00-	0--0 -00-
PIE2	OSCFIE	—	—	EEIE	—	LVDIE	TMR3IE	—	0--0 -00-	0--0 -00-
IPR2	OSCFIP	—	—	EEIP	—	LVDIP	TMR3IP	—	1--1 -11-	1--1 -11-
ADRESH	A/D Result Register High Byte								xxxx xxxx	uuuu uuuu
ADRESL	A/D Result Register Low Byte								xxxx xxxx	uuuu uuuu
ADCON0	VCFG1	VCFG0	—	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON	00-0 0000	00-0 0000
ADCON1	—	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0	-000 0000	-000 0000
ADCON2	ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0	0-00 0000	0-00 0000
PORTA	RA7[3]	RA6[2]	RA5[1]	RA4	RA3	RA2	RA1	RA0	qq0x 0000	uu0u 0000
TRISA	TRISA7[3]	TRISA6[2]	—	PORTA Data Direction Register					qq-1 1111	11-1 1111
PORTB	Read PORTB pins, Write LATB Latch								xxxx xxxx	uuuu uuuu
TRISB	PORTB Data Direction Register								1111 1111	1111 1111
LATB	PORTB Output Data Latch								xxxx xxxx	uuuu uuuu

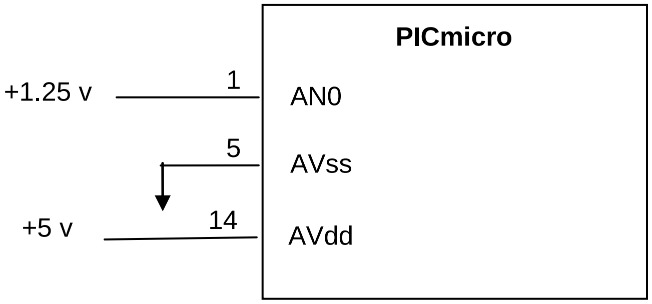
Legend: x = unknown, u = unchanged, q = depends on CONFIG1H<3:0>, – = unimplemented, read as ‘0’.
Shaded cells are not used for A/D conversion.

Note 1: RA5 port bit is available only as an input pin when the MCLRE bit in the configuration register is ‘0’.

2: RA6 and TRISA6 are available only when the primary oscillator mode selection offers RA6 as a port pin; otherwise, RA6 always reads ‘0’, TRISA6 always reads ‘1’ and writes to both are ignored (see CONFIG1H<3:0>).

3: RA7 and TRISA7 are available only when the internal RC oscillator is configured as the primary oscillator in CONFIG1H<3:0>; otherwise, RA7 always reads ‘0’, TRISA7 always reads ‘1’ and writes to both are ignored.

- Example – Channel 0 is enabled, PICmicro is configured as an A/D convertor and the range is set from AVss to AVdd. What is the value of the A/D convertor output?



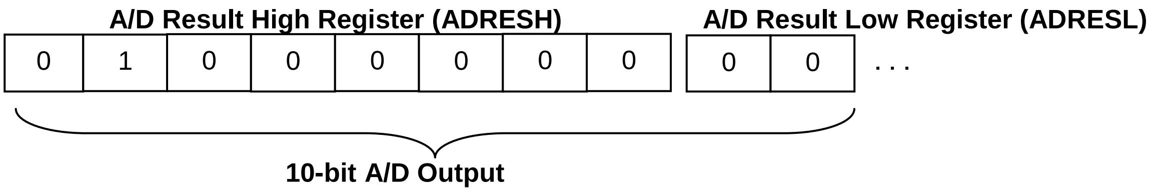
Solution:
A/D has 10-bit output which means there are 2^{10} counts between 0 and 5 v.

voltage/count = (5-0) / (2^{10}) = 5 / (2^{10})

Output count at 1.25 = 1.25 /{ 5 / (2^{10})} = 256

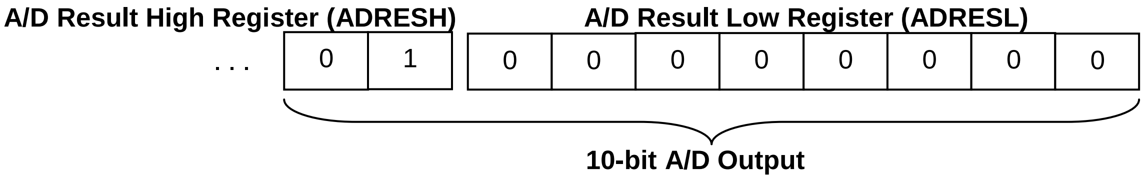
10 bit A/D output → "01 0000 0000"

When 10-bit A/D Result is Left Justified:



A/D Result High Register (ADRESH)								A/D Result Low Register (ADRESL)		
0	1	0	0	0	0	0	0	0	0	. . .

When 10-bit A/D Result is Right Justified:



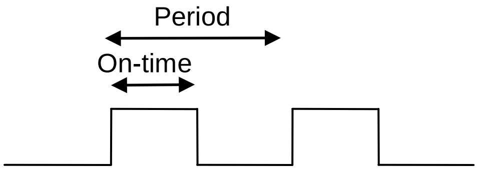
A/D Result High Register (ADRESH)			A/D Result Low Register (ADRESL)							
. . .	0	1	0	0	0	0	0	0	0	0

4.10. Pulse Width Modulation (PWM)

Pulse Width Modulation(PWM) uses a square wave whose duty cycle is modulated resulting in the variation of the average power delivered by the waveform. Modulating duty cycle means changing the percentage of the period that is at high voltage (not zero). This technique is used to control power levels in electrical devices such as light sources, motors, heaters and other devices.

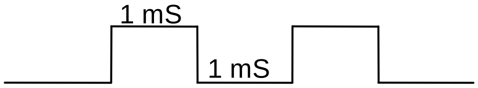
For example, in the following diagram, waveform A delivers twice as much average power as waveform B. This is useful in modulating electrical device performance. One of the important uses of PWM is in DC motor speed control.

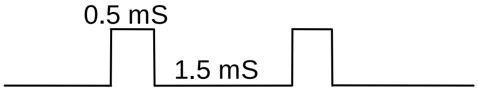
PWM Output pulse Definition:



PWM Output Waveform (duty Cyle = 100*(on-time/period)%

Examples:





Waveform A. 50% Duty Cycle

Waveform B. 25% Duty Cycle

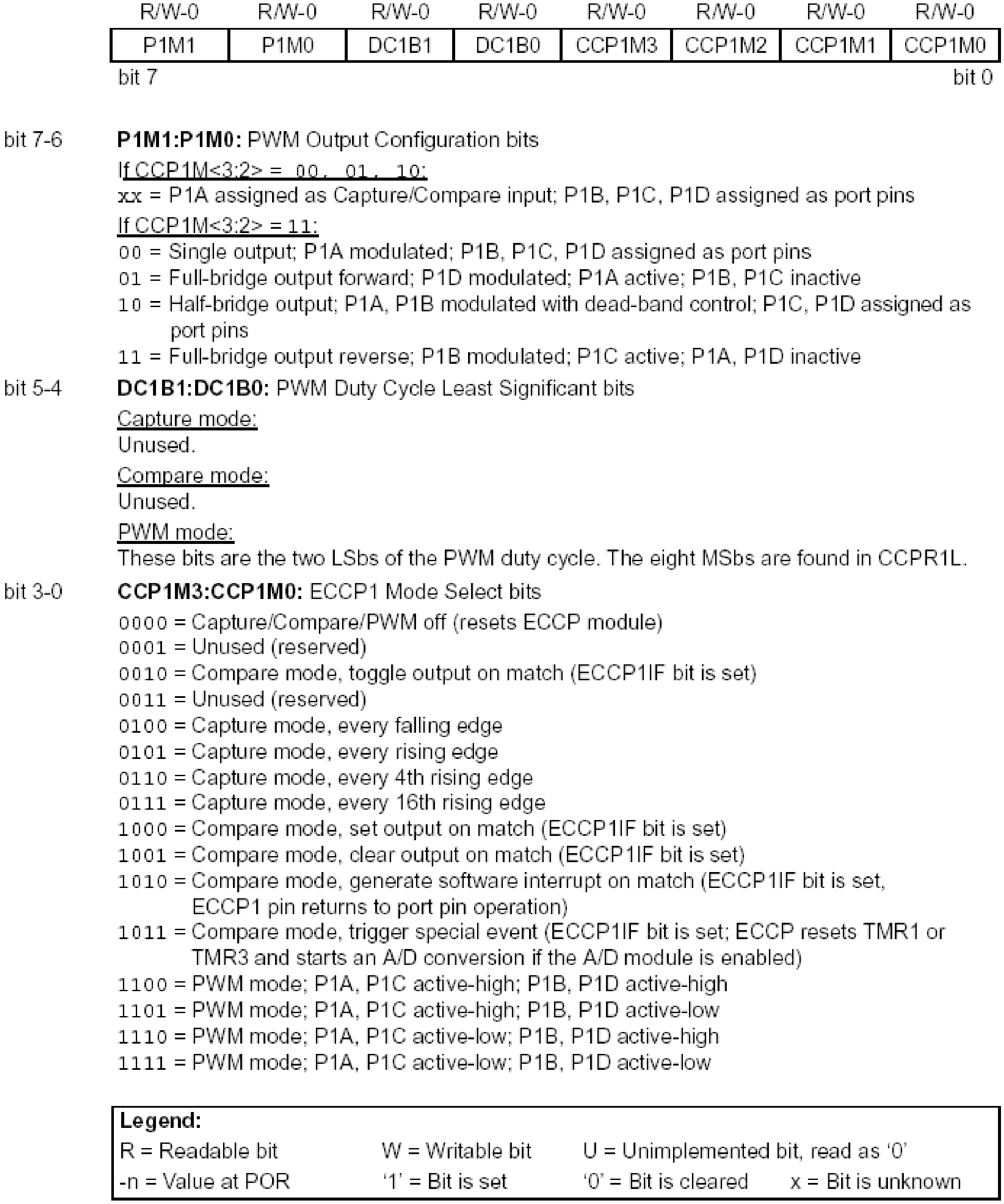
PWM is implemented in PICmicro as one of the three features of the Enhance Capture/Compare/PWM (ECCP) module. Below is a list of ECCP key features::

- One, two or four PWM outputs
- Selectable polarity
- Programmable dead time (Low)
- Auto-Shutdown and Auto-Restart
- Capture is 16-bit, max resolution 6.25 ns (T_{CY}/16)
- Compare is 16-bit, max resolution 100 ns (T_{CY})

Below are high-level steps to set up a Pulse-Wide-Modulation

1) Set PR2 (PWMperiod = ((PR2)+1)*4 * TOSC * (TMR2 Prescale Value))
2) Configure and Clear Timer2 (T2CON, 2)
3) Set up PWM Duty Cyle (CCPR1L & CCP1CON)
4) Set mode (CCP1CON)

- CCP1CON register controls ECCP operation



R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0
bit 7							bit 0

bit 7-6 **P1M1:P1M0:** PWM Output Configuration bits

If CCP1M<3:2> = 00, 01, 10:
xx = P1A assigned as Capture/Compare input; P1B, P1C, P1D assigned as port pins

If CCP1M<3:2> = 11:
00 = Single output; P1A modulated; P1B, P1C, P1D assigned as port pins
01 = Full-bridge output forward; P1D modulated; P1A active; P1B, P1C inactive
10 = Half-bridge output; P1A, P1B modulated with dead-band control; P1C, P1D assigned as port pins
11 = Full-bridge output reverse; P1B modulated; P1C active; P1A, P1D inactive

bit 5-4 **DC1B1:DC1B0:** PWM Duty Cycle Least Significant bits

Capture mode:
Unused.

Compare mode:
Unused.

PWM mode:
These bits are the two LSbs of the PWM duty cycle. The eight MSbs are found in CCPR1L.

bit 3-0 **CCP1M3:CCP1M0:** ECCP1 Mode Select bits

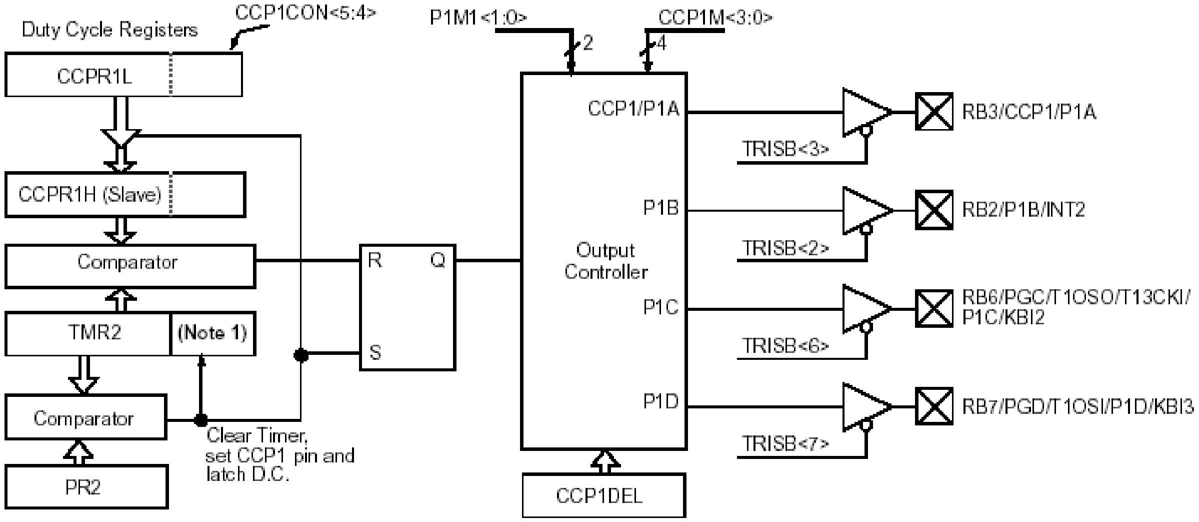
0000 = Capture/Compare/PWM off (resets ECCP module)
0001 = Unused (reserved)
0010 = Compare mode, toggle output on match (ECCP1IF bit is set)
0011 = Unused (reserved)
0100 = Capture mode, every falling edge
0101 = Capture mode, every rising edge
0110 = Capture mode, every 4th rising edge
0111 = Capture mode, every 16th rising edge
1000 = Compare mode, set output on match (ECCP1IF bit is set)
1001 = Compare mode, clear output on match (ECCP1IF bit is set)
1010 = Compare mode, generate software interrupt on match (ECCP1IF bit is set, ECCP1 pin returns to port pin operation)
1011 = Compare mode, trigger special event (ECCP1IF bit is set; ECCP resets TMR1 or TMR3 and starts an A/D conversion if the A/D module is enabled)
1100 = PWM mode; P1A, P1C active-high; P1B, P1D active-high
1101 = PWM mode; P1A, P1C active-high; P1B, P1D active-low
1110 = PWM mode; P1A, P1C active-low; P1B, P1D active-high
1111 = PWM mode; P1A, P1C active-low; P1B, P1D active-low

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

Note: PWM only uses Timer2.

The Enhanced PWM Mode provides additional PWM output options for a broader range of control applications. The module is an upwardly compatible version of the standard CCP module and offers up to four outputs, designated P1A through P1D. Users are also able to select the polarity of the signal (either active-high or active-low). The module’s output mode and polarity are configured by setting the P1M1:P1M0 and CCP1M3:CCP1M0 bits of the CCP1CON register (CCP1CON<7:6> and CCP1CON<3:0>, respectively).

The following figure shows a simplified block diagram of PWM operation.



Note: The 8-bit TMR2 register is concatenated with the 2-bit internal Q clock, or 2 bits of the prescaler to create the 10-bit time base.

All control registers are double-buffered and are loaded at the beginning of a new PWM cycle (the period boundary when Timer2 resets) in order to prevent glitches on any of the outputs. The exception is the PWM Delay register, ECCP1DEL, which is loaded at either the duty cycle boundary or the boundary period (whichever comes first). Because of the buffering, the module waits until the assigned timer resets instead of starting immediately. This means that Enhanced PWM waveforms do not exactly match the standard PWM waveforms, but are instead offset by one full instruction cycle (4 TOSC).

As before, the user must manually configure the appropriate TRIS bits for output.

- PWM Period

The PWM period is specified by writing to the PR2 register. The PWM period can be calculated using the equation:

PWM Period = [(PR2) + 1] • 4 • TOSC • (TMR2 Prescale Value)

PWM frequency is defined as 1/[PWM period]. When TMR2 is equal to PR2, the following three events occur on the next increment cycle:

- TMR2 is cleared
- The CCP1 pin is set (if PWM duty cycle = 0%, the CCP1 pin will not be set)
- The PWM duty cycle is copied from CCPR1L into CCPR1H

Note that The Timer2 postscaler is not used in the determination of the PWM frequency.

- PWM Duty Cycle
  The PWM duty cycle is specified by writing to the CCPR1L register and to the CCP1CON<5:4> bits. Up to 10-bit resolution is available. The CCPR1L contains the eight MSbs and the CCP1CON<5:4> contains the two LSbs. This 10-bit value is represented by CCPR1L:CCP1CON<5:4>. The PWM duty cycle is calculated by the equation:

  PWM Duty Cycle = (CCPR1L:CCP1CON<5:4>) • TOSC • (TMR2 Prescale Value)

  CCPR1L and CCP1CON<5:4> can be written to at any time, but the duty cycle value is not copied into CCPR1H until a match between PR2 and TMR2 occurs (i.e., the period is complete). In PWM mode, CCPR1H is a read-only register.

  The CCPR1H register and a 2-bit internal latch are used to double-buffer the PWM duty cycle. This double-buffering is essential for glitch-less PWM operation. When the CCPR1H and 2-bit latch match TMR2, concatenated with an internal 2-bit Q clock or two bits of the TMR2 pre-scalar, the CCP1 pin is cleared. The maximum PWM resolution (bits) for a given PWM frequency is given by the equation:

  PWM Resolution $= \frac{\log\left(\frac{F_{osc}}{F_{PWM}}\right)}{\log(2)} bits$

  Note: If the PWM duty cycle value is longer than the PWM period, the CCP1 pin will not be cleared.

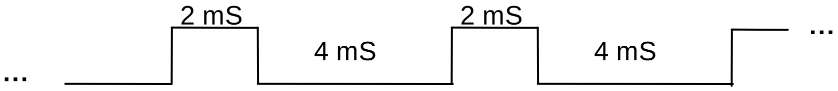
- PWM Output Configuration
  The P1M1:P1M0 bits in the CCP1CON register allow one of four configurations:

  - Single Output
  - Half-Bridge Output
  - Full-Bridge Output, Forward mode
  - Full-Bridge Output, Reverse mode

- Example – Determine register values to set up PWM of PICmicro to generate a signal on P1A pin (use internal RC clock) that has a period of 6 msec. and 30% duty cycle.

  Solution:

  Signal to be generated



  1) Use the following equations:

  PWM Period = [(PR2) + 1] • 4 • TOSC • (TMR2 Prescale Value)
  Tosc = 32 µsec. for internal RC clock

  .   to set the values of PR2 and the TMR2 Prescale.

  2) Use the following equation:

  PWM Duty Cycle = (CCPR1L:CCP1CON<5:4>) • TOSC • (TMR2 Prescale Value)

  to Calculate and set the value for registers CCPR1L:CCP1CON<5:4>.

  3) Configure & Clear Timer 2
  TMR2 = 0 and PIR1=0 and TMR2IF=0

- Example – Write a code segment to configure and use a single channel PWM to control a DC motor.

  Solution:
  Refer to Lab documentation.

4.11. Additional Resources

- Staff. Microchip PIC 18F1220/1320 Data Sheet. (2004) Microchip Technology Incorporated.

- Huang,. PIC Microcontroller: An Introduction to Software & Hardware Interfacing, (2004) Thomson.

- Reese. Microprocessor: From Assembly Language to C using the PIC18Fxxx2. (2003) Course Technology.

- Peterson. Computer Organization and Design, (2007) Elsevier Service.

4.12. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 5. ARITHMETIC & LOGIC OPERATIONS

Key concepts and Overview

- ❖ Arithmetic Operations
- ❖ Move, Set and Clear Operations
- ❖ Logic Operations
- ❖ Branch Operations
- ❖ Specialty Operations
- ❖ IEEE Standards for Floating Point
- ❖ Additional Resources

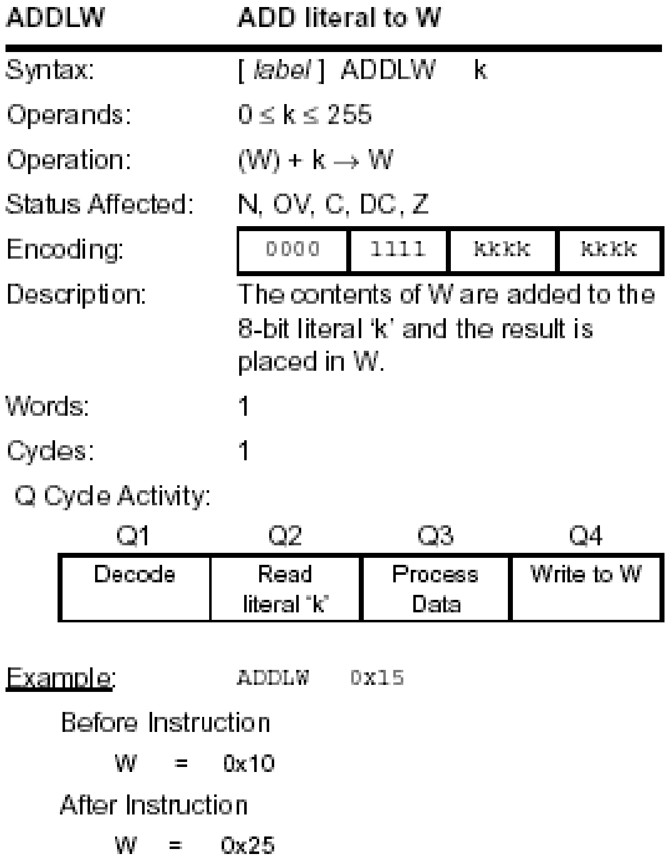
5.1. Arithmetic Operations

PICmicro offers a wide range of arithmetic operations as do the majority of the processors on the market. In this section each of the arithmetic instructions are described in detail.

It is recommended that the reader utilize the PICmicro development and simulation tools to verify and explore the full operation of these instructions.

The remainder of this section covers Add, Increment, Multiply and Subtract operations.

- ADDLW literal and WREG



ADDLW **ADD literal to W**

Syntax: [*label*] ADDLW k

Operands: 0 ≤ k ≤ 255

Operation: (W) + k → W

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

Encoding:

0000	1111	kkkk	kkkk

Description: The contents of W are added to the 8-bit literal 'k' and the result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: ADDLW 0x15

Before Instruction

W = 0x10

After Instruction

W = 0x25

- Example – Given Wreg=25 and the following statement is executed:

  ADDLW 0xF5

  What are the status bit values?

  Solution:

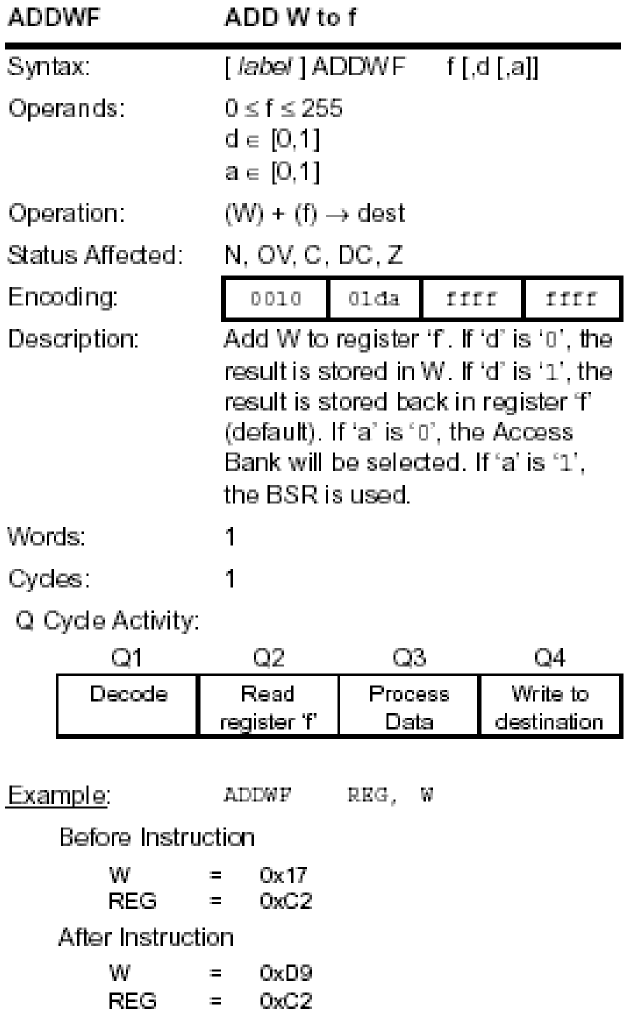
N	OV	C	DC	Z
0	1	1	1	0

Note: DC is carry over from lower nibble.

- *What value of Wreg and K will cause "Z" & "C" to be set to zero when ADDLW instruction is executed.*

  Solution:

- Add WREG and f



ADDWF	**ADD W to f**
Syntax:	[*label*] ADDWF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) + (f) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	`0010` `01da` `ffff` `ffff`
Description:	Add W to register 'f'. If 'd' is '`0`', the result is stored in W. If 'd' is '`1`', the result is stored back in register 'f' (default). If 'a' is '`0`', the Access Bank will be selected. If 'a' is '`1`', the BSR is used.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ADDWF REG, W`

Before Instruction

W	=	0x17
REG	=	0xC2

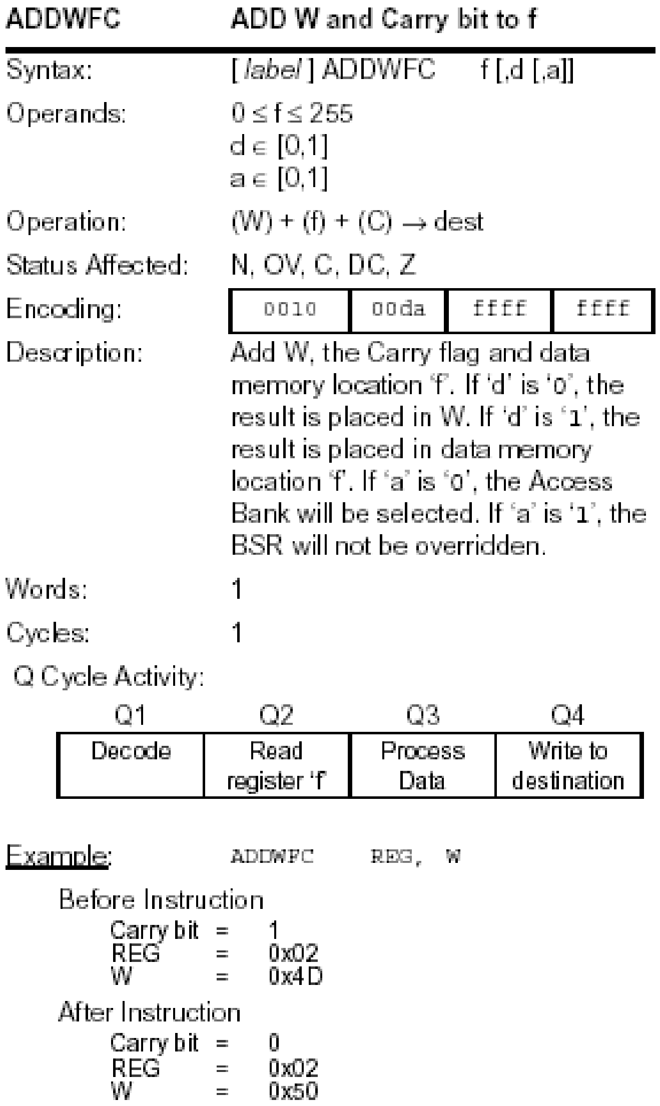
After Instruction

W	=	0xD9
REG	=	0xC2

Notes:

-

- ❖ Add WREG and Carry bit to f



ADDWFC **ADD W and Carry bit to f**

Syntax:	[*label*] ADDWFC f [,d [,a]]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(W) + (f) + (C) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	`0010` `00da` `ffff` `ffff`
Description:	Add W, the Carry flag and data memory location 'f'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed in data memory location 'f'. If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR will not be overridden.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ADDWFC REG, W`

Before Instruction

Carry bit	=	1
REG	=	0x02
W	=	0x4D

After Instruction

Carry bit	=	0
REG	=	0x02
W	=	0x50

Notes:

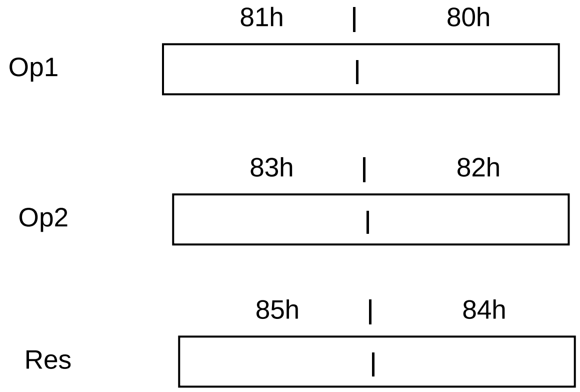
- Write an Assembly code segment that add A (location 0x10) and B (location 0x12) and stores the results in C(location 0x14).

  Solution:

- Write a C code segment that uses pointers to add A (location 0x10) and B (location 0x12) and stores the results in C(location 0x14).

  ```
  //Hint:
  //The following code declares and
  // initializes px to 0x12 and pointer
  // to integer (16 bit)

  Int    *px =0x12;
  *px = 25;            //set reg. 0x12 to 25
  ```

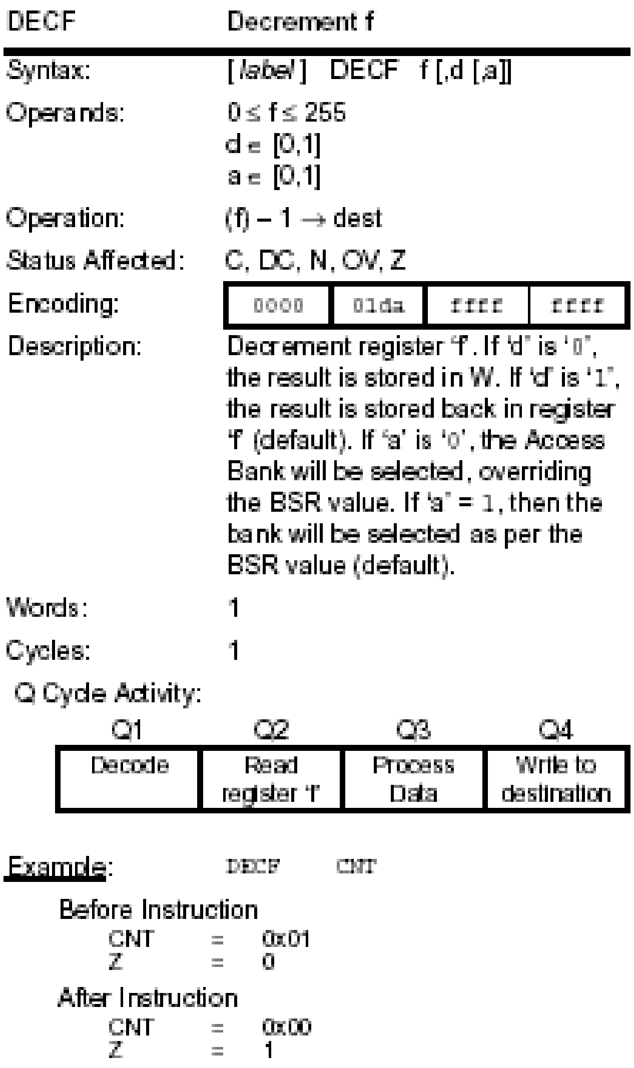
- Example – Write a function “Add16” that accepts two 15-bit input (Op1 and Op2) and places the 16-bit result from the addition of Op1 & Op2 into res:



Write pseudo code before writing the assembly code for the function “Add16”.

Solution:

- Decrement f



DECF	Decrement f
Syntax:	[*label*] DECF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) – 1 → dest
Status Affected:	C, DC, N, OV, Z
Encoding:	0000 | 01da | ffff | ffff
Description:	Decrement register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: DECF CNT

Before Instruction

CNT	=	0x01
Z	=	0

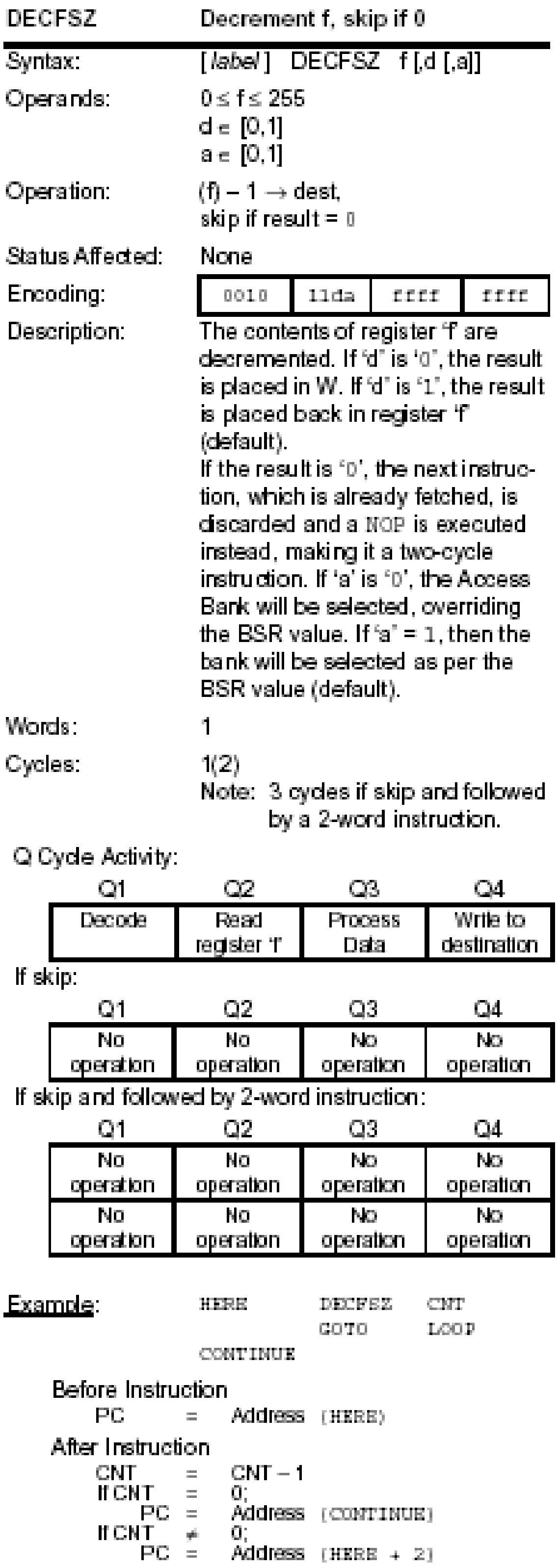
After Instruction

CNT	=	0x00
Z	=	1

Notes:

-

❖ Decrement f, Skip if 0



DECFSZ	**Decrement f, skip if 0**
Syntax:	[*label*] DECFSZ f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) – 1 → dest, skip if result = 0
Status Affected:	None
Encoding:	0010 | 11da | ffff | ffff
Description:	The contents of register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is '0', the next instruction, which is already fetched, is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1(2) Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

<u>Example</u>:

```
HERE        DECFSZ   CNT
            GOTO     LOOP
CONTINUE
```

Before Instruction
```
    PC       =   Address (HERE)
```
After Instruction
```
    CNT      =   CNT – 1
    If CNT   =   0;
        PC   =   Address (CONTINUE)
    If CNT   ≠   0;
        PC   =   Address (HERE + 2)
```

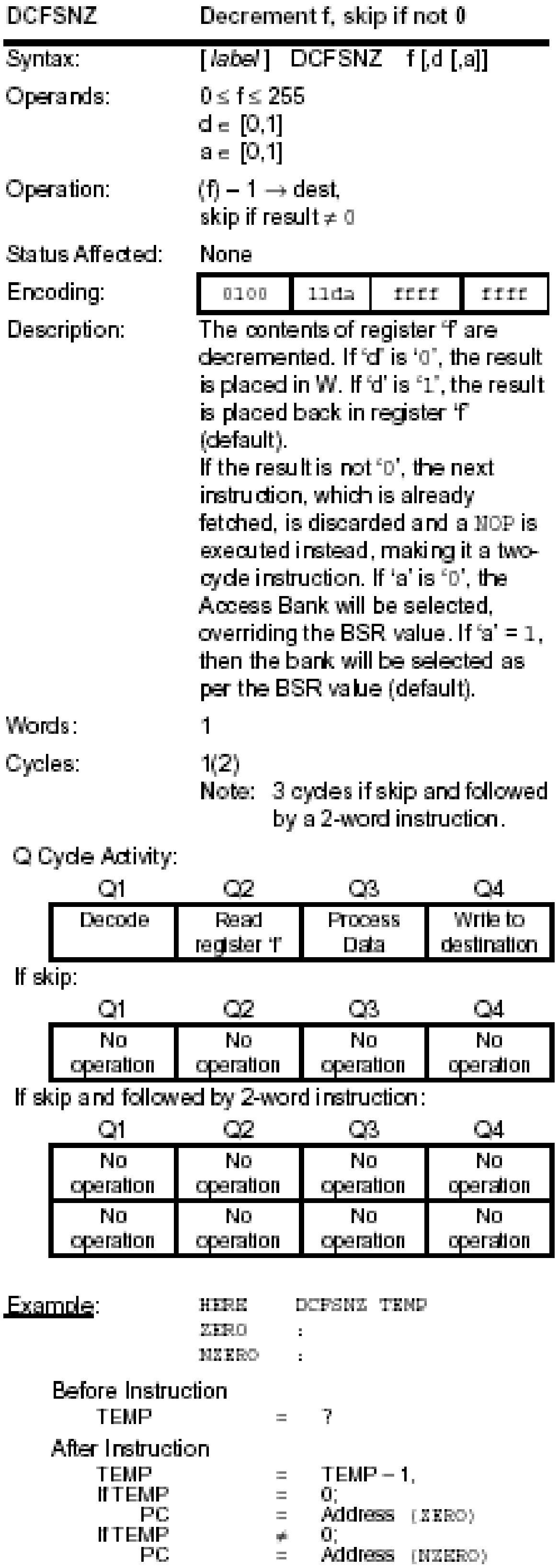
▪ Example - Write an Assembly code segment to implement the functionality of the following C code segment:

```
Unsigned char * pcount;
pcount = (unsigned char *) 0x80;

*pcount = *pcount -1;
if (*pcount != 0 ) {
        *pcount = 0x00;
}
```

<u>Solution</u>

```
pcount              equ    0x80;

L1:     DECFSZ      pcount,1
        CLRF        pcount
Done:
```

- Decrement f, Skip if Not 0



DCFSNZ **Decrement f, skip if not 0**

Syntax: [*label*] DCFSNZ f [,d [,a]]

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [0,1]

Operation: (f) – 1 → dest,
skip if result ≠ 0

Status Affected: None

Encoding:

0100	11da	ffff	ffff

Description: The contents of register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).
If the result is not '0', the next instruction, which is already fetched, is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    DCFSNZ TEMP
ZERO    :
NZERO   :
```

Before Instruction
TEMP = ?

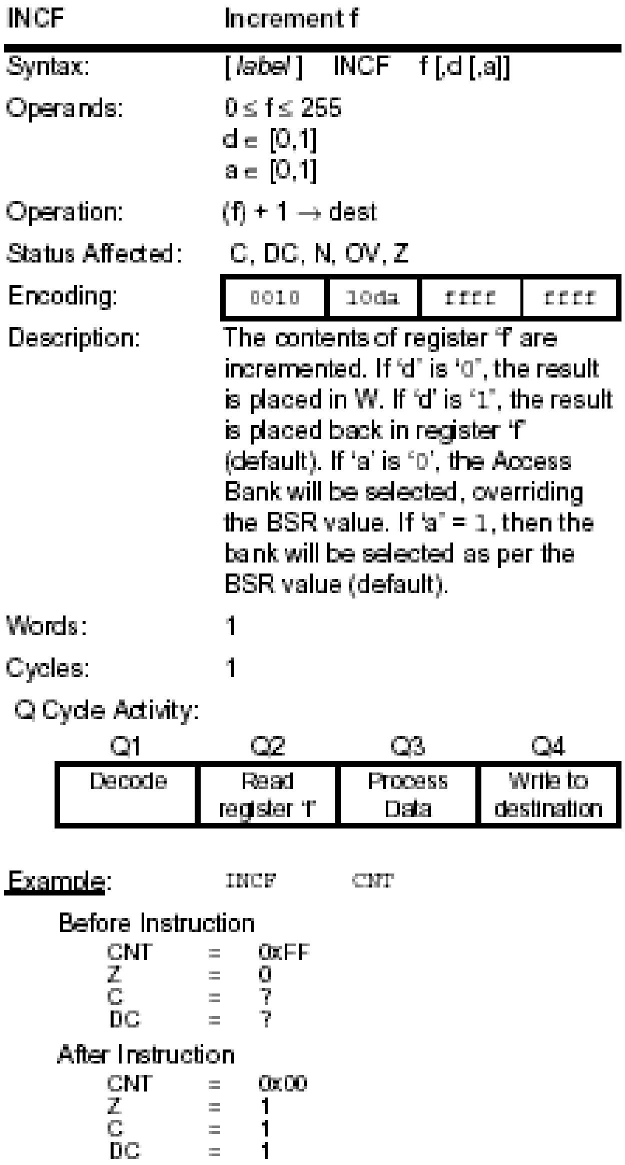
After Instruction
TEMP = TEMP – 1,
If TEMP = 0;
PC = Address (ZERO)
If TEMP ≠ 0;
PC = Address (NZERO)

- Example - Write an Assembly code segment that uses DCFSNZ to implement the functionality of the following C code segment:

*Unsigned char *pcount, i;*
*pcount = (unsigned char *) 0x80;*
for (i=50; i≥ 0 ; i--) {
    **pcount = i;*
}

Solution

❖ Increment f



INCF	Increment f
Syntax:	[*label*] INCF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) + 1 → dest
Status Affected:	C, DC, N, OV, Z
Encoding:	0010 10da ffff ffff
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: INCF CNT

Before Instruction

CNT	=	0xFF
Z	=	0
C	=	?
DC	=	?

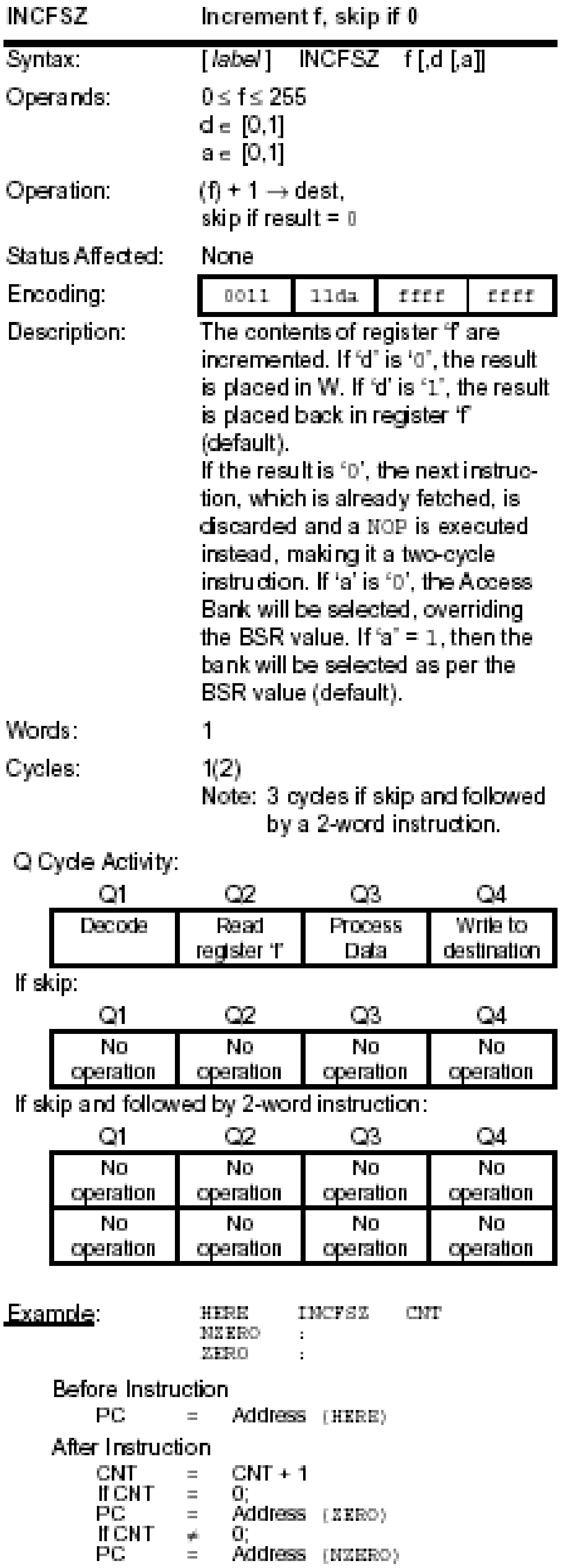
After Instruction

CNT	=	0x00
Z	=	1
C	=	1
DC	=	1

Notes:

-

- Increment f, Skip if 0



INCFSZ Increment f, skip if 0

Syntax: [*label*] INCFSZ f [,d [,a]]

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [0,1]

Operation: (f) + 1 → dest,
skip if result = 0

Status Affected: None

Encoding:

0011	11da	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).
If the result is '0', the next instruction, which is already fetched, is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

<u>Example</u>:

```
HERE     INCFSZ    CNT
NZERO    :
ZERO     :
```

Before Instruction
PC = Address (HERE)

After Instruction
CNT = CNT + 1
If CNT = 0;
PC = Address (ZERO)
If CNT ≠ 0;
PC = Address (NZERO)

Notes:

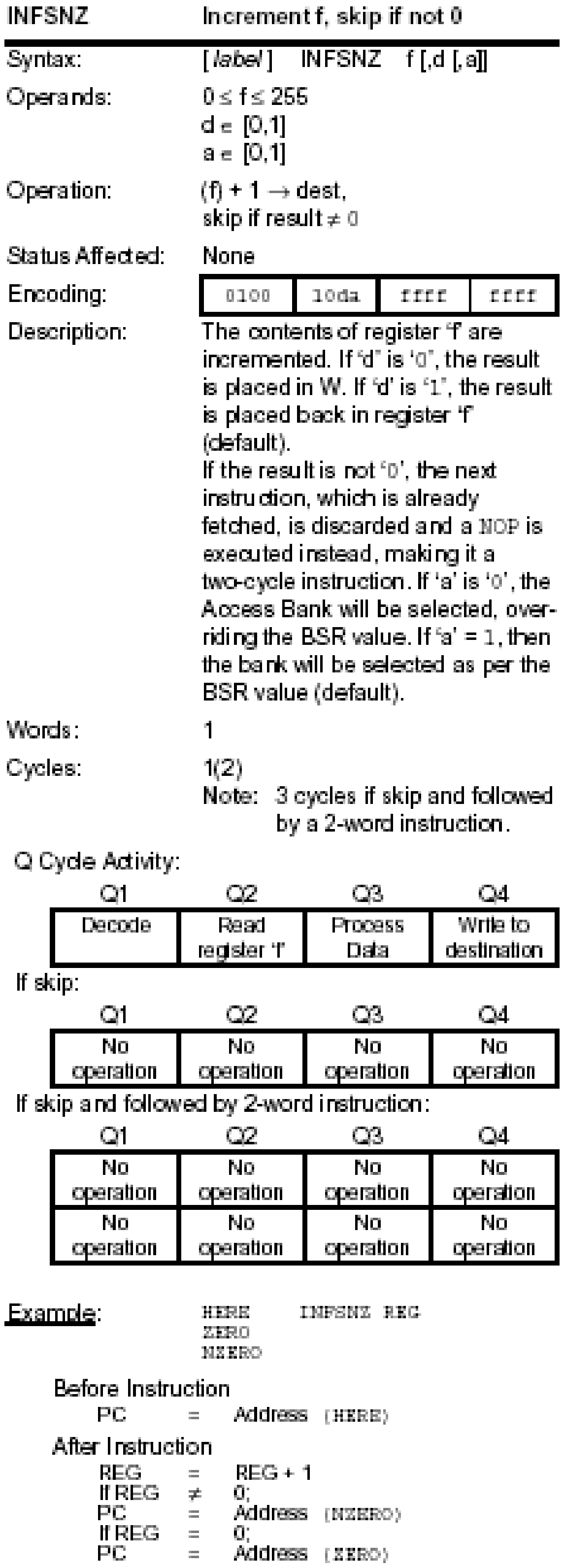
- Write an Assembly code segment to implement the functionality of the following C code segment:

*Unsigned char *pcount;*
int i;
*pcount = (unsigned char *) 0x80;*
for (i=25; i<256 ; i++) {
**pcount++ =(unsigned char)i;*
}

<u>Solution</u>

```
pcount      equ       0x80;

            MOVLW     25
            MOVWF     pcount

L1:         INFSZ     pcount,1
            BRA  L1
            DECFpcount
```

- Increment f, Skip if Not 0



INFSNZ **Increment f, skip if not 0**

Syntax: [*label*] INFSNZ f [,d [,a]]

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [0,1]

Operation: (f) + 1 → dest,
skip if result ≠ 0

Status Affected: None

Encoding:

0100	10da	ffff	ffff

Description: The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default).
If the result is not '0', the next instruction, which is already fetched, is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE      INFSNZ REG
ZERO
NZERO
```

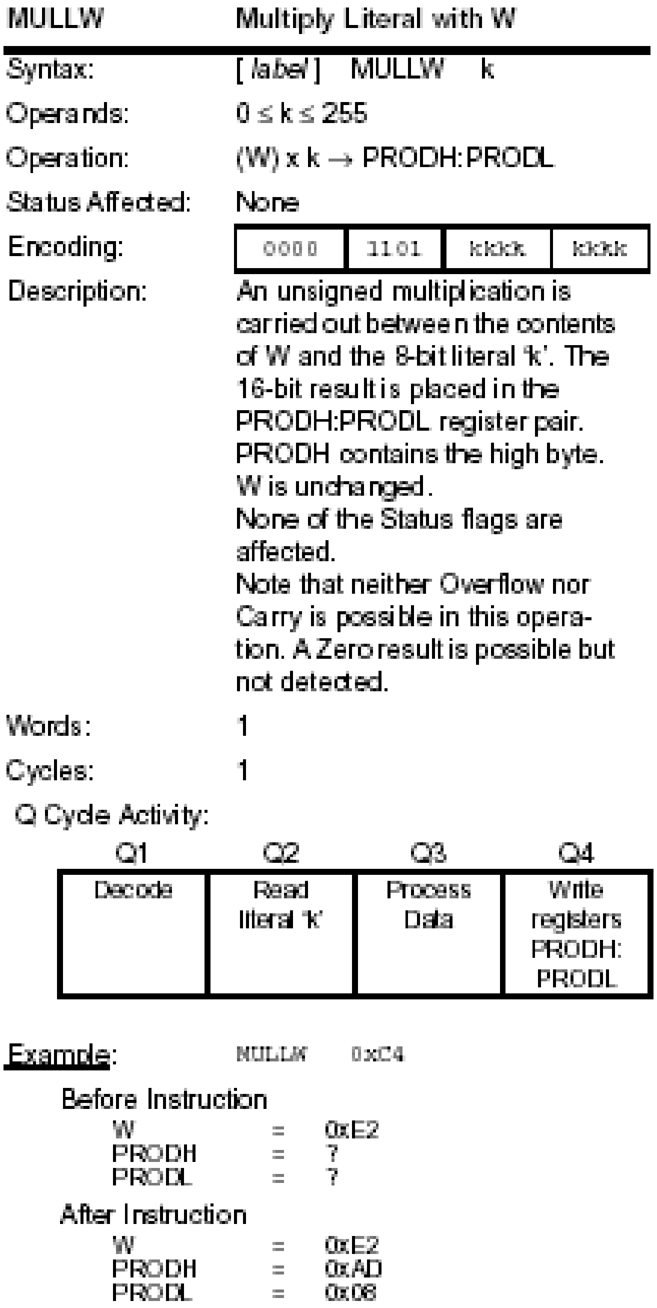
Before Instruction
PC = Address (HERE)

After Instruction
REG = REG + 1
If REG ≠ 0;
PC = Address (NZERO)
If REG = 0;
PC = Address (ZERO)

Notes:

-

- Multiply WREG with f



MULLW **Multiply Literal with W**

Syntax:	[*label*] MULLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) x k → PRODH:PRODL
Status Affected:	None
Encoding:	0000 | 1101 | kkkk | kkkk
Description:	An unsigned multiplication is carried out between the contents of W and the 8-bit literal 'k'. The 16-bit result is placed in the PRODH:PRODL register pair. PRODH contains the high byte. W is unchanged. None of the Status flags are affected. Note that neither Overflow nor Carry is possible in this operation. A Zero result is possible but not detected.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write registers PRODH: PRODL

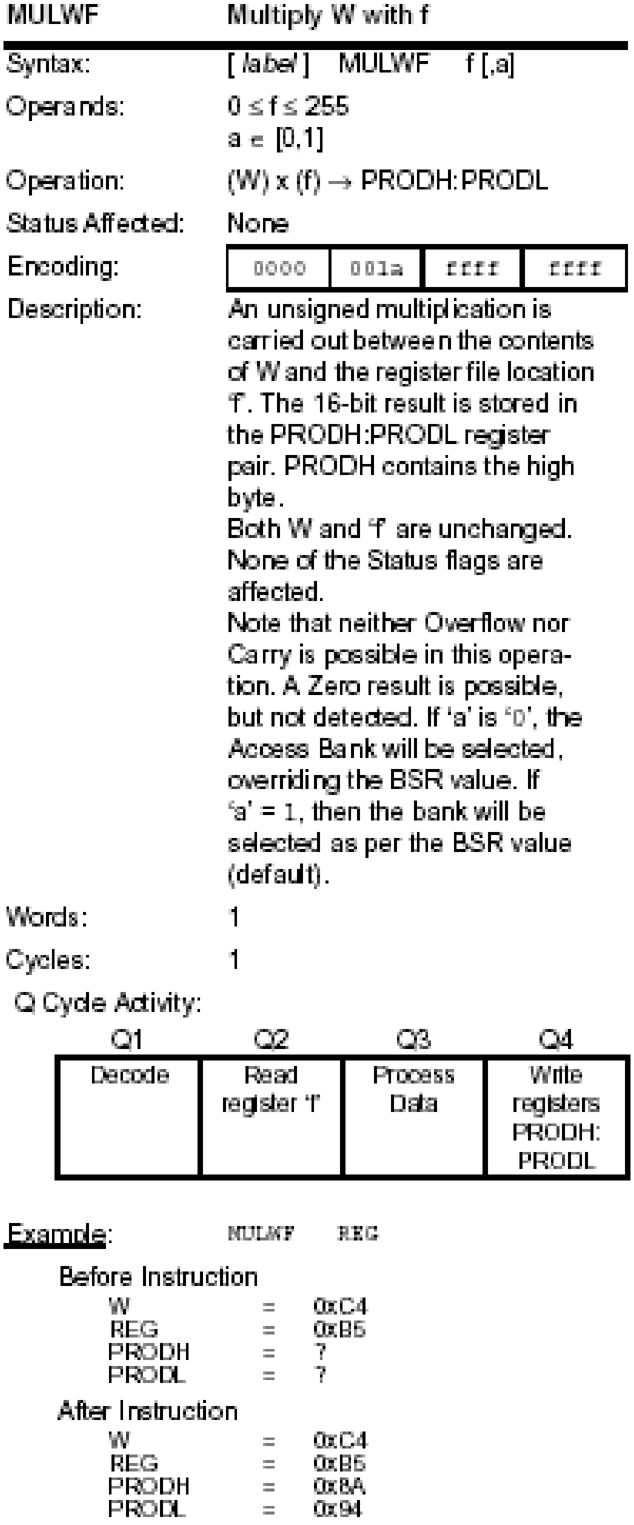
Example: `MULLW 0xC4`

Before Instruction
```
W       =   0xE2
PRODH   =   ?
PRODL   =   ?
```
After Instruction
```
W       =   0xE2
PRODH   =   0xAD
PRODL   =   0x08
```

Notes:

-

❖ Multiply WREG with f



MULWF	Multiply W with f
Syntax:	[*label*] MULWF f [,a]
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	(W) x (f) → PRODH:PRODL
Status Affected:	None
Encoding:	0000 | 001a | ffff | ffff
Description:	An unsigned multiplication is carried out between the contents of W and the register file location 'f'. The 16-bit result is stored in the PRODH:PRODL register pair. PRODH contains the high byte. Both W and 'f' are unchanged. None of the Status flags are affected. Note that neither Overflow nor Carry is possible in this operation. A Zero result is possible, but not detected. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write registers PRODH: PRODL

Example: MULWF REG

Before Instruction

W	=	0xC4
REG	=	0xB5
PRODH	=	?
PRODL	=	?

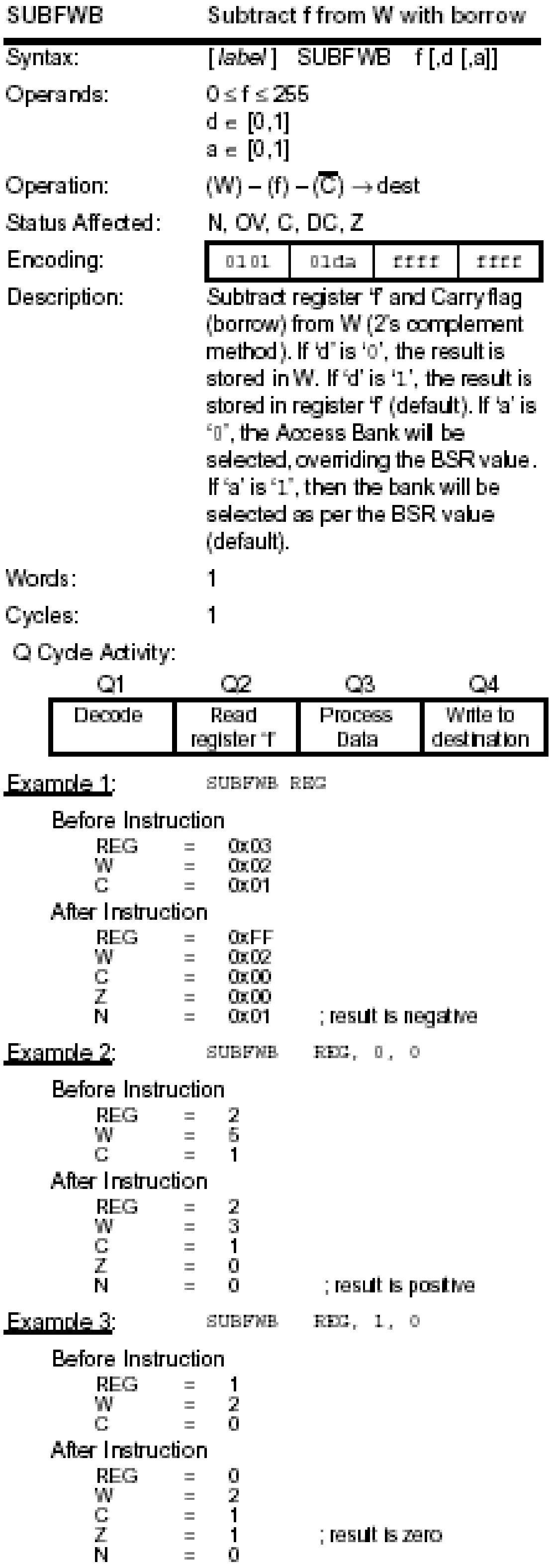
After Instruction

W	=	0xC4
REG	=	0xB5
PRODH	=	0x8A
PRODL	=	0x94

Notes:

▪

❖ Subtract f from WREG with borrow



SUBFWB **Subtract f from W with borrow**

Syntax: [*label*] SUBFWB f [,d [,a]]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: (W) – (f) – ($\overline{C}$) → dest

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

Encoding:

0101	01da	ffff	ffff

Description: Subtract register 'f' and Carry flag (borrow) from W (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: SUBFWB REG

```
Before Instruction
    REG   =   0x03
    W     =   0x02
    C     =   0x01
After Instruction
    REG   =   0xFF
    W     =   0x02
    C     =   0x00
    Z     =   0x00
    N     =   0x01       ; result is negative
```

Example 2: SUBFWB REG, 0, 0

```
Before Instruction
    REG   =   2
    W     =   5
    C     =   1
After Instruction
    REG   =   2
    W     =   3
    C     =   1
    Z     =   0
    N     =   0          ; result is positive
```

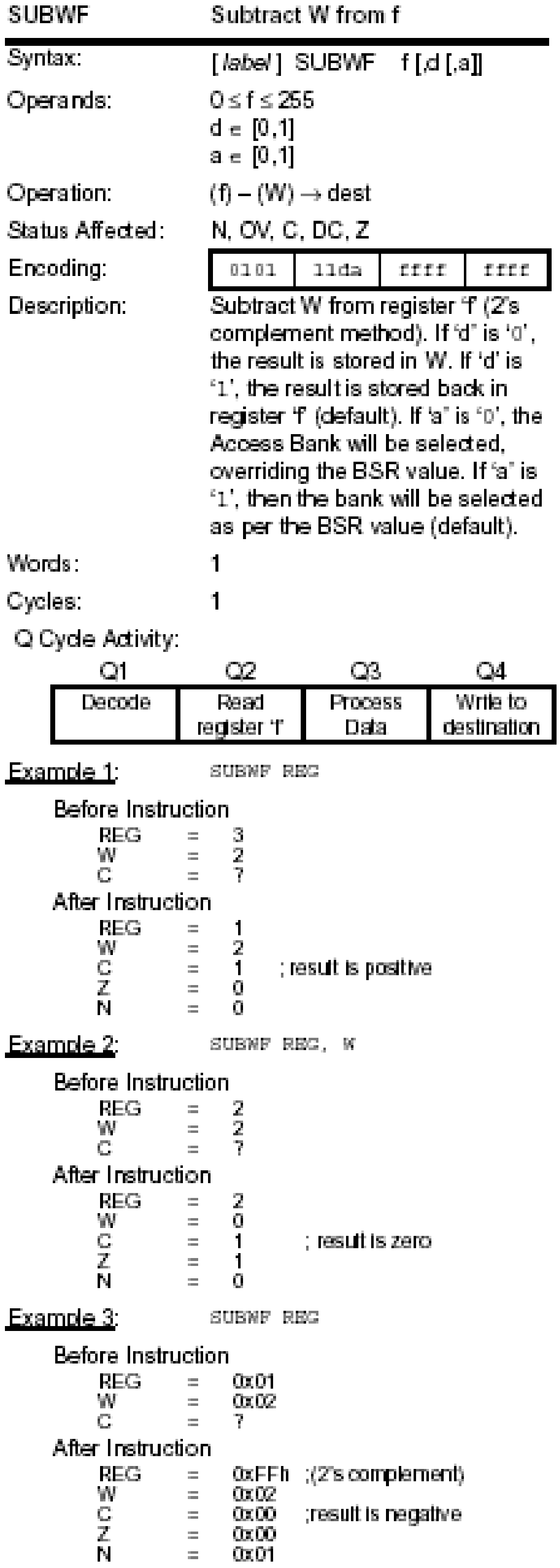
Example 3: SUBFWB REG, 1, 0

```
Before Instruction
    REG   =   1
    W     =   2
    C     =   0
After Instruction
    REG   =   0
    W     =   2
    C     =   1
    Z     =   1          ; result is zero
    N     =   0
```

Notes:

-

- Subtract WREG from f



SUBWF **Subtract W from f**

Syntax:	[*label*] SUBWF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f) – (W) → dest
Status Affected:	N, OV, C, DC, Z

Encoding:

0101	11da	ffff	ffff

Description: Subtract W from register 'f' (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBWF REG`

```
Before Instruction
    REG    =   3
    W      =   2
    C      =   ?
After Instruction
    REG    =   1
    W      =   2
    C      =   1    ; result is positive
    Z      =   0
    N      =   0
```

Example 2: `SUBWF REG, W`

```
Before Instruction
    REG    =   2
    W      =   2
    C      =   ?
After Instruction
    REG    =   2
    W      =   0
    C      =   1      ; result is zero
    Z      =   1
    N      =   0
```

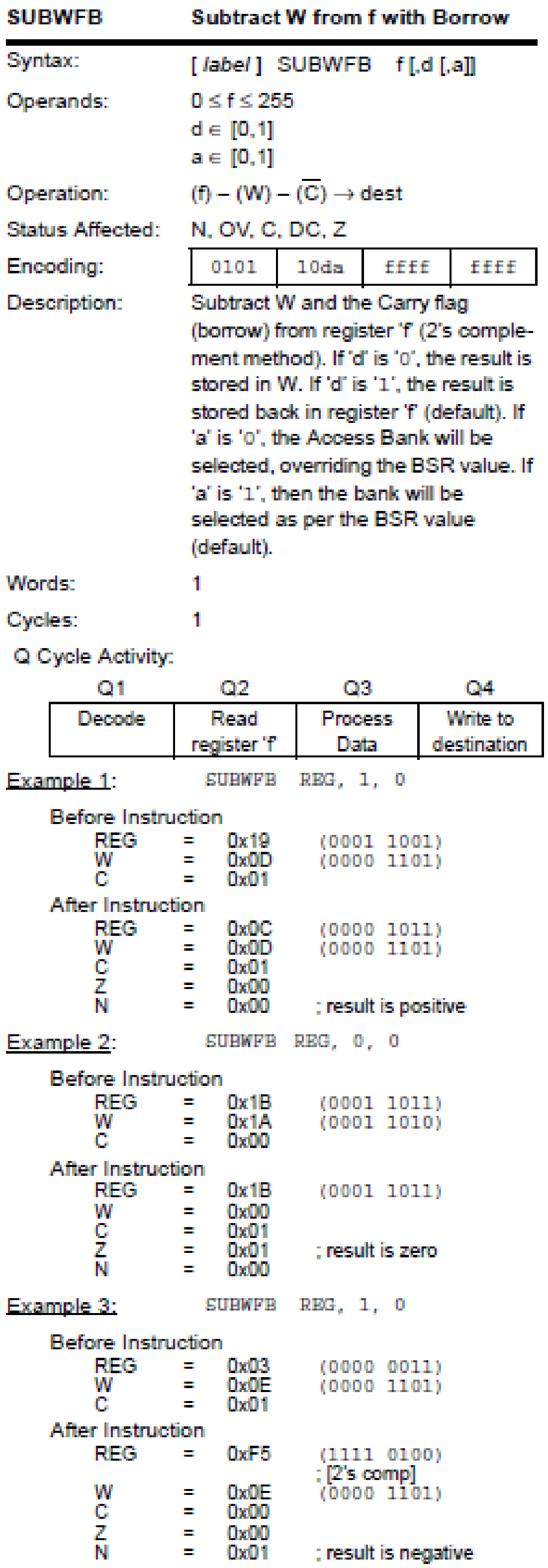
Example 3: `SUBWF REG`

```
Before Instruction
    REG    =   0x01
    W      =   0x02
    C      =   ?
After Instruction
    REG    =   0xFFh  ;(2's complement)
    W      =   0x02
    C      =   0x00   ;result is negative
    Z      =   0x00
    N      =   0x01
```

Notes:

-

❖ Subtract WREG from f with borrow



SUBWFB	**Subtract W from f with Borrow**
Syntax:	[*label*] SUBWFB f [,d [,a]]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(f) – (W) – ($\overline{C}$) → dest
Status Affected:	N, OV, C, DC, Z
Encoding:	`0101` | `10da` | `ffff` | `ffff`
Description:	Subtract W and the Carry flag (borrow) from register 'f' (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example 1: `SUBWFB REG, 1, 0`

```
Before Instruction
      REG    =    0x19     (0001 1001)
      W      =    0x0D     (0000 1101)
      C      =    0x01
After Instruction
      REG    =    0x0C     (0000 1011)
      W      =    0x0D     (0000 1101)
      C      =    0x01
      Z      =    0x00
      N      =    0x00     ; result is positive
```

Example 2: `SUBWFB REG, 0, 0`

```
Before Instruction
      REG    =    0x1B     (0001 1011)
      W      =    0x1A     (0001 1010)
      C      =    0x00
After Instruction
      REG    =    0x1B     (0001 1011)
      W      =    0x00
      C      =    0x01
      Z      =    0x01     ; result is zero
      N      =    0x00
```

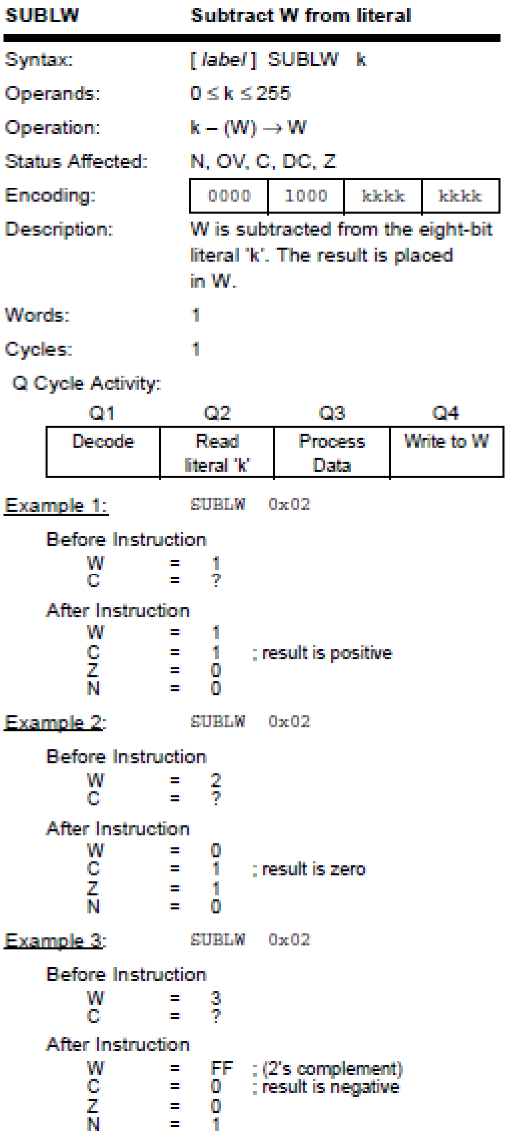
Example 3: `SUBWFB REG, 1, 0`

```
Before Instruction
      REG    =    0x03     (0000 0011)
      W      =    0x0E     (0000 1101)
      C      =    0x01
After Instruction
      REG    =    0xF5     (1111 0100)
                           ; [2's comp]
      W      =    0x0E     (0000 1101)
      C      =    0x00
      Z      =    0x00
      N      =    0x01     ; result is negative
```

Notes:

-

❖ Subtract WREG from literal



SUBLW **Subtract W from literal**

Syntax:	[*label*] SUBLW k
Operands:	0 ≤ k ≤ 255
Operation:	k – (W) → W
Status Affected:	N, OV, C, DC, Z
Encoding:	0000 1000 kkkk kkkk
Description:	W is subtracted from the eight-bit literal 'k'. The result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example 1: `SUBLW 0x02`

```
Before Instruction
    W    =  1
    C    =  ?
After Instruction
    W    =  1
    C    =  1    ; result is positive
    Z    =  0
    N    =  0
```

Example 2: `SUBLW 0x02`

```
Before Instruction
    W    =  2
    C    =  ?
After Instruction
    W    =  0
    C    =  1    ; result is zero
    Z    =  1
    N    =  0
```

Example 3: `SUBLW 0x02`

```
Before Instruction
    W    =  3
    C    =  ?
After Instruction
    W    =  FF   ; (2's complement)
    C    =  0    ; result is negative
    Z    =  0
    N    =  1
```

Notes:

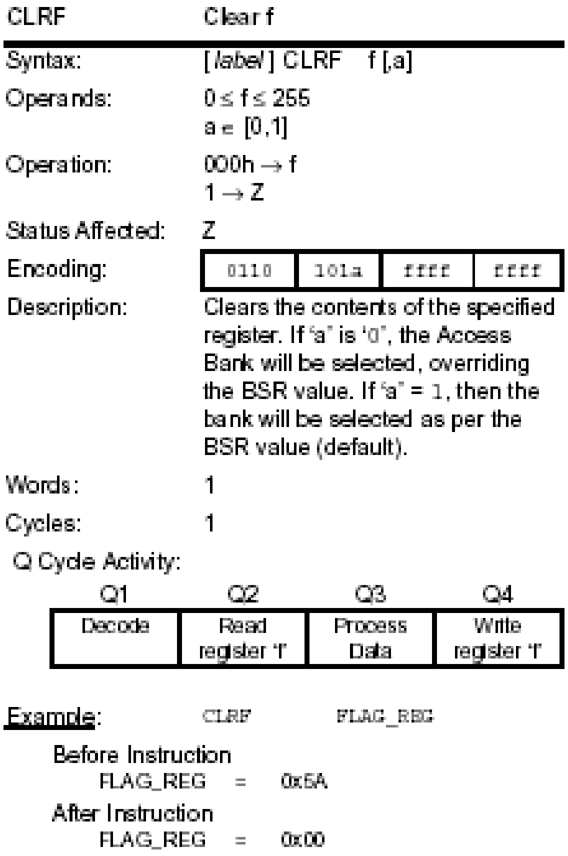
-

5.2. Move, Set and Clear Operations

Most applications require an efficient movement of data from one memory location to another. Processors in general have instructions dedicated to this type of operation. PICmicro also offers a wide range of operations to move, set and clear data as do other processors.

The remainder of this section provides detailed description of Clear, Complement, Compare, Move, Negate, Set, Table (block move) and Swap instructions.

❖ Clear f



CLRF Clear f

Syntax: [*label*] CLRF f [,a]

Operands: 0 ≤ f ≤ 255
a ∈ [0,1]

Operation: 000h → f
1 → Z

Status Affected: Z

Encoding:

0110	101a	ffff	ffff

Description: Clears the contents of the specified register. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

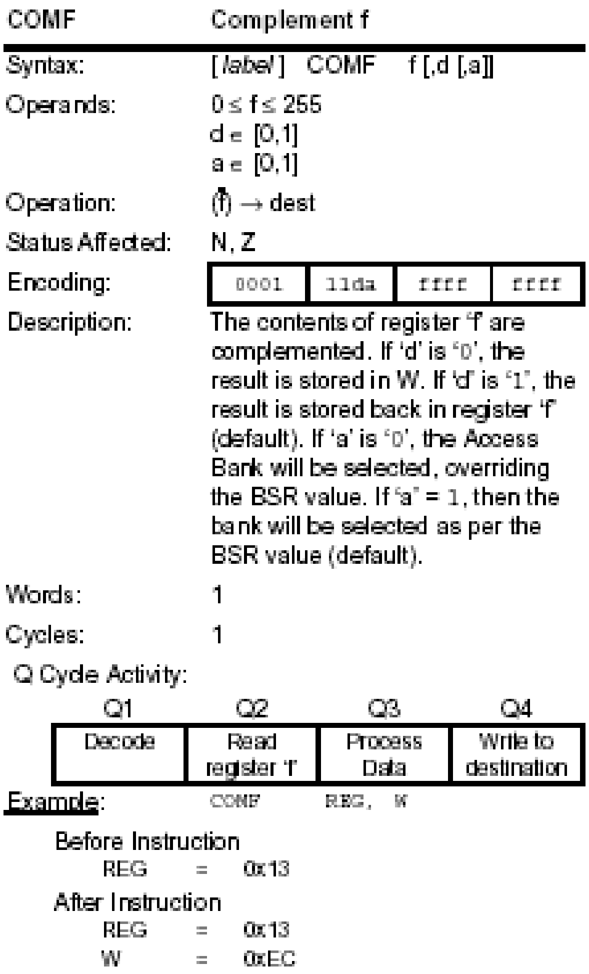
Example: CLRF FLAG_REG

Before Instruction
FLAG_REG = 0x5A
After Instruction
FLAG_REG = 0x00

Notes:

▪

- Complement f



COMF **Complement f**

Syntax:	[*label*] COMF f [,d [,a]]
Operands:	$0 \leq f \leq 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	$(\overline{f}) \rightarrow$ dest
Status Affected:	N, Z
Encoding:	`0001` | `11da` | `ffff` | `ffff`
Description:	The contents of register 'f' are complemented. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

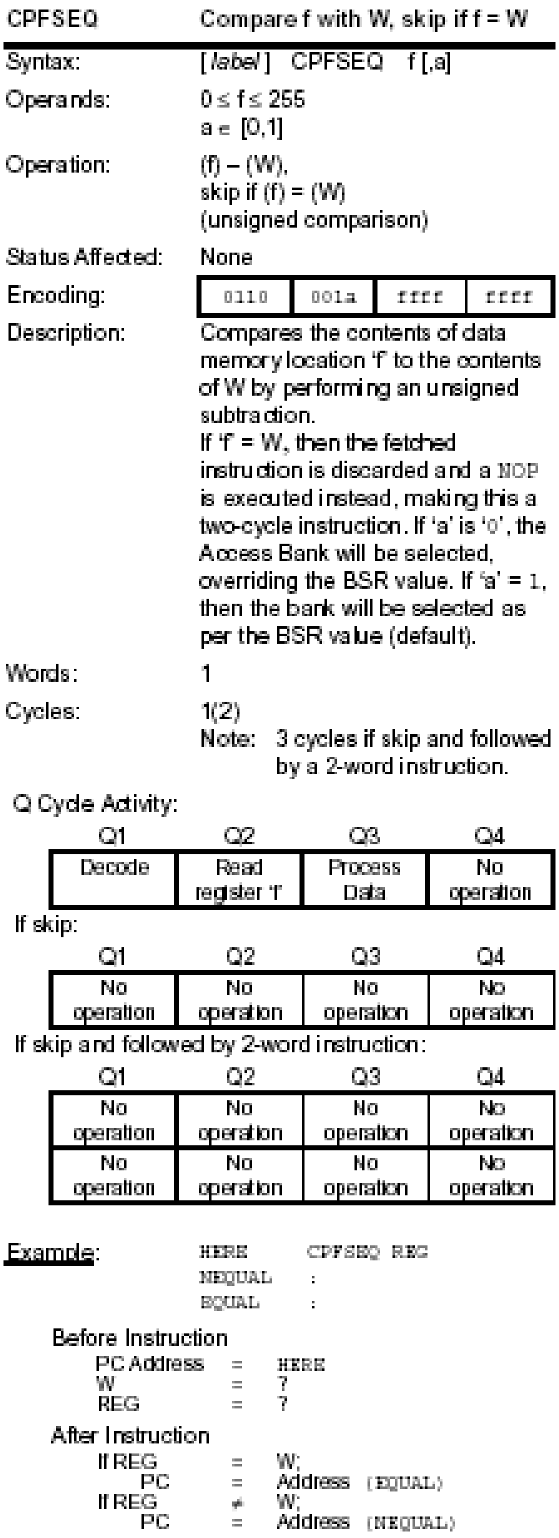
Example: `COMF REG, W`

Before Instruction
REG = 0x13
After Instruction
REG = 0x13
W = 0xEC

Notes:

▪

- Compare f with WREG, skip



CPFSEQ **Compare f with W, skip if f = W**

Syntax: [*label*] CPFSEQ f [,a]

Operands: 0 ≤ f ≤ 255
a ∈ [0,1]

Operation: (f) – (W),
skip if (f) = (W)
(unsigned comparison)

Status Affected: None

Encoding:

0110	001a	ffff	ffff

Description: Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction.
If 'f' = W, then the fetched instruction is discarded and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE      CPFSEQ REG
NEQUAL    :
EQUAL     :
```

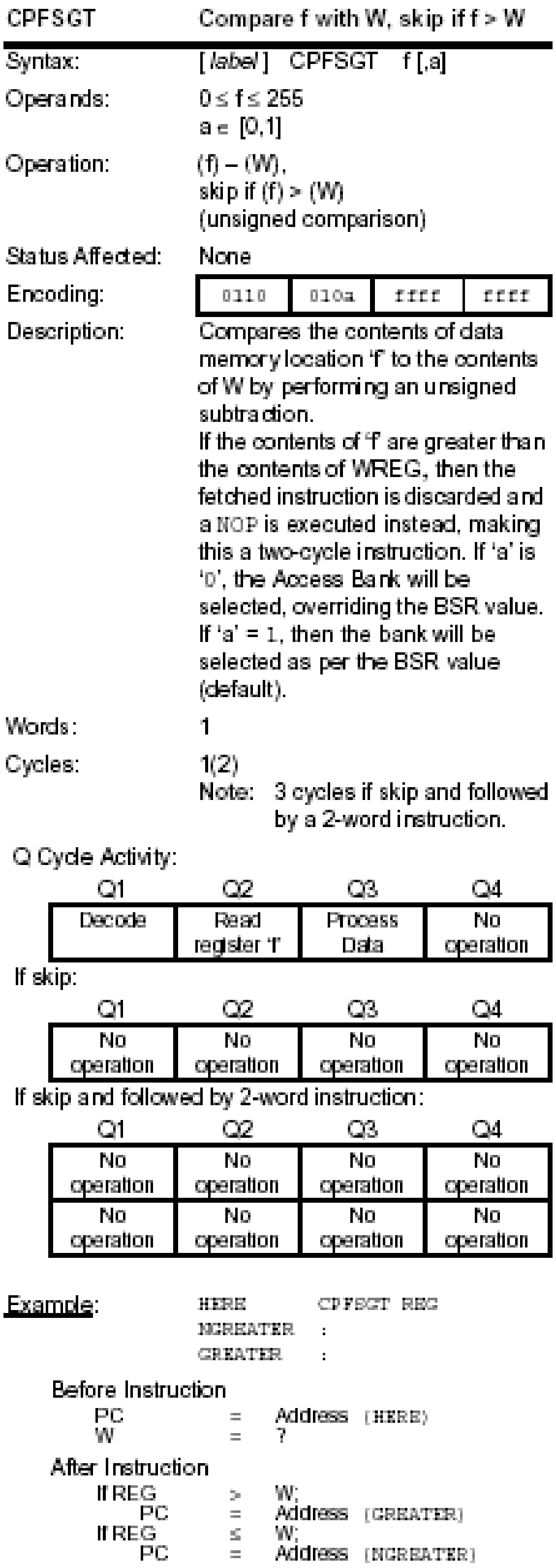
Before Instruction
PC Address = HERE
W = ?
REG = ?

After Instruction
If REG = W;
PC = Address (EQUAL)
If REG ≠ W;
PC = Address (NEQUAL)

Notes:

-

- Compare f with WREG, skip if >



CPFSGT Compare f with W, skip if f > W

Syntax: [*label*] CPFSGT f [,a]

Operands: $0 \le f \le 255$
$a \in [0,1]$

Operation: (f) – (W),
skip if (f) > (W)
(unsigned comparison)

Status Affected: None

Encoding:

0110	010a	ffff	ffff

Description: Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction.
If the contents of 'f' are greater than the contents of WREG, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

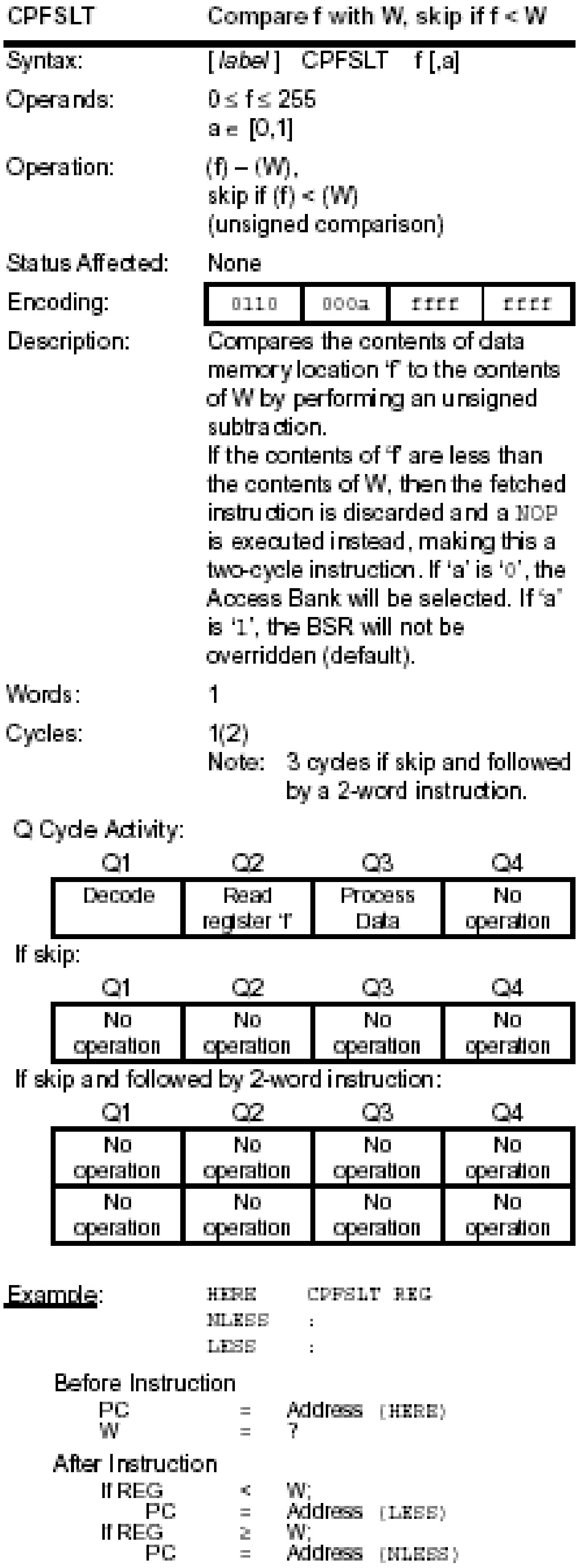
```
HERE       CPFSGT REG
NGREATER   :
GREATER    :
```

```
Before Instruction
    PC        =    Address (HERE)
    W         =    ?
After Instruction
    If REG    >    W;
       PC     =    Address (GREATER)
    If REG    ≤    W;
       PC     =    Address (NGREATER)
```

Notes:

-

- Compare f with WREG, skip if <



CPFSLT **Compare f with W, skip if f < W**

Syntax: [*label*] CPFSLT f [,a]

Operands: $0 \leq f \leq 255$
$a \in [0,1]$

Operation: (f) – (W),
skip if (f) < (W)
(unsigned comparison)

Status Affected: None

Encoding:

0110	000a	ffff	ffff

Description: Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction.
If the contents of 'f' are less than the contents of W, then the fetched instruction is discarded and a `NOP` is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR will not be overridden (default).

Words: 1

Cycles: 1(2)
Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE     CPFSLT REG
NLESS    :
LESS     :
```

Before Instruction

```
PC       =    Address (HERE)
W        =    ?
```

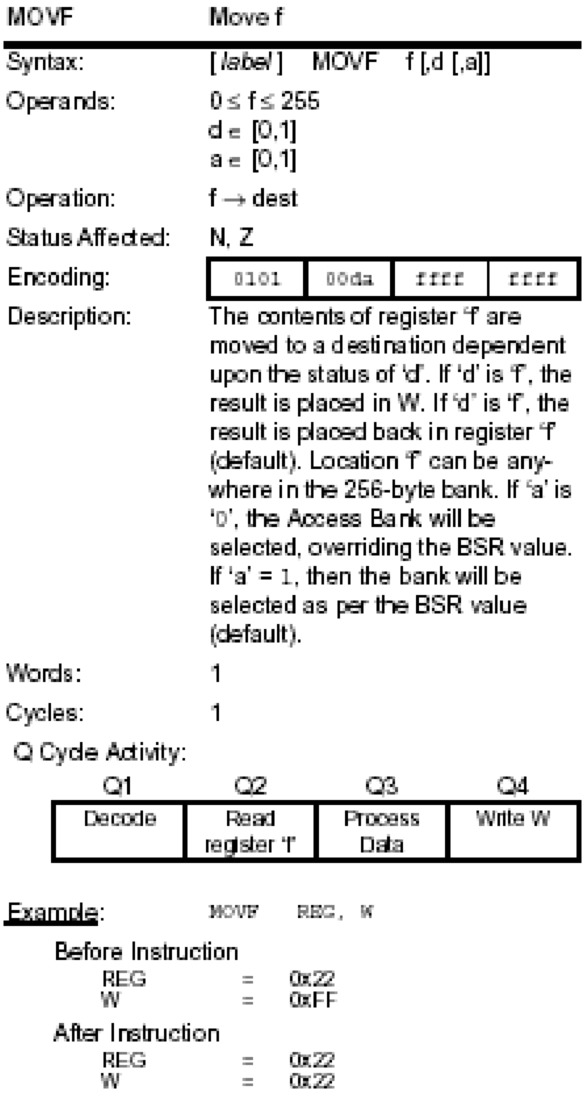
After Instruction

```
If REG   <    W;
    PC   =    Address (LESS)
If REG   ≥    W;
    PC   =    Address (NLESS)
```

Notes:

-

❖ Move f



MOVF	Move f
Syntax:	[*label*] MOVF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	f → dest
Status Affected:	N, Z
Encoding:	0101 00da ffff ffff
Description:	The contents of register 'f' are moved to a destination dependent upon the status of 'd'. If 'd' is 'f', the result is placed in W. If 'd' is 'f', the result is placed back in register 'f' (default). Location 'f' can be anywhere in the 256-byte bank. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write W

Example: `MOVF REG, W`

Before Instruction
REG = 0x22
W = 0xFF

After Instruction
REG = 0x22
W = 0x22

Notes:

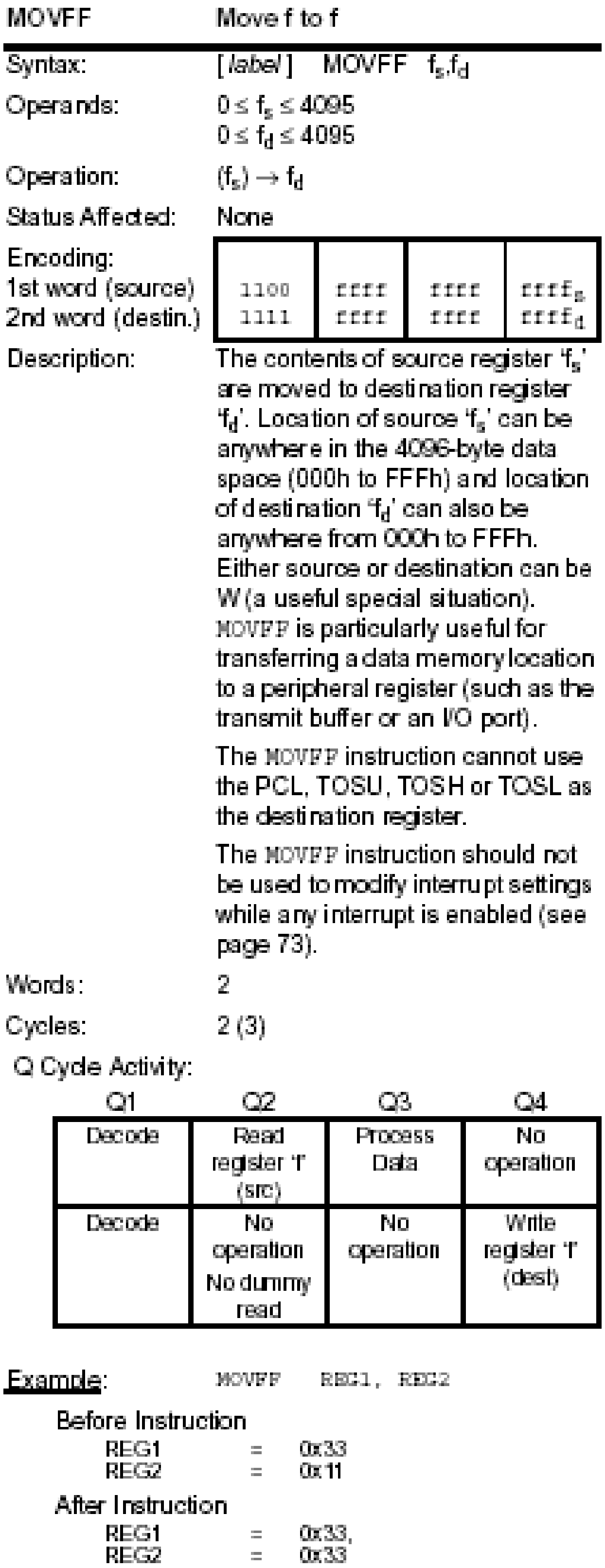
Selecting the destination register:

// If d=0, the value in register 0x21 is placed in W register
“(W) ← (0x21)”
MOVF 0x21, 0

// If d=1, the value in register 0x21 is placed back in the same register
“(0x21) ← (0x21)””
MOVF 0x21, 1

// If d is not specified, it defaults to 1, so the value in register 0x21 is placed back in the same register
“(0x21) ← (0x21)””
MOVF 0x21

- Move fs (source) to 1st word, fd (destination) 2nd word



MOVFF Move f to f

Syntax: [*label*] MOVFF f_s,f_d

Operands: $0 \le f_s \le 4095$
$0 \le f_d \le 4095$

Operation: $(f_s) \rightarrow f_d$

Status Affected: None

Encoding:

1st word (source)	1100	ffff	ffff	ffff$_s$
2nd word (destin.)	1111	ffff	ffff	ffff$_d$

Description: The contents of source register 'f_s' are moved to destination register 'f_d'. Location of source 'f_s' can be anywhere in the 4096-byte data space (000h to FFFh) and location of destination 'f_d' can also be anywhere from 000h to FFFh. Either source or destination can be W (a useful special situation). MOVFF is particularly useful for transferring a data memory location to a peripheral register (such as the transmit buffer or an I/O port).

The MOVFF instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.

The MOVFF instruction should not be used to modify interrupt settings while any interrupt is enabled (see page 73).

Words: 2

Cycles: 2 (3)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f' (src)	Process Data	No operation
Decode	No operation No dummy read	No operation	Write register 'f' (dest)

Example: MOVFF REG1, REG2

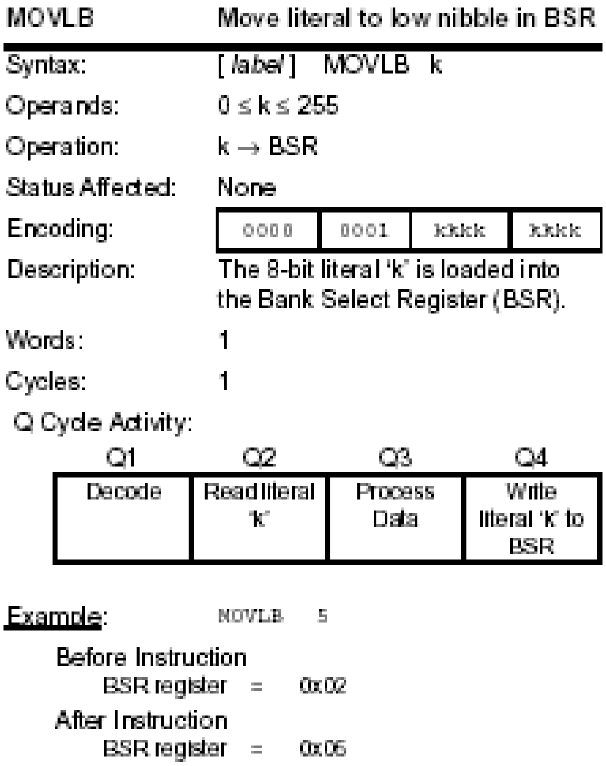
Before Instruction
REG1 = 0x33
REG2 = 0x11

After Instruction
REG1 = 0x33,
REG2 = 0x33

Notes:

-

❖ Move literal to BSR<3:0>



MOVLB Move literal to low nibble in BSR

Syntax:	[*label*] MOVLB k
Operands:	0 ≤ k ≤ 255
Operation:	k → BSR
Status Affected:	None
Encoding:	0000 0001 kkkk kkkk
Description:	The 8-bit literal 'k' is loaded into the Bank Select Register (BSR).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write literal 'k' to BSR

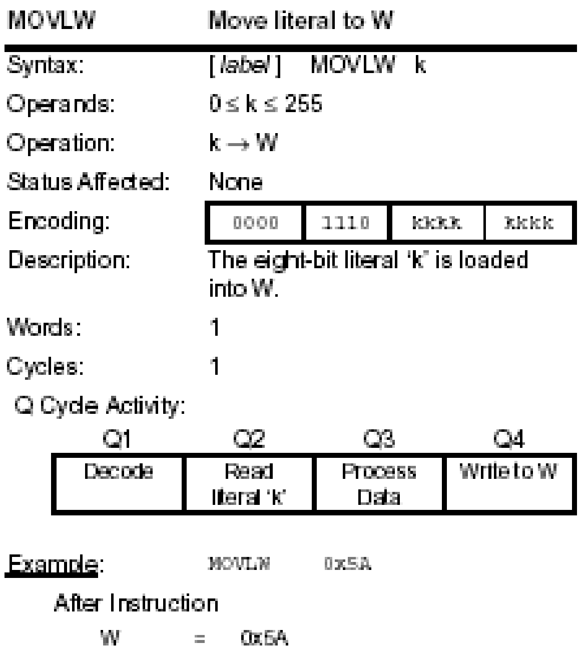
Example: MOVLB 5

Before Instruction
BSR register = 0x02
After Instruction
BSR register = 0x05

Notes:

▪

❖ Move literal to WREG



MOVLW Move literal to W

Syntax:	[*label*] MOVLW k
Operands:	0 ≤ k ≤ 255
Operation:	k → W
Status Affected:	None
Encoding:	0000 1110 kkkk kkkk
Description:	The eight-bit literal 'k' is loaded into W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

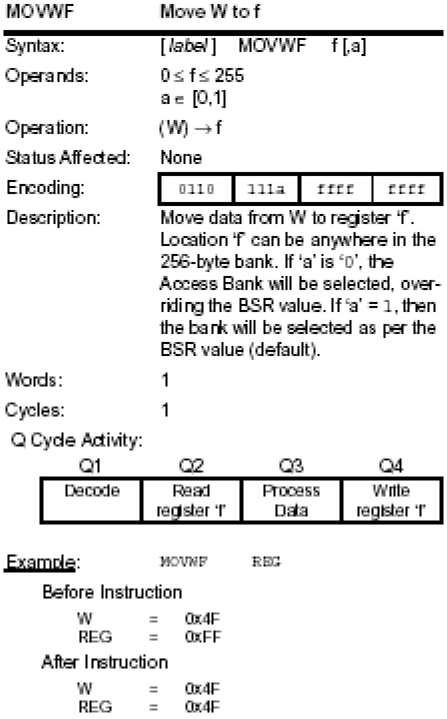
Example: MOVLW 0x5A

After Instruction
W = 0x5A

Notes:

▪

❖ Move WREG to f



MOVWF **Move W to f**

Syntax: [*label*] MOVWF f [,a]

Operands: $0 \leq f \leq 255$
$a \in [0,1]$

Operation: $(W) \rightarrow f$

Status Affected: None

Encoding:

0110	111a	ffff	ffff

Description: Move data from W to register 'f'. Location 'f' can be anywhere in the 256-byte bank. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: MOVWF REG

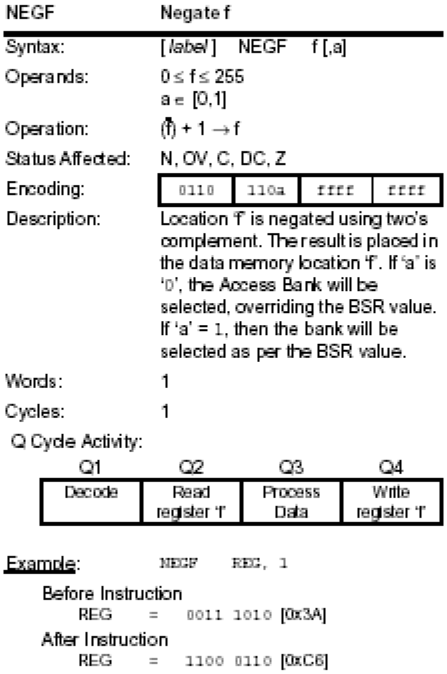
Before Instruction
W = 0x4F
REG = 0xFF

After Instruction
W = 0x4F
REG = 0x4F

Notes:

▪

❖ Negate f



NEGF **Negate f**

Syntax: [*label*] NEGF f [,a]

Operands: $0 \leq f \leq 255$
$a \in [0,1]$

Operation: $(\overline{f}) + 1 \rightarrow f$

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

Encoding:

0110	110a	ffff	ffff

Description: Location 'f' is negated using two's complement. The result is placed in the data memory location 'f'. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: NEGF REG, 1

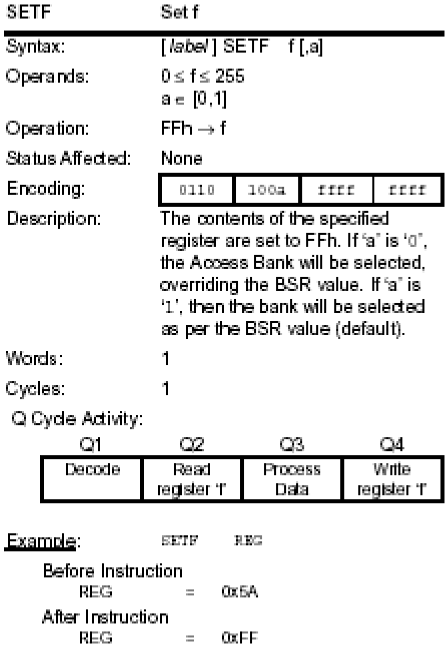
Before Instruction
REG = 0011 1010 [0x3A]

After Instruction
REG = 1100 0110 [0xC6]

Notes:

▪

- Set f



SETF	Set f
Syntax:	[*label*] SETF f [,a]
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	FFh → f
Status Affected:	None
Encoding:	0110 | 100a | ffff | ffff
Description:	The contents of the specified register are set to FFh. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: SETF REG

Before Instruction
REG = 0x5A

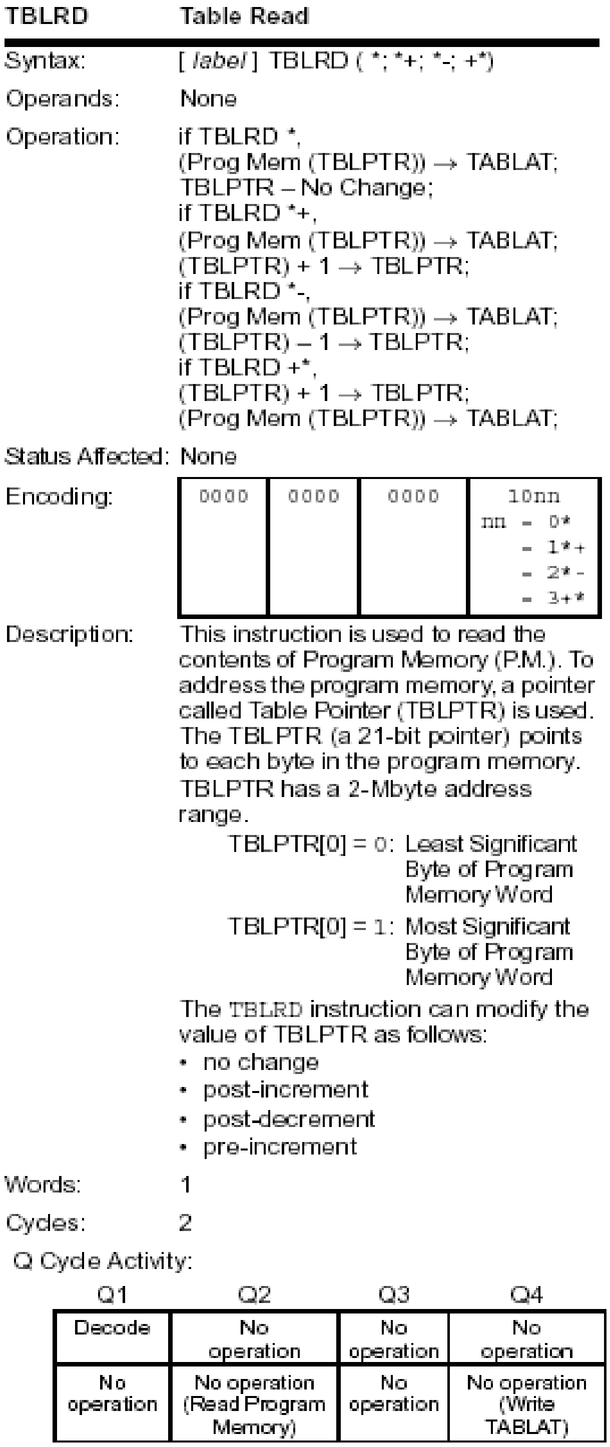
After Instruction
REG = 0xFF

Notes:

-

❖ Table Read "TBLRD"

The **Memory-Block Transfer** reads and/or writes to a range of memory locations. The following two bullets show all the various options for table read "TBLRD" and table write "TBLWT".



TBLRD Table Read

Syntax: [*label*] TBLRD (*; *+; *-; +*)

Operands: None

Operation:

```
if TBLRD *,
(Prog Mem (TBLPTR)) → TABLAT;
TBLPTR – No Change;
if TBLRD *+,
(Prog Mem (TBLPTR)) → TABLAT;
(TBLPTR) + 1 → TBLPTR;
if TBLRD *-,
(Prog Mem (TBLPTR)) → TABLAT;
(TBLPTR) – 1 → TBLPTR;
if TBLRD +*,
(TBLPTR) + 1 → TBLPTR;
(Prog Mem (TBLPTR)) → TABLAT;
```

Status Affected: None

Encoding:

0000	0000	0000	10nn nn = 0* = 1*+ = 2*- = 3+*

Description: This instruction is used to read the contents of Program Memory (P.M.). To address the program memory, a pointer called Table Pointer (TBLPTR) is used. The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-Mbyte address range.

TBLPTR[0] = 0: Least Significant Byte of Program Memory Word

TBLPTR[0] = 1: Most Significant Byte of Program Memory Word

The TBLRD instruction can modify the value of TBLPTR as follows:

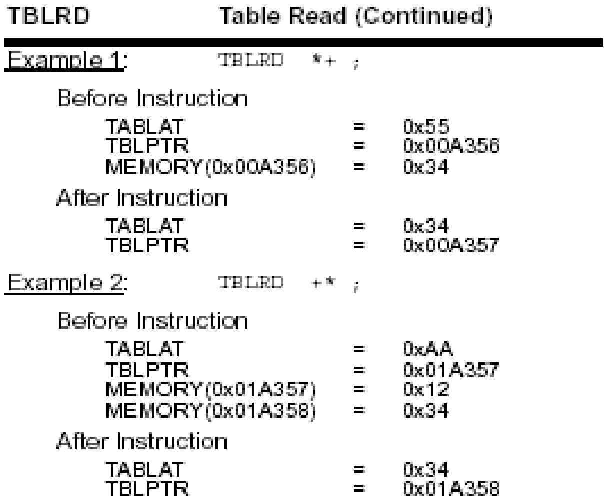
- no change
- post-increment
- post-decrement
- pre-increment

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation
No operation	No operation (Read Program Memory)	No operation	No operation (Write TABLAT)



TBLRD Table Read (Continued)

Example 1: TBLRD *+ ;

Before Instruction

TABLAT	=	0x55
TBLPTR	=	0x00A356
MEMORY(0x00A356)	=	0x34

After Instruction

TABLAT	=	0x34
TBLPTR	=	0x00A357

Example 2: TBLRD +* ;

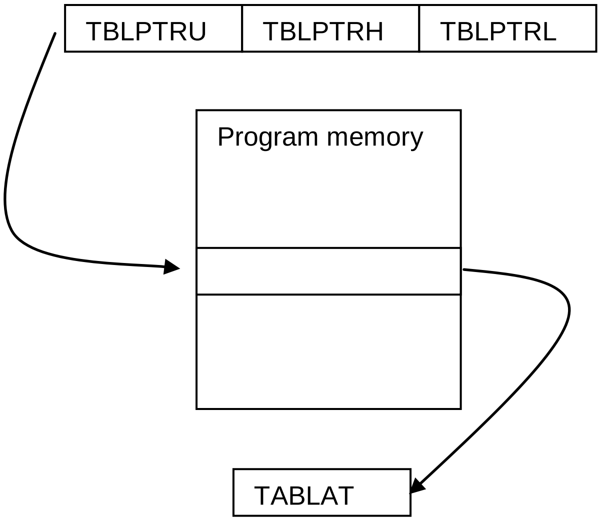
Before Instruction

TABLAT	=	0xAA
TBLPTR	=	0x01A357
MEMORY(0x01A357)	=	0x12
MEMORY(0x01A358)	=	0x34

After Instruction

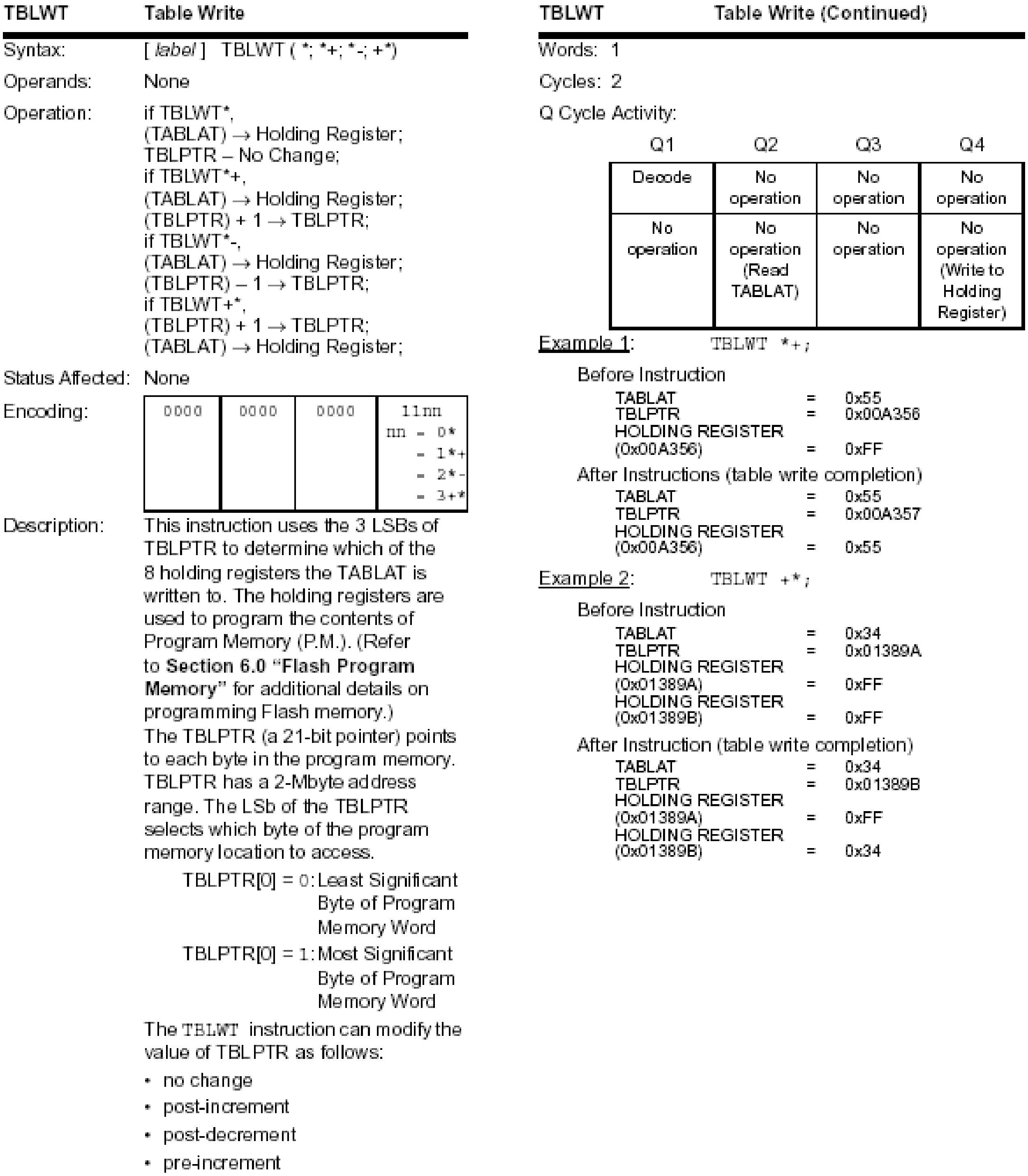
TABLAT	=	0x34
TBLPTR	=	0x01A358

Note:
"TBLPTR" value is stored in three registers:



After executing of TBLRD instruction, the content of TBLPTR location is stored in TABLAT

- Table Write "TBLWT"
  TBLWT instruction performs the reverse of the TBLRD instruction by moving the content of register TABLAT to the location pointed to by the TBLPTR in memory.



TBLWT **Table Write**

Syntax: [*label*] TBLWT (*; *+; *-; +*)

Operands: None

Operation:
if TBLWT*,
(TABLAT) → Holding Register;
TBLPTR – No Change;
if TBLWT*+,
(TABLAT) → Holding Register;
(TBLPTR) + 1 → TBLPTR;
if TBLWT*-,
(TABLAT) → Holding Register;
(TBLPTR) – 1 → TBLPTR;
if TBLWT+*,
(TBLPTR) + 1 → TBLPTR;
(TABLAT) → Holding Register;

Status Affected: None

Encoding:

0000	0000	0000	11nn nn = 0* = 1*+ = 2*- = 3+*

Description: This instruction uses the 3 LSBs of TBLPTR to determine which of the 8 holding registers the TABLAT is written to. The holding registers are used to program the contents of Program Memory (P.M.). (Refer to **Section 6.0 "Flash Program Memory"** for additional details on programming Flash memory.)

The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-Mbyte address range. The LSb of the TBLPTR selects which byte of the program memory location to access.

TBLPTR[0] = 0: Least Significant Byte of Program Memory Word

TBLPTR[0] = 1: Most Significant Byte of Program Memory Word

The TBLWT instruction can modify the value of TBLPTR as follows:

- no change
- post-increment
- post-decrement
- pre-increment

TBLWT **Table Write (Continued)**

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation
No operation	No operation (Read TABLAT)	No operation	No operation (Write to Holding Register)

Example 1: TBLWT *+;

Before Instruction

TABLAT	=	0x55
TBLPTR	=	0x00A356
HOLDING REGISTER (0x00A356)	=	0xFF

After Instructions (table write completion)

TABLAT	=	0x55
TBLPTR	=	0x00A357
HOLDING REGISTER (0x00A356)	=	0x55

Example 2: TBLWT +*;

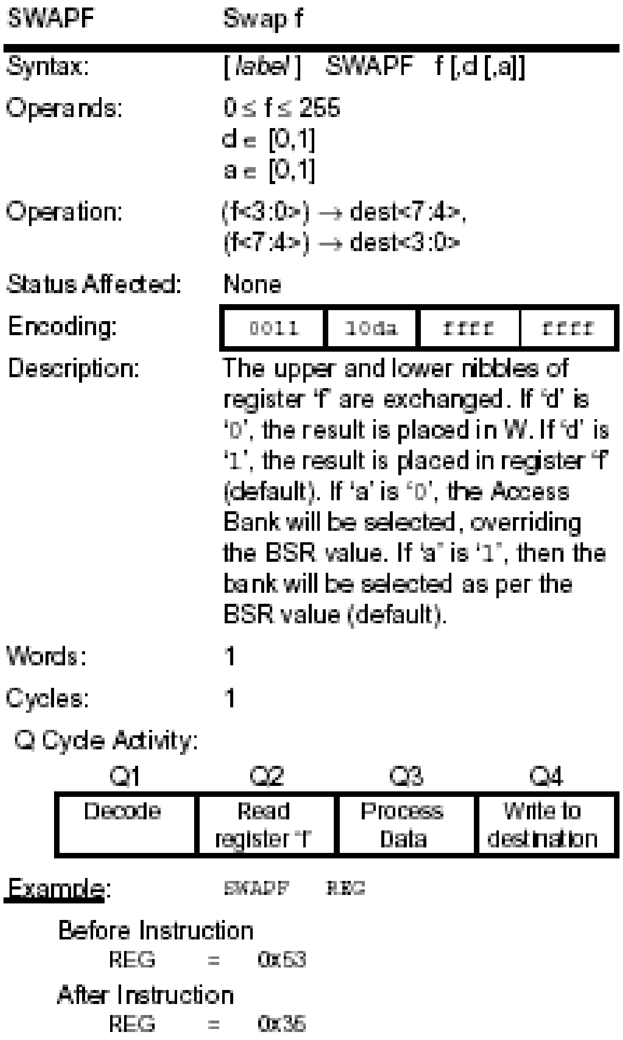
Before Instruction

TABLAT	=	0x34
TBLPTR	=	0x01389A
HOLDING REGISTER (0x01389A)	=	0xFF
HOLDING REGISTER (0x01389B)	=	0xFF

After Instruction (table write completion)

TABLAT	=	0x34
TBLPTR	=	0x01389B
HOLDING REGISTER (0x01389A)	=	0xFF
HOLDING REGISTER (0x01389B)	=	0x34

❖ Swap nibbles in f



SWAPF Swap f

Syntax: [*label*] SWAPF f [,d [,a]]

Operands: $0 \le f \le 255$
$d \in [0,1]$
$a \in [0,1]$

Operation: $(f<3:0>) \rightarrow dest<7:4>$,
$(f<7:4>) \rightarrow dest<3:0>$

Status Affected: None

Encoding:

0011	10da	ffff	ffff

Description: The upper and lower nibbles of register 'f' are exchanged. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `SWAPF REG`

Before Instruction
REG = 0x53

After Instruction
REG = 0x35

Notes:

▪

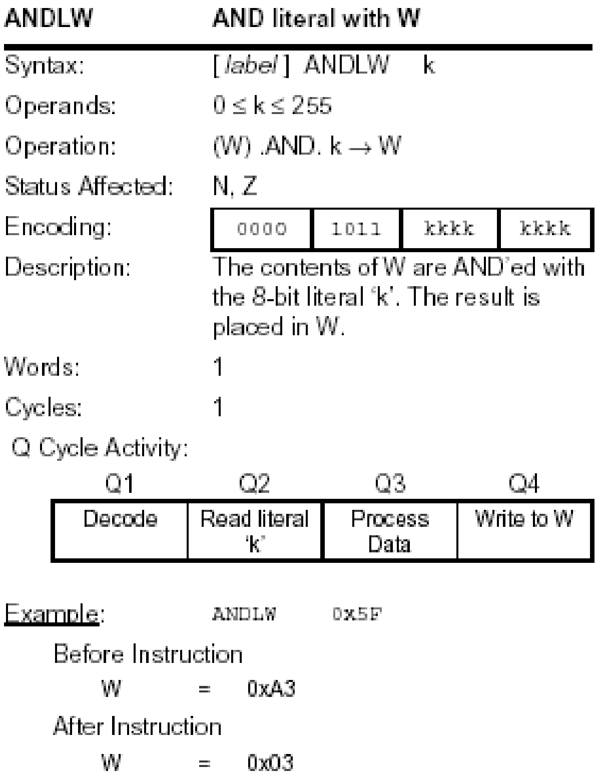
5.3. Logical Operation

Programs require the ability to test for validity of certain conditions based on the last operation executed or the contents of two memory locations, so processors provide a set of logical instructions that may be used to test validity of certain conditions.

PICmicro also offers a wide range of logical operations. These operations are used for modifying data as well as setting flags based on the results. These flags can be used later for decision making.

The remainder of this section will describe in detail the logical operation of AND, Bit Set/Clear/Test, OR, XOR, Rotate and Test for PICmirco.

- ❖ AND Literal with WREG



ANDLW **AND literal with W**

Syntax:	[*label*] ANDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) .AND. k → W
Status Affected:	N, Z
Encoding:	0000 | 1011 | kkkk | kkkk
Description:	The contents of W are AND'ed with the 8-bit literal 'k'. The result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `ANDLW 0x5F`

Before Instruction

W = 0xA3

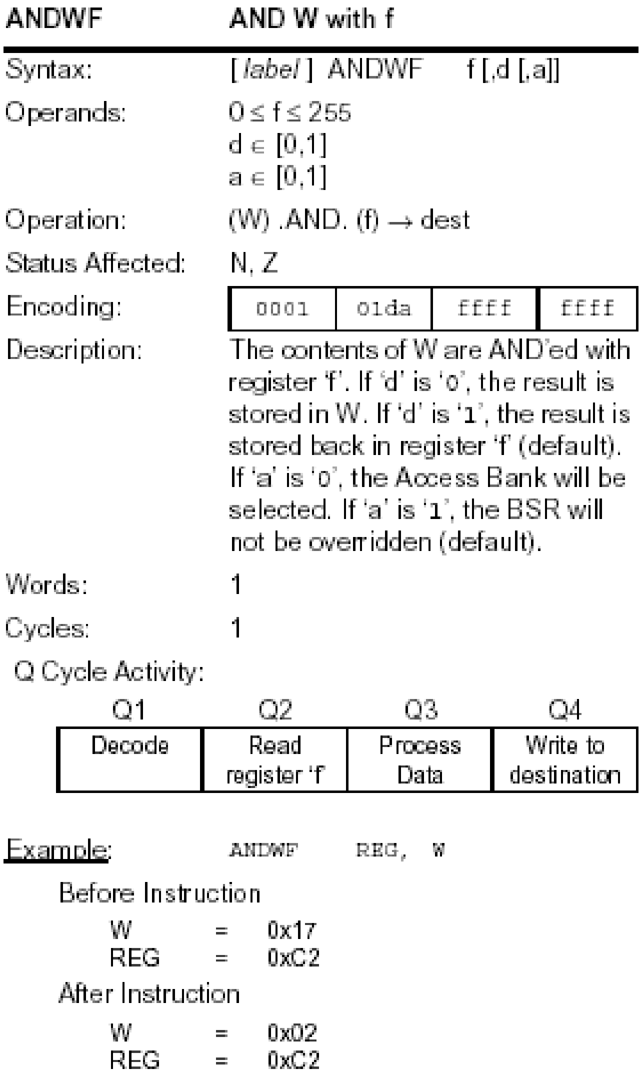
After Instruction

W = 0x03

Notes:

-

❖ AND WREG with f



ANDWF **AND W with f**

Syntax:	[*label*] ANDWF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) .AND. (f) → dest
Status Affected:	N, Z
Encoding:	0001 | 01da | ffff | ffff
Description:	The contents of W are AND'ed with register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR will not be overridden (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `ANDWF REG, W`

Before Instruction

W	=	0x17
REG	=	0xC2

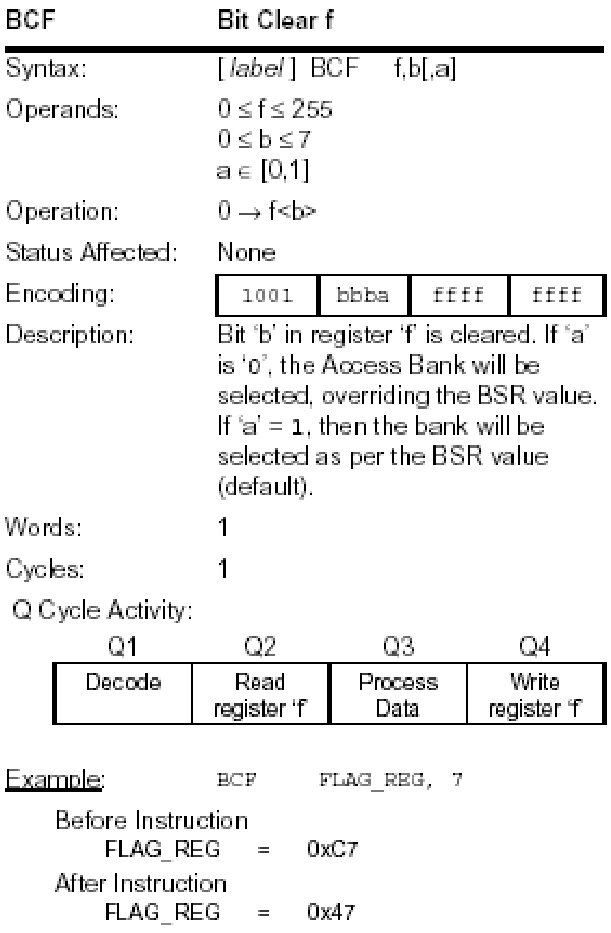
After Instruction

W	=	0x02
REG	=	0xC2

Notes:

▪

- Bit Clear f



BCF **Bit Clear f**

Syntax: [*label*] BCF f,b[,a]

Operands: 0 ≤ f ≤ 255
0 ≤ b ≤ 7
a ∈ [0,1]

Operation: 0 → f<b>

Status Affected: None

Encoding:

1001	bbba	ffff	ffff

Description: Bit 'b' in register 'f' is cleared. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

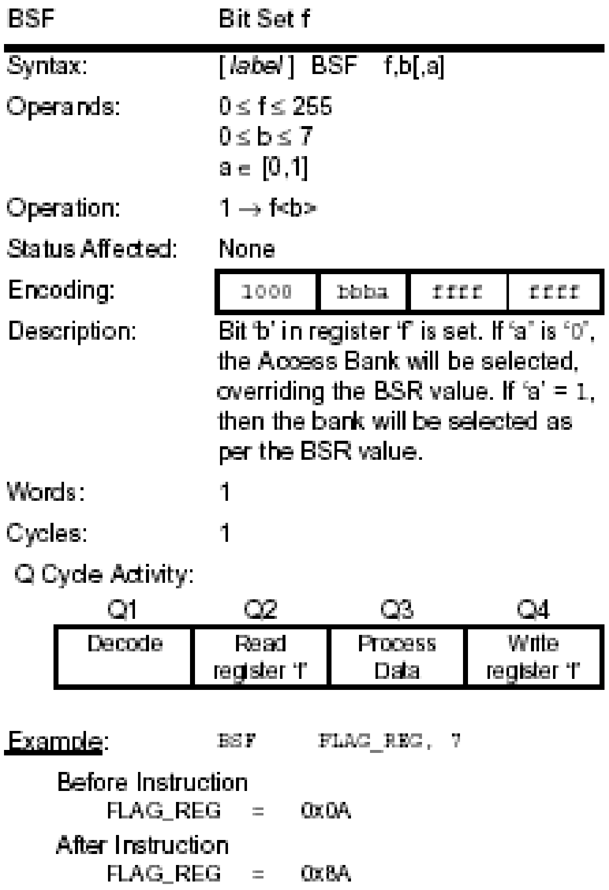
Example: `BCF FLAG_REG, 7`

Before Instruction
FLAG_REG = 0xC7
After Instruction
FLAG_REG = 0x47

Notes:

-

❖ Bit Set f



BSF	Bit Set f
Syntax:	[*label*] BSF f,b[,a]
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	1 → f<b>
Status Affected:	None
Encoding:	1000 bbba ffff ffff
Description:	Bit 'b' in register 'f' is set. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

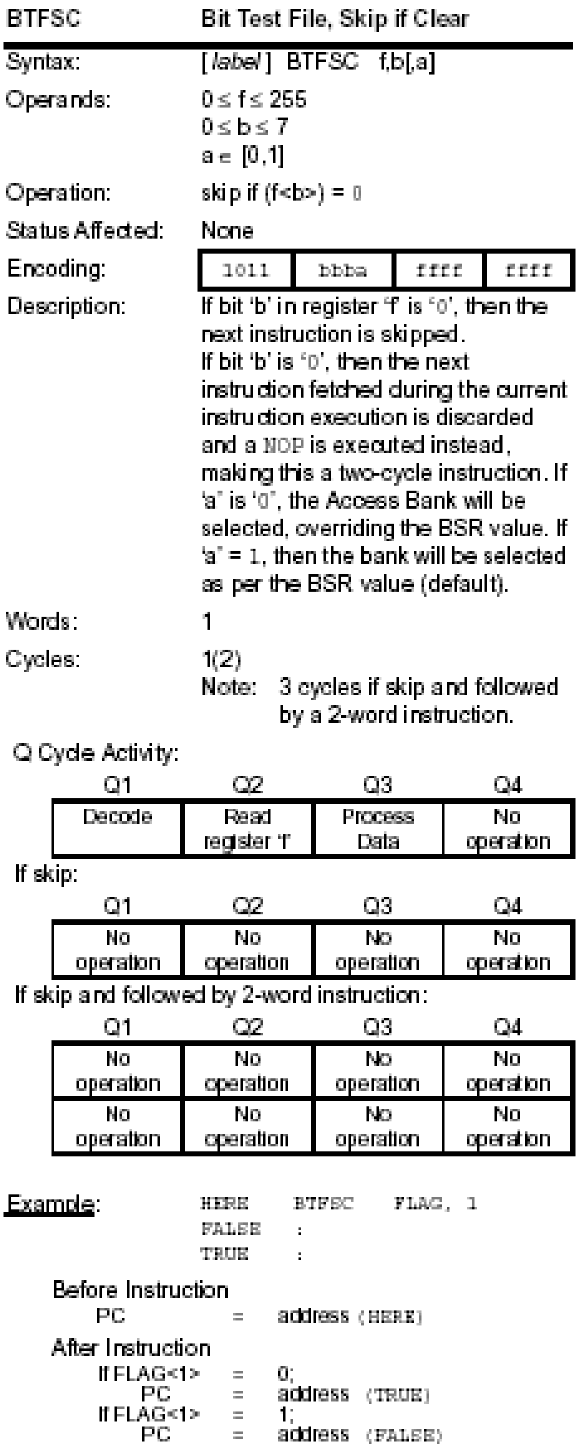
Example: `BSF FLAG_REG, 7`

Before Instruction
FLAG_REG = 0x0A
After Instruction
FLAG_REG = 0x8A

Notes:

-

❖ Bit Test f, Skip if Clear



BTFSC **Bit Test File, Skip if Clear**

Syntax:	[*label*] BTFSC f,b[,a]
Operands:	0 ≤ f ≤ 255 0 ≤ b ≤ 7 a ∈ [0,1]
Operation:	skip if (f<b>) = 0
Status Affected:	None
Encoding:	1011 | bbba | ffff | ffff
Description:	If bit 'b' in register 'f' is '0', then the next instruction is skipped. If bit 'b' is '0', then the next instruction fetched during the current instruction execution is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1(2) Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    BTFSC   FLAG, 1
FALSE   :
TRUE    :
```

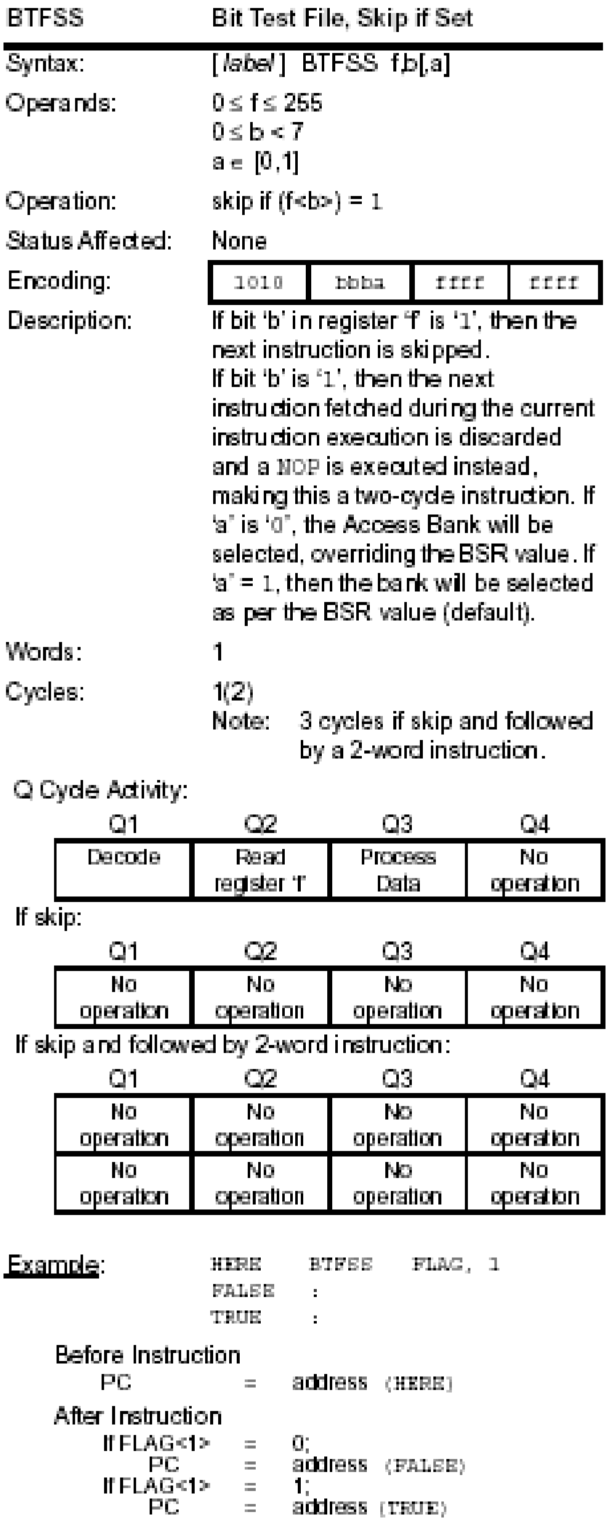
Before Instruction
PC = address (HERE)

After Instruction
If FLAG<1> = 0;
PC = address (TRUE)
If FLAG<1> = 1;
PC = address (FALSE)

Notes:

▪

- Bit Test f, Skip if Set



BTFSS	Bit Test File, Skip if Set
Syntax:	[*label*] BTFSS f,b[,a]
Operands:	0 ≤ f ≤ 255 0 ≤ b < 7 a ∈ [0,1]
Operation:	skip if (f<b>) = 1
Status Affected:	None
Encoding:	1010 bbba ffff ffff
Description:	If bit 'b' in register 'f' is '1', then the next instruction is skipped. If bit 'b' is '1', then the next instruction fetched during the current instruction execution is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1(2) Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE    BTFSS   FLAG, 1
FALSE   :
TRUE    :
```

Before Instruction

```
    PC          =   address (HERE)
```

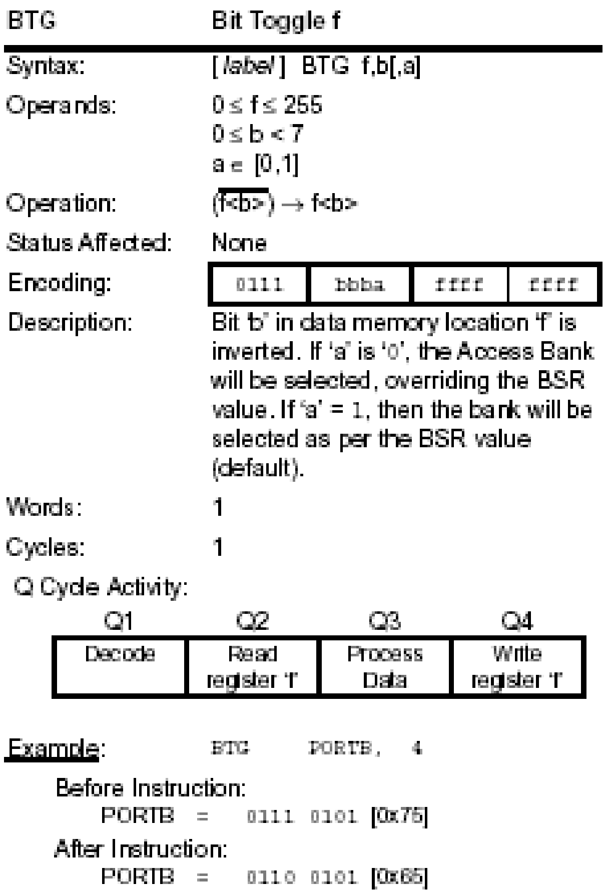
After Instruction

```
    If FLAG<1>  =   0;
       PC       =   address (FALSE)
    If FLAG<1>  =   1;
       PC       =   address (TRUE)
```

Notes:

▪

- Bit Toggle f



BTG	Bit Toggle f
Syntax:	[*label*] BTG f,b[,a]
Operands:	0 ≤ f ≤ 255 0 ≤ b < 7 a ∈ [0,1]
Operation:	$(\overline{f<b>}) \rightarrow f<b>$
Status Affected:	None
Encoding:	0111 bbba ffff ffff
Description:	Bit 'b' in data memory location 'f' is inverted. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write register 'f'

Example: BTG PORTB, 4

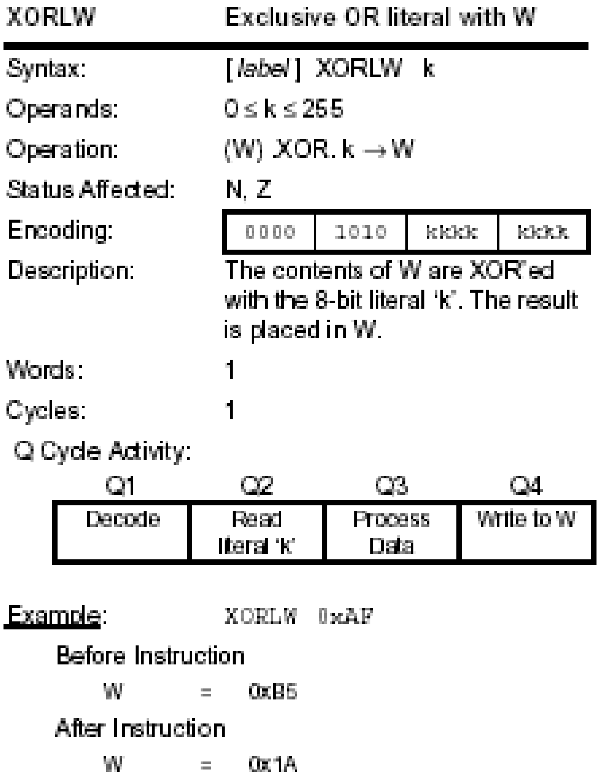
Before Instruction:
PORTB = 0111 0101 [0x75]

After Instruction:
PORTB = 0110 0101 [0x65]

Notes:

▪

- Exclusive OR literal with WREG



XORLW **Exclusive OR literal with W**

Syntax: [*label*] XORLW k

Operands: $0 \le k \le 255$

Operation: (W) .XOR. k → W

Status Affected: N, Z

Encoding:

0000	1010	kkkk	kkkk

Description: The contents of W are XOR'ed with the 8-bit literal 'k'. The result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `XORLW 0xAF`

Before Instruction

W = 0xB5

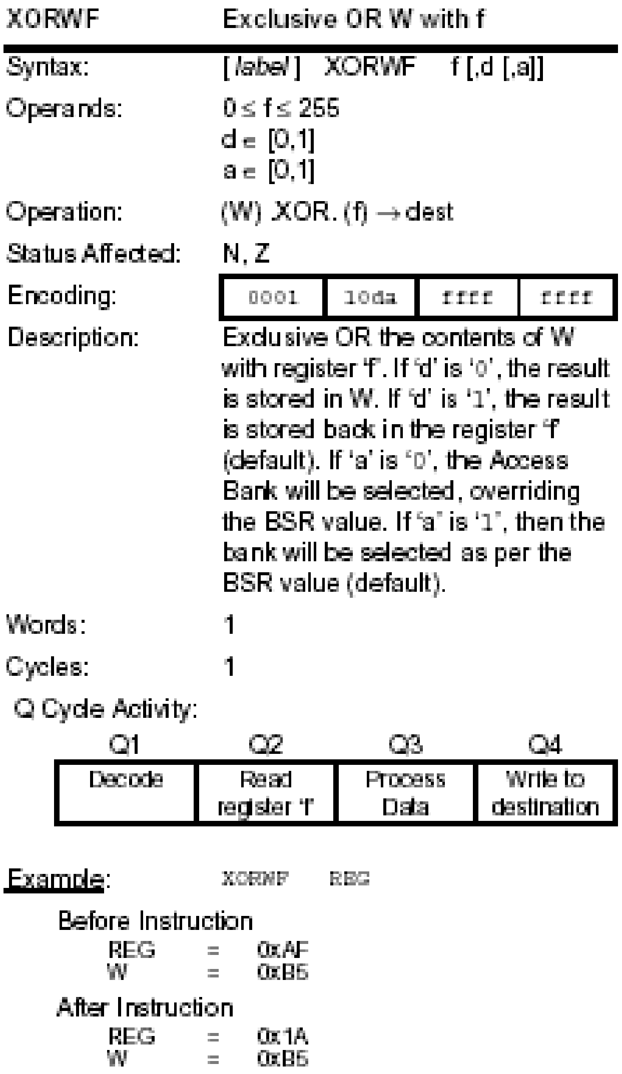
After Instruction

W = 0x1A

Notes:

-

- Exclusive OR WREG with f



XORWF	Exclusive OR W with f
Syntax:	[*label*] XORWF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) .XOR. (f) → dest
Status Affected:	N, Z
Encoding:	0001 10da ffff ffff
Description:	Exclusive OR the contents of W with register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in the register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: XORWF REG

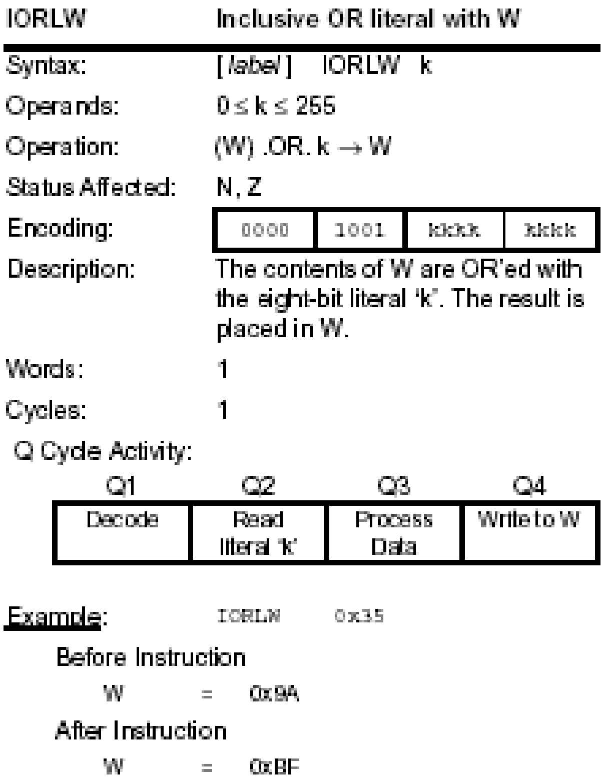
Before Instruction
REG = 0xAF
W = 0xB5

After Instruction
REG = 0x1A
W = 0xB5

Notes:

▪

- Inclusive OR literal with WREG



IORLW **Inclusive OR literal with W**

Syntax: [*label*] IORLW k

Operands: $0 \le k \le 255$

Operation: (W) .OR. k → W

Status Affected: N, Z

Encoding:

0000	1001	kkkk	kkkk

Description: The contents of W are OR'ed with the eight-bit literal 'k'. The result is placed in W.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

Example: `IORLW 0x35`

Before Instruction

W = 0x9A

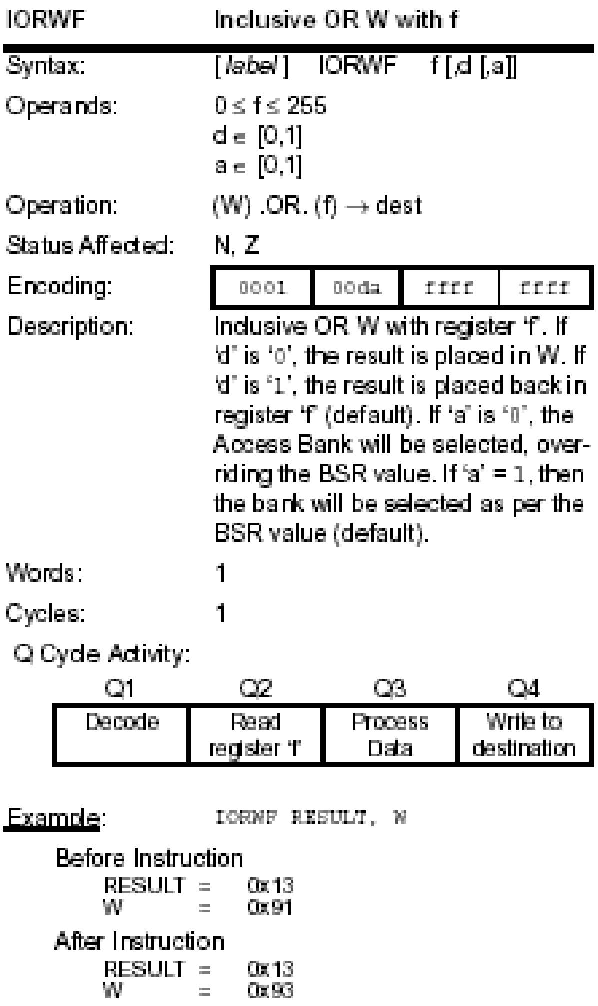
After Instruction

W = 0xBF

Notes:

▪

❖ Inclusive OR WREG with f



IORWF	Inclusive OR W with f
Syntax:	[*label*] IORWF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(W) .OR. (f) → dest
Status Affected:	N, Z
Encoding:	0001 00da ffff ffff
Description:	Inclusive OR W with register 'f'. If 'd" is '0', the result is placed in W. If 'd" is '1', the result is placed back in register 'f' (default). If 'a' is '0", the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: IORWF RESULT, W

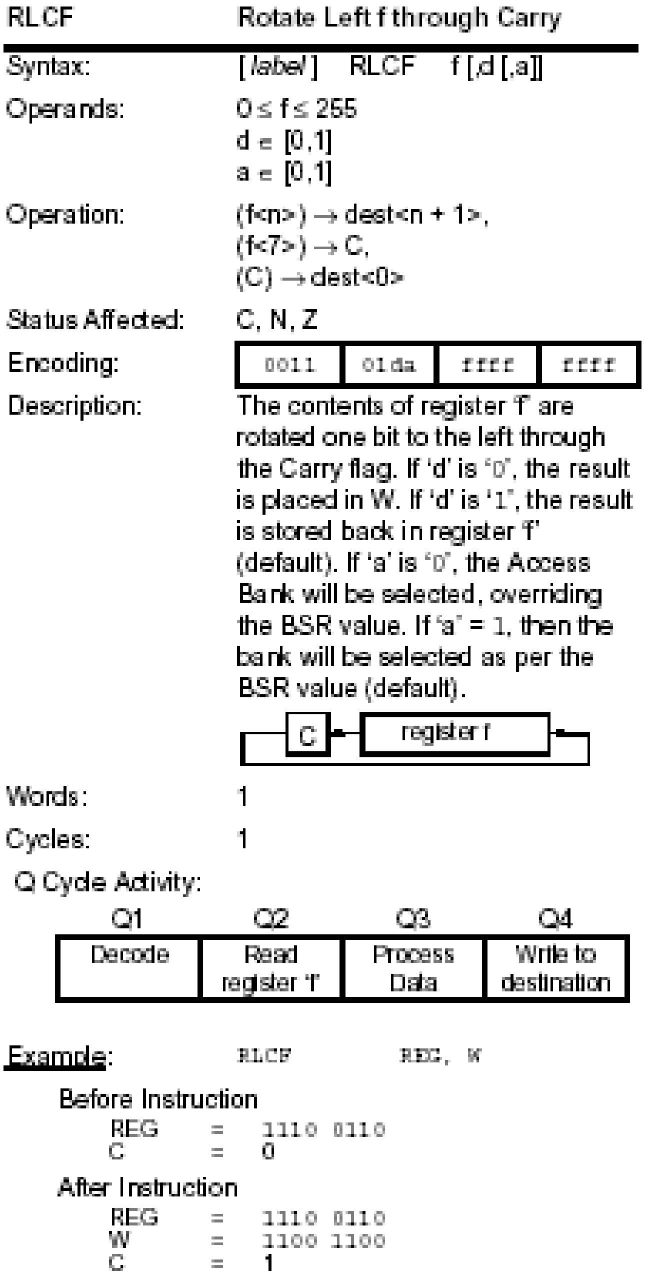
Before Instruction
RESULT = 0x13
W = 0x91

After Instruction
RESULT = 0x13
W = 0x93

Notes:

-

- Rotate Left f through Carry



RLCF Rotate Left f through Carry

Syntax: [*label*] RLCF f [,d [,a]]

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [0,1]

Operation: (f<n>) → dest<n + 1>,
(f<7>) → C,
(C) → dest<0>

Status Affected: C, N, Z

Encoding:

0011	01da	ffff	ffff

Description: The contents of register 'f' are rotated one bit to the left through the Carry flag. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: RLCF REG, W

Before Instruction

```
REG  =  1110 0110
C    =  0
```

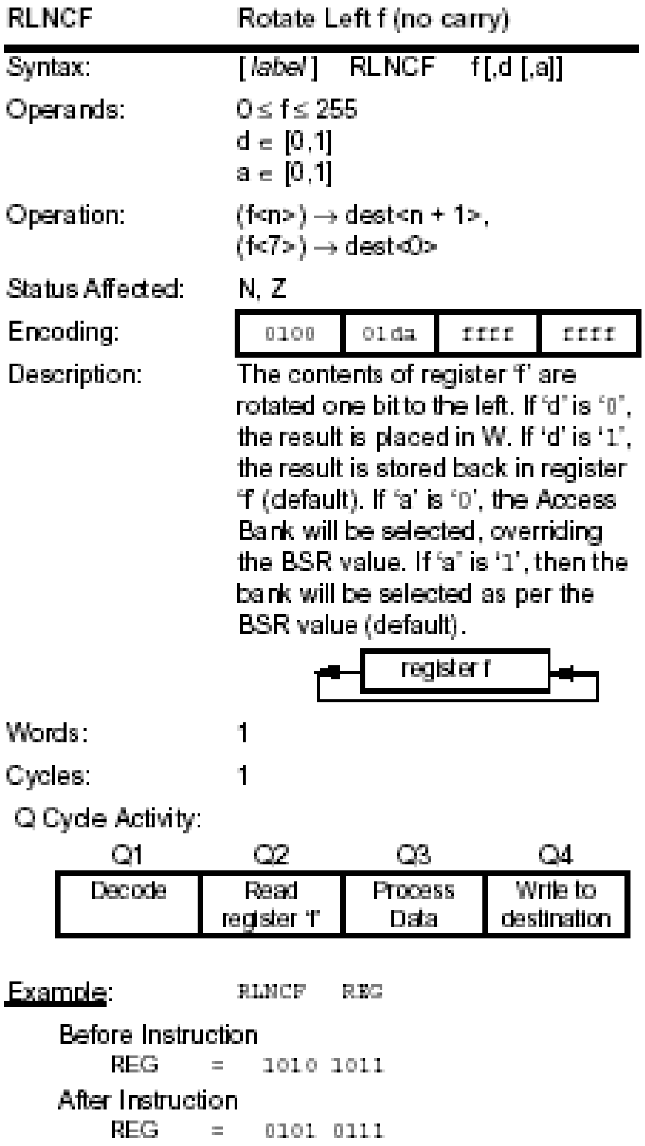
After Instruction

```
REG  =  1110 0110
W    =  1100 1100
C    =  1
```

Notes:

▪

- Rotate Left f (No Carry)



RLNCF	Rotate Left f (no carry)
Syntax:	[*label*] RLNCF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f<n>) → dest<n + 1>, (f<7>) → dest<0>
Status Affected:	N, Z
Encoding:	0100 | 01da | ffff | ffff
Description:	The contents of register 'f' are rotated one bit to the left. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).

Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: RLNCF REG

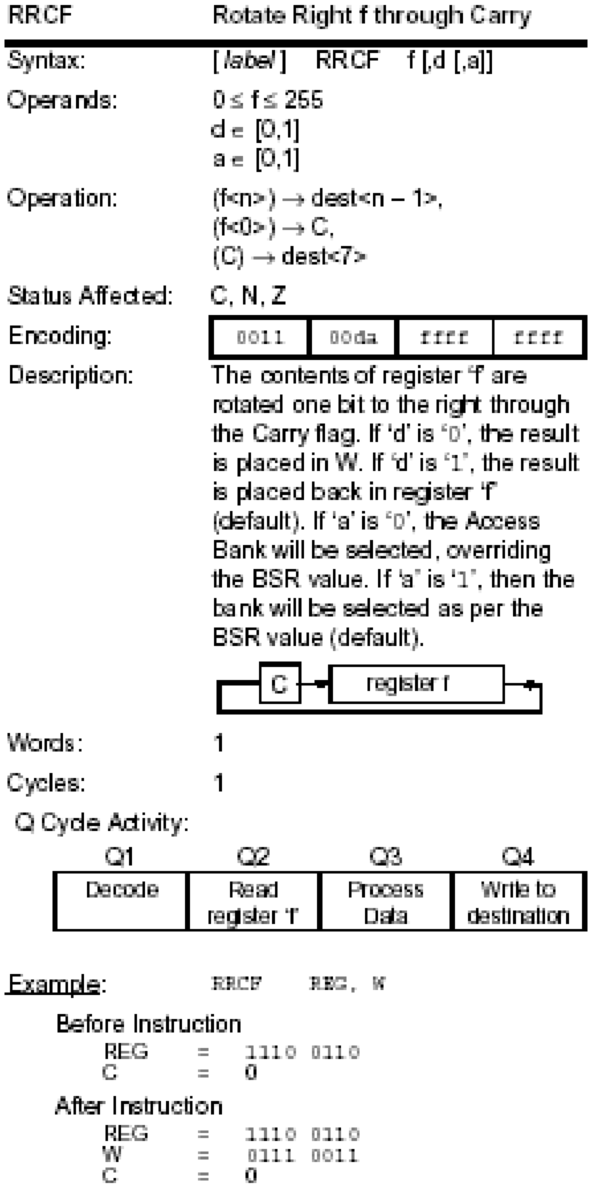
Before Instruction
REG = 1010 1011

After Instruction
REG = 0101 0111

Notes:

-

- Rotate Right f through Carry



RRCF Rotate Right f through Carry

Syntax:	[*label*] RRCF f [,d [,a]]
Operands:	0 ≤ f ≤ 255 d ∈ [0,1] a ∈ [0,1]
Operation:	(f<n>) → dest<n – 1>, (f<0>) → C, (C) → dest<7>
Status Affected:	C, N, Z
Encoding:	0011 00da ffff ffff
Description:	The contents of register 'f' are rotated one bit to the right through the Carry flag. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

Example: `RRCF REG, W`

Before Instruction

REG	=	1110 0110
C	=	0

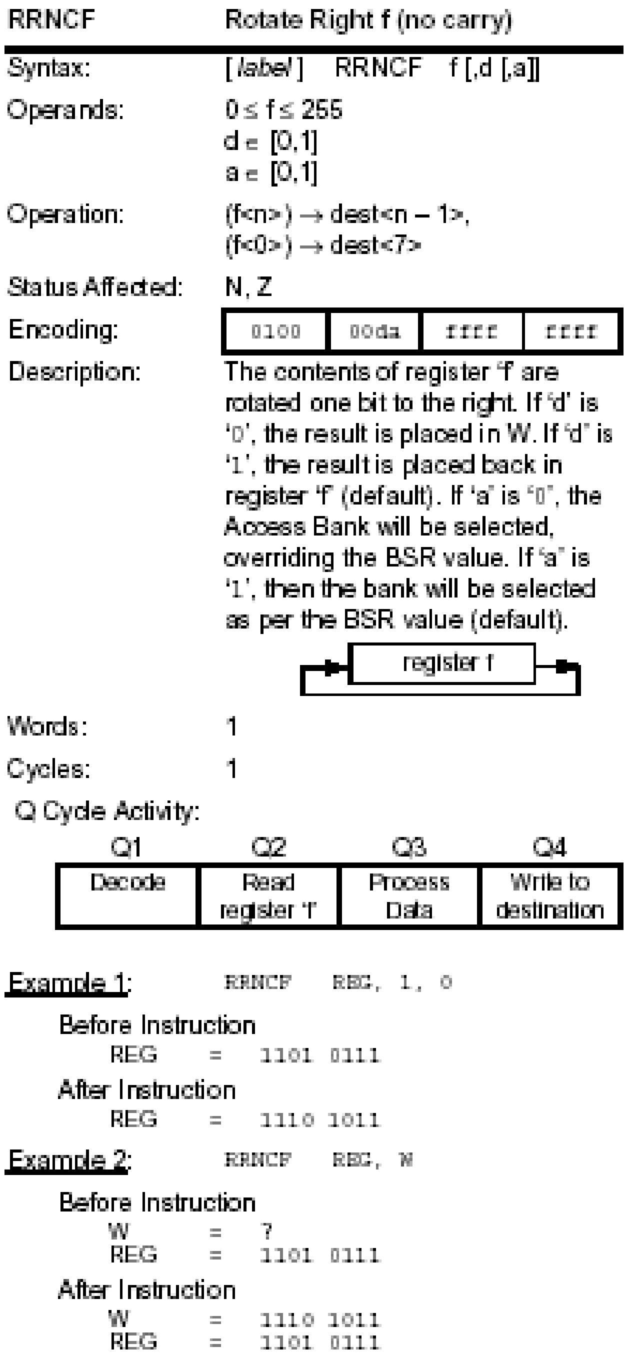
After Instruction

REG	=	1110 0110
W	=	0111 0011
C	=	0

Notes:

-

- Rotate Right f (No Carry)



RRNCF	Rotate Right f (no carry)
Syntax:	[*label*] RRNCF f [,d [,a]]
Operands:	$0 \le f \le 255$ $d \in [0,1]$ $a \in [0,1]$
Operation:	(f<n>) → dest<n – 1>, (f<0>) → dest<7>
Status Affected:	N, Z
Encoding:	0100 00da ffff ffff
Description:	The contents of register 'f' are rotated one bit to the right. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
	register f
Words:	1
Cycles:	1

Q Cycle Activity:

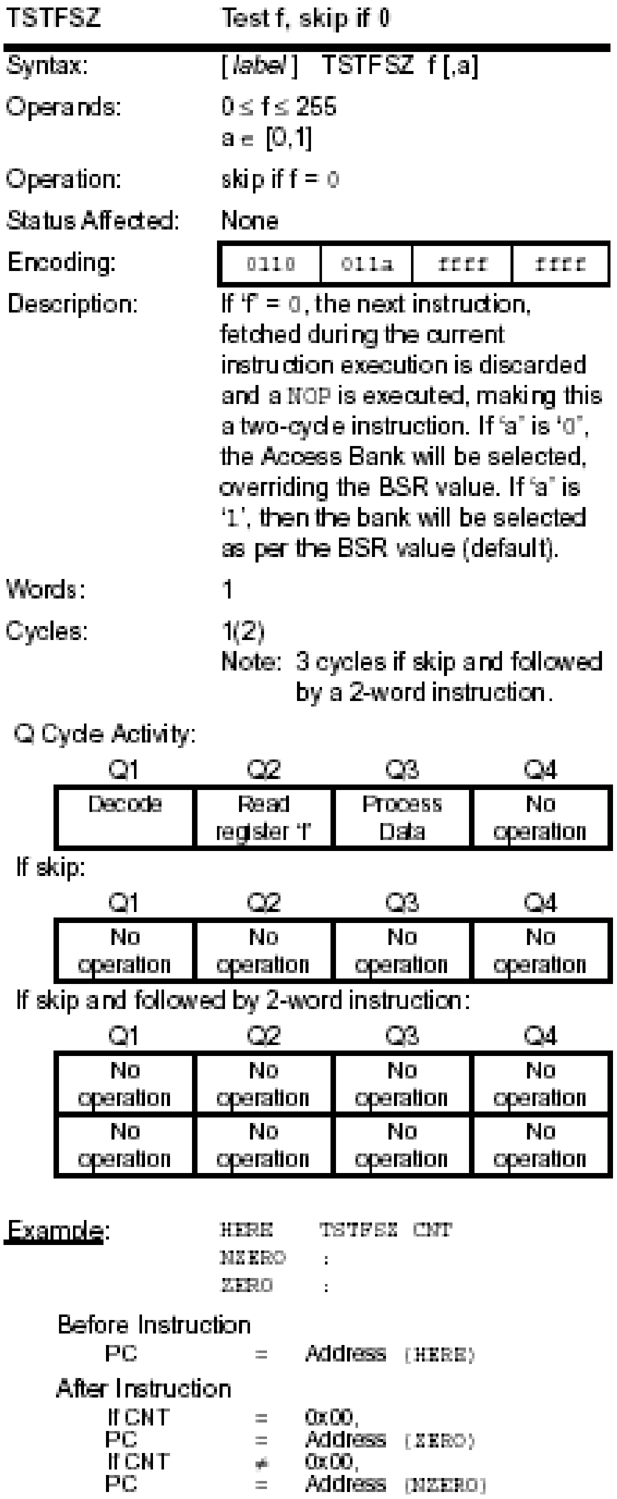
Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	Write to destination

```
Example 1:        RRNCF    REG, 1, 0
    Before Instruction
        REG    =    1101 0111
    After Instruction
        REG    =    1110 1011
Example 2:        RRNCF    REG, W
    Before Instruction
        W      =    ?
        REG    =    1101 0111
    After Instruction
        W      =    1110 1011
        REG    =    1101 0111
```

Notes:

-

- Test f, skip if 0



TSTFSZ **Test f, skip if 0**

Syntax:	[*label*] TSTFSZ f [,a]
Operands:	0 ≤ f ≤ 255 a ∈ [0,1]
Operation:	skip if f = 0
Status Affected:	None
Encoding:	0110 011a ffff ffff
Description:	If 'f' = 0, the next instruction, fetched during the current instruction execution is discarded and a NOP is executed, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Words:	1
Cycles:	1(2) Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process Data	No operation

If skip:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation

If skip and followed by 2-word instruction:

Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

Example:

```
HERE     TSTFSZ  CNT
NZERO    :
ZERO     :
```

Before Instruction
PC = Address (HERE)

After Instruction
If CNT = 0x00,
PC = Address (ZERO)
If CNT ≠ 0x00,
PC = Address (NZERO)

Notes:

- Write a C code segment and an Assembly code segment that sort the content of locations 0x120, 0x122, and 0x124 such that 0x120 contains the smallest value and 0x124 contains the largest value.

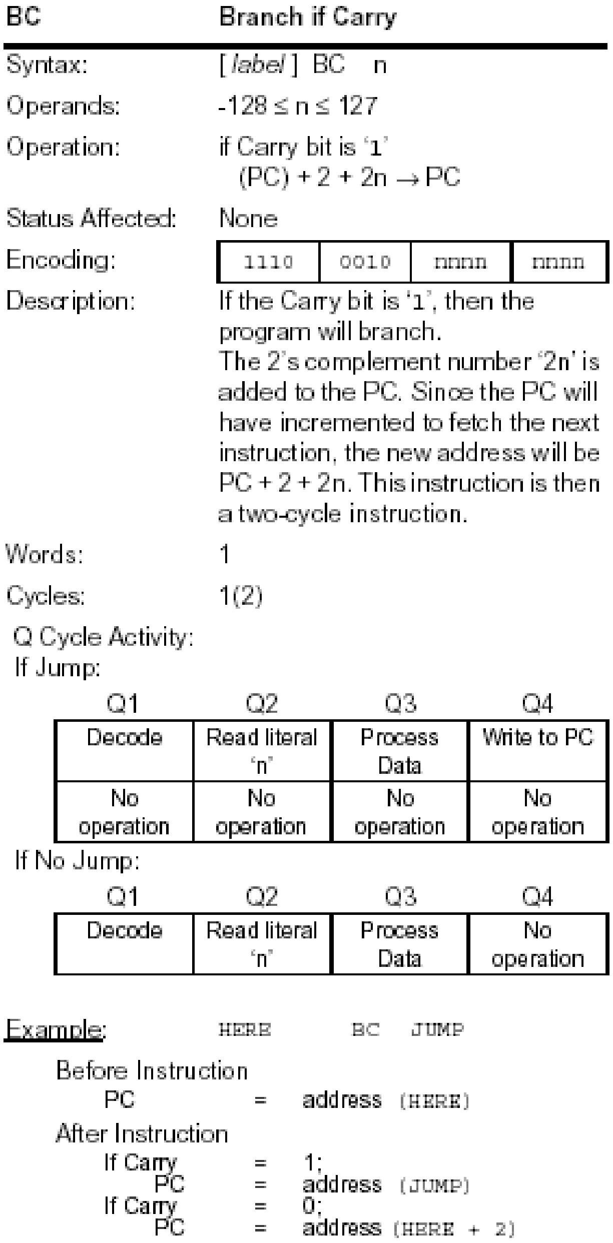
Solution

5.4. Branch Operations

Processors execute one instruction after another unless interrupted or redirected. In order to implement high level language constructs such as conditional statements (i.e. If-Then-Else, Switch) or loop statements (i.e. For, While), processors provide an ability to branch to other locations in program memory based on conditions. Branch instructions allow the PC value to be redirected to locations in memory other than the next instruction (PC + 2). In other words, in normal execution, once an instruction is executed, the PC is changed to PC+2. But if the condition for the branch is true, then the PC will be changed to the new location specified by the branch instruction.

PICmicro provides a set of branch and GOTO instructions. The remainder of this section covers branch instructions that redirect PC based on status of Carry, Overflow, Negative, Zero flags, or unconditionally.

❖ Branch if Carry



BC **Branch if Carry**

Syntax:	[*label*] BC n
Operands:	$-128 \le n \le 127$
Operation:	if Carry bit is '1' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 0010 nnnn nnnn
Description:	If the Carry bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

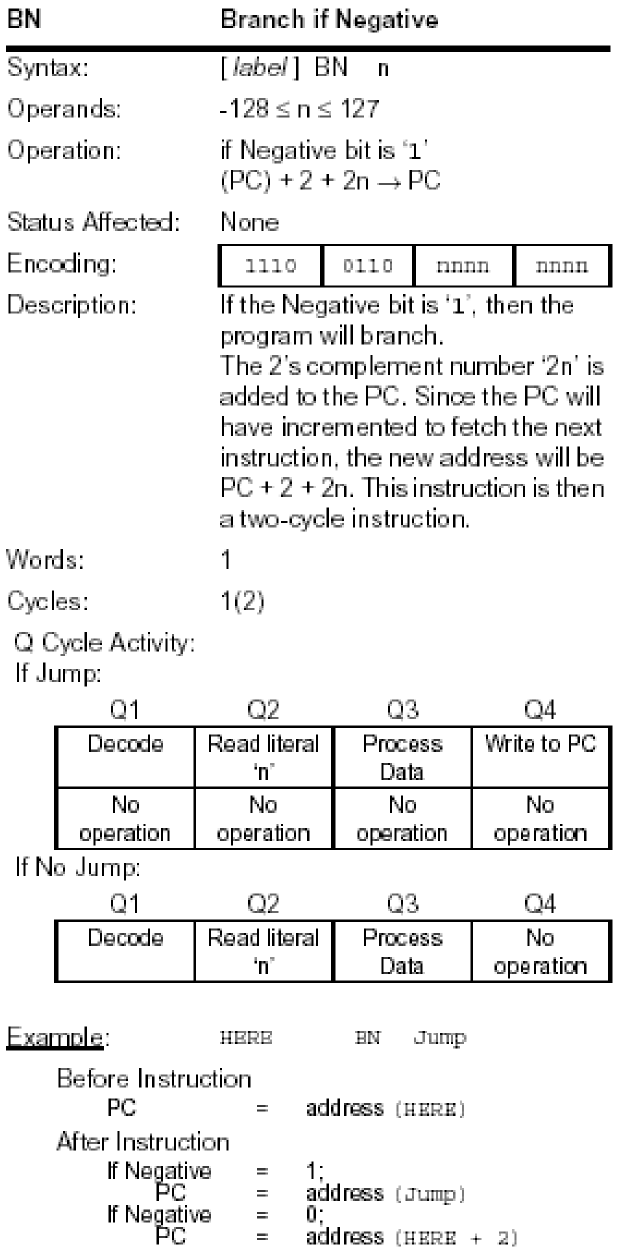
Example: HERE BC JUMP

Before Instruction
PC = address (HERE)
After Instruction
If Carry = 1;
PC = address (JUMP)
If Carry = 0;
PC = address (HERE + 2)

Notes:

▪

- Branch if Negative



BN **Branch if Negative**

Syntax:	[*label*] BN n
Operands:	-128 ≤ n ≤ 127
Operation:	if Negative bit is '1' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 0110 nnnn nnnn
Description:	If the Negative bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BN Jump`

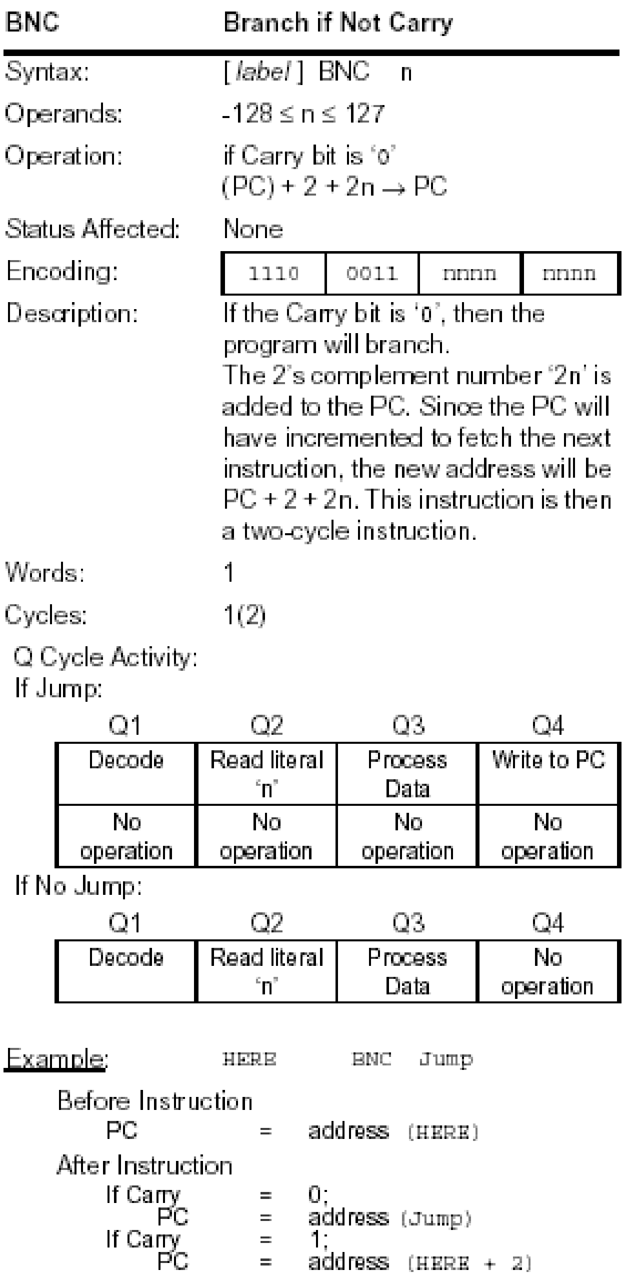
Before Instruction
PC = address (HERE)

After Instruction
If Negative = 1;
PC = address (Jump)
If Negative = 0;
PC = address (HERE + 2)

Notes:

▪

- Branch if Not Carry



BNC **Branch if Not Carry**

Syntax: [*label*] BNC n

Operands: -128 ≤ n ≤ 127

Operation: if Carry bit is '0'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0011	nnnn	nnnn

Description: If the Carry bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNC Jump`

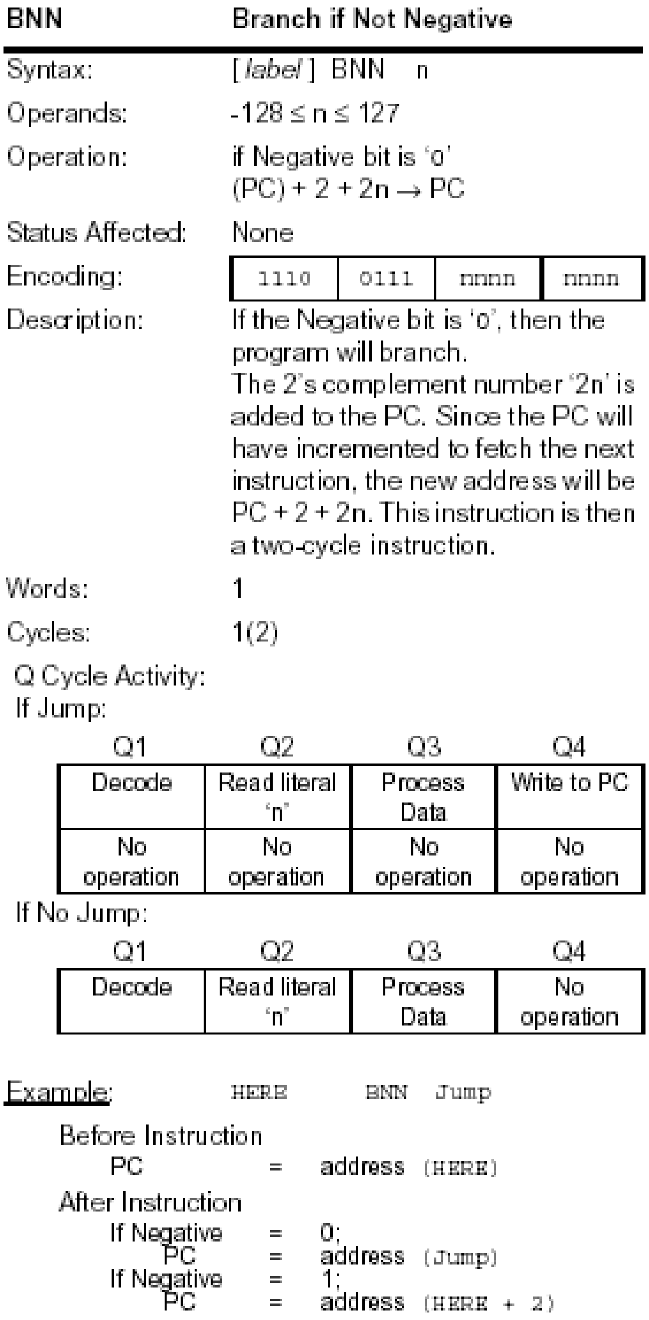
Before Instruction
PC = address (HERE)

After Instruction
If Carry = 0;
PC = address (Jump)
If Carry = 1;
PC = address (HERE + 2)

Notes:

▪

- Branch if Not Negative



BNN **Branch if Not Negative**

Syntax: [*label*] BNN n

Operands: -128 ≤ n ≤ 127

Operation: if Negative bit is '0'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0111	nnnn	nnnn

Description: If the Negative bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

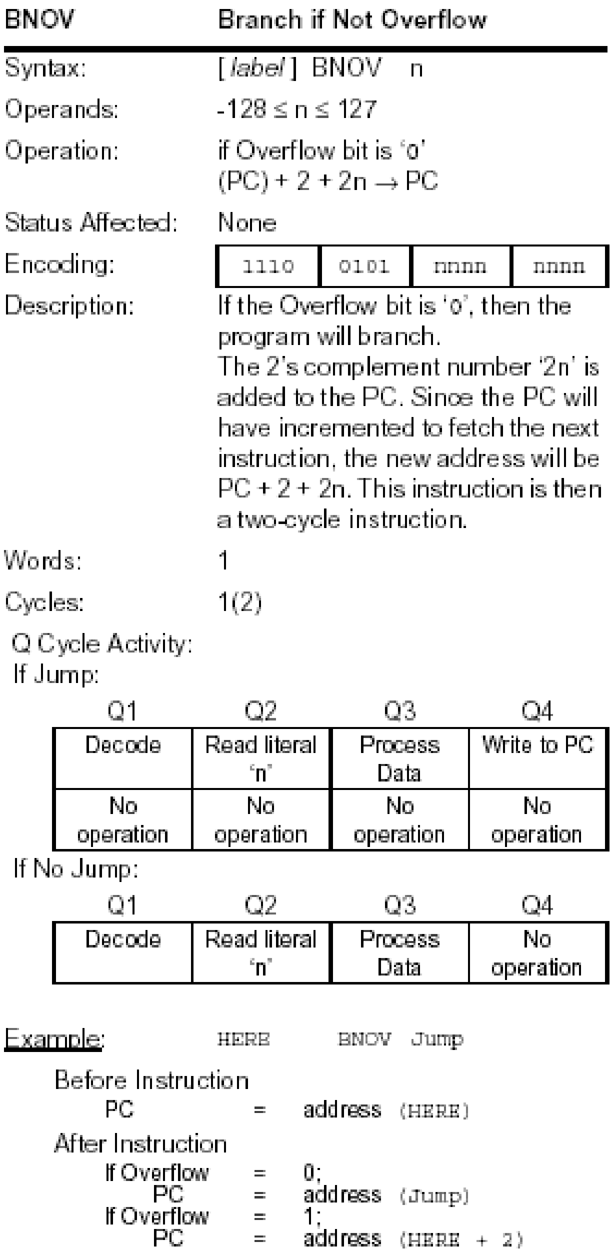
Example: `HERE BNN Jump`

Before Instruction
PC = address (HERE)
After Instruction
If Negative = 0;
PC = address (Jump)
If Negative = 1;
PC = address (HERE + 2)

Notes:

-

- Branch if Not Overflow



BNOV **Branch if Not Overflow**

Syntax: [*label*] BNOV n

Operands: -128 ≤ n ≤ 127

Operation: if Overflow bit is '0'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0101	nnnn	nnnn

Description: If the Overflow bit is '0', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNOV Jump`

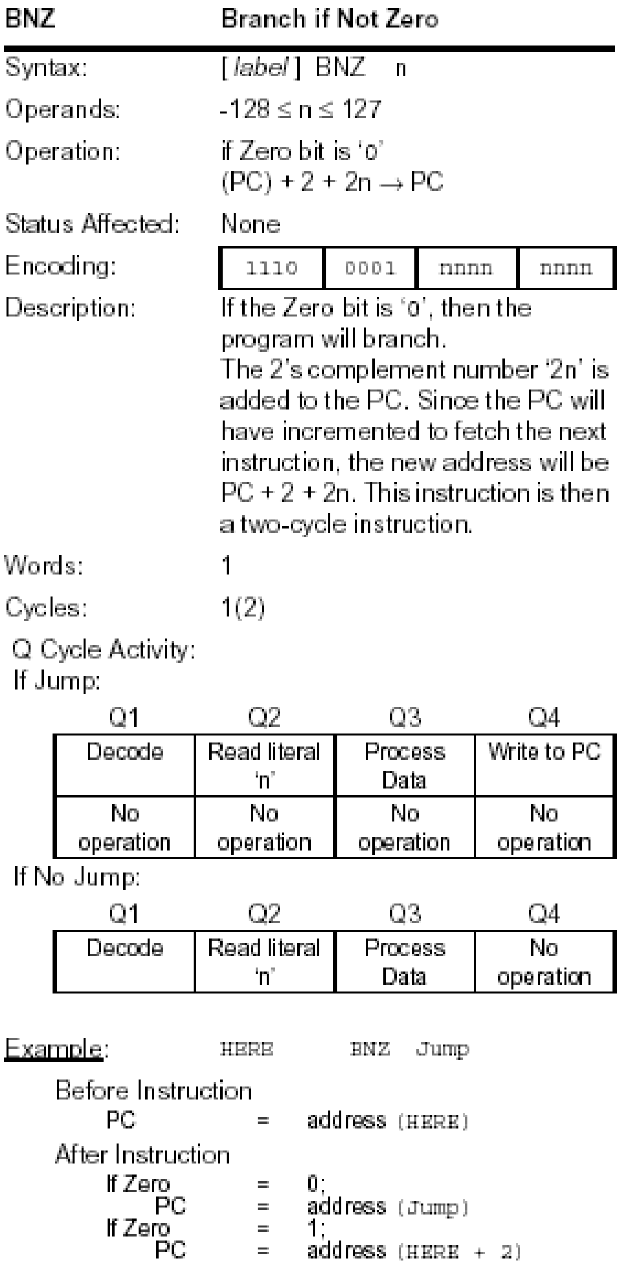
Before Instruction
PC = address (HERE)

After Instruction
If Overflow = 0;
PC = address (Jump)
If Overflow = 1;
PC = address (HERE + 2)

Notes:

-

- Branch if Not Zero



BNZ **Branch if Not Zero**

Syntax:	[*label*] BNZ n
Operands:	-128 ≤ n ≤ 127
Operation:	if Zero bit is '0' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 0001 nnnn nnnn
Description:	If the Zero bit is '0', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BNZ Jump`

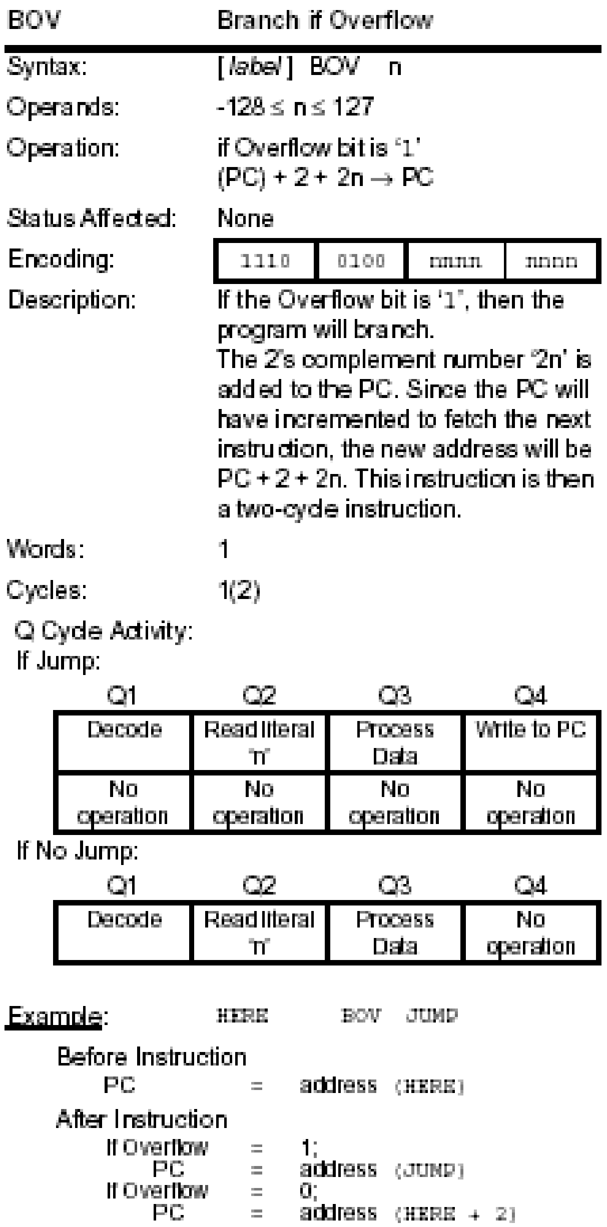
Before Instruction
PC = address (HERE)

After Instruction
If Zero = 0;
PC = address (Jump)
If Zero = 1;
PC = address (HERE + 2)

Notes:

-

- Branch if Overflow



BOV **Branch if Overflow**

Syntax: [*label*] BOV n

Operands: -128 ≤ n ≤ 127

Operation: if Overflow bit is '1'
(PC) + 2 + 2n → PC

Status Affected: None

Encoding:

1110	0100	nnnn	nnnn

Description: If the Overflow bit is '1', then the program will branch.
The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:
If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BOV JUMP`

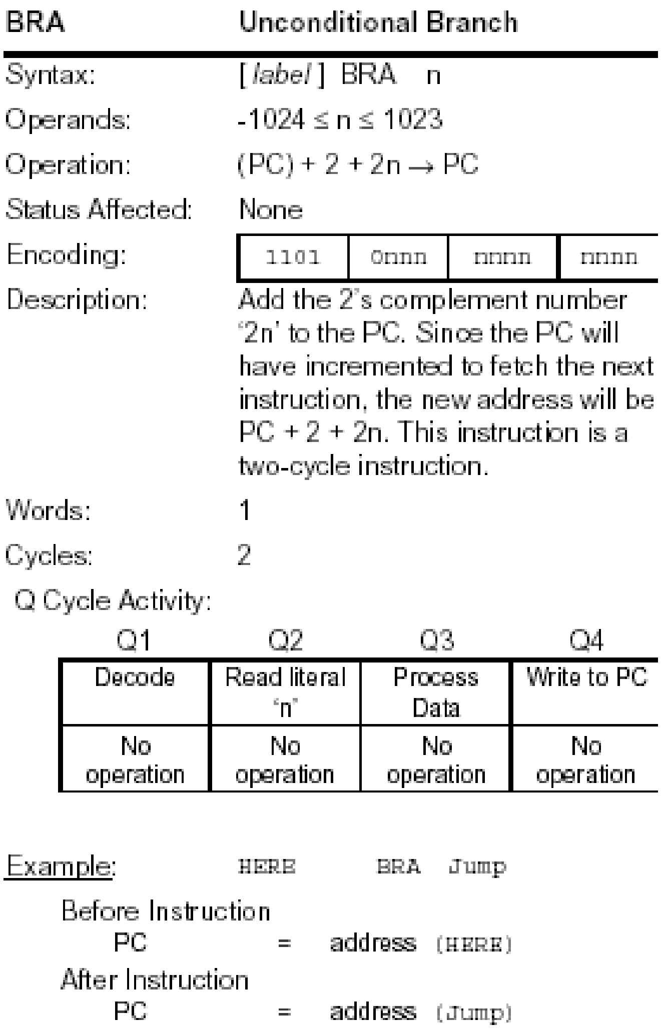
Before Instruction
PC = address (HERE)

After Instruction
If Overflow = 1;
PC = address (JUMP)
If Overflow = 0;
PC = address (HERE + 2)

Notes:

-

- Branch Unconditionally



BRA **Unconditional Branch**

Syntax: [*label*] BRA n

Operands: $-1024 \leq n \leq 1023$

Operation: $(PC) + 2 + 2n \rightarrow PC$

Status Affected: None

Encoding:

1101	0nnn	nnnn	nnnn

Description: Add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is a two-cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

Example: `HERE BRA Jump`

Before Instruction
PC = address (`HERE`)

After Instruction
PC = address (`Jump`)

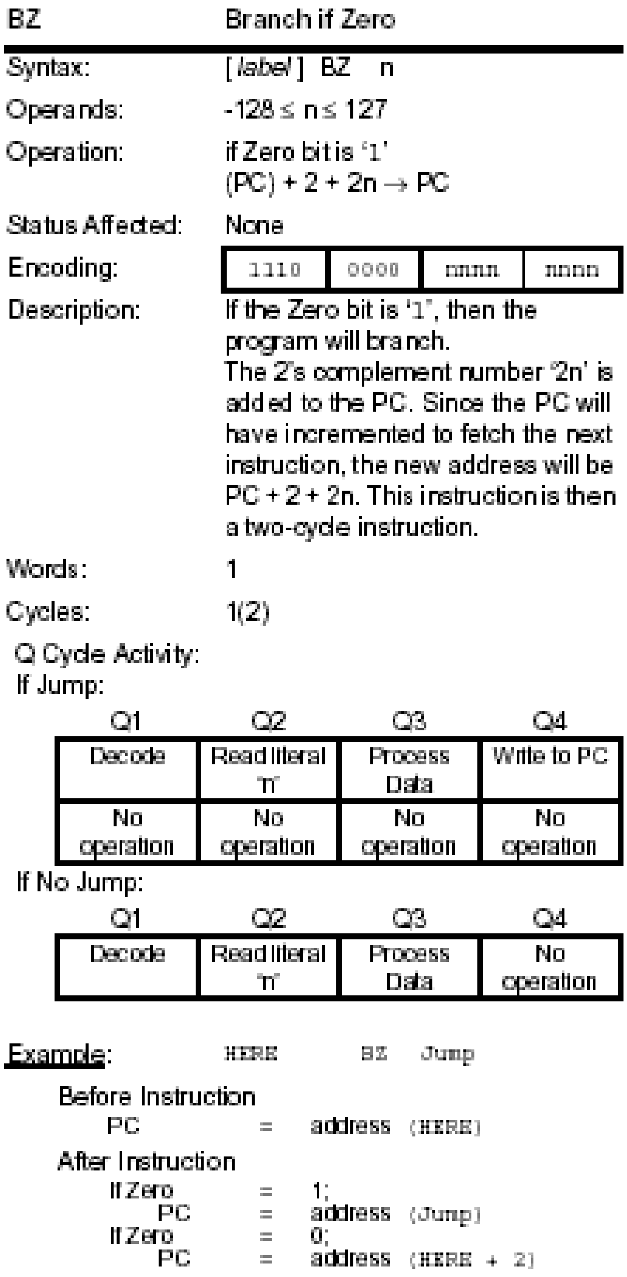
Notes:

▪

- Branch if Zero

Notes:

▪



BZ **Branch if Zero**

Syntax:	[*label*] BZ n
Operands:	-128 ≤ n ≤ 127
Operation:	if Zero bit is '1' (PC) + 2 + 2n → PC
Status Affected:	None
Encoding:	1110 | 0000 | nnnn | nnnn
Description:	If the Zero bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC + 2 + 2n. This instruction is then a two-cycle instruction.
Words:	1
Cycles:	1(2)

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

Example: `HERE BZ Jump`

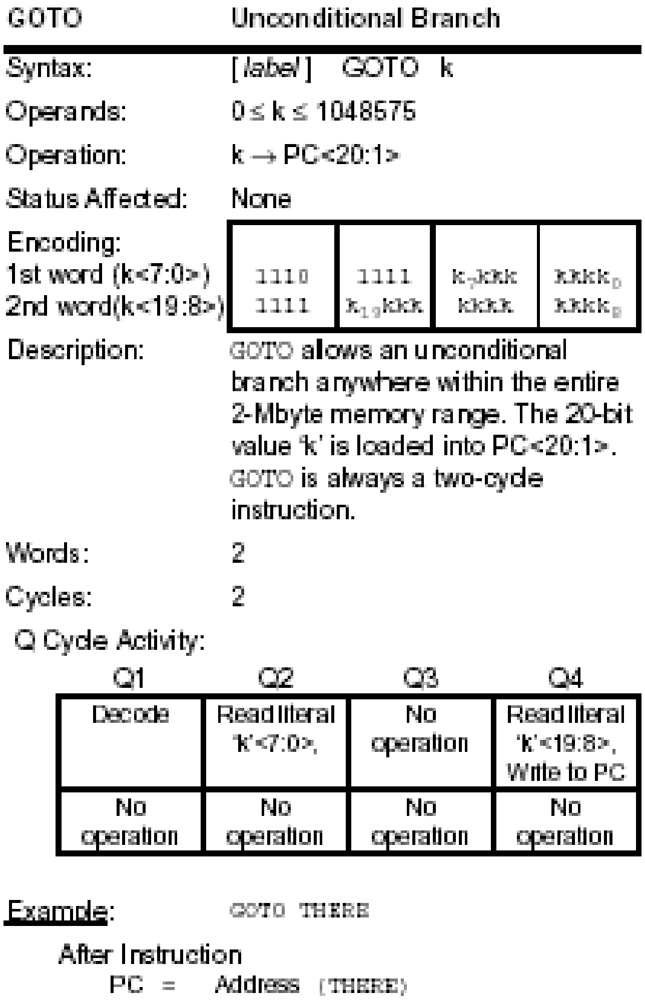
Before Instruction

```
PC      =  address (HERE)
```

After Instruction

```
If Zero =  1;
     PC =  address (Jump)
If Zero =  0;
     PC =  address (HERE + 2)
```

❖ Go to address 1st word, 2nd word



GOTO **Unconditional Branch**

Syntax:	[*label*] GOTO k
Operands:	0 ≤ k ≤ 1048575
Operation:	k → PC<20:1>
Status Affected:	None

Encoding:

1st word (k<7:0>)	1110	1111	$k_7$kkk	kkkk_0
2nd word(k<19:8>)	1111	k_{19}kkk	kkkk	kkkk_8

Description: GOTO allows an unconditional branch anywhere within the entire 2-Mbyte memory range. The 20-bit value 'k' is loaded into PC<20:1>. GOTO is always a two-cycle instruction.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'<7:0>,	No operation	Read literal 'k'<19:8>, Write to PC
No operation	No operation	No operation	No operation

Example: GOTO THERE

After Instruction
PC = Address (THERE)

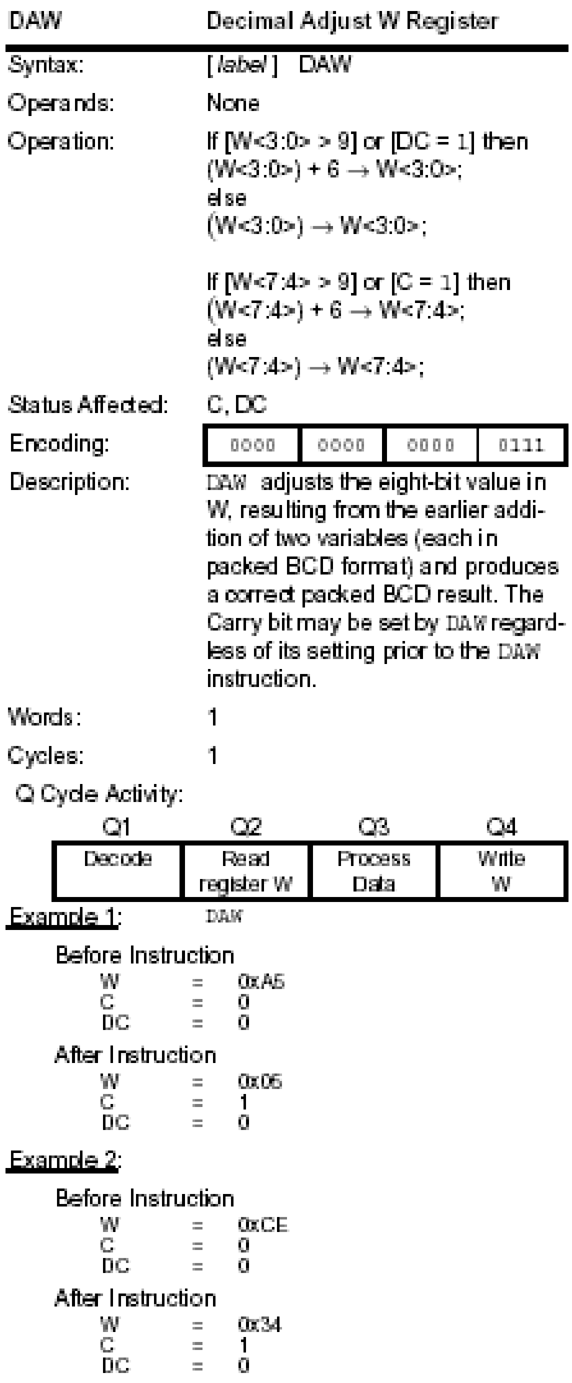
Notes:

▪

5.5. Specialty Operations

This section contains detailed descriptions of PICmicro instructions that are useful, but do not fit into any of the tradition instruction categories. The three instructions discussed here are “Decimal Adjust WREG, DAW”, “No Operation, NOP” and “Software Device Reset, RESET”.

❖ Decimal Adjust WREG



DAW **Decimal Adjust W Register**

Syntax:	[*label*] DAW
Operands:	None
Operation:	If [W<3:0> > 9] or [DC = 1] then (W<3:0>) + 6 → W<3:0>; else (W<3:0>) → W<3:0>; If [W<7:4> > 9] or [C = 1] then (W<7:4>) + 6 → W<7:4>; else (W<7:4>) → W<7:4>;
Status Affected:	C, DC
Encoding:	0000 0000 0000 0111
Description:	`DAW` adjusts the eight-bit value in W, resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result. The Carry bit may be set by `DAW` regardless of its setting prior to the `DAW` instruction.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register W	Process Data	Write W

Example 1: `DAW`

Before Instruction

W	=	0xA5
C	=	0
DC	=	0

After Instruction

W	=	0x05
C	=	1
DC	=	0

Example 2:

Before Instruction

W	=	0xCE
C	=	0
DC	=	0

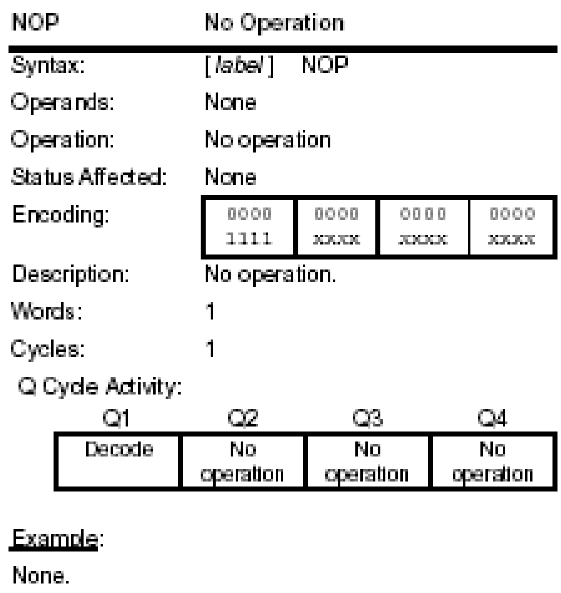
After Instruction

W	=	0x34
C	=	1
DC	=	0

Notes:

-

❖ No Operation



NOP No Operation

Syntax: [*label*] NOP

Operands: None

Operation: No operation

Status Affected: None

Encoding:

0000 1111	0000 xxxx	0000 xxxx	0000 xxxx

Description: No operation.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	No operation	No operation

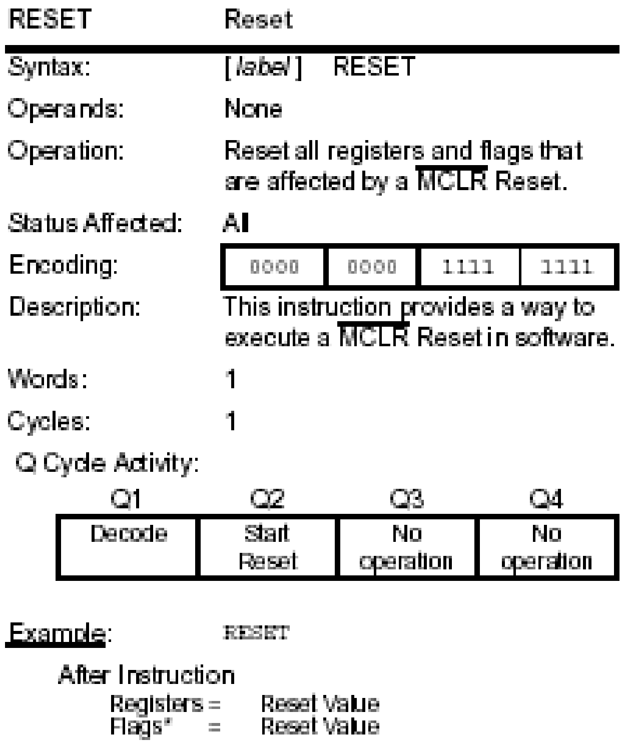
Example:

None.

Notes:

▪

❖ Software Device Reset



RESET Reset

Syntax: [*label*] RESET

Operands: None

Operation: Reset all registers and flags that are affected by a $\overline{\text{MCLR}}$ Reset.

Status Affected: All

Encoding:

0000	0000	1111	1111

Description: This instruction provides a way to execute a $\overline{\text{MCLR}}$ Reset in software.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Start Reset	No operation	No operation

Example: RESET

After Instruction

Registers	=	Reset Value
Flags*	=	Reset Value

Notes:

▪

5.6. IEEE Standards for Floating Point

As much as we like integers, real world problems have fractions and decimals so we have to learn to deal with real numbers. Here are a few real numbers.

π → 3.14159265…
e → 2.71828….

There are also large numbers that are not fractions, but still cannot be represented using the normal variable sizes (i.e. 32-bit) to store them. For example:

436,972,000,000,000,000 → 4.36972×10^{17}

This number is the normalized (no leading 0) scientific notation ($d.ddddd \times 10^{n}$).

The need to represent real numbers and extremely large or small numbers has lead to the need for floating point representation. IEEE 754 floating-point standards, which are found in virtually every computer system since 1980, address these requirements.

Some microprocessors have floating point instructions built-in standard, while in others it has to been implemented in software. PICmicro does not have built-in floating point support, but the floating point operation may be implemented using the available instructions.

The remainder of this section discusses the IEEE 754 floating-point standards.

- Computer representation

  Real numbers are represented as binary Floating Point format which is shown below:

  $1.ffffffff \times 2^{eeeeee}$

  Where:

  fffffff is the binary number representing the fractions
  eeee is the binary number representing the exponent

  The 1 before the decimal point is assumed in floating point and it is not explicitly stored.

  The benefits of always using Floating Point (Normalized Scientific notation in binary) are:

  - Simplifies exchange of data – no conversion required
  - Simplifies arithmetic algorithms – no conversion required
  - Increases the accuracy of the stored number

- Single Precision Floating Point (Float) Representation

  The Floating Point designer must make tradeoffs between the size of the fraction and the size of the exponent since word size is limited. In other words, the trade off is between precision (fraction), and range (exponent).

  In both Single and Double Precision Floating Point format a single bit is used to represent the sign of fraction, where s=1 is negative and s=0 is positive.

- Single Precision Format

	Exponent				***Fraction***			
31	30	29	…	23	22	…	1	0
s	8 bits of signed exponents (Bias = 127)				2^{-1}	23 bits of fraction	2^{-22}	2^{-23}

Note: If we number the fraction bits from left to right f_1, f_2, f_3, …
$(-1)^s$ x (1 + fraction) x $2^{(\text{exponent – bias})}$ = $(-1)^s$ x {1 + (f_1 x 2^{-1}) + (f_2 x 2^{-2}) + (f_3 x 2^{-3}) + . . .} x $2^{(\text{exponent-bias})}$

Example of binary word equivalent to floating point numbers:

$-1.25\text{x}2^{18}$ → 1 10010001 01000000000000000000000 or $(\text{C8 90 00 00})_{hex}$
$1.25\text{x}2^{-1}$ → 0 01111110 01000000000000000000000 or $(\text{3F 10 00 00})_{hex}$

Note: The 1 left of decimal point (1.ffff) is implicit and is not represented in the binary format.

In floating point, the programmer has to watch out for errors with a focus on the exponents. Below are the two error cases:

- Overflow
  A situation in which a positive exponent becomes too large to fit in the exponent field.
- Underflow
  A situation in which a negative exponent becomes too large to fit in the exponent field.

- Example – Given a single precision floating point “FEA0 0000h” write its equivalent decimal real number.

  Solution:
  1) Write in Binary equivalent → 1 111 1110 1010 … 0
  2) convert to Decimal → - 1.25 x 2^{126}

- Double Precision Format
  In order to represent larger numbers with more precision (reducing the possibility of underflow or overflow), IEEE 754 double precision floating point format is used. Here is an outline of double precision floating point format:

32-bit word								32-bit word		
63	62	61	…	52	51	…	32	31	…	0
s	11 bits of exponents (Bias = 1023)				2^{-1}	52 bits of fraction	2^{-20}	2^{-21}		2^{-52}

Note: If we number the fraction bits from left to right f_1, f_2, f_3, …
$(-1)^s$ x (1 + fraction) x $2^{(\text{exponent – bias})}$ = $(-1)^s$ x {1 + (f_1 x 2^{-1}) + (f_2 x 2^{-2}) + (f_3 x 2^{-3}) + . . .} x $2^{(\text{exponent-bias})}$

- Summary of IEEE 754 Floating-Point Standards
  Since the 1 to the left of the decimal is implicit we could say that the precision is 24 bit for single precision and 53 bits for the double precision floating point arithmetic.

  For example, if we number the fraction bits from left to right f_1, f_2, f_3, the value may be represented by:
  $(-1)^s$ x (1 + fraction) x 2^{exponent} = $(-1)^s$ x {1 + (f1 x 2^{-1}) + f2 x 2^{-2}) + f3 x 2^{-3}) + . . .} x $2^{(\text{exponent-bias})}$
  The following table outlines number ranges (valid and invalid) when using IEEE 754 floating point

format:

Single Precision		Double Precision		Object Represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	nonzero	0	nonzero	± de-normalized number
1-254	anything	1-2046	anything	± floating –point number
255	0	2047	0	± infinity
255	nonzero	2047	nonzero	Nan (Not a Number)

- Example - Single Precision
  Convert -.75 to MIPS single precision binary format

  $-0.75 = (-1)^1 x(1+1x2^{-1})(2^{-1}) = (-1)^s x \{1 + (f1 x 2^{-1}) + f2 x 2^{-2}) + f3 x 2^{-3}) + . . .\} x 2^{(exponent-bias)}$

  for single precision bias is 127 → exponent – bias = -1 → exponent = 126 = $(0111\ 1110)_2$

  Therefore:

1	0111 1110	100 0000 0000 0000 0000 0000
s	8 bits of signed exponents (Bias = 127)	23 bits of fraction 2^{-1} 2^{-22} 2^{-23}

- Example - Double Precision
  Convert -.75 to MIPS Double precision binary format

  $-0.75 = (-1)^1 x(1+1x2^{-1})(2^{-1}) = (-1)^s x \{1 + (f1 x 2^{-1}) + f2 x 2^{-2}) + f3 x 2^{-3}) + . . .\} x 2^{(exponent-bias)}$

  for double precision bias is 1023 → exponent – bias = -1 → exponent = 1022 = $(011\ 1111\ 1110)_2$

  Therefore:

1	011 1111 1110	100 0000 0000 . . . 0000 0000 0000
s	11 bits of signed exponents (Bias = 1023)	52 bits of fraction 2^{-1} 2^{-52}

- Example – Convert $-5.25 x2^{-2}$ to double precision floating point binary format.

  Solution:
  “student exercise”

- Example – Covert 1.25 x10^{-1} to double precision floating point binary format.

  Solution:
  “student exercise”

- Example – Write the decimal equivalent of the single precision floating point “C5D8 0000”.

  Solution:
  “student exercise”

  ”Ans: -6912”

- Example – Write the single precision floating point binary equivalent for the decimal number “258.6875”.

  Solution:
  “student exercise”

  ”Ans: 0100 0011 1000 0000 0000 0000 010 1011”

- Example – What’s the largest and smallest possible number in:
  a) Single Precision Floating point format.
  b) Double Precision Floating point format.

  Solution:
  “student exercise”

- Floating-Point Addition
  Here are the steps in the example of adding .1 and -.4375

  Step 1. Adjust the smaller exponent to match the largest exponent (Fraction digit at the right place)

  $0.1 \rightarrow 1.0 \times 2^{-1}$ Unmodified
  $-0.4375 \rightarrow -0.0111 \times 2^{0} \rightarrow -1.11 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$

  Step 2. Add the adjusted significant (1.fffff)

  Result Significant = 1.0 – 0.111 = 0.001

  Step 3. Normalize the result

  results $=(1.000 \times 2^{-3})_2$

  Step 4. Round any additional fraction to the number of bits available

- Floating-Point Multiply
  Here are the steps in the example of multiplying .1 → $(1.000 \times 2^{-1})_2$ and -.4375 → $(-1.110 \times 2^{-2})_2$.

  Step 1. Add the exponents
  If you are working with biased exponents that after adding subtract one bias out to correct for having double amount of bias in the result.

  $-1 + (-2) = -3$

  Step 2. Multiply the significant (1.fffff)

  Result Significant = 1.000 x (1.110)= 1.110

  Step 3. Normalize the result & Check for overflow

  results $=(1.110 \times 2^{-3})_2$

  Step 4. Round any additional fraction to the number of bits available

  No changes

  Step 5. Figure out the sign (if the operands' signs are the same then the product is positive and if the operands' signs are different then the product is negative)

  results = $(-1.110 \times 10^{-3})_2$

5.7. Additional Resources

- Staff. Microchip PIC 18F1220/1320 Data Sheet. (2004) Microchip Technology Incorporated.

- Huang,. PIC Microcontroller: An Introduction to Software & Hardware Interfacing, (2004) Thomson.

- Reese. Microprocessor: From Assembly Language to C using the PIC18Fxxx2. (2003) Course Technology.

- Peterson. Computer Organization and Design, (2007) Elsevier Service.

- IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754-1985), (1985 with 2008 revision) Institute of Electrical and Electronics Engineers.

5.8. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 6. C/ASSEMBLY/MACHINE LANGUAGE EQUIVALENCIES

Key concepts and Overview

- Introduction
- Indirect Addressing
- Functions/Procedures
- Data Types
- Program Flow Controls
- Additional Resources

6.1. Introduction

In previous chapters, the underlying processor structure, instructions and logical design has been discussed. The objective of this chapter is to explore the compile process with specific focus on the equivalency between high level language (C language) and Assembly language.

We will be using PICmicro as the target processor and MPLAB's C18 as the compiler and development environment, which will be used to demonstrate examples of programming environment and build process. Refer to chapter 3 for step-by-step guide to installing, configuring and development using MPLAB's C18 IDE. Additional information regarding MPASM, C18 compiler and MPLAB IDE is available through the "Help>Topics" menu of the MPLAB IDE.

As discussed earlier, the high level language is compiled to Assembly and then to Machine language. The object code is combined with other pre-existing object codes to create the executable code that can be downloaded to the processor's program memory for execution. Although the steps described are common across the various systems, each processor and development environment would have its own unique file name and may combine one or more steps. Below are files that are generated during the build process of a C program in PICmicro environment:

- file.c
  C program file containing the C language code. Although it is shown here as a single file, the C program commonly consists of many files and directories.

  C code typically includes files that define data and references required by the C program. In PICmicro environment, each processor is defined through one such file. We are using processor P18F1220 therefore including file "p18f1220.h" (default location is C:\MCC18\h) would provide the register names, constants and other required definitions.

- file.lst
  Listing file is generated after the compilation process and includes the c program and the corresponding assembly code. The listing file is placed in the same directory as the project by default. A text editor such as notepad may be best tool to view file.lst files.

  Disassembly Listing which can be accessed from MPLAB IDE menu *"View > Disassembly Listing"* is a good tool for viewing the relationship between C and corresponding Assembly code. The rest of this chapter relies on this file to discuss the relationship between C and Assembly code.

- file.map
  Map file is generated by the linker and contains the symbols (variables, functions, …) and their value. This file provides detailed information about the allocation of data and code.

- file.o
  Object file is generated after the assembly program and contains the machine code (Binary). This code is combined with other object files required by the program to create the executable code that is downloaded to program memory for execution.

In the next few pages, an example of each of the above files for a simple C program is presented. The longer files have been truncated to show the type of content, and the reader is encouraged to use MAPLAB ID to view each file in its entirety and become familiar with type of information provided by each of these files.

- C program file (c2asm_into.c)

```
/***********************************************************
 * File: c2asm_into.c
 * Project: c to Assembly Language Equivalency
 * Author: Class
 * Updated: 2/14/10
 ***********************************************************/

//Process Specific definitions
#include <p18f1220.h>

// main() is the entry point to the program.
// PICmicro does not accept or return parameters.
void main(void)
{
       int count;

       count = count + 1;

} //main()
```

“p18f1220.h” include file (default location is C:\MCC18\h).

```
/*-------------------------------------------------------------------------
 * $Id: p18f1220.h,v 1.11.2.1 2005/07/25 18:23:27 nairnj Exp $
 * MPLAB-Cxx  PIC18F1220 processor header
 *
 * (c) Copyright 1999-2005 Microchip Technology, All rights reserved
 *-------------------------------------------------------------------------*/

#ifndef __18F1220_H
#define __18F1220_H

extern volatile near unsigned char       PORTA;
extern volatile near union {
  struct {
    unsigned RA0:1;
    unsigned RA1:1;
    unsigned RA2:1;
    unsigned RA3:1;
    unsigned RA4:1;
    unsigned RA5:1;
    unsigned RA6:1;
    unsigned RA7:1;
  };
  struct {
    unsigned AN0:1;
    unsigned AN1:1;
    unsigned AN2:1;
    unsigned AN3:1;
    unsigned :1;
    unsigned MCLR:1;
    unsigned CLKO:1;
    unsigned CLKI:1;

    <<<<<   MIDDLE SECTION OF THIS FILE HAS BEEN DELETED >>>>>

/*-------------------------------------------------------------------------
 *   CONFIG6H (0x30000b)
 *-------------------------------------------------------------------------*/
```

```
#define _CONFIG6H_DEFAULT    0xE0
#define _WRTB_ON_6H          0xBF
#define _WRTB_OFF_6H         0xFF

#define _WRTC_ON_6H          0xDF
#define _WRTC_OFF_6H         0xFF

#define _WRTD_ON_6H          0x7F
#define _WRTD_OFF_6H         0xFF

/*--------------------------------------------------------------------------
 *   CONFIG7L (0x30000c)
 *--------------------------------------------------------------------------*/
#define _CONFIG7L_DEFAULT    0x03
#define _EBTR0_ON_7L         0xFE
#define _EBTR0_OFF_7L        0xFF

#define _EBTR1_ON_7L         0xFD
#define _EBTR1_OFF_7L        0xFF

/*--------------------------------------------------------------------------
 *   CONFIG7H (0x30000d)
 *--------------------------------------------------------------------------*/
#define _CONFIG7H_DEFAULT    0x40
#define _EBTRB_ON_7H         0xBF
#define _EBTRB_OFF_7H        0xFF


#endif
```

- ❖ Listing file (c2asm_into.lst)

```
Address  Value    Disassembly              Source
-------  -------  -----------------------  ---------------------------------------------------
                                           /* $Id: c018i.c,v 1.3.14.1 2006/01/24 14:50:12 rhinec
Exp $ */
                                           /* Copyright (c)1999 Microchip Technology */
                                           /* MPLAB-C18 startup code, including initialized data
*/
                                           /* external reference to the user's main routine */
                                           extern void main (void);
                                           /* prototype for the startup function */
                                           void _entry (void);
                                           void _startup (void);
                                           /* prototype for the initialized data setup */
                                           void _do_cinit (void);

                                           extern volatile near unsigned long short TBLPTR;
                                           extern near unsigned FSR0;
                                           extern near char __FPFLAGS;
                                           #define RND 6

                                           #pragma code _entry_scn=0x000000
                                           void
                                           _entry (void)
                                           {
000000   ef81     GOTO      0x102          _asm goto _startup _endasm
000002   f000

000004   0012     RETURN    0x0            }
                                           #pragma code _startup_scn
                                           void
                                           _startup (void)
                                           {
                                             _asm
                                               // Initialize the stack pointer
000102   ee10     LFSR      0x1,0x80           lfsr 1, _stack
000104   f080
000106   ee20     LFSR      0x2,0x80           lfsr 2, _stack
000108   f080

00010a   6af8     CLRF      0xf8,0x0           clrf TBLPTRU, 0 // 1st silicon doesn't do this on
POR

00010c   9c01     BCF       0x1,0x6,0x0        bcf __FPFLAGS,RND,0 // Initialize rounding flag
for floating point libs

                                               _endasm
00010e   ec16     CALL      0x2c,0x0           _do_cinit ();
000110   f000


                                           loop:

                                             // Call the user's main routine
000112   ec65     CALL      0xca,0x0         main ();
000114   f000

000116   d7fd     BRA       0x112            goto loop;
000118   0012     RETURN    0x0            }                                /* end _startup() */


                   <<<<<   MIDDLE SECTION OF THIS FILE HAS BEEN DELETED >>>>>

0000ba   c0c9     MOVFF     0xc9,0xff8
0000bc   fff8
                                                 /* next entry... */
0000be   0100     MOVLB     0x0                  curr_entry--;
0000c0   07c5     DECF      0xc5,0x1,0x1
0000c2   0e00     MOVLW     0x0
0000c4   5bc6     SUBWFB    0xc6,0x1,0x1
0000c6   d7bf     BRA       0x46                 goto test;
                                           done:
```

```
                                           ;
0000c8   0012     RETURN    0x0            }

/*************************************************************
                                            * File: c2asm_into.
                                            * Project: c to Assembly Language
                                            * Author:
                                            * Updated: 2/14/

********************************************************

                                           //Process Specific definitions
                                           #include <p18f1220.h>
```

```
                                           // main() is the entry point to the program and does
not accept or return parameters.
0000ca   cfd9     MOVFF     0xfd9,0xfe6    void main(void)
0000cc   ffe6
0000ce   cfe1     MOVFF     0xfe1,0xfd9
0000d0   ffd9
0000d2   0e02     MOVLW     0x2
0000d4   26e1     ADDWF     0xe1,0x1,0x0
                                           {
                                            int count;

0000d6   cfde     MOVFF     0xfde,0x2       count = count + 1;
0000d8   f002
0000da   cfdd     MOVFF     0xfdd,0x3
0000dc   f003
0000de   0e01     MOVLW     0x1
0000e0   2602     ADDWF     0x2,0x1,0x0
0000e2   0e00     MOVLW     0x0
0000e4   2203     ADDWFC    0x3,0x1,0x0
0000e6   c002     MOVFF     0x2,0xfde
0000e8   ffde
0000ea   c003     MOVFF     0x3,0xfdd
0000ec   ffdd

0000ee   0e02     MOVLW     0x2            } //main()
0000f0   5ce1     SUBWF     0xe1,0x0,0x0
0000f2   e202     BC        0xf8
0000f4   6ae1     CLRF      0xe1,0x0
0000f6   52e5     MOVF      0xe5,0x1,0x0
0000f8   6ee1     MOVWF     0xe1,0x0
0000fa   52e5     MOVF      0xe5,0x1,0x0
0000fc   cfe7     MOVFF     0xfe7,0xfd9
0000fe   ffd9
000100   0012     RETURN    0x0
                                           LIST P=18F1220

                                           END
                                           ; RCS Header $Id: cmath18.asm,v 1.4.12.1 2006/01/13
04:11:25 nairnj Exp $
                                           ; CMATH18 DATA DEFINITION FILE

;*******************************************************************************
                                           ; VARIABLE ALLOCATION - Core math library routines

;*******************************************************************************

                                           MATH_DATA      UDATA_
                                          SIGN             RES 1     ; save location for sign in
MSB
                                           __FPFLAGSbits
                                           __FPFLAGS       RES 1     ; floating point library
exception flags

                                          GLOBAL  SIGN, __FPFLAGS, __

                                          END
```

- Disassembly Listing - MPLAB IDE menu *“View > Disassembly Listing”*
  The remainder of this chapter, Disassembly Listing will be used to discuss the C program build process and resulting machine code.

```
---  C:\MCC18\src\traditional\startup\c018i.c  ---------------------------
1:                 /* $Id: c018i.c,v 1.3.14.1 2006/01/24 14:50:12 rhinec Exp $ */
2:
3:                 /* Copyright (c)1999 Microchip Technology */
4:
5:                 /* MPLAB-C18 startup code, including initialized data */
6:
7:                 /* external reference to the user's main routine */
8:                 extern void main (void);
9:                 /* prototype for the startup function */
10:                void _entry (void);
11:                void _startup (void);
12:                /* prototype for the initialized data setup */
13:                void _do_cinit (void);
14:
15:                extern volatile near unsigned long short TBLPTR;
16:                extern near unsigned FSR0;
17:                extern near char __FPFLAGS;
18:                #define RND 6
19:
20:                #pragma code _entry_scn=0x000000
21:                void
22:                _entry (void)
23:                {
24:                _asm goto _startup _endasm
   000    EF81     GOTO 0x102
   002    F000     NOP
25:
26:                }
   004    0012     RETURN 0
27:                #pragma code _startup_scn
28:                void
29:                _startup (void)
30:                {
31:                  _asm
32:                    // Initialize the stack pointer
33:                    lfsr 1, _stack
   102    EE10     LFSR 0x1, 0x80
   104    F080     NOP
34:                    lfsr 2, _stack
   106    EE20     LFSR 0x2, 0x80
   108    F080     NOP
35:
36:                    clrf TBLPTRU, 0 // 1st silicon doesn't do this on POR
   10A    6AF8     CLRF 0xff8, ACCESS
37:
38:                    bcf __FPFLAGS,RND,0 // Initialize rounding flag for floating
point libs
   10C    9C01     BCF 0x1, 0x6, ACCESS
39:
40:                    _endasm
41:                    _do_cinit ();

    <<<<<   MIDDLE SECTION OF THIS FILE HAS BEEN DELETED >>>>>

---  F:\1Mydata\lab\MPLAB\c2asm_intro\test.c  ---------------------------------------
-------------
```

```
1:                 /**********************************************************
2:                  * File: c2asm_into.c
3:                  * Project: c to Assembly Language Equivalency
4:                  * Author: Class
5:                  * Updated: 2/14/10
6:                  **********************************************************/
7:
8:                 //Process Specific definitions
9:                 #include <p18f1220.h>
10:
11:                // main() is the entry point to the program and does not accept or
return parameters.
12:                void main(void)
   0CA    CFD9     MOVFF 0xfd9, 0xfe6
   0CC    FFE6     NOP
   0CE    CFE1     MOVFF 0xfe1, 0xfd9
   0D0    FFD9     NOP
   0D2    0E02     MOVLW 0x2
   0D4    26E1     ADDWF 0xfe1, F, ACCESS
13:                {
14:                      int count;
15:
16:                      count = count + 1;
   0D6    CFDE     MOVFF 0xfde, 0x2
   0D8    F002     NOP
   0DA    CFDD     MOVFF 0xfdd, 0x3
   0DC    F003     NOP
   0DE    0E01     MOVLW 0x1
   0E0    2602     ADDWF 0x2, F, ACCESS
   0E2    0E00     MOVLW 0
   0E4    2203     ADDWFC 0x3, F, ACCESS
   0E6    C002     MOVFF 0x2, 0xfde
   0E8    FFDE     NOP
   0EA    C003     MOVFF 0x3, 0xfdd
   0EC    FFDD     NOP
17:
18:                } //main()
   0EE    0E02     MOVLW 0x2
   0F0    5CE1     SUBWF 0xfe1, W, ACCESS
   0F2    E202     BC 0xf8
   0F4    6AE1     CLRF 0xfe1, ACCESS
   0F6    52E5     MOVF 0xfe5, F, ACCESS
   0F8    6EE1     MOVWF 0xfe1, ACCESS
   0FA    52E5     MOVF 0xfe5, F, ACCESS
   0FC    CFE7     MOVFF 0xfe7, 0xfd9
   0FE    FFD9     NOP
   100    0012     RETURN 0
```

❖ Map file (c2asm_into.map)

```
MPLINK 4.02, Linker
Linker Map File - Created Sun Feb 28 15:10:08 2010

                                 Section Info
                  Section        Type    Address   Location Size(Bytes)
                ---------  ---------  ---------  ---------  ---------
               _entry_scn       code   0x000000    program   0x000006
                   .cinit    romdata   0x00002a    program   0x000002
               _cinit_scn       code   0x00002c    program   0x00009e
             .code_test.o       code   0x0000ca    program   0x000038
             _startup_scn       code   0x000102    program   0x000018
         .idata_c018i.o_i    romdata   0x00011a    program   0x000000
         .romdata_c018i.o    romdata   0x00011a    program   0x000000
            .code_c018i.o       code   0x00011a    program   0x000000
          .idata_test.o_i    romdata   0x00011a    program   0x000000
          .romdata_test.o    romdata   0x00011a    program   0x000000
                MATH_DATA      udata   0x000000       data   0x000002
                 .tmpdata      udata   0x000002       data   0x000002
                   .stack      udata   0x000080       data   0x000040
           .udata_c018i.o      udata   0x0000c0       data   0x00000a
           .idata_c018i.o      idata   0x0000ca       data   0x000000
            .udata_test.o      udata   0x0000ca       data   0x000000
            .idata_test.o      idata   0x0000ca       data   0x000000
            SFR_UNBANKED0      udata   0x000f80       data   0x000080



                              Program Memory Usage
                               Start         End
                           ---------   ---------
                            0x000000    0x000005
                            0x00002a    0x000119
            246 out of 4376 program addresses used, program memory utilization is 5%



                              Symbols - Sorted by Name
                     Name    Address   Location    Storage File
                ---------  ---------  ---------  --------- ---------
       ___return_lbl00000   0x000004    program     static
       ___return_lbl00001   0x000118    program     static
       ___return_lbl00002   0x0000c8    program     static

    <<<<<   MIDDLE SECTION OF THIS FILE HAS BEEN DELETED >>>>>

                  TBLPTRL   0x000ff6       data     extern
                   TBLPTR   0x000ff6       data     extern
                  TBLPTRH   0x000ff7       data     extern
                  TBLPTRU   0x000ff8       data     extern
                      PCL   0x000ff9       data     extern
                       PC   0x000ff9       data     extern
                   PCLATH   0x000ffa       data     extern
                   PCLATU   0x000ffb       data     extern
               STKPTRbits   0x000ffc       data     extern
                   STKPTR   0x000ffc       data     extern
                     TOSL   0x000ffd       data     extern
                      TOS   0x000ffd       data     extern
                     TOSH   0x000ffe       data     extern
                     TOSU   0x000fff       data     extern
```

6.2. Indirect Addressing (INDFn)

Compliers use Indirect addressing to generate relocatable code and dynamically change the memory address to be accessed. This section outlines indirect addressing based on PICmicro's implementation.

Indirect addressing uses three FSRn registers (FSR0, FSR1, FSR2) as pointers to the data memory location that is to be read or written. These register contain the address of the data memory being affected. The size of memory on PICmicro is 4096 bytes which means an address is 12 bits and requires two bytes to store. The three sets of indirect addressing subsystems are addressed by:

- FSR0: composed of FSR0H:FSR0L "FEA : FE9"
- FSR1: composed of FSR1H:FSR1L "FE2 : FE1"
- FSR2: composed of FSR2H:FSR2L "FDA : FD9"

Typically, FSRn is initialized by LFSR instruction as shown by the following example:

```
        LFSR    FSR0, 0x1065            ;Sets the initial address of indirect addressing subsystem 0 to
1065h
```

The data is read or written by accessing one of the special function registers associated with each FSRn. The following list provides an overview of each of the special registers and their corresponding activity:

- Indirectly access register (pointed to by FSRn), then do nothing else (no change) – INDFn
- Indirectly access register, then auto-decrement FSRn (post-decrement) – POSTDECn
- Indirectly access register, then auto-increment FSRn (post-increment) – POSTINCn
- Auto-increment FSRn, then indirectly access register (pre-increment) – PREINCn
- Use the value in the WREG register as an offset to FSRn. It will not modify the value of the WREG or the FSRn register after an indirect access (nochange) – PLUSWn

Summary of all the Special Function Registers associated with each Indirect Addressing Subsystem (IASn):

IAS 0	**IAS 1**	**IAS 2**
FSR0H : FSR0L	FSR1H : FSR1L	FSR2H :FSR2L
INDF0	INDF1	INDF2
POSTDEC0	POSTDEC1	POSTDEC2
POSTINC0	POSTINC1	POSTINC2
PREINC0	PREINC1	PREINC2
PLUSW0	PLUSW1	PLUSW2

❖ **Example -** Describe the function performed by the following Code Segment.

```
            LFSR    FSR0,0x100
NEXT:       SETF    POSTINC0
            BTFSS   FSR0H, 1
            GOTO    NEXT
CONT:       BRA     CONT
```

Solution:

Sets locations 0x100 through 0x1FF to the value 0xFF

❖ **Example** - Write a PICmicro code segment using indirect addressing to move content from location 100-150h to 2050-2000h.

Solution:

```
                LFSR FSR0, 0x100
                LFSR FSR1, 0x2050
                MOVLW 0x51
                MOVWF 0x80

MOVE_IT:
                MOVFF POSTINC0, POSTDEC1
                DECF 0x80
                BNZ MOVE_IT
```

- **Example** - Show the content of memory that has been changed by the following code segment and their new contents.

```
                MOVLW           12h
                MOVWF           FSR0L
                MOVLW           23h
                MOVWF           FSR0H
                MOVLW           15h
                MOVWF           POSTDEC0
                ADDLW           2h
                MOVWF           POSTINC0
                ADDLW           5
                MOVWF           INDF0
```

Solution:

Location and content in Hex →

Location	Content
2311	17h
2312	1Ch

6.3. Functions / Procedures

- Code Entry Point, main()

  - Syntax

    // entry into the code
    void main (void)
    {
    Statements

    }

  - Example - Disassembly Listing

```
1:                 /**********************************************************
2:                  * File: c2asm_into.c
3:                  * Project: c to Assembly Language Equivalency
4:                  * Author: Class
5:                  * Updated: 2/14/10
6:                  **********************************************************/
7:
8:                 //Process Specific definitions
9:                 #include <p18f1220.h>
10:
11:                // main() is the entry point to the program and does not accept or return
parameters.
12:                void main(void)
   0CA    CFD9     MOVFF 0xfd9, 0xfe6
   0CC    FFE6     NOP
   0CE    CFE1     MOVFF 0xfe1, 0xfd9
   0D0    FFD9     NOP
   0D2    0E02     MOVLW 0x2
   0D4    26E1     ADDWF 0xfe1, F, ACCESS
13:                {
14:                   int count;
15:
16:                   count = count + 1;
   0D6    CFDE     MOVFF 0xfde, 0x2
   0D8    F002     NOP
   0DA    CFDD     MOVFF 0xfdd, 0x3
   0DC    F003     NOP
   0DE    0E01     MOVLW 0x1
   0E0    2602     ADDWF 0x2, F, ACCESS
   0E2    0E00     MOVLW 0
   0E4    2203     ADDWFC 0x3, F, ACCESS
   0E6    C002     MOVFF 0x2, 0xfde
   0E8    FFDE     NOP
   0EA    C003     MOVFF 0x3, 0xfdd
   0EC    FFDD     NOP
17:
18:                } //main()
   0EE    0E02     MOVLW 0x2
   0F0    5CE1     SUBWF 0xfe1, W, ACCESS
   0F2    E202     BC 0xf8
   0F4    6AE1     CLRF 0xfe1, ACCESS
   0F6    52E5     MOVF 0xfe5, F, ACCESS
   0F8    6EE1     MOVWF 0xfe1, ACCESS
   0FA    52E5     MOVF 0xfe5, F, ACCESS
   0FC    CFE7     MOVFF 0xfe7, 0xfd9
   0FE    FFD9     NOP
   100    0012     RETURN 0
```

- Function/Procedures

  - Syntax
    - Call
      name(argument list, if any);

    - Definition
      name(argument list, if any)
      argument declarations, if any
      {
          declarations and statements, if any
      }

  - Example - Disassembly Listing

```
---  F:\1Mydata\lab\MPLAB\c2asm_intro\test.c
14:                 void main(void)
   0CA    CFD9      MOVFF 0xfd9, 0xfe6
   0CC    FFE6      NOP
   0CE    CFE1      MOVFF 0xfe1, 0xfd9
   0D0    FFD9      NOP
   0D2    0E02      MOVLW 0x2
   0D4    26E1      ADDWF 0xfe1, F, ACCESS
15:                 {
16:                    int count;
17:
18:
19:                    fun_add(count);   // Call
   0D6    CFDE      MOVFF 0xfde, 0xfe6
   0D8    FFE6      NOP
   0DA    CFDD      MOVFF 0xfdd, 0xfe6
   0DC    FFE6      NOP
   0DE    D80C      RCALL 0xf8
   0E0    52E5      MOVF 0xfe5, F, ACCESS
   0E2    52E5      MOVF 0xfe5, F, ACCESS
20:
21:                 } //main()
   0E4    0E02      MOVLW 0x2
   0E6    5CE1      SUBWF 0xfe1, W, ACCESS
   0E8    E202      BC 0xee
   0EA    6AE1      CLRF 0xfe1, ACCESS
   0EC    52E5      MOVF 0xfe5, F, ACCESS
   0EE    6EE1      MOVWF 0xfe1, ACCESS
   0F0    52E5      MOVF 0xfe5, F, ACCESS
   0F2    CFE7      MOVFF 0xfe7, 0xfd9
   0F4    FFD9      NOP
   0F6    0012      RETURN 0
22:
23:                 // function definition
24:                 int fun_add(int op)
   0F8    CFD9      MOVFF 0xfd9, 0xfe6
   0FA    FFE6      NOP
   0FC    CFE1      MOVFF 0xfe1, 0xfd9
   0FE    FFD9      NOP
25:                 {
26:                    op = op + 1;
   100    0E01      MOVLW 0x1
   102    6EE7      MOVWF 0xfe7, ACCESS
   104    0EFD      MOVLW 0xfd
   106    CFDB      MOVFF 0xfdb, 0x2
   108    F002      NOP
   10A    0EFE      MOVLW 0xfe
   10C    CFDB      MOVFF 0xfdb, 0x3
   10E    F003      NOP
   110    50E7      MOVF 0xfe7, W, ACCESS
   112    2602      ADDWF 0x2, F, ACCESS
   114    0E00      MOVLW 0
   116    2203      ADDWFC 0x3, F, ACCESS
   118    0EFD     MOVLW 0xfd
   11A    C002     MOVFF 0x2, 0xfdb
   11C    FFDB     NOP
   11E    0EFE     MOVLW 0xfe
   120    C003     MOVFF 0x3, 0xfdb
   122    FFDB     NOP
27:                    return (op);
   124    0EFD     MOVLW 0xfd
   126    CFDB     MOVFF 0xfdb, 0x2
   128    F002     NOP
   12A    0EFE     MOVLW 0xfe
   12C    CFDB     MOVFF 0xfdb, 0x3
   12E    F003     NOP
   130    C002     MOVFF 0x2, 0xff3
   132    FFF3     NOP
   134    C003     MOVFF 0x3, 0xff4
   136    FFF4     NOP
   138    D000     BRA 0x13a
28:                } // fun_add
   13A    52E5     MOVF 0xfe5, F, ACCESS
   13C    CFE7     MOVFF 0xfe7, 0xfd9
   13E    FFD9     NOP
   140    0012     RETURN 0
```

6.4. Data Types

- Constant
  - Syntax
    #define CONSTANT_NAME Value
  - Assembly Equivalent
    CONSTANT_NAME equ Value
  - Examples
    - C Example
      #define CONST_EX 10
      #define CHAR_EX ‘h’
    - Assembly Equivalent
      CONST_EX equ 10
      CHAR_EX equ ‘h’
- Character
  - Syntax
    char ch_ex;
  - Example – .lst file

```
17:                        char ch_ex;
18:
19:                        ch_ex = 'h';
   0EC    0E68     MOVLW 0x68
   0EE    6EDF     MOVWF 0xfdf, ACCESS
```

- String
  String is a list of characters terminated by a null character ‘\0’. C language does not support string declaration as type different from Character.

- Integer
  - Syntax
    int int_ex; // typically size of int is equal to processor word size
  - Example – Disassembly Listing (PIC micro implements integer in 16 bits)

```
14:                        int int_ex;
15:
16:                        int_ex = 0x29;
   0D6    0E29     MOVLW 0x29
   0D8    6EDE     MOVWF 0xfde, ACCESS
   0DA    6ADD     CLRF 0xfdd, ACCESS
17:
```

- Float “single-precision floating point” & Double “double-precision floating point”
  PICmicro has implemented float and double based on IEEE single precision format discussed in Chapter 5. The float range is shown below:

$$2^{-126} = 1.17549435\,E-38 \quad to \quad 2^{128} * (2-2^{-15}) = 6.80564693\,E+38$$

  - Syntax

    ```
    float  float_ex;            // uses IEEE Single precision format
    double  double_ex;                  // uses IEEE Single precision format
    ```

  - Example – Disassembly Listing

Float

```
14:                        float flt_ex;
15:
16:                        flt_ex = 29.35;
   0D6    0ECD     MOVLW 0xcd
   0D8    6EDE     MOVWF 0xfde, ACCESS
   0DA    0ECC     MOVLW 0xcc
   0DC    6EDE     MOVWF 0xfde, ACCESS
   0DE    0EEA     MOVLW 0xea
   0E0    6EDE     MOVWF 0xfde, ACCESS
   0E2    0E41     MOVLW 0x41
   0E4    6EDD     MOVWF 0xfdd, ACCESS
   0E6    52DD     MOVF 0xfdd, F, ACCESS
   0E8    52DD     MOVF 0xfdd, F, ACCESS
17:
```

Double

```
14:                        double dbl_ex;
15:
16:                        dbl_ex = 29.35;
   0D6    0ECD     MOVLW 0xcd
   0D8    6EDE     MOVWF 0xfde, ACCESS
   0DA    0ECC     MOVLW 0xcc
   0DC    6EDE     MOVWF 0xfde, ACCESS
   0DE    0EEA     MOVLW 0xea
   0E0    6EDE     MOVWF 0xfde, ACCESS
   0E2    0E41     MOVLW 0x41
   0E4    6EDD     MOVWF 0xfdd, ACCESS
   0E6    52DD     MOVF 0xfdd, F, ACCESS
   0E8    52DD     MOVF 0xfdd, F, ACCESS
17:
```

- Pointers

  - Syntax
    type *var_p;   // declares pointer to a variable of declared type
    type var;   // declaring a variable of declared type

    var= *var_p;   // Assign the content of the address pointed to by a pointer to a variable
    var_p = &var;   // Assign address of variable to the pointer variable

  - Example – Disassembly Listing

```
13:                {
14:                       char chv;              // decalre a variable
15:                       char *chp;             // declare a pointer
16:
17:                       chv = 'h';             // set variable to h
   0D6    0E68     MOVLW 0x68
   0D8    6EDF     MOVWF 0xfdf, ACCESS
18:                       chp = &chv;            // move content of pointer to variable
   0DA    CFD9     MOVFF 0xfd9, 0x2
   0DC    F002     NOP
   0DE    CFDA     MOVFF 0xfda, 0x3
   0E0    F003     NOP
   0E2    0E01     MOVLW 0x1
   0E4    C002     MOVFF 0x2, 0xfdb
   0E6    FFDB     NOP
   0E8    0E02     MOVLW 0x2
   0EA    C003     MOVFF 0x3, 0xfdb
   0EC    FFDB     NOP
19:                       *chp = 'g';            // set the location pointed to by chp to g
   0EE    0E01     MOVLW 0x1
   0F0    CFDB     MOVFF 0xfdb, 0xfe9
   0F2    FFE9     NOP
   0F4    0E02     MOVLW 0x2
   0F6    CFDB     MOVFF 0xfdb, 0xfea
   0F8    FFEA     NOP
   0FA    0E67     MOVLW 0x67
   0FC    6EEF     MOVWF 0xfef, ACCESS
20:                       chv = *chp;            // move content of pointer to variable
   0FE    0E01     MOVLW 0x1
   100    CFDB     MOVFF 0xfdb, 0xfe9
   102    FFE9     NOP
   104    0E02     MOVLW 0x2
   106    CFDB     MOVFF 0xfdb, 0xfea
   108    FFEA     NOP
   10A    CFEF     MOVFF 0xfef, 0xfdf
   10C    FFDF     NOP
21:
```

- Arrays

  - Syntax
    type ar_name[size dim1];   // declare an array

  - Example – Disassembly Listing

```
   0CA    CFD9     MOVFF 0xfd9, 0xfe6
   0CC    FFE6     NOP
   0CE    CFE1     MOVFF 0xfe1, 0xfd9
   0D0    FFD9     NOP
   0D2    0E0A     MOVLW 0xa
   0D4    26E1     ADDWF 0xfe1, F, ACCESS
13:                {
14:                       char ch[10];          // decalre a variable
15:
16:                       ch[0] = 'a';          // set the first element to a
   0D6    0E61     MOVLW 0x61
   0D8    6EDF     MOVWF 0xfdf, ACCESS
17:                       ch[9] = 'j';          // set the last element to j
   0DA    0E6A     MOVLW 0x6a
   0DC    6EF3     MOVWF 0xff3, ACCESS
   0DE    0E09     MOVLW 0x9
   0E0    CFF3     MOVFF 0xff3, 0xfdb
   0E2    FFDB     NOP
18:
```

- Structures

  - Syntax
    - Defining a new type

      // new type
      struct new-type{
          list of declarations
      };

      struct new_type new_struct; // defines a variable new_struct of the type new_type

    - Defining a new structure
      // new type
      struct {
          list of declarations
      } new_struct1, new_struct2;

  - Example – Disassembly Listing

```
12:                 void main(void)
   0CA    CFD9      MOVFF 0xfd9, 0xfe6
   0CC    FFE6      NOP
   0CE    CFE1      MOVFF 0xfe1, 0xfd9
   0D0    FFD9      NOP
   0D2    0E0F      MOVLW 0xf
   0D4    26E1      ADDWF 0xfe1, F, ACCESS
13:                 {
14:                        char name;
15:                        struct record
16:                        {
17:                               int id;
18:                               char name[10];
19:                               int grade;
20:                        };
21:                        struct record student = {1,"Great", 100};
   0D6    0E01      MOVLW 0x1
   0D8    6EF3      MOVWF 0xff3, ACCESS
   0DA    CFF3      MOVFF 0xff3, 0xfdb
   0DC    FFDB      NOP
   0DE    0E02      MOVLW 0x2
   0E0    6ADB      CLRF 0xfdb, ACCESS
   0E2    50D9      MOVF 0xfd9, W, ACCESS
   0E4    0F03      ADDLW 0x3
   0E6    6EE9      MOVWF 0xfe9, ACCESS
   0E8    CFDA      MOVFF 0xfda, 0xfea
   0EA    FFEA      NOP
   0EC    0E47      MOVLW 0x47
   0EE    6EEE      MOVWF 0xfee, ACCESS
   0F0    0E72      MOVLW 0x72
   0F2    6EEE      MOVWF 0xfee, ACCESS
   0F4    0E65      MOVLW 0x65
   0F6    6EEE      MOVWF 0xfee, ACCESS
   0F8    0E61      MOVLW 0x61
   0FA    6EEE      MOVWF 0xfee, ACCESS
   0FC    0E74      MOVLW 0x74
   0FE    6EEE      MOVWF 0xfee, ACCESS
   100    6AEE      CLRF 0xfee, ACCESS
   102    0E3E      MOVLW 0x3e
   104    6EEE      MOVWF 0xfee, ACCESS
   106    6EEE      MOVWF 0xfee, ACCESS
   108    6EEE      MOVWF 0xfee, ACCESS
   10A    6EEE      MOVWF 0xfee, ACCESS
   10C    0E64      MOVLW 0x64
   10E    6EF3      MOVWF 0xff3, ACCESS
   110    0E0D     MOVLW 0xd
   112    CFF3     MOVFF 0xff3, 0xfdb
   114    FFDB     NOP
   116    0E0E     MOVLW 0xe
   118    6ADB     CLRF 0xfdb, ACCESS
22:
23:                } //main()
   11A    0E0F     MOVLW 0xf
   11C    5CE1     SUBWF 0xfe1, W, ACCESS
   11E    E202     BC 0x124
   120    6AE1     CLRF 0xfe1, ACCESS
   122    52E5     MOVF 0xfe5, F, ACCESS
   124    6EE1     MOVWF 0xfe1, ACCESS
   126    52E5     MOVF 0xfe5, F, ACCESS
   128    CFE7     MOVFF 0xfe7, 0xfd9
   12A    FFD9     NOP
   12C    0012     RETURN 0
```

Note: NOP instructions listed above are actually the second word of two-word instruction.

- **Example** – The following C program segment:

```
// Available data memory start at 0x80

Struct {
        char name[30];              // 1 byte/char
        int sid;                    // integer is 2 bytes
        char grade[2];
} Students [20];
```

a) Find the location of student[2].grade[1].
b) Find the location for student [9].sid;

Solutions

6.5. Program Flow Controls

- If-Then-Else

  - Syntax

```
if (condition) {
        statements
}
else {                  // else is optional
        statements
}
```

  - Example – Disassembly Listing

```
12:                void main(void)
   0CA    CFD9     MOVFF 0xfd9, 0xfe6
   0CC    FFE6     NOP
   0CE    CFE1     MOVFF 0xfe1, 0xfd9
   0D0    FFD9     NOP
   0D2    0E02     MOVLW 0x2
   0D4    26E1     ADDWF 0xfe1, F, ACCESS
13:                {
14:                       int count=8;
   0D6    0E08     MOVLW 0x8
   0D8    6EDE     MOVWF 0xfde, ACCESS
   0DA    6ADD     CLRF 0xfdd, ACCESS
15:
16:                       if (count < 5){
   0DC    CFDE     MOVFF 0xfde, 0x2
   0DE    F002     NOP
   0E0    CFDD     MOVFF 0xfdd, 0x3
   0E2    F003     NOP
   0E4    90D8     BCF 0xfd8, 0, ACCESS
   0E6    5003     MOVF 0x3, W, ACCESS
   0E8    E604     BN 0xf2
   0EA    0E05     MOVLW 0x5
   0EC    5C02     SUBWF 0x2, W, ACCESS
   0EE    0E00     MOVLW 0
   0F0    5803     SUBWFB 0x3, W, ACCESS
   0F2    E20D     BC 0x10e
17:                               count = count + 5;
   0F4    CFDE     MOVFF 0xfde, 0x2
   0F6    F002     NOP
   0F8    CFDD     MOVFF 0xfdd, 0x3
   0FA    F003     NOP
   0FC    0E05     MOVLW 0x5
   0FE    2602     ADDWF 0x2, F, ACCESS
   100    0E00     MOVLW 0
   102    2203     ADDWFC 0x3, F, ACCESS
   104    C002     MOVFF 0x2, 0xfde
   106    FFDE     NOP
   108    C003     MOVFF 0x3, 0xfdd
   10A    FFDD     NOP
18:                       }
19:                       else{
   10C    D00C     BRA 0x126
20:                               count = count - 5;
   10E    CFDE     MOVFF 0xfde, 0x2
   110    F002     NOP
   112    CFDD     MOVFF 0xfdd, 0x3
   114    F003     NOP
   116    0E05     MOVLW 0x5
   118    5E02     SUBWF 0x2, F, ACCESS
   11A    0E00     MOVLW 0
   11C    5A03     SUBWFB 0x3, F, ACCESS
   11E    C002     MOVFF 0x2, 0xfde
   120    FFDE     NOP
   122    C003     MOVFF 0x3, 0xfdd
   124    FFDD     NOP
21:                        }
22:
23:                } //main()
   126    0E02     MOVLW 0x2
   128    5CE1     SUBWF 0xfe1, W, ACCESS
   12A    E202     BC 0x130
   12C    6AE1     CLRF 0xfe1, ACCESS
   12E    52E5     MOVF 0xfe5, F, ACCESS
   130    6EE1     MOVWF 0xfe1, ACCESS
   132    52E5     MOVF 0xfe5, F, ACCESS
   134    CFE7     MOVFF 0xfe7, 0xfd9
   136    FFD9     NOP
   138    0012     RETURN 0
```

- **Example** – Write the following C code in its equivalent PICmicro assembly:

```
Swap (char a, char b){
        char temp;
        if (a > b){
                temp = a;
                a = b;
                b = temp;
        } else  a = b;
        return;
}
```

Solution

- While Loop

  - Syntax
    while (condition){
            statements
    }

  - Example – Disassembly Listing

```
11:                 // main() is the entry point to the program and does not accept or return
parameters.
12:                 void main(void)
   0CA    CFD9      MOVFF 0xfd9, 0xfe6
   0CC    FFE6      NOP
   0CE    CFE1      MOVFF 0xfe1, 0xfd9
   0D0    FFD9      NOP
   0D2    0E02      MOVLW 0x2
   0D4    26E1      ADDWF 0xfe1, F, ACCESS
13:                 {
14:                        int count;
15:                        while (count <= 10){
   0D6    CFDE      MOVFF 0xfde, 0x2
   0D8    F002      NOP
   0DA    CFDD      MOVFF 0xfdd, 0x3
   0DC    F003      NOP
   0DE    3403      RLCF 0x3, W, ACCESS
   0E0    E204      BC 0xea
   0E2    5002      MOVF 0x2, W, ACCESS
   0E4    080A      SUBLW 0xa
   0E6    0E00      MOVLW 0
   0E8    5403      SUBFWB 0x3, W, ACCESS
   0EA    E305      BNC 0xf6
   0F4    D7F0      BRA 0xd6
16:                               count++;
   0EC    2ADF      INCF 0xfdf, F, ACCESS
   0EE    0E01      MOVLW 0x1
   0F0    E301      BNC 0xf4
   0F2    2ADB      INCF 0xfdb, F, ACCESS
17:                        }
18:
19:
20:                 } //main()
   0F6    0E02      MOVLW 0x2
   0F8    5CE1      SUBWF 0xfe1, W, ACCESS
   0FA    E202      BC 0x100
   0FC    6AE1      CLRF 0xfe1, ACCESS
   0FE    52E5      MOVF 0xfe5, F, ACCESS
   100    6EE1      MOVWF 0xfe1, ACCESS
   102    52E5      MOVF 0xfe5, F, ACCESS
   104    CFE7      MOVFF 0xfe7, 0xfd9
   106    FFD9      NOP
   108    0012      RETURN 0
```

- For Loop

  - Syntax
    for (Intialization$_{\text{optional}}$;Condition$_{\text{optional}}$; Action$_{\text{optional}}$){
    statements
    }

  - Example – Disassembly Listing

```
12:                void main(void)
   0CA    CFD9     MOVFF 0xfd9, 0xfe6
   0CC    FFE6     NOP
   0CE    CFE1     MOVFF 0xfe1, 0xfd9
   0D0    FFD9     NOP
   0D2    0E02     MOVLW 0x2
   0D4    26E1     ADDWF 0xfe1, F, ACCESS
13:                {
14:                       int count;
15:                       for (count=0; count<10 ; count++){
   0D6    6ADE     CLRF 0xfde, ACCESS
   0D8    6ADD     CLRF 0xfdd, ACCESS
   0DA    CFDE     MOVFF 0xfde, 0x2
   0DC    F002     NOP
   0DE    CFDD     MOVFF 0xfdd, 0x3
   0E0    F003     NOP
   0E2    90D8     BCF 0xfd8, 0, ACCESS
   0E4    5003     MOVF 0x3, W, ACCESS
   0E6    E604     BN 0xf0
   0E8    0E0A     MOVLW 0xa
   0EA    5C02     SUBWF 0x2, W, ACCESS
   0EC    0E00     MOVLW 0
   0EE    5803     SUBWFB 0x3, W, ACCESS
   0F0    E205     BC 0xfc
   0F2    2ADF     INCF 0xfdf, F, ACCESS
   0F4    0E01     MOVLW 0x1
   0F6    E301     BNC 0xfa
   0F8    2ADB     INCF 0xfdb, F, ACCESS
   0FA    D7EF     BRA 0xda
16:                       }
17:
18:
19:                } //main()
```

6.6. Additional Resources

- Kernighan & Ritchie. The C Programing Language, (1978) Prentice-Hall
- Peterson. Computer Organization and Design, (2007) Elsevier Service.
- Staff. Microchip PIC 18F1220/1320 Data Sheet. (2004) Microchip Technology Incorporated.

6.7. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 7. PERFORMANCE

Key concepts and Overview

- CPU Performance and Relating Factors
- Evaluating Performance and Bench Marking
- Performance Bench Marking Design
- Additional Resources

7.1. CPU Performance and Relating Factors

As discussed earlier, performance is growing in importance as criteria of microprocessor design. As the memory size and functionality have grown, performance becomes one of the most important factors in design of computer system.

The first step in understanding, analyzing and designing a system with respect to performance is to agree on these key definitions.

- ❖ Defining Performance

  Depending on your application, you may emphasize a subset of performance attributes in your selection or design of computer systems. For example, if you are designing an enterprise system for a fortune 500 corporation, you will have different needs than if you are designing a gaming computer system for a hobbyist.

  Here are a few terminologies to consider:

  - ➢ Performance and Execution Time

    It is common to use Performance and Execution Time to refer to the overall performance of a system. The total time required for the computer to complete a task, including disk access, memory access, I/O activities, Operating system overhead, CPU execution time and others may be referred to as the execution time. As shown below, execution time is inversely proportional to the performance as shown below:

    $$Performance = \frac{1}{Execution\,Time}$$

    To maximize performance is to minimize the Execution time. If computer X runs faster than Y, then it is said that computer X is n time faster than Y, when:

    $$n = \frac{Performance_X}{Performance_Y} = \frac{Execution\,Time_Y}{Execution\,Time_X}$$

    It can be confusing to use the terms “increasing” and “decreasing” in conjunction with “performance” and “execution time” since they denote the opposites. For example, an increase in performance is desirable. On the other hand, increased execution time is undesirable. So to remove this confusion, the industry typically uses the words “Improve performance” or “Improve Execution time “ instead of the terms “increase performance” or “decrease execution time”

  - ➢ Measuring Performance

    Computer performance is measured in term of execution time in seconds per program.

    - Elapsed Time

      Elapsed Time is defined by the wall-clock time, elapsed time, also called “response time,” refers to the time a program takes to execute from the start to the end of as is observed by the user. This includes all aspects of activities such as memory, execution and delays.

    - CPU Execution Time or CPU Time (corresponding to CPU performance)

      A processor is typically shared amongst multiple programs. CPU execution time or CPU time, is the time the processor, is actually executing the program. Note that in this case, CPU time does not include activities such as memory access, disk access and others.

      CPU time can be further classified as:

- User CPU Time
  CPU time spent on the program

- System CPU Time
  CPU time spent on the operating system performing tasks on behalf of the program.

- Clock or System Clock
  Computer systems have a main clock. The Clock's frequency (f) and period (T=1/f) are used in discussion of bottom up performance.

As mentioned earlier, measuring performance depends on many factors and the type of applications being considered. Therefore, there are a variety of techniques in measuring performance. In some cases, the designer has to consider CPU performance in terms of number of instructions and number of cycles per instruction. This method is referred to as the bottom up method.

On the other hand, there are cases when the underlying application and system code are not available or are too complex for an instruction by instruction performance measurement. In these type cases, benchmark performance measure will be used.

❖ CPU Performance Factors

When we have access to the code and the application is not too complex, we are able to do a detailed analysis of the number of clock cycles the CPU takes to perform a specific task.

➢ CPU time in terms of CPU Clock is one the most basic measurements of performance.

CPU Execution time for a program =
(# of CPU Clock Cycle for a Program) * (Clock Cycle Time)
or

CPU Execution time for a program =
(# of CPU Clock Cycle for a Program) / (Clock freq. or rate)

So, to improve performance is to either use less clock cycles or reduce clock cycle time. But many techniques to reduce number of clock cycles will also increase the clock cycle time.

- Example
  Let's say your computer is running GTW (Good Time Waster) game with a 1.2 second response time.

  Company VGC (Very Good Computer) is claiming that their new computer, VIC, instruction set requires only half the clock cycles of your computer and the Clock Frequency is 20% higher.

  What would you expect the GTW game response time to be on VIC.

  Solution:
  For your computer, we have CPU Execution time = A / B = 1.2 seconds where:
  A is # of CPU cycles and
  B is the CPU clock frequency

  For VIC, we have CPU execution time = (A/2) / (1.2 B)
  = (A/B)(1/2.4) = (1.2 sec)(1/2.4) = 0.5 Sec.

As a result, VIC would be a higher-performing computer compared to the current computer.

- Example – What's the execution time of PIC micro system with 10 Mhz clock running the following code:

```
            CLRF      0x30
Loop:       MOVWF     0x29
            DECF      0x30
            ADDWF     0x31
            BNZ       Loop
```

Solution:

- Average Clock Cycle per Instruction
  If you have access to the code but the application is becoming more complex, you can simplify have your performance measure by using average Clock Cycles per Instruction (CPI) measure. At the core, CPI is the average number of cycles to execute an instruction in a code segment. CPI allows one to count # of instruction and not have the responsibility to know the number of cycles required by each instruction.

  Using the above Definition we can write the following relationships:

  # CPU Clock Cycles =
  (# of instructions for a program) * (Average Clock Cycle Per Instruction, CPI)

  Using the above relationship we can find the CPU Time:

  CPU Time = (# CPU Clock Cycles) *(Clock Period)
  = (# CPU Clock Cycles) / (Clock Frequency)

  Therefore

  CPU Time = (# Instruction per program) * (CPI) * (Clock Period)

  Another way to write the same thing:

  CPU Time = (# Instruction per program) * (CPI) / (Clock Frequency)

  The above equation is especially useful, since it separates the three key factors (Number of Instructions, CPI and Clock Frequency) that affect performance

- Time (CPU Time or CPU Execution Time) is the measure of performance
  In general the following relationship can be used to figure out the performance:

$$Time = \frac{Seconds}{\text{Program}} = \frac{Instructions}{\text{Program}} * \frac{Clock\ Cycle}{Instruction} * \frac{Seconds}{Clock\ Cycle}$$

Where:

Components of Performance	Units of Measure
CPU Execution time for a program (Time)	Seconds per program
Instruction count	Instructions executed for the program
Clock Cycles per instruction (CPI)	Average number of clock cycles per instruction
Clock cycle time	Seconds per clock cycle

Average CPI requires a fair amount of work to determine and it also varies from code segment to code segment. For more accurate calculations at the time, you may need to use the following formula:

CPU Clock Cycle = $\sum_{i=1}^{n}(CPI_i * C_i)$ Where

C_i is the count of the number of instructions in Class i
CPI_i is the average number of cycles per instruction for Class i
n is the number of instruction classes

- Effect of Software Components on CPU Performance
  Another way to study performance is understanding the effect of software development components on the performance. The following table attempts to represent the relationship:

Software Components	What is affected?	How it is affected?
Algorithm	Instruction Count CPI	Algorithms say how the work is done at a high level which affect the type of instruction and number of instructions used
Programming Language	Instruction Count CPI	Programming language will directly affect the instructions used.
Compiler	Instruction Count CPI	Complier is the component that writes the assembly code so decisions here would also have an effect on instruction count and CPI.

- Example
  An algorithm for sorting has been designed and compiled using Java. The execution code generated include three classes of code:

15 instruction of ”A” class with 3 clocks cycles per instruction (3 CPIs)
12 instruction of ”B” class with 5 clock cycles per instruction (5 CPIs)
20 instruction of ”C” class with 12 clock cycles per instruction (12 CPIs)

The same sorting algorithm has been designed and complied using C. The execution code generated includes three classes of code:

30 instruction of ”X” class with 2 clocks cycles per instruction (2 CPIs)
8 instruction of ”Y” class with 7 clock cycles per instruction (7 CPIs)

15 instruction of ”Z” class with 10 clock cycles per instruction (10 CPIs)

Which solution provide you with a better performance? And what is the total execution time for the better performing solution if the code was running on a PICmicro with the clock speed of 8 MHz.

Solution:

For Java → CPU Clock Cycles = $\sum_{i=1}^{n}(CPI_i * C_i)$ = (15x3) + (12x5) + (20x12) = 345 clock cycles

For C→ CPU Clock Cycles = $\sum_{i=1}^{n}(CPI_i * C_i)$ = (30x2) + (8x7) + (15x10) = 266 clock cycles

C language solution has better performance

At clock frequency of f=8 MHz, Cycle time is T = 1/f = 125 * 10^{-9} Seconds.

Therefore: Total execution time = (CPU Clock Cycle) * T = 266 * (125*10^{-9}) seconds

- Example – Estimate execution time for a PICmicro processor with an 8 MHz external crystal to sort an array with 1000 integers using bubble sort. Below is an example of Bubble Sort C code segment:

```
swapped = 0;
while (swapped == 0){
    for (i=0 , i≤(1000-2), i++){
        if (A(i) > A[i+1)){
            temp=A(i);
            A(i) = A(i+1);
            A(i+1) = temp;
            swapped = 1;
        }
    }    // for
 }   //while
```

Solution:
“Student Exercise”

7.2. Evaluating Performance

Most users run a set of programs or applications on their computer systems to accomplish their tasks. Their main interested is on the performance of the total system, not each piece individually. Additionally, the user does not have access to the code for analysis, even if the user has the time and interest to do so. Typically in this situation, the instruction by instruction or bottom up performance comparison is not workable due to complexity and lack of access.

Most commonly, the application source code is not available. Even if available, there are multiple layers of application code, which would require the user to run some standard set of tasks and compare the response time of the system. For most types of solutions, there are a set of programs or instruction chosen to predict performance for a particular work load and application. These types of performance measuring codes are called benchmarks. Benchmarks are a good way for users to choose the appropriate type of computers without having to analyze each individual component of the hardware and software.

So if you are planning to select a computer system for a Computer Aided Design (CAD) application, then your benchmark program should include common instructions used in CAD program. On the other hand, if you plan to use the computer system for gaming, you may consider a different set of benchmarks for example emphasis on graphics capability of the system.

One word of caution, companies understand this fact and continually work to show their products in the best possible light. They may knowingly or unintentionally design benchmarks that are not representative of the final performance for your specific need, so “Buyer beware”. Fortunately, most industries and application have standard benchmarks which are unbiased.

Benchmarks may focus on a specific portion of the system or attempt to predict end-to-end performance of a system. Some examples of Benchmarks include:

- SPEC23b99 benchmark
  Designed to evaluate Web Server performance

- EEMBC benchmark
  Designed to evaluate Embedded System performance

- SPEC CPU 2000 latest release of SPEC CPU
  Designed to measure the CPU performance with respect to integer and floating point operations.

- Transaction Processing Performance Council
  Designed to measure database and transaction processing performance. They even list cost $/tpmc.

There are thousands of benchmarks. Each is designed for a specific set of applications and use. It is recommended that the user research additional benchmarks.

7.3. Performance Bench Marking Design

Bench marking is an important step in understanding performance need and selecting solution that meet the required needs. The following three parameters are integral to the decision:

- Key attributes of application/solution
- Scenarios that exercises key attributes
- Run benchmark on all solutions

7.4. Additional Resources

- Stallins. Computer Organization & Architecture: Designing for Performance, (2003) Prentice Hall
- Peterson. Computer Organization and Design, (2007) Elsevier Service.
- Robertazzi. Computer Network and Systems: Queuing Theory & Performance Evaluation, (2008) Springer
- Lilja. Measuring Computer Performance, (2000) Cambridge University Press

7.5. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 8. MEMORY & STORAGE HIERARCHY

Key concepts and Overview

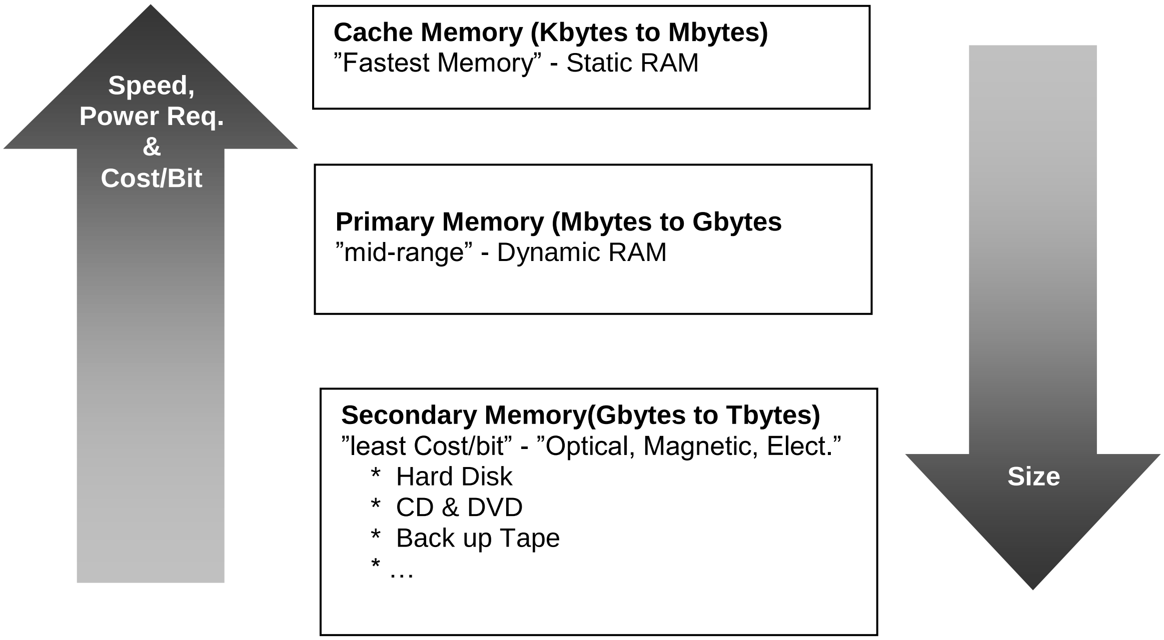
- ❖ Memory & Storage Basics
- ❖ Cache Memory
- ❖ Primary Memory
- ❖ Secondary Storage
- ❖ Virtual Memory
- ❖ Additional Resources

8.1. Memory & Storage Overview

Computer memory structure is driven by four main factors: size, speed, power and cost. It is rare if not impossible to find a computer user who does not want the largest and fastest memory available. The factors that limit the users are the cost and power requirements.

These factors has resulted in memory structures which attempt to minimize the size of high speed memory used, while striving to maximize the utilization of the fast memory that's available. Virtual Memory Management attempts to map the slow memory into higher speed memory such as cache for frequently executed instruction of data.

In a typical computer the following memory types are found:

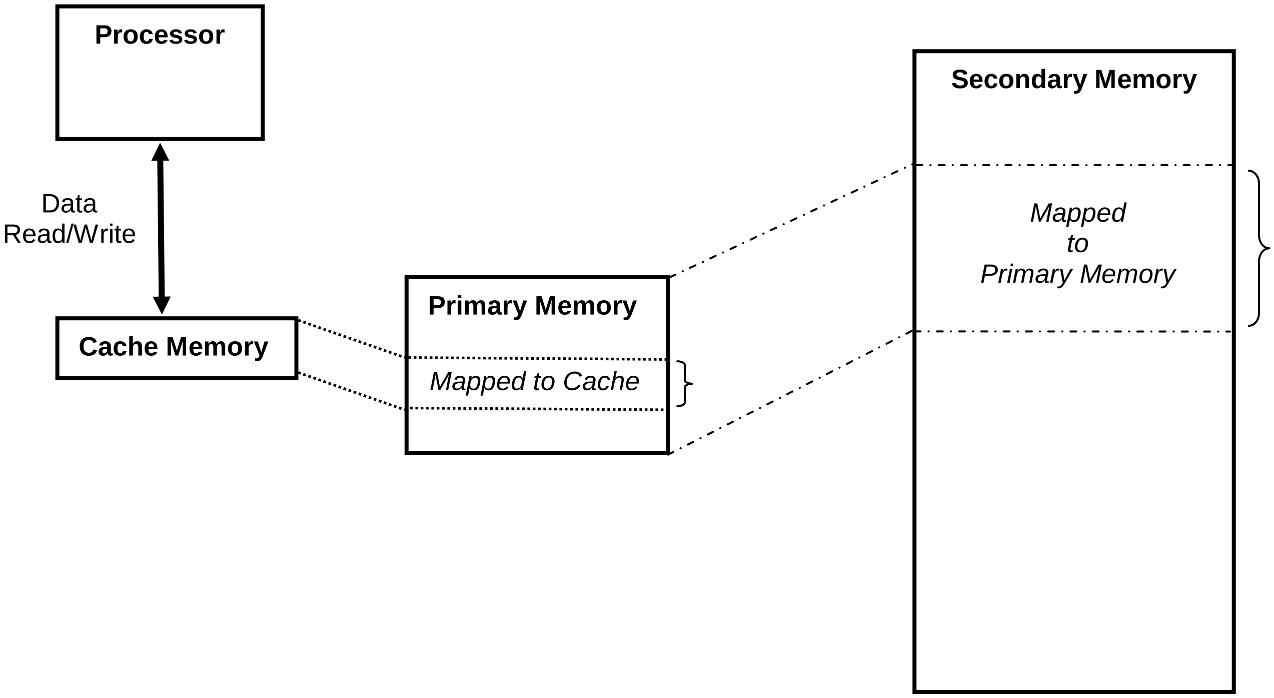


Cache Memory (Kbytes to Mbytes)
"Fastest Memory" - Static RAM

Primary Memory (Mbytes to Gbytes
"mid-range" - Dynamic RAM

Secondary Memory(Gbytes to Tbytes)
"least Cost/bit" - "Optical, Magnetic, Elect."
* Hard Disk
* CD & DVD
* Back up Tape
* …

In a typical computer system, these three types of storage are related to each other as shown in the following diagram:


Processor
Data
Read/Write
Cache Memory
Primary Memory
Mapped to Cache
Secondary Memory
Mapped
to
Primary Memory

8.2. Cache Memory

Cache contains a partial copy of primary memory content that can be accessed by the processor faster than any other type of memory. If the processor can find the code/data needed in Cache (referred to as a cache hit) resulting in improved performance. If the information is not in cache it has to be copied form primary memory which is slower. Therefore, designers continually improve the Caching policy to maximize the Cache hit rate (also known as hit ratio). In addition to policy, Cache type, cost and size is continually changing.

To complete this section, the reader is expected to perform the following exploration exercise:

- Exploration Exercise

  For your current PC, identify the following Cache parameters:
  - Memory Type and read/write time
  - Cost/bit of the memory
  - The size of the cache
  - Cache policy

  Solution:

  Student Exercise

8.3. Primary Memory

Even though Primary Memory is typically orders of magnitude larger than cache, it only contains a partial copy of secondary storage content. In a typical computer, processor is unable to directly execute code from secondary storage. Virtual Memory Manager (software component) is responsible for ensuring that the required data/program is copied into the primary memory for execution and access by the processor. if the program/data is already in primary memory, the performance would be much better than when information is in secondary memory and has to be copied to primary memory – this condition is referred to as a miss.

To complete this section, the reader is expected to perform the following exploration exercise:

- Exploration Exercise
  For your current PC, Identify the following Primary Memory parameters:
  - Memory Type and read/write time
  - Cost/bit of the memory
  - The size of the Primary Memory (How does it compare to cache size)

Solution:

Student Exercise

8.4. Secondary Storage

Secondary storage contain all the programs and data that can be used by the computer but first they have to be moved to primary memory and/or cache. Although secondary storage technology is more stable than other memory types, secondary storage has continued to become faster, larger in size and lower cost/bit.

To complete this section, the reader is expected to perform the following exploration exercise:

- Exploration Exercise

  For your current PC, answer the following questions:

  - How many secondary storage is installed in your PC?
  - What are the cost/bit for each type of secondary storage types in your PC?
  - What each of secondary storage types are used for?

  Solution:

  Student Exercise

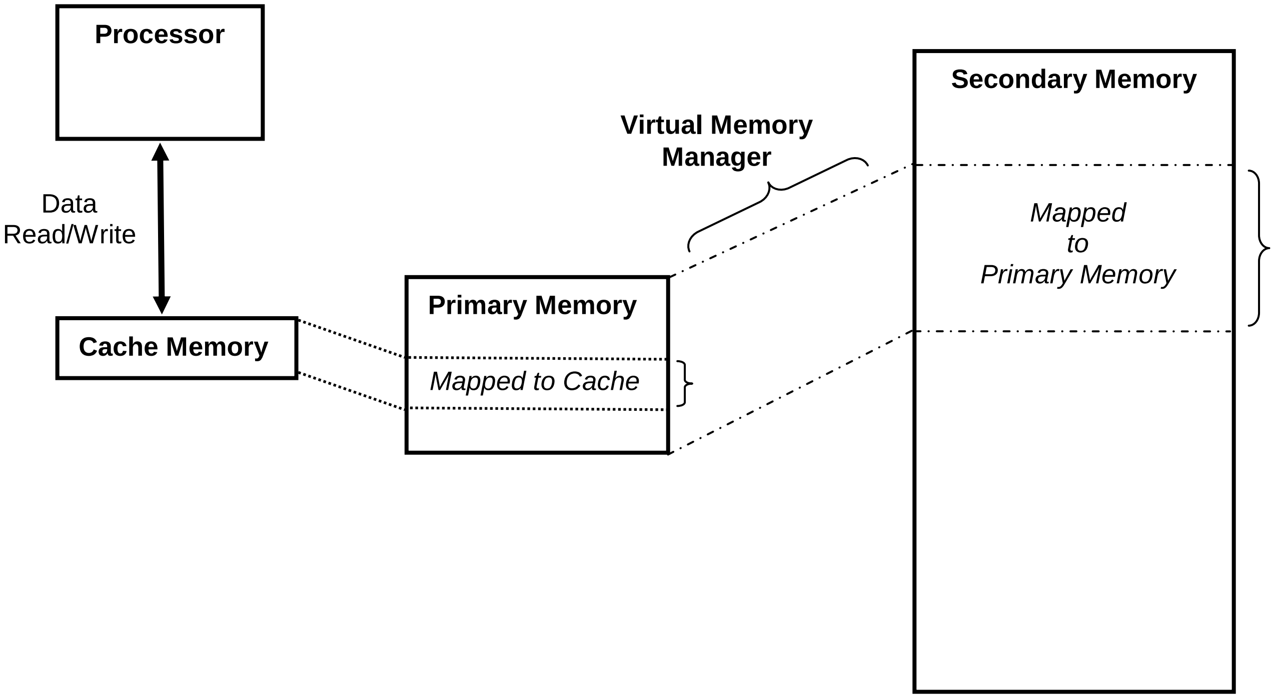
8.5. Virtual Memory Management

Virtual Memory Manager allows each process/program to use all the space that is allocated to it from primary and secondary storage seamlessly. In other words, the application running in a given process can use all the space required without having to explicitly move data between the primary memory and secondary storage. The Virtual Memory Manger does all the work of moving data to create a continuous memory transparently.

The simplest view of Virtual Memory Manager is a system program that bring in blocks of Secondary Storage into primary memory as their content are required by the processor. If the system is running out of primary memory, then a block that is no longer needed is over written by the new block.

The block to be over-written is chosen based on the Virtual Memory Manager's Policy. Some common ones are First-in-First-out (FIFO) or Last-In-First-Out (LIFO). Of course there are much more complex policies based on the need and usage model of the system.

The following diagram shows the role of Virtual Memory Management in the context of memory types:



To complete this section, the reader is expected to perform the following exploration exercise:

❖ Exploration Exercise
For your current PC, answer the following questions:
- What is the name of the Virtual Memory Manager and the vendor?
- What is the smallest block size that is copies?
- What is the replacement policy when Primary Memory is full?

Solution:
Student Exercise

8.6. Additional Resources

- Peterson. Computer Organization and Design, (2007) Elsevier Service.
- Gorman. Understanding the Linux Virtual Memory Manager, (2004) Prentice Hall
- Staff. Microchip PIC 18F1220/1320 Data Sheet, (2004) Microchip Technology In.

8.7. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 9. CONCURRENCY IN COMPUTING

Key concepts and Overview

- Overview of Parallelism
- Pipelining
- Multi-processing
- Multi-core Processors
- Multi-Processor Systems
- Additional Resources

9.1. Overview of Parallelism

As the performance has become the key parameter used in selecting a computer system, the vendors are increasing investment in development of parallel computing solutions in order achieve higher performance.

One way to characterize the computer system parallelism options is outlined below:

- Pipelining
  A pipelined processor is able to operate on multiple instruction concurrently. For example a single processor fetching one instruction while executing another instruction.

- Multi-processing
  A single processor allowing multiple processes to remain active by giving each process a portion of time. A functioning multi-process will provide user with the impression that all processors are running simultaneously.

- Multi-core Processors
  In this case there are multiple processor cores but still within a single processor which allows for multiple processes to run at the same time. Cores typically share peripherals and memory.

- Multi-Processor Systems
  Many processors executing one or more programs simultaneously.

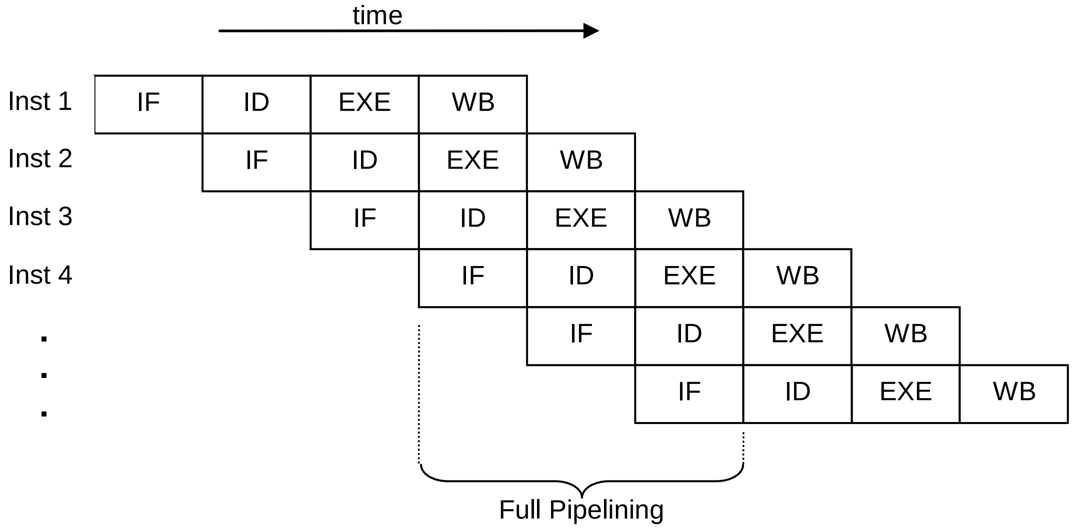
Although parallelism improves speed, it also adds complexity and overhead to the system. It is important that sufficient performance improvement is gained to justify the additional complexity and cost associated with the selected parallelism technique. Also, a given system design may incorporate one or more of the above options.

9.2. Pipelining

An instruction pipeline is a technique used in the design of computer systems and processors to increase performance. Pipelining reduces cycle time of a processor which leads to increased instruction throughput, the number of instructions that can be executed in a unit of time. The instruction processing is divided into four distinct phases:

1) Instruction fetch (IF)
2) Instruction decode (ID)
3) Execute (EXE)
4) Write Back (WB)

In a non-pipelined system, these phased are completed sequentially while in a pipelined system there is some level of parallelism. If a system is able to execute a new instruction every cycle, it is said to be fully pipelined. The following diagram show a fully pipelined system:



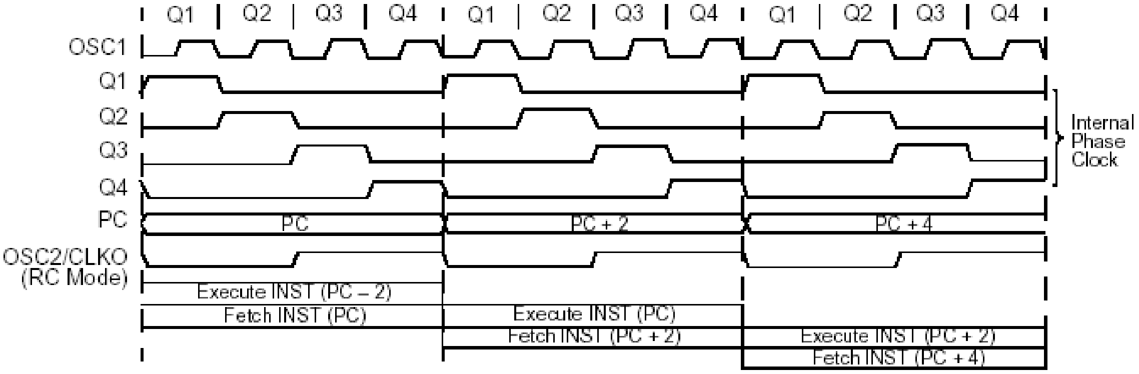
The major Advantages of pipelining is reduction of cycle time of the processor leading to increased instruction processing speed and performance. In achieving this improvement, designer have to be aware and handle three of issues:

1) The processor executes only a single instruction at a time. This prevents branch delays (in effect, every branch is delayed) and problems with serial instructions being executed concurrently. Consequently the design is simpler and cheaper to manufacture.

2) The instruction latency in a non-pipelined processor is slightly lower than in a pipelined equivalent. This is due to the fact that extra flip flops must be added to the data path of a pipelined processor.

3) A non-pipelined processor will have a stable instruction bandwidth. The performance of a pipelined processor is much harder to predict and may vary more widely between different programs.

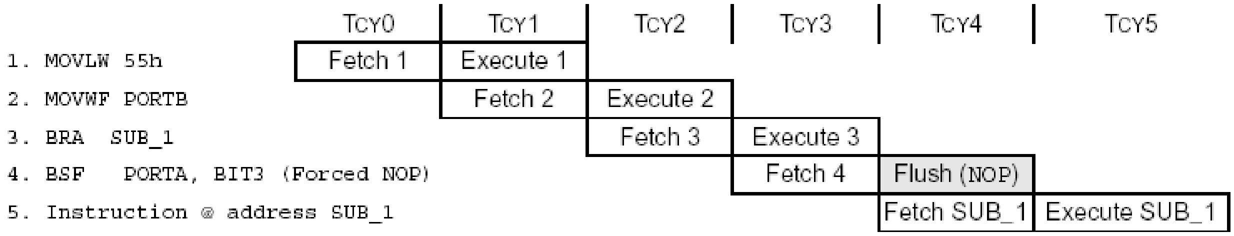
PICmicro is also a pipelined processor. But before discussing the pipelining, we need to talk about the instruction cycles. The clock input (from OSC1) is internally divided by four to generate four non-overlapping Quarter clocks, namely Q1, Q2, Q3 and Q4. Internally, the Program Counter (PC) is incremented every Q1, the instruction is fetched from the program memory and latched into the instruction register in Q4. The instruction is decoded and executed during the following Q1 through Q4. The clocks and instruction execution flow are shown in the following figure:



As mentioned earlier an “Instruction Cycle” consists of four Q cycles (Q1,Q2, Q3 and Q4). The instruction fetch and execute are pipelined such that fetch takes one instruction cycle, while decode and execute takes another instruction cycle. However, due to the pipelining, each instruction effectively executes in one cycle. If an instruction causes the program counter to change (e.g., GOTO), then two cycles are required to complete the instruction.

In PICmirco, a fetch cycle begins with the Program Counter (PC) incrementing in Q1. In the execution cycle, the fetched instruction is latched into the “Instruction Register” (IR) in cycle Q1. This instruction is then decoded and executed during the Q2, Q3 and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write).

An example of PICmicro pipelined instruction execution is shown in the following figure:



	TCY0	TCY1	TCY2	TCY3	TCY4	TCY5
1. MOVLW 55h	Fetch 1	Execute 1				
2. MOVWF PORTB		Fetch 2	Execute 2			
3. BRA SUB_1			Fetch 3	Execute 3		
4. BSF PORTA, BIT3 (Forced NOP)				Fetch 4	Flush (NOP)	
5. Instruction @ address SUB_1					Fetch SUB_1	Execute SUB_1

All instructions are single cycle, except for any program branches. These take two cycles, since the fetch instruction is “flushed” from the pipeline, while the new instruction is being fetched and then executed.

When a programmer (or compiler) writes assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. This assumption may be invalidated by pipelining. When this causes a program to behave incorrectly, the situation is known as a hazard. Various techniques for resolving hazards such as forwarding and stalling exist.

The instruction cycle is easy to implement, however, it is extremely inefficient. The answer to this inefficiency is pipelining. Pipelining improves performance significantly in program code execution. This is done by decreasing the time that any component inside the CPU is idle. Pipelining does not completely cancel out idle time in a CPU but a significant impact is made. Processors with pipelining are organized inside into (stages) which can semi-independently work on separate jobs. Each stage is organized and linked into a 'chain' so each stage's output is inputted to another stage until the job is done. This organization of the processor allows overall processing time to be significantly reduced.

Unfortunately, not all instructions are independent. In a simple pipeline, completing an instruction may require 5 stages. To operate at full performance, this pipeline will need to run 4 subsequent independent instructions while the first is completing. If 4 instructions that do not depend on the output of the first instruction are not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. Fortunately, techniques such as forwarding can significantly reduce the cases where stalling is required. While pipelining can in theory increase performance over an unpopulated core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

To complete this section, the reader is expected to perform the following exploration exercise:

- Exploration Exercise
  For your current PC:
  - Identify the pipeline approach used
  - Show the content of the full pipeline

  Solution:
  Student Exercise

9.3. Multi-processing

Commercially viable computer in today's market including multi-processing capable operating systems where multiple processes and applications may be active. The single available processor is shared amongst the active processes which means at any point in time only one process is being executed. From the user's point of view, it seems that application are running simultaneously (Other the occasional choppiness when the system is over used) since each process is given sufficient time to respond to user commands frequently.

To complete this section, the reader is expected to perform the following exploration exercise:

❖ Exploration Exercise
For your current PC, answer the following::

- How many processes are active currently and which processor is using the highest percentage of the processor (i.e. task manager on the Microsoft Windows has the needed data)?
- What is the maximum number of processes that can be active at the same time?

Solution:
Student Exercise

9.4. Multi-core Processors

Today's PCs have multi-core which basically means that there are multiple processor core embedded into a single processor chip. With the help of coordinating software (typically part of operating system), applications and/or processes are divided amongst the cores to execute. Ideally, multiple cores deliver higher performance. This is not guaranteed since the management overhead may consume any gains made from the multi-core set up.

To complete this section, the reader is expected to perform the following exploration exercise:

- Exploration Exercise
  In the current PC market:
  - Identify a PC with multi-core processor.
  - For the identified processor, what is function of each core and how are the cores managed?
  - What is the expected performance improvement from the selected multi-core compared to an equivalent single core system.

Solution:

Student Exercise

9.5. Multi-Processor Systems

Multi-Processor systems are typically used for specialized application that are highly processor intensive. Over time, there has been various attempts to develop multi-processor systems that are able to efficiently run any program. But we continue to see the best multi-processor performance for applications design specifically for the multi-processor design.

To complete this section, the reader is expected to perform the following exploration exercise:

- ❖ Exploration Exercise

  In the current market:

  - Identify a multi-processor system and the vendor
  - For the identified system, what are the topology of processor (how are the processors connected)?
  - Does this system only runs specialized applications or is able to improve performance of general purpose applications.

  Solution:

  Student Exercise

9.6. Additional Resources

- Jordan. Fundamentals of Parallel Processing, (2003) Prentice Hall
- Peterson. Computer Organization and Design, (2007) Elsevier Service.
- Roosta. Parallel Processing and Parallel Algorithms, (1999) Springer-Verlag

9.7. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

CHAPTER 10. NETWORKING

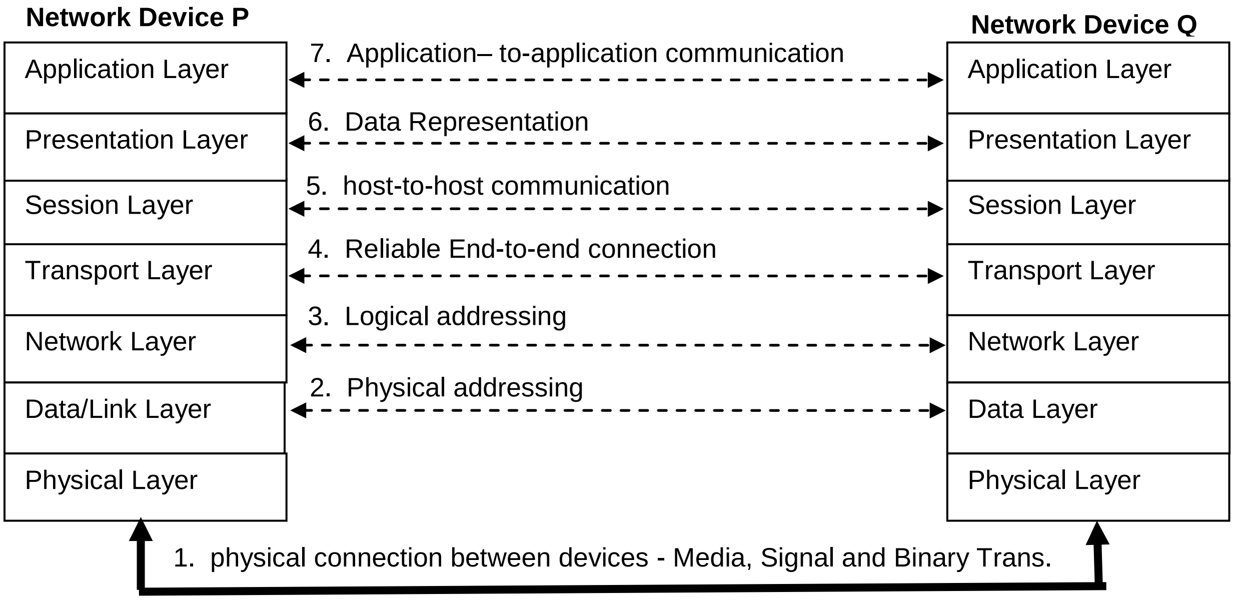
Key concepts and Overview

- Networking Overview & OSI Model
- Medial Layers (Physical, Link & Network)
- Host Layers (Transport, Session , Presentation and Application)
- Additional Resources

10.1. Networking Overview & OSI Model

Networking is an integral part of computing world and numerous designs have been developed to meet the needs of the computing industry. The best way to discuss networking is to use the abstract Open System Interconnection Reference Model (OSI Model) developed as part of the Open System Interconnection (OSI) initiative in 1970s by the International Organization for Standardization (ISO).

OSI Model groups the network functionality into seven layers. Each layer relies on the layers below to complete its task. In communicating across the network, the two parties to the communication will have defined protocol at each layer of the model as shown below between two networked devices (P & Q):



Layers are typically divided into two groups based on where they are implemented, in the host or the networking interface:

- Media Layers – Physical, Data/Link and Network Layers
- Host Layers – Transport, Session, Presentation and Application layers

The following sections provide additional description of each of the seven layers in the above two categories with the most common implementation examples of each layer.

10.2. Medial Layers (Physical, Data/Link & Network)

Physical layer defines the electromagnetic and physical specifications for device connection to the network. Items included in this description of this layer includes connector, voltage/current, timing and other specifications.

Data/Link layer is responsible for defining and packaging fixed size data that include physical address. Also it has processes to ensure that a packet is reliability delivered by the physical layer to the intended physical address. If not, then it would have steps to either flag an error or attempt to correct the problem by re-transmission.

For examples of Data and Physical layer implementations refer to IEEE 802.3 (Wired LAN), IEEE 802.11 (wireless LAN) and IEEE 902.16 (WiMax) and IEEE 802.15 (Bluetooth-Personal Network).

Networking layer provides reliable transfer of variable length data sequences from one device to one or more devices on the network. This layer performs the routing function for the devices. Router provides functionality from physical to networking layer. The most commonly known Network layer implementation is the Internet Protocol which is commonly refer to as IP. IP enable variable length data to travel through multiple hops from source to the intended destination. Network layer also serve as the interface with Host layers.

10.3. Host Layers (Transport, Session , Presentation and Application)

Transport layer is the lowest layer of the Host layers. It provides reliable data transfer services between end users. It uses flow control, error control, segmentation, retransmission to ensure the end user data has successfully been transmitted and received. Again the best known Transport layer implementation example is Transmission Control Protocol (TCP) which is used in most systems. TCP/IP referring to Transmission Control Protocol and Internet Protocol are one of the most popular implementation of Network and Transport layer in use today.

Session Layer manages the connection between networked devices. Session layer uses the lower layers of OSI to establish, manager and terminate connections between applications. Socket (also called shared socket) is an example of Session layer implementation for TCP/IP environment. Sockets allows devices to connection application across the network or within the same system. A process read from the socket to receive the data from another process and the process sends data by writing into the socket. Communicating processor may be on the same physical computer (Local) or across the network in another physical computer and location (Remote).

Presentation layer allows mapping of different data format to be translated into session protocol data units that can be transmitted through session layer services. MIME Protocol is a Session layer implementation example which is designed to enable sending and receiving emails across variety of email applications.

Application layer is the highest level of OSI layer. As the name implies this is the layer that contain software application which interfaces with the user. Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP) are two examples of Application layer implementation.

The following section provide additional description of each layers into two groups:

- ❖ Exploration Exercise
  In the current market:
  - Identify an network enabled application.
  - Map the functionality/components of the selected application to the OSI model.

Solution:
Student Exercise

10.4 Additional Resources

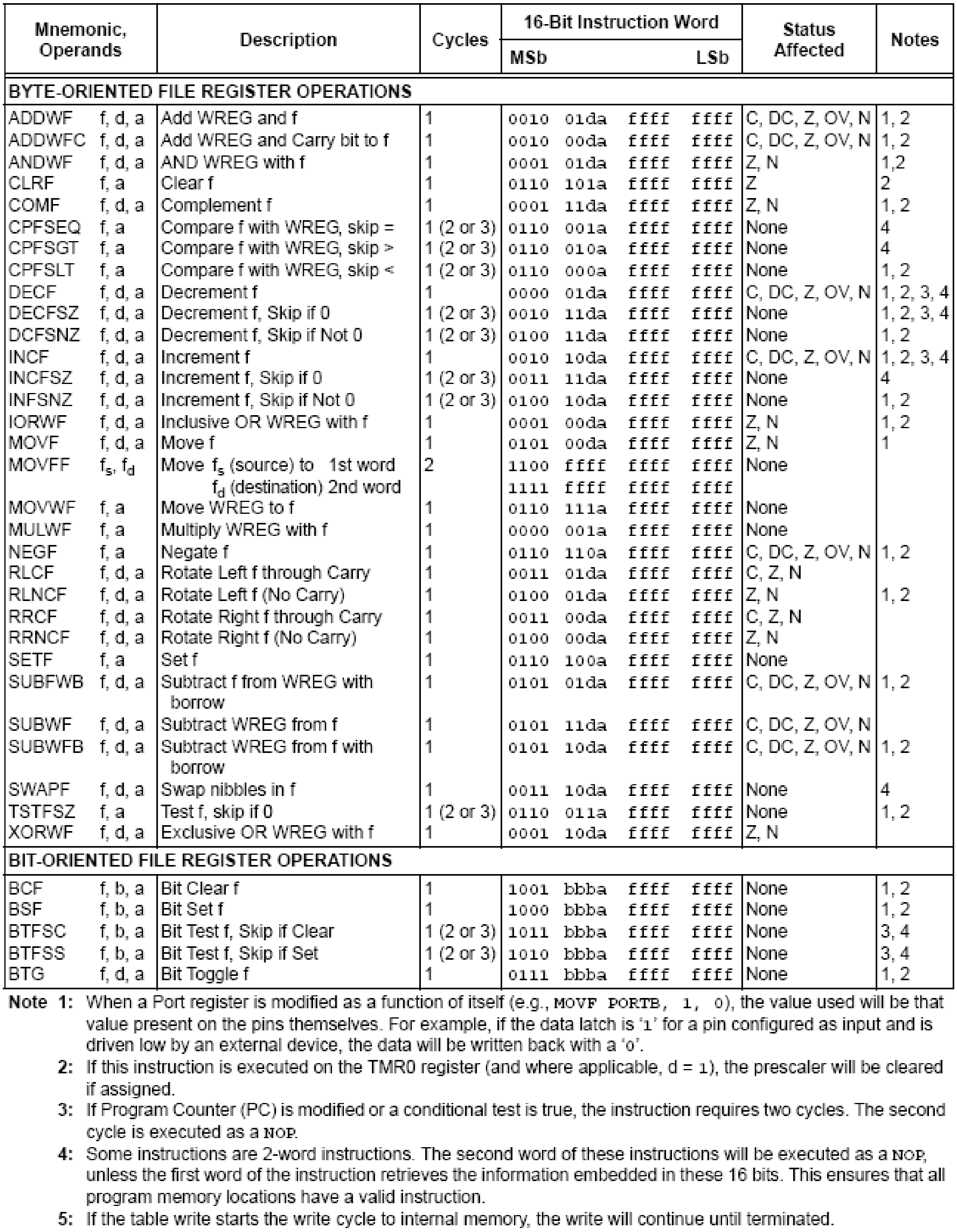
- Kurose. Computing Networking, (2010) Addison-Wesley.
- Peterson. Computer Organization and Design, (2007) Elsevier Service.
- Lekkus. Network Processors, (2003) McGraw Hill.

10.5. Problems

Refer to www.EngrCS.com or online course page for complete solved and unsolved problem set.

APPENDIX A. PICMICRO INSTRUCTION SET SUMMARY

Source: Microchip Data Sheet



Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word				Status Affected	Notes
				MSb			LSb		
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	0010	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and Carry bit to f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1,2
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f, a	Compare f with WREG, skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	1, 2
DECF	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if Not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if Not 0	1 (2 or 3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word f_d (destination) 2nd word	2	1100 1111	ffff ffff	ffff ffff	ffff ffff	None	
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	1, 2
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N	
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N	1, 2
RRCF	f, d, a	Rotate Right f through Carry	1	0011	00da	ffff	ffff	C, Z, N	
RRNCF	f, d, a	Rotate Right f (No Carry)	1	0100	00da	ffff	ffff	Z, N	
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None	
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	1, 2
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1 (2 or 3)	0110	011a	ffff	ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N	
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC	f, b, a	Bit Test f, Skip if Clear	1 (2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS	f, b, a	Bit Test f, Skip if Set	1 (2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2

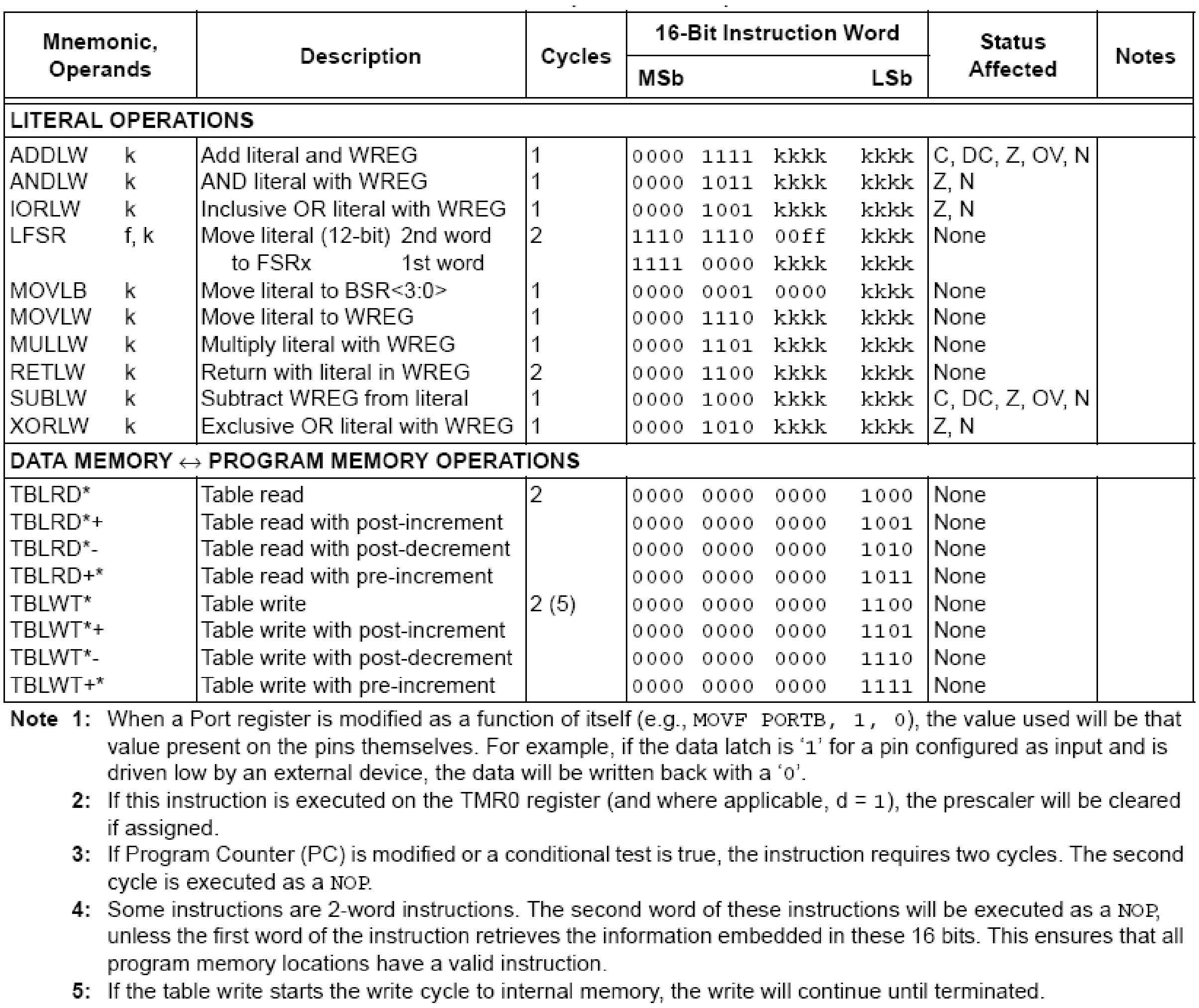
Note 1: When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), the value used will be that value present on the pins themselves. For example, if the data latch is '`1`' for a pin configured as input and is driven low by an external device, the data will be written back with a '`0`'.

2: If this instruction is executed on the TMR0 register (and where applicable, d = `1`), the prescaler will be cleared if assigned.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOP`.

4: Some instructions are 2-word instructions. The second word of these instructions will be executed as a `NOP`, unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

5: If the table write starts the write cycle to internal memory, the write will continue until terminated.



Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word	Status Affected	Notes
				MSb LSb		
LITERAL OPERATIONS						
ADDLW	k	Add literal and WREG	1	0000 1111 kkkk kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000 1011 kkkk kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000 1001 kkkk kkkk	Z, N	
LFSR	f, k	Move literal (12-bit) 2nd word to FSRx 1st word	2	1110 1110 00ff kkkk 1111 0000 kkkk kkkk	None	
MOVLB	k	Move literal to BSR<3:0>	1	0000 0001 0000 kkkk	None	
MOVLW	k	Move literal to WREG	1	0000 1110 kkkk kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000 1101 kkkk kkkk	None	
RETLW	k	Return with literal in WREG	2	0000 1100 kkkk kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000 1000 kkkk kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000 1010 kkkk kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS						
TBLRD*		Table read	2	0000 0000 0000 1000	None	
TBLRD*+		Table read with post-increment		0000 0000 0000 1001	None	
TBLRD*-		Table read with post-decrement		0000 0000 0000 1010	None	
TBLRD+*		Table read with pre-increment		0000 0000 0000 1011	None	
TBLWT*		Table write	2 (5)	0000 0000 0000 1100	None	
TBLWT*+		Table write with post-increment		0000 0000 0000 1101	None	
TBLWT*-		Table write with post-decrement		0000 0000 0000 1110	None	
TBLWT+*		Table write with pre-increment		0000 0000 0000 1111	None	

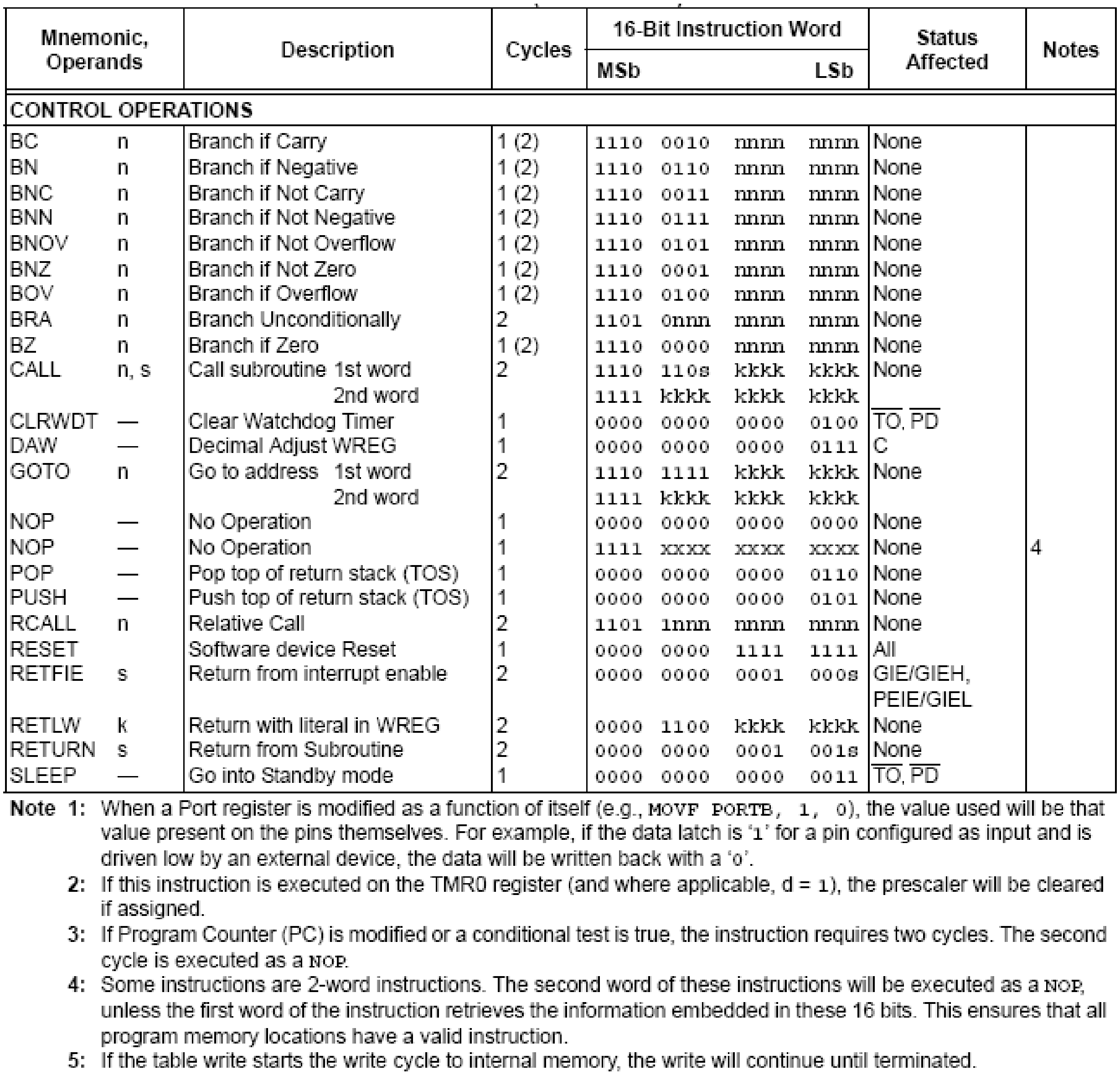
Note 1: When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and where applicable, d = 1), the prescaler will be cleared if assigned.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

4: Some instructions are 2-word instructions. The second word of these instructions will be executed as a NOP, unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

5: If the table write starts the write cycle to internal memory, the write will continue until terminated.



Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word				Status Affected	Notes
				MSb			LSb		
CONTROL OPERATIONS									
BC	n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None	
BN	n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None	
BNN	n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None	
BNOV	n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None	
BNZ	n	Branch if Not Zero	1 (2)	1110	0001	nnnn	nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110	0100	nnnn	nnnn	None	
BRA	n	Branch Unconditionally	2	1101	0nnn	nnnn	nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110	0000	nnnn	nnnn	None	
CALL	n, s	Call subroutine 1st word	2	1110	110s	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
CLRWDT	—	Clear Watchdog Timer	1	0000	0000	0000	0100	$\overline{TO}$, $\overline{PD}$	
DAW	—	Decimal Adjust WREG	1	0000	0000	0000	0111	C	
GOTO	n	Go to address 1st word	2	1110	1111	kkkk	kkkk	None	
		2nd word		1111	kkkk	kkkk	kkkk		
NOP	—	No Operation	1	0000	0000	0000	0000	None	
NOP	—	No Operation	1	1111	xxxx	xxxx	xxxx	None	4
POP	—	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None	
PUSH	—	Push top of return stack (TOS)	1	0000	0000	0000	0101	None	
RCALL	n	Relative Call	2	1101	1nnn	nnnn	nnnn	None	
RESET		Software device Reset	1	0000	0000	1111	1111	All	
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
RETURN	s	Return from Subroutine	2	0000	0000	0001	001s	None	
SLEEP	—	Go into Standby mode	1	0000	0000	0000	0011	$\overline{TO}$, $\overline{PD}$	

Note 1: When a Port register is modified as a function of itself (e.g., `MOVF PORTB, 1, 0`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and where applicable, d = 1), the prescaler will be cleared if assigned.

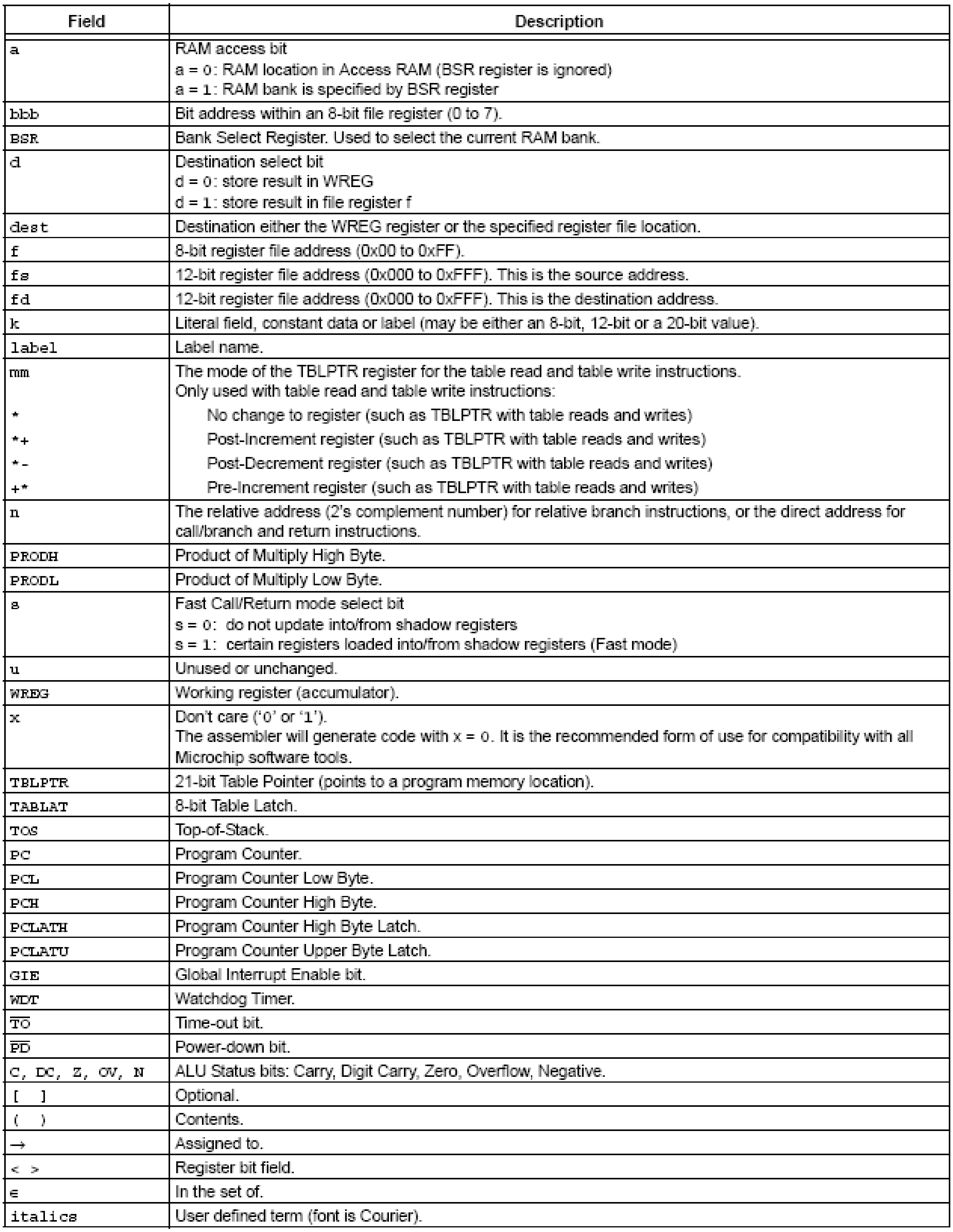
3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a `NOP`.

4: Some instructions are 2-word instructions. The second word of these instructions will be executed as a `NOP`, unless the first word of the instruction retrieves the information embedded in these 16 bits. This ensures that all program memory locations have a valid instruction.

5: If the table write starts the write cycle to internal memory, the write will continue until terminated.

APPENDIX B. PICMICRO OPCODE FIELD DESCRIPTION

Source: Microchip Data Sheet



Field	Description
a	RAM access bit a = 0: RAM location in Access RAM (BSR register is ignored) a = 1: RAM bank is specified by BSR register
bbb	Bit address within an 8-bit file register (0 to 7).
BSR	Bank Select Register. Used to select the current RAM bank.
d	Destination select bit d = 0: store result in WREG d = 1: store result in file register f
dest	Destination either the WREG register or the specified register file location.
f	8-bit register file address (0x00 to 0xFF).
fs	12-bit register file address (0x000 to 0xFFF). This is the source address.
fd	12-bit register file address (0x000 to 0xFFF). This is the destination address.
k	Literal field, constant data or label (may be either an 8-bit, 12-bit or a 20-bit value).
label	Label name.
mm	The mode of the TBLPTR register for the table read and table write instructions. Only used with table read and table write instructions:
*	No change to register (such as TBLPTR with table reads and writes)
*+	Post-Increment register (such as TBLPTR with table reads and writes)
*-	Post-Decrement register (such as TBLPTR with table reads and writes)
+*	Pre-Increment register (such as TBLPTR with table reads and writes)
n	The relative address (2's complement number) for relative branch instructions, or the direct address for call/branch and return instructions.
PRODH	Product of Multiply High Byte.
PRODL	Product of Multiply Low Byte.
s	Fast Call/Return mode select bit s = 0: do not update into/from shadow registers s = 1: certain registers loaded into/from shadow registers (Fast mode)
u	Unused or unchanged.
WREG	Working register (accumulator).
x	Don't care ('0' or '1'). The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
TBLPTR	21-bit Table Pointer (points to a program memory location).
TABLAT	8-bit Table Latch.
TOS	Top-of-Stack.
PC	Program Counter.
PCL	Program Counter Low Byte.
PCH	Program Counter High Byte.
PCLATH	Program Counter High Byte Latch.
PCLATU	Program Counter Upper Byte Latch.
GIE	Global Interrupt Enable bit.
WDT	Watchdog Timer.
$\overline{\text{TO}}$	Time-out bit.
$\overline{\text{PD}}$	Power-down bit.
C, DC, Z, OV, N	ALU Status bits: Carry, Digit Carry, Zero, Overflow, Negative.
[ ]	Optional.
( )	Contents.
→	Assigned to.
< >	Register bit field.
∈	In the set of.
italics	User defined term (font is Courier).

APPENDIX C. REGISTER FILE SUMMARY

Source: Microchip Data Sheet

The following two tables contains the summary of the PICmicro Register file. The following Information will be useful in reading the register summary:

Legends:

x = unknown, u = unchanged, – = unimplemented, q = value depends on condition

Notes:

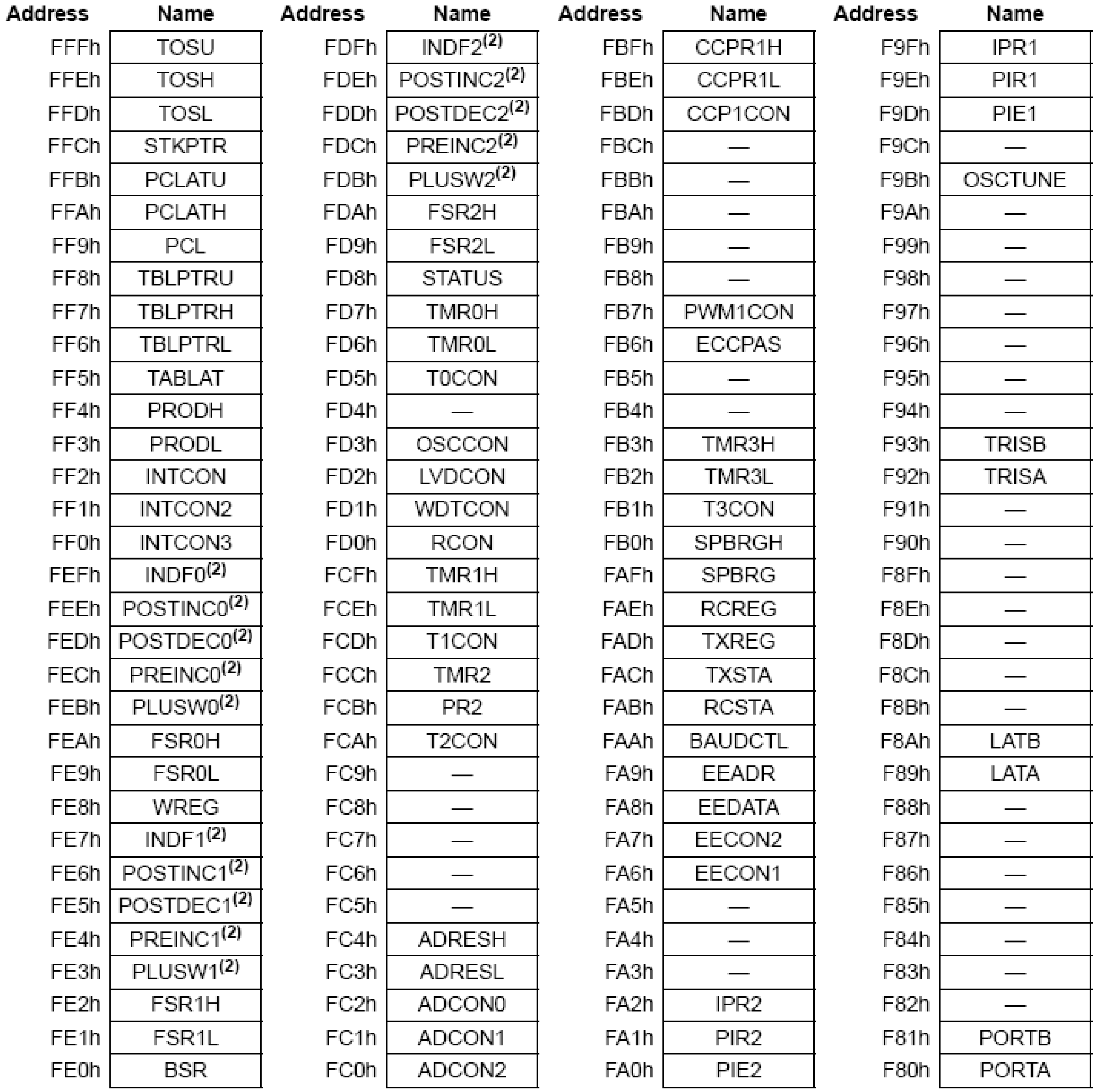
1: RA6 and associated bits are configured as port pins in RCIO, ECIO and INTIO2 (with port function on RA6) Oscillator mode only and read ‘0’ in all other oscillator modes.

2: RA7 and associated bits are configured as port pins in INTIO2 Oscillator mode only and read ‘0’ in all other modes.

3: Bit 21 of the PC is only available in Test mode and Serial Programming modes.

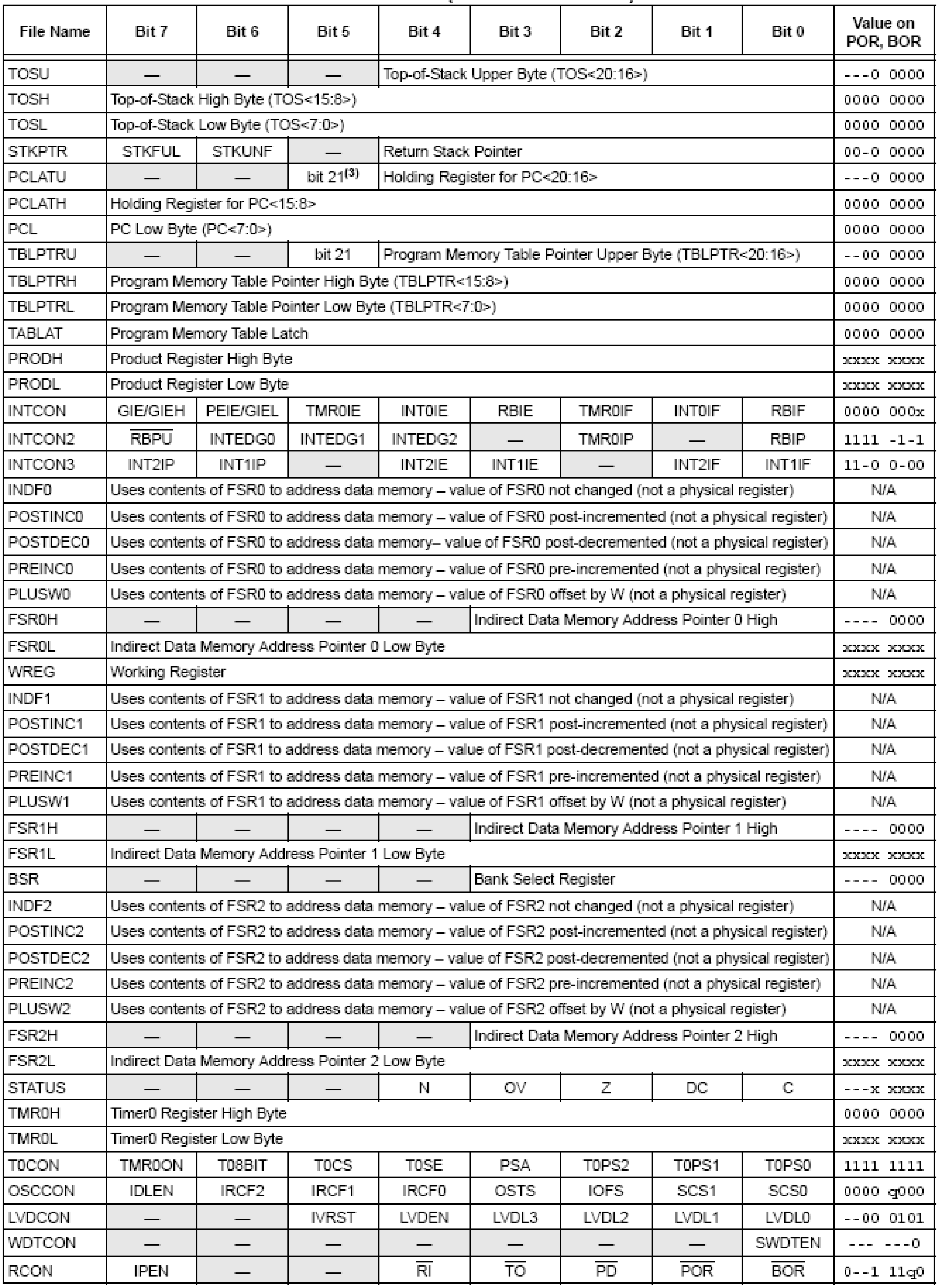
4: The RA5 port bit is only available when MCLRE fuse (CONFIG3H<7>) is programmed to ‘0’. Otherwise, RA5 reads ‘0’. This bit is read-only.

Special Function Registers (SFR) Map



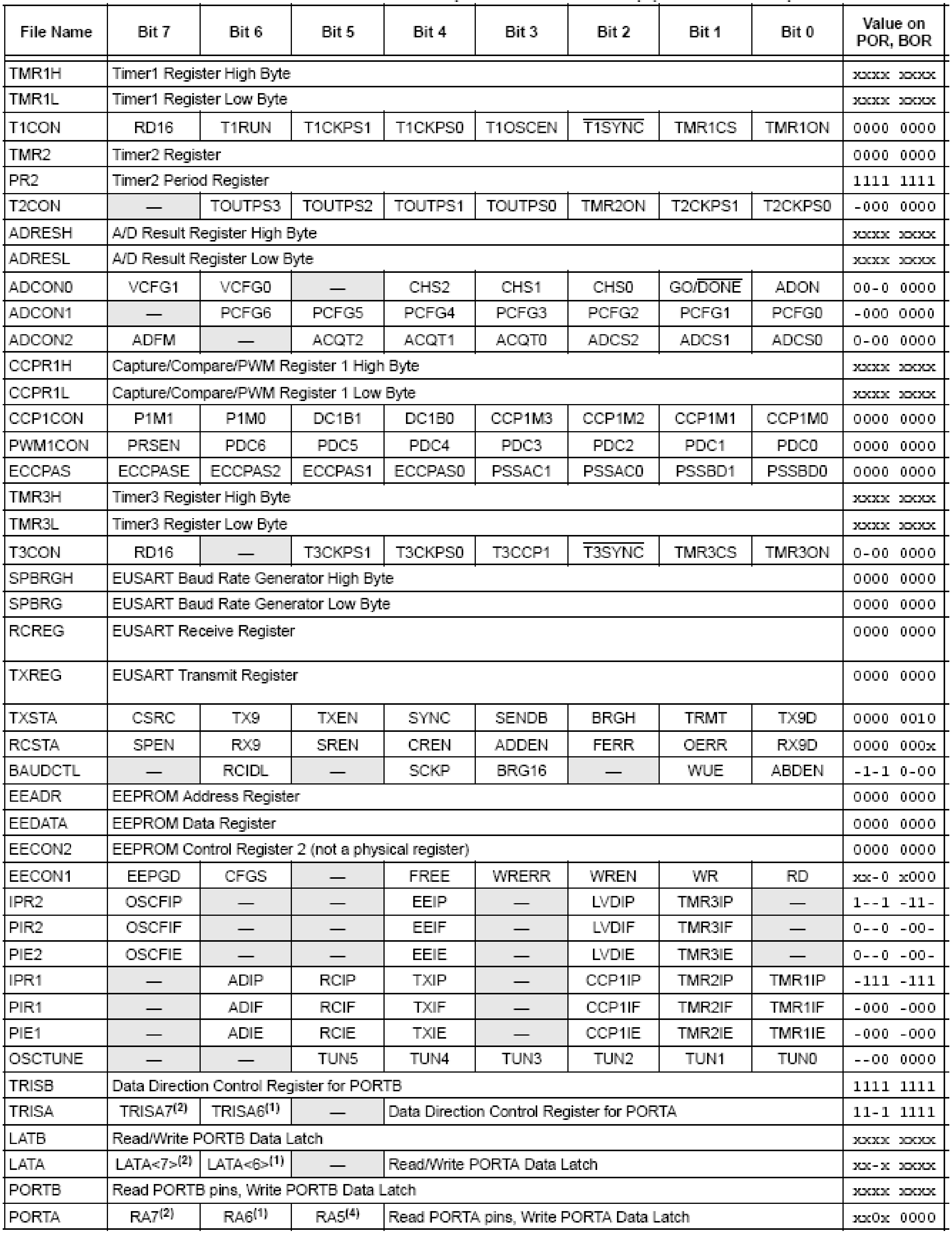
Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2[2]	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2[2]	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2[2]	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2[2]	FBCh	—	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2[2]	FBBh	—	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	—	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	PWM1CON	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCPAS	F96h	—
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	—
FF4h	PRODH	FD4h	—	FB4h	—	F94h	—
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—
FEFh	INDF0[2]	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC0[2]	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0[2]	FCDh	T1CON	FADh	TXREG	F8Dh	—
FECh	PREINC0[2]	FCCh	TMR2	FACh	TXSTA	F8Ch	—
FEBh	PLUSW0[2]	FCBh	PR2	FABh	RCSTA	F8Bh	—
FEAh	FSR0H	FCAh	T2CON	FAAh	BAUDCTL	F8Ah	LATB
FE9h	FSR0L	FC9h	—	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	—	FA8h	EEDATA	F88h	—
FE7h	INDF1[2]	FC7h	—	FA7h	EECON2	F87h	—
FE6h	POSTINC1[2]	FC6h	—	FA6h	EECON1	F86h	—
FE5h	POSTDEC1[2]	FC5h	—	FA5h	—	F85h	—
FE4h	PREINC1[2]	FC4h	ADRESH	FA4h	—	F84h	—
FE3h	PLUSW1[2]	FC3h	ADRESL	FA3h	—	F83h	—
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	—
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

General Register Map, 1/2



File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR
TOSU	—	—	—	Top-of-Stack Upper Byte (TOS<20:16>)					---0 0000
TOSH	Top-of-Stack High Byte (TOS<15:8>)								0000 0000
TOSL	Top-of-Stack Low Byte (TOS<7:0>)								0000 0000
STKPTR	STKFUL	STKUNF	—	Return Stack Pointer					00-0 0000
PCLATU	—	—	bit 21[3]	Holding Register for PC<20:16>					---0 0000
PCLATH	Holding Register for PC<15:8>								0000 0000
PCL	PC Low Byte (PC<7:0>)								0000 0000
TBLPTRU	—	—	bit 21	Program Memory Table Pointer Upper Byte (TBLPTR<20:16>)					--00 0000
TBLPTRH	Program Memory Table Pointer High Byte (TBLPTR<15:8>)								0000 0000
TBLPTRL	Program Memory Table Pointer Low Byte (TBLPTR<7:0>)								0000 0000
TABLAT	Program Memory Table Latch								0000 0000
PRODH	Product Register High Byte								xxxx xxxx
PRODL	Product Register Low Byte								xxxx xxxx
INTCON	GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF	0000 000x
INTCON2	$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG2	—	TMR0IP	—	RBIP	1111 -1-1
INTCON3	INT2IP	INT1IP	—	INT2IE	INT1IE	—	INT2IF	INT1IF	11-0 0-00
INDF0	Uses contents of FSR0 to address data memory – value of FSR0 not changed (not a physical register)								N/A
POSTINC0	Uses contents of FSR0 to address data memory – value of FSR0 post-incremented (not a physical register)								N/A
POSTDEC0	Uses contents of FSR0 to address data memory– value of FSR0 post-decremented (not a physical register)								N/A
PREINC0	Uses contents of FSR0 to address data memory – value of FSR0 pre-incremented (not a physical register)								N/A
PLUSW0	Uses contents of FSR0 to address data memory – value of FSR0 offset by W (not a physical register)								N/A
FSR0H	—	—	—	—	Indirect Data Memory Address Pointer 0 High				---- 0000
FSR0L	Indirect Data Memory Address Pointer 0 Low Byte								xxxx xxxx
WREG	Working Register								xxxx xxxx
INDF1	Uses contents of FSR1 to address data memory – value of FSR1 not changed (not a physical register)								N/A
POSTINC1	Uses contents of FSR1 to address data memory – value of FSR1 post-incremented (not a physical register)								N/A
POSTDEC1	Uses contents of FSR1 to address data memory – value of FSR1 post-decremented (not a physical register)								N/A
PREINC1	Uses contents of FSR1 to address data memory – value of FSR1 pre-incremented (not a physical register)								N/A
PLUSW1	Uses contents of FSR1 to address data memory – value of FSR1 offset by W (not a physical register)								N/A
FSR1H	—	—	—	—	Indirect Data Memory Address Pointer 1 High				---- 0000
FSR1L	Indirect Data Memory Address Pointer 1 Low Byte								xxxx xxxx
BSR	—	—	—	—	Bank Select Register				---- 0000
INDF2	Uses contents of FSR2 to address data memory – value of FSR2 not changed (not a physical register)								N/A
POSTINC2	Uses contents of FSR2 to address data memory – value of FSR2 post-incremented (not a physical register)								N/A
POSTDEC2	Uses contents of FSR2 to address data memory – value of FSR2 post-decremented (not a physical register)								N/A
PREINC2	Uses contents of FSR2 to address data memory – value of FSR2 pre-incremented (not a physical register)								N/A
PLUSW2	Uses contents of FSR2 to address data memory – value of FSR2 offset by W (not a physical register)								N/A
FSR2H	—	—	—	—	Indirect Data Memory Address Pointer 2 High				---- 0000
FSR2L	Indirect Data Memory Address Pointer 2 Low Byte								xxxx xxxx
STATUS	—	—	—	N	OV	Z	DC	C	---x xxxx
TMR0H	Timer0 Register High Byte								0000 0000
TMR0L	Timer0 Register Low Byte								xxxx xxxx
T0CON	TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0	1111 1111
OSCCON	IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0	0000 q000
LVDCON	—	—	IVRST	LVDEN	LVDL3	LVDL2	LVDL1	LVDL0	--00 0101
WDTCON	—	—	—	—	—	—	—	SWDTEN	--- ---0
RCON	IPEN	—	—	$\overline{\text{RI}}$	$\overline{\text{TO}}$	$\overline{\text{PD}}$	$\overline{\text{POR}}$	$\overline{\text{BOR}}$	0--1 11q0

General Register Map, 2/2



File Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR
TMR1H	Timer1 Register High Byte								xxxx xxxx
TMR1L	Timer1 Register Low Byte								xxxx xxxx
T1CON	RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	0000 0000
TMR2	Timer2 Register								0000 0000
PR2	Timer2 Period Register								1111 1111
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000
ADRESH	A/D Result Register High Byte								xxxx xxxx
ADRESL	A/D Result Register Low Byte								xxxx xxxx
ADCON0	VCFG1	VCFG0	—	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON	00-0 0000
ADCON1	—	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0	-000 0000
ADCON2	ADFM	—	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0	0-00 0000
CCPR1H	Capture/Compare/PWM Register 1 High Byte								xxxx xxxx
CCPR1L	Capture/Compare/PWM Register 1 Low Byte								xxxx xxxx
CCP1CON	P1M1	P1M0	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0	0000 0000
PWM1CON	PRSEN	PDC6	PDC5	PDC4	PDC3	PDC2	PDC1	PDC0	0000 0000
ECCPAS	ECCPASE	ECCPAS2	ECCPAS1	ECCPAS0	PSSAC1	PSSAC0	PSSBD1	PSSBD0	0000 0000
TMR3H	Timer3 Register High Byte								xxxx xxxx
TMR3L	Timer3 Register Low Byte								xxxx xxxx
T3CON	RD16	—	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON	0-00 0000
SPBRGH	EUSART Baud Rate Generator High Byte								0000 0000
SPBRG	EUSART Baud Rate Generator Low Byte								0000 0000
RCREG	EUSART Receive Register								0000 0000
TXREG	EUSART Transmit Register								0000 0000
TXSTA	CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D	0000 0010
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x
BAUDCTL	—	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN	-1-1 0-00
EEADR	EEPROM Address Register								0000 0000
EEDATA	EEPROM Data Register								0000 0000
EECON2	EEPROM Control Register 2 (not a physical register)								0000 0000
EECON1	EEPGD	CFGS	—	FREE	WRERR	WREN	WR	RD	xx-0 x000
IPR2	OSCFIP	—	—	EEIP	—	LVDIP	TMR3IP	—	1--1 -11-
PIR2	OSCFIF	—	—	EEIF	—	LVDIF	TMR3IF	—	0--0 -00-
PIE2	OSCFIE	—	—	EEIE	—	LVDIE	TMR3IE	—	0--0 -00-
IPR1	—	ADIP	RCIP	TXIP	—	CCP1IP	TMR2IP	TMR1IP	-111 -111
PIR1	—	ADIF	RCIF	TXIF	—	CCP1IF	TMR2IF	TMR1IF	-000 -000
PIE1	—	ADIE	RCIE	TXIE	—	CCP1IE	TMR2IE	TMR1IE	-000 -000
OSCTUNE	—	—	TUN5	TUN4	TUN3	TUN2	TUN1	TUN0	--00 0000
TRISB	Data Direction Control Register for PORTB								1111 1111
TRISA	TRISA7(2)	TRISA6(1)	—	Data Direction Control Register for PORTA					11-1 1111
LATB	Read/Write PORTB Data Latch								xxxx xxxx
LATA	LATA<7>(2)	LATA<6>(1)	—	Read/Write PORTA Data Latch					xx-x xxxx
PORTB	Read PORTB pins, Write PORTB Data Latch								xxxx xxxx
PORTA	RA7(2)	RA6(1)	RA5(4)	Read PORTA pins, Write PORTA Data Latch					xx0x 0000

APPENDIX D. SPECIAL FEATURES OF PICMICRO

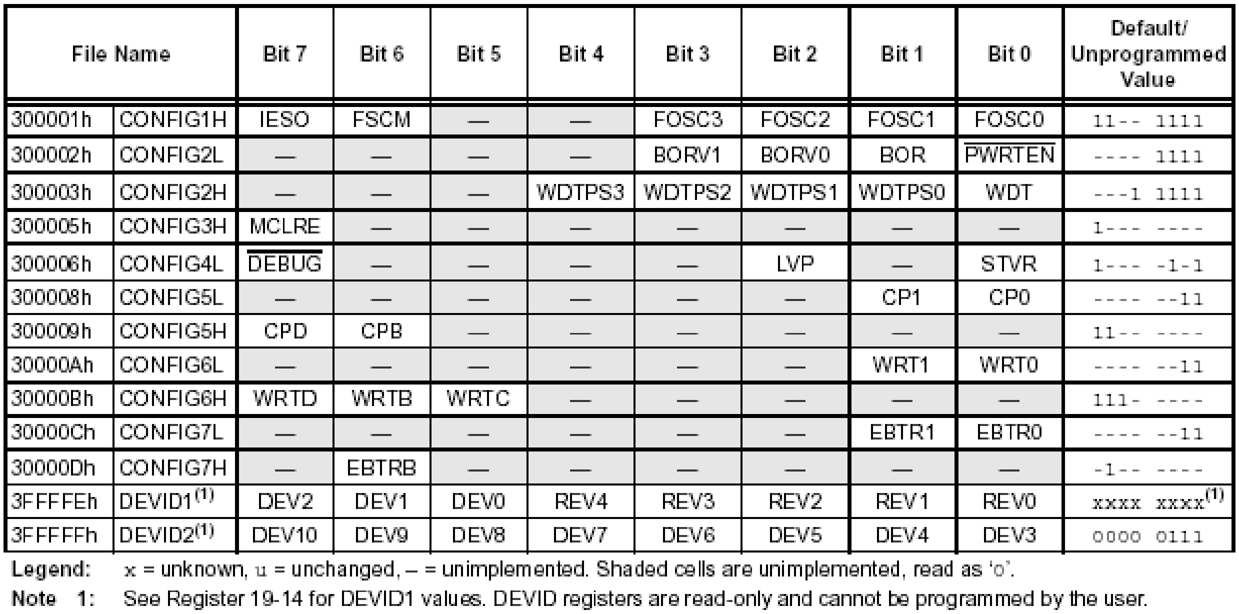
PICmicro includes features intended to maximize system reliability, minimize cost through elimination of external components and offer code protection. These are:

- Oscillator Selection
- Resets:
  - Power-on Reset (POR)
  - Power-up Timer (PWRT)
  - Oscillator Start-up Timer (OST)
  - Brown-out Reset (BOR)
- Interrupts
- Watchdog Timer (WDT)
- Fail-Safe Clock Monitor
- Two-Speed Start-up
- Code Protection
- ID Locations
- In-Circuit Serial Programming

Although most configurations can be done by modifying the SFR registers, the more central configuration is done by modifying the configuration bits.

The configuration bits can be programmed (read as ‘0’), or left un-programmed (read as ‘1’), to select various device configurations. These bits are mapped starting at program memory location 300000h which is beyond the program and user program memory space. In fact, it belongs to the configuration memory space (300000h-3FFFFFh). This space can only be accessed using the table read and table write instructions.

Programming the configuration registers is done in a manner similar to programming the Flash memory. The EECON1 register WR bit starts a self-timed write to the configuration register. In normal operation mode, a TBLWT instruction, with the TBLPTR pointing to the configuration register, sets up the address and the data for the configuration register write. Setting the WR bit starts a long write to the configuration register. The configuration registers are written a byte at a time. To write or erase a configuration cell, a TBLWT instruction can write a ‘1’ or a ‘0’ into the cell. For additional details on Flash programming, refer to PICmicro data sheet.



File Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default/ Unprogrammed Value
300001h	CONFIG1H	IESO	FSCM	—	—	FOSC3	FOSC2	FOSC1	FOSC0	`11-- 1111`
300002h	CONFIG2L	—	—	—	—	BORV1	BORV0	BOR	$\overline{\text{PWRTEN}}$	`---- 1111`
300003h	CONFIG2H	—	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDT	`---1 1111`
300005h	CONFIG3H	MCLRE	—	—	—	—	—	—	—	`1--- ----`
300006h	CONFIG4L	$\overline{\text{DEBUG}}$	—	—	—	—	LVP	—	STVR	`1--- -1-1`
300008h	CONFIG5L	—	—	—	—	—	—	CP1	CP0	`---- --11`
300009h	CONFIG5H	CPD	CPB	—	—	—	—	—	—	`11-- ----`
30000Ah	CONFIG6L	—	—	—	—	—	—	WRT1	WRT0	`---- --11`
30000Bh	CONFIG6H	WRTD	WRTB	WRTC	—	—	—	—	—	`111- ----`
30000Ch	CONFIG7L	—	—	—	—	—	—	EBTR1	EBTR0	`---- --11`
30000Dh	CONFIG7H	—	EBTRB	—	—	—	—	—	—	`-1-- ----`
3FFFFEh	DEVID1[(1)]	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	`xxxx xxxx`[(1)]
3FFFFFh	DEVID2[(1)]	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	`0000 0111`

Legend: x = unknown, u = unchanged, – = unimplemented. Shaded cells are unimplemented, read as '0'.

Note 1: See Register 19-14 for DEVID1 values. DEVID registers are read-only and cannot be programmed by the user.

APPENDIX E. ADDITIONAL RESOURCES

- Website www.EngrCS.com provide access to additional supporting hardware/software documentation, Microchip PIC 18F1220 Data Sheet and development environment.

- The latest development tools, documentation and tutorial on MPLAB software and other hardware development tools are available at www.Microchip.com.