MICROCONTROLLER THEORY AND APPLICATIONS WITH THE PIC18F

MICROCONTROLLER THEORY AND APPLICATIONS WITH THE PIC18F

Second Edition

M. RAFIQUZZAMAN Ph.D., PE
Professor
California State Polytechnic University
Pomona, California

and

President
Rafi Systems, Inc.
Diamond Bar, California

WILEY

This edition first published 2018
© 2018 John Wiley & Sons, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, except as permitted by law. Advice on how to obtain permission to reuse material from this title is available at http://www.wiley.com/go/permissions.

The right of M. Rafiquzzaman to be identified as the author of this work has been asserted in accordance with law.

Registered Offices
John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA

Editorial Office
111 River Street, Hoboken, NJ 07030, USA

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Limit of Liability/Disclaimer of Warranty
The publisher and the authors make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties; including without limitation any implied warranties of fitness for a particular purpose. This work is sold with the understanding that the publisher is not engaged in rendering professional services. The advice and strategies contained herein may not be suitable for every situation. In view of on-going research, equipment modifications, changes in governmental regulations, and the constant flow of information relating to the use of experimental reagents, equipment, and devices, the reader is urged to review and evaluate the information provided in the package insert or instructions for each chemical, piece of equipment, reagent, or device for, among other things, any changes in the instructions or indication of usage and for added warnings and precautions. The fact that an organization or website is referred to in this work as a citation and/or potential source of further information does not mean that the author or the publisher endorses the information the organization or website may provide or recommendations it may make. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this works was written and when it is read. No warranty may be created or extended by any promotional statements for this work. Neither the publisher nor the author shall be liable for any damages arising here from.

Library of Congress Cataloging-in-Publication Data

Names: Rafiquzzaman, Mohamed, author.
Title: Microcontroller theory and applications with the PIC18F / M. Rafiquzzaman, professor, California State Polytechnic University, Pomona, California and president, Rafi Systems, Inc., Diamond Bar, California.
Description: Hoboken, NJ, USA : Wiley, 2018. | Includes bibliographical references and index. |
Identifiers: LCCN 2017048388 (print) | LCCN 2017050854 (ebook) | ISBN 9781119448433 (pdf) | ISBN 9781119448440 (epub) | ISBN 9781119448419 (cloth)
Subjects: LCSH: Microcontrollers. | PIC microcontrollers. | C++ (Computer program language) | Assembly languages (Electronic computers)
Classification: LCC TK7895.E42 (ebook) | LCC TK7895.E42 R34 2018 (print) | DDC 005.13/3–dc23
LC record available at https://lccn.loc.gov/2017048388

Cover design by Wiley
Cover image: © William Potter/Shutterstock

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my wife, Kusum; and our grandsons (triplets), Ayaaz, Adyan, and Safir

CONTENTS

PREFACE **XIII**
CREDITS **XVII**
1. INTRODUCTION TO MICROCONTROLLERS **1**
1.1 Explanation of Terms 3
1.2 Microcontroller Data Types 7
1.2.1 Unsigned and Signed Binary Numbers 7
1.2.2 ASCII and EBCDIC Codes 10
1.2.3 Unpacked and Packed Binary-Coded-Decimal Numbers 10
1.3 Evolution of the Microcontroller 11
1.4 Embedded Controllers 14
2. MICROCONTROLLER BASICS **17**
2.1 Basic Blocks of a Microcomputer 17
2.1.1 System Bus 18
2.1.2 Clock Signals 19
2.2 Microcontroller architectures 20
2.3 Central Processing Unit (CPU) 21
2.3.1 Register Section 21
2.3.2 Control Unit 29
2.3.3 Arithmetic and Logic Unit (ALU) 30
2.3.4 Simplified Explanation of Control Unit design 30
2.4 Basic concept of pipelining 31
2.5 RISC vs. CISC 33
2.6 Functional Representation of a Typical Microcontroller---- The PIC18F4321 34
QUESTIONS AND PROBLEMS 36
3. MICROCONTROLLER MEMORY AND INPUT/OUTPUT (I/O) **37**
3.1 Introduction to Microcontroller Memory 37
3.1.1 Main memory 38
3.1.2 READ and WRITE Timing Diagrams 40
3.1.3 Main Memory Organization 42
3.2 Microcontroller Input/Output (I/O) 45
3.2.1 Overview of digital output circuits 47
3.2.2 Simple I/O Devices 49
3.2.3 Programmed I/O 50

3.2.4 Unconditional and Conditional Programmed I/O 52
3.2.5 Interrupt I/O 53
QUESTIONS AND PROBLEMS 56
4. PROGRAMMING LANGUAGES 59
4.1 Computer Programming Languages 59
4.2 Machine Language 60
4.3 Assembly Language 60
4.3.1 Types of Assemblers 61
4.3.2 Assembler Delimiters 62
4.3.3 Specifying Numbers by Typical Assemblers 63
4.3.4 Assembler Directives or Pseudoinstructions 63
4.3.5 Assembly Language Instruction Formats 65
4.3.6 Typical Instruction Set 67
4.3.7 Typical Addressing Modes 73
4.3.8 Subroutine Calls in Assembly Language 74
4.4 High-Level Language 74
4.5 Introduction to C Language 76
4.5.1 Data types 78
4.5.2 Bit manipulation operators 79
4.5.3 Control structures 81
4.5.4 The `if-else` construct 81
4.5.5 The `switch` construct 82
4.5.6 The `while` construct 83
4.5.7 The `for` construct 84
4.5.8 The `do-while` construct 85
4.5.9 Structures, and Unions 85
4.5.10 Functions in C 86
4.5.11 Arrays 88
4.5.12 Macros 88
4.6 Choosing a programming language 88
4.7 Flowcharts 89
QUESTIONS AND PROBLEMS 90
5. PIC18F ARCHITECTURE AND ADDRESSING MODES 93
5.1 Basic features of the PIC18F family 93
5.2 PIC18F Register Architecture 96
5.3 PIC18F Memory Organization 100
5.3.1 PIC18F Program Memory 101
5.3.2 PIC18F Data Memory 101
5.4 PIC18F Addressing Modes 104
5.4.1 Literal or Immediate Addressing Mode 104
5.4.2 Inherent or Implied Addressing Mode 104
5.4.3 Direct or Absolute Addressing Mode 104
5.4.4 Indirect Addressing Mode 105
5.4.5 Relative Addressing Mode 110
5.4.6 Bit Addressing Mode 112
QUESTIONS AND PROBLEMS 113
6. ASSEMBLY LANGUAGE PROGRAMMING WITH THE PIC18F: PART 1 115
6.1 Introduction to the PIC18F MPLAB assembler 115

6.2 PIC18F Instruction Format 120
6.3 PIC18F Instruction Set 121
6.3.1 Data Movement Instructions 124
6.3.2 Arithmetic Instructions 130
6.3.3 Logic Instructions 138
6.3.4 Rotate Instructions 141
6.3.5 Bit Manipulation Instructions 147
QUESTIONS AND PROBLEMS 151
7. ASSEMBLY LANGUAGE PROGRAMMING WITH THE PIC18F: PART 2 155
7.1 PIC18F Jump/Branch instructions 155
7.2 PIC18F Test, Compare, and Skip instructions 157
7.3 PIC18F Table Read/Write instructions 163
7.4 PIC18F Subroutine instructions 168
7.5 PIC18F System Control instructions 170
7.6 PIC18F Hardware vs. Software stack 171
7.7 Multiplication and Division algorithms 178
7.7.1 Signed Multiplication algorithm 178
7.7.2 Unsigned Division algorithm 180
7.7.3 Signed Division algorithm 182
7.8 Advanced Programming Examples 184
7.9 PIC18F Delay Routine 188
QUESTIONS AND PROBLEMS 191
8. PIC18F PROGRAMMED I/O USING ASSEMBLY & C 195
8.1 PIC18F Pins and Signals 195
8.1.1 Clock 196
8.1.2 PIC18F Reset 202
8.1.3 High Voltage and Low Voltage Programming (HVP and LVP) 205
8.1.4 “pragma config” and “config” directives 205
8.1.5 A simplified setup for the PIC18F4321 206
8.1.6 Downloading programs into the PIC18F4321 using the PICKit3 interface 206
8.2 PIC18F4321 Programmed I/O 208
8.2.1 I/O instructions in PIC18F assembly 211
8.2.2 Configuring PIC18F4321 I/O ports using PIC18F assembly 211
8.2.3 Configuring PIC18F4321 I/O ports using C 213
8.2.4 Interfacing LED’s (Light Emitting Diodes) and Seven-segment displays 215
8.2.5 Programmed I/O examples using PIC18F assembly 216
8.2.6 Programmed I/O examples using C Language 220
QUESTIONS AND PROBLEMS 228
9. PIC18F INTERRUPT I/O, LCD, AND KEYBOARD INTERFACING 231
9.1 Basics of Polled I/O vs. Interrupt I/O 231
9.2 PIC18F Interrupts 236
9.2.1 Interrupt Procedure 236
9.2.2 PIC18F Interrupt Types 238
9.2.3 Programming the PIC18F External Interrupts 238
9.2.4 Programming PIC18F external interrupts using assembly 243

9.2.5 Programming PIC18F external interrupts using C 243
9.2.6 Accessing PIC18F on-chip peripheral devices using Polled I/O vs. Interrupt I/O 252
9.3 PIC18F Interface to a typical LCD (Liquid Crystal Display) 252
9.4 Interfacing PIC18F4321 to a hexadecimal keyboard and a seven-segment display 259
9.4.1 Basics of Keyboard and Display Interface to a Microcontroller 259
9.4.2 PIC18F4321 Interface to a Hexadecimal Keyboard and a Seven-Segment Display 261
QUESTIONS AND PROBLEMS 270
10. PIC18F TIMERS AND ANALOG INTERFACE 273
10.1 PIC18F Timers 273
10.1.1 Timer0 275
10.1.2 Timer1 283
10.1.3 Timer2 289
10.1.4 Timer3 293
10.2 Analog Interface 301
10.2.1 PIC18F on-chip ADC (A/D Converter) 302
10.2.2 Interfacing an external D/A (Digital to Analog) Converter using C 315
QUESTIONS AND PROBLEMS 317
11. PIC18F CCP AND SERIAL I/O 321
11.1 PIC18F CCP (Capture/Compare/PWM (Pulse Width Modulation) Module 321
11.1.1 CCP Registers 322
11.1.2 CCP modules and associated timers 322
11.1.3 PIC18F4321 Capture mode 322
11.1.4 PIC18F4321 Compare mode 326
11.1.5 PIC18F4321 PWM (Pulse Width Modulation) mode 329
11.2 DC Motor Control 332
11.3 Serial Interface 336
11.3.1 Synchronous Serial Data Transmission 336
11.3.2 Asynchronous Serial Data Transmission 337
11.3.3 Basics of SPI and I^2C 337
11.4 PIC18F Serial I/O 338
11.4.1 PIC18F SPI mode 338
11.4.2 PIC18F I^2C (Inter-Integrated Circuit) mode 348
QUESTIONS AND PROBLEMS 359
APPENDIX A: ANSWERS TO SELECTED PROBLEMS 363
APPENDIX B: GLOSSARY 371
APPENDIX C: PIC18F INSTRUCTION SET (ALPHABETICAL ORDER) 383
APPENDIX D: PIC18F INSTRUCTION SET — DETAILS 389
APPENDIX E: PIC18F4321 SPECIAL FUNCTION REGISTERS 433
APPENDIX F: TUTORIAL FOR ASSEMBLING AND DEBUGGING A PIC18F ASSEMBLY LANGUAGE PROGRAM USING THE MPLAB 435

APPENDIX G: TUTORIAL FOR COMPILING AND DEBUGGING A C-PROGRAM USING THE MPLAB **465**

APPENDIX H: INTERFACING THE PIC18F4321 TO A PERSONAL COMPUTER OR A LAPTOP USING PICKIT™ 3 **493**

H.1 INITIAL HARDWARE SETUP FOR THE PIC18F4321 493

H.2 CONNECTING THE PERSONAL COMPUTER (PC) OR THE LAPTOP TO THE PIC18F4321 VIA PICkit3 494

H.3 PROGRAMMING THE PIC18F4321 FROM A PERSONAL COMPUTER OR A LAPTOP USING THE PICkit3 495

BIBLIOGRAPHY **499**

INDEX **501**

PREFACE

Microcontrollers play an important role in the design of digital systems. They are found in a wide range of applications including office automation systems (copiers and fax machines), consumer electronics (microwave ovens), digital instruments, and robotics.

The second edition of this book is written in a very simplified manner to present the fundamental concepts of assembly and C language programming and interfacing techniques associated with typical microcontrollers. Microchip Technology's PIC18F4321 is used for this purpose. The PIC18F family is inexpensive, and continues to be popular. Emphasis in this edition is given on chip-level design and implementation. This is very essential for the students to understand the hardware and software aspects of a specific microcontroller from chip level in a first course on microcontrollers. The PIC18F family is an excellent educational tool for this purpose.

The PIC18F uses Harvard architecture with a RISC-based CPU. Conventional CPUs complete fetch, decode and execute cycles of an instruction in sequence. However, the PIC18F uses pipelining, in which instruction fetch and execution cycles are overlapped. This speeds up the instruction execution time of the PIC18F. A brief coverage of CPU architectures, RISC vs. CISC, pipelining, assembly / C language programming and I/O techniques associated with typical microcontrollers are provided in the first part of the book. These topics are then related to a popular member of the PIC18F family such as the PIC18F4321.

As far as the programming is concerned, assembly and C language programming are covered in this book using the PIC18F. Coverage of assembly language programming will provide an exposure to the internal architecture of microcontrollers. Note that although microcontrollers in typical industrial applications include mostly C and some assembly, an introduction to assembly language programming for students is essential.

Several assembly and C language programs along with I/O examples are developed using Microchip's MPLAB, and PICkit3. The MPLAB software package includes a text editor, PIC18F assembler, C compiler, and a simulator. The PICkit3 is a programmer interface provided by Microchip. One can build an inexpensive PIC18F-based system on a breadboard using one of the PIC18F devices such as the PIC18F4321. The programmer can download the compiled or assembled programs using the PICkit3 from the Personal Computer or laptop, and then perform meaningful experiments. This is the most inexpensive way of implementing laboratory experiments using a typical microcontroller such as the PIC18F4321. This will enable the students with a thorough and basic background in microcontroller-based applications from chip level.

The book is self-contained and includes a number of basic topics. A background in basic digital logic and C language programming is assumed. Characteristics and principles

common to typical microcontrollers are emphasized and basic microcontroller interfacing techniques are demonstrated via examples using the simplest possible devices such as switches, LEDs, A/D and D/A converters, the hexadecimal keyboard, seven-segment displays and LCDs (Liquid Crystal Displays). Most of the examples are implemented successfully in the laboratory.

The text is divided into 11 chapters. In Chapter 1, we provide a review of terminology, number systems, and evolution of microcontrollers. Finally, a comparison of the basic features of some members of the PIC18F family and typical microcontroller applications are also included.

Chapters 2 through 11 form the nucleus of the book. Chapter 2 covers typical microcontroller architectures. The concepts of CPU architecture, program and data memory units, pipelining, and RISC vs. CISC are included.

Chapter 3 is focused on the memory organization and I/O (Input / Output) techniques associated with typical microcontrollers. The basic concepts associated with main memory array design, including memory maps, are also covered. Typical microcontroller input/output techniques including programmed I/O and interrupt I/O are included.

Chapter 4 contains programming concepts associated with typical microcontrollers. Topics include machine, assembly and C language programming, typical addressing modes, and instruction sets.

Chapter 5 includes PIC18F architecture and addressing modes. The PIC18F pipelining, register architecture, memory maps, and addressing modes are provided.

The concepts of assembly language programming covered in Chapter 4 are demonstrated in Chapters 6 and 7 by means of a typical 8-bit microcontroller. A specific device from the PIC18F family such as the PIC18F4321 is used to illustrate the concepts. Several PIC18F assembly language programming examples are included.

The I/O techniques covered in Chapter 3 are demonstrated in Chapters 8 and 9 using the PIC18F4321. Several I/O examples (programmed, polled, and interrupt I/O) using PIC18F assembly and C languages are also included. These chapters also demonstrate how the software and hardware work together by interfacing simple I/O devices such as switches, LEDs, seven-segment displays, LCDs and a hexadecimal keyboard.

The PIC18F timers and analog modules (on-chip hardware timers and ADC) are described in Chapter 10. Typical examples include the design and successful implementation of a PIC18F4321-based voltmeter using both assembly and C.

The basics of CCP (Compare/Capture/PWM) and Serial I/O relating them to the PIC18F are covered in Chapter 11. Also, a DC motor is successfully interfaced to the PIC18F4321 using PWM (Pulse Width Modulation) mode of the CCP module. Both SPI and I^2C modes of the PIC18F serial I/O are illustrated by means of successful implementation in the laboratory.

Courses on "Introduction to C" and "DC Circuits" are normally prerequisites for the "Digital Logic Design". A course on "Introduction to Microcontrollers" is typically a core course for electrical engineering curriculum at the undergraduate level . With more than 40 years of academic and industrial experiences, the author believes that electrical engineers should have a background in "Introduction to Assembly Language". This will provide students with exposure to internal architecture of a typical microcontroller. Note that assembly language programming is not normally a core course in electrical engineering curriculum. In addition, hands-on experience with I/O (Programmed, Interrupt, ADC, Timers, CCP) using C is essential. Students with experience in both assembly and C using

a typical microcontroller such as the PIC18F will be better prepared for industries after graduation. A book like this will provide such a background.

The book can easily be adopted as a text for a one- semester or one - quarter course in microcontroller taught at the undergraduate level in electrical/computer engineering and computer science departments. The students are expected to have a background in "Introduction to C" and "Introduction to Digital Logic Design". The book will also be useful for practicing microcontroller system designers. Practitioners of microcontroller-based applications will find more simplified explanations in this book, together with examples and comparison considerations, compared to that found in manufacturers' manuals.

The author is especially indebted to Sandra Grayson, editor, Wiley, UK for her inspiration to write this second edition. The author would also like to express his sincere appreciation to his students, Imed Abdellaoui, Loc Phan, Zhuohui Li, Carlos Landaverde, Luke Stankiewicz, Thomas Leung, and to others; to his colleague Professor R. Chandra of California State Polytechnic University, Pomona for making some constructive suggestions. The author is grateful to CJ Media of California for preparing the final version of the manuscript; to Mary Vang of Rafi Systems Inc. for drawing some of the figures in the book; to Lauren Le for initial editing of the manuscript. Finally, the author wishes to express his sincere appreciation to his editor, Brett Kurzman of Wiley, USA for his personal commitment, dedication, and excellent job in bringing this book to publication.

M. RAFIQUZZAMAN
Pomona, California

CREDITS

The material cited here is used by permission of the sources listed below.

Copyright of Microchip Technology, Inc. 2009, Used by Permission: PIC18F4321 Family Data Sheet, DS39689F. All mnemonics of Microchip PIC18F Microcontroller Family are courtesy of Microchip Technology, Inc.

The following are registered trademarks of Microchip in the U.S.A. and other countries: AnyRate, Atmel, AVR, AVR Freaks, BitCloud, CHIPKIT, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, Heldo, JukeBLOX, KEELOQ, Kleer, LANCheck, LINK MD, maXStylus, maXTouch, megaAVR, MediaLB, MOST, MPLAB, OptoLyzer, PIC, picoPower, PICSTART, Prochip Designer, QTouch, RightTouch, SAM-BA, SST, SuperFlash, tinyAVR, UNI/O and XMEGA.

The following are registered trademarks of Microchip in the U.S.A: ClockWorks, ETHERSYNCH, Hyper Speed Control, HyperLight Load, IntelliMOS, KleerNet, mTouch, Precision Edge, QUIET-WIRE and The Embedded Control Solutions Company.
The following are registered trademarks of Microchip in other countries: BeaconThings, GestIC, Silicon Storage Technology.

The following are trademarks of Microchip in the U.S.A. and other countries: Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, BeaconThings, BodyCom, CodeGuard, CryptoAuthentication, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Averaged Matching, DAM, ECAN, EtherGREEN, ICSP, In-Circuit Serial Programming, Inter-Chip Connectivity, JitterBlocker, Mindi, MiWi, motorBench, MPASM, MPF, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation (OCG), PICDEM, PICDEM.net, PICkit, PICtail, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, SAM-ICE, SMART-I.S., SQI, Serial Quad I/O, SuperSwitcher, SuperSwitcher II, Total Endurance, TSHARC, USBCheck, VariSense, ViewSpan, WiperLock, Wireless DNA and ZENA.

The following is a service mark of Microchip in the U.S.A.: SQTP

1
INTRODUCTION TO MICROCONTROLLERS

Digital systems are designed to store, process, and communicate information in digital form. They are found in a wide range of applications including process control, communication systems, digital instruments, and consumer products. A digital computer, more commonly known simply as a *computer*, is an example of a typical digital system.

A computer manipulates information in digital, or more precisely, binary form. A *binary number* has only two discrete values: zero or one. Each discrete value is represented by the OFF or ON status of an electronic switch called a *transistor*. All computers understand only binary numbers. Any decimal number (base 10, with ten digits from 0 to 9) can be represented by a binary number (base 2, with digits 0 and 1).

The basic blocks of a computer are the central processing unit (CPU), the memory, and the input/output (I/O). The CPU of a computer is basically the same as the brain of a human being; so, computer memory is conceptually similar to human memory. Asking a question to a human being is analogous to entering a program into a computer using an input device such as a keyboard, and a person answering a question is similar in concept to outputting the program result to a computer output device such as a printer. The main difference is that human beings can think independently, whereas computers can only answer questions for which they are programmed. Computer *hardware* includes such components such as memory, CPU, transistors, nuts, bolts, and so on. Programs can perform a specific task, such as addition, if the computer has an electronic circuit capable of adding two numbers. Programmers cannot change these electronic circuits but can perform tasks on them using instructions.

Computer *software* consists of a collection of programs. Note that programs contain instructions and data for performing a specific task. All programs, written using a programming language (e.g., C or assembly language) must be translated into binary prior to execution by a computer because the computer understands only binary numbers. Therefore, a translator is necessary to convert such a program into binary. This is achieved using a translator program called a *compiler* (for C) or *assembler* (for assembly). Programs in binary form containing 0's and 1's are then stored in the computer memory for execution.

In an assembly language program, an instruction may include immediate data (using immediate addressing mode-to be discussed later) or the address of data (using absolute or direct addressing mode-to be discussed later). During instruction execution, the computer obtains immediate data from the "program memory" while the computer obtains data for the 'instruction specified with the address of data' from the "data memory". Hence, the computer memory with a CPU using an architecture called the "Harvard architecture" (to be discussed later) can be classified into two different memory types. They are "Program memory" containing instructions and immediate data and "Data memory" containing only data.

Microcontroller Theory and Applications with the PIC18F, Second Edition. M. Rafiquzzaman.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

Due to advances in semiconductor technology, it is possible to fabricate a CPU on a single chip. The result is a *microprocessor*. Both metal-oxide semiconductor (MOS) and bipolar technologies are used in the fabrication process. The CPU can be placed on a single chip when MOS technology is used. However, several chips are required with bipolar technology. At present, HCMOS (high-speed complementary MOS) or BICMOS (combination of bipolar and HCMOS) technology is normally used to fabricate a microprocessor on a single chip. Along with the microprocessor chip, appropriate memory and I/O chips can be used to design a *microcomputer*. The pins on the microprocessor chip are connected to the proper pins on the memory and I/O chips using wires to build a microcomputer. These wires are called the *system bus* and carry address, data, and control signals. In the past, some manufacturers designed a complete microcomputer (CPU, memory, and I/O) on a single chip with limited capabilities. Single-chip microcomputers such as the Intel 8048 were used in a wide range of industrial and home applications.

Microcontrollers evolved from single-chip microcomputers. Microcontrollers are normally used for dedicated applications such as automotive systems, home appliances, and home entertainment systems. Typical microcontrollers include a CPU, memory, I/O, along with certain peripheral functions such as timers, and ADC (Analog-to-Digital Converter) all in a single chip. Microchip Technology's PIC (Peripheral Interface Controller) is an example of a typical microcontroller.

Note that an ADC (Analog-to-Digital Converter) converts a DC voltage into a binary number. For example, an 8-bit ADC chip will convert a DC voltage, between 0V and 5V, into an 8-bit binary number. This means that the lowest voltage , 0V at the ADC input will be converted to an 8-bit binary 00000000 at the ADC output; on the other hand, the highest voltage, 5V at the ADC input, will be converted to an 8-bit binary number 11111111 (255 in decimal). Other voltages between 0V and 5V will be converted accordingly. Since microcontrollers only understand binary numbers, these binary values can be inputted into the microcontrollers via programming to perform meaningful applications. The example depicted in Figure 1.1 illustrates this.

To put microcontrollers into perspective, it is important to explore a simple application. For example, consider the microcontroller-based dedicated controller shown in Figure 1.1. Suppose that it is necessary to maintain the temperature of a furnace to a desired level to maintain the quality of a product. Assume that the designer has decided to control this temperature by adjusting the fuel. This can be accomplished using a typical microcontroller such as the PIC18F along with the interfacing components as follows.

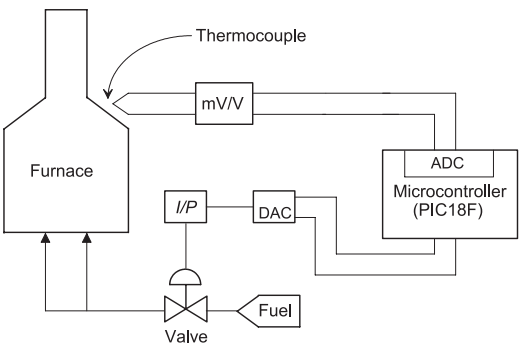


FIGURE 1.1 Furnace temperature control.

Temperature is an analog (continuous) signal. It can be measured by a temperature-sensing (measuring) device such as a thermocouple. The thermocouple provides the measurement in millivolts (mV) equivalent to the temperature.

Since microcontrollers only understand binary numbers (0's and 1's), each analog mV signal must be converted to a binary number using the microcontroller's on-chip analog-to-digital converter (ADC). Note that the PIC18F contains on-chip ADC. However, the PIC18F does not include on-chip Digital-to-Analog Converter (DAC). An external DAC chip can be interfaced to the PIC18F.

First, the millivolt signal is amplified by an mV/V amplifier to make the signal compatible for ADC input. A microcontroller such as the PIC18F can be programmed to solve an equation with the furnace temperature as an input. This equation compares the temperature measured with the temperature desired which can be entered into the microcontroller via programming. The output of this equation will provide the proper opening and closing of the fuel valve to maintain the appropriate temperature. Since this output is computed by the microcontroller, it is a binary number. This binary output must be converted into an analog current or voltage signal.

The DAC (Digital-to-Analog Converter) chip inputs this binary number and converts it into a current (*I*). This signal is then outputted to the current/pneumatic (*I/P*) transducer for opening or closing the fuel input valve by air pressure to adjust the fuel to the furnace. The desired furnace temperature can thus be achieved. Note that a transducer converts one form of energy (electrical current in this case) to another form (air pressure in this example).

Some of the typical microcontroller applications include the following:

- Automotive including cruise control, ignition system, and radiator fan
- Microwave oven
- Barcode readers
- Hotel card key writers
- Robotics

Next, we first define some basic terms associated with microcontrollers. Then, we will briefly describe the evolution of microcontrollers. Finally, a comparison of basic features of popular microcontrollers along with an overview of embedded controllers will be included.

1.1 Explanation of Terms

Before we go on, it is necessary to understand some basic terms.

- Address is a pattern of 0's and 1's that represents a specific location in memory or a particular I/O device. An 8-bit microcontroller with 16 address bits for program memory can produce 2^{16} unique 16-bit patterns from 0000000000000000 to 1111111111111111, representing 65,536 different address combinations (addresses 0 to 65,535 in decimal). This means that the maximum size of the program memory of this microcontroller is 64 k or 64 kB (64 Kilobytes since $2^{16} = 2^6 \times 2^{10} = 64 \times 1k = 64k$). All programs stored in this program memory can be executed by the microcontroller's CPU. This memory is also called the "Main memory" or "Directly addressable memory".
- Addressing mode is the manner in which the microcontroller determines the operand (data) and destination addresses during execution of an instruction.

- Arithmetic-logic unit (ALU) is a digital circuit that performs arithmetic and logic operations on two n-bit numbers. The value of n for microcontrollers can be 8-bit or 16-bit or 32-bit. Typical operations performed by an ALU are addition, subtraction, ANDing, ORing, and comparison of two n-bit numbers. The size of the ALU defines the size of the microcontroller. For example, an 8-bit microcontroller such as Microchip's PIC18F contains an 8-bit ALU.
- Big endian convention is used to store a 16-bit number such as 16-bit data in two bytes of memory locations as follows: the low memory address stores the high byte while the high memory address stores the low byte. The NXP/Freescale/Motorola HC11 8-bit microcontroller follows the big Endian format.
- Bit is an abbreviation for the term binary digit. A binary digit can have only two values, which are represented by the digits 0 and 1, whereas a decimal digit can have 10 values, represented by the digits 0 through 9. The bit values are easily implemented in electronic and magnetic media by two-state devices whose states portray either of the binary digits 0 or 1. Examples of such two-state devices are a transistor that is conducting or not conducting, a capacitor that is charged or discharged, and a magnetic material that is magnetized north to south or south to north.
- Bit size refers to the number of bits that can be processed simultaneously by the basic arithmetic unit of a microcontroller. A number of bits taken as a group in this manner is called a word. For example, an 8-bit microcontroller can process an 8-bit word. An 8-bit word is referred to as a byte and a 4-bit word is known as a nibble.
- Bus consists of a number of wires that connects different elements inside a microcontroller. The wires in a bus can be grouped in terms of their functions. A microcontroller normally has an address bus, a data bus, and a control bus. Address bits are sent to memory or to an external device on the address bus. Instructions from memory and data to/from memory or external devices normally travel on the data bus. Control signals such as the read/write are transmitted on the control bus. Buses such as the data bus are bidirectional; information that can be transmitted in either direction on the bus; but some buses such as the address bus are unidirectional-information that can travel only in one direction.
- Clock is analogous to human heart beats. The microcontroller requires synchronization among its components, and this is provided by a clock or timing circuits.
- Complex Instruction Set Computer (CISC) contains a large instruction set. NXP/ Freescale/Motorola HC11 is a CISC-based microcontroller.
- CPU (Central Processing Unit) contains several registers (fast memory elements), an ALU, and a control unit. Note that the control unit translates instructions and generates enable signals for appropriate hardware units inside the CPU to perform the task desired by an instruction.
- EEPROM or E^2PROM (Electrically Erasable Programmable ROM) is nonvolatile. EEPROMs can be programmed without removing the chip from the socket. EEPROMs are called Read Most Memories (RMMs), because they have much slower write times than read times. Therefore, these memories are usually suited for applications when mostly reading rather than writing is performed. An example of EEPROM is the 2864 (8k x 8).

- EPROM (Erasable Programmable ROM) is nonvolatile. EPROMs can be programmed and erased. The EPROM chip must be removed from the socket for programming. This memory is erased by exposing the chip to ultraviolet light via a lid or window on the chip. Typical erase times vary between 10 to 30 minute. The EPROM is programmed by inserting the chip into a socket of the EPROM programmer, and providing proper addresses and voltage pulses at the appropriate pins of the chip. An example of EPROM is the 2764 (8k x 8).
- Flash memory is designed using a combination of EPROM and EEPROM technologies. Flash memory is nonvolatile and was invented by Toshiba in the mid 1980s. Flash memory can be programmed electrically while embedded on the board. One can change multiple bytes at a time. An example of flash memory is the Intel 28F020 (256K x 8). Flash memory is typically used in cell phones, digital camera, and contemporary microcontrollers for storing program memory.
- Harvard architecture is a type of CPU architecture which uses separate program and data memory units along with separate buses for program and data. This means that these processors can execute programs and access data simultaneously. Processors designed with this architecture require four buses for program memory and data memory. These are one data bus for instructions, one address bus for addresses of instructions, one data bus for data and one address bus for addresses of data. The sizes of the address and data buses for instructions may be different from the address and data buses for data. Several microcontrollers including the PIC18F are designed using the Harvard architecture. This is because it is inexpensive to implement these buses inside the chip since both program and data memories are internal to the chip. Note that the PIC18F contains a 21-bit address bus and a 16-bit data bus for program memory, and a 12-bit address bus and an 8-bit data bus for data memory.
- Instruction set of a microcontroller is a list of commands that the microcontroller is designed to execute. Typical instructions are ADD, SUBTRACT, and STORE. Individual instructions are coded as unique bit patterns which are recognized and executed by the microcontroller. If a microcontroller has three bits allocated to the representation of instructions, the microcontroller will recognize a maximum of 2^3, or eight, different instructions. The microcontroller will then have a maximum of eight instructions in its instruction set.
- Little endian convention is used to store a 16-bit number such as 16-bit data in two bytes of memory locations as follows: the low memory address stores the low byte while the high memory address stores the high byte. The PIC18F microcontroller follows the little-endian format.
- Microcomputer typically consists of a microprocessor (CPU) chip, input and output chips, and memory chips in which programs (instructions and data) are stored.
- Microcontroller is implemented in a single chip containing a CPU, memory, and IOP (I/O and peripherals). Note that a typical IOP contains an I/O unit, timers, ADC (Analog-to-Digital) and other peripheral functions (to be discussed later).
- Microprocessor is the CPU of a microcomputer contained in a single chip and must be interfaced with memory and I/O chips in order to function.
- Pipelining is a technique that overlaps instruction fetch (instruction read) with execution. This allows a microcontroller's processing operation to be broken down

into several steps (dictated by the number of pipeline levels or stages) so that the individual step outputs can be handled by the microcontroller in parallel. Pipelining is often used to fetch the microcontroller's next instruction while executing the current instruction, which speeds up the overall operation of the microcontroller considerably. Microchip technology's PIC18F (8-bit microcontroller) uses a two-stage instruction pipeline in order to speed up instruction execution.

- Program contains instructions and data. Two conventions are used to store a 16-bit number such as 16-bit data in two bytes of memory locations. These are called little endian and big endian byte ordering. In little endian convention, the low memory address stores the low byte while the high memory address stores the high byte. For example, the 16-bit hexadecimal number, 2050 will be stored as two bytes in two 16-bit locations (Hex 5000 and Hex 5001) as follows: Address 5000 will contain 50 while address 5001 will store 20. In big endian convention, on the other hand, the low memory address stores the high byte while the high memory address stores the low byte. For example, the same 16-bit hexadecimal number, 2050 will be stored as two bytes in two 16-bit locations (Hex 5000 and Hex 5001) as follows: Address 5000 will contain 20 while address 5001 will store 50. NXP/Freescale/Motorola HC11 (8-bit microcontroller) follows big endian convention. Microchip PIC18F (8-bit microcontroller), on the other hand, follows the little-endian format.
- Random-access memory (RAM) is a read/write memory. A RAM normally provides volatile storage, which means that its contents are lost if the power is turned off. There are two types of RAM: static RAM (SRAM), and dynamic RAM (DRAM). Static RAM stores data in flip-flops. Therefore, this memory does not need to be refreshed. An example of SRAM is 6116 (2k $\times$ 8). Dynamic RAM, on the other hand, stores data as charge in capacitors. That is, it can hold data for a few milliseconds. Hence, dynamic RAMs are refreshed typically by using refresh circuitry. Dynamic RAMs (DRAMs) are used in applications requiring large memory. DRAMs have higher densities than static RAMs (SRAMs). Typical examples of DRAMs are the 4464 (6k $\times$ 4), 44256 (256k $\times$ 4), and 41000 (1M $\times$ 1). DRAMs are inexpensive compared to SRAMs, occupy less space, and dissipate less power than SRAMs.
- Read-only memory (ROM) is a storage device whose contents can only be read. Its contents cannot be altered once programmed. A typical ROM is fabricated on a chip and can store, for example, 2048 eight-bit words, which can be accessed individually by presenting to it one of 2048 addresses. This ROM is referred to as a 2K by 8-bit ROM. 10110111 is an example of an 8-bit word that might be stored in one location in this memory. A ROM is a nonvolatile storage device, which means that its contents are retained in the case that power is turned off. Because of this characteristic, ROMs are used to store permanent programs such as a keyboard monitor program. ROMs are programmed by the manufacturers.
- Reduced Instruction Set Computer (RISC) contains a simple instruction set. Because of a reduced instruction set, the RISC microcontrollers need fewer transistors which enable a smaller die size of the integrated circuitry (IC). Thus, the RISC microcontrollers consume less power compared to CISC, and hence, are suitable for portable devices. Microchip's PIC18F is a RISC-based microcontroller.
- Register can be considered as volatile storage for a number of bits. These bits may be entered into the register simultaneously (in parallel) or sequentially (serially) from

right to left or from left to right, 1 bit at a time. An 8-bit register storing the bits 11110000 is represented as follows:

1	1	1	1	0	0	0	0

- von Neumann (Princeton) architecture uses a single memory unit and the same bus for accessing both instructions and data. CPUs designed using this architecture are slower compared to Harvard architecture. Instructions and data cannot be accessed simultaneously because of the single bus. Typical microprocessors use this architecture. This is because memory units such as ROMs, EPROMs, and RAMs are external to the microprocessor. This will require almost half the number of wires on the mother board since address and data pins for only two buses rather than four buses (Harvard architecture) are required. This is the reason Harvard architecture would be very expensive if utilized for designing microprocessors. Note that microcontrollers using Harvard architecture internally will have to use von Neumann architecture externally. Texas Instruments' MSP430 uses the von Neumann architecture. MSP430 contains the same memory for both program memory and data memory with a 16-bit address bus and a 16-bit data bus.

1.2 Microcontroller Data Types

In this section we will discuss data types used by typical microcontrollers: unsigned and signed binary numbers, binary-coded decimal (BCD), ASCII (American Standard Code for Information Interchange), and EBCDIC (extended binary coded decimal interchange code).

1.2.1 Unsigned and Signed Binary Numbers

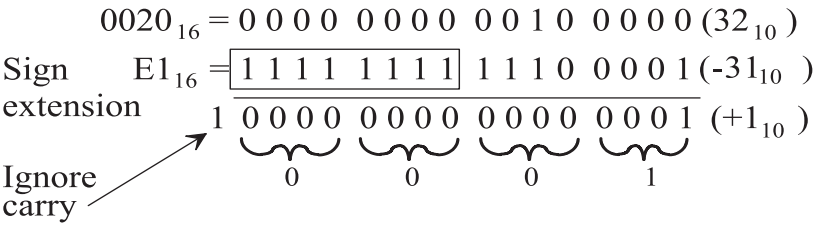
An *Unsigned binary number* is always positive. Typical examples are your age or a memory address, which are always positive numbers. An 8-bit unsigned binary integer represents all numbers from 00_{16} through FF_{16}(0_{16} through 255_{10}).

A *signed binary number*, on the other hand, includes both positive and negative numbers. It is represented in the microcontroller in two's-complement form. For example, the decimal number +15 is represented in 8-bit two's-complement form as 00001111 (binary) or 0F (hexadecimal). The decimal number -15 can be represented in 8-bit two's-complement form as 11110001 (binary) or F1 (hexadecimal). Also, the most significant bit (MSB) of a signed number represents the sign of the number. For example, bit 7 of an 8-bit number, bit 15 of a 16-bit number, and bit 31 of a 32-bit number represent the signs of the respective numbers. A "0" at the MSB represents a positive number; a "1" at the MSB represents a negative number. Note that the 8-bit binary number 11111111 is 255_{10} when represented as an unsigned number. On the other hand, 11111111_2 is -1_{10} when represented as a signed number.

One can convert an unsigned binary number from lower to higher length using zero extension. For example, an 8-bit unsigned number FF (hex) can be converted to a 16-bit unsigned number 00FF (hex) by extending 0's to the upper byte of 00FF (hex). Both FF (hex) and 00FF (hex) have the same decimal value of 255. This is called zero extension. Zero extension is useful for performing arithmetic operations between two unsigned binary numbers of different lengths.

A signed binary number, on the other hand, can be converted from lower to higher length using sign extension. For example, an 8-bit signed number FF (hex) can be converted to a 16-bit signed number FFFF (hex) by extending the sign bit ('1' in this case) to the upper byte of FFFF (hex). Both FF (hex) and FFFF (hex) have the same decimal value of -1. Sign extension is useful for performing arithmetic operations between two signed binary numbers of different lengths.

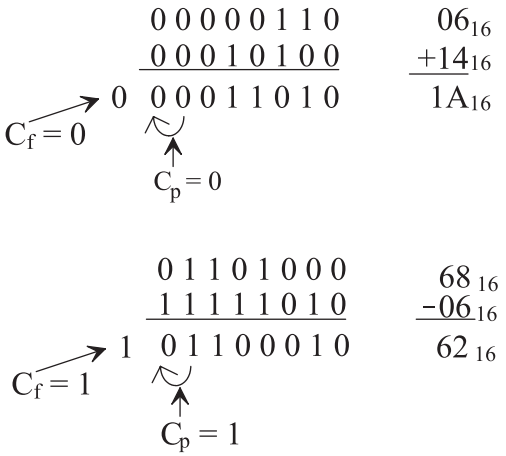
Sign extension is useful when one wants to perform an arithmetic operation on two signed numbers of different lengths. For example, the 16-bit signed number 0020 (hex) can be added with the 8-bit signed number E1 (hex) by sign-extending E1 as follows:



An error (indicated by overflow in a microcontroller) may occur while performing twos complement arithmetic. The microcontroller automatically sets an overflow bit to 1 if the result of an arithmetic operation is too big for the microcontroller's maximum word size; otherwise, it is cleared to 0. For signed arithmetic operations such as addition, the overflow $V = C_f \oplus C_p$ where C_f is the final carry and C_p is the previous carry. This can be illustrated by the following examples.

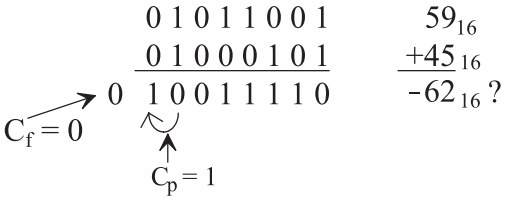
Consider the following examples for two 8-bit signed numbers. Let C_f be the final carry (carry out of the most significant bit or sign bit) and C_p be the previous carry (carry out of bit 6 or seventh bit). We will show by means of numerical examples that as long as C_f and C_p are the same, the result is always correct. If, however, C_f and C_p are different, the result is incorrect and sets the overflow bit to 1. Now, consider the following cases.

Case 1: C_f and C_p are the same.



Therefore, when C_f and C_p are either both 0 or both 1, a correct answer is obtained.

Case 2: C_f and C_p are different.



$C_f = 0$ and Cp = 1 give an incorrect answer because the result shows that the addition of two positive numbers is negative.

$C_f = 1$ and Cp = 0 provide an incorrect answer because the result indicates that the addition of two negative numbers is positive. Hence, the overflow bit (V) will be set to zero if the carries C_f and C_p are the same, that is, if both C_f and C_p are either 0 or 1. On the other hand, the overflow bit (V) will be set to 1 if carries C_f and C_p are different. The relationship among C_f , C_p , and V can be summarized in a truth table as follows:

Inputs		Output
C_f	C_p	V
0	0	0
0	1	1
1	0	1
1	1	0

From the truth table, overflow, $V = C_f \oplus C_p$

Note that the symbol $\oplus$ represents exclusive-OR logic operation. Exclusive-OR means that when two inputs are the same (both one or both zero), the output is zero. On the other hand, if two inputs are different, the output is one. The overflow can be considered as an output while C_f and C_p are the two inputs. The answer is incorrect when the overflow bit is set to 1; the answer is correct if the overflow bit is 0.

Also, typical 16- and 32-bit microprocessors such as NXP/Freescale/Motorola's 68000/68020 have separate unsigned and signed multiplication and division instructions as follows: MULU (multiply two unsigned numbers), MULS (multiply two signed numbers), DIVU (divide two unsigned numbers), and DIVS (divide two signed numbers). It is important for the programmer to understand clearly how to use these instructions.

For example, suppose that it is desired to compute $x^2/255$. If X is a signed 8-bit number, the programmer should use the MULS instruction to compute $X * X$ which is always unsigned (the square of a number is always positive), and then use DIVU to compute $X^2/255$ (16-bit by 8-bit unsigned divide) since 255_{10} is positive. But if the programmer uses DIVS, both $X * X$ and 255_{10} (FF_{16}) will be interpreted as signed numbers. If FF_{16} will be interpreted as -1_{10}, the result will be wrong. On the other hand, if X is an unsigned number the programmer needs to use MULU and DIVU to compute $X^2/255$.

The PIC18F microcontroller includes unsigned multiplication instructions. However, the PIC18F does not provide any signed multiplication and division (both signed and unsigned) instructions. Hence, as shown in chapter 7, these instructions can be achieved by writing subroutines using PIC18F instructions.

1.2.2 ASCII and EBCDIC Codes

If it is to be very useful, a microcontroller must be capable of handling nonnumeric information. In other words, a microcontroller must be able to recognize codes that represent numbers, letters, and special characters. These codes are classified as alphanumeric or character codes. A complete and adequate set of necessary characters includes the following:

- 26 lowercase letters
- 26 uppercase letters
- 10 numerical digits (0–9)
- Approximately 25 special characters, which include +, /, #, %, and others.

This totals to 87 characters. To represent 87 characters with some type of binary code would require at least 7 bits. With 7 bits there are $2^7 = 128$ possible binary numbers; 87 of these combinations of 0 and 1 bits serve as the code groups representing the 87 different characters.

The two most common alphanumerical codes are the American Standard Code for Information Interchange (ASCII) and the extended binary-coded-decimal interchange code (EBCDIC). ASCII is typically used with microprocessors; IBM uses EBCDIC code. Eight bits are used to represent characters, although 7 bits suffice, because the eighth bit is frequently used to test for errors and is referred to as a *parity bit*. It can be set to 1 or 0 so that the number of 1 bits in the byte is always odd or even.

Note that decimal digits 0 through 9 are represented by 30_{16} through 39_{16} in ASCII. On the other hand, these decimal digits are represented by $F0_{16}$ through $F9_{16}$ in EBCDIC.

A microcontroller program is usually written for code conversion when input/output devices of different codes are connected to the microcontroller. For example, suppose that it is desired to enter the number 5 into a computer via an ASCII keyboard and to print this data on an EBCDIC printer. The ASCII keyboard will generate 35_{16} when the number 5 is pushed. The ASCII code 35_{16} for the decimal digit 5 enters the microcontroller and resides in the memory. To print the digit 5 on the EBCDIC printer, a program must be written that will convert the ASCII code 35_{16} for 5 to its EBCDIC code, $F5_{16}$. The output of this program is $F5_{16}$. This would be inputted to the EBCDIC printer. Because the printer understands only EBCDIC codes, it inputs the EBCDIC code $F5_{16}$ and prints the digit 5. Note that EBCDIC code is obsolete; it is used here merely for illustrative purposes. Typical microprocessors such as the Intel Pentium include instructions to provide correct unpacked BCD after performing arithmetic operations in ASCII. The Pentium instruction AAA (ASCII adjust for addition) is such an instruction. The PIC18F does not provide such an instruction.

1.2.3 Unpacked and Packed Binary-Coded-Decimal Numbers

The 10 decimal digits 0 through 9 can be represented by their corresponding 4-bit binary numbers. The digits coded in this fashion are called *binary-coded-decimal digits* in 8421 code, or BCD digits. Two unpacked BCD bytes are usually packed into a byte to form *packed BCD*. For example, two unpacked BCD bytes 02_{16} and 05_{16} can be combined as a packed BCD byte 25_{16}.

Let us consider entering data decimal 24 via an ASCII keyboard into a microcontroller. Two keys (2 and 4) will be pushed on the ASCII keyboard. This will generate 32 and 34 (32 and 34 are ASCII codes in hexadecimal for 2 and 4, respectively)

inside the microcontroller. A program can be written to convert these ASCII codes into unpacked BCD 02_{16} and 04_{16}. This data can be converted to packed BCD 24 or to binary. Unpacked BCD 02_{16} and 04_{16} can be converted into packed BCD 24 (00100100_2) by logically shifting 02_{16} to obtain 20_{16}, then logically ORing with 04_{16}. On the other hand, to convert unpacked BCD 02_{16} and 04_{16} into binary, one needs to multiply 02_{16} by 10 and then add 04_{16} to obtain 00011000_2 (the binary equivalent of 24).

Note that BCD correction (adding 6) is necessary for the following:

i) If the binary sum is greater than or equal to decimal 16 (This will generate a carry of one).

ii) If the binary sum is 1010 through 1111.

For example, consider adding packed BCD numbers 97 and 39:

111←Intermediate Carries

97	1001	0111	BCD for 97
+39	0011	1001	BCD for 39
136	1101	0000	invalid sum
	+0110	+0110	add 6 for correction
0001	0011	0110	← correct answer 136
1	3	6	

Typical 32-bit microprocessors such as the NXP/Freescale/Motorola 68020 include PACK and UNPK instructions for converting an unpacked BCD number to its packed equivalent, and vice versa. The PIC18F microcontroller contains an instruction called DAW which provides the correct BCD result after binary addition of two packed BCD numbers.

1.3 Evolution of the Microcontroller

Intel Corporation is generally acknowledged as the company that introduced the first microprocessor successfully into the marketplace. Its first microprocessor, the 4004, was introduced in 1971 and evolved from a development effort while making a calculator chip set.

The 4004 microprocessor was the central component in the chip set, which was called the MCS-4. The other components in the set included a 4001 ROM, a 4002 RAM, and a 4003 shift register.

Shortly after Intel 4004 appeared in the commercial marketplace, three other general-purpose microprocessors were introduced: Rockwell International 4-bit PPS-4, Intel 8-bit 8008, and National Semiconductor 16-bit IMP-16. Other companies, such as General Electric, RCA, and Viatron, also made contributions to the development of the microprocessor prior to 1971.

The microprocessors introduced between 1971 and 1972 were the first-generation systems designed using PMOS technology. In 1973, second-generation microprocessors such as the Motorola 6800 and the Intel 8080 (8-bit microprocessors) were introduced. The second-generation microprocessors were designed using NMOS technology. This technology resulted in a significant increase in instruction execution speed over PMOS and higher chip densities. Since then, microprocessors have been fabricated using a variety of technologies and designs. NMOS microprocessors such as the Intel 8085, Zilog Z80, and Motorola 6800/6809 were introduced based on second-generation microprocessors. A third

generation HMOS microprocessor, introduced in 1978, is typically represented by Intel 8086 and Motorola 68000; both of them are 16-bit microprocessors.

During the 1980's, fourth-generation HCMOS and BICMOS (a combination of bipolar and HCMOS) 32-bit microprocessors evolved. Intel introduced the first commercial 32-bit microprocessor and the problematic Intel 432, which was eventually discontinued. Since 1985, more 32-bit microprocessors have been introduced. These include Motorola's 68020, 68030, 68040, 68060, PowerPC, Intel's 80386, 80486, the Intel Pentium family, Core Duo, and Core2 Duo microprocessors.

The performance offered by the 32-bit microprocessor is more comparable to that of superminicomputers such as Digital Equipment Corporation's VAX11/750 and VAX11/780. Intel and Motorola also introduced RISC microprocessors which had simplified instruction sets: the Intel 80960 and Motorola 88100/PowerPC. Note that the purpose of RISC microprocessors is to maximize speed by reducing clock cycles per instruction. Almost all computations can be obtained from a simple instruction set. Note that, in order to enhance performance significantly, Intel Pentium Pro and other succeeding members of the Pentium family and Motorola 68060 are designed using a combination of RISC and CISC.

Single-chip microcomputers such as the Intel 8048 evolved during the 80's. Soon afterwards, based on the concept of single-chip microcomputers, Intel introduced the first 8-bit microcontroller: the Intel 8051 which uses Harvard architecture and is designed by using CISC. The 8051 contains a CPU, memory, I/O, ADC and DAC, timers, serial communication interface, all in a single chip. The microcontrollers became popular during the 80's.

8-bit microcontrollers gained popularity over the last several years. These microcontrollers are small enough for many embedded applications, but also powerful enough to allow a lot of complexity and flexibility in the design process of an embedded system. Several billion 8-bit microcontrollers were sold during the last decade. Several contemporary microcontroller manufacturers use RISC architecture, which provide a cost effective approach. In addition, typical 8-bit microcontrollers such as PIC18F implemented several on-chip enhanced peripheral functions including PWM (Pulse-width modulation) and flash memories. Note that NXP/Freescale/Motorola's popular 8-bit microcontroller HC11 does not have on-chip flash memory and PWM functions. PWM function is a very desirable feature for applications such as automotive and motor control. These applications may include driving servo motors. In HC11, timer section is used to generate PWM signals. However, Freescale implemented these features in the HC12 which is a 16-bit microcontroller. Note that the HC11 has been popular because of its rich instruction set.

Some of the popular manufacturers of microcontrollers include Microchip Technology, Texas Instruments, Freescale, and Atmel. Microchip Technology's first microcontroller, the PIC1650, was originally developed by General Instrument's Microelectronics Division in the late 70's. The early models of PIC contained an on-chip ROM and erasable EPROMs for program storage. State-of-the-art models include flash memory for program storage. Note that flash memory can be erased electrically only in blocks. Since the PIC18F uses Harvard architecture, program and data memories are separate. The PIC18F uses flash memory for program memory. SRAM, on the other hand, is used for data memory. Note that 'F' is included in the part number 'PIC18F' to indicate that the chip contains flash memory.

Some of the different PIC models include the PIC18F family (8-bit microcontroller chips), and the PIC24F family (16-bit microcontroller chips). Microchip has introduced

several different versions of the PIC18F microcontroller over the years. All members of the PIC18F family basically contain the same instruction set. However, certain features such as memory sizes, number of I/O ports, ADC channels, and PWM modules may vary from one version to another.

Microchip provides extensive software development support for the PIC18F family of microcontrollers. This software is known as MPLAB which includes assemblers, C compilers, and programmer/debugger hardware under the MPLAB and PICKit series.

Atmel developed the AVR family of microcontrollers in 1996. Note that AVR is not an acronym. AVR microcontrollers use Harvard architecture. Like the PIC18F, the AVR microcontrollers contain on-chip flash memory for program memory and SRAM is used as data memory. AVR microcontrollers include AVR 8- and 32-bit microcontrollers. Atmel also makes a microcontroller that uses Intel 8051 architecture. Furthermore, Atmel makes the ARM-based Cortex-M4 flash microcontrollers . Note that Microchip bought Atmel in 2016.

ARM (Advanced RISC Machine) is a UK-based company that makes ARM designs and licenses its designs to chip manufacturers. ARM offers 32- and 64-bit CPU core. The microcontroller manufacturers utilize the ARM CPU, builds all the peripherals around it, and manufactures the microcontroller. ARM is very popular and is particularly used in portable devices such as smart phones, tablets, and multimedia players due to low power consumption and reasonable performance. Freescale Semiconductor (formerly the semiconductor division of Motorola which was bought by NXP semiconductor in 2015) is a major manufacturer of 8-, 16-, and 32-bit microcontrollers.

Note that Philips Semiconductors changed its name to NXP in 2006. Some of the popular 8- and 16-bit NXP/Freescale microcontrollers include 68HC11 (8-bit) and 68HC16 (16-bit). However, NXP manufacturer introduced several high performance NXP low-power 32-bit microcontrollers in 2013. They are based on ARM Cortex core architecture. As an example, the NXP LPC54113 is a low-power microcontroller with an ARM Cortex-based CPU, and contains 256 kB flash and 192 kB SRAM with a typical clock speed of 48 MHz.

TABLE 1.1 Comparison of basic features of typical microcontrollers.

	PIC18F	MSP 430	HC11	AVR model ATtiny
Manufacturer	Microchip Technology	Texas Instruments	NXP/Freescale /Motorola	Microchip/Atmel
Introduced	2000; the first PIC in 1989.	Late 1990s	1985	2003
Size	8-bit	16-bit	8-bit	8-bit
Architecture	Harvard	von Neumann	von Neumann	Modified Harvard
Design approach	RISC	RISC	CISC	RISC
On-chip flash memory	Yes	Yes	No	Yes. First to offer on-chip flash.
On-chip PWM	Yes	Yes	No	Yes
CPU Clock	40-MHz (Maximum)	16-MHz (Maximum)	4-MHz (Maximum)	20-MHz (Maximum)
Total Instructions	75	27	153	136
Total addressing modes	6	7	6	7

Texas Instruments introduced the MSP 430 16-bit microcontroller during the late 90's. MSP430 is a RISC microcontroller based on von Neumann architecture. Texas Instruments announced its 32-bit microcontroller called the MSP 432 in 2015. MSP 432 is based on ARM Cortex core with ultra low-power consumption. Finally, Arduino (an Italian based company) offers licensing for microcontroller manufacturers including Microchip/ Atmel and ARM to provide single boards with their microcontrollers. Arduino boards are available commercially in preassembled form or as do-it-yourself kits. A typical Arduino board consists of an Atmel 8-bit microcontroller with varying amounts of flash memory, pins, and serial I/O.

In this book, a specific PIC18F chip such as the PIC18F4321 will be considered in detail. This is because Microchip's PIC18F is inexpensive and offers a simple instruction set with desirable on-chip features including ADC, PWM, and timers. In addition, Microchip provides user-friendly development support such as MPLAB and PICKit. This makes the PIC18F an excellent educational tool for a thorough coverage in a first course on microcontrollers. Table 1.1 provides a comparison of some of the basic features of some of the typical microcontrollers comparable to the PIC18F.

1.4 Embedded Controllers

Embedded microcontroller systems, also called embedded controllers, are designed to manage specific tasks. Once programmed, the embedded controllers can manage the functions of a wide variety of electronic products. In embedded applications, the microcontrollers are embedded (hidden) in the host system; their presence and operation are basically hidden from the host system.

Typical embedded control applications include office automation products such as copiers, laser products, fax machines, and consumer products including microwave ovens. Applications such as a printer typically utilize a microcontroller. The RISC microcontrollers are ideal for these types of applications. Note that the Personal Computer (PC) interfaced to the printer is the host.

The microcontroller hidden inside the printer is the "embedded controller". The purpose of the microcontroller, in this case, is to input data from the host and print it. Thus, an embedded controller performs only one task only (printing in this case). A PC is normally connected to several peripherals such as a printer, a keyboard, a mouse, and a hard disk controller. Each one of these peripherals contain a microcontroller. Each microcontroller is programmed to execute a specific task desired by the peripheral. For example, the embedded microcontroller for the keyboard will perform keyboard functions for the PC including detecting a key actuation, debouncing it, and then decoding the key.

For large applications, an embedded controller contains three components: a microcontroller, an application software, and a Real Time Operating System (RTOS). The RTOS is not needed for a small embedded system. For a large embedded system, the RTOS supervises the application program and sets the rules during its execution.

RISC microcontrollers such as the PIC18F are well suited for applications including robotics, controls, instrumentation, and consumer electronics. The key features of the RISC microcontrollers that make them ideal for these applications are their relatively low level of integration in the chip, and instruction pipeline architecture. These characteristics result in low power consumption, fast instruction execution, and fast recognition of interrupts.

Although simple and popular microcontrollers such as the PIC18F are considered ideal for many embedded applications, sometimes they might not be able

to perform certain tasks. For example, applications such as laser printers require a high performance microcontroller with on-chip floating-point hardware. NXP/Freescale Kinetis microcontroller family with ARM's Cortex-M7 with on-chip floating-point hardware is ideal for these types of applications. Note that the Personal Computer interfaced to the laser printer is the host. The PIC18F will not be suitable for such an application since it does not provide floating-point instructions.

2
MICROCONTROLLER BASICS

In this chapter we will describe the fundamental material needed to understand the basic characteristics of microcontrollers. It includes topics such as typical microcontroller architectures, timing signals, CPU organization, and status flags. An overview of pipelining and RISC vs. CISC is included. Finally, an introduction to the functional characteristics of the PIC18F will be included.

2.1 Basic Blocks of a Microcomputer

In order to understand the functions performed by typical modules contained in a microcontroller, it is necessary to cover the basic blocks of a microcomputer.

A microcomputer has three basic blocks: a microprocessor (CPU on a chip), a memory unit, and an input/output (I/O) unit. Figure 2.1 shows the basic blocks of a microcomputer. A system bus (comprised of several wires) connects these blocks. The CPU executes all the instructions and performs arithmetic and logic operations on data. The CPU of the microcomputer contains all the registers, control unit, and arithmetic-logic circuits.

A *memory unit* stores both instructions and data. Note that microprocessors normally use von Neumann architecture in which both instructions and data are stored in the same memory. The memory section typically contains ROM and RAM chips. The ROM can only be read and is nonvolatile; that is, it retains its contents when the power is turned off. A ROM is typically used to store instructions and data that do not change. For example, it might store a table of seven-segment codes for outputting data to a display external to the microcomputer for turning on a digit from 0 through 9.

One can read from and write into a RAM. The RAM is volatile; that is, it does not retain its contents when the power is turned off. A RAM is used to store programs and data that are temporary and might change during the course of executing a program. An *I/O unit*

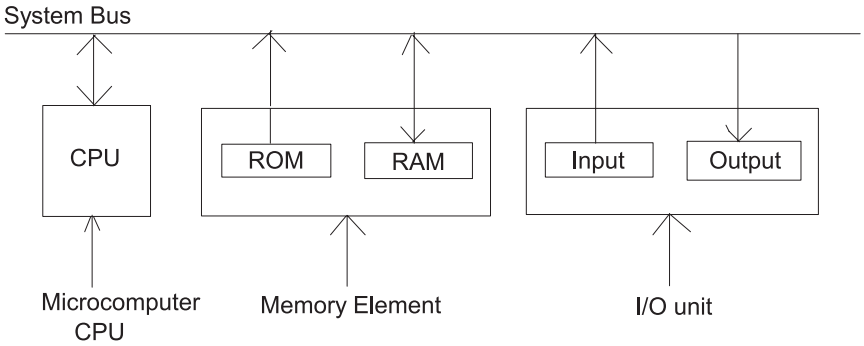


FIGURE 2.1 Basic Blocks of a Microcomputer

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

transfers data between the microcomputer and the external devices via I/O ports (registers). The transfer involves data, status, and control signals.

In a single-chip microcomputer, these three elements are on one chip, whereas in a single-chip microprocessor, separate chips are required for memory and I/O. Microcontrollers, which evolved from single-chip microcomputers, are typically used for dedicated applications such as automotive systems, home appliances, and home entertainment systems. Microcontrollers include a CPU, memory, and IOP (I/O and Peripherals) on a single chip. Note that a typical IOP contains an I/O unit of a microcomputer, timers, an A/D (Analog-to-Digital) converter, analog comparators, a serial I/O, and other peripheral functions (to be discussed later). Two popular microcontrollers are Microchip Technology's 8-bit PIC18F microcontroller which uses Harvard architecture (separate memories for instructions and data) while NXP/Freescale/Motorola's HC11 (8-bit) uses von Neumann architecture (same memory for instructions and data).

Since the microcomputer is an integral part of a microcontroller, it is necessary to investigate a typical microcomputer in detail. Once such a clear understanding is obtained, it will be easier to work with any specific microcontroller. Figure 2.2 illustrates a very simplified version of a typical microcomputer and shows the basic blocks of a microcomputer system. The various buses that connect these blocks are also shown. Although this figure looks very simple, it includes all the main elements of a typical microcomputer system.

2.1.1 System Bus

The microcomputer's system bus (internal to the microcontroller) contains three buses that carry all the address, data, and control information involved in program execution. These buses connect the CPU to each of the ROM, RAM, and I/O chips so that information transfer between the CPU and any of the other elements can take place. In a microcomputer, typical information transfers are carried out with respect to the memory or I/O. When a memory or an I/O chip receives data from the microprocessor, it is called a *WRITE operation*, and data is written into a selected memory location or an I/O port (register). When a memory or an I/O chip sends data to the microprocessor, it is called a *READ operation*, and data is read from a selected memory location or an I/O port.

In the *address bus*, information transfer takes place in only one direction, from the microprocessor to the memory or I/O elements. Therefore, this is called a *unidirectional*

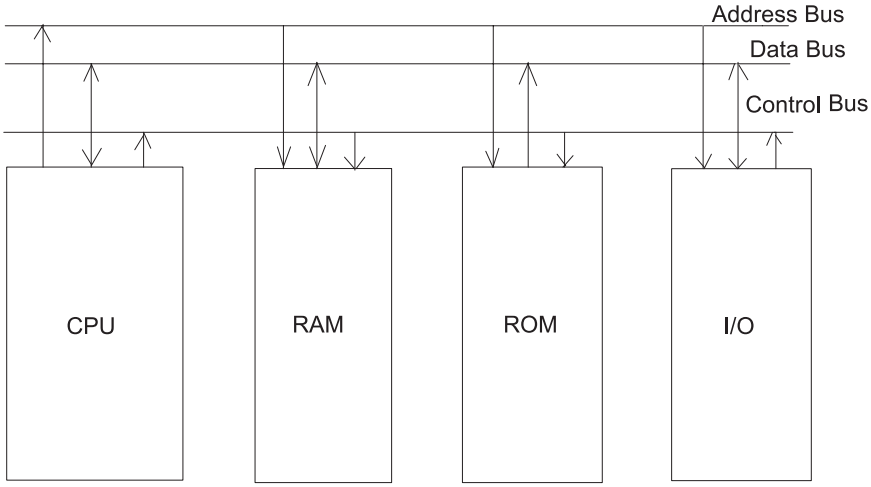


FIGURE 2.2 Simplified version of a typical microcomputer

bus. The size of the address bus determines the total number of memory addresses available in which programs can be executed by the microprocessor. The address bus is specified by the total number of address bits required by the CPU. This also determines the direct addressing capability or the size of the main memory of the microcontroller. The microcontroller's CPU can only execute programs located in the main memory. For example, a CPU with 16 address bits can generate $2^{16} = 64{,}536$ bytes [64 kilobytes (kB)] of different possible addresses (combinations of 0's and 1's) on the address bus. The CPU includes addresses from 0 to $65{,}535_{10}$ (0000_{16} through $FFFF_{16}$). A memory location can be represented by each of these addresses. For example, an 8-bit data item $2B_{16}$ can be stored at 16-bit address 0200_{16}.

When a CPU with a 16-bit address bus wants to transfer information between itself and a certain memory location, it generates the 16-bit address from an internal register on its 16 address pins, A_0–A_{15}, which then appear on the address bus. These 16 address bits are decoded to determine the desired memory location. The decoding process normally requires hardware (decoders) not shown in Figure 2.2.

In the *data bus*, data can flow in both directions, that is, to or from the CPU. Therefore, this is a bidirectional bus.

The *control bus* consists of a number of signals that are used to synchronize operation of the individual microcomputer elements. The CPU sends some of these control signals to the other elements to indicate the type of operation being performed. Each microcontroller has a unique set of control signals. However, some control signals are common to most microcontrollers. We will describe some of these control signals later in this section.

2.1.2 Clock Signals

The system clock signals are contained in the control bus. These signals generate the appropriate clock periods during which instruction executions are carried out by the CPU. Typical microcontrollers have an internal clock generator circuit to generate a clock signal.

Figure 2.3 shows a typical clock signal.

The number of cycles per second (Hertz, abbreviated as Hz) is referred to as the *clock frequency*. The CPU clock frequencies of typical microcontrollers vary from 1-MHz (1×10^6Hz) to 40-MHz (40×10^6Hz) . The clock defines the speed of the microcontroller. Note that one clock cycle = $1/f$ where f is the clock frequency. The execution times of microcontroller instructions are provided in terms of the number of clock cycles.

For example, suppose that execution time for the addition instruction by a microcontroller is one cycle. This means that a microcontroller with a 40-MHz clock will execute the ADD instruction in 25 nanoseconds [clock cycle = $1/(40 \times 10^6) = 25$ nanoseconds]. On the other hand, for a 4-MHz microcontroller, the addition instruction will be executed in 250 nanoseconds [clock cycle = $1/(4 \times 10^6) = 250$ nanoseconds]. This implies that the higher the clock frequency, the faster the microcontroller can execute the instructions.

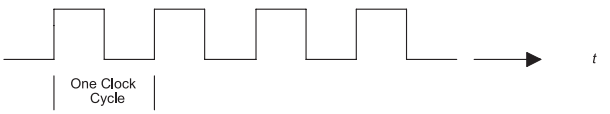


FIGURE 2.3 Typical clock signal.

2.2 Microcontroller architectures

The microcontroller requires memory to store programs and data. The various microcontrollers available today are basically the same in principle. The main variations are in the number of memory units, address and data buses they use. As mentioned in Chapter 1, two types of CPU architectures are used for designing microcontrollers**.** They are von Neumann (Princeton) architecture and Harvard architecture. The main difference between the two architectures is the way programs and data are stored in memory. Note that either data (also called "immediate data") or the address of data is included with an instruction. Immediate data is considered as part of the instruction. Hence, a program contains instructions and immediate data (data included with instructions). Data whose addresses are specified with the instructions are called "data". Hence, the names "program memory" containing instructions and immediate data and "data memory" containing data whose addresses are specified with the instructions are used. Program memory and data memory may be of different sizes. The PIC18F4321, for example, contains a program memory with 13-bit address (4k × 16) and a data memory with 12-bit address (4k × 8).

In von Neumann architecture, a single memory system with the same address and data buses is used for accessing both programs and data. This means that programs and data cannot be accessed simultaneously. This may slow down the overall speed. Texas Instrument's MSP 430 uses von Neumann architecture. Figure 2.4 shows a block diagram of the von Neumann architecture.

Harvard architecture is a type of CPU architecture which uses separate program and data memory units along with separate buses for instructions and data. This means that these processors can execute instructions and access data simultaneously. Processors designed with this architecture require four buses for program memory and data memory: one data bus for instructions, one address bus for addresses of instructions, one data bus for data, and one address bus for addresses of data. The sizes of the address and data buses for instructions may be different from the address and data buses for data. Several microcontrollers including the PIC18F are designed using the Harvard architecture. Figure 2.5 shows a block diagram of the Harvard architecture.

Most microcontrollers use the Harvard architecture because it is inexpensive to implement these buses inside the chip since both program and data memories are internal to the chip.

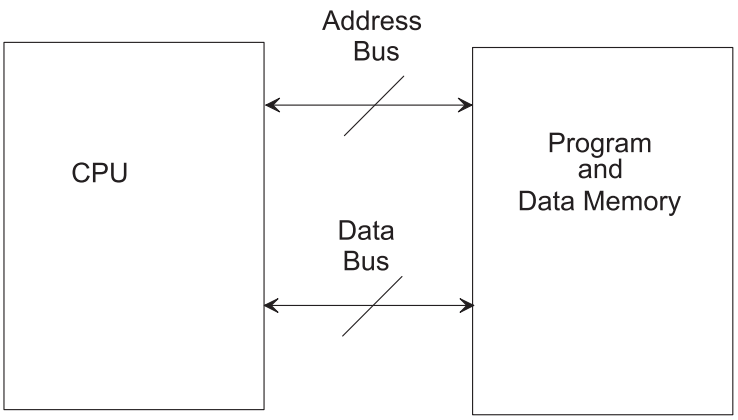


FIGURE 2.4 von Neumann Architecture

Although processors designed using the von Neumann architecture are slower compared to the Harvard architecture since instructions and data cannot be accessed simultaneously in the von Neumann architecture because of the single bus. Typical microprocessors such as the Pentium use von Neumann architecture. This is because memory units such as ROMs and RAMs are external to the microprocessor. This will require almost half the number of wires on the mother board since address and data pins for only two buses rather than four buses (Harvard architecture) are required. This is the reason Harvard architecture would be very expensive if utilized in designing microprocessors. Note that microcontrollers using Harvard architecture internally, will have to use von Neumann architecture externally.

2.3 Central Processing Unit (CPU)

As mentioned earlier, the CPU is the brain of the microcontroller. Therefore, the power of the microcontroller is determined by the capabilities of the CPU. The CPU's clock frequency determines the speed of the microcontroller. The number of data and address bits on the CPU make up the microcontroller's word size and maximum memory size. The microcontroller's I/O and interfacing capabilities are determined by the control bus of the CPU.

The logic inside the CPU can be divided into three main areas: the register section, the control unit, and the arithmetic-logic unit (ALU). A CPU chip with these three sections is shown in Figure 2.6.

2.3.1 Register Section

The number, size, and types of registers vary from one CPU to another. However, the various registers in all CPUs carry out similar operations. The register structures of

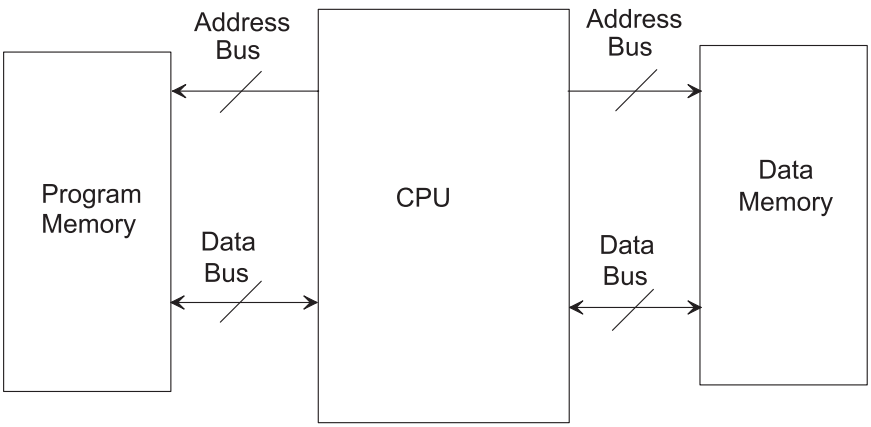


FIGURE 2.5 Harvard Architecture

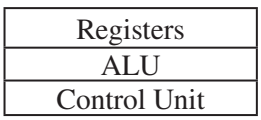


FIGURE 2.6 CPU with the main functional elements

CPUs play a major role in the design of a microcontroller. Also, the register structures for a specific CPU determine how convenient and easy it is to program the microcontroller. We first describe the most basic types of CPU registers, their functions, and how they are used. We then will consider other common types of registers.

Basic CPU Registers There are four basic CPU registers: instruction register, program counter, memory address register, and accumulator.

- Instruction Register (IR). The instruction register stores instructions. The contents of an instruction register are always decoded by the CPU as an instruction. After fetching an instruction code from memory, the CPU stores it in the instruction register. The instruction is decoded (translated) internally by the CPU, which then performs the operation required. The word size of the CPU normally determines the size of the instruction register.
- Program Counter (PC). The program counter contains the address of the instruction or operation code (op-code). The program counter normally contains the address of the next instruction to be executed. Note the following features of the program counter:
    1. Upon activating the CPU's RESET input, the address of the first instruction to be executed is normally loaded into the program counter.
    2. To execute an instruction, the CPU typically places the contents of the program counter on the address bus and reads ("fetches") the contents of this address (i.e., instruction) from memory. The program counter contents are incremented automatically by the CPU's internal logic. The CPU thus executes a program sequentially, unless the program contains an instruction such as a JUMP instruction, which changes the sequence.
    3. The size of the program counter is determined by the size of the address bus.
    4. Many instructions, such as JUMP and conditional JUMP, change the contents of the program counter from its normal sequential address value. The program counter is loaded with the address specified in these instructions.
- Memory Address Register (MAR). The memory address register contains the address of data. The CPU uses the address, which is stored in the memory address register, as a direct pointer to memory. The contents of the address is the actual data that is being transferred. MAR's are basically memory pointers.
- Accumulator (A). The accumulator is typically an 8-bit register. It stores the results after most ALU operations. These 8-bit CPUs have instructions to shift or rotate the accumulator one bit to the right or left through the carry flag. The accumulator is typically used for inputting a byte into the accumulator from an external device or for outputting a byte to an external device from the accumulator. The accumulator in the PIC18F is called the working register (WREG).

Depending on the register section, the CPU can be classified either as an accumulator or general-purpose register-based machine. In an accumulator-based microcontroller (PIC18F), the data is assumed to be held in a register called the accumulator. All arithmetic and logic operations are performed using this register as one of the data sources. The result of the operation is stored in the accumulator. Microchip Technology's PIC18F (8-bit microcontroller) is accumulator-based.

Texas Instrument's MSP 430, on the other hand, is a general-purpose register-based 16-bit microcontroller. The term *general-purpose* comes from the fact that these registers can hold data, memory addresses, or the results of arithmetic or logic operations.

The number, size, and types of registers vary from one microcontroller to another. Most registers are general-purpose, but some, such as the program counter (PC), are provided for dedicated functions. The PC normally contains the address of the next instruction to be executed.

As mentioned before, upon activating the CPU's RESET input pin, the PC is normally initialized with the address of the first instruction. For example, the PIC18F, upon hardware reset, reads the first instruction from address 0. Note that the PC in PIC18F is 21-bit wide. Hence, upon hardware reset, the PC in PIC18F will contain 21 zeros. To execute the instruction, the PIC18F will place the PC contents (0 in this case) on the address bus and reads (fetches) the first instruction from internal memory. The program counter contents are then incremented automatically by the ALU. The PC normally points to the next instruction.

Use of basic CPU registers To provide a clear understanding of how the basic registers in an accumulator-based CPU using the Harvard architecture are used, a binary addition program will be considered. The program logic will be explained by showing how each instruction changes the contents of some of the basic CPU registers (PC, IR, A). Assume that the address of program memory is 16-bit wide with 16-bit contents while the address of data memory is 8-bit wide with 8-bit contents. Suppose that the contents of the data memory address 0x20 are to be added to the contents of the accumulator. and then store the result in data memory address 0x20. Assume that [NNNN] represents the contents of the memory location NNNN. Now, assume that [0x20] = 0x05.

The steps involved in adding [0x20] with the contents of the accumulator can be summarized as follows:

1. Load 'A' with the first data (0x02) to be added.
2. Add the contents of the accumulator, 'A' to [0x20], and store the result in address, 0x20.

The following instructions for the PIC18F will be used to achieve the above addition:

0x0E02	Load 0x02 into 'A'
0x2620	Add [A] with [0x20] and store result address 0x20.

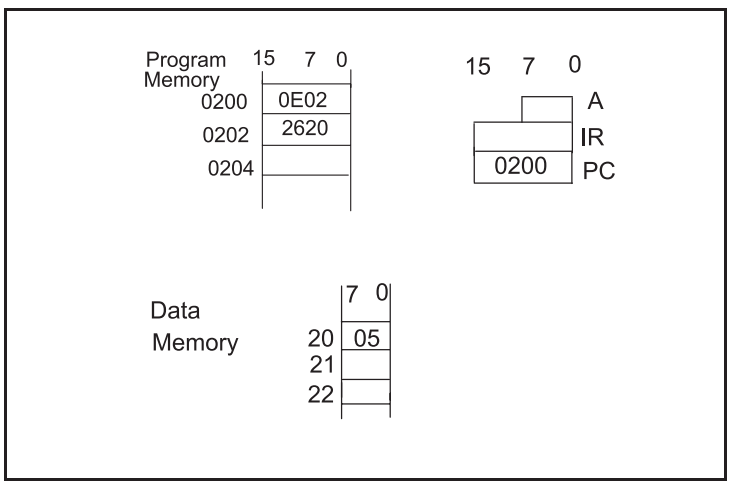


FIGURE 2.7 Addition program with initial register and memory contents

The complete program in hexadecimal starting at location 0x200 (arbitrarily chosen) is given in Figure 2.7. Note that Program memory address stores 16 bits. Hence, memory addresses are shown in increments of 2. Data memory, on the other hand, stores 8-bit data. Hence, memory addresses are shown in increments of 1. Assume that the CPU can be instructed so that the starting address of the program is 0x0200. This means that the program counter can be initialized to contain 0x0200, the address of the first instruction to be executed. Note that the contents of the other three registers (IR, MAR, A) are not known at this point. The CPU loads the contents of memory location addressed by the program counter into IR. Thus, the first instruction $0E02_{16}$ stored in address 0x200 is transferred into IR.

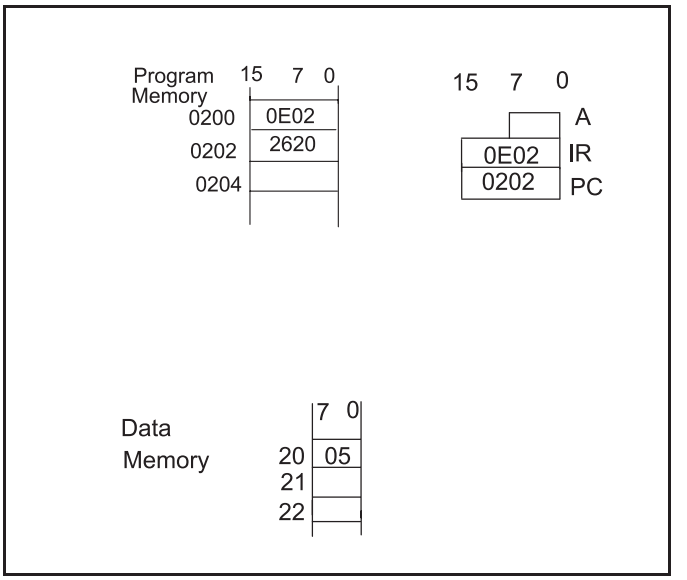


FIGURE 2.8 Addition program (modified during execution)

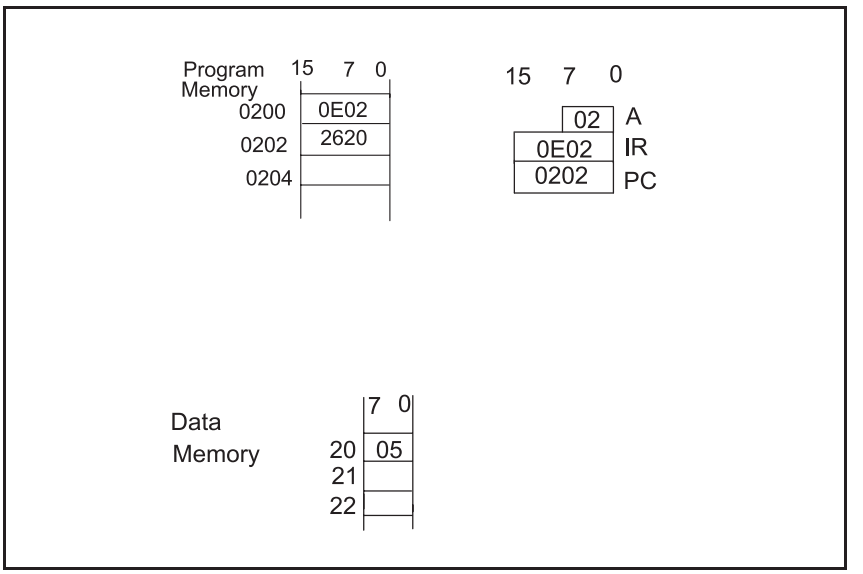


FIGURE 2.9 Addition program (modified during execution)

The program counter contents are then incremented by 2 by the ALU to hold 0x0202. The register contents along with the program are shown in Figure 2.8.

The binary code 0x0E02 in the IR is executed by the CPU. The CPU then takes appropriate actions. Note that the instruction 0x0E02 loads 0x02 into 'A' register. This is shown in Figure 2.9.

Next, the CPU loads the contents of the memory location addressed by the PC into the IR; thus, 0x2620 is loaded into the IR. The PC contents are then incremented by 2 to hold 0x0204. This is shown in Figure 2.10. In response to the instruction 0x2620, the contents of the data memory location addressed by the data memory address 0x20 are added to the contents in the accumulator A; thus, 0x05 is added to 0x02. The result 0x07 is stored in data memory address 0x20. Note that the previous contents (0x05) of data

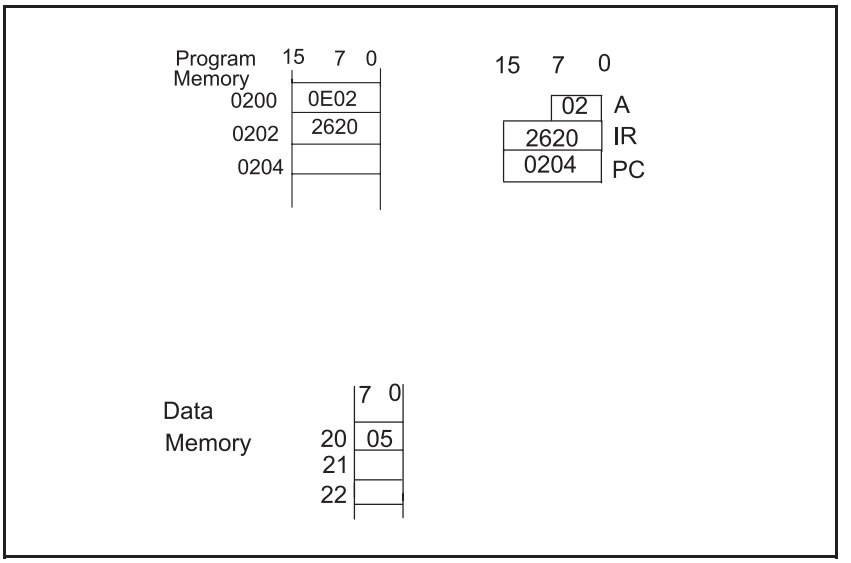


FIGURE 2.10 Addition program (modified during execution)

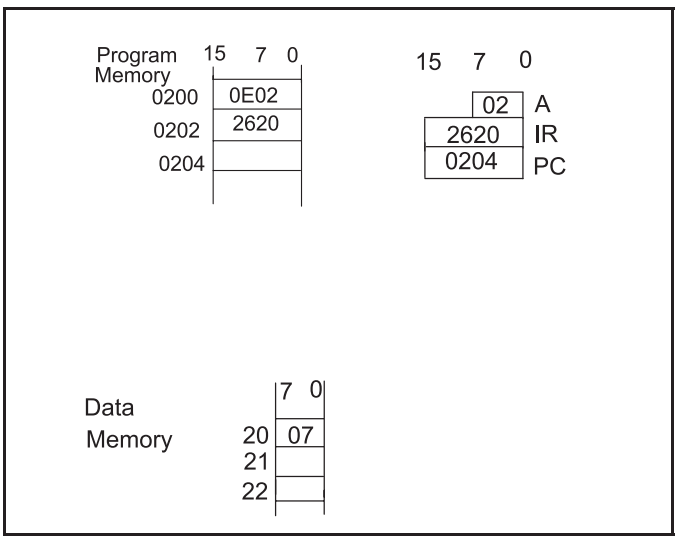


FIGURE 2.11 Addition program (modified during execution)

memory address 0x20 are lost. The contents of the PC are not incremented this time. This is because 0x05 is obtained from data memory. Figure 2.11 shows the details.

Other CPU Registers In the following, we will describe other CPU registers such as general-purpose registers, an index register, a status register, and a stack pointer register.

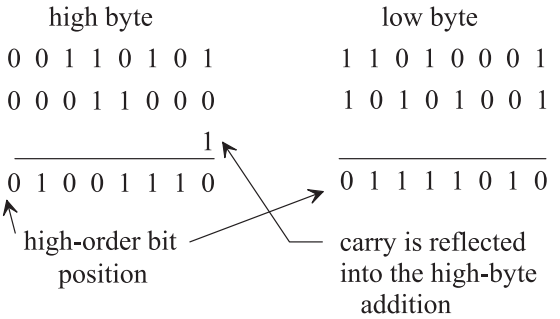
General-Purpose Registers Some microcontrollers such as the Texas Instrument's MSP430 have a number of general-purpose registers for storing temporary data or for carrying out data transfers between various registers. The use of general-purpose registers speeds up the execution of a program because the microprocessor does not have to read data from external memory via the data bus if data is stored in one of its general-purpose registers. Some of the typical functions performed by instructions associated with the general-purpose registers are given here. We will use [REG] to indicate the contents of the general-purpose register and [M] to indicate the contents of a memory location.

1. Move [REG] to or from memory: [M] ← [REG] or [REG] ← [M].
2. Move the contents of one register to another: [REG1] ← [REG2].
3. Increment or decrement [REG] by 1: [REG] ← [REG] + 1 or [REG] ← [REG] - 1.
4. Load 16-bit data into a register [REG] : [REG] ← 16-bit data.

Index Register Some microcontrollers such as the PIC18F provide indexed addressing mode using an index register to access an element in an array. An *index register* is typically used as a counter in address modification for an instruction or for general storage functions. The index register is particularly useful with instructions that access tables or arrays of data. In this operation the index register is used to modify the address portion of the instruction. Thus, the appropriate data in a table can be accessed. This is called *indexed addressing*. This addressing mode is normally available to the programmers of PIC18F. The effective address for an instruction using the indexed addressing mode is determined by adding the address portion of the instruction to the contents of the index register. Note that the accumulator is used as the index register in the PIC18F.

Status Register A *status register,* also known as a *processor status word register* or *condition code register*, contains individual bits with each bit having special significance. The bits in the status register are called *flags*. The status of a specific microcontroller operation is indicated by each flag, which is set or reset by the microcontroller's internal logic to indicate the status of certain operations such as arithmetic and logic operations. The status flags are also used in conditional JUMP instructions. We describe some of the common flags in the following.

A *carry flag* is used to reflect whether or not the result generated by an arithmetic operation is greater than the microcontroller's word size. As an example, the addition of two 8-bit numbers might produce a carry. The carry is generated out of the 8th bit position (bit 7), which results in setting the carry flag. However, the carry flag will be zero if no carry is generated from the addition. As mentioned before, in multibyte arithmetic, any carry out of the low-byte addition must be added to the high-byte addition to obtain the correct result. This can be illustrated by the following 16-bit addition example:



While performing BCD arithmetic with microcontrollers, the carry out of the low nibble (4 bits) has a special significance. Because a BCD digit is represented by 4 bits, any carry out of the low 4 bits must be propagated into the high 4 bits for BCD arithmetic. This carry flag is known as *"Digit Carry (DC)"* flag in the PIC18F and is set to 1 if the carry out of the low 4 bits is 1; otherwise, it is 0. A *zero flag* is used to show whether the result of an operation is zero. It is set to 1 if the result is zero, and it is reset to 0 if the result is nonzero. A *sign flag* (sometimes called a negative flag) is used to indicate whether the result of the last operation is positive or negative. If the most significant bit of the last operation is 1, this flag is set to 1 to indicate that the result is negative. This flag is reset to 0 if the most significant bit of the result is zero: that is, if the result is positive.

As mentioned earlier, an *overflow flag* arises from representation of the sign flag by the most significant bit of a word in signed binary operation. The overflow flag is set to 1 if the result of an arithmetic operation is too big for the microcontroller's maximum word size; otherwise, it is reset to 0. Let C_f be the final carry out of the most significant bit (sign bit) and C_p be the previous carry. It was shown in Chapter 1 that the overflow flag is the exclusive- OR of the carries C_p and C_f.

$$\text{overflow} = C_p \oplus C_f$$

For 8-bit signed arithmetic operations, the overflow flag will be set to one if the result is greater than +127 or less than or equal to - 128.

Stack Pointer Register A *stack* consists of a number of RAM locations set aside for reading data from or writing data into these locations and is typically used by subroutines (a *subroutine* is a program that performs operations frequently needed by the main or calling program). The address of the stack is contained in a register called a *stack pointer*. The size of the stack memory is normally the same as the Program counter. For example, since the program counter in PIC18F is 21-bit wide, the stack memory is also 21-bit. However, the size of the stack pointer in PIC18F is 5-bit which provides 32 (2^5) locations for the stack.

Two instructions, PUSH (Writing to the stack) and POP (reading from the stack), are usually available with a stack. The *PUSH operation* is defined as writing to the top or bottom of the stack, whereas the *POP operation* means reading from the top or bottom of the stack. Some microcontrollers access the stack from the top; others access via the bottom. When the stack is accessed from the bottom, the stack pointer is incremented after a PUSH and decremented after a POP operation. On the other hand, when the stack is accessed from the top, the stack pointer is decremented after a PUSH and incremented after a POP. Microcontrollers typically use internal registers for performing PUSH or POP operations. The incrementing or decrementing of a stack pointer depends on whether the operation is PUSH or POP and on whether the stack is accessed from the top or the bottom.

We will now illustrate stack operations in more detail. We use 16-bit registers

and 16-bit addresses in Figures 2.12 through 2.15. All data (hex) are chosen arbitrarily. In Figure 2.12, the stack pointer is incremented by 2 (16-bit register) after the PUSH to contain the value 20CA. Now, consider the POP operation of Figure 2.13. The stack pointer is decremented by 2 after the POP. The contents of address 20CA are assumed to be empty conceptually after the POP operation. Next, consider the PUSH operation of Figure 2.14. The stack is accessed from the top. The stack pointer is decremented by 2 after a PUSH. Finally, consider the POP operation of Figure 2.15. The Stack pointer is incremented by 2 after the POP. The contents of address 20C6 are assumed to be empty conceptually after a POP operation.

Note that the stack is a LIFO (last in first out) memory. As mentioned earlier, a stack is typically used during subroutine CALLs. The CPU automatically PUSHes the return address onto a stack after executing a subroutine CALL instruction in the main program. After executing a RETURN from a subroutine instruction (placed by the programmer as

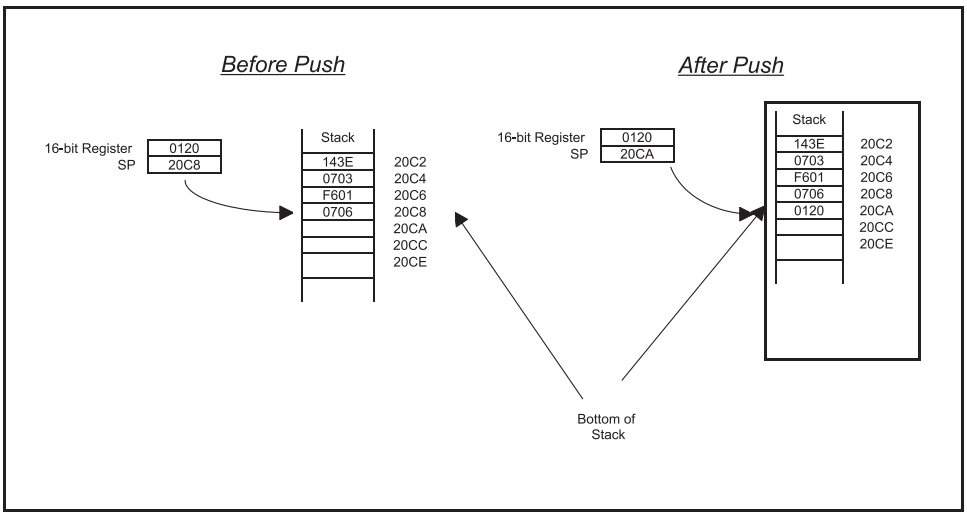


FIGURE 2.12 PUSH operation when accessing a stack from the bottom.

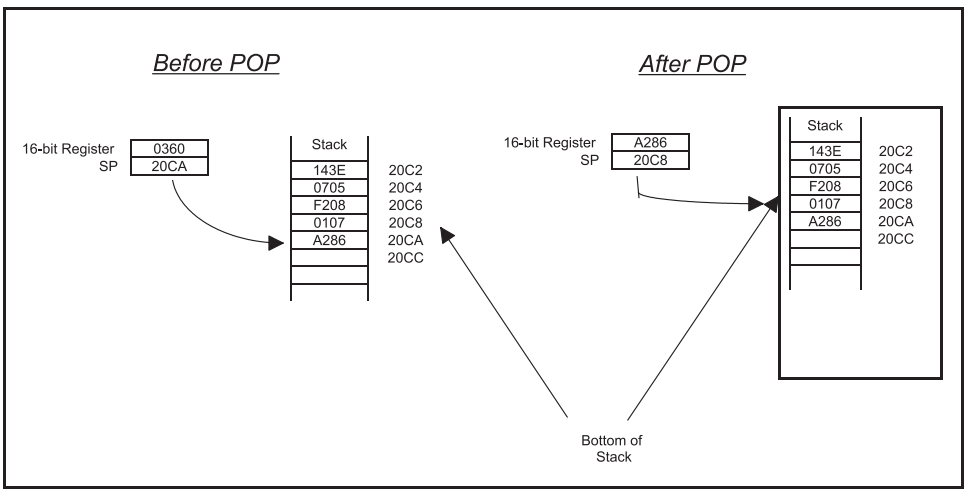


FIGURE 2.13 POP operation when accessing a stack from the bottom.

the last instruction of the subroutine), the CPU automatically POPs the return address from the stack (previously PUSHed) and then returns control to the main program.

Note that the PIC18F accesses stack from the bottom. This means that the stack pointer in the PIC18F holds the address of the bottom of the stack. Hence, in the PIC18F, the stack pointer is incremented after a PUSH, and decremented after a POP.

2.3.2 Control Unit

The main purpose of the control unit is to read and decode instructions from the program memory. To execute an instruction, the control unit steps through the appropriate blocks of the ALU based on the op-codes contained in the instruction register. The op-codes define the operations to be performed by the control unit to execute an instruction. The control unit interprets the contents of the instruction register and then responds to the instruction by generating a sequence of enable signals. These signals activate the appropriate ALU logic blocks to perform the required operation.

Control Bus Signals The control unit generates the *control signals*, which are output to the other microcontroller elements via the control bus. The control unit also takes appropriate actions in response to the control signals on the control bus provided by the other microcontroller elements. The control signals vary from one CPU to another. For

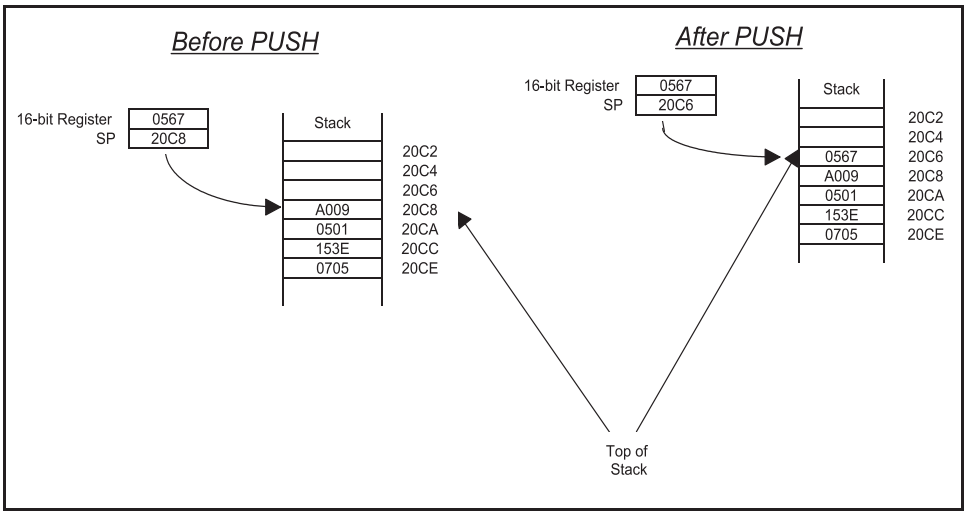


FIGURE 2.14 PUSH operation when accessing a stack from the top.

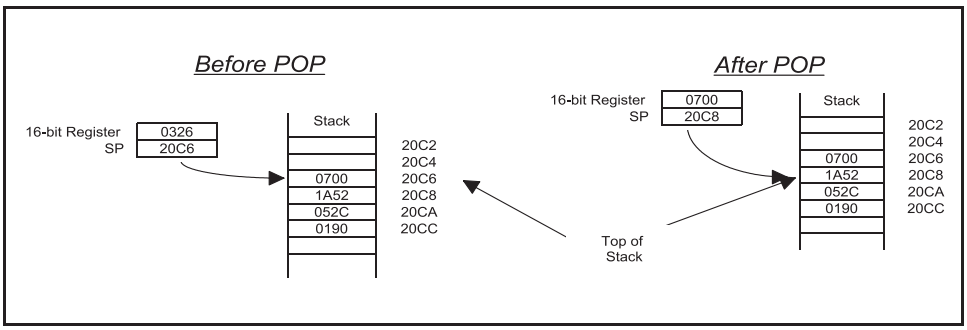


FIGURE 2.15 POP operation when accessing a stack from the top.

each specific CPU, these signals are described in detail in the manufacturer's manual. It is impossible to describe all the control signals for various manufacturers. However, we will cover some of the common ones in the following discussion.

$\overline{RESET}$. This input is common to all CPUs. When this input pin is driven HIGH or LOW (depending on the CPU), the program counter is loaded with a predefined address specified by the manufacturer. As mentioned before, the PIC18F upon hardware reset loads the 21-bit program counter with 0's. This means that the instruction stored at memory location 0 is executed first.

READ/ $\overline{WRITE}$ (R/$\overline{W}$). This output line is common to all CPUs. The status of this line tells the other microcontroller elements whether the CPU is performing a READ or a WRITE operation. A HIGH signal on this line indicates a READ operation, and a LOW signal indicates a WRITE operation. Some CPUs have separate READ and WRITE inputs.

INTERRUPT REQUEST. The external I/O devices can interrupt the microcontroller via this input signal on the CPU. When this signal is activated by the external devices, the CPU jumps to a special program called the *interrupt service routine*. This program is normally written by the user for performing tasks that the interrupting device wants the CPU to carry out. After completing this program, the CPU returns to the main program it was executing when the interrupt occurred. This topic will be covered in more detail in chapters 3, 8, 9, and 10.

2.3.3 Arithmetic and Logic Unit (ALU)

The ALU performs all the data manipulations, such as arithmetic and logic operations, inside a CPU. The size of the ALU conforms to the word length of the microcontroller. This means that an 8-bit microcontroller will have an 8-bit ALU. Some of the typical functions performed by the ALU are:

1. Binary addition and logic operations
2. Finding the one's complement of data
3. Shifting or rotating the contents of a general-purpose register 1 bit to the left or right through a carry

2.3.4 Simplified Explanation of Control Unit design

The main purpose of the control unit is to translate or decode instructions and generate appropriate enable signals to accomplish the desired operation. Based on the contents of the instruction register, the control unit sends the data items selected to the appropriate processing hardware at the right time. The control unit drives the associated processing hardware by generating a set of signals that are synchronized with a master clock.

The control unit performs two basic operations: instruction interpretation and instruction sequencing. In the interpretation phase, the control unit reads (fetches) an instruction from the memory addressed by the contents of the program counter into the instruction register. The control unit inputs the contents of the instruction register. It recognizes the instruction type, obtains the necessary operands, and routes them to the appropriate functional units of the execution unit (registers and ALU). The control unit then issues the necessary signals to the execution unit to perform the desired operation and routes the results to the destination specified. In the sequencing phase, the control unit generates the address of the next instruction to be executed and loads it into the program counter.

There are two methods for designing a control unit: hardwired control and microprogrammed control. In the hardwired approach, synchronous sequential circuit design procedures are used in designing the control unit. Note that a control unit is a clocked sequential circuit. The name *hardwired control* evolved from the fact that the final circuit is built by physically connecting components such as gates and flip-flops. In the microprogrammed approach, on the other hand, all control functions are stored in a ROM inside the control unit. This memory is called the *control memory*. The words in this memory, called *control words*, specify the control functions to be performed by the control unit. The control words are fetched from the control memory and the bits are routed to appropriate functional units to enable various gates. An instruction is thus executed. The PIC18F uses the hardwired approach for designing its control unit for the RISC-based CPU.

Design of control units using microprogramming (sometimes called *firmware* to distinguish it from hardwired control) is more expensive than using hardwired controls. To execute an instruction, the contents of the control memory in microprogrammed control must be read, which reduces the overall speed of the control unit.The most important advantage of microprogramming is its flexibility; alterations can be made simply by changing the microprogram in the control memory. A small change in the hardwired approach may lead to redesigning the entire system.

Microprogramming is typically used by a CPU designer to program the logic performed by the control unit. On the other hand, assembly language programming is a popular programming language used for programming a microcontroller to perform a desired function. A microprogram is stored in the control unit. An assembly language program is stored in the program memory. The assembly language program is called a *macroprogram*. A macroinstruction (or simply, an instruction) initiates execution of a complete microprogram.

2.4 Basic concept of pipelining

To execute a program, a conventional CPU repeats the following three steps for completing each instruction:

1. *Fetch.* The CPU fetches (instruction read) the instruction from the program memory (external to the CPU) into the instruction register.
2. *Decode.* The CPU decodes or translates the instruction using the control unit. The control unit inputs the contents of the instruction register, and then decodes (translates) the instruction to determine the instruction type.
3. *Execute.* The CPU executes the instruction using the control unit. To accomplish the task, the control unit generates a number of enable signals required by the instruction.

For example, suppose that it is desired to add the contents of two registers, X and Y, and store the result in register Z. To accomplish this, the conventional CPU performs the following steps:

1. The CPU fetches the instruction into the instruction register.
2. The control unit (CU) decodes the contents of the instruction register.
3. The CU executes the instruction by generating enable signals for the register and ALU sections to perform the following:

a. The CU transfers the contents of registers X and Y from the Register section into the ALU.

b. The CU commands the ALU to ADD.

c. The CU transfers the result from the ALU into register Z of the register section.

Hence, the conventional CPU executes a program by completing one instruction at a time and then proceeding to the next. This means that the control unit would have to wait until the instruction is fetched from memory. Also, the ALU would have to wait until the required data are obtained. Since the speeds of microcontrollers are increasing at a more rapid rate than memory speeds, the control unit and ALU will be idle while the conventional CPU fetches each instruction and obtains the required data. Typical microcontrollers such as the PIC18F utilize the control unit and ALU efficiently by prefetching the next instruction(s) and the required data before the control unit and ALU require them. As mentioned earlier, conventional CPUs execute programs in sequence; typical microcontrollers such as the PIC18F, on the other hand, implement the feature called "pipelining" to prefetch the next instruction while the control unit is busy executing the current instruction. Hence, PIC18F implements pipelining to increase system throughput.

The basic concepts associated with pipelining will be considered next. Assume that a task T is carried out by performing four activities: Al, A2, A3, and A4, in that order. Hardware Hi is designed to perform activity Ai. Hi is referred to as a segment and essentially contains combinational circuit elements. Consider the arrangement shown in Figure 2.16. In this configuration, a latch is placed between two segments so the result computed by one segment can serve as an input to the following segment during the next clock period.

The execution of four tasks Tl, T2, T3, and T4 using the hardware of Figure 2.16 is described using the space-time chart shown in Figure 2.17.

Initially, task Tl is handled by segment 1. After the first clock, segment 2 is busy with Tl while segment 1 is busy with T2. Continuing in this manner, task Tl is completed at the end of the fourth clock. However, following this point, one task is shipped out per clock. This is the essence of the pipelining concept. A pipeline gains efficiency for the

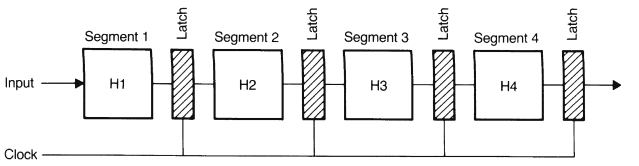


FIGURE 2.16 Four-segment pipeline.

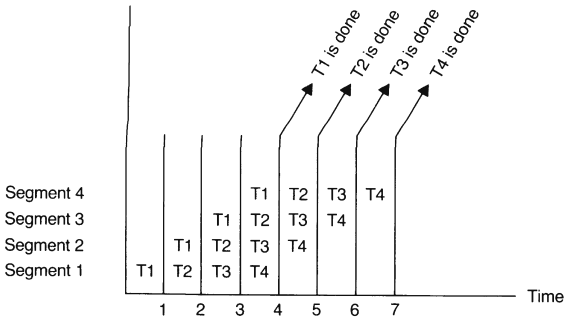


FIGURE 2.17 Overlapped execution of four tasks using a pipeline.

same reason as an assembly line does. Several activities are performed but not on the same material.

The PIC18F implements a two-stage pipeline. As mentioned earlier, the execution of an instruction by a typical CPU is completed in two stages. During the first stage, the instruction is fetched from program memory. During the second stage, the task specified in the instruction is accomplished. Note that the PIC18F CPU fetches the instruction during the first stage like a typical CPU. However, during the second stage, the PIC18F CPU while executing the instruction, fetches the next instruction. This is called "two-stage instruction pipelining", and is used by the PIC18F to increase the speed of instruction execution. It should be mentioned that when the PIC18F fetches a branch instruction, it clears or flushes the pipeline and executes a new sequence of instructions starting at the new branch address.

2.5 RISC vs. CISC

There are two types of CPU architectures: RISC and CISC. A RISC microcontroller such as the PIC18F emphasizes simplicity and efficiency. RISC designs start with a necessary and sufficient instruction set. The purpose of using RISC architecture is to maximize speed by reducing clock cycles per instruction. Almost all computations can be obtained from a few simple operations. The goal of RISC architecture is to maximize the effective speed of a design by performing infrequent operations in software and frequent functions in hardware, thus obtaining a net performance gain. The following list summarizes the typical features of a RISC CPU:

1. The RISC CPU is designed using hardwired control with little or no microcode. Note that variable-length instruction formats generally require microcode design. All RISC instructions have fixed formats, so microcode design is not necessary.
2. A RISC CPU executes most instructions in a single cycle.
3. The instruction set of a RISC CPU typically includes only register, load, and store instructions. All instructions involving arithmetic operations use registers, and load and store operations are utilized to access memory.
4. The instructions have a simple fixed format with few addressing modes.
5. A RISC CPU processes several instructions simultaneously and thus includes pipelining.
6. Software can take advantage of more concurrency. For example, jumps occur after execution of the instruction that follows. This allows the RISC CPU to fetch the next instruction during execution of the current instruction.

RISC CPUs are suitable for embedded applications. *Embedded controllers* are embedded in the host system. This means that the presence and operation of these controllers are basically hidden from the host system. Typical embedded control applications include office automation systems such as printers.

RISC CPUs are well suited for applications such as image processing, robotics, and instrumentation. The key features of the RISC CPUs that make them ideal for these applications are their relatively low level of integration in the chip and instruction pipeline architecture. These characteristics result in low power consumption, fast instruction execution, and fast recognition of interrupts.

CISC CPUs such as the Motorola /Freescale HC11 CPU contain a large number of instructions and many addressing modes. In contrast, RISC CPUs such as the PIC18F include a simple instruction set with a few addressing modes. Almost all computations can be obtained from a few simple operations. RISC basically supports a small set of

commonly used instructions that are executed at a fast clock rate compared to CISC, which contains a large instruction set (some of which are rarely used) executed at a slower clock rate. To implement the fetch/execute cycle for supporting a large instruction set for CISC, the clock is typically slower.

In CISC, most instructions can access memory while RISC contains mostly load/store instructions. The complex instruction set of CISC requires a complex control unit, thus requiring microprogrammed implementation. RISC utilizes hardwired control which is faster. CISC is more difficult to pipeline; RISC provides more efficient pipelining. An advantage of CISC over RISC is that complex programs require fewer instructions in CISC with fewer fetch cycles, while RISC requires a large number of instructions to accomplish the same task with several fetch cycles. However, RISC can significantly improve its performance with a faster clock, more efficient pipelining, and compiler optimization.

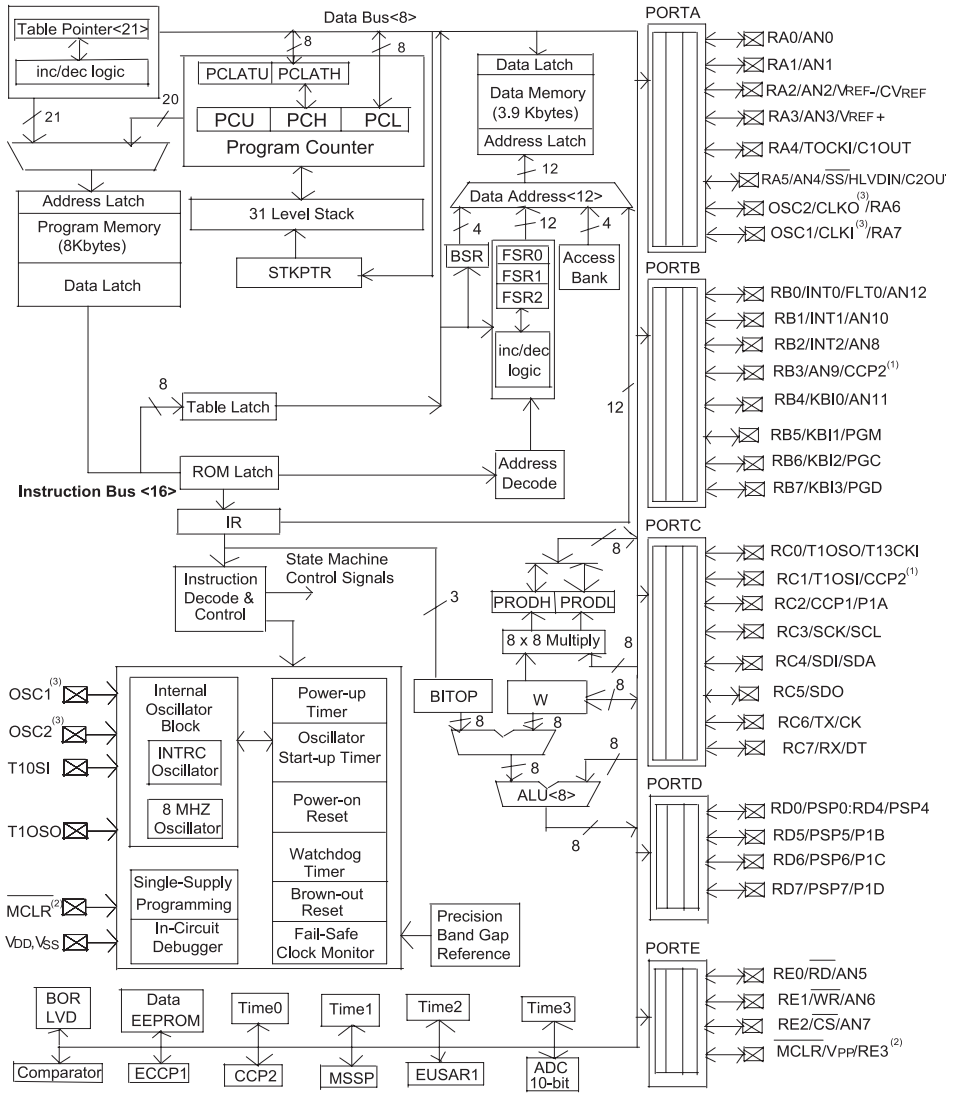
2.6 Functional Representation of a Typical Microcontroller---- The PIC18F4321

Figure 2.18 depicts the functional block diagram of the PIC18F4321 microcontroller. The block diagram can be divided into three sections, namely CPU, Memory, and I/O (Input/Output). A brief description of these blocks will be provided in the following.

The PIC18F4321 CPU contains registers, ALU, an instruction decode and control unit, and oscillator blocks. Typical CPU registers include IR (Instruction Register), W (Accumulator), Program Counter (PC), three Memory Address Registers (FSR0 through FSR2), and Stack Pointer (STKPTR). An on-chip hardware multiplier is also included for performing unsigned multiplication. These registers are described in more detail in chapter 5.

The on-chip memory contains program memory and data memory. As mentioned before, the PIC18F4321 is designed using the Harvard architecture, a separate 21-bit address bus for program memory, and a separate 12-bit address bus for data memory shown in Figure 2.18.

The on-chip I/O block includes five I/O ports (Port A through Port E), four hardware timers, 10-bit ADC (Analog to Digital Converter), CCP (Capture, Compare, PWM), and associated modules. As mentioned before, the PIC18F4321 can perform functions such as capture, compare, and pulse width modulation (PWM) using the timers and CCP (Capture / Compare / PWM) modules. The PIC18F4321 can compute the period of an incoming signal using the capture module. The PIC18F4321 can produce a periodic waveform or time delays using the compare module. The PIC18F4321's on-chip PWM can be used to obtain pulse waveforms with a particular period and duty cycle which are ideal for applications such as motor control.



Note 1: CCP2 is multiplexed with RC1 when Configuration bit, CCP2MX, is set, or RB3 when CCP2MX is not set.

2: RE3 is only available when MCLR functionality is disabled.

3: OSC1/CLKI and OSC2/CI KO are only available in select oscillator modes and when these pins are not being used as digital I/O.

Figure 2.18 PIC18F4321 Block Diagram

QUESTIONS AND PROBLEMS

2.1 What is the difference between a single-chip microcomputer and a microcontroller?

2.2 What is an 8-bit microcontroller? Name one commercially available 8-bit microcontroller.

2.3 What is the difference between:
(a) A program counter and a memory address register?
(b) An accumulator and an instruction register?
(c) A general-purpose register-based CPU and an accumulator-based CPU.

2.4 Assuming signed numbers, find the sign, carry, zero, and overflow flags of:
(a) $09_{16} + 17_{16}$
(b) $A5_{16} - A5_{16}$
(c) $71_{16} - A9_{16}$
(d) $6E_{16} + 3A_{16}$
(e) $7E_{16} + 7E_{16}$

2.5 What is the difference between PUSH and POP operations in the stack?

2.6 Suppose that an 8-bit microcontroller has a 16-bit stack pointer and uses a 16-bit register to access the stack from the top. Assume that initially the stack pointer and the 16-bit register contain $20C0_{16}$ and 0205_{16} respectively. After the PUSH operation:
(a) What are the contents of the stack pointer?
(b) What are the contents of memory locations $20BE_{16}$ and $20BF_{16}$ using little-endian format?

2.7 What is the main purpose of the hardware reset pin on the microcontroller chip?

2.8 How many bits are needed to access a 4 MB data memory? What is the hexadecimal value of the last address in this memory?

2.9 If the last address of an on-chip memory is 0x7FF, determine its size.

2.10 What is the difference between von Neumann and Harvard CPU architectures? Provide an example of a commercially available microcontroller using each type of CPU.

2.11 What is the basic difference between program execution by a conventional CPU and the PIC18F CPU?

2.12 Discuss the basic features of RISC and CISC.

2.13 Discuss briefly the purpose of the functional units (CCP, A/D, Serial communication) implemented in the PIC18F.

2.14 What does pipelining mean?

2.15 Summarize the basic features of PIC18F pipelining.

3
MICROCONTROLLER MEMORY AND INPUT/OUTPUT (I/O)

In this chapter we describe basic concepts of memory organization and Input/Output techniques associated with typical microcontrollers. We will also discuss signals common to CPU, memory, and I/O circuits inside typical microcontroller chips. Topics include main memory array design, and programmed and interrupt I/O.

3.1 Introduction to Microcontroller Memory

A memory unit is an integral part of any microcontroller, and its primary purpose is to hold instructions and data. The major design goal of on-chip memory inside the microcontroller is to allow it to operate at a speed close to that of the CPU. In a broad sense, a microcontroller memory system can be divided into two groups:

- CPU registers
- Primary or main memory

Microcontroller CPU registers are used to hold temporary results when a computation is in progress. Also, there is no speed disparity between these registers and the CPU because they are fabricated using the same technology. However, the cost involved in this approach limits a microcontroller architect to include only a few registers in the CPU.

In an accumulator-based CPU such as the PIC18F microcontroller, typical registers inside the CPU include the Accumulator, Program Counter, Stack Pointer, and Status Register. A general-purpose-register-based microcontroller such as the Texas Instrument's MSP430 contains both dedicated registers, and general-purpose registers. Typical dedicated registers include Program Counter, Stack Pointer, and Status Register. In addition, several general-purpose registers are also provided and any of these registers can be used as an accumulator.

Primary or main memory is the storage area in which all programs are executed. The microcontroller can directly access only those items that are stored in main memory. Therefore, all programs must be in the main memory prior to execution. CMOS technology is normally used in the main memory design. The size of the main memory is usually much larger than the number of registers, and its operating speed is slower than that of processor registers.

Typical microcontrollers such as the PIC18F4321 use Harvard architecture, and hence, its main memory is divided into two types of memory: Program memory and data memory. The program memory is contained in flash memory while the data memory is contained in SRAM.

Microcontroller Theory and Applications with the PIC18F, Second Edition. M. Rafiquzzaman.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

3.1.1 Main memory

As mentioned before, the main memory (or simply, the memory) stores both instructions and data. For 8-bit microcontrollers, the memory is divided into a number of 8-bit units called *memory words*. An 8-bit unit of data is termed a *byte*. Therefore, for an 8-bit microcontroller, *memory word* and *memory byte* mean the same thing. For 16-bit microcontrollers, a word contains two bytes (16 bits). A memory word is identified in the memory by an address. For example, the PIC18F is an 8-bit microcontroller, and can directly address a maximum of two Megabytes (2^{21}) of program memory space. The data memory address, on the other hand, is 12-bit wide. Hence, the PIC18F can directly address data memory of up to 4kbytes (2^{12}). This provides a maximum of $2^{12} = 4096$ bytes of data memory addresses, ranging from 000 to FFF in hexadecimal.

An important characteristic of a memory is whether it is volatile or nonvolatile. The contents of a volatile memory are lost if the power is turned off. On the other hand, a nonvolatile memory retains its contents after power is switched off. ROM is a typical example of nonvolatile memory. RAM is a volatile memory unless backed up by batteries.

Large areas of data memory require an efficient addressing scheme to make rapid access to any address possible. Ideally, this means that an entire address does not need to be provided for each read or write operation. For PIC18F, this is accomplished with a RAM

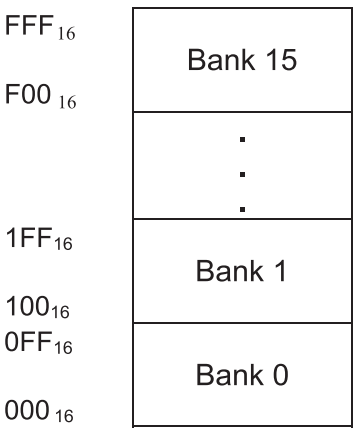


FIGURE 3.1 PIC18F data memory.

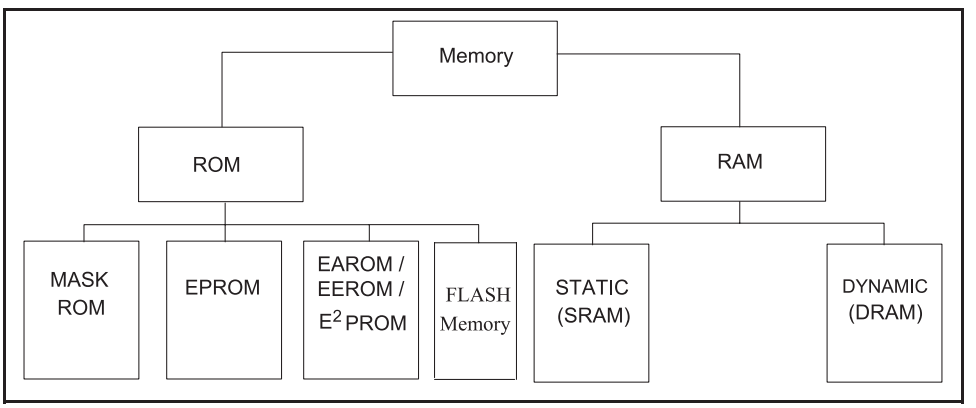


FIGURE 3.2 Summary of available semiconductor memories for microcontroller systems.

banking scheme. This divides the memory space into 16 contiguous banks (bank 0 through 15) of 256 bytes. Depending on the instruction, each location can be addressed directly by its full 12-bit address, or an 8-bit low-order address and a 4-bit Bank Pointer (also called Bank Select Register, BSR).

There are some advantages of using memory banks. Once the Bank Select Register (4 bits in this case) is initialized with the bank number using instructions, fewer bits (8 bits in this case) are needed for data memory addresses. The PIC18F data memory uses memory banks.

Figure 3.1 shows a simplified data memory layout of the PIC18F. In the figure, the high 4 bits of an address specify the bank number. As an example, consider address 0x105 of segment 1. The high 4 bits, 0001, of this address define the location as in bank 1, and the low 8 bits, 0x05, specify the particular address in bank 1.

Memories can be categorized into two main types: read-only memory (ROM) and random-access memory (RAM). As shown in Figure 3.2, ROMs and RAMs are then divided into a number of subcategories, which are discussed next.

Read-Only Memory ROMs (Read-Only memories) can only be read, so it is nonvolatile memory. CMOS technology is used to fabricate ROMs. ROMs are divided into two common types: mask ROM and Erasable Programmable ROM (EPROM) such as the 2732, and EAROM (Electrically Alterable ROM) [also called EEPROM or E^2PROM (electrically erasable PROM)] such as the 2864.

Mask ROMs are programmed by a masking operation performed on a chip during the manufacturing process. The contents of mask ROMs are permanent and cannot be changed by the user. EPROMs can be programmed, and their contents can also be altered by using special equipment, called an *EPROM programmer*. When designing a microcontroller for a particular application, permanent programs are stored in ROMs. Control memories used to microprogram the control unit are ROMs.

EPROMs can be reprogrammed and erased. The EPROM chip must be removed from the system before programming. This memory is erased by exposing the chip to ultraviolet light via a lid or window on the chip. Typical erase times vary between 10 and 20 minutes. The EPROM can be programmed by inserting the EPROM chip into a socket of the EPROM programmer and providing proper addresses and voltage pulses at the appropriate pins of the chip.

EEPROMs can be programmed without removing the memory from the ROM's sockets. These memories are also called *read-mostly memories* (RMMs), because they have much slower write times than read times. Therefore, these memories are usually suited for operations when mostly reading rather than writing is performed.

Flash Memory Another type of memory, called *Flash memory* (nonvolatile), invented in the mid-1980s by Toshiba, is designed using a combination of EPROM and E^2PROM technologies. Flash memory can be reprogrammed electrically while embedded on the board. One can change multiple bytes at a time. An example of flash memory is the Intel 28F020 (256K x 8-bit). Flash memory is typically used in cellular phones and digital cameras. Note that the PIC18F uses flash memory as its program memory.

Random-Access Memory There are two types of RAM: static RAM (SRAM), and dynamic RAM (DRAM). *Static RAM* stores data in flip-flops. Therefore, this memory does not need to be refreshed. RAMs are volatile unless backed up by battery. The PIC18F uses SRAM for its data memory.

Dynamic RAM stores data in capacitors. That is, it can hold data for a few milliseconds. Hence, dynamic RAMs are refreshed typically by using external refresh circuitry. Dynamic RAMs (DRAMs) are used in applications requiring large memory. DRAMs have higher densities than static RAMs (SRAMs). Typical examples of DRAMs are the 4464 (64k × 4-bit), 44256 (256k × 4-bit), and 41000 (1M × 1-bit). DRAMs are inexpensive, occupy less space, and dissipate less power than SRAMs. Two enhanced versions of DRAM are EDO DRAM (extended data output DRAM) and SDRAM (synchronous DRAM).

3.1.2 READ and WRITE Timing Diagrams

To execute an instruction, the CPU of the microcontroller reads or fetches the op-code via the data bus from a memory location in the ROM/RAM external to the CPU. It then

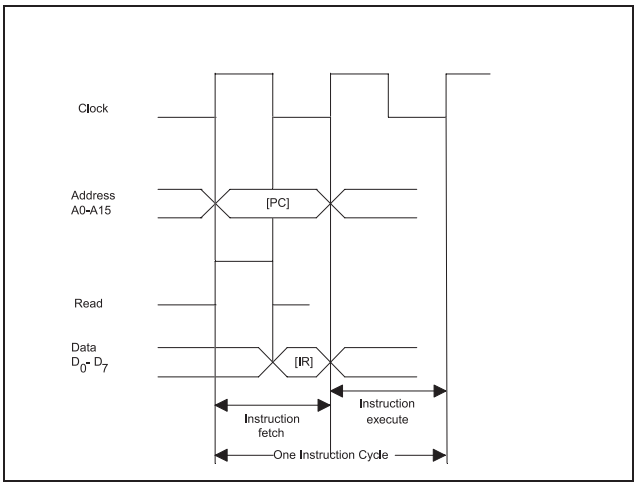


FIGURE 3.3 Typical instruction fetch timing diagram for an 8-bit microprocessor.

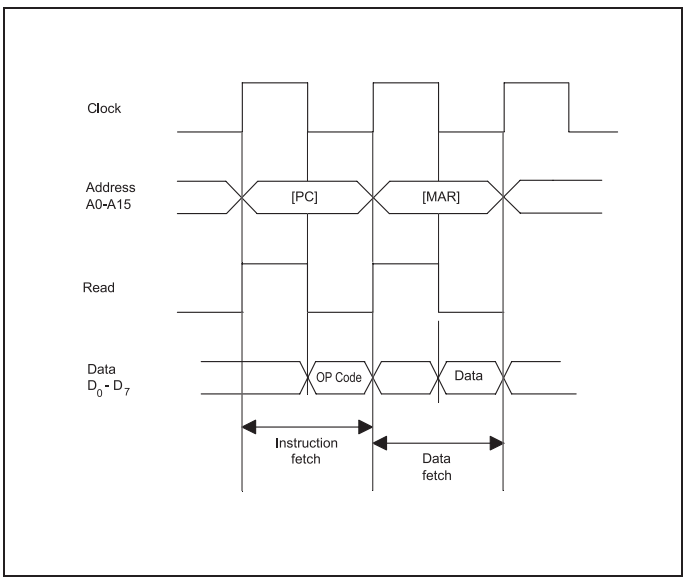


FIGURE 3.4 Typical memory READ timing diagram.

places the op-code (instruction) in the instruction register. Finally, the CPU executes the instruction. Therefore, the execution of an instruction consists of two portions, instruction fetch and instruction execution. We consider the instruction fetch, memory READ, and memory WRITE timing diagrams in the following using a single clock signal. Figure 3.3 shows a typical instruction fetch timing diagram.

In Figure 3.3, to fetch an instruction, when the clock signal goes to HIGH, the CPU places the contents of the program counter on the address bus via address pins A_0–A_{15} on the chip. Note that since each of lines A_0–A_{15} can be either HIGH or LOW, both transitions are shown for the address in Figure 3.3. The instruction fetch is basically a memory READ operation. Therefore, the CPU raises the signal on the READ pin to HIGH. As soon as the clock goes to LOW, the logic external to the CPU gets the contents of the memory location addressed by A_0–A_{15} and places them on the data bus D_0–D_7. The CPU then takes the data and stores it in the instruction register so that it gets interpreted as an instruction. This is called *instruction fetch*. The CPU performs this sequence of operations for every instruction.

We now describe the READ and WRITE timing diagrams. A typical READ timing diagram is shown in Figure 3.4. Memory READ is basically loading the contents of a memory location of the main ROM/RAM into an internal register of the CPU. The address of the location is provided by the contents of the memory address register (MAR). Let us now explain the READ timing diagram of Figure 3.4.

1. The CPU performs the instruction fetch cycle as before to READ the op-code.
2. The CPU interprets the op-code as a memory READ operation.
3. When the clock pin signal goes HIGH, the CPU places the contents of the memory address register on the address pins A_0–A_{15} of the chip.
4. At the same time, the CPU raises the READ pin signal to HIGH.
5. The logic external to the CPU gets the contents of the location in the main ROM/RAM addressed by the memory address register and places it on the data bus.
6. Finally, the CPU gets this data from the data bus via pins D_0 – D_7 and stores it in an internal register.

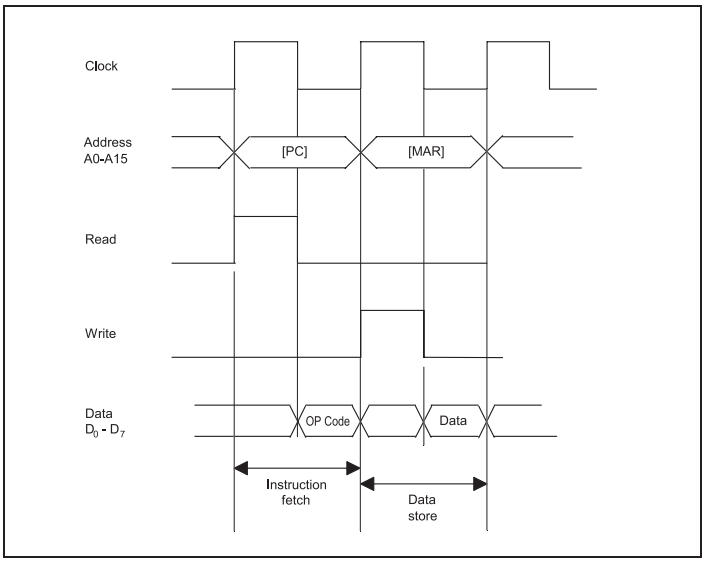


FIGURE 3.5 Typical memory WRITE timing diagram.

Memory WRITE is basically storing the contents of an internal register of the CPU into a memory location of the main RAM. The contents of the memory address register provide the address of the location where data is to be stored. Figure 3.5 shows a typical WRITE timing diagram.

1. The CPU fetches the instruction code as before.
2. The CPU interprets the instruction code as a memory WRITE instruction and then proceeds to perform the DATA STORE cycle.
3. When the clock pin signal goes HIGH, the CPU places the contents of the memory address register on the address pins A_0–A_{15} of the chip.
4. At the same time, the CPU raises the WRITE pin signal to HIGH.
5. The CPU places data to be stored from the contents of an internal register onto data pins D_0–D_7.
6. The logic external to the CPU stores the data from the register into a RAM location addressed by the memory address register.

3.1.3 Main Memory Organization

Typical microcontroller on-chip main memory, also called "memory module", may include ROM/EPROM/EPROM, and SRAM. As mentioned earlier, the PIC18F4321 main memory (program memory and data memory) consists of Flash memory and SRAMs. A microcontroller system designer is normally interested in how the microcontroller memory is organized, or in other words, how to connect the memory units to the CPU, and then determine the memory map of the microcontroller. That is, the PIC18F4321 designer would be interested in finding out what memory locations are assigned to the Flash memory and SRAMs.

Main Memory Array Design In a typical microcontroller, the designer has to implement the required capacity by interconnecting several memory circuits to the CPU. This concept is known as *memory array design*. We address this topic in this section and show how to interface a data memory system (SRAM) with a typical CPU. In the following, we will use common signals associated with the CPU and memory units internal to typical microcontrollers.

Now let us discuss how to design SRAM arrays. In particular, our discussion is focused on the design of memory arrays for a hypothetical CPU. The pertinent signals of a typical CPU necessary for main memory interfacing are shown in Figure 3.6. There are 16 address lines, A_{15}-A_0, with A_0 being the least significant bit. This means that this CPU can address directly a maximum of $2^{16} = 65{,}536$ or 64k bytes of memory locations.

The control line M/$\overline{IO}$ goes LOW if the CPU executes an I/O instruction; it is held HIGH if the CPU executes a memory instruction. Similarly, the CPU drives control line R/$\overline{W}$ HIGH for READ operation; it is held LOW for WRITE operation. Note that all 16

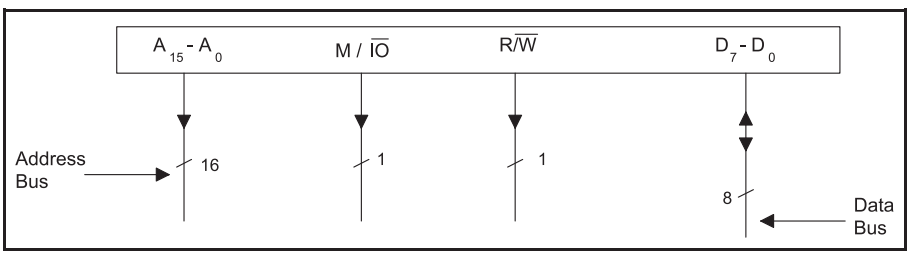


FIGURE 3.6 Pertinent signals of a typical CPU required for main memory interfacing.

address lines and the two control lines ($M/\overline{IO}$, $R/\overline{W}$) described so far are unidirectional in nature; that is, information can always travel on these lines from the processor to external units. Eight bidirectional data lines, D_7-D_0 (with D_0 being the least significant bit), are also shown in Figure 3.6. These lines are used to allow data transfer from the CPU to memory module, and vice versa.

The block diagram of a typical 1k × 8 RAM SRAM unit is shown in Figure 3.7. In this circuit, there are 10 address lines, A_9-A_0, so one can read or write 1024 ($2^{10} = 1024$) different memory words. Also, in this chip there are eight bidirectional data lines, D_7-D_0, so that information can travel back and forth between the CPU and the memory module. The three control lines $\overline{CS1}$, CS2, and $R/\overline{W}$ are used to control the SRAM unit according to the truth table shown in Table 3.1 from which it can be concluded that the RAM chip is enabled only when $\overline{CS1} = 0$ and CS2 = 1. Under this condition, $R/\overline{W} = 0$ and $R/\overline{W} = 1$ imply write and read operations, respectively.

To connect a CPU to the memory module, two address decoding techniques are commonly used for each memory type: linear decoding and full decoding. Let us discuss first how to interconnect a CPU with a 4K SRAM array comprised of the four 1k SRAM units of Figure 3.7 using the linear decoding technique. Figure 3.8 uses linear decoding to accomplish this. In this approach, address lines A_9-A_0 of the CPU are connected to all SRAM units. Similarly, the control lines $M/\overline{IO}$ and $R/\overline{W}$ of the CPU are connected to control lines CS2 and $R/\overline{W}$ respectively to each of the SRAM units. The high-order address bits A_{10}-A_{13} act directly for selecting memory units. In particular, address lines A_{10} and A_{11} select SRAM units I and II, respectively. Similarly, the address lines A_{12} and A_{13} select the SRAM units III and IV, respectively. A_{15} and A_{14} are don't cares and are assumed to be zero.

TABLE 3.1 Truth Table for Controlling SRAM unit.

$\overline{CS1}$	CS2	$R/\overline{W}$	Function
0	1	0	Write Operation
0	1	1	Read Operation
1	X	X	The chip is not selected
X	0	X	The chip is not selectd

X means Don't Care

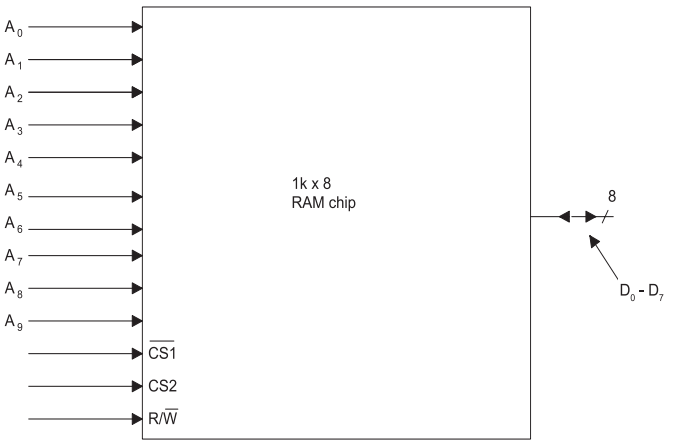


FIGURE 3.7 Typical 1k × 8 SRAM unit.

Table 3.2 describes how the addresses are distributed among the four 1k SRAM units. The primary advantage in this method, known as *linear select decoding*, is that it does not require decoding hardware. However, if two or more of lines A_{10}-A_{13} are low at the same time, more than one SRAM units are selected, and this causes a bus conflict.

Because of this potential problem, the software must be written such that it never reads into or writes from any address in which more than one of bits A_{13}-A_{10} are low. Another disadvantage of this method is that it wastes a large amount of address space. For

TABLE 3.2 Address Map of the Memory Organization of Figure 3.8

Address Range (Hex)I	SRAM Number
3800-3BFF	I
3400-37FF	II
2C00-2FFF	III
1C00-1FFF	IV

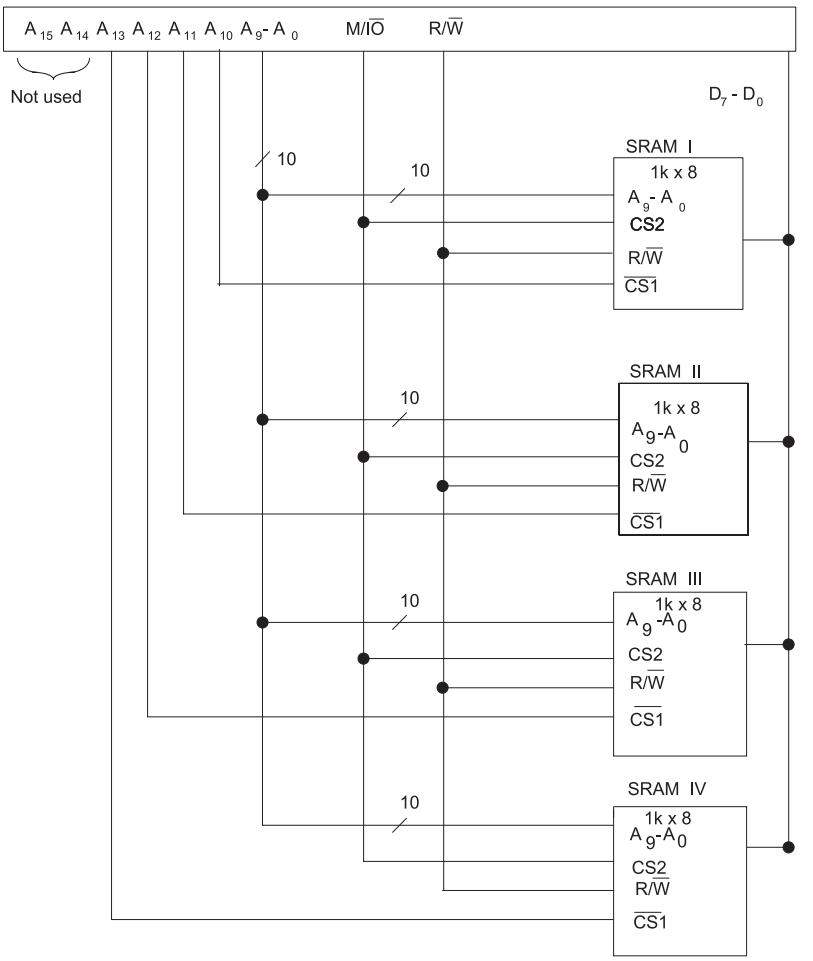


FIGURE 3.8 CPU connected to 4k SRAM using the linear select decoding technique.

example, whenever the address value is B800 or 3800, SRAM chip I is selected. In other words, address 3800 is the mirror reflection of address B800 (this situation is also called *memory foldback*). This technique is therefore limited to a small system. The system of Figure 3.8 can be expanded up to a total capacity of 6k using A_{14} and A_{15} to select two more 1k SRAM units.

To resolve problems with linear decoding, we use full decoded memory addressing. In this technique we use a decoder. The 4K memory system designed using this technique is shown in Figure 3.9. In Figure 3.9 the decoder output selects one of the four 1K SRAM's, depending on the values of A_{12}, A_{11}, and A_{10} (Table 3.3).

Note that the decoder output will be enabled only when $\overline{E3} = \overline{E2} = 0$ and $E1 = 1$. Therefore, in the organization of Figure 3.9, when any one of the high-order bits A_{15}, A_{14}, or A_{13} is 1, the decoder will be disabled, and thus none of the SRAM's will be selected. In this arrangement, the memory addresses are assigned as shown in Table 3.4.

This approach does not waste any address space since the unused decoder outputs (don't cares) can be used for memory expansion. For example, the 3-to-8 decoder of Figure 3.9 can select eight 1k SRAM's. Also, this method does not generate any bus conflict. This is because the decoder output selected ensures enabling of one memory unit at a time.

3.2 Microcontroller Input/Output (I/O)

The technique of data transfer between a microcontroller and an external device is called *input/ output* (I/O). One communicates with a microcontroller via the I/O devices interfaced to it. The user can enter programs and data using the keyboard on a terminal and execute the programs to obtain results. Therefore, the I/O devices connected to a microcontroller provide an efficient means of communication between the microcontroller and the outside world. These I/O devices, commonly called *peripherals*, include keyboards, seven-segment displays, and LCD's (Liquid Crystal Displays).

There are two ways of transferring data between a microcontroller and I/O devices. These are programmed I/O, and interrupt I/O. Using *programmed I/O*, the CPU executes a program to perform all data transfers between the CPU and the external device. The main characteristic of this type of I/O technique is that the external device carries out the functions dictated by the program contained in the microcontroller memory. In other

TABLE 3.3 Decoding Guide.

A_{12}	A_{11}	A_{10}	SRAM Number
0	0	0	I
0	0	1	II
0	1	0	III
0	1	1	IV

TABLE 3.4 Address Map of the Memory Organization of Figure 3.9.

Address Range (Hex)	SRAM Number
0000-03FF	I
0400-07FF	II
0800-0BFF	III
0C00-0FFF	IV

words, the CPU controls all transfers completely.

In *interrupt I/O*, an external device can force the CPU to stop executing the current program temporarily so that it can execute another program known as an *interrupt service routine*. This routine satisfies the needs of the external device. After completing this program, a return from interrupt instruction can be executed at the end of the service routine to return control at the right place in the main program.

The interrupt procedure is similar in concept to the procedure associated with subroutine CALL and RETURN instructions. The subroutine CALL /RETURN includes a main program and a subroutine while the interrupt contains a main program and a service routine. The subroutine CALL instruction pushes the current contents of the program counter onto the stack. The RETURN instruction placed at the end of the subroutine pops the previously pushed program counter, and returns control to the main program.

The interrupt, on the other hand, is initiated externally via hardware or internally via occurrence of events such as completion of ADC (Analog to Digital Conversion). Once the interrupt is recognized, the microcontroller normally pushes the Program Counter (PC) and the Status Register (SR) onto the stack, and automatically branches to an address predefined by the manufacturer. The user writes a program called "interrupt

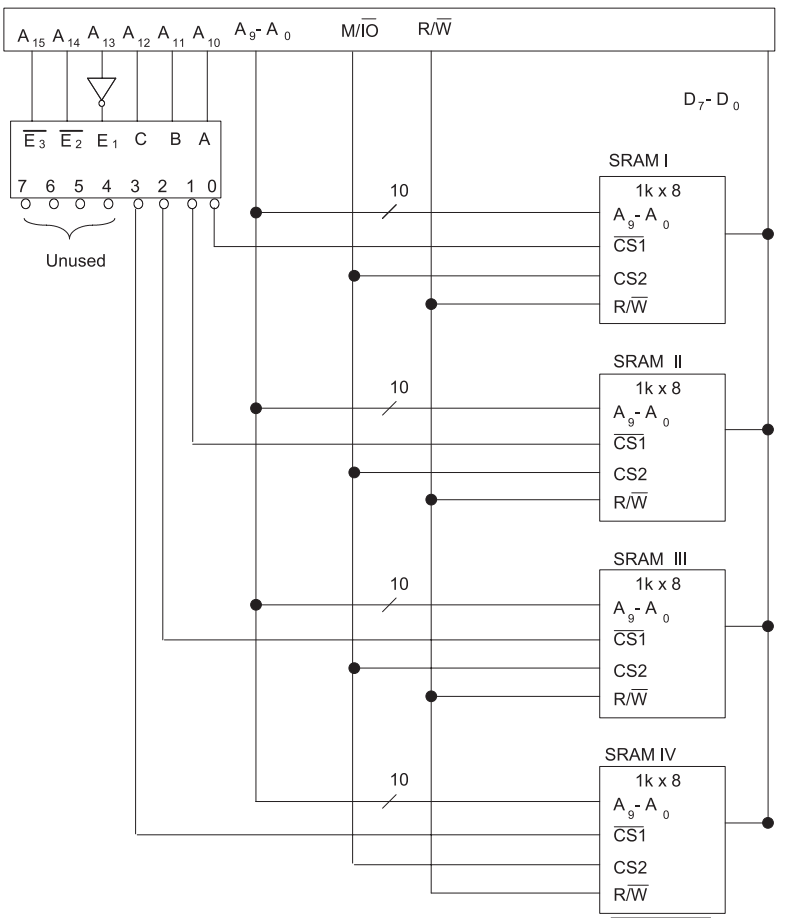


FIGURE 3.9 Interconnecting a CPU with a 4k RAM using full decoded memory addressing.

service routine" at this address. This program is similar to the subroutine. A "Return from Interrupt" instruction placed by the user at the end of the interrupt service routine will pop the previously pushed PC and SR , and will return control to the main program at the proper location.

3.2.1 Overview of digital output circuits

For simplicity, a basic background in TTL outputs will be provided next. Since the CMOS technology is used in designing typical microcontrollers, these concepts will then be related to CMOS outputs.

TTL Outputs There are three types of output configurations for TTL. These are open-collector output, totem-pole output, and tristate (three-state) output. The open-collector output means that the TTL output is a transistor with nothing connected to the collector. The collector voltage provides the output of the gate. For the open-collector output to work properly, a resistor (called the pull-up resistor), with a value of typically 1 Kohm, should be connected between the open collector output and a +5 V power supply.

If the outputs of several open-collector gates are tied together with an external resistor (typically 1 Kohm) to a +5 V source, a logical AND function is performed at the connecting point. This is called wired-AND logic.

Figure 3.10 shows two open-collector outputs (*A* and *B*) that are connected together to a common output point *C* via a 1 KΩ resistor and a +5 V source.

The common-output point *C* is HIGH only when both transistors are in cutoff (OFF) mode, providing *A* = HIGH and *B* = HIGH. If one or both of the two transistors is turned ON, making one (or both open-collector outputs) LOW will drive the common output *C* to LOW. Note that a LOW (Ground for example) signal when connected to a HIGH (+5V for example) signal generates a LOW. Thus, *C* is obtained by performing a logical AND operation of the open collector outputs *A* and *B*.

Let us briefly review the totem-pole output circuit shown in Figure 3.11. The circuit operates as follows:

When transistor Q_1 is ON, transistor Q_2 is OFF. When Q_1 is OFF, Q_2 is ON. This is how the totem-pole output is designed. The complete TTL gate connected to the bases of transistors Q_1 and Q_2 is not shown; only the output circuit is shown.

In the figure, Q_1 is turned ON when the logic gate circuit connected to its base sends a HIGH output. The switches in transistor Q_1 and diode D close while the switch in Q_2 is open. A current flows from the +5 V source through R, Q_1, and D to the output. This

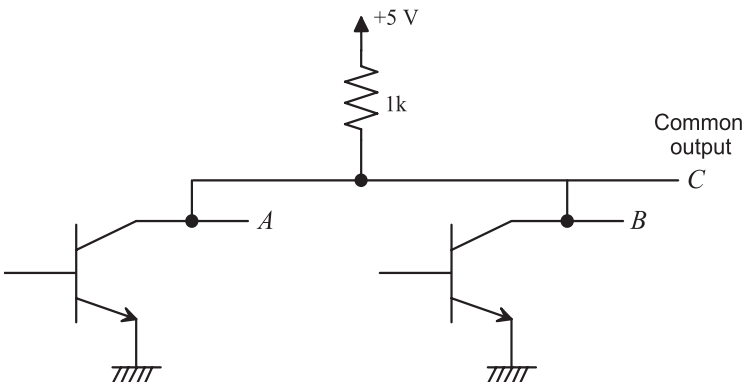


FIGURE 3.10 Two open-collector outputs A and B tied together

current is called I_{source} or output high current, I_{OH}. This is typically represented by a negative sign in front of the current value in the TTL data book, a notation indicating that the chip is losing current. For a low output value of the logic gate, the switches in Q_1 and D are open and the switch in Q_2 closes. A current flows from the output through Q_2 to ground. This current is called I_{sink} or Output Low current, I_{OL}. This is represented by a positive sign in front of the current value in the TTL data book, indicating that current is being added to the chip. Either I_{source} or I_{sink} can be used to drive a typical output device such as an LED. I_{source} (I_{OH}) is normally much smaller than I_{sink} (I_{OL}). I_{source} (I_{OH}) is typically -0.4 mA (or $-400\ \mu$A) at a minimum voltage of 2.7 V at the output. I_{source} is normally used to drive devices that require high currents. A current amplifier (buffer) such as a transistor or an inverting buffer chip such as 74LS368 needs to be connected at the output if I_{source} is used to drive a device such as an LED requiring high current (10 mA to 20 mA). I_{sink} is normally 8 mA.

The totem-pole outputs must not be tied together. When two totem-pole outputs are connected together with the output of one gate HIGH and the output of the second gate LOW, the excessive amount of current drawn can produce enough heat to damage the transistors in the circuit.

Tristate is a special totem-pole output that allows connecting the outputs together like the open-collector outputs. When a totem-pole output TTL gate has this property, it is called a tristate (three state) output. A tristate has three output states:

1. A LOW level state when the lower transistor in the totem-pole is ON and the upper transistor is OFF.
2. A HIGH level when the upper transistor in the totem-pole is ON and the lower transistor is OFF.
3. A third state when both output transistors in the totem-pole are OFF. This third state provides an open circuit or high-impedance state which allows a direct wire connection of many outputs to a common line called the bus.

CMOS Outputs Like TTL, the CMOS logic offers three types of outputs. These are push-pull (totem-pole in TTL), open drain (open collector in TTL), and tristate outputs. For example, the 74HC00 contains four independent 2-input NAND gates and includes push-pull output. The 74HC03 also contains four independent 2-input NAND gates, but has open drain outputs. The 74HC03 requires a pull-up resistor for each gate. The 74HC125 contains four independent tri-state buffers in a single chip. Note that CMOS technology is normally used in designing microcontrollers.

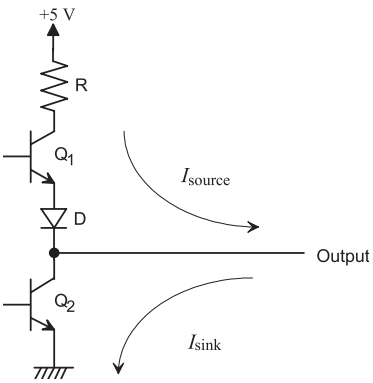


FIGURE 3.11 TTL Totem-pole output

3.2.2 Simple I/O Devices

A simple input device such as a DIP switch can be connected to a microcontroller's I/O port as shown in Figure 3.12. The figure shows a switch circuit that can be used as a single bit input into an I/O port. When the DIP switch is open, V_{IN} is HIGH. When the switch is closed, V_{IN} is LOW. V_{IN} can be used as an input bit for performing laboratory experiments. Note that unlike TTL, a 1Kohm resistor is connected between the switch and the input of the MOS gate. This provides protection against static discharge.

For performing simple I/O experiments using programmed I/O, light-emitting diodes (LEDs) and seven-segment displays can be used as output devices. An LED is typically driven by low voltage and low current, which makes it a very attractive device for use with microcontrollers.

Table 3.5 provides the current and voltage requirements for red, yellow, and green LEDs. Basically, an LED will be ON, generating light when its cathode is sufficiently negative with respect to its anode. A microcontroller can therefore light an LED either by grounding the cathode (if the anode is tied to +5 V) or by applying +5 V to the anode (if the cathode is grounded) through an appropriate resistor value. A typical hardware interface between a microcontroller and an LED is depicted in Figure 3.13.

A typical microcontroller such as the PIC18F outputs adequate current to turn an LED ON or OFF. In Figure 3.13 (a), a '1' from the microcontroller will turn the LED ON while a '0' will turn it OFF. In Figure 3.13 (b) on the other hand, a '0' from the microcontroller will turn the LED ON while a '1' will turn it OFF.

From Table 3.5, a red LED requires 10 mA current at 1.7V. In Figure 3.13 (a), a HIGH at the microcontroller output will turn the LED ON. This will allow a path of current to flow from the +5 V source through R and the LED to the ground. In Figure 3.13 (b), a LOW at the microcontroller output will turn the LED ON. This will allow a path of current to flow from the +5V source through R and the LED to the ground (microcontroller I/O port). The appropriate value of R needs to be calculated to satisfy the voltage and current requirements of the LED. The value of R can be calculated as follows:

$$R = \frac{5 - 1.7}{10\ \text{mA}} = \frac{5 - 1.7}{10\ \text{mA}} = 330\ \Omega$$

Therefore, the interface design is complete, and a value of R = 330 Ω is required. A seven-segment display can be used with programmed I/O to display, for example, decimal numbers from 0 to 9. The name *seven segment* is based on the fact that there are

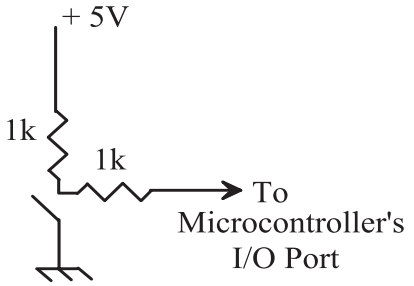


FIGURE 3.12 Typical switch for a microcontroller's input.

TABLE 3. 5 Current and Voltage Requirements of LEDs

LEDs	Red	Yellow	Green
Current	10 mA	10 mA	20 mA
Voltage	1.7 V	2.2V	2.4V

seven LEDs, one in each segment of the display. Figure 3.14 shows a typical seven-segment display. In the figure, each segment contains an LED. All decimal numbers from 0 through 9 can be displayed by turning the appropriate segment ON or OFF. For example, a '0' can be displayed by turning the LED in segment *g* OFF and turning the other six LEDs in segments *a* through *f* ON. There are two types of seven-segment displays: send a HIGH to light a segment and a LOW to turn it off. In a common-anode configuration, on the other hand, the microcontroller sends a LOW to light a segment and a HIGH to turn it off.

Seven-segment displays can be interfaced to typical microcontrollers using programmed I/O. BCD to seven-segment code converter chips such as 7447 or 7448 can be replaced by a lookup table**.** This table can be stored in a microcontroller's memory. An assembly language program can be written to read the appropriate code for a BCD digit stored in this table. This data can be outputted to display the BCD digit on a seven-segment display connected to an I/O port of the microcontroller. Programs to accomplish this are written in PIC18F assembly and C languages in Chapter 8.

3.2.3 Programmed I/O

A microcontroller communicates with an external device via one or more registers called *I/O ports* using programmed I/O. Each bit in the port can be configured individually as either input or output. Each port can be configured as an input or output port by another register usually called the *Data Direction Register (DDR)*. The port contains the actual input or output data. The data-direction register is an output register and can be used to configure the bits in the port as inputs or outputs.

Each bit in the port can be set up as an input or output, normally by writing a 0 or a 1 in the corresponding bit of the DDR. The PIC18F microcontroller makes an I/O port bit an input by writing a '1' in the corresponding bit in DDR. On the other hand, writing a '0' in a particular bit in DDR will configure the corresponding bit in the port as an output.

For example, if an 8-bit DDR in the PIC18F contains 0xCB (CB Hex), the corresponding port is defined as shown in Figure 3.15. In this example, because 0xCB (1100 1011) is stored in the data-direction register, bits 0, 1, 3, 6, and 7 of the port are set

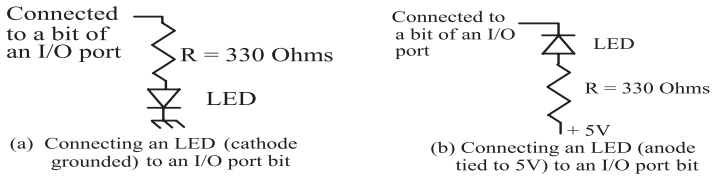


FIGURE 3.13 Interfacing LED to PIC18F

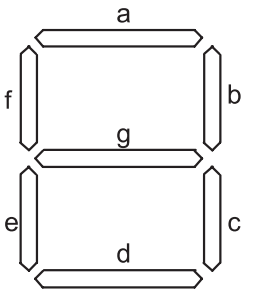


FIGURE 3.14 A seven-segment display

up as inputs, and bits 2, 4, and 5 of the port are defined as outputs. The microcontroller can then send output to external devices, such as LEDs, connected at bits 2, 4, and 5 through a proper interface.

Similarly, the microcontroller can input the status of external devices, such as switches, through bits 0, 1, 3, 6, and 7. To input data from the input switches, the microcontroller inputs the complete byte, including the bits to which LEDs are connected. While receiving input data from an I/O port, however, the microcontroller places a value, probably 0, at the bits configured as outputs and the program must interpret them as "don't cares." At the same time, the microcontroller's outputs to bits configured as inputs are disabled.

I/O ports are addressed using either standard I/O or memory-mapped I/O techniques. *Using Standard I/O* or sometimes called *port I/O* (also called *isolated I/O* by Intel), the CPU outputs an internal signal such as the M/$\overline{IO}$ for memory and I/O units on the microcontroller chip. The CPU outputs a HIGH on M/$\overline{IO}$ to indicate to memory and the I/O that a memory operation is taking place. A LOW output from the CPU to M/$\overline{IO}$ indicates an I/O operation. Execution of an IN or OUT instruction makes the M/$\overline{IO}$ LOW, whereas memory-oriented instructions, such as MOVE, drive the M/$\overline{IO}$ to HIGH.

In standard I/O, the CPU uses the M/ $\overline{IO}$ output signal to distinguish between I/O and memory. Intel microcontrollers such as the 8051 uses standard I/O.

In *memory-mapped I/O*, the CPU does not use the M/$\overline{IO}$ control signal. Instead, the CPU uses an unused address pin to distinguish between memory and I/O. The CPU uses a portion of the memory addresses to represent I/O ports. The I/O ports are mapped as part of the microcontroller's main memory addresses which may not exist physically, but are used by the microcontroller's memory-oriented instructions, such as MOVE, to generate the necessary control signals to perform I/O. The PIC18F uses memory-mapped I/O.

When standard I/O is used, microcontrollers normally use an IN or OUT instruction with 8-bit ports as follows:

```
IN      A, PORTA        ; Inputs 8-bit data from PORTA into the 8-bit
                        ; accumulator A.
OUT     PORTA,A         ; Outputs the contents of the 8-bit accumulator A
                        ; into PORTA
```

With memory-mapped I/O, the microcontroller normally uses an instruction (i.e., MOV as follows:

```
MOV     PORTA, reg      ; Inputs the contents of a port called "PORTA"
                        ; mapped as a memory location into a register.
MOV     reg,PORTA       ; outputs the contents of a register to a port called
                        ; "PORTA" mapped as a memory location.
```

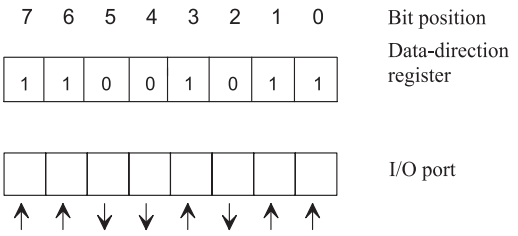


FIGURE 3.15 I/O port with the corresponding data-direction register**.**

3.2.4 Unconditional and Conditional Programmed I/O

There are typically two ways in which programmed I/O can be utilized: unconditional I/O and conditional I/O. The microcontroller can send data to an external device at any time using *unconditional Programmed I/O*. The external device must always be ready for data transfer. A typical example is that of a microcontroller outputting a 7-bit code through an I/O port to drive a seven-segment display connected to this port.

In *conditional Programmed I/O*, the microcontroller waits for a particular condition to occur, and then outputs data to an external device based on the condition. Conditional Programmed I/O is sometimes called *Polled I/O*.

As an example of conditional programmed I/O, consider Figure 3.16. Suppose that a comparator is connected to bit 0 of Port C , and an LED is connected to bit 1 of Port D of the PIC18F4321 microcontroller. It is desired to turn the LED ON when the comparator output becomes HIGH (Vx > Vy). In a situation like this, the microcontroller needs to wait in a loop until the condition "Vx > Vy" occurs. The microcontroller will send a HIGH to bit 1 of Port D as soon as the condition occurs.

Note that TRISC is the Data Direction Register (DDR) for Port C and TRISD is the Data Direction Register (DDR) for Port D. A '1' in a particular position will make the corresponding bit in each of these ports as an input while a '0' will make it an output. Also, note that the PIC18F4321 uses memory-mapped I/O. Hence, the following assembly language program starting at address 0x200 for the PIC18F4321 microcontroller will accomplish this:

```
        ORG     0x200
        SETF    TRISC       ; Make Port C as input by setting all bits of TRISC to 1's
        CLRF    TRISD       ; Make Port C as output by clearing all bits of TRISD to 0's
WAIT    MOVF    PORTC, W    ; Input comparator output into WREG
                            ; (ACCUMULATOR) via bit 0 of PORTC
        ANDLW   0x01        ; AND to check bit 0 (comparator output) of WREG is 1
        BZ      WAIT        ; Wait in loop if comparator output is 0 or Z flag is 1
        MOVLW   0x02        ; Move 1 to bit 1 of WREG (Accumulator) register
        MOVWF   PORTD       ; Turn the LED ON
        SLEEP               ; HALT
```

The PIC18F instructions used in the above program will now be explained. The "SETF TRISC" in the above program sets all bits in TRISC (DDR for PORTC) to 1's and thus, configures bit 0 of PORTC as an input bit. The " CLRF TRISD", on the other hand, clears all bits in TRISD (DDR for PORTD) to 0's and configures bit 1 of PORTD as an output bit. The "MOVF PORTC, W" instruction moves (inputs) the contents of PORTC into the WREG register (accumulator of the PIC18F). Thus, the comparator output connected to bit 0 of PORTC is inputted into bit 0 of the WREG register.

The "ANDLW 0x01 logically ANDs the contents of WREG with 0x01, and stores the result in WREG. The contents of WREG will be zero (Z=1) if the comparator output at bit 0 of PORTC is 0; the contents of WREG will be one (Z = 0) if the comparator output is 1. The "BZ WAIT" instruction checks the Z flag. If the Z flag is 1 (comparator output is 0), the program branches back to WAIT , and stays in the loop until the comparator output is 1. As soon as the comparator output is 1 (Z = 0), the "MOVLW 0x02" moves 0x02 into WREG, and thus, the bit 1 of WREG is a '1'. The MOVWF PORTD" instruction moves the contents of WREG to PORTD. Thus, a '1' is outputted to bit 1 of PORTC , and the LED is turned ON. The "SLEEP" instruction then halts the microcontroller.

Note that in the program, the PIC18F4321 has to wait in a loop indefinitely for the comparator output to become one (Vx > Vy). This is called "Conditional" or "Polled I/O", and is obviously inefficient because of the wait loop.

3.2.5 Interrupt I/O

As mentioned before, a disadvantage of conditional programmed I/O is that the CPU needs to check the status bit (output of the Comparator) by waiting in a loop. This type of I/O transfer is dependent on the occurrence of the external condition. This waiting may slow down the CPU's ability to process other data. The interrupt I/O technique is efficient in this type of situation.

Interrupt I/O is a device-initiated I/O transfer. The external device is connected to a pin called the *interrupt* (INT) pin on the microcontroller chip. When the device needs an I/O transfer with the microcontroller, it activates its interrupt pin. The microcontroller usually completes the current instruction and saves the contents of the current program counter and the status register onto the stack.

The microcontroller then loads an address into the program counter automatically to branch to a subroutine-like program called the *interrupt service routine*. This program is written by the user. The external device wants the microcontroller to execute this program to transfer data. The last instruction of the service routine is a RETURN, which is typically similar in concept to the RETURN instruction used at the end of a subroutine. The

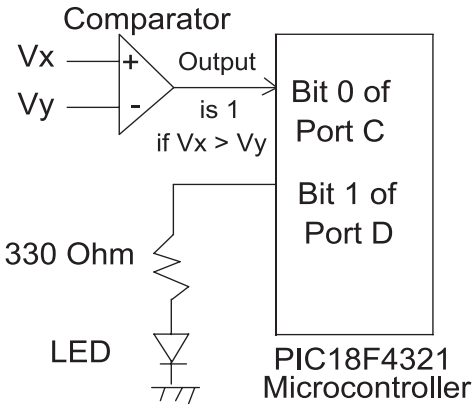


FIGURE 3.16 Example illustrating conditional or polled I/O

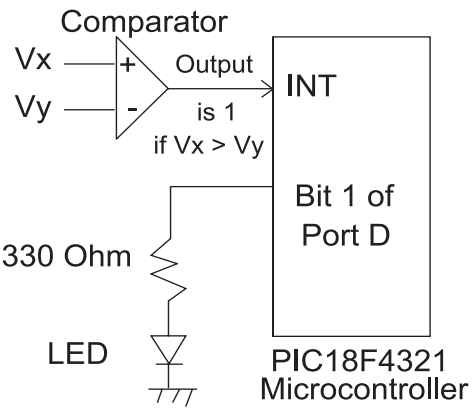


FIGURE 3.17 Example illustrating interrupt I/O

RETURN from interrupt instruction typically restores the program counter and the status register with the information saved in the stack before going to the service routine.

Figure 3.17 provides a simple example for illustrating the concept of interrupt I/O. This is the same example used to illustrate polled I/O of Figure 3.16 except that the comparator output is connected to the microcontroller's interrupt (INT) pin instead of bit 0 of Port C.

Assume that the PIC18F 4321 microcontroller is executing the following main program:

```
        ORG     0x200
        SETF    TRISC   ; Make Port C as input by setting all bits of TRISC to 1's
        CLRF    TRISD   ; Make Port C as output by clearing all bits of TRISD to 0's
        MOVLW   0x15
        MOVWF   STKPTR  ; Initialize STKPTR to 0x15
        MOVLW   3       ; Move  3 into WREG register
BEGIN   MOVWF   0x30    ; Move WREG into 0x30
        -
        -
        -
```

Note that the last two instructions MOVLW and MOVWF are chosen arbitrarily. In the above program, the SETF and CLRF instructions configure PORT C and PORT D of Figure 3.17. The "MOVLW 0x15" and "MOVWF STKPTR" initializes the PIC18F stack pointer (STKPTR) to 0x15. The value of the STKPTR is chosen arbitrarily.

Since interrupt I/O uses stack to save the return address, the stack pointer should be initialized in the main program. The PIC18F4321 then continues with execution of the "MOVLW 3" instruction. Suppose that during execution of the "MOVLW 3" instruction, the output of the comparator becomes HIGH, indicating that Vx is greater than Vy. This drives the INT signal to HIGH, interrupting the microcontroller. The microcontroller completes execution of the current instruction, "MOVLW 3". It then saves the current contents of the program counter (address BEGIN) automatically and executes a subroutine-like program called the *service routine*. This program is usually written by the user. The microcontroller manufacturer normally specifies the starting address of the service routine. This address is 0x000008 in the PIC18F. The user writes a service routine at this address to turn the LED ON, and then returns to the main program as follows:

```
ORG     0x000008    ;  Starting address of the service routine
MOVLW   0x02        ;  Move 1 to bit 1 of WREG (Accumulator) register
MOVWF   PORTD       ;  Turn the LED ON
RETFIE              ;  Restore PC and SR, and return from interrupt
```

In this service routine, using the MOVLW and MOVWF instructions, the microcontroller turns the LED ON. The return instruction RETFIE, at the end of the service routine, loads the address BEGIN from the stack into the program counter. The microcontroller executes the "MOVWF 0x30" instruction at the address BEGIN and continues with the main program. The basic characteristics of interrupt I/O have been discussed so far. The main features of interrupt I/O provided with a typical microcontroller are discussed next.

Interrupt Types There are typically two types of interrupts: external interrupts and internal interrupts. *External interrupts* are initiated through a microcontroller's interrupt pins by external devices such as the comparator in the previous example. External interrupts

can be divided further into two types: maskable and nonmaskable. The nonmaskable interrupt cannot be enabled or disabled by instructions, whereas a microcontroller's instruction set typically contains instructions to enable or disable maskable interrupt. A nonmaskable interrupt has a higher priority than a maskable interrupt. If maskable and nonmaskable interrupts are activated at the same time, the processor will service the nonmaskable interrupt first.

A nonmaskable interrupt is typically used as a power failure interrupt. Microcontrollers normally use +5 V dc, which is transformed from 110 V ac. If the power falls below 90 V ac, the DC voltage of +5 V cannot be maintained. However, it will take a few milliseconds before the ac power drops below 90 V ac. In these few milliseconds, the power-failure-sensing circuitry can interrupt the processor. The interrupt service routine can be written to store critical data in nonvolatile memory such as battery-backed CMOS RAM, and the interrupted program can continue without any loss of data when the power returns.The PIC18F does not have any nonmaskable interrupts.

Internal interrupts are maskable, and can be disabled by instructions. They are activated internally by conditions such as completion of A/D (Analog to Digital) conversion, Timer interrupt, or interrupt due to serial I/O. Interrupts are handled in the same way as external interrupts. The user writes a service routine to take appropriate action to handle the interrupt. Some microcontrollers include software interrupt instructions. When one of these instructions is executed, the microcontroller is interrupted and serviced similarly to external or internal interrupts.

Some microcontrollers such as the Motorola/Freescale HC11/HC12 provide both external (maskable and nonmaskable) and internal interrupts. The PIC18F provides external maskable interrupts only. The PIC18F does not have any external nonmaskable interrupts. However, the PIC18F provides internal interrupts. The internal interrupts are activated internally by conditions such as timer interrupts, completion of analog to digital conversion, and serial I/O.

Interrupt Address Vector The technique used to find the starting address of the service routine (commonly known as the *interrupt address vector*) varies from one processor to another. The microcontroller manufacturers typically define the fixed starting address for each interrupt.

Saving the Microcontroller Registers When a microcontroller is interrupted, it normally saves the program counter (PC) and the status register (SR) onto the stack so that the microcontroller can return to the main program with the original values of PC and SR after executing the service routine. The user should know the specific registers the microcontroller saves prior to executing the service routine. This will allow the user to use the appropriate return instruction at the end of the service routine to restore the original conditions upon return to the main program.

QUESTIONS AND PROBLEMS

3.1 What is the basic difference between main memory and secondary memory?

3.2 A microcontroller has 24 address pins. What is the maximum size of the main memory?

3.3 What is the basic difference between: (a) EPROM and EEPROM? (b) SRAM and DRAM?

3.4 What is flash memory?

3.5 Given a memory with a 14-bit address and an 8-bit word size:
- (a) How many bytes can be stored in this memory?
- (b) If this memory was constructed from 1k × 1 RAMs, how many memory chips would be required?
- (c) How many bits would be used for chip select?

3.6 Draw a block diagram showing the address and data lines for the 2732 and 2764 EPROM chips.

3.7
- (a) How many address and data lines are required for a 1M × 16 memory chip?
- (b) What is the size of a decoder with one chip enable ($\overline{CE}$) to obtain a 64k × 32 memory from 4k × 8 chips? Where are the inputs and outputs of the decoder connected?

3.8 A microcontroller with 24 address pins and eight data pins is connected to a 1k × 8 memory with one enable. How many unused address bits of the microcontroller are available for interfacing other 1k × 8 memory units? What is the maximum directly addressable memory available with this microcontroller?

3.9 Name the methods used in main memory array design. What are the advantages and disadvantages of each?

3.10 The block diagram of a 512 × 8 RAM is shown in Figure P3.10. In this arrangement the memory unit is enabled only when $\overline{CS1}$ = L and CS2 = H. Design a 1k × 8

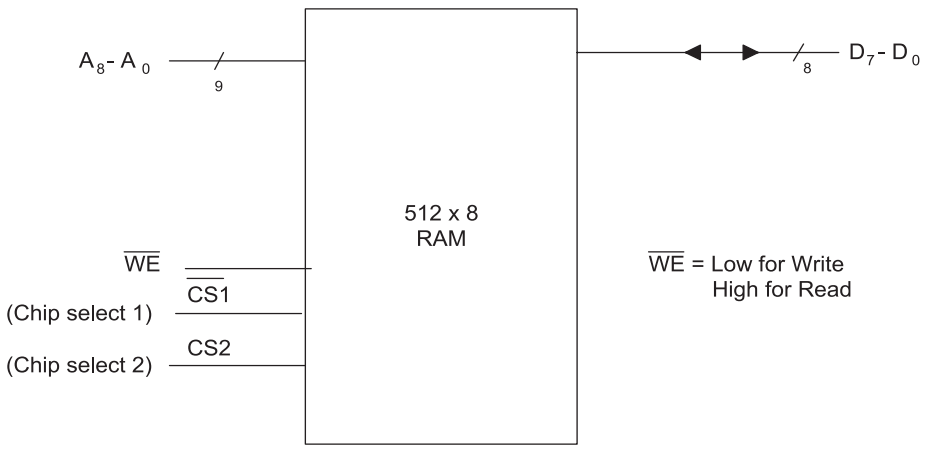


FIGURE P3.10

RAM system using the 512 × 8 RAM as the building block. Draw a neat logic diagram of your implementation. Assume that the CPU can directly address 64k with a R/$\overline{W}$ and eight data pins. Using linear decoding and don't-care conditions as 1's, determine the memory map in hexadecimal.

3.11 Consider the hardware schematic shown in Figure P3.11.

(a) Determine the address map of this system. *Note:* $\overline{MEMR}$ = 0 for read, $\overline{MEMR}$ = 1 for write, M/$\overline{IO}$ = 0 for I/O and M/ $\overline{IO}$ = 1 for memory.

(b) Is there any possibility of bus conflict in this organization? Clearly justify your answer.

3.12 Interface a CPU with 16-bit address pins and 8-bit data pins and a pin to a 1k × 8 EPROM and two 1k × 8 RAM's such that the memory map shown in Table P3.12 is obtained:
Assume that both EPROM and RAM contain two enable pins: $\overline{CE}$ and $\overline{OE}$ for the EPROM, $\overline{CE}$ and $\overline{WE}$ for each RAM. Note that $\overline{WE}$ = 1 and $\overline{WE}$ = 0 indicate read and write operations for the RAM chip. Use a decoder block identical to the 74138.

TABLE P3.12

Device	Size	Address Assignment (Hex)
EPROM	1k × 8	8000–83FF
RAM chip 0	1k × 8	9000–93FF
RAM chip 1	1k × 8	C000–C3FF

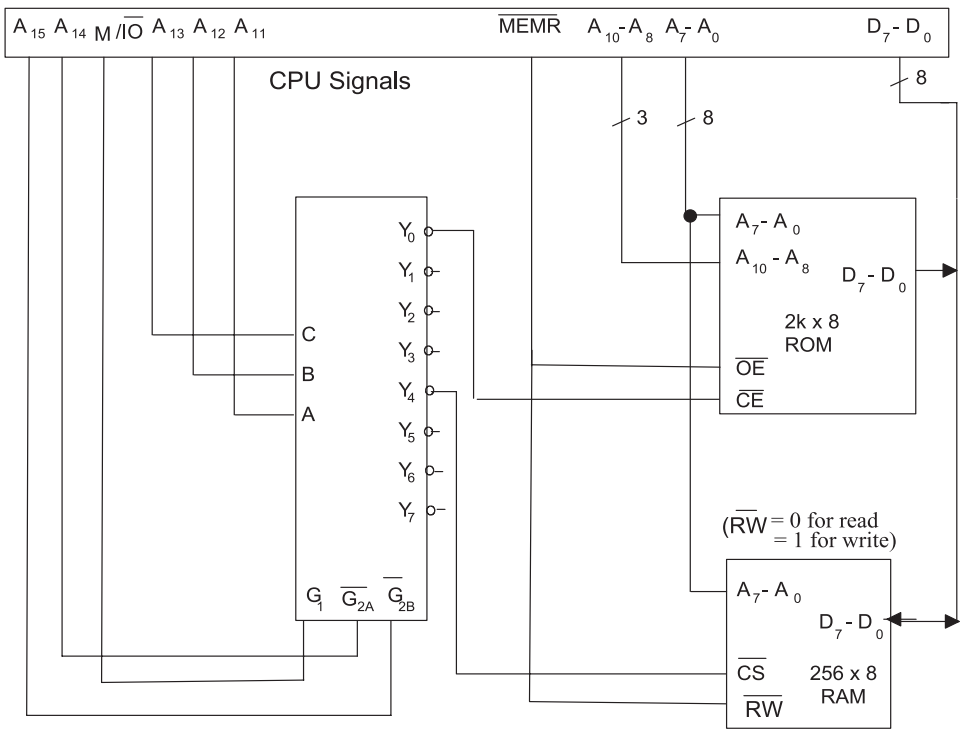


FIGURE P3.11

TABLE P3.13

Device	*Size*	*Address Assignment in hex*
EPROM	1k × 8	7000–73FF
RAM 0	1k × 8	D000–D3FF
RAM 1	1k × 8	F000–F3FF

3.13 Repeat Problem 3.12 to obtain the memory map shown in Table P3.13 using a decoder block identical to the 74138.

3.14 What is meant by *foldback* in linear decoding?

3.15 Define the three types of I/O. Identify each as either CPU-initiated or device-initiated.

3.16 What is the basic difference between standard I/O and memory-mapped I/O? Identify the programmed I/O technique used by the PIC18F.

3.17 What is the difference between memory mapping in a microcontroller and memory-mapped I/O?

3.18 Discuss the basic difference between polled I/O and interrupt I/O.

3.19 What is the difference between subroutine and interrupt I/O?

3.20 What is an interrupt address vector?

3.21 Summarize the basic difference between maskable and nonmaskable interrupts. Describe how power failure interrupt is normally handled.

3.22 Discuss the basic difference between internal and external interrupts.

4
PROGRAMMING LANGUAGES

In this chapter we provide the fundamental concepts associated with programming languages. Basic features of assembly and C language programming are discussed.

4.1 Computer Programming Languages

Computers are typically programmed using semi-English-language statements (assembly language). In addition to assembly languages, computers use a more understandable human-oriented language called *high-level language*. No matter what type of language is used to write programs, computers understand only binary numbers. Therefore, all programs must eventually be translated into their appropriate binary forms. The principal ways to accomplish this are discussed later.

Computer programming languages can typically be divided into three main types: machine language, assembly language, and high-level language. A *machine language program* consists of either binary or hexadecimal op-codes. Programming a computer with either one is relatively difficult, because one must deal only with numbers. The CPU architecture of the computer determines all its instructions. These instructions are called the computer's *instruction set*. Programs in *assembly* and *high-level languages* are represented by instructions that use English-language-type statements. The programmer finds it relatively more convenient to write programs in assembly or high-level language than in machine language. Note that high level language such as C is very popular for programming the microcontrollers. However, knowledge of programming the microcontrollers using assembly language is essential. This topic will be discussed later.

Since computers only understand binary numbers, a translator program must be used to convert programs written in High level or assembly into binary so that the computer can execute the programs. This is shown in Figure 4.1.

An *assembler* translates a program written in assembly language into a machine language program. A *compiler* or *interpreter*, on the other hand, converts a high-level language program such as C into a machine language program. Assembly or high-level language programs are called *source codes*. Machine language programs are known as *object codes*. A *translator* converts source codes to object codes. Next, we discuss the three types of programming languages relating them to microcontrollers in more detail.

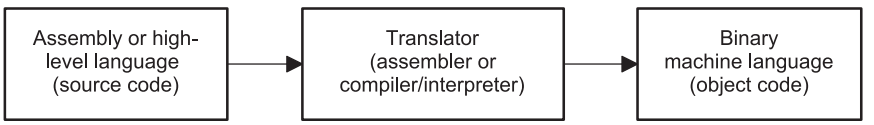


FIGURE 4.1 Translating assembly or high-level language into binary machine language.

Microcontroller Theory and Applications with the PIC18F, Second Edition. M. Rafiquzzaman.
© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

4.2 Machine Language

A microcontroller has a unique set of machine language instructions defined by its manufacturer. No two microcontrollers by two different manufacturers have the same machine language instruction set. For example, the Microchip Technology's PIC18F microcontroller uses the code $D7FF_{16}$ for the assembly language statement "HERE BRA HERE" (branch always to HERE), whereas the Motorola/Freescale HC11 microcontroller uses the code $20FE_{16}$ for the same statement with its BRA instruction. Therefore, a machine language program for one microcontroller will not run on the microcontroller of a different manufacturer.

At the most elementary level, a microcontroller program can be written using its instruction set in binary machine language. As an example, the following program adds two numbers using the PIC18F machine language:

0000111000000010
0000111100000011
0110111001000000
1110111100000011

Obviously, the program is very difficult to understand unless the programmer remembers all the PIC18F codes, which is impractical. Because one finds it very inconvenient to work with 0's and 1's, it is almost impossible to write an error-free program on the first try. Also, it is very tiring for a programmer to enter a machine language program written in binary into the microcontroller's RAM. For example, the programmer needs a number of binary switches to enter the binary program. This is definitely subject to errors.

To increase the programmer's efficiency in writing a machine language program, hexadecimal numbers rather than binary numbers are used. The following is the same addition program in hexadecimal using the PIC18F instruction set:

0E02
0F03
6E40
EF03

It is easier to detect an error in a hexadecimal program because each byte contains only two hexadecimal digits. One would enter a hexadecimal program using a hexadecimal keyboard. A keyboard monitor program in ROM provides interfacing of the hexadecimal keyboard with the microcontroller. This program converts each key actuation into binary machine language in order for the microcontroller to understand the program. However, programming in hexadecimal is not used.

4.3 Assembly Language

The next programming level is to use assembly language. Each line in an assembly language program includes four fields:

- Label field
- Instruction, mnemonic, or op-code field
- Operand field
- Comment field

As an example, a typical program for adding two 8-bit numbers written in PIC18F assembly language is as follows:

Label	Mnemonic	Operand	Comment
	MOVLW	1	; Move 1 into accumulator
	ADDLW	2	; Add 2 with 1, store result in accumulator
	SLEEP		; Halt

Obviously, programming in assembly language is more convenient than programming in machine language, because each mnemonic gives an idea of the type of operation it is supposed to perform. Therefore, with assembly language, the programmer does not have to find the numerical op-codes from a table of the instruction set, and programming efficiency is improved significantly.

An assembly language program is translated into binary via a program called an *assembler*. The assembler program reads each assembly instruction of a program as ASCII characters and translates them into the respective binary op-codes. For example, the PIC18F assembler translates the SLEEP (places the PIC18F in sleep mode; same as HALT instruction in other processors) instruction into its 16-bit binary op-code as 0000 0000 0000 0011 (0003 in hex), as depicted in Table 4.1.

An advantage of the assembler is address computation. Most programs use addresses within the program as data storage or as targets for jumps or calls. When programming in machine language, these addresses must be calculated by the programmer. The assembler solves this problem by allowing the programmer to assign a symbol to an address. The programmer may then reference that address elsewhere by using the symbol. The assembler computes the actual address for the programmer and fills it in automatically. One can obtain hands-on experience with a typical assembler for a microcontroller by downloading it from the Internet.

4.3.1 Types of Assemblers

Most assemblers use two passes to assemble a program. This means that they read the input program text twice. The first pass is used to compute the addresses of all labels in the program. To find the address of a label, it is necessary to know the total length of all the binary code preceding that label. Unfortunately, however, that address may be needed in that preceding code. Therefore, the first pass computes the addresses of all labels and stores them for the next pass, which generates the actual binary code. Various types of assemblers are available today:

- One-Pass Assembler. This assembler goes through an assembly language program once and translates it into a machine language program. This assembler has the problem of defining forward references. This means that a JUMP instruction using an address that appears later in the program must be defined by the programmer after the program is assembled.

TABLE 4.1 Conversion of 0PIC18F SLEEP instruction into its Binary Op-Code

Assembly Code	Binary Form of ASCII Codes as seen by the Assembler	Binary Op-Code Created by the MPLAB PIC18F Assembler
S	0101 0011	
L	0100 1000	0000 0000 0000 0011
E	0100 0101	
E	0100 0101	
P	0101 0000	

- Two-Pass Assembler. This assembler scans an assembly language program twice. In the first pass, this assembler creates a symbol table. A symbol table consists of labels with addresses assigned to them. This way, labels can be used for JUMP statements and no address calculation has to be done by the user. On the second pass, the assembler translates the assembly language program into machine code. The two-pass assembler is more desirable and much easier to use.

  Note that MPLAB PIC18F assembler is a two-pass assembler.
- Macroassembler. This type of assembler translates a program written in macro language into machine language. This assembler lets the programmer define all instruction sequences using macros. Note that by using macros, the programmer can assign a name to an instruction sequence that appears repeatedly in a program. The programmer can thus avoid writing an instruction sequence that is required many times in a program by using macros. The macroassembler replaces a macroname with the appropriate instruction sequence each time it encounters a macroname.

  It is interesting to see the difference between a subroutine and a macroprogram. A specific subroutine occurs once in a program. A subroutine is executed by CALLing it from a main program. The program execution jumps out of the main program and executes the subroutine. At the end of the subroutine, a RET instruction is used to resume program execution following the CALL SUBROUTINE instruction in the main program. A macro, on the other hand, does not cause the program execution to branch out of the main program. Each time a macro occurs, it is replaced by the appropriate instruction sequence in the main program. Typical advantages of using macros are shorter source programs and better program documentation. A typical disadvantage is that effects on registers and flags may not be obvious.

  Conditional macroassembly is very useful in determining whether or not an instruction sequence is to be included in the assembly, depending on a condition that is true or false. If two different programs are to be executed repeatedly based on a condition that can be either true or false, it is convenient to use conditional macros. Based on each condition, a particular program is assembled. Each condition and the appropriate program are typically included within IF and ENDIF pseudoinstructions.
- Cross assembler. This type of assembler is typically resident in a processor and assembles programs for another for which it is written. The cross assembler program is written in a high-level language so that it can run on different types of processors that understand the same high-level language.
- Resident assembler. This type of assembler assembles programs for a processor in which it is resident. The resident assembler may slow down operation of the processor on which it runs.
- Meta-assembler. This type of assembler can assemble programs for many different types of processors. The programmer usually defines the particular processor being used.

4.3.2 Assembler Delimiters

As mentioned before, each line of an assembly language program consists of four fields: label, mnemonic or op-code, operand, and comment. The assembler ignores the comment field but translates the other fields. The label field must start with an uppercase alphabetic character. The assembler must know where one field starts and another ends.

Most assemblers allow the programmer to use a special symbol or delimiter to indicate the beginning or end of each field. Typical delimiters used are spaces, commas, semicolons, and colons:

- Spaces are used between fields.
- Commas (,) are used between addresses in an operand field.
- A semicolon (;) is used before a comment.
- A colon (:) or no delimiter is used after a label.

4.3.3 Specifying Numbers by Typical Assemblers

To handle numbers, most assemblers consider all numbers as decimal numbers unless specified otherwise. All assemblers will also specify other number systems, including hexadecimal numbers. The user must define the type of number system used in some way. This is generally done by using a letter or a symbol before or after the number. For example, Intel uses the letter H after a number to represent it as a hex number, whereas Microchip's MPLAB assembler uses several ways to represent a hex number. Three most common ways are: 0x before the number, or H after the number, or default (without using a letter or a symbol before or after the number), to represent it as a hex number. As an example, 60 in hexadecimal is represented by the MPLAB assembler as either 0x60 or 60H, or simply 60.

Some assemblers such as the MASM 32 assembler for the Pentium microprocessor require hexadecimal numbers to start with a digit (0 through 9). A 0 is typically used if the first digit of the hexadecimal number is a letter. This is done to distinguish between numbers and labels. For example, typical assemblers such as MASM32 will normally require the number F3H to be represented as 0F3H; otherwise, the assembler will generate an error. However, MPLAB assembler used in this book for assembling PIC18F assembly language programs does not require '0' to be used if the first digit of a hexadecimal number is a letter.

The MPLAB uses D before a 'number' to specify a decimal number. For example, decimal number 60 can be represented as D'60'. A binary number is specified by the MPLAB using B before the 'Number'. For example, 8-bit binary number 01011100 can be represented by the MPLAB as B'01011100'.

4.3.4 Assembler Directives or Pseudoinstructions

Assemblers use pseudoinstructions or directives to make the formatting of the edited text easier. Pseudoinstructions are not translated directly into machine language instructions. They equate labels to addresses, assign the program to certain areas of memory, or insert titles, page numbers, and so on. To use the assembler directives or pseudoinstructions, the programmer puts them in the op-code field, and if the pseudoinstructions require an address or data, the programmer places them in the label or data field. Typical pseudoinstructions are ORIGIN (ORG), EQUATE (EQU), DEFINE BYTE (DB), and DEFINE WORD (DW).

ORIGIN (ORG) The directive ORG lets a programmer place programs anywhere in memory. Internally, the assembler maintains a program counter type of register called an *address counter*. This counter maintains the address of the next instruction or data to be processed.

An ORG directive is similar in concept to a JUMP instruction. Note that the JUMP

instruction causes a processor to place a new address in the program counter. Similarly, the ORG pseudoinstruction causes the assembler to place a new value in the address counter.

Typical ORG statements are

```
ORG  0x100
SLEEP
```

The MPLAB assembler will generate the following code for these statements:

```
100 0003
```

Most assemblers assign a value of zero to the starting address of a program if the programmer does not define this by means of an ORG.

Equate (EQU) The directive EQU assigns a value in its operand field to an address in its label field. This allows the user to assign a numerical value to a symbolic name. The user can then use the symbolic name in the program instead of its numeric value. This reduces errors.

A typical example of EQU is START EQU 0x0200, which assigns the value 0200 in hexadecimal to the label START.

Note that if a label in the operand field is equated to another label in the label field, the label in the operand field must have been defined previously. For example, the EQU statement

```
BEGIN  EQU    START
```

will generate an error unless START is defined previously with a numeric value.

Define Byte (DB) The directive DB (some assemblers use DC.B) is generally used to set a memory location in the program memory to a certain byte value. For example,

```
ORG    0x100
DB     0x45
```

will store the data value 45 hex in address 0x100. With some assemblers, the DB pseudoinstruction can be used to generate a table of data as follows:

```
ORG    0x200
DB     0x20, 0x30, 0x40, 0x50
```

In this case, 20 hex is the first data of the memory location 0x200; 30 hex, 40 hex, and 50 hex occupy the next three memory locations. Therefore, the data in memory will look like this:

```
200    20
201    30
202    40
203    50
```

Define Word (DW) The directive DW (some assemblers use DC.W) is typically used to assign a 16-bit value to two memory locations in the program memory. For example,

```
ORG    0x100
DW     0x4AC2
```

will assign C2 to location 100 and 4A to location 101. It is assumed that the assembler will follow little endian; that is, it will assign the low byte first (C2) and then the high byte (4A). With some assemblers, the DW directive can be used to generate a table of 16-bit data as follows:

```
ORG    0x80
DW     0x5000, 0x6000, 0x7000
```

In this case, the three 16-bit values 0x5000, 0x6000, and 0x7000 are assigned to memory

locations starting at the address 0x80. That is, the array would look like this:

80	00
81	50
82	00
83	60
84	00
85	70

END This directive indicates the end of the assembly language source program.

4.3.5 Assembly Language Instruction Formats

In this section, assembly language instruction formats available with typical microcontrollers are discussed. Depending on the number of addresses specified, the following instruction formats can be used: three-address, two-address, one-address, zero-address. Because all instructions are stored in the main memory, instruction formats are designed in such a way that instructions take less space and have more processing capabilities. It should be emphasized that the microcontroller architecture has considerable influence on a specific instruction format. The following are some important technical points that have to be considered while designing an instruction format:

- The size of an instruction word is chosen such that it facilitates the specification of more operations by a designer. For example, with 4- and 8-bit op-code fields, we can specify 16 and 256 distinct operations, respectively.
- Instructions are used to manipulate various data elements, such as integers, and character strings. In particular, all programs written in a symbolic language such as C are stored internally as characters. Therefore, memory space will not be wasted if the word length of the machine is some integral multiple of the number of bits needed to represent a character. Because all characters are represented using typical 8-bit character codes such as ASCII , it is desirable to have 8-, or 16-bit words for the word length for typical microcontrollers.
- The size of the address field is chosen such that high resolution is guaranteed. Note that in any microcontroller, the ultimate resolution is a bit. Memory resolution is a function of the instruction length, and in particular, short instructions provide less resolution. For example, in a microcontroller with 32K 16-bit memory words, at least 19 bits are required to access each bit of the word. (This is because $2^{15} = 32K$ and $2^4 = 16$.)

The general form of a *three-address instruction* is

<op-code> Addr1,Addr2,Addr3

Some typical *three-address instructions* are

```
MUL   A,B,C      ;    C ← A * B
ADD   A,B,C      ;    C ← A + B
SUB   R1,R2,R3   ;    R3 ← R1 - R2
```

In this specification, all alphabetic characters are assumed to represent memory addresses, and the string that begins with the letter R indicates a register. The third address of this type of instruction is usually referred to as the *destination address*. The result of an operation is always assumed to be saved in the destination address.

Typical programs can be written using three-address instructions. For example, consider the following sequence of three-address instructions:

```
MUL    A, B, R1        ;       R1 ← A * B
MUL    C, D, R2        ;       R2 ← C * D
MUL    E, F, R3        ;       R3 ← E * F
ADD    R1,R2,R1        ;       R1 ← R1 + R2
SUB    R1,R3,Z         ;       Z  ← R1 - R3
```

This sequence implements the statement Z = A * B + C * D - E * F. The three-address format, in addition to the other formats, is normally used by typical microcontrollers such as the PIC18F.

If we drop the third address from the three-address format, we obtain the two-address format, whose general form is

<op-code> Addr1,Addr2

Some typical *two-address instructions* are

```
MOV    A,R1            ;       R1 ← A
ADD    C,R2            ;       R2 ← R2 + C
SUB    R1,R2           ;       R2 ← R2 - R1
```

In this format, the addresses Addr1 and Addr2 represent source and destination addresses, respectively.

The following sequence of two-address instructions is equivalent to the program using three-address format presented earlier:

```
MOV    A,R1            ;       R1 ← A
MUL    B,R1            ;       R1 ← R1 * B
MOV    C,R2            ;       R2 ← C
MUL    D,R2            ;       R2 ← R2 * D
MOV    E,R3            ;       R3 ← E
MUL    F,R3            ;       R3 ← R3 * F
ADD    R2,R1           ;       R1 ← R1 + R2
SUB    R3,R1           ;       R1 ← R1 - R3
MOV    R1,Z            ;       Z  ← R1
```

Some typical *one-address instructions* are

```
LDA    B       ;       Acc ← B
ADD    C       ;       Acc ← Acc + C
MUL    D       ;       Acc ← Acc * D
STA    E       ;       E   ← Acc
```

Note that Acc means the accumulator.

The following program illustrates how we can translate the C language statement, z = (a * b) + (c * d) - (e * f); into a sequence of one-address instructions:

```
lda    e       ;       Acc ← e
mul    f       ;       Acc ← e * f
sta    t1      ;       t1 ← Acc
lda    c       ;       Acc ← c
mul    d       ;       Acc ← c * d
sta    t2      ;       t2 ← Acc
lda    a       ;       Acc ← a
mul    b       ;       Acc ← a * b
add    t2      ;       Acc ← Acc + t2
sub    t1      ;       Acc ← Acc - t1
sta    z       ;       Z ← Acc
```

In this program, t1 and t2 represent the addresses of memory locations used to store temporary results. Instructions that do not require any addresses are called *zero-*

address instructions. All microcontrollers include some zero-address instructions in the instruction set. Typical examples of zero-address instructions are CLC (clear carry) and NOP (No operation) .

4.3.6 Typical Instruction Set

An instruction set of a specific microcontroller consists of all the instructions that it can execute. The capabilities of a microcontroller are determined to some extent by the types of instructions it is able to perform. Each microcontroller has a unique instruction set designed by its manufacturer to do a specific task. We discuss some of the instructions that are common to all microcontrollers. We group together chunks of these instructions which have similar functions. These instructions typically include:

- **Arithmetic Instructions** Typical arithmetic instructions include ADD, SUBTRACT, COMPARE, MULTIPLY, and DIVIDE. Note that the SUBTRACT instruction provides the result and also affects the status flags, whereas the COMPARE instruction performs subtraction without any result and affects the flags based on the result.

Typical microcontrollers utilize common hardware to perform addition and subtraction operations for both unsigned and signed numbers. The instruction set for a microprocessor typically includes the same ADD and SUBTRACT instructions for both unsigned and signed numbers. The interpretations of unsigned and signed ADD and SUBTRACT operations are performed by the programmer. For example, consider adding two 8-bit numbers, A and B ($A = FF_{16}$ and $B = FF_{16}$), using the ADD instruction by a microprocessor as follows:

$$
\begin{array}{rl}
 & 1111111 \leftarrow \text{Intermediate carries} \\
 & FF_{16} = 11111111 \\
+ & FF_{16} = 11111111 \\
\hline
\text{final carry} & \rightarrow 111111110 = FE_{16}
\end{array}
$$

When the addition above is interpreted by the programmer as an unsigned operation, the result will be $A + B = FF_{16} + FF_{16} = 255_{10} + 255_{10} = 510_{10}$ which is FE_{16} with a carry, as shown above. However, if the addition is interpreted as a signed operation, then $A + B = FF_{16} + FF_{16} = (-1_{10}) + (-1_{10}) = -2_{10}$ which is FE_{16} as shown above, and the final carry must be discarded by the programmer. Similarly, the unsigned and signed subtraction can be interpreted by the programmer.

The unsigned and signed multiplication and division algorithms will be discussed in the following.

Unsigned Multiplication Several unsigned multiplication algorithms are available. Multiplication of two unsigned numbers can be accomplished via repeated addition. For example, to multiply 4_{10} by 3_{10}, the number 4_{10} can be added twice to itself to obtain the result, 12_{10}.

Signed Multiplication Signed multiplication can be performed using various algorithms. A simple algorithm follows. Assume that M (multiplicand) and Q (multiplier) are in two's-complement form. Assume that M_n and Q_n are the most significant bits (sign bits) of the multiplicand (M) and the multiplier (Q), respectively The sign bit of the product is determined as $M_n \oplus Q_n$, where M_n and Q_n are the most significant bits (sign bits) of the multiplicand (M) and the multiplier (Q), respectively. To perform signed multiplication, proceed as follows:

1. If $M_n = 1$, compute the two's complement of M.
2. If $Q_n = 1$, compute the two's complement of Q.
3. Multiply the n - 1 bits of the multiplier and the multiplicand using unsigned multiplication.
4. The sign of the result, $S_n = M_n \oplus Q_n$.
5. If $S_n = 1$, compute the two's-complement of the result obtained in step 3.

Next, consider a numerical example. Assume that M and Q are two's-complement numbers. Suppose that $M = 1100_2$ and $Q = 0111_2$. Because $M_n = 1$, take the two's-complement of $M = 0100_2$; because $Q_n = 0$, do not change Q. Multiply 0111_2 and 0100_2 using the unsigned multiplication method discussed before. The product is 00011100_2. The sign of the product $S_n = M_n \oplus Q_n = 1 \oplus 0 = 1$. Hence, take the two's-complement of the product 00011100_2 to obtain 11100100_2, which is the final answer: -28_{10}.

Unsigned Division Unsigned division can be accomplished via repeated subtraction. For example, consider dividing 7_{10} by 3_{10} as follows:

Dividend	Divisor	Subtraction Result	Counter
7_{10}	3_{10}	$7 - 3 = 4$	1
		$4 - 3 = 1$	$1 + 1 = 2$

Quotient = counter value = 2

Remainder = subtraction result = 1

Here, 1 is added to a counter whenever the subtraction result is greater than the divisor. The result is obtained as soon as the subtraction result is smaller than the divisor

Signed Division Signed division can be performed using various algorithms. A simple algorithm follows. Assume that DV (Dividend) and DR *(*Divisor) are in two's-complement form. For the first case, perform unsigned division using repeated subtraction of the magnitudes without the sign bits. The sign bit of the quotient is determined as $DV_n \oplus DR_n$, where DV_n and DR_n are the most significant bits (sign bits) of the dividend (DV) and the divisor (DR) respectively. To perform signed division, proceed as follows:

1. If $DV_n = 1$, compute the two's complement of DV, else keep DV unchanged.
2. If $DR_n = 1$, compute the two's complement of DR, else keep DR unchanged.
3. Divide the $n - 1$ bits of the dividend by the divisor using unsigned division algorithm (repeated subtraction).
4. The sign of the Quotient $Q_n = DV_n \oplus DR_n$. The sign of the remainder is the same as the sign of the dividend unless the remainder is zero. The following numerical examples illustrate this:
   The general equation for division can be used for signed division. Note that the general equation for division is *dividend = quotient * divisor + remainder*. For example, consider dividend = – 9, divisor = 2. Three possible solutions are shown below:
   (a) – 9 = – 4 * 2 – 1, Quotient = – 4, Remainder = – 1.
   (b) – 9 = – 5 * 2 + 1, Quotient = – 5, Remainder = +1.
   (c) – 9 = – 6 * 2 + 3, Quotient = – 6, Remainder = +3.
   However, the correct answer is shown in (a), in which, the Quotient = – 4 and the remainder = – 1. Hence, for signed division, the sign of the remainder is the same as the sign of the dividend, unless the remainder is zero.
5. If $Q_n = 1$, compute the two's-complement of the quotient obtained in step 3, else keep the quotient unchanged.

The above algorithm will be verified using numerical examples provided in the following:

i) Signed division with zero remainder

Assume 4-bit numbers
Dividend = +6 = 0110_2 Divisor = -2 = Two's complement of 2 = 1110_2

Since the sign bit of Dividend is 0, do not change the dividend. Because the sign bit of the divisor is 1, take 2's complement of 1110 which is 0010. Now, divide 0110 by 0010 using repeated subtraction as follows:

DIVIDEND	DIVISOR	SUBTRACTION RESULT USING 2'S COMPLEMENT	COUNTER
0110	0010	0110-0010=0100	0001
		0100-0010=0010	0010
		0010-0010=0000	0011

Result of unsigned division: Quotient = Counter value = 0011_2,
Remainder = Subtraction result = 0000_2
Result of signed division 6 (0110) divided by -2 (1110):
Sign of the quotient = (Sign of dividend)⊕(Sign of divisor) = 0 ⊕ 1= 1
Hence, Quotient = 2's complement of 0011_2= 1101_2= -3_{10} , Remainder = 0000_2

ii) Signed division with nonzero remainder

Assume 4-bit numbers

Dividend = -5 = Two's complement of 0101_2 = 1011_2 Divisor = -2 = Two's complement of 2 = 1110_2

Since the sign bit of Dividend is 1, take 2's complement of 1011 which is 0101. Because the sign bit of the divisor is 1, take 2's complement of 1110 which is 0010. Now, divide 0101 by 0010 using repeated subtraction as follows:

DIVIDEND	DIVISOR	SUBTRACTION RESULT USING 2'S COMPLEMENT	COUNTER
0101	0010	0101-0010=0011	0001
		0011-0010=0001	0010

Result of unsigned division**:** Quotient = Counter value = 0010_2,
Remainder = Subtraction result = 0001_2
Result of signed division -5 (1011) divided by -2 (1110):
Sign of the quotient = (Sign of dividend) ⊕ (Sign of divisor) = 1 ⊕ 1= 0. Hence, do not take two's complement of Quotient.
Quotient =0010_2 = $+2_{10}$, remainder has the same sign as the dividend which is negative (bit 3 = 1).
Hence, Remainder = 2's complement of 0001_2 = 1111_2= -1_{10}.

Note that the Sign of the quotient = (Sign of dividend) ⊕ (Sign of divisor). However, the sign of the remainder is the same as the sign of the dividend unless the remainder is 0. This can be verified by the following numerical examples using decimal numbers:

Case 1: when the remainder is 0

i) Assume both the dividend and divisor are positive
Dividend = +6 Divisor = +2
Result: Quotient = +3 Remainder = 0

ii) Assume the dividend is negative and the divisor is positive.
Dividend = -6 Divisor = +2
Result: Quotient = -3 Remainder = 0

iii) Assume the dividend is positive and the divisor is negative.
Dividend = +6 Divisor = -2
Result: Quotient = -3 Remainder = 0

iv) Assume both the dividend and divisor are negative.
Dividend = -6 Divisor = -2
Result: Quotient = +3 Remainder = 0

Case 2: when the remainder is nonzero

Since, Dividend = Quotient x Divisor + Remainder
Hence, Remainder = Dividend - Quotient x Divisor.

i) Assume both the dividend and divisor are positive.
Dividend = +5 Divisor = +2
Result: Quotient = +2, Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = +5 - (+2 x +2) = +1.

ii) Assume the dividend is negative and the divisor is positive.
Dividend = -5 Divisor = +2
Result: Quotient = -2 Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = -5 - (-2 x +2) = - 1.

iii) Assume the dividend is positive and the divisor is negative.
Dividend = +5 Divisor = -2
Result: Quotient = -2, Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = +5 - (-2 x --2) = + 1.

iv) Assume both dividend and divisor are negative.
Dividend = -5 Divisor = -2
Result: Quotient = +2, Remainder can be obtained from the equation, Remainder = Dividend - Quotient x Divisor. Hence, Remainder = -5 - (+2 x -2) = - 1.
From the above, the sign of the remainder is the same as the sign of the dividend unless the remainder is zero.

RISC microcontrollers such as the PIC18F include the unsigned multiplication instruction. The PIC18F instruction set does not contain instructions for signed multiplication, and unsigned and signed division. However, subroutines using PIC18F assembly language can be written to obtain them using the above algorithms.

- Logic Instructions. Typical logic instructions perform traditional Boolean operations such as AND, OR, and Exclusive-OR. The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in a word, the word can be logically ANDed with appropriate data to accomplish this. For

example, the bit value at bit 2 of an 8-bit number 0100 1Y10 (where an unknown bit value of Y is to be determined) can be obtained as follows:

```
          0 1 0  0  1 Y 1 0  -- 8-bit number
AND       0 0 0  0  0 1 0  0 -- masking  data
          ---------------------
          0 0 0  0  0 Y 0 0 -- result
```

If the bit value Y at bit 2 is 1, the result is nonzero (flag Z = 0); otherwise, the result is zero (Flag Z = 1) **.** The Z flag can be tested using typical conditional JUMP instructions such as JZ (Jump if Z=1) or JNZ (Jump if Z = 0) to determine whether Y is 0 or 1. This is called a *masking operation*. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the least significant bit (LSB) of the number (LSB = 0 for even and LSB = 1 for odd). The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits. For example, a 1 can be inserted using the OR instruction at bit 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits:

```
          0 1 1 1 0 0 1 1 -- 8-bit  number
OR        0 0 0 0 1 0 0 0 -- data for inserting a 1 at  bit 3
          -------------------
          0 1 1 1 1 0 1 1 -- result
```

The Exclusive-OR instruction can be used to find the one's-complement of a binary number by XORing the number with all 1's as follows:

```
          0 1 0 1 1 1 0 0 --  8-bit number
XOR       1 1 1 1 1 1 1 1 --  data
          --------------------------
          1 0 10 0 0 1 1 -- Result ( One's Complement  of the 8-bit
                                number  0 1 0 1 1 1 0 0)
```

- Shift and Rotate Instructions. Next, the concept of logic and arithmetic shift and rotate operations is reviewed. In a logical shift operation, a bit that is shifted out will be lost, and the vacant position will be filled with a 0. For example, if we have the number $(11)_{10}$, after a logical right shift operation, the register contents shown in Figure 4.2 will occur. Typical examples of logic/arithmetic and shift/rotate operations are given in Table 4.2.

It must be emphasized that a logical left or right shift of an unsigned number by n positions implies multiplication or division of the number by 2^n, respectively, provided that a 1 is not shifted out during the operation.

In the case of true arithmetic left or right shift operations, the sign bit of the number to be shifted must be retained. However, in computers, this is true for right shift and not for left shift operation. For example, if a register is shifted right arithmetically, the most significant bit (MSB) of the register is preserved, thus ensuring that the sign of the number will remain unchanged. This is illustrated in Figure 4.3.

There is no difference between arithmetic and logical left shift operations. If the most significant bit changes from 0 to 1 or vice versa in an arithmetic left shift, the result is incorrect and the CPU sets the overflow flag to 1. For example, if the original value of the register is $(+3_{10})$, the results of two successive arithmetic left shift operations are

interpreted as follows:

Original	*After first shift*	*After second shift*
$0011_2 = (+3_{10})$	$0110_2 = (+6_{10})$	$1100_2 = (-4_{10})$; the result is incorrect
	$3 \times 2 = 6$; correct	$6 \times 2 = 12$ not -4; incorrect

The positive number $(+3_{10})$ is shifted arithmetically twice to the left generating a negative result (-4_{10}). Hence the overflow flag will be set to one.

- Instructions for controlling microcontroller operations. These instructions typically include those that set the reset specific flags and halt or stop the CPU.
- Data movement instructions. These instructions move data from a register to memory, and vice versa, between registers and between a register and an I/O device.
- Instructions using memory addresses. An instruction in this category typically contains a memory address, which is used to read a data word from memory into a microcontroller register or for writing data from a register into a memory location. Many instructions under data processing and movement fall in this category.
- Conditional and unconditional JUMP. These instructions typically include one of the

TABLE 4.2 Typical Logic/Arithmetic and Shift/ Rotate Operations

Shift type	Logic	Arithmetic	Rotate
Right	0 → Lost	MSB → Lost	
Left	0 → Lost	0 → Lost	

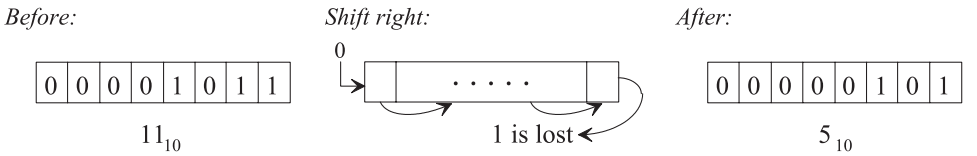


FIGURE 4.2 Logical right shift operation.

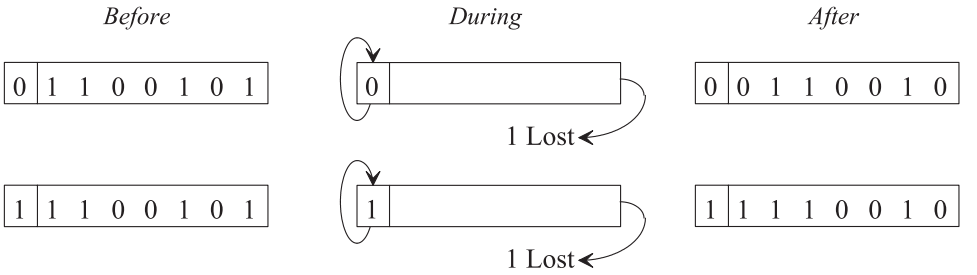


FIGURE 4.3 True arithmetic right shift operation.

following:

1. An unconditional JUMP, which always transfers the memory address specified in the instruction into the program counter.
2. A conditional JUMP, which transfers the address portion of the instruction into the program counter based on the conditions set by one of the status flags in the flag register.

4.3.7 Typical Addressing Modes

One of the tasks performed by a microcontroller during execution of an instruction is the determination of the operand and destination addresses. The manner in which a microcontroller accomplishes this task is called the "addressing mode". Now, let us present the typical microcontroller addressing modes, relating them to the instruction set of the PIC18F.

An instruction is said to have "implied or inherent addressing mode" if it does not have any operand. For example, consider the following instruction: SLEEP, which is equivalent to the HALT instruction in other microcontrollers. The SLEEP instruction is a no-operand instruction.

Whenever an instruction/operand contains data, it is called an "immediate mode" instruction. For example, consider the following PIC18F instruction:

```
ADDLW  3        ;  [WREG] ← [WREG] + 3
```

Note that the accumulator in the PIC18F is called WREG register. This instruction adds 3 to the contents of the WREG and then stores the result in WREG.

An instruction is said to have an *absolute* or *direct addressing mode* if it contains a memory address in the operand field. For example, consider the PIC18F instruction:

```
MOVWF  0x20   ;  [0x20] ← [WREG]
```

The MOVWF 0x20 instruction moves the contents of the WREG register into a memory location whose address is 0x20. The contents of WREG are unchanged. MOVWF 0x20 uses direct address mode since address 0x20 is directly specified in the MOVWF instruction.

When an instruction specifies a microcontroller register to hold the address, the resulting addressing mode is known as the *register indirect mode*. For example, consider the PIC18F instruction:

```
MOVWF        INDF0   ; Move contents of WREG into a data RAM
                     ; address pointed  to by FSR0 since INDF0
                     ; is associated with FSR0
```

The above instruction moves the 8-bit contents of WREG to a data memory location whose address is in PIC18F's FSR0 register. This instruction uses the contents of FSR0 register as a pointer to data memory. Also, INDF0 in the instruction MOVWF INDF0 means that FSR0 register will hold the address of data memory.

Conditional branch instructions are used to change the order of execution of a program based on the conditions set by the status flags. Some microcontrollers use conditional branching using the absolute mode. The op-code verifies a condition set by a particular status flag. If the condition is satisfied, the program counter is changed to the value of the operand address (defined in the instruction). If the condition is not satisfied, the program counter is incremented, and the program is executed in its normal order.

Typical microcontrollers such as the PIC18F use conditional branch instructions. Some conditional branch instructions are 16 bits wide. The first byte is the op-code for checking a particular flag. The second byte is an 8-bit offset, which is added to the contents of the program counter if the condition is satisfied to determine the effective address. This offset is considered as a signed binary number with the most significant bit as the sign bit. It means that the offset can vary from -128_{10} to $+127_{10}$ (0 being positive). This is called the *relative mode*.

This means that for forward branching, the range of the offset value is from 0x00 to 0x7F. For backward branching, this range varies from 0x80 to 0xFF. Since conditional branch instructions are 16-bit wide in the PIC18F, the PC (Program Counter) is incremented by 2 to point to the next instruction while executing the conditional branch instruction. The offset is multiplied by 2 and then added to PC+2 to find the branch address if the condition is true. Note that the offset is multiplied by 2 since the contents of the PC must always be an even number for 16- and 32-bit instruction lengths.

As an example, consider BNZ 0x03. Note that BNZ stands for "Branch if not zero". If the Z (zero flag) in the Status register is 0, then the PC is loaded with the address computed from (PC + 2 + 03H x 2). When the PIC18F executes the BNZ instruction, the PC points to the next instruction. This means that if BNC is located at address 0050H in program memory, the PC will contain 0052H (PC + 2) when the PIC18F executes BNZ. Hence, if Z = 0, then after execution of the BNZ 0x03 instruction, the PC will be loaded with address 0058H (0052H + 03H x 2). Hence, the program will branch to address 0058H which is 6 steps forward relative to the current contents of PC. This is called "Relative Addressing Mode". Note that the Relative mode is useful for developing position independent code.

4.3.8 Subroutine Calls in Assembly Language

It is sometimes desirable to execute a common task many times in a program. Consider the case when the sum of squares of numbers is required several times in a program. One could write a sequence of instructions in the main program for carrying out the sum of squares every time it is required. This is all right for short programs. For long programs, however, it is convenient for the programmer to write a small program known as a *subroutine* for performing the sum of squares, and call this program each time it is needed in the main program. Therefore, a subroutine can be defined as a program carrying out a particular function that can be called by another program, known as the *main program*. The subroutine only needs to be placed once in memory starting at a particular memory location. Each time the main program requires this subroutine, it can branch to it, typically by using PIC18F's CALL to subroutine (CALL) instruction along with its starting address. The subroutine is then executed. At the end of the subroutine, PIC18F's RETURN from subroutine instruction takes control back to the main program.

4.4 High-Level Language

As mentioned earlier, a programmer's efficiency increases significantly with assembly language compared to machine language. However, the programmer needs to be well acquainted with the CPU architecture and its instruction set. Furthermore, the programmer has to provide an op-code for each operation that the CPU has to carry out in order to execute a program. As an example, for adding two numbers, the programmer would instruct the CPU to load the first number into a register, add the second number to the register, and

then store the result in memory. However, the programmer might find it tedious to write all the steps required for a large program. Also, to become a reasonably good assembly language programmer, one needs to have a lot of experience.

High-level language programs comprised of English-language-type statements rectify all these deficiencies of machine and assembly language programming. The programmer does not need to be familiar with the internal microcontroller structure or its instruction set. Also, each statement in a high-level language corresponds to a number of assembly or machine language instructions. For example, consider the statement f = a + b; written in a high-level language called C. This single statement adds the contents of a with b and stores the result in f. This is equivalent to a number of steps in machine or assembly language, as mentioned before. It should be pointed out that the letters a, b, and f do not refer to particular registers within the CPU. Rather, they are memory locations.

C, a very popular language, is widely used for microcontrollers. A high-level language is a problem-oriented language. The programmer does not have to know the details of the architecture of the microcontroller and its instruction set. Basically, the programmer follows the rules of the particular language being used to solve the problem at hand. A second advantage is that a high level language such as C is portable. That is, a program written in a particular high-level language can be executed by two different microcontrollers, provided that they both understand that language. For example, a program written in C for a PIC18F microcontroller will run on a Texas Instrument's MSP 430 microcontroller because both microcontrollers have a compiler to translate the C language into their particular machine language; minor modifications are required for I/O programs.

Typical microcontrollers are also provided with a program called an "Interpreter". This is provided as part of the software development package. The interpreter reads each high-level statement such as F = A + B and directs the microcontroller to perform the operations required to execute the statement. The interpreter converts each statement into machine language codes but does not convert the entire program into machine language codes prior to execution. Hence, it does not generate an object program. Therefore, an interpreter is a program that executes a set of machine language instructions in response to each high-level statement in order to carry out the function. A compiler, however, converts each statement into a set of machine language instructions and also produces an object program that is stored in memory. This program must then be executed by the CPU to perform the required task in the high-level program.

In summary, an interpreter executes each statement as it proceeds, without generating an object code, whereas a compiler converts a high-level program into an object program that is stored in memory. This program is then executed. Compilers for microprocessors normally provide inefficient machine codes because of the general guidelines that must be followed for designing them. However, modern C Compilers for microcontrollers generate very light and efficient codes. Also, C is a high-level language that includes Input/Output instructions. Hence, C is a very popular programming language with microcontrollers. However, the compiled codes generate many more lines of machine code than an equivalent assembly language program. Therefore, the assembled program will take up less memory space and will execute much faster compared to the compiled C code.

4.5 Introduction to C Language

The C Programming language was developed by Dennis Ritchie of Bell Labs in 1972. C has become a very popular language for many engineers and scientists, primarily because it is portable except for I/O; however, can be used to write programs requiring I/O operations with minor modifications. This means that a program written in C for the PIC18F4321 will run on the Texas Instruments MSP430 with some modifications related to I/O as long as C compilers for both microcontrollers are available.

C is a general-purpose programming language and is found in numerous applications as follows:

- Systems Programming. Many operating systems (such as UNIX and its variant LINUX), compilers, and assemblers are written in C. Note that an operating system typically is included with the personal computer when it is purchased. The operating system provides an interface between the user and the hardware by including a set of commands to select and execute the software on the system.
- Computer-Aided Design (CAD) Applications. CAD programs are written in C. Typical tasks to be accomplished by a CAD program are logic synthesis and simulation.
- Numerical Computation. To solve mathematical problems such as solving a linear system of equations and matrix inversion. Industry standard MATLAB software is written in C.
- Other Applications. These include programs for printers and floppy disk controllers, and digital control algorithms such as PI (Proportional Integral) and PID (Proportional Integral Derivative) algorithms using microcontrollers.

Typical C programs A C- program may be viewed as a collection of functions. Execution of a C program will always begin by a call to the function called "main." This means that all C programs should have its main program named as **main**. However, one can give any name to other functions.

A simple C program that prints "I wrote a C-program" is

```
/* First C-program */
# include <stdio.h>
main ( )
{
        printf ("I wrote C-program") ;
}
```

Here, `main ()` is a function of no arguments, indicated by ( ). The parenthesis must be present even if there are no arguments. All C-programs must include a "main ( )" function which can appear only once. The keyword "void" can optionally be placed within the brackets to clarify that there are no parameters. The statement, main (void) means no arguments are required for function call. The" void" type does not provide a result value to the caller. Hence, "main ( )" means that it does not return a value. Since "main" is classified as a function, all codes must be placed within curly brackets. The braces { } enclose the statements that make up the function. The line "printf("I wrote a C-program");" is a function call that calls a function named "printf", with the argument "I wrote a C-program" . "printf" is a library function that prints output on the terminal. Note that `/* */` is used to enclose comments. These are not translated by the compiler. Compilers for C++ programs are used to compile C programs these days. Hence, // followed by a comment can be used instead of /* */. Note that the comments in C++ are written after //.

A variation of the C program just described is

```
// Another C program
# include  <stdio.h>
main ( )
{
        printf (“I wrote”);
        printf (“a C-”);

        printf (“program);
        printf (“\n”);
}
```

Here, "# include" is a preprocessor directive for the C compiler. These directives give instructions to the compiler that are performed before the program is compiled. The directive `#include <stdio.h>` inserts additional statements in the program. These statements are contained in the file stdio.h. The file `stdio.h` is included with the standard C library. The `stdio.h` file contains information related to the input/output statement.

The `\n` in the last line of the program is C notation for the newline character. Upon printing, the cursor moves forward to the left margin on the next line. `printf` never supplies a newline automatically. Therefore, multiple `printf`'s may be used to output “I wrote a C-program” on a single line in a few steps. The escape sequence `\n` can be used to print three statements on three different lines. An illustration is given in the following:

```
# include <stdio.h>
void main (void )
{
        printf (“I wrote  a C-program  \n”);
        printf (“This will be printed on a new line \n”);
        printf (“So also is this line \ n” );
}
```

All variables in C must be declared before use, normally at the start of the function before any executable statements. The compiler provides an error message if one forgets a declaration. A declaration includes a type and a list of variables that have that type. For example, the declaration ” int a,b” implies that the variables `a` and `b` are integers.

Next, write a program to add two integers a and b where a = 100 and b = 200. The C program is

```
# include <stdio.h>
main ( )
{       int a = 0x64, b = 0xc8;  // a and b are specified in hex
        printf (“The sum is %x\n”, a+b);
}
```

This program shows how to declare two integers and initialize them with hexadecimal numbers. The format specifier %x allows the sum to be printed as a hexadecimal number. This program will print the sum as 0x12C which is 300 in decimal.

The scanf allows the programmer to enter data from the keyboard. A typical expression for scanf is

```
scanf(“%d%d”, &a, &b);
```

This expression indicates that the two values to be entered via the keyboard are in decimal. These two decimal numbers are to be stored in addresses `a` and `b`. Note that the

symbol & is an address operator.

The C program for adding and subtracting two integers a and b using scanf is

```
// C program that performs basic I/O
# include <stdio.h>
main ( )
{
        int a, b;
        printf (“Input two integers: ”);
        scanf (“%d%d” , &a, &b) ;
        printf (“Their sum is: %d\n”, a+b);
        printf (“The difference is: %d\n” , a - b);
}
```

In summary, writing a working C program involves four steps as follows:

Step 1: Using a text editor, prepare a file containing the C code. This file is called the “source file.”

Step 2: Preprocess the code. The preprocessor makes the code ready for compiling. The preprocessor looks through the source file for lines that start with a #. In the previous programming examples, #include <stdio. h> is a preprocessor. This preprocessor instruction copies the contents of the standard header file `stdio.h` into the source code. This header file stdio.h describes typical input/output functions such as scanf( ) and printf( )functions.

Step 3: The compiler translates the preprocessed code into machine code. The output from the compiler is called object code.

Step 4: The linker combines the object file with code from the C libraries. For instance, in the examples shown here, the actual code for the library function printf( ) is inserted from the standard library to the object code by the linker. The linker generates an executable file. Thus, the linker makes a complete program.

In summary, the C language offers the following features:

- provides support to structured programming
- Portable
- includes many operators for low and high level operations
- provides data structures such as arrays, strings, structures, and unions
- Provides I/O operations

4.5.1 Data types

The data types in C language include `char, int, float`, and `double`. A variable declared as a `char` (character) usually holds eight bits of data. A variable of `int` (integer) type, on the other hand, can hold 16 or 32 bits of data. The type `float` specifies a 32-bit single precision floating point number. The type `double` can be used to declare a data as a 64-bit double precision floating point number. We will only use `char` and `int` data types in this book. Note that `float` and `double` data types are not needed in most of the microcontroller-based applications. In addition, floating-point computations are too costly in terms of space and time.

The qualifiers `unsigned` and `signed` can be used with `char` and `int` data types. The `unsigned char` is always positive, and covers a range of values from 0 to 255. Typical examples of `unsigned char` include age and memory address which are always positive.

The `signed char` covers a range of values from -128 to +127 (0 being positive). The C compilers use `signed char` as default. Hence, using `char` instead of `signed char` will specify the data as a `signed character`. Typical examples of `signed char` include voltage and temperature which can be positive and negative.

The `unsigned integer` covers a range of values from 0 to 65535 while the `signed integer` covers a range of values from - 32,768 to + 32767 (0 being positive). The `signed int` or simply `int` (default for C compilers) can be used to specify values from -32,768 to +32,767 (0 being positive).

The lengths of integers can be modified using the qualifiers `shortlong` and `long`. For typical C compilers, shortlong is 24-bit while long is 32-bit.

Examples of declaring `char` and `int` data types are provided below:

```
unsigned char i; /* specifies i as an unsigned 8-bit number*/
char x; /*declares x as a signed 8-bit number
unsigned int b; /*declares b as an unsigned 16-bit integer*/
int a;  /* defines variable as a 16-bit signed integer*/
```

4.5.2 Bit manipulation operators

C provides six bit manipulation operators as shown in Table 4.3. Typical examples for each of these operators are provided below:

```
0x24 & 0x0F = 0x04
0x70 | 0x02 = 0x72
0xE1 ^ 0xFF = 0x1E
      ~ 0x25 = 0xDA
0x27 >>2    = 0x09
0xA1 << 3   = 0x08
```

The bit manipulation operators are very common in I/O operations. Hence, some examples showing their applications in bit manipulation for PIC18F I/O ports are provided in the following.

The AND operator is typically used for clearing one or more bits to 0. For example, the C statement `PORTC = PORTC & 0x7F;` will clear bit 7 of PORTC to 0 without changing the other bits of PORTC. The OR operator is typically used to set one or more bits to 1.

For example, the C statement `PORTD = PORTD | 0x05;` will insert 1's at

TABLE 4.3 Bit Manipulation Operations in C

Logic Operators	Operation performed
&	AND
|	OR
^	XOR
! or ~	NOT
>>	Right Shift
<<	Left Shift

bits 0 and 2 of PORTD without changing the other bits of PORTD. The XOR is typically used to find one's complement of (toggle) one or more bits. For example, the C statement `PORTC = PORTC ^ 0xFF;` will toggle all bits of PORTC.

The above three statements can also be specified in a compact form as shown below:

PORTC & = 0x7F;
PORTD | = 0x05;
PORTC ^ = 0xFF;

Next, let us discuss some of the applications of the logic operators.

The left shift operation is very useful for multiplying an unsigned number by 2^n, shifting it n times to the left provided a '1' is not shifted out of the most significant bit. For example, consider y = 10*x. Note that 10 in binary is 1010_2.
Also, 10*x = 8*x + 2*x. This means that 8*x = x<<3, and 2*x = x<<1.
Hence, y = (x<<3) + (x<<1).

The right shift operation, on the other hand, is very convenient for dividing an unsigned number by 2 by shifting it n times to the right provided a '1' is not shifted out of the least significant bit. As an example, consider y = (a + b)/2.

Note that y = (a + b)/2 = (a+b) >>1

Similarly, y = (a + b +c +d)/4 = (a + b + c + d) >>2

In order to illustrate the Exclusive-OR (XOR) operator, consider X = = Y. This expression is the same as !(X^Y). When X equals Y then X^Y = 0x00, and !(0x00) evaluates to true. The XOR can also be used to swap two variables without the need for a temporary variable. The following example illustrates this:

SWAP with temporary variable

```
temp = x;
   x = y;
   y = temp;
```

SWAP without temporary variable

```
y = x ^ y;
x = x ^ y;
y = x ^ y;
```

Let us verify the above using a numerical example. With x = 1001 and y = 0111, y = x ^ y = (1001)^(0111) = 1110, x = x ^ y = (1001) ^ (1110) = 0111, y = x ^ y = (0111)^(1110) = 1001. Hence, y = x. The above swapping of x with y works because of the following identity,

$p \oplus (p \oplus q) = q$

Also, the above statements can be represented in compact form as:

```
y ^ = x;
x ^ = y;
y ^ = x;
```

Example 4.1 Write a C program to convert a 16-bit number with each byte containing an ASCII digit into packed BCD. The 16-bit number is stored in two consecutive locations (from LOW to HIGH) in data memory with the low byte pointed to by address 0x40, and

the high byte pointed to by address 0x41. Store the packed BCD result in 0x70.

Solution

```
# include <p18f4321.h>
void      main   (void)
   {
         unsigned char  a, b, c;
         unsigned char  addr1 = 0x40;
         unsigned char  addr2 = 0x41;
         unsigned char  addr3 = 0x70;
         addr1 = addr1 &  0x0f;          // mask  off upper four bits of  the low byte
         addr2 =  addr2 & 0x0f;          // mask  off upper four bits of the high byte
         addr2 =  addr2 << 4;            // shift N1 high byte  4 times to left
         addr3 = addr2 | addr1;          // packed BCD byte of  N1 in addr3
   }
```

4.5.3 Control structures

Typical structured control structures in C include `if-else`, `switch`, `while`, `for` and, `do-while`.

4.5.4 The `if-else` construct

The syntax for the `if-else` construct is as follows:

```
if  (cond)
    statement1;
else
    statement2;
```

Figure 4.4 shows the flowchart for the `if-else` construct.

This means that if the condition is true, statement1 is executed; else (if the condition is false), statement1 is skipped, and statement2 is executed. An example of the `if-else` structure is provided in the following:

```
unsigned char x, y, z;
if  (x < y)
    z = x + y;
else
    z = x - y;
```

In the above, if x is less than y, the unsigned 8-bit numbers x and y are added, and the 8-bit result is stored in z. On the other hand, if x is not less than y, then the statement z = x - y is executed. As another example, consider the following. This code finds the larger of the two 8-bit unsigned numbers, a and b, and saves the result in "big".

```
unsigned   char a, b, big;
 if (a>b)
   big = a;
 else
   big = b;
```

Braces are required if multiple statements need to be executed if the condition is true or false. Consider x and y as two unsigned numbers. If x>y, then find x+y, and x/y

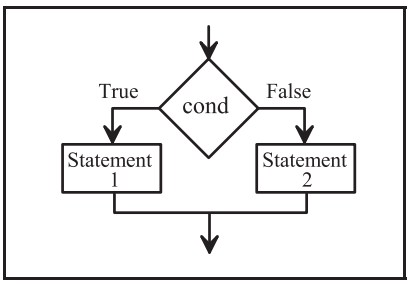


FIGURE 4.4 The `if-else` construct

and store them in u and v respectively. Otherwise, find x-y and increment w by 4, and store them in z and w respectively. Finally, compute x*y, and store it in t.

```
unsigned char t, u, v, w, x, y, z;
if    (x < y) {
      u = x + y;          // add x with y, and store in u
      v = x / y;          // divide x by y and store in v
}
else  {
      z = x - y;          // subtract y from x and store result in z
      w+ =  4;            // value of  w  is incremented by 4
}
      t = x * y           // multiply x by y and store in t
```

In the above, if x < y then the statements for u and v in the braces are executed. The statement t = x * y is then executed. If x is not greater than y then only the statements for z and w+ in the braces are executed. The statement for t is then executed. In either case, t = x * y is executed.

4.5.5 The `switch` construct

The syntax for the `switch` expression is

```
switch ( integer){
case 1:
     statements;
      break;
case 2:
      statements;
      break:
           ---
           ---
           ---

case n:
      statements;
 }
```

In the above, the integer included with the switch statement is compared with each of the integers included with the case statements. If the values match, the statements associated with that case statement are executed. For example, consider switch (2). If

integer = 2, then the statements with case 2 are executed. The break exits from the switch construct.

4.5.6 The `while` construct

The `while` construct allows a programmer to describe loops. The syntax for the `while` construct is provided below:

```
while (condition)  {
        statements
}
next statement;
```

In the above, if the condition is true, the statements are executed, and then control is returned to the top of the loop. The condition is tested again. If the condition is true again, statements are executed, and control is returned to the top of the loop. The process is repeated as long as the condition is true. However, if the condition is false at the start or the duration of repeating the process when the condition is checked, the statements in the braces are not executed. The next statement following the second brace is executed. Note that braces are not required for a single statement. Figure 4.5 depicts the flowchart for the while construct.

As an example, consider the following with a single statement in the loop:

```
int   n = 4;
while  ( n <= 16)
      n = n + 4;
       z = n;
```

In the above, n is 4 before entering the while loop. The condition n<= 16 is true the first time. The expression n = n + 4 continuously changes the value of n in increments of 4 to 8, 12, 16, 20. As soon as n = 16, the condition n <= 16 is false, the next statement z = n is executed which assigns z with the value of 20.

Finally, note that an infinite loop occurs when the condition in the `while` construct is always true (‘1’ in C means true). Note that

```
while  (1)
```

describes an infinite loop in C , and is equivalent to

```
here   bra     here
```

in PIC18F assembly language.

The `while` loops can also be used to write software delay routines. A simple delay loop using the `while` construct is provided below:

```
unsigned  int  k = 1000; // initialize  k  to 1000
   while (k>10)
         k -- ;
```

A nested delay loop using the while construct is provided below:

```
unsigned  int i, j, k ;
                  i  = 0; k = 1000;
                  while (i < k) {
                     j = 1;
                     while (j < 100)
                         j++;
                     }
```

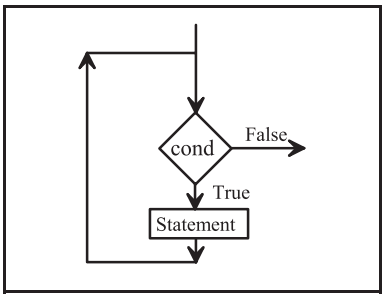


FIGURE 4.5 The `while` construct

4.5.7 The `for` construct

The `for` construct is another loop structure supported by the C language. It is more flexible than the while construct. Hence, the for construct is often a preferred choice. The syntax of a `for` construct is as follows:

```
for  (e1; e2; e3) {
        s;
}
```

where e1, e2, and e3 are C expressions and 's' represents one or more valid C statements. Figure 4.6 shows the flowchart for the `for` loop.

In Figure 4.6, the expression e1 is evaluated first. The expression e2 is then evaluated; if it is false the loop terminates. Otherwise, 's' representing one or more C statements will be executed. The expressions e3, and e2 are then evaluated, and the process continues. The expression e1 normally contains code to initialize the loop. The expression e2 describes the exit condition. The purpose of the expression e3 is to modify such as increment the initialized value of expression e1. This will be the exit condition so that the loop terminates at some point.

For example, assume e1 as k = 4, e2 as k<8, and e3 as k = k + 1 in Figure 4.7. With the above data, the "for" loop can be written as

: for (k = 4; k<8; k = k + 1)

In this "for" loop, 'k' is initialized with 4. Since, 4 is less than 8, the condition is true and hence, the statement(s) 's' within the curly brackets will be executed. 'k' is then incremented to 5. The loop then goes back to e2 to check the condition. The loop will be repeated until k = 7. This can be illustrated by the following C-program:

```
#include <stdio.h>
int k;
main ( ) {
for (k = 4; k<8; k = k+ 1){
     printf ("k= %d\n", k);
}
}
```

After compilation and execution of the above C-program, the following result will be generated:

```
k = 4
k = 5
k = 6
k = 7
```

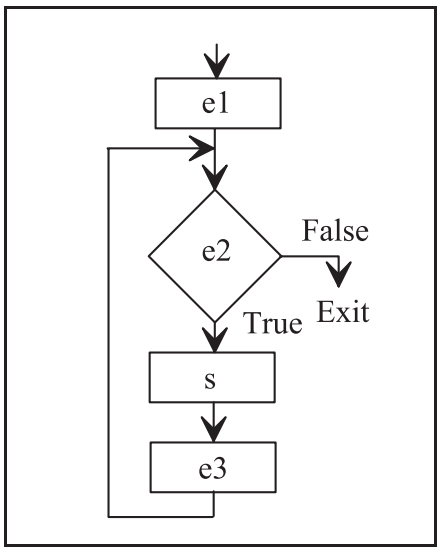


FIGURE 4.6 Flowchart for the `for` loop

4.5.8 The `do-while` construct

The C language provides another loop structure called the `do-while`. The syntax for the `do-while` construct is provided below:

```
do {
        statements
} while (condition);
```

Figure 4.7 shows the flowchart for the `do-while` construct.

The `do-while` loop is a post-checked loop (condition is checked afterwards) in the sense that the exit condition is evaluated after executing all statements. If the condition evaluates to False, the loop terminates. Otherwise, the process continues until the condition becomes false. Whatever is accomplished by a `do-while` construct can be achieved by a `for` loop construct. Hence, this loop structure is not as popular as `while` or `for` loop structures.

4.5.9 Structures, and Unions

In addition to built-in data types such as `char` and `int`, C supports user-defined data types. A structure permits the programmer to access a group of different data types using a common user-defined name. The structure can be declared using the keyword `struct` followed by a user-defined name. An example of the structure declaration is provided below:

```
struct    struct_name {
              int a;
              char b ;
          } my_struct ;
```

Note that in the above, the `name` is optional. However, if it is present, it defines the tag or user-defined name of the structure, and the tag can be used later. The variable name `my_struct` is of type `struct_name`. Also, Size of the structure = Sum of the sizes of its components. Hence, Size of the above structure = Size(int) + Size (char) = 2 bytes + 1 byte = 3 bytes.

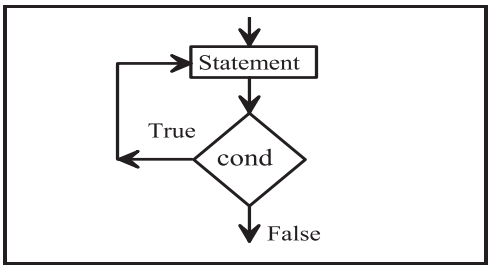


FIGURE 4.7 Figure for the `do-while` construct

Union is a space saving structure. The memory referenced by the keyword `union` can store different types of data with the restriction that at any one time, the memory holds a single type of data. Note that different data types of the union share the same memory space. The data in a union must be referenced by a member of the proper data type. As an example, a union can be declared as follows:

```
union   num     {
char    x ;
int     a ;
} ;
```

Note that Size of a union = Size of the largest data type held by the union. In the above, Size of the union "num" = Max {size of (int), size of (char)} = Max {2, 1} = 2 bytes.

In the above code, any variable of type `union` num can hold a char or an int. The names 'x' and 'a' specify which type of data is referenced. To specify a variable `y` of type `union num`, the following statement can be used:

```
union num y;
```

Now, a character B can be assigned to x using the following:

```
y . x = 'B' ;
```

Next, an integer 2756 can be assigned to using the following:

```
y . a = 2756 ;
```

Since at any one time y holds data of a single type, the assignment of 2756 to y cancels the assignment of 'B' to y.

As mentioned before, Union offers a space saving structure. For union variable, memory space is allocated to hold the largest number. This memory can then be used to hold the smaller numbers. As an example, consider the following:

```
union   {
        int a ;
        char c[2] ;
} myu ;
```

In the above union, 16 bits of memory space are allocated to integer 'a'; the same storage space is shared by 8-bit characters c[0] and c[1] as shown in Figure 4.8.

4.5.10 Functions in C

A function in C is similar to subroutine in assembly. Hence, a function is a block of C codes that performs a particular task. There are two types of functions in C. They are Standard library functions and User defined functions. The Standard library functions are contained in the C compilers. They handle tasks such as mathematical computations and I/O processing. These functions are defined in the compiler's header file. For example, the "printf ( )" is defined as a standard library function in the C compiler's "stdio.h" header

file. The "printf ( )" outputs formatted data on the screen.

User-defined functions are defined by the programmers. An example of a user-defined function "minus ( ) " is provided in the following:

```
#include <stdio.h>
int  minus (int m, int n);  // function prototype
main  ( ) {
  int a = 5;
  int b = 3;
  int result;
  result = minus (a, b); //function call
  printf ("result =   %d", result);
  return 0;
}
int minus (int m, int n)
{
   int subtract;
   subtract = m - n;
   return subtract;  // return statement
}
```

In the above, "int minus (int m, int n)" is called the function prototype which specifies in the above, the return type of the function as "int", function name as "minus ( )" and two arguments (m, n) of type "int" are passed to the function. The function prototype is not required if the user-defined function is defined before the "main ( )" function.

In the above example, the function is called using minus (a, b) which is a statement inside the main ( ). After the function call, program control is transferred to the user-defined function.

The arguments mean the variables are passed to the function. In the above example, two variables 'a 'and 'b' are passed during function call. The parameters 'm' and 'n' accept the passed arguments in the function definition since they are of the same type (int).

The return statement ends the execution of a function and returns the value to the calling function. The program control is transferred to the calling function after the return statement. In the above example, the value of the variable "subtract" is returned to the variable "result" in the
main ( ) function.

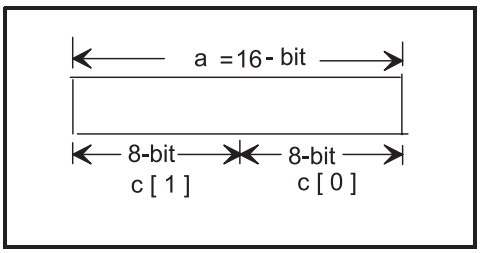


FIGURE 4.8 Example of space saving with union

4.5.11 Arrays

An array contains a number of the same type of data elements stored under a common name in consecutive memory locations. The same type of data elements means that data elements can be, for example, character (char) elements or integer (int) elements. The size of each element in the array can be 8-, 16- or 32-bit. Arrays can be one-dimensional (table) or two-dimensional (matrices). One-dimensional arrays will be considered in this discussion.

An array is specified by an array type, array name, and array elements. If the array stores integer data, then the array type is "int". On the other hand, if the array stores characters, then the array type is "char".

An array is assigned with a name comprised of a string of characters. The elements of an array are stored within curly brackets. The first element of the array is identified as element 0, the second element as element 1, the third element as element 2, and so on.

An example of creating an array containing 10 BCD numbers is provided below:

```
unsigned char table [10] = {0,1, 2, 3, 4, 5, 6, 7, 8, 9};
```

In the above, the array name is "table" with element 0 as 0, element 1 as 1, and so on.

4.5.12 Macros

Macros can make programming in C easier by reducing the amount of code the programmer has to actually write. This is accomplished by letting the compiler produce those redundant pieces of program that are used routinely in various places in the program. Basically a macro is a set of codes used repeatedly throughout the program. When the macro is written , it is assigned a name. Rather than writing the same sequence of codes each time they are needed, the name of the macro is inserted in their place. During compiling whenever the name of the macro is encountered, the compiler will insert the sequence of codes that this name represents.

The macro is in some way similar to a function, in that the usage is repeated. As an example, consider the following macro**:**

```
#define   MPY10 (X)  (X << 3) + (X<< 1)
```

This macro multiplies X by 10 using two left shift operations discussed earlier. A typical macro call is `Z = MPY10 (Y);`

Note that there is no transfer of control in macro. In contrast, a function call is performed via transfer of control in the program. The C compiler expands the macro in the program. For example,

`Z = MPY10 (A) + MPY10 (B)` is expanded as

```
Z = (A<<3) + (A<< 1) + (B<<3) + (B<<1);
```

Macros are faster than functions or subroutines since there is no overhead in macro associated with transfer of control.

4.6 Choosing a programming language

Compilers used to provide inefficient machine codes for microprocessors because of the general guidelines that must be followed for designing them. However, modern C compilers with microcontrollers generate very tight and efficient codes. Some of the other advantages

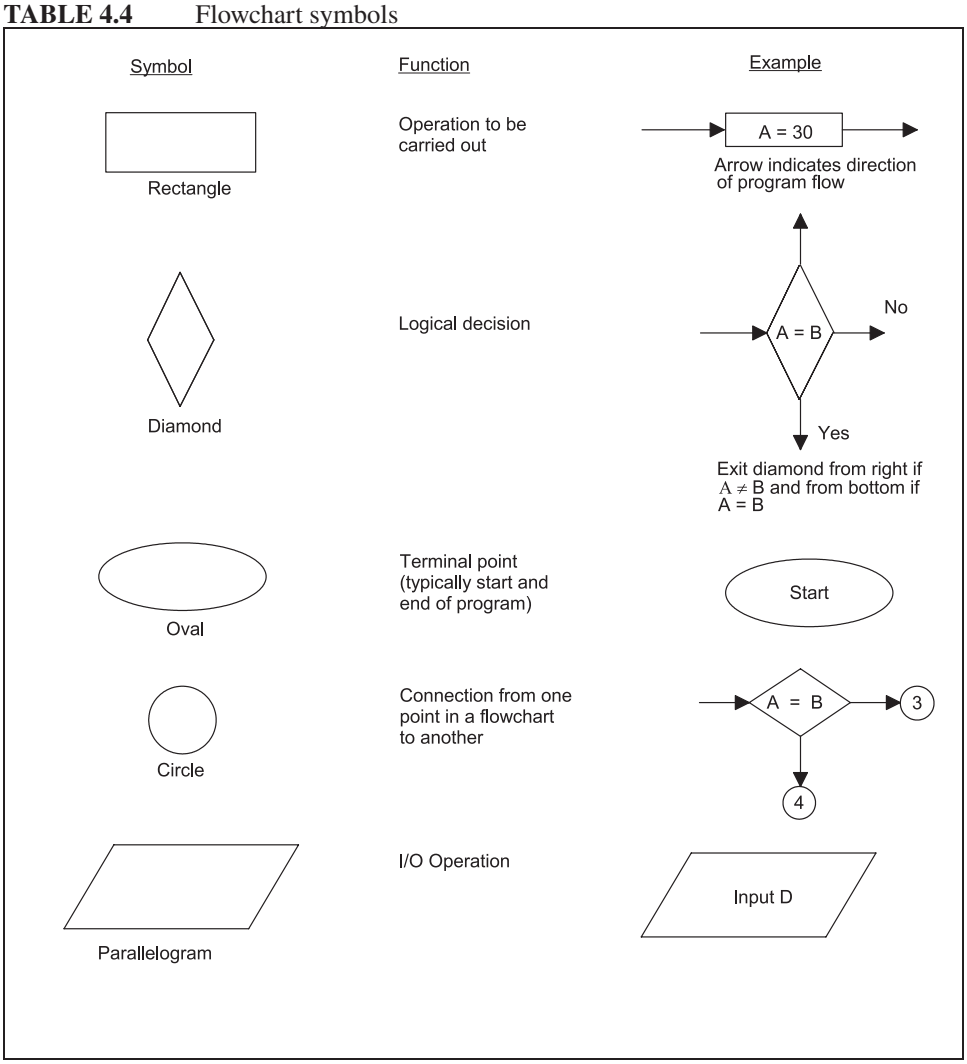
of using C language over assembly were mentioned in the last section. Hence, C is widely used these days with microcontrollers.

Assembly language programming, on the other hand, is essential for understanding the internal architecture of a microcontroller. Furthermore, familiarization with assembly language may be helpful in understanding how to write microcontroller interrupt programs using C. This topic will be discussed later. Finally, assembly language may be useful for debugging a C-program.

4.7 Flowcharts

Before writing an assembly language program for a specific operation, it is convenient to represent the program in a schematic form called a *flowchart.* A brief listing of the basic shapes used in a flowchart and their functions is given in Table 4.4.

TABLE 4.4 Flowchart symbols



QUESTIONS AND PROBLEMS

4.1 What is the basic difference between assembly and high-level languages? Why would you choose one over the other?

4.2 Assume that two microcontrollers, the PIC18F and the HC12, have C compilers. Will a program written in C language run on both microcontrollers?

4.3 Will a program written in Microchip's PIC18F assembly language run on microcontrollers from other manufacturers?

4.4 Determine the contents of address 0x23 after assembling the following:

(a)
```
ORG   0x20
DB    00H, 05H, 07H, 00H, 03H
```
(b)
```
ORG   0x20
DW    0702H, 123FH, 7020H, 0000H
```

4.5 What is the difference between:

(a) A cross assembler and a resident assembler?

(b) A two-pass assembler and a meta-assembler?

4.6 Write a program equivalent to the C language assignment statement

```
z = a + (b * c) + (d * e) - (f / g) - (h * i);
```

Use only:

(a) Three-address instructions

(b) Two-address instructions

4.7 Assume that a microcontroller has only two registers, R1 and R2, and that only the following instruction is available:

```
XOR   Ri,Rj    ; Rj <- Ri  Rj
               ; i,j = 1,2
```

Using this XOR instruction, find an instruction sequence to exchange the contents of registers R1 and R2.

4.8 Assume 2 two's-complement signed numbers, $M = 11111111$ and $Q = 11111100$. Perform signed multiplication using the algorithm described in Section 4.3.6.

4.9 Using the signed division algorithm described in section 4.3.6, find the quotient and remainder of (-25)/3.

4.10 Find the logic operation and 8-bit data for clearing bits 2 and 4 of an 8-bit number, 7E to 0's without changing the other bits.

4.11 Find the logic operation and 8-bit data for setting bits 0 and 7 of an 8-bit number, 3A to 1's without changing the other bits.

4.12 Find the overflow bit after performing an arithmetic shift on B6 three times to the left.

4.13 Describe the meaning of each of the following addressing modes.

(a) Immediate (b) Absolute
(c) Register indirect (d) Relative
(e) Implied

4.14 What are the advantages of subroutines?

4.15 Explain the use of a stack in implementing subroutine calls.

4.16 Which programming language is popular with microcontrollers? Why?

4.17 Find the results after each of the following operations:

(a) 0x24 & 0x81 (b) 0x60 | 0x90 (c) 0x81 ^ 0xA2
(d) ! 0x25 (e) 0x1F <<4

4.18 What is the keyword in C to transfer control from a function back to the calling function?

4.19 Does every function have to return a value? Why or Why not?

4.20 Name the data type that can be checked in a switch-case statement.

4.21 What will happen if you assign a value to an array element in a C-program whose subscript exceeds the size of the array?

4.22 What is the advantage of using union?

4.23 Why would you use Macros?

5

PIC18F ARCHITECTURE AND ADDRESSING MODES

In this chapter we describe the PIC18F microcontroller architecture and addressing modes. Topics include an introduction to the PIC18F, memory maps, pipelining, register architecture, and addressing modes.

5.1 Basic features of the PIC18F family

The PIC18F is Microchip's 8-bit RISC-based microcontroller. Since the PIC18F CPU uses the Harvard architecture, program and data memory units use separate memory spaces along with their own buses. This allows the PIC18F to access both programs and data simultaneously. The program memory content of an address of all microcontrollers in the PIC18F family is organized in words of 12-, 14-, or 16-bit; the data memory is based on 8 bits. The PIC18F uses flash memory to store program memory while RAM contains data memory. Note that F in PIC18F indicates that the chip contains flash memory.

Several registers are mapped as memory locations, and these registers are called "Special Function Registers (SFRs)". This makes it convenient for the programmers to use the register names in the program; the PIC18F assembler or C18 compiler automatically inserts their binary addresses in the program. For example, PORTC is mapped as an SFR with address 0xF82. The programmer can use the name PORTC rather than its physical address, 0xF82. Note that SFRs are CPU registers and registers of the peripheral module. Typical CPU registers include Stack Pointer, STATUS register, and Program Counter. Typical peripheral registers, on the other hand, include I/O ports, Timer registers, ADC registers, and CCP registers. The PIC18F assembler and the C18 compiler can access these registers by name. They can be read and written like a variable defined in the application.

Several PIC18F family members are provided with additional EEPROM to store data that will be nonvolatile. Some versions of the PIC family contain one-time programmable ROM for program memory; this is in addition to data RAM. The PIC16C432 is an example of such a chip. The letter C in PIC16C432 indicates that the chip contains one-time programmable ROM for program memory. The PIC18F is normally used for product development. Once developed, the PIC18C is used for mass production of the product.

The hardware stack of the PIC18F provided by Microchip is not part of the data memory. It has its own independent memory space. Therefore, the PIC18F hardware stack is limited.

The maximum size of the Program Counter (PC) of the PIC18F is 21-bit wide. Hence, the PIC18F can directly address a maximum of two Megabytes (2^{21}) of program

Microcontroller Theory and Applications with the PIC18F, Second Edition. M. Rafiquzzaman.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

memory space.

The data memory address, on the other hand, is 12-bit wide. Hence, the PIC18F can directly address data memory of up to 4Kbytes (2^{12}) . However, the sizes of program memory and data memory may vary from one of the PIC18F family members to another.

As mentioned before, there are several versions of the PIC18F microcontroller. The sizes of program and data memories, number of Input/Output (I/O) ports, and the clock frequency may vary from one version to another. For example, the PIC18F4321 contains 256 bytes of SRAM. The PIC18F4520, on the other hand, includes 1536 bytes of RAM. Two popular PIC18F family members include PIC18F4321 and PIC18F20. Table 5.1 summarizes the basic differences between them.

Most of the features in Table 5.1 are self-explanatory. However, the purpose of some of the PIC18F on-chip features such as timers, CCP modules, ADC converter, and serial communication interface will be provided in the following.

The PIC18F can perform functions such as capture, compare, and pulse width modulation (PWM) using the timers and CCP (Capture / Compare / PWM) modules. The PIC18F can compute the period of an incoming signal using the capture module. The PIC18F can produce a periodic waveform or time delays using the compare module. The PIC18F's on-chip PWM can be used to obtain pulse waveforms with a particular period and duty cycle which are ideal for applications such as motor control. The PIC18F serial communication interface can be used to facilitate data transmission for serial peripheral devices which can transmit or receive data one bit at a time.

The on-chip 10-bit A/D converter of the PIC18F can convert an analog signal into 10-bit binary equivalent. This is very convenient in practice since physical variables such as temperature, flow, and pressure are analog in nature and must first be converted into an

TABLE 5.1 Basic differences of two popular members of the PIC18F Family (F in PIC18F indicates on-chip flash memory)

On-chip features	PIC18F4321	PIC18F4520
Flash memory (Program memory)	4k x 16	4k x 16
RAM data memory (bytes)	256	1536
EEPROM (bytes)	256	256
Maximum Operating Frequency	40 MHz	40 MHz
I/O Pins	36	36
Timers	4	4
Capture/Compare/PWM (CCP) modules	2	2
Serial communication interface	Yes	Yes
10-bit ADC (Can also be configured as 8-bit)	13 Channels	13 Channels
Instruction set	77 instructions; 83 with extended set enabled	75 instructions; 83 with extended set enabled
Number of pins	40, 44	40, 44

analog electrical signal. Since the microcontrollers only understand binary numbers, the on-chip ADC of the PIC18F then converts the electrical signal to an 8-bit or 10-bit binary value before processing.

All PIC18F microcontrollers use pipelining to execute instructions. The pipelining is performed in two stages which is defined as a single instruction cycle.
The PIC18F microcontrollers use the main oscillators (OSC1) to synchronize its internal operations.

The PIC18F clock input is internally divided by four to generate four non-overlapping clocks (Q1, Q2, Q3 and Q4). A machine cycle is defined as four pulses from the main oscillator (OSC1). During the time Q1 is active in any machine cycle, the program counter is incremented by 2 (since the PIC18F instruction size is normally 16-bit) to point to the next instruction to be fetched. This instruction will be fetched during Q4. Simultaneously, the previous instruction is being executed during the whole machine cycle (from Q1 to Q4).

As mentioned in Chapter 2, there are three phases of an instruction execution. They are fetch, decode, and execution. During the fetching phase, the instruction is fetched

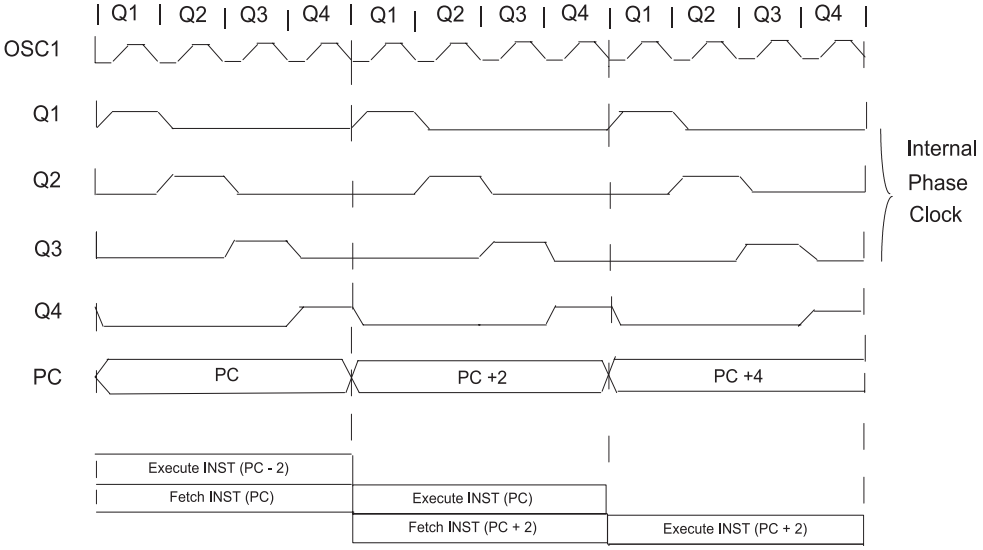
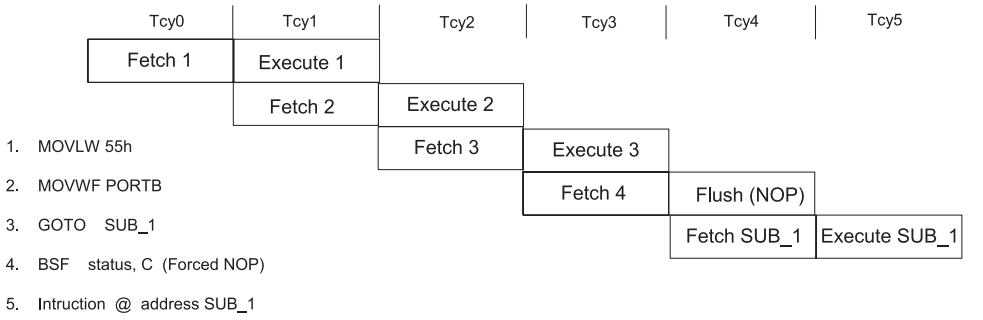


FIGURE 5.1 Clock/Instruction Cycle



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

FIGURE 5.2 Instruction Pipeline Flow

from the program memory and stored into the Instruction Register (IR) during Q4. During the decoding phase, the control unit translates the contents of the IR, and determines the specified operation to be performed. Finally, during the execution phase, the microcontroller executes the instruction. The clocks and instruction execution flow are shown in Figure 5.1.

As mentioned before, the PIC18F uses a two-stage pipeline. This means that execution of the previous instruction is overlapped with fetching of the current instruction. Hence, the PIC18F basically executes instructions in one machine cycle. The basic concepts associated with PIC18F pipelining will be described in the following.

Figure 5.2 shows the PIC18F instruction pipeline flow. Four PIC18F instructions (MOVLW, MOVWF, GOTO, BSF) are used to illustrate the PIC18F pipelining. These instructions are chosen arbitrarily. The meaning of these four instructions are provided below:

MOVLW 0x55 moves 55 hex (8-bit immediate data) into WREG (Accumulator).
MOVWF PORTB moves the contents of WREG (Accumulator) into PORTB.
GOTO SUB_1 unconditionally jumps to address SUB_1.
BSF STATUS, C sets carry flag in the status register to 1.

In the PIC18F, an "Instruction Cycle" consists of four Q cycles: Q1 through Q4. The instruction fetch and execute cycles are pipelined in such a manner that a fetch takes one instruction cycle, while the decode and execute phases take another instruction cycle. However, due to the pipelining, each instruction effectively is executed in one cycle. If an instruction causes the program counter to change (e.g., GOTO), then two cycles are required to complete the instruction. A fetch cycle begins with the Program Counter (PC) incrementing in Q1. In the execution cycle, the fetched instruction is stored 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).

The PIC18F executes each instruction in a single cycle, except for any branch instructions. This is explained in the following.

Consider Figure 5.2. The PIC18F fetches MOVLW instruction into IR during Tcy0. The PIC18F executes MOVLW in Tcy1 and also fetches the next instruction, MOVWF in Tcy1. The PIC18F then executes MOVWF in Tcy2 and fetches GOTO instruction into IR in Tcy2. The PIC18F executes GOTO into IR in Tcy3, and also fetches BSF into IR in Tcy3.

The PIC18F unconditionally jumps to address SUB_1, and executes the instruction at address SUB_1. Hence, the pipeline is flushed (NOP) in Tcy4, and the instruction at address SUB_1 is fetched in Tcy4. The instruction at address SUB_1 is executed in Tcy5. Hence, each of the instructions, MOVLW and MOVWF, is executed in a single cycle. The GOTO instruction, on the other hand, takes two cycles since the instruction BSF is flushed from the pipeline while the new instruction at address SUB_1 is fetched and then executed.

In order to illustrate the basic features of microcontrollers, one of the PIC18F family members such as the PIC18F4321 is used in this book for developing the programming examples and illustrating interfacing techniques with the PIC18F.

5.2 PIC18F Register Architecture

In order to program the PIC18F in assembly language, one must be familiar with the registers of the PIC18F. Hence, a description of these registers is provided in this section.

Figure 5.3 shows the PIC18F CPU registers**.** All registers in the PIC18F are mapped as addresses in the data memory, and are called SFRs (Special Function Registers). Hence, each register is assigned with a unique 12-bit memory address. We now briefly describe the functions of these registers in the following.

WREG The WREG (Working register) is 8-bit wide. This is the PIC18F accumulator, and has its usual meaning. Most arithmetic and logic operations are performed using the WREG.

SP The SP (Stack Pointer) register (called STKPTR in the PIC18F) is 8-bit wide. The PIC18F hardware stack is a group of 31 21-bit registers to hold memory addresses. Note that the PIC18F program counter is 21-bit wide. Hence the stack is 21-bit wide. The low five bits of the STKPTR are used to address the stack. The PIC18F hardware stack is accessed from the bottom. Hence, the SP is incremented by 1 after PUSH and decremented by 1 after POP. Example 7.2 illustrates this. Figure 5.4 shows the details of the SP. Note the hardware stack is implemented by Microchip and is independent of the data memory. The PIC18F subroutine CALL and RETURN instructions push and pop the program counter using the hardware stack. The hardware stack is different from the software stack. The hardware stack is designed by the manufacturer, and is normally used by subroutines and interrupts in PIC18F assembly language while the software stack is created by the programmer using pointers and appropriate addressing modes. The software stack is normally used to store local variables.

The 31 stack registers are neither part of program memory nor data memory. As shown in figure 5.4, the low five bits of the SP address the stack. The stack overflow bit

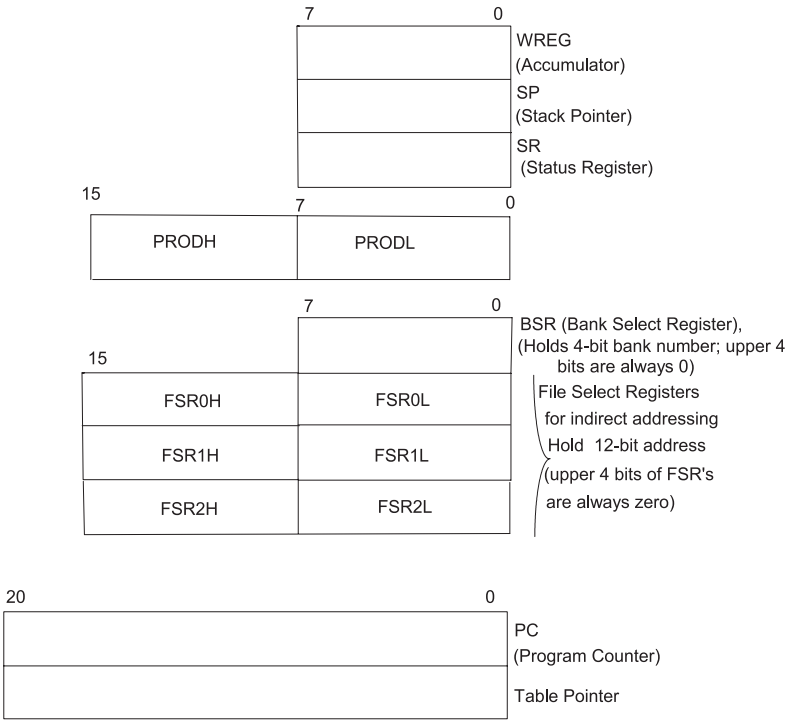


FIGURE 5.3 PIC18F registers

(STKFUL, bit 7) is set to one if more than 31 registers are attempted for pushing addresses onto the stack by the programmer; otherwise, the stack overflow bit is cleared to zero. The stack underflow bit (STKUNF, bit 6), on the other hand, is set to one if more addresses that are stored in the stack are attempted to be popped by the programmer; otherwise, the stack underflow bit is cleared to zero. Bit 5 is not implemented and is read as 0.

PC The maximum size of the PIC18F's PC (Program Counter) is 21-bit wide. The size of the PC varies from one PIC18F family member to another. For example, the size of the PC of the PIC18F4321 is 13-bit wide. The PC normally points to the next instruction. As mentioned before, the 21-bit PC provides the PIC18F with direct addressing capability of a maximum of 2MB (2) of program memory. Upon hardware reset, the PC is loaded with zero so that the PIC18F CPU fetches the first instruction from address 0.

The 21-bit PC is comprised of three 8-bit registers, namely PCL (PC Low byte), PCLATH (PC latch High byte), and PCLATU (PC Latch Upper 5 bits). The 21-bit PC is stored in these registers as follows: bits 0 through 7 in PCL, bits 8 through 15 in PCLATH, and bits 16 through 20 in the low five bits of PCATU. Registers PCL, PLATCH, and PCLATU are mapped as special function registers in the data SRAM by the PIC18F.

Table Pointer The PIC18F uses the 21-bit Table Pointer register as a pointer to a table in program memory for copying bytes between program memory and data memory. This register is mapped by the PIC18F as three 8-bit special function registers in the data SRAM.

BSR The BSR (Bank Select Register) is 8-bit wide. The low four bits are used to provide the bank address from 0 to F; upper four bits of BSR are zero. The BSR provides the upper four bits of a 12-bit address of data memory. BSR is used for directly addressing the data SRAM.

FSR The FSR (File Select Register) consists of three 16-bit registers (FSR0, FSR1, FSR2) with the upper four bits of each FSR as zero. The low 12 bits of FSR0, FSR1, or FSR2 are used to hold the 12-bit memory address of the data SRAM. These registers are basically memory pointers. Hence, they are typically used for handling arrays and pointer-based data accessing. The PIC18F indirectly uses these registers to access data in data SRAM. Each of these three registers are divided into two 8-bit registers as follows:
FSR0H (high byte of FSR0) and FSR0L (low byte of FSR0), FSR1H (high byte of FSR1) and FSR1L (low byte of FSR1), FSR2H (high byte of FSR2) and FSR2L (low byte of FSR2).

PRODH / PRODL Each of the PRODH and PRODL registers is 8-bit wide. The PIC18F has two 8-bit x 8-bit unsigned multiplication instructions providing 16-bit product. The upper byte of the product is stored in the PRODH register while the lower byte of the product is placed in the PRODL register.

7	6	5	4	3	2	1	0
STKFUL	STKUNF	——	SP4	SP3	SP2	SP1	SP0

FIGURE 5.4 SP (Stack Pointer)

7	6	5	4	3	2	1	0
–	–	–	N	OV	Z	DC	C

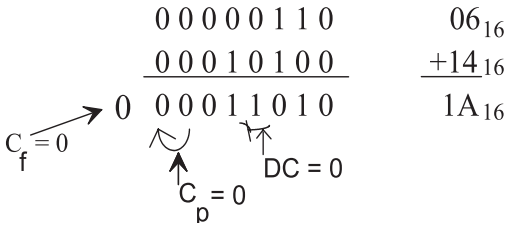
FIGURE 5.5 SR (Status Register)

SR The SR (Status Register) is 8-bit wide. Figure 5.5 shows the PIC18F status register which contains the flags. The meaning of these flags will be explained in the following:

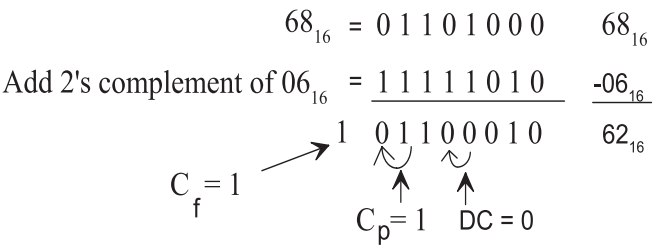
- C (Carry flag) is set to 1 if there is a carry from addition or a borrow from subtraction; otherwise, C = 0.
- DC (Digit Carry flag) is set to 1 if there is a carry due to addition of the low 4 bits into the high 4 bits or a borrow due to the subtraction of the low 4 bits from the high 4 bits of a number; otherwise, DC = 0. This flag is used by BCD arithmetic instructions.
- Z (Zero flag) is set to 1 if the result is zero; Z = 0 for a nonzero result.
- OV (Overflow flag) is set to 1 if there is an arithmetic overflow (i.e., if the size of the result exceeds the capacity of the destination location); otherwise, OV = 0. Note that in overflow, OV = $C_f \oplus C_p$ where C_f is the final carry and C_p is the previous carry. Overflow is used for signed arithmetic (2's Complement).
- N (Negative flag) is set to 1 if the most significant bit of the result is 1 indicating a negative number; N = 0 if the most significant of the result is 0 indicating a positive number.
- Bits 5 through 7 are not implemented, and are read as zero.

In order to provide a clear understanding of how status flags are affected by arithmetic instructions, numerical examples will be provided in the following.

Consider adding 06_{16} with 14_{16} as follows:



In the above, C = C_f = 0, DC = 0 (No carry from bit 3 to bit 4), Z = 0 (nonzero result), OV = $C_f \oplus C_p$ = 0 ⊕ 0 = 0 (meaning correct result), N = 0 (Most significant bit of the result is 0 indicating a positive number). Next, consider subtracting 06_{16} from 68_{16} using 2's complement. The result will be 62_{16}.



In the above, C (Borrow) = one's complement of C= 0, DC = 0 (No carry from bit 3 to bit 4), Z = 0 (nonzero result), OV = $C_f \oplus C_p$ = 1 ⊕ 1 = 0 (meaning correct result), N = 0 (Most significant bit of the result is 0 indicating a positive number). Note that while obtaining two's complement subtraction analytically, the correct borrow is always the one's complement of the borrow obtained analytically. Hence, microcontrollers perform one's

complement operation on the borrow in order to reflect the correct borrow which will be useful in multiprecision subtraction.

5.3 PIC18F Memory Organization

Three types of memories are utilized in the PIC18F. They are Flash memory, SRAM and EEPROM. The Flash memory is used to store programs. The SRAM, on the other hand, contains data. Some versions of the PIC18F family contain EEPROM along with SRAM to hold data. Note that SRAM is a volatile read/write memory. The EEPROM, on the other hand, is a nonvolatile memory.

As mentioned before, data can be specified in the instruction as immediate data or by the address of data. For example, consider the MOVLW 2 instruction in the PIC18F. The MOVLW 2 instruction will load the WREG (Accumulator) register with immediate data 2. After assembling, MOVLW 2 will be part of the program memory. On the other hand, the MOVF 0x20, W instruction will load the contents of data memory address 0x20 into WREG. Now, if the contents of 0x20 is 02H, then 02H will be loaded into WREG. In this case, after assembling, the instruction MOVF 0x20, W will be part of the program memory while data 02H will be located in data memory.

The EEPROM is separate from the data SRAM and program flash memory. The EEPROM is used for long-term storage of critical data. The EEPROM is normally used as a read-mostly memory since its read time is faster than write times. It is not directly mapped in either the register file or program memory space but is indirectly addressed through the Special Function Registers (SFRs). One of the main advantages of including the EEPROM in the PIC18F is that all critical data stored in the EEPROM can be protected from reading or writing by other users. This can be accomplished by programming appropriate bits in the corresponding SFR. Note that the PIC18F4321 contains 256 bytes of EEPROM.

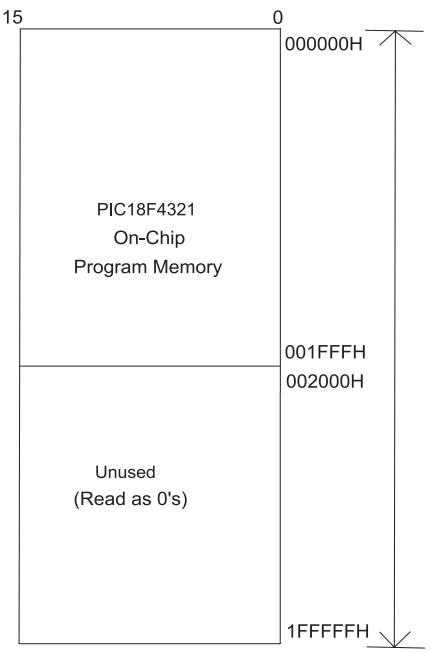


FIGURE 5.6 PIC18F4321 Program Memory Map

The data memory in PIC18F devices is implemented as static RAM. Each register in the data memory has a 12-bit address, allowing up to 4096 bytes (2^{12}) of data memory. The memory space is divided into as many as 16 banks that contain 256 bytes each.

5.3.1 PIC18F Program Memory

As mentioned before, the Program Counter (PC) contains the address of the instruction to be fetched for execution. The maximum size of the PIC18F PC is 21 bits wide. The PC addresses bytes in the program memory. To prevent the PC from becoming misaligned with 16- or 32-bit wide instructions, the Least Significant bit of PC is fixed to a value of '0'. This is because the address is an even number for 16-bit or 32-bit instructions. The PC increments by 2 or 4 to address sequential 16- or 32-bit wide instructions in the program memory.

Program memory is implemented in flash memory in the PIC18F. With a maximum of 21-bit address, the PIC18F address range goes from 21 zeros to 21 ones (addresses 0x 000000 to 0x1FFFFF). This 21-bit address provides a maximum program memory size of $2^{21}(2^{10} \text{ x } 2^{10} \text{ x } 2^{1}) = 1\text{k x } 1\text{k x } 2 = 2$ MB). The size of the program memory may vary from one member to another in the PIC18F family.

The PIC18F4321 contains 4k 16-bit words of on-chip flash memory, and can store up to 4096 (4k) 16-bit word instructions. Note that most PIC18F instructions are 16-bit wide. Figure 5.6 shows the program memory map for the PIC18F4321 with addresses from 13 zeroes to 13 ones providing an address range from 0x000000 to 0x001FFF; the unused address bits of the 21-bit address are always read as 0's.

5.3.2 PIC18F Data Memory

The PIC18F data memory is also called "file register" or "data register" by Microchip. Figure 5.7 shows the data memory organization of the PIC18F4321. The size

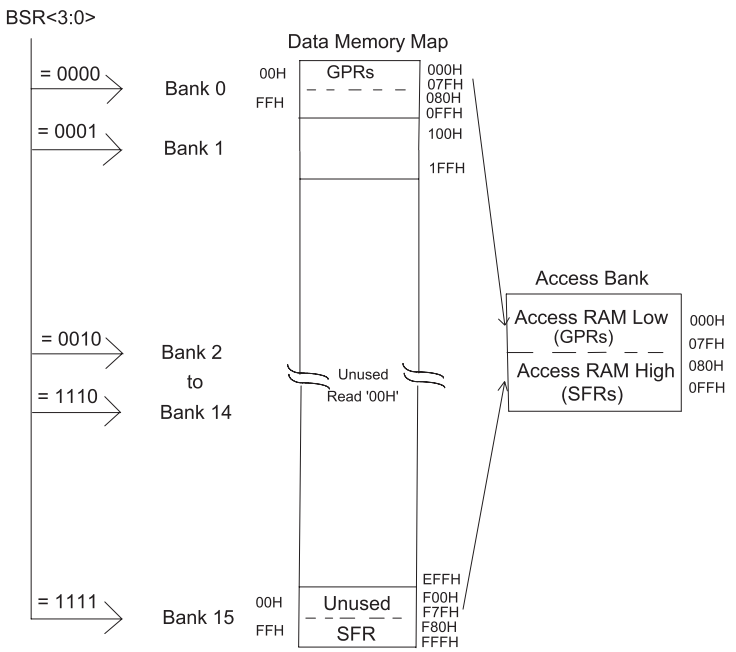


FIGURE 5.7 PIC18F4321 Data Memory Map

of the PIC18F4321 address is 12-bit wide and the contents are 8-bit wide. As mentioned before, the PIC18F data memory is implemented in SRAM. With 12-bit address, the PIC18F4321 can have a data memory of up to 4096 (2^{12}) bytes; 12-bit address (addresses from 0x000 to 0xFFF) is needed to address each location. The data memory is divided into 16 banks of 256 bytes each with a total of 4k bytes (16 x 256 = 4096 bytes = 4k bytes). Upper four bits of the 12-bit address specifies the bank number, and is contained in the low four bits of BSR with upper four bits of BSR as 0's.

The low eight bits of the 12-bit address provide the 8-bit address in a bank from addresses 0x00 to 0xFF. For example, 12-bit address 0x0FE will mean that upper four bits 0H of 0x0FE will specify bank 0 and the lower 8 bits will provide the address 0xFE. Note that the low four bits of the 8-bit register must be loaded with 0H to define bank number 0.

Figure 5.7 shows the data memory organization of the PIC18F4321. Data registers in band 0 and bank 1 with a total of 512 bytes are available in the PIC28F4321. Switching from bank 0 to bank 1 or vice versa can be accomplished using the "MOVLB K" instruction. Upon hardware reset, the programmer can access a special bank called the "Access Bank". Note that the access bank consists of the LOW 128 bytes of RAM (addresses 0x00 - 0x7F) in bank 0 called "Access RAM Low", and the last 128 bytes of RAM (addresses 0x80 - 0xFF) in bank F called "Access RAM High" as shown in Figure 5.7. Also, the programmer can access the registers in the access bank without the BSR. This provides faster execution of programs. For simplicity, the access bank LOW (addresses 00H - 7FH in bank 0) will be used as data memory in this book.

The Access RAM Low contains GPRs (General Purpose Registers) while the Access RAM High contains SFRs. GPRs are data registers used for accessing data during programming. The purpose of SFRs was already explained earlier. A list of some of these SFRs is given in Table 5.1. A complete list of all SFRs in the PIC18F along with their addresses is provided in Appendix E.

Next, a more detailed explanation of the PIC18F bank switching scheme will be provided.

Large areas of data memory require an efficient addressing scheme to make rapid access to any address possible. Ideally, this means that an entire address does not need to be provided for each read or write operation. For the PIC18F, this is accomplished with a RAM banking scheme. This divides the memory space into 16 contiguous banks of 256 bytes. Depending on the instruction, each location can be addressed directly by its full 12-bit address, or an 8-bit low-order address and a 4-bit Bank Pointer.

Most instructions in the PIC18F instruction set make use of the Bank Pointer, known as the Bank Select Register (BSR). The BSR holds the four Most Significant bits of a location's address; the PIC18F instruction contains the 8 Least Significant bits. Only the four lower bits of the BSR are implemented (BSR3:BSR0). The value of the BSR indicates the bank in data memory; the 8 bits in the instruction show the location in the bank and can be thought of as an offset from the bank's lower boundary. The relationship between the BSR's value and the bank division in data memory is shown in Figure 5.7.

In order to access a memory location from one bank to a memory location in a different bank, bank switching is required. For example, to access address F56H in bank F (specified by the upper four bits of the address) from address 150H in bank 1, the programmer must change the bank number from 1 to F. This can be accomplished using the MOVLB K instruction where K is an 8-bit number. The low four bits of K are used to specify the bank , and the upper four bits are always cleared to 0's. For example, in order to switch from bank 1 (assuming active bank) to bank F, the instruction MOVLB 0x0F can

be used. This instruction will load 0FH into BSR, and will select bank number F. All data registers in bank F will now become active.

However, sometimes the need for bank switching creates a major problem for the programmer. Obviously, programs will not work if the programmer forgets about bank switching. To facilitate access for the most commonly used data memory locations, the data memory is configured with an "Access Bank", which allows users to access a mapped block of memory without bank switching. As mentioned before, the lower half is known as the "Access RAM" and is comprised of GPRs . This upper half is also where the device's SFRs are mapped. These two areas are mapped contiguously in the Access Bank and can be addressed in a linear fashion by an 8-bit address (Figure 5.7). It is convenient to use access bank for file registers. The user does not have to worry about bank switching. Hence, one should use access bank whenever possible. Note that upon power-up, the PIC18F uses the

TABLE 5.1 Selected Special Function Registers (SFRs)

Address	**Name**	**Description**
0xFFF	TOSU	Top of stack (Upper 5 bits)
0xFFE	TOSH	Top of stack (High byte)
0xFFD	TOSL	Top of stack (Low byte)
0xFFC	STKPTR	Stack Pointer
0xFFB	PCLATU	Program Counter Latch (Upper 5 bits)
0xFFA	PCLATH	Program Counter Latch (Upper byte)
0xFF9	PCL	Program Counter Latch (Lower byte)
0xFF8	TBLPTRU	Table Pointer (Upper 5 bits)
0xFF7	TBLPTRH	Table Pointer (High byte)
0xFF6	TBLPTRL	Table Pointer (Low byte)
0xFF5	TABLAT	Table Latch
0xFF4	PRODH	Product Register (High byte)
0xFF3	PRODL	Product Register (Low byte)
0xFEF	INDF0(**1**)	Indirect File Register 0; associated with FSR0
0xFEE	POSTINC0(**1**)	Postincrement Pointer 0; uses FSR0
0xFED	POSTDEC0(**1**)	Postdecrement Pointer 0; uses FSR0
0xFEC	PREINC0(**1**)	Predecrement Pointer 0 ; uses FSR0
0xFEB	PLUSW0(**1**)	Add FSR0 to WREG and uses as pointer for data registers
0xFEA	FSR0H	File Select Register 0 (High byte)
0xFE9	FSR0L	File Select Register 0 (Low byte)
0xFE8	WREG	Working Register (Accumulator)
0xFE7	INDF1(**1**)	Indirect File Register 1; associated with FSR1
0xFE6	POSTINC1(**1**)	Postincrement Pointer 1; uses FSR1
0xFE5	POSTDEC1(**1**)	Postdecrement Pointer 1; uses FSR1
0xFE4	PREINC1(**1**)	Predecrement Pointer 1 ; uses FSR1
0xFE3	PLUSW1(**1**)	Add FSR1 to WREG and uses as pointer for data registers
0xFE2	FSR1H	File Select Register 1 (High byte)
0xFE1	FSR1L	File Select Register 1 (High byte)
0xFE0	BSR	Branch Select Register
0xFDA	FSR2H	File Select Register 2 (High byte)
0xFD9	FSR2L	File Select Register 2 (Low byte)
0xFD8	SR	Status Register

Note 1: This is not a physical register.

access bank of the file registers as the default bank. In the core PIC18F instruction set, only the MOVFF instruction fully specifies the 12-bit address of the source and target registers. The size of this instruction is two words. This instruction ignores the BSR completely when it executes. All other instructions include only the low-order address as an operand and must use either the BSR or the Access Bank to locate their target registers.

5.4 PIC18F Addressing Modes

Most instructions contain one or more operands. Some instructions have no operands. The manner in which a microcontroller specifies location(s) of operand(s) and destination addresses is called the "addressing mode". Note that an operand may be immediate data (literal), or data stored in a register or in data memory.

The PIC18F provides six addressing modes:

1. Literal or Immediate Addressing Mode
2. Inherent or Implied Addressing Mode
3. Direct or Absolute Addressing Mode
4. Indirect Addressing Mode
5. Relative Addressing Mode
6. Bit Addressing Mode

An additional addressing mode, Indexed Literal Offset, is available when the extended instruction set is enabled. However, this mode will not be described here.

5.4.1 Literal or Immediate Addressing Mode

In the *literal* or *immediate mode*, the operand data is a literal or constant data. Immediate data is part of the instruction. This means that the data follows the opcode after assembling an instruction with immediate addressing mode. Constant data can be moved into the WREG or any other specified register such as BSR. However, constant data cannot be moved into file registers using the Literal or Immediate mode. This mode can also be used by PIC18F arithmetic and logic instructions to be covered in Chapter 6.

An example is the MOVLW 0x2A instruction in which MOVLW uses immediate mode, and moves 8-bit data 2AH into the WREG register.

5.4.2 Inherent or Implied Addressing Mode

In the *inherent* or *implied mode*, instructions do not require operands. These instructions are also called *no-operand instructions*. An example of an instruction with inherent or implied mode is the DAW instruction. The DAW is a no-operand instruction. It adjusts the sum (adds 6 for BCD correction) in the WREG register stored after addition of two 8-bit packed BCD numbers. Note that the DAW instruction implicitly or inherently uses the WREG register .

5.4.3 Direct or Absolute Addressing Mode

In the *direct or absolute addressing mode*, the address is included as part of the instruction. This address specifies either a data register address in one of the banks of data SRAM or a location in the Access Bank as the data source for the instruction. Direct addressing mode specifies all or part of the source and/or destination address of the operation within the opcode itself.

An example of the direct addressing mode is MOVWF 0x50. Note that the letter F in the instruction MOVWF indicates the address of the file register in direct addressing mode, and "0x" means hexadecimal number.

The MOVWF 0x50 instruction moves the contents of the WREG register into a file register in the data SRAM whose address is 0x50. The contents of WREG are unchanged. MOVWF 0x50 uses direct address mode since address 0x50 is directly specified in the MOVWF instruction.

Note that the 12-bit address, 0x050 in the access bank, is specified by 8-bit number 0x50 in the MOVWF instruction since the upper four bits (0H) of the 12-bit address 0x050 specify the bank number 0 (access bank) in this case. The 8-bit address (0x50) is included in the MOVWF (16-bit wide) instruction. As mentioned before, in the PIC18F, addresses are specified as 8-bit numbers while the bank number is specified by the access bank or BSR.

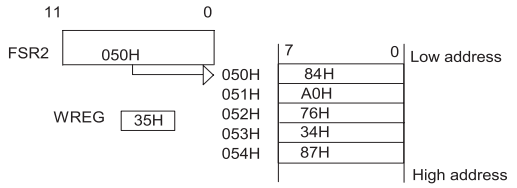
5.4.4 Indirect Addressing Mode

In *Indirect Addressing Mode*, a register is used as a pointer to an address in the data memory. In the PIC18F, three registers, namely FSR0, FSR1, and FSR2 are used for this purpose. Note that "FSR" stands for file select register. The PIC18F CPU contains these registers. As mentioned before, each of these 16-bit registers is divided into low byte and high byte as follows: FSR0 as FSR0H and FSR0L, FSR1 as FSR1H and FSR1L, FSR2 as FSR2H and FSR2L. Each of these registers holds a 12-bit memory address to point to a data memory location. Since FSR's are 16-bit wide, the 12-bit address is stored in low 12 bits (bits 0 through 11) of the FSRs with the upper four bits (bits 12 through 15) as 0's.

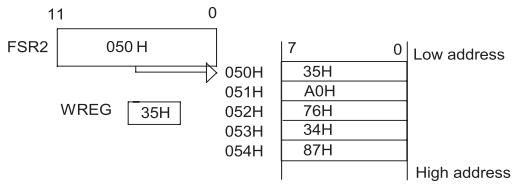
Each FSR is associated with an SFR (Special Function Register) called INDF (Indirect File) as follows: FSR0 is associated with SFR called INDF0, FSR1 is associated with SFR called INDF1, and FSR2 is associated with SFR called INDF2. The FSR's can be initialized using the LFSR (Load FSR) instruction. The following examples illustrate this:

```
LFSR    0, 0x0010       ; Load  0010H into FSR0
LFSR    1, 0x0040       ; Load 0040H into FSR1
LFSR    2, 0x0080       ; Load 0080H into FSR2
```

After initializing one of the FSR's, data can be moved into a RAM location indirectly using the associated SFR called INDF. For example, in order to move the



(a) Memory contents after execution of MOVLW and LFSR instructions in the example for indirect addressing



(b) Memory contents after execution of MOVWF in the example for indirect addressing

FIGURE 5.8 Illustration of the Indirect Addressing Mode

contents of the WREG register into a 12-bit data RAM location 050H using the FSR2 register indirectly, the following PIC18F instruction sequence can be used:

```
MOVLW   0x35     ; Move 35H into WREG
LFSR    2, 0x50  ; Initialize FSR2 with the RAM location 050H
MOVWF   INDF2    ; Move contents of WREG (35H) into a data RAM address pointed
                 ; to by FSR2 (address 050H) since INDF2 is associated with FSR2
```

The above instruction sequence loads 8-bit data, 0x35 into a 12-bit data memory location 0x050 via the WREG register. This is depicted in Figure 5.8.

The PIC18F provides the indirect addressing mode with four sub-modes as follows:

1. Indirect with postincrement mode
2. Indirect with postdecrement mode
3. Indirect with preincrement mode
4. Indirect with 8-bit indexed mode

These four sub-modes are described next. As mentioned earlier, the PIC18F includes three File Select Registers (FSR0 through FSR2). Each FSR is comprised of two 8-bit registers: FSRH and FSRL. Also, each FSR has a corresponding INDFx register (INDF0- INDF3) used for indirect addressing. In addition to these registers, the four sub-modes utilize four Special Function Registers (SFRs) namely: POSTINC, POSTDEC, PREINC, and PLUSW. These Special Function Registers use the contents of FSR0 through FSR2 to achieve the four sub-modes. The sub-modes can be used with various PIC18F instructions. The SFRs are utilized by the sub-modes as follows:

- Indirect with postincrement mode uses SFR's called POSTINC0 through POSTINC2 registers. POSTINC0 is associated with FSR0, POSTINC1 with FSR1, and POSTINC2 with FSR2.

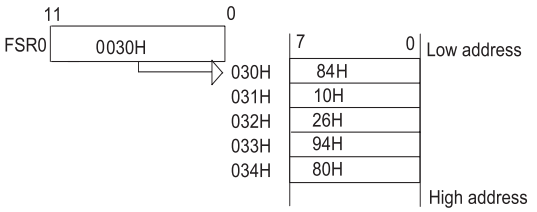
- Indirect with postdecrement mode uses SFR's called POSTDEC0 through POSTDEC2. POSDEC0 is associated with FSR0, POSDEC1 with FSR1, and POSDEC2 with FSR2.

- Indirect with preincrement mode uses SFR's called PREINC0 through PREINC2 registers. PREINC0 is associated with FSR0, PREINC1 with FSR1, and PREINC2 with FSR2.

- Indirect with 8-bit indexed mode uses SFR's called PLUSW0 through PLUSW2. PLUSW0 is associated with FSR0, PLUSW1 with FSR1, and PLUSW2 with FSR2.

Indirect with postincrement mode reads the contents of the FSR specified in the instruction, using a low 12-bit value as the address for the operation to be performed. The specified FSR is then incremented by 1 to point to the next address. As mentioned before the Special Function Register, POSTINC, is used for this purpose.

As an example, consider CLRF POSTINC0. Prior to execution of this instruction, suppose that the 16-bit contents of FSR0 are 0030H, and the 8-bit contents of the data memory location addressed by 12-bit address 030H are 84H. After execution of the instruction, CLRF POSTINC0, the contents of address 030H, and will be cleared to 00H, and the contents of FSR0 will be incremented by 1 to hold 0031H. This may be used as a pointer to the next data. This is depicted in Figure 5.9. Note that all addresses and data are chosen arbitrarily.



(a) Memory contents before execution of CLRF POSTINC0.

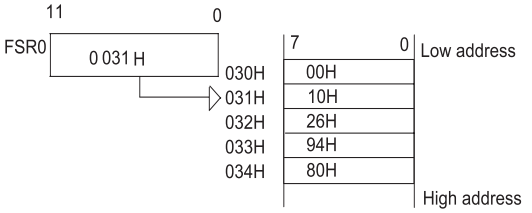


FIGURE 5. 9 Illustration of Indirect with Postincrement Mode

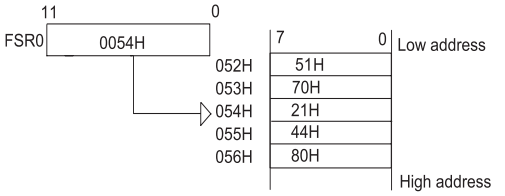
```
        MOVLW   D'20'       ; Move 20 decimal into WREG
        MOVWF   0x10        ; Initialize Counter 0x10 with 20 decimal
        LFSR    0,0x30      ; Initialize pointer FSR0 with starting address 0x030
REPEAT  CLRF    POSTINC0    ; Clear a location to 0 and increment FSR0 by 1
        DECF    0x10,F      ; Decrement Counter by 1
        BNZ     REPEAT      ; Branch to REPEAT if Zero flag = 0, otherwise
                            ; go to the next instruction
```

FIGURE 5.10 Instruction sequence for illustrating the postincrement mode

The postincrement mode is typically used with memory arrays stored from LOW to HIGH memory locations. For example, to clear 20 bytes starting at data memory address 030H and above, the instruction sequence in Figure 5.10 can be used. In Figure 5.10, MOVLW D'20' moves 20 decimal into WREG while MOVWF 0x10 moves the contents of WREG (20 decimal) into address 010H. This will initialize the counter register with 20 decimal. The LFSR 0,0x30 loads FSR0 with 0030H; 030H is the address of the first byte in the array to be cleared to 0. The CLRF POSTINC0 clears the contents of the data memory addressed by FSR0 to 0. This is because POSTINC0 is associated with FSR0. Since the 16-bit contents of FSR0 are 0030H, contents addressed by the the low 12 bits (030H) of FSR0 are cleared to 0. The DECF 0x10,F decrements the contents of data register 010H by one and then places the result in the data register 010H. After the first pass, data register 010H will contain 19 decimal.

The BNZ REPEAT instruction checks if Z flag in the flag register is 0. Note that after first iteration, $Z = 0$ since the contents of counter are nonzero (19) after execution of DECF. The program branches to label REPEAT, and the loop will be performed 20 times clearing 20 bytes of the array to 0's.

Indirect with postdecrement mode reads the contents of the FSR specified in the instruction, and uses a low 12-bit value as the address for the operation to be performed. The specified FSR is then decremented by 1. The Special Function Register, POSTDEC, is used for this purpose.



(a) Memory contents before execution of CLRF POSTDEC0.

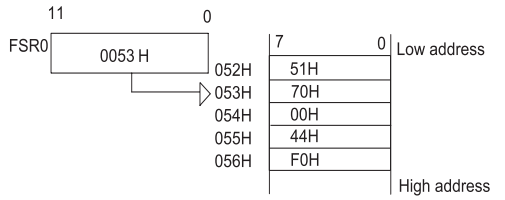


FIGURE 5.11 Illustration of Postdecrement mode

```
        MOVLW   D'100'      ; Move 100 decimal into WREG
        MOVWF   0x20        ; Initialize Counter reg (0x20) with 100 decimal
        LFSR    0,0x44      ; Initialize pointer FSR0 with starting address 044H
REPEAT  CLRF    POSTDEC0    ; Clear a location to 0 and  decrement FSR0 by 1
        DECF    0x20,F      ; Decrement Counter by 1
        BNZ     REPEAT      ; Branch to  REPEAT if  Zero flag = 0, otherwise
                            ; go to the next instruction
```

FIGURE 5.12 Instruction sequence for illustrating posdecrement mode

As an example, consider CLRF POSTDEC0. Prior to execution of this instruction, suppose that the 16-bit contents of FSR0 are 0054H, and the 8-bit contents of the data register addressed by 12-bit address 054H are 21H. After execution of the instruction, CLRF POSTDEC0, the contents of address 054H will be cleared to 00H and the contents of FSR0 will be decremented by 1 to hold 0053H. This is depicted in Figure 5.11. Note that all addresses and data are chosen arbitrarily.

The postdecrement mode can be used with arrays stored from HIGH to LOW addresses. For example, to clear 100 bytes starting at address 044H and below, the instruction sequence in Figure 5.12 can be used. Note that all instructions are self-explanatory from the example of Figure 5.10.

Indirect with preincrement mode First, the contents of the FSR are incremented by 1 to contain the next address. The low 12 bits of the FSR are then used as the address for the operation to be performed. The Special Function Register, PREINC, is used for this purpose.

As an example, consider CLRF PREINC0. Prior to execution of this instruction, suppose that the 16-bit contents of FSR0 are 0030H, and the 8-bit contents of the data register addressed by 12-bit address 031H are 84H. After execution of the instruction, CLRF PREINC0, the contents of FSR0 will be incremented by one to hold 0031H. The contents of data register with address 031H will be cleared to 00H.

Indirect with 8-bit indexed mode adds the contents of the FSR specified in the instruction with the 8-bit contents of the WREG register. The sum is used as an address of a data

register in the RAM. The instruction is then executed using this data. The contents of the specified FSR and WREG are unchanged. As an example, consider CLRF PLUSW2. Prior to execution of this instruction, suppose that the 16-bit contents of FSR2 are 0020H, the 8-bit contents of WREG are 04H, and the 8-bit contents of address 024H in data RAM are 37H. After execution of CLRF PLUSW2, the contents of the data register 024H will be cleared to 00H. The contents of FSR2 and WREG are 0020H (unchanged) and 04H (unchanged) respectively.

The indirect with 8-bit indexed mode can be used for code conversion. Two 8-bit ports (Port C and Port D) of the PIC18F will be used to illustrate this example. Assume that a PIC18F is interfaced to an ASCII keyboard via port C and to an EBCDIC printer via port D. Suppose that it is desired to enter numerical data via the ASCII keyboard and then print them on the EBCDIC printer.

Note that numerical data entered into the PIC18F via the keyboard will be in ASCII code. Since the printer only understands EBCDIC code, an ASCII-to-EBCDIC code conversion program is required. As discussed in Chapter 1, the ASCII codes for numbers 0 through 9 are 30H through 39H, while the EBCDIC codes for numbers 0 to 9 are F0H to F9H.

An array can be stored in the access bank starting at address 030H; EBCDIC code F0H (decimal 0) at address 030H, EBCDIC code F1H (decimal 1) at address 031H, and so on. Now, suppose that '1' is pushed on the ASCII keyboard. The PIC18F can input this data via PORT C into WREG as 31H (ASCII for 1). The EBCDIC printer will print '1' if the PIC18F outputs F1H to Port D. This can be accomplished by initializing one of the FSR's (assume FSR2 in this example) with 0000H and then execute MOVFF instruction using indirect with 8-bit indexed mode as follows:

```
LFSR    2, 0x0000          ; Load 0000H into FSR2
MOVFF   PLUSW2,PORTD       ; Add WREG to FSR2, move
                           ; the byte content of that address to Port D
```

In the above instruction sequence, since the content of WREG, in this example, is 31H, the instruction MOVFF PLUSW2,PORTD will output the contents of address 031H (F1H), to the EBCDIC printer. The printer will then print '1'.

Addressing modes for TBLRD* and TBLWT* instructions In the PIC18F, the address and data sizes of the program memory and data memory are not compatible. The 16-bit contents of the program memory are addressed by 21-bit addresses while the 8-bit contents of the data memory (data registers) are addressed by 12-bit addresses. In order to transfer data between program and data memories, the PIC18F is provided with two Special Function Registers, namely 21-bit TBLPTR (Table Pointer) and 8-bit TABLAT (Table Latch). The TABLPTR is used as a pointer for program memory. The TABLAT, on the other hand, is used to hold a byte to be transferred from program memory to data memory.

Two instructions, namely TBLRD* and TBLWT*, are provided with the PIC18F in order to perform data transfers between program and data memories. The TABLRD* instruction reads a byte from the program memory into the TABLAT register that can be moved into a data register. The TBLWT*, on the other hand, writes a data byte from the TABLAT register (already moved from a data register) into the program memory. The TBLRD* and TBLWT* instructions can use register indirect, postincrement, postdecrement, and preincrement modes. These instructions along with the addressing modes can be used to

transfer a block of data between program memory and data memory. The concept of Table Pointers for transferring data between program memory and data memory will be covered in details in Chapter 6.

5.4.5 Relative Addressing Mode

All conditional and one unconditional branch (BRA) instructions in the PIC18F use relative addressing mode.

The conditional branch instructions in the PIC18F are based on four flags, namely C, Z, OV, and N. Each conditional branch instruction specifies an 8-bit offset. This offset is a two's complement signed number. This means that for forward branching, the range of the offset value is from 00H to 7FH. For backward branching, this range varies from 80H to FFH. Since conditional branch instructions are 16-bit wide in the PIC18F, the PC (Program Counter) is incremented by 2 to point to the next instruction while executing the conditional branch instruction. The offset is multiplied by 2 and then added to PC+2 to find the branch address if the condition is true. Note that the offset is multiplied by 2 since the contents of the PC must always be an even number for 16- and 32-bit instruction lengths.

As an example, consider BNC 0x03. Note that BNC stands for "Branch if no carry". If the C (carry flag) in the Status register is 0, then the PC is loaded with the (PC + 2 + 03H x 2). When the PIC18F executes the BNC instruction, the PC points to the next instruction. This means that if BNC is located at address 0050H in program memory, the PC will contain 0052H (PC + 2) when the PIC18F executes BNC. Hence, if C = 0, then after execution of the BNC 0x03 instruction, the PC will be loaded with address 0058H (0052H + 03H x 2). Hence, the program will branch to address 0058H which is 6 steps forward relative to the current contents of PC. This is called "Relative Addressing Mode". Note that the Relative mode is useful for developing position independent code.

The unconditional branch instruction, BRA (Branch Always), also uses the relative addressing mode. However, the BRA d instruction unconditionally branches to (PC + 2 + d x 2) where offset 'd' is a signed 11-bit number specifying a range from -1024_{10} to $+1023_{10}$ with 0 being positive. Note that "BRA d" is assembled as 16-bit. The upper 4-bit (bits 12-15) is the opcode, bit 11 is always 0, and the low 11-bit (bits 0-10) is the offset, d. The assembler computes the target address by sign-extending 2xd and then adding it to PC + 2. For example, consider BRA 0x05 is stored at location 0040H in the program memory. This means that the PC will contain 0042H (PC + 2) when the PIC18F executes BRA. Hence, after execution of the BRA 0x05 instruction, the PC will be loaded with address 004CH (0042H + 05H x 2). Hence, the program will branch to address 004CH which is 10 (A_{16}) steps forward relative to the current contents of PC. Next, consider the following instruction sequence:

```
        ORG   0x100
HERE    BRA   HERE
```

The machine code for the above instruction is 1101011111111111_2 ($D7FF_{16}$). Note that 1101_2=0xD is the opcode and 11111111111_2 (Eleven 1's sign-extended = -1_{10}) is the offset. During execution of the BRA instruction, the PC points to 0x102. The target branch address = (PC + 2 + d x 2) = 0x102 + (-1) x 2 = 0x100. The instruction HERE BRA HERE unconditionally branches to address 0x100. This is equivalent to HALT instruction in other processors.

To illustrate the concept of relative branching, consider the following PIC18F disassembled instruction sequence along with the machine code (all numbers in hex):

```
                                    1:              #INCLUDE<P18F4321.INC>
                                    2:              ORG     0x00
0000    0E02    MOVLW   0x2     3:  BACK    MOVLW   0x02
0002    0802    SUBLW   0x2     4:          SUBLW   0x02
0004    E001    BZ      0x8     5:          BZ      DOWN
0006    0E04    MOVLW   0x4     6:          MOVLW   0x04
0008    0804    SUBLW   0x4     7:  DOWN    SUBLW   0x04
000A    E0FA    BZ  0           8:          BZ      BACK
000C    0003    SLEEP           9:          SLEEP
```

Note that all instructions, addresses, and data are chosen arbitrarily. The first branch instruction, BZ DOWN (line 5) at address 0x0004, has a machine code 0xE001. Upon execution of the instruction BZ (branch if Z-flag = 1), the PIC18F branches to label DOWN if Z = 1 (condition true); the PIC18F executes the next instruction at address 0x0006 if Z = 0 (condition false).

The BZ instruction uses the relative addressing mode. This means that DOWN is a positive number (the number of steps forward relative to the current program counter) indicating a forward branch. The machine code 0xE001 means that the op-code for BZ is 0xE0 and the relative 8-bit signed offset value is 0x01 (+1). This is a positive value indicating a forward branch. Note that while executing BZ DOWN at address 0x0004, the PC points to address 0x0006 since the program counter is incremented by 2. This means that the program counter contains 0x0008. The offset 0x01 is multiplied by 2 and added to address 0x0006 to find the target branch address where the program will jump if Z = 1.

The branch address can be calculated as follows:

```
  0x0006   =  0000 0000  0000 0110
+ 0x0002   =  0000 0000  0000 0010   (0x01 is multiplied by 2, and sign-
                                      extended to 16 bits)
-------------------------------------------------------------------------
                    0000 0000  0000  1000 = 0x0008
```

Hence, the PIC18F branches to address 0x0008 if Z = 1. This can be verified in the instruction sequence above.

Next, consider the second branch instruction, BZ BACK (line 8). Upon execution of BZ BACK, the PIC18F branches to label BACK if Z = 1 (condition true); if Z = 0 (condition false), the PIC18F executes the next instruction. The machine code this instruction at address 0x000A is 0xE0FA, where 0xE0 is the op-code and 0xFA is the signed 8-bit offset value. The offset is represented as an 8-bit two's complement number. Since 0xFA is a negative number (-6_{10}), this is a backward jump. Note that while executing BZ BACK at address 0x000A, the PC points to address 0x000C since the program counter is incremented by 2. This means that the program counter contains 0x000C. The offset -6 is multiplied by 2, and then added to 0x000C to find the address value where the program will branch if Z = 1. The branch address is calculated as follows:

```
   0x000C =  0000 0000  0000 1100
+  0xFFF4 =  1111  1111  1111 0100 (0xFA is multiplied by 2 and then sign-
                                     extended to 16  bits)
-------------------------------------------------------------------------
              1  0000 0000  0000 0000  = 0x0000
            ↗
Ignore final carry
```

The branch address is 0x0000, which can be verified in the instruction sequence above. As mentioned in Chapter 1, in order to add a 16-bit signed number with an 8-bit signed number, the 8-bit signed number must first be sign-extended to 16 bits. The two 16-bit numbers can then be added. Any carry resulting from the addition must be discarded. This will provide the correct answer.

5.4.6 Bit Addressing Mode

The instructions using the Bit addressing mode directly specify a single bit to be operated on. For example, BCF 0x10,3 will clear bit 3 to 0 in the data register addressed by 0x10. For example, if the contents of the data register 0x10 are 01111000_2 (78H), then after execution of BCF 0x10,3, the contents of data register 0x10 are 01110000_2(70H). Note that, in this addressing mode, bit 3 to be cleared to zero is directly specified in the instruction, BCF 0x10,3. Three bits in the instructions using bit addressing mode are used to specify the bit number from 0 to 7. These instructions will be covered in Chapter 6.

QUESTIONS AND PROBLEMS

5.1 What is the size of the Program Counter in the PIC18F? What is the maximum size of the PIC18F program memory? Justify your answer.

5.2 What type of memory is used for Program memory in the PIC18F?

5.3 What is the maximum size of the data memory in the PIC18F? How many bits are needed to address data memory? Justify your answer.

5.4 What type of memory is used for data memory in the PIC18F?

5.5 What are the maximum sizes of the Program and data memories in the PIC18F4321? What is the main purpose of the EEPROM?

5.6 What is the maximum clock frequency of the PIC18F4321?

5.7 What is the purpose of the CCP module in the PIC18F?

5.8 Summarize the basic features of the PIC18F pipeline.

5.9 Assume PIC18F microcontroller. What is the basic difference between:
(a) Program Counter (PC) and Function Select Registers (FSRs)?
(b) Working Register (WREG) and Instruction Register (IR)?

5.10 Assume that the PIC18F is currently executing a 16-bit instruction addressed by 0150H. What are the current contents of the Program Counter?

5.11 What is the PIC18F instruction for switching from address 0x120 in bank 1 to address 0xF20 in bank F in the data memory?

5.12 What are the addresses of the PIC18F Status and Stack Pointer registers?

5.13 Is the stack in the PIC18F implemented in the data SRAM? Discuss briefly how the hardware stack is implemented in the PIC18F.

5.14 Find the sign, carry, zero and overflow flags in the PIC18F for the following arithmetic operation: 6AH - 6AH.

5.15 What is the advantage of incorporating the “Access Bank” in the PIC18F?

5.16 What is meant by addressing mode?

5.17 Identify the addressing modes for the following instructions:
(a) NOP (b) MOVLW 0x2A (c) CLRF PREINC2

5.18 Using the PIC18F instructions described in section 5.4.4, write an instruction sequence using indirect with a preincrement mode to clear 50 bytes in the data memory starting at address 010H and above. Use register addresses of your choice.

6

ASSEMBLY LANGUAGE PROGRAMMING WITH THE PIC18F: PART 1

In this chapter we provide the first part of the PIC18F's instruction set. We also cover the fundamental concepts associated with assembly language programming relating to the PIC18F microcontroller. Topics include basics of the PIC18F MPLAB assembler, PIC18F instruction format, instruction set, and assembly language programming.

6.1 Introduction to the PIC18F MPLAB assembler

As mentioned in Chapter 3, the assembly language program is translated into binary via a program called an *assembler*. In this section, we will review the concepts described in Chapter 3 relating them to the PIC18F MPLAB assembler. For convenience, some of the topics will be repeated.

The assembler reads each instruction of an assembly language program as ASCII characters and translates them into the respective binary op-codes. For example, the PIC18F assembler translates the SLEEP (places the PIC18F in sleep mode; same as HALT instruction in other processors) instruction into its 16-bit binary op-code is 0000 0000 0000 0011 (0003 in hex), as depicted in Table 6.1.

A notable advantage of the assembler is its ability to handle address computation. Most programs use addresses within the program as data storage or as targets for jumps or calls. When programming in machine language, these addresses must be calculated by hand. The assembler solves this problem by allowing the programmer to assign a symbol to an address. The programmer may then reference that address elsewhere by using the symbol. The assembler computes the actual address for the programmer and fills it in automatically.

One can obtain hands-on experience with a typical assembler for a microcontroller by downloading it from the Internet. The MPLAB assembler/simulator is used to assemble and debug all PIC18F assembly language programs in this book. It can be downloaded free of charge from the web site: www.microchip.com.

As mentioned in Chapter 4, each line in an assembly language program includes four fields:

- Label field
- Mnemonic or op-code field
- Operand field
- Comment field

The rules associated with the MPLAB will be used in explaining the meaning of these fields. The assembler ignores the comment field but translates the other fields. The

Microcontroller Theory and Applications with the PIC18F, Second Edition. M. Rafiquzzaman.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

TABLE 6.1 Conversion of SLEEP instruction into its Binary Op-Code

Assembly Code	Binary Form of ASCII Codes as Seen by the Assembler	Binary Op-Code Created by the MPLAB PIC18F Assembler
S	0101 0011	
L	0100 1000	0000 0000 0000 0011
E	0100 0101	
E	0100 0101	
P	0101 0000	

label field must start with an alphabetic character, and can have a maximum length of 32 characters.

The assembler must know where one field starts and another ends. The MPLAB, like most assemblers, allows the programmer to use a special symbol or delimiter to indicate the beginning or end of each field. Typical delimiters used are spaces, commas, and semicolons:

- Spaces are used between fields.
- Commas (,) are used between addresses in an operand field.
- A semicolon (;) is used before a comment.
- No symbol is used after a label.

To handle numbers, most assemblers consider all numbers as decimal numbers unless specified otherwise. The MPLAB assembler, however, recognizes a number without any prescript or postscript as a hexadecimal number. Most assemblers, including the MPLAB assembler, will also allow other number systems, including hexadecimal. For example, with the MPLAB assembler, the user can define a hexadecimal number usually in three ways using:

- 0x before the number
- H after the number
- default (without any prescript or postscript)

This means that 60 in hexadecimal can be represented as 0x60, 60H, or 60. A '0' is used by most assemblers if the first digit of the hexadecimal number is a letter; otherwise, the assembler will generate an error. This is done to distinguish between numbers and labels. However, the MPLAB assembler does not require a 0 to be placed in front of the number if the first digit of the hexadecimal number is a letter. These symbols (0x or H or none) will be used interchangeably throughout the book to represent hexadecimal numbers.

The MPLAB uses D before a 'number' to specify a decimal number. For example, decimal number 60 can be represented as D'60'. A binary number is specified by the MPLAB using B before the 'Number'. For example, 8-bit binary number 01011100 can be represented by the MPLAB as B'01011100'.

ASCII characters are represented using MPLAB by the symbols ' ' or A ' '. For example, PIC18F is represented as 'PIC18F' or A'PIC18F' to be recognized as ASCII characters in MPLAB.

Assemblers use pseudoinstructions or directives to make the formatting of the edited text easier. These directives are not translated directly into machine language instructions. Typical assembler directives are discussed in the following.

ORIGIN (ORG) The directive ORG specifies the starting address of a program or data. For example, after assembling the following statements, the MPLAB will place the assembled code for MOVLW 0x50 starting at address 0x100:

```
ORG     0x100
MOVLW   0x50
```

Equate (EQU) The EQU assigns a value in its operand field to an address in its label field. This allows the user to assign a numerical value to a symbolic name. The user can then use the symbolic name in the program instead of its numerical value. A typical example of EQU is START EQU 0x20, which assigns the value 20 in hexadecimal to the label START.

To illustrate the EQU directive, consider the following instruction sequence:

```
TEST    EQU     0x20
        MOVLW   TEST
        MOVWF   TEST
```

The instruction MOVLW TEST, in the above, moves the constant 0x20 into WREG. The instruction MOVWF TEST, on the other hand, moves the contents of WREG into the data register with address 0x20.

Define Byte (DB) The directive DB is generally used to set a memory location to a certain byte value. For example,

```
START   DB      0x45
```

will store the data value 45 hex to the address START. The DB directive can be used to generate a table of data as follows:

```
ORG     0x70
DB      0x20,0x30,0x40,0x50
```

In this case, 20 hex is the first data of the memory location 70 hex; 30 hex, 40 hex, and 50 hex occupy the next three memory locations. Therefore, the data in memory will look like this:

```
70      20
71      30
72      40
73      50
```

Define Word (DW) The directive DW is typically used to assign a 16-bit value to two memory locations. For example,

```
ORG     0x50
DW      0x4AC2
```

will assign C2 to location 50 hex and 4A to location 51 hex. It is assumed that the assembler will assign the low byte first (C2) and then the high byte (4A). The DW directive can be used to generate a table of 16-bit data as follows:

```
ORG     0x80
DW      0x5000,0x6000, 0x7000
```

In this case, the three 16-bit values 5000 hex, 6000 hex, and 7000 hex are assigned to memory locations starting at the address 80 hex. That is, the array would look like this:

80	00
81	50
82	00
83	60
84	00
85	70

INCLUDE The directive INCLUDE includes source code or a file from the MPLAB library for a specific device. This will allow the MPLAB to assemble a program for that device. Note that using INCLUDE <P18F4321.INC> at the beginning of a program will add appropriate files from the MPLAB library required to assemble the program.

For example, the INCLUDE <P18F4321.INC> includes files which will insert SFR addresses during the assembly. This will allow the programmer to use variable names rather than their physical addresses. The programmer can use variable names such as PORTA in the program, and the assembler will insert its SFR address (F80H) whenever it will encounter PORTA in the program. This is very convenient for the programmer.

END The END directive indicates the end of assembly language program.

Let us now explain how one can write a typical PIC18F assembly language program. A simple program for adding two 8-bit numbers written in PIC18F assembly language is provided below:

```
Label Field   Mnemonic Field   Operand Field        Comment Field
-----------------------------------------------------------------
              INCLUDE          <P18F4321.INC>
SUM           EQU              0x40
              ORG              0x100          ; STARTING ADDRESS
              MOVLW            0x02           ; MOVING 2 INTO WREG
              ADDLW            0x05           ; ADDING WREG and 5
              MOVWF            SUM            ; STORE RESULT IN SUM
              SLEEP                           ; HALT
              END                             ; END OF PROGRAM
```

Let us explain the above program. INCLUDE <PIC18F4321.INC> at the beginning of the program will add appropriate files from the MPLAB library required to assemble the program.

SUM EQU 0x40 assigns 40 (hex) to label SUM. ORG 0x100 assembles the program at address 100 (hex). MOVLW 0x02 moves 02 (hex) into WREG. ADDLW 0x05 adds the contents of WREG with 05 (hex), and stores the 8-bit result, 07 (hex) in WREG. The instruction MOVWF SUM stores the 8-bit contents of WREG (07 hex) in file register SUM with address 40 (hex). The SLEEP instruction halts the program. Finally, the MPLAB assembler directive "end" indicates the end of the program. Note that the directive "end" must be present at the end of all PIC18F assembly language programs.

The assembly language program described above called a *source file*, contains all the instructions required to execute a program. It should be mentioned that the source code for the MPLAB assembler is not case sensitive. However, it is a good idea to type the

source program in all uppercase or all lower-case letters. Also, it is a good practice to use the TAB key between two fields while typing the source code. This will enhance the visual appearance of the program.

The assembler converts the source file into an object file containing the binary codes or machine codes that the PIC18F will understand. In typical assemblers including the MPLAB (Microchip's PIC18F assembler), the source file must be stored with a file extension called .ASM. Suppose that the programmer stores the source file as SUM.ASM. To assemble the program, the source file SUM.ASM is presented as input to the assembler. The assembler typically generates two files: SUM.OBJ (object file) and SUM.LST (list file).

SUM.OBJ is an *object file*, a binary file containing the machine code and data that correspond to the assembly language program in the source file (SUM.ASM). The SUM.LST is a *list file* which shows how the assembler interprets the source file SUM.ASM. The list file may be displayed on the screen. The source file SUM.ASM is assembled using the MPLAB assembler. The SUM.LST file is provided below:

```
                                1:          INCLUDE <P18F4321.INC>
                                2: SUM      EQU     0x40
                                3:          ORG     0x100   ; STARTING ADDRESS
0100 0E02 MOVLW 0x2             4:          MOVLW   0x02    ; MOVING 2 INTO WREG
0102 0F05 ADDLW 0x5             5:          ADDLW   0x05    ; ADDING WREG and 5
0104 6E40 MOVWF 0x40            6:          MOVWF   SUM     ; STORE RESULT
0106 0003 SLEEP                 7:          SLEEP           ; HALT
```

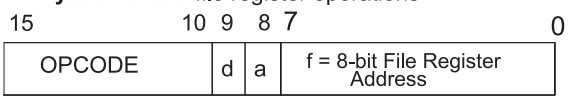
Note that the assembled code shown on the left above is in hex. ORG 0x100 (line 3) generates the starting address 0100 in hex. The machine code (0E02 hex) for the first instruction, MOVLW 0x2, is stored at the address 0100 (hex). Since this instruction takes 16-bit (two bytes), the machine code for the next instruction, ADDLW 0x5, starts at address 0102 (hex). Similarly, the machine codes for MOVWF SUM and SLEEP start at addresses 0104 (hex) and 0106 (hex) respectively. Note that the comment fields are not translated by the MPLAB assembler.

When a large program is being developed by a group of programmers, each programmer may write only a portion of the whole program. The individual program parts must be tested and assembled to ensure their proper operation. When all portions of the program are verified for correct operation, their object files must be combined into a single object program using a *Linker*, a program that checks each object file and finds certain characteristics, such as the size in bytes and its proper location in the single object program. The linker also resolves any problems with regard to cross-references to labels. A *library* of object files is typically used to reduce the size of the source file. The library files may contain frequently used subroutines and/or sections of codes. Rather than writing these codes repeatedly in the source file, a special pseudo instruction is used to tell the assembler that the code must be inserted by the linker at linking time. When linking is completed, the final object file is called an executable (.EXE) file. Finally, a program called a *Loader* can be used to load the .EXE file in memory for execution.

Appendix F provides a tutorial showing a step-by-step procedure of assembling and debugging a PIC18F assembly language program using Microchip MPLAB PIC18F assembler/debugger.

TABLE 6.2 General Format for Instructions

Byte-oriented file register operations



Example Instruction

ADDWF f, d, a

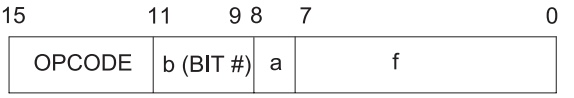
d = 0 for result destination to be WREG register

d = 1 for result destination to be file register (f)

a = 0 means Access Bank, a = 1 means BSR to select the bank

f = 8-bit file register address

Bit-oriented file register operatios



BSF f, b, a

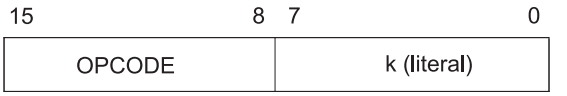
b = 3-bit position of bit in file register (f)

a = 0 to force Access Bank

a = 1 for BSR to select bank

f = 8-bit file register address

Literal operations

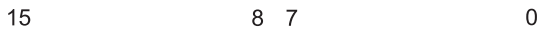


MOVLW 0x2A

k = 8-bit immediate value

Control operations

Branch operation



6.2 PIC18F Instruction Format

The instruction format is specified by the operation performed. The instruction set is grouped into four basic categories based on the operations:

- Byte-oriented operations
- Bit-oriented operations
- Literal operations
- Control operations

Table 6-2 lists these operations.

Most byte-oriented instructions have three operands:

1. The file register (specified by ‘f’)
2. The destination of the result (specified by ‘d’)
3. The access memory (specified by ‘a’)

The file register designator 'f' specifies which file register is to be used by the instruction. 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. If a = 0, the file register is the access bank. On the other hand, if a = 1, the BSR specifies the data memory bank.

For example, Consider ADDWF 0x04, 0, 0. This instruction adds the contents of WREG register with the contents of File register 0x04 and stores the result in WREG. By inspecting ADDWF 0x04, 0, 0, and comparing with the byte-oriented format of Table 6.2, d = 0, a = 0, and f = 0x04. Since the 6-bit opcode for ADDWF is 001001, the binary 16-bit code for ADDWF 0x04, 0, 0 is 0010010000000100 (2404H).

All bit-oriented instructions have three operands:

1. The file register (specified by 'f')
2. The bit in the file register (specified by 'b')
3. The accessed memory (specified by 'a')

The bit field designator 'b' selects the number of the bit affected by the operation, while the file register designator 'f' represents the file address in which the bit is located. If a = 0, the file register is the access bank. On the other hand, if a = 1, the BSR specifies the data memory bank. For example, consider BSF 0x27, 5, 0. This instruction sets bit 5 to one in file register 0x27 in the access bank and comparing with the bit-oriented format of Table 6.2, b = 101, a = 0, and f = 0x27. Since the 4-bit opcode for BSF is 1000, the binary 16-bit code for BSF 0x27, 5, 0 is 1000101000100111 (8A27H).

As an example of a literal instruction, consider MOVLW 0x2A. This instruction moves 0x2A into the WREG register. Comparing with the literal-oriented format of Table 6.2, k = 00101010 (0x2A). Since the 8-bit opcode for MOVLW is 00001110, the binary 16-bit code for MOVLW 0x2A is 0000111000101010 (0E2AH).

An example of the control instructions includes conditional branch instructions as follows:

BZ n where 'n' is a signed 8-bit offset (specified by 'n')

As an example, consider BZ 0x04 where 04 (hex) is the offset (n). This instruction branches to an address (PC+2+ 2 x 4) if Z = 1; otherwise, it executes the next instruction. Since the 8-bit opcode for BZ is 11100000 and n = 00000100 (0x04), the binary 16-bit code for BZ 0x04 is 1110000000000100 (E004H).

Most PIC18F instructions are 16-bit (word) wide; only four instructions are double-word instructions. These instructions were made double-word to contain the required information in 32 bits. The double-word instructions execute in two instruction cycles.

6.3 PIC18F Instruction Set

The PIC18F instruction set contains a total of 75 core instructions. In addition, the PIC18F provides an extended set of 8 new instructions. The PIC18F instruction set is highly orthogonal. This means that most instructions can use all addressing modes with data of any type.

The PIC18F instructions can be classified into ten groups as follows:

1. Data movement instructions
2. Arithmetic instructions
3. Logic instructions

4. Rotate instructions
5. Bit manipulation instructions
6. Jump/Branch instructions
7. Test, Compare, and Skip instructions
8. Table Read/Write instructions
9. Subroutine instructions
10. System control instructions

Instruction groups 1 through 5 are covered in this chapter. Instruction groups 6 through 10 are included in chapter 7. Table 6.3 lists the instructions along with the status flags. Appendix C provides a summary of the PIC18F instruction set (alphabetical order). A detailed description of the PIC18F instructions are included in Appendix D.

It should be mentioned that there are several PIC18F instructions that specify the destination as the WREG or a file register. For example, consider ADDWF F, d, a where 'F ' is the file register , 'd' defines the destination bit, and 'a' specifies the bank. This instruction adds the contents of the specified file register, 'F ' with the contents of WREG. If d = 0, then the result is stored in the WREG register. However, if d = 1, then the result is placed in the file register, 'F ' specified in the instruction. Note that a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank. As mentioned before, the file (data) register can be one of the General-Purpose Registers (GPRs) or one of the Special Function Registers (SFRs) in the access bank.

For example, ADDWF 0x18, 0, 0 will add the contents of file register, 0x18 with the contents of WREG. The result will be placed in WREG since d = 0. The instruction, ADDWF 0x18, 1, 0 will also add the contents of file register 0x18 with the contents of WREG. But the result will be stored in memory location 0x18 since d = 1.

Note that for instructions such as ADDWF F, d, a, the MPLAB assembler allows the PIC18F programmer to use 'W' instead of '0' and 'F' instead of '1' as far as 'd' is concerned. This means that the programmer, in this case, can use ADDWF 0x18, W, 0 instead of ADDWF 0x18, 0, 0 and ADDWF 0x18, F, 0 instead of ADDWF 0x18, 1, 0. Also, as mentioned in Chapter 5, upon power-up, the PIC18F uses the access bank. Note that the purpose of access bank in general is for fast access to any of the selected location in the access bank. This is because the BSR is not required for the access bank, and only an 8-bit rather than 12-bit data memory address is required.

In the lower half of the access bank, a = 0, and the file (data) registers are used as the default bank with addresses 0x00 through 0x7F. Hence, we will not specify 'a' in the instruction (assuming the access bank with a = 0). Also, for better clarity, the MPLAB assembler uses W or F instead of 0 or 1 for 'd'. This means that ADDWF 0x18, W will be used instead of ADDWF 0x18, 0,0 and ADDWF 0x18, F will be used instead of ADDWF 0x18,1,0. No bank switching will be used in all the programming examples.

In the following, the brackets [ ] are used to indicate the contents of a register or a data memory location. For example, [WREG] will mean the contents of WREG. Also, all numbers are chosen arbitrarily in order to illustrate the instructions using numerical examples. In the PIC18F, the term "file register" can be one of the GPR's in the low access bank (addresses 0x00 to 0x7F in bank 0) or one of the SFR's in high access bank (addresses 0x80 to 0xFF in bank F).

TABLE 6.3 PIC18F Instructions and the status flags

Instruction	N	OV	Z	DC	C
ADDLW, ADDWF, ADDWFC	✓	✓	✓	✓	✓
ANDLW, ANDWF	✓	–	✓	–	–
BC, BNC, BZ, BNZ, BN, BNN	–	–	–	–	–
BOV, BNOV, BRA	–	–	–	–	–
BCF, BSF, BTG	–	–	–	–	–
BTFSC, BTFSS	–	–	–	–	–
CALL	–	–	–	–	–
CLRF	–	–	✓	–	–
CLRWDT	–	–	–	–	–
COMF	✓	–	✓	–	–
CPFSEQ, CPFSGT, CPFSLT	–	–	–	–	–
DAW	–	–	–	–	–
DECF	✓	✓	✓	✓	✓
DECFSZ, DECFSNZ	–	–	–	–	–
GOTO	–	–	–	–	–
INCF	✓	✓	✓	✓	✓
INCFSZ, INCFSNZ	–	–	–	–	–
IORLW, IORWF	✓	–	✓	–	–
LFSR	–	–	–	–	–
MOVF	✓	–	✓	–	–
MOVFF, MOVWF, MULWF	–	–	–	–	–
MOVLB, MOVLW	–	–	–	–	–
NEGF	✓	✓	✓	✓	✓
NOP	–	–	–	–	–
POP, PUSH	–	–	–	–	–
RCALL, RETFIE,	–	–	–	–	–
RESET	✓	✓	✓	✓	✓
RETLW, RETURN,	–	–	–	–	–
RLCF, RRCF	✓	–	✓	–	✓
RLNCF, RRNCF	✓	–	✓	–	–
SETF	–	–	–	–	–
SLEEP, SWAPF	–	–	–	–	–
SUBFWB, SUBLW	✓	✓	✓	✓	✓
SUBWF, SUBWFB,	✓	✓	✓	✓	✓
XORLW, XORWF	✓	–	✓	–	–

✓ Affected, – Not Affected

Note: TBLRD*, TBLRD*+, TBLRD*-, TBLRD+*, TBLWT*, TBLWT*+, TBLWT*-, and TBLWT+* do not affect any flags.

- All instructions in the above table except MOVFF and LFSR are executed in one cycle; MOVFF and LFSR are executed in two cycles.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.
- For destination: d = 0 means that the destination is WREG while d = 1 means that the destination is file register.

TABLE 6.4 PIC18F Data Movement Instructions

Instruction	*Comment*
CLRF F, a	Clear data register F to zero. F is located in access bank if a = 0 and F is located in the bank specified by BSR if a =1
LFSR F, K	Load low 12 bits of K into the specified File Select Register (F can be 0 or 1 or 2)
MOVLB K	Move 8-bit value K to BSR; low 4 bits of K are moved into low 4 bits of BSR with upper 4 bits of BSR are always cleared to zero regardless of upper 4 bits of K.
MOVLW data	Move 8-bit immediate data into WREG.
MOVWF F, a	Move data from WREG into data register, F. F is located in access bank if a = 0 and F is located in the bank specified by BSR if a =1.
MOVFF Fs, Fd	Move data from Fs (source data register) to Fd (destination data reg).
MOVF F, d, a	Move the contents of the file register into WREG (d = 0) or into the same file register (d = 1). F is located in access bank if a = 0 and F is located in the bank specified by BSR if a =1. This is the only MOVE instruction that affects N and Z status flags.
SETF F, a	Set all eight bits in the specified data register F to ones. See note for 'a'.
SWAPF F, d, a	Swaps low order 4 bits with the high-order 4 bits of the file register, F; see note for 'a' and 'd'

- The size of each instruction except LFSR and MOVFF is one word; the sizes of LFSR and MOVFF are two words).

6.3.1 Data Movement Instructions

The PIC18F data movement instructions are given in Table 6.4. Next, we will explain the data movement instructions using the access bank and specifying F or W in place of 'd'. In the following, typical examples are provided within brackets after each instruction for better understanding (assuming access bank).

CLRF F, a (CLRF 0x20) clears the contents of the specified data register, F to zero. If a = 0, then F is located in the access bank. On the other hand, if a = 1, then F is located in the bank specified by BSR . For example, CLRF 0x20 will clear the 8-bit contents of data register with address 0x20 to zero. After clearing, the Z-flag is set to one. No other flags are affected.

LFSR F, K (LFSR 2, 0x0020) loads low 12 bits of K into one of three FSR's (File Select Registers). The specified FSR can then be used to point to a data register. For example, LFSR 2, 0x0020 will load 00H into FSR2H and 20H into FSR2L. The low 12-bit contents of FSR2 (0x020) can then be used as a pointer to data memory. Note that since FSR's are 16-bit wide registers, the 12-bit address is stored in low 12 bits (bits 0 through 11) of the FSRs with the upper four bits (bits 12 through 15) as 0's.

MOVLB K (MOVLB 0x01) moves 8-bit value K to BSR; low 4 bits of K are moved into low 4 bits of BSR with upper 4 bits of BSR are always cleared to zero regardless of the upper 4 bits of K. For example, MOVLB 0x01 will move 01H into BSR. This instruction is useful for bank switching.

MOVLW data8 (MOVLW 0x25) moves an 8-bit literal (constant data) into WREG. For example, MOVLW 0x25 will move 25H into the WREG register. The previous contents of WREG are lost.

MOVWF F, a (MOVWF 0x40) moves data from WREG into data register, F. F is located in the access bank if a = 0 and F is located in the bank specified by BSR if a =1. As an example, consider MOVWF 0x40.

Prior to execution of MOVWF 0x40: [0x40] = F1H, [WREG] = 53H

After execution of MOVWF 0x40: [0x40] = 53H, [WREG] = 53H (unchanged)

MOVFF Fs, Fd (MOVFF 0x04, 0x03) moves data from source data register Fs to destination data register Fd. WREG can be used as either Fs or Fd. Also, Fs and Fd can be any data memory location from 000H to FFFH. This is a two-word (32 bits) instruction. As an example, consider MOVFF 0x04, 0x03.

Prior to execution of MOVFF 0x04, 0x03: [0x03] = 2FH, [0x04] = 57H

After execution of MOVFF 0x04, 0x03: [0x03] = 57H, [0x04] = 57H (unchanged)

MOVF F, d, a (MOVF 0x30, W or MOVF 0x30, F) moves the contents of the data register F into WREG (d = 0) or into the same data register, F (d = 1). F is located in the access bank if a = 0 and F is located in the bank specified by BSR if a =1.

As an example, consider MOVF 0x30, W.

Prior to execution of MOVF 0x30, W: [WREG] = 70H, [0x30] = 2AH

After execution of MOVF 0x30, W: [WREG] = 2AH, [0x30] = 2AH (unchanged)

As another example, consider MOVF 0x30, F.
Prior to execution of MOVF 0x30, F: [WREG] = 70H, [0x30] = 2AH
After execution of MOVF 0x30, F: [WREG] = 70H, [0x30] = 2AH (unchanged). This is the same as NOP (No operation instruction), and can be used for time delays.

SETF F, a (SETF 0x20) sets the contents of the specified data register (F) to FFH. If a = 0, then the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank. As an example, consider SETF 0x20.

Prior to execution of SETF 0x20: [0x20] = 24H

After execution of SETF 0x20: [0x20] = FFH

The SETF instruction can be used to configure an I/O port. For example, the TRISB register in the PIC18F is used to configure Port B. Writing 1's to all 8 bits of TRISB will configure Port B as an input port. This can be accomplished using the SETF TRISB instruction. This topic will be discussed later.

SWAPF F, d, a (SWAPF 0x60, W or SWAPF 0x60, F) exchanges low order 4 bits with the high-order 4 bits of the file register, F. F is located in access bank if a = 0 and F is located in the bank specified by BSR if a =1. d = 0 means that the destination is WREG while d = 1 means that the destination is a file register.

As an example, consider SWAPF 0x60, W.

Prior to execution of the instruction, SWAPF 0x60, W: [0x60] = 48H, [WREG] = 50H.

After execution of the instruction, SWAPF 0x60, W: [0x60] = 48H, [WREG] = 84H.

As another example, consider SWAPF 0x60, F.

Prior to execution of the instruction, SWAPF 0x60, F: [0x60] = 48H, [WREG] = 50H.

After execution of the instruction, SWAPF 0x60, F: [0x60] =84H, [WREG] = 50H.

Example 6.1 Determine the effect of each of the following PIC18F instructions:

- CLRF PREINC1
- MOVWF INDF1
- MOVFF 0x40, 0x81,
- MOVF 0x40, F; also find the flags which are affected
- SWAPF 0x45, W

Assume the following initial configuration before each instruction is executed; also, assume that all numbers are in hex:

[FSR0] = 0044,	[FSR1] = 0075
[043] = 66,	[076] = FF
[075] = 24,	[WREG] = 33
[040] = 78,	[045] = 61
[081] = 55	

Solution See Table 6.5

Example 6.2 Find the affected FSR, WREG, and data register contents for the following PIC18F instruction sequence:

```
LFSR    2,0x0044
MOVLW   D'20'
MOVWF   0x40
MOVFF   PLUSW2, 0x40
CLRF    POSTINC2
SETF    0x40
```

Assume [0x58] = 1AH prior to execution of the instruction sequence.

Solution

After execution of LFSR 2,0x0044, the File Select register FSR2 is loaded with 0044H. MOVLW D'20' moves immediate decimal data 20 (14H) into WREG. The instruction MOVWF 0x40 moves [WREG] into data register 0x40. Hence, [0x40] = 14H.

MOVFF PLUSW2, 0x40 adds [WREG] with [FSR2] and then moves the byte content of that address into data register 0x40. This means that the contents of data register 0x58 are moved to WREG. Hence, [0x40] = 1AH.

CLRF POSTINC2 clears the contents of data register 0x44 to zero, and then

TABLE 6. 5 Results for Example 6.1 (All numbers in hex)

Instruction	Affected Register	Net Effect (Hex)
CLRF PREINC1	Data Register Address = 076	[076]= 00
MOVWF INDF1	Data Register Address = 075	[075] = 33
MOVFF 0x40, 0x81	Data Register Address = 081	[0x81] = 78
MOVF 0x40, F	Data Register Address = 040	[0x40] = 78,N = 0, Z=0
SWAPF 0x45, W	WREG	[WREG] = 16

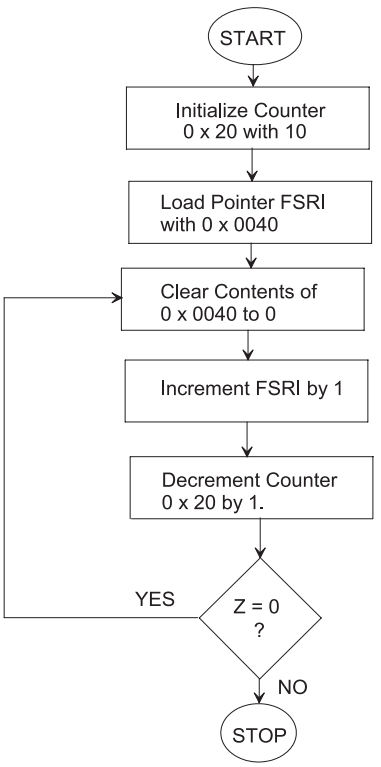
increments FSR2 by 1. Hence, [0x44] = 00H, and [FSR2] = 0x45.

SETF 0x40 sets all bits in data register 0x40 to ones. Hence, [0x40] = FFH.

Example 6.3 It is desired to clear 10 consecutive bytes to zero from LOW to HIGH data register addresses starting at data register 0x40 addressed by FSR1.

(a) Flowchart the problem.

(b) Convert the flowchart to a PIC18F assembly language program starting at address 0x100.



Solution

(a) The flowchart is provided above.

(b) The flowchart is converted to PIC18F assembly language program as follows:

```
INCLUDE     <P18F4321.INC>
  ORG       0x100
```

```
        MOVLW   D'10'       ; Move 10 decimal into WREG
        MOVWF   0x20        ; Initialize Counter reg (20H) with 10 decimal
        LFSR    1, 0x0040   ; Initialize FSR1 with 040H one below starting
                            ; address since preincrement mode is to be used
REPEAT  CLRF    POSTINC1    ; then Clear a location to 0 and increment FSR1 by 1
        DECF    0x20, F     ; Decrement Counter by 1
        BNZ     REPEAT      ; Branch to  REPEAT if  Zero flag = 0, otherwise
                            ; go to the next instruction
        SLEEP               ; HALT
        END
```

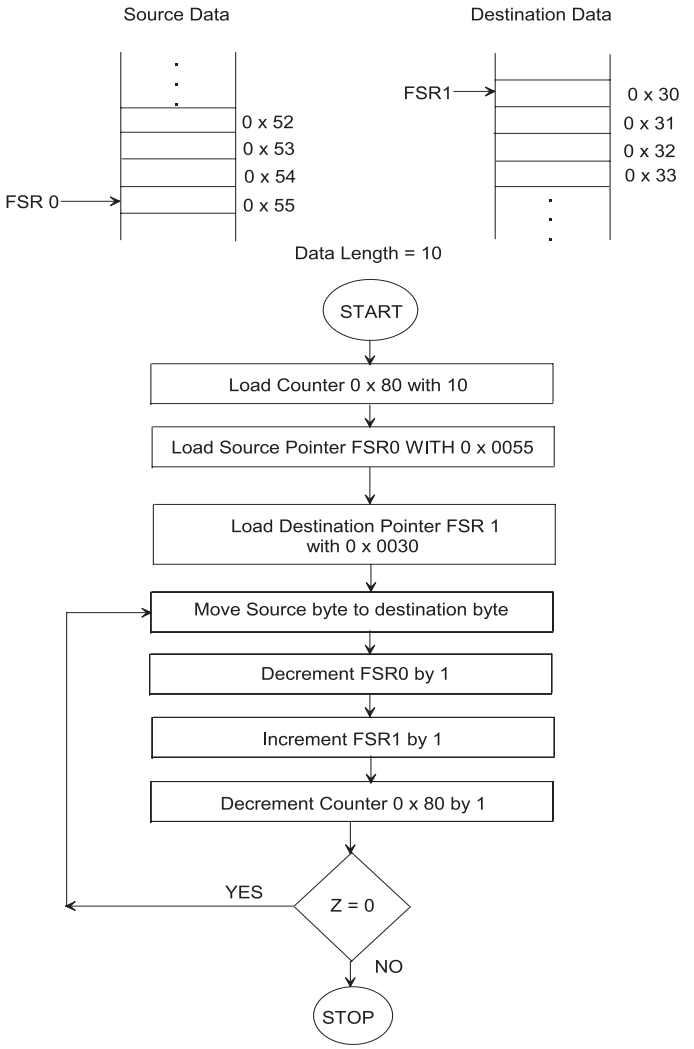
Example 6.4 It is desired to move a block of 8-bit data of length 10 from the source block (from HIGH to LOW address) starting at data register address 0x55 to the destination block (from LOW to HIGH addresses) starting at data register address 0x30. That is, [0x55] will be moved to [0x30], [0x54] to [0x31], and so on. Assume that data for the source block and the destination block are already stored in data memory addresses.

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Solution

(a) The flowchart along with the data memory layout is provided below:



(b) The flowchart can be converted to PIC18F assembly language program as follows:

```
            INCLUDE <P18F4321.INC>
COUNTER     EQU     0x80
            ORG     0x100
            MOVLW   D'10'           ; Move 10 decimal into WREG
            MOVWF   COUNTER         ; Initialize Counter reg (0x80) with 10 decimal
            LFSR    0, 0x0055       ; Initialize FSR0 with source starting address
            LFSR    1, 0x0030       ; Initialize FSR1 with destination starting address
BACK        MOVFF   POSTDEC0, POSTINC1  ; Move Source data to destination
            DECF    COUNTER, F      ; Decrement Counter by 1
            BNZ     BACK            ; Branch to BACK if Zero flag = 0, otherwise
                                    ; go to the next instruction
            SLEEP                   ; HALT
            END
```

6.3.2 Arithmetic Instructions

The PIC18F arithmetic instructions allow:

- 8-bit addition and subtraction
- 8-bit by 8-bit unsigned multiplication. The PIC18F does not provide any signed multiplication and division (Signed and Unsigned) instructions.
- Negate instruction
- Decrement and Increment Instructions
- BCD Adjust (BCD Correction)

As mentioned before, typical microcontrollers utilize common hardware to perform addition and subtraction operations for both unsigned and signed numbers. The instruction set of microcontrollers include the same ADD and SUBTRACT instructions for both unsigned and signed numbers. The interpretations of unsigned and signed ADD and SUBTRACT operations are performed by the programmer.

Unsigned and signed multiplication and division operations can be performed using various algorithms. Typical 32-bit microprocessors such as the Pentium contain separate instructions for performing these multiplication and division operations. The PIC18F provides only unsigned multiplication instruction. The other multiplication and division instructions can be obtained by writing PIC18F assembly language programs using appropriate algorithms. This is shown in Chapter 7.

The PIC18F arithmetic instructions are summarized in Table 6.6. Next, we will explain the arithmetic instructions using the access bank and specifying F or W in place of 'd'.

- All instructions in Table 6.6 are executed in one cycle.
- The size of each instruction is one word.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.
- For destination: d = 0 means that the destination is WREG while d = 1 means that the destination is file register. As mentioned before, W or F instead of 0 or 1 will be used in this book for better clarity.
- The PIC18F does not provide any multiplication (signed) and division (signed and unsigned) instructions.

Addition and Subtraction Instructions The PIC18F addition and subtraction instructions are illustrated by means of numerical examples in the following. All flags are affected. Assume signed numbers.

ADDLW data8 (ADDLW 0x02) instruction adds the 8-bit contents of WREG with 8-bit immediate data, and stores the result in WREG. For example, Consider ADDLW 0x02.

Prior to execution of ADDLW 0x02: [WREG] = 12H.
After execution of ADDLW 0x02: [WREG] = 12H + 02H = 14H.
The flags are affected based on the result as follows:

```
                    Previous carry →  0000  010  Intermediate Carries
                   [WREG]=  12H = 0001  0010
          Add  immediate data, 02H = 0000 0010
                                     ----------
                    final carry → 0   0001 0100 = 14H
```

TABLE 6.6 PIC18F Arithmetic Instructions

Instruction	*Operation*
Addition and Subtraction Instructions	
ADDLW data8	[WREG] + 8-bit data → [WREG]
ADDWF F, d, a	[WREG] + [F] destination; see note for 'a' and 'destination'
ADDWFC F, d, a	[WREG] + [F] + Carry→ destination; see note for 'a' and 'destination'
SUBLW data8	8-bit data - [WREG] → [WREG]
SUBWF F, d, a	[F] – [WREG] → destination; see note for 'a' and 'destination'
SUBWFB F, d, a	[F] – [WREG] – Carry → destination; see note for 'a' and 'destination'
SUBFWB F, d, a	[WREG] – [F]– Carry → destination; see note for 'a' and 'destination'
Unsigned Multiplication Instructions	
MULLW data8	[WREG] x 8-bit data → [PRODH]: [PRODL] (unsigned multiplication)
MULWF F, a	[WREG]x [F] → [PRODH]: [PRODL] (unsigned multiplication)
Negate instruction	
NEGF F, a	0 – [F]→[F]; see note for 'a'.
Decrement and Increment instructions	
DECF F, d, a	[F] – 1→ destination; see note for 'a' and 'destination'
INCF F, d, a	[F] + 1→ destination; see note for 'a' and 'destination'
BCD Adjust (BCD Correction) *Instruction*	
DAW	Decimal Adjust [WREG]

N = 0 (most significant bit of the result is 0), OV = 0 (no overflow since the previous carry and the final carry are the same), Z = 0 (nonzero result), DC = 0 (No carry from bit 3 to bit 4), C = 0 (no Carry).

ADDWF F, d, a (ADDWF 0x50, W or ADDWF 0x50, F) instruction adds the contents of WREG with the contents of the specified data register (F). The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in access bank if a = 0 or specified by BSR if a = 1.

As an example, consider ADDWF 0x50, W.

Prior to execution of ADDWF 0x50, W: [WREG] = 73H, [0x50] = 2AH.

After execution of ADDWF 0x50, W: [WREG]= 73H + 2AH = 9DH, [0x50] = unchanged = 2AH

The flags are affected based on the result as follows:

Previous carry →1100 010 ← Intermediate Carries

[WREG] = 73H = 0111 0011

Add [0x50] = 2AH = 0010 1010

---

final carry → 0 1001 1101 = 9DH

N = 1 (most significant bit of the result is 0), OV = $C_f \oplus C_p = 0 \oplus 1 = 1$ (overflow indicating wrong result; addition of two positive numbers generates a negative result), Z = 0 (nonzero result), DC = 0 (No carry from bit 3 to bit 4), C = 0 (no Carry). Note that the correct result can be rectified by increasing the number of bits for the two signed numbers to be added (73H and 2AH).

As another example, consider ADDWF 0x50, F.

Prior to execution of ADDWF 0x50, F: [WREG] = 73H, [0x50] = 2AH

After execution of ADDWF 0x50, F: [0x50] = 73H + 2AH = 9DH, [WREG] =unchanged = 73H

ADDWFC F, d, a (ADDWFC 0x60, W or ADDWFC 0x60, F) adds the contents of WREG with the contents of the specified data register and the carry flag. The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in access bank if a = 0 or specified by BSR if a = 1.

As an example, consider ADDWFC 0x60, W.

Prior to instruction execution: Carry bit = 1, [0x60] = 03H, [WREG] = 07H

After instruction execution: C = 0, [0x60] = 03H (unchanged), [WREG] = 0BH,
N = 0, OV = 0, Z = 0, DC = 0, C = 0.

Note that ADDWFC 0x60, F performs the same operation as ADDWFC 0x60, W except that the addition result is stored in 0x60.

SUBLW data8 (SUBLW 07H) subtracts [WREG] from 8-bit immediate data. The result is placed in WREG. As an example, consider SUBLW 07H.

Prior to instruction execution: [WREG] = 03H.

After Instruction execution: [WREG] = 07H – 03H = 04H.

The flags are affected as follows:

Using two's complement subtraction, 1111 111 ← Intermediate Carries

Immediate data = 07H = 0000 0111

Add two's complement of 03 = 1111 1101

---

final carry →1 0000 0100 (04H)

The final carry is one's-complemented after subtraction to reflect the correct borrow. Hence, C = 0. Also, N = 0 (most significant bit of the result is zero), OV = $C_f \oplus C_p$ = $1 \oplus 1 = 0$, Z = 0 (nonzero Result), DC = 1. Note that in the above, the final carry is 1 indicating a borrow while performing the operation (07H - 03H). The correct result should have been 04H without any borrow. But, in the above, the result is 04H with a borrow. While performing two's complement subtraction analytically using pencil and paper, the borrow is always one's complement of the true borrow. Hence, the PIC18F complements the carry to reflect the true borrow.

SUBWF F, d, a (SUBWF 0x20, W or SUBWF 0x20, F) instruction subtracts the contents of WREG from the contents of the specified data register. The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in access bank if a = 0 or specified by BSR if a = 1. The flags are affected in the same way as the SUBLW instruction. Note that the carry is one's complemented to reflect the correct borrow.

SUBWFB F, d, a (SUBWFB 0x30, W or SUBWFB 0x30, F) instruction subtracts the contents of WREG and the Carry from the contents of the specified data register. The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in access bank if a = 0 or specified by BSR if a = 1.

SUBFWB F, d, a (SUBFWB 0x20, W or SUBFWB 0x20, F) instruction subtracts the contents of specified data register and the Carry from the contents of the WREG. That is, the SUBWFB performs the following operation using two's complement: [dest] ← [F] - [WREG + Carry]. The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in access bank if a = 0 or specified by BSR if a = 1 .

Multiplication Instructions The PIC18F includes an 8 x 8 hardware multiplier as part of the ALU. The multiplier performs an unsigned operation and provides a 16-bit result that is stored in the product register pair, PRODH:PRODL. Because of hardware implementation,

the multiplier executes the multiplication operation in a single instruction cycle. This has the advantages of higher computational throughput and reduced code size for multiplication algorithms and allows the PIC18F to be used in many applications previously reserved for digital signal processors (DSPs). The PIC18F includes two instructions for performing 8-bit x 8-bit unsigned multiplication providing 16-bit result in the product register PRODH:PRODL. 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. The PIC18F does not provide either signed multiplication instruction nor any division instruction (signed and unsigned). The PIC18F unsigned multiplication instructions are described in the following.

- MULLW data8 (MULLW 0x03) instruction performs an unsigned multiplication between the 8-bit contents of WREG and 8-bit immediate data. The 16-bit result is placed in the PRODH:PRODL register pair. PRODH contains the high byte, and PRODL contains the low byte. The contents of WREG are unchanged. As an example, consider MULLW 0x03.

  Prior to instruction execution: [WREG] = 02H
  After Instruction execution: [PRODH] = 00H, [PRODL] = 06H, [WREG] = 02H = unchanged

- MULWF F, a (MULWF 0x50) instruction performs an unsigned multiplication between the 8-bit contents of WREG and 8-bit contents of the specified data register. The 16-bit result is placed in the PRODH:PRODL register pair. PRODH contains the high byte, and PRODL contains the low byte. The contents of the WREG and the data register are unchanged. The data register is in the access bank if a = 0 or specified by BSR if a = 1.

Negate instruction The PIC18F Negate instruction is illustrated by means of numerical examples in the following.

- NEGF F, a (NEGF 0x70) instruction negates the contents of the specified data register using two's complement. The result is stored in the data register. The data register is in access bank if a = 0 or specified by BSR if a = 1. An example is NEGF 0x70.

  Prior to instruction execution: [0x70] = 02H.
  After instruction execution: [0x70] = FEH = -2_{10}.

Decrement and Increment Instructions The PIC18F decrement and increment instructions are illustrated in the following by means of numerical examples.

- DECF F, d, a (DECF 0x50, W or DECF 0x50, F) decrements the contents of the specified data register by 1. The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in the access bank if a = 0 or specified by BSR if a = 1. All flags are affected. An example is DECF 0x50, F.

  Prior to instruction execution: [0x50] = 01H.
  After instruction execution: [0x50] = 00H.

  Note that DECF 0x50, W decrements [0x50] by 1 and the result is stored in WREG.

- INCF F, d, a (INCF 0x50, W or INCF 0x50, F) increments the contents of the specified data register by 1. The result is stored in WREG if d = 0 or in the data register if d = 1. The data register is in the access bank if a = 0 or specified by BSR if a = 1. An example is INCF 0x50, F.

Prior to instruction execution: [0x50] = FFH = -1_{10}.
After instruction execution: [0x50] = 00H.
INCF 0x50, W performs the same operation as INCF 0x50, F except that the result is stored in WREG.

BCD adjust (BCD correction) instruction The PIC18F contains a BCD adjust instruction which will be illustrated in the following by means of a numerical example.

- DAW instruction adjusts the eight-bit result in WREG after adding two packed BCD numbers using ADDLW or ADDWF or ADDWFC to provide the correct packed BCD result.

If, after the addition, the low 4 bits of the result in WREG are greater than 9 (or if DC = 1), the DAW adds 6 to the low 4 bits of WREG. On the other hand, if the high 4 bits of the result in WREG are greater than 9 (or if C = 1), DAW adds 6 to the high 4 bits in WREG. Consider the following instruction sequence:

```
MOVLW 0x29   ; Move 29H into WREG
ADDLW 0x54   ; Add 29H with 54H and store the result in WREG
DAW          ; Decimal adjust WREG to provide the correct packed BCD result
```

The details of the result obtained by the instruction sequence above are provided in the following:

```
[WREG] = 29H = 0010 1001  (Packed BCD 29, same as 29H)
    Add  54H = 0101 0100  (Packed BCD 54, same as 54H)
           ---------------
 [WREG] = 0111 1101
              0110   Add 6 (BCD correction by DAW since low 4 bits of the
           ---------------      sum in WREG are greater than 9)
          1000 0011 = 83H correct packed BCD result since 29 + 54 = 83
```

Note that packed BCD is covered in section 1.2.3 of Chapter 1.

Example 6.5 Write a PIC18F assembly language program to implement the following C segment;

```
if (x<0)
y++;
else
y--;
```

Solution

```
        MOVF   X, F     ; Move [X] into [F], MOVF affects N flag
        BNN    ELSE     ; Branch to ELSE if N = 0  (if [X] is positive)
        INCF   Y        ; else, if N = 1 ( [X] is negative), Increment [Y] by 1
        BRA    NEXT
ELSE    DECF   Y        ; Decrement [Y] by 1 if N = 0
NEXT    GOTO   NEXT     ; Halt
```

Example 6.6 Write a PIC18F assembly language instruction sequence to add four numbers 1,2,3,4, and then store the result in a data register 0x40 as follows:
(a) without using a loop (b) using a loop

Solution

(a) without using a loop

```
SUM     EQU     0x40
        MOVLW   1       ; Move 1 into WREG
        ADDLW   2       ; Add 2 with [WREG] and store 3 in WREG
        ADDLW   3       ; Add 3 with [WREG] and store 6 in WREG
        ADDLW   4       ; Add 4 with [WREG] and store 10 in WREG
        MOVWF   SUM     ; Store [WREG] in SUM
```

(b) using a loop

```
SUM     EQU     0x40
COUNTER EQU     0x50
        CLRF    SUM         ; Clear  SUM to 0
        MOVLW   4           ; Move 4 into WREG
        MOVWF   COUNTER     ; Initialize COUNTER with 4
        MOVF    SUM, W      ; Move 0 to WREG
LOOP    ADDWF   COUNTER, W  ; Add [COUNTER] with [WREG], store result in W
        DECF    COUNTER     ; Decrement [COUNTER] by 1
        BNZ     LOOP        ; Branch to loop if Z is not equal to 0
        MOVWF   SUM         ; Move result (10) from WREG into SUM
```

Example 6.7 Write a PIC18F instruction sequence to implement the following C statement:

c = a + b;

Assume data registers 0x50, 0x60, and 0x70 store a, b, and c respectively.

Solution

```
A       EQU     0x50
B       EQU     0x60
C       EQU     0x70
        MOVF    A, W    ; Move [A] to [WREG]
        ADDWF   B, W    ; Add [B] with [WREG], result in WREG
        MOVWF   C       ; store result in WREG in C
```

Example 6.8 Write a PIC18F instruction sequence to implement the following C statement:

c = 2a + b;

Assume data registers 0x24, 0x31, and 0x50 store a, b, and c respectively.

Solution

```
A       EQU     0x24
B       EQU     0x31
C       EQU     0x50
        MOVF    A, W    ; Move [A] to [WREG]
        ADDWF   A, W    ; Add [A] with [WREG], result 2 x [A] in WREG
        ADDWF   B, W    ; Add 2x [A] with [B], store result in WREG
        MOVWF   C       ; store result in WREG in C
```

Example 6.9 Write a PIC18F assembly language program at address 0x100 that implements the following C language program segment:

```
sum = 0;
for (i = 0; i <= 9; i = i + 1)
sum = sum + a[i];
```

where sum is the address of the 8-bit result of addition. Assume sum as 0x50 and the address of the first element of the array, a[0], is stored in data register 0x20, the second element, a[1], in data register 0x21, and so on. Assume that the array is already stored in data memory. Also, assume addition of two consecutive will generate no carry.

Solution

Assume that register FSR0 holds the address of the first element of the array. The assembly language program is listed below:

```
        INCLUDE  <P18F4321.INC>
        ORG      0x100
SUM     EQU      0x50          ; Initialize SUM to 0x50 for result
        LFSR     0, 0x0020     ; Point FS0 to a[0]
        CLRF     SUM           ; Clear [SUM] to zero
        MOVLW    D'10'         ; Move  WREG with 10
        MOVWF    0x70          ; Initialize 0x70 with loop count (10)
LOOP    MOVF     SUM, W        ; Move [SUM] into WREG
        ADDWF    POSTINC0, W   ; Add and Store result in WREG
        MOVWF    SUM           ; Save [WREG] IN SUM
        DECF     0x70, F       ; Decrement counter 0x70 by 1
        BNZ      LOOP          ; Branch to LOOP if  Z not equal 0
        SLEEP                  ; Halt
        END
```

Example 6.10 Write a PIC18F assembly language program at address 0x100 to add two 16-bit numbers as follows:

```
    [0x51] [0x50]
PLUS [0x61] [0x60]
    ------------------------
    [0x61] [0x60]
```

Assume data registers 0x50 and 0x51 contain low and high bytes of the first 16-bit numbers while data registers 0x60 and 0x61 contain the second 16-bit numbers. Also, assume that data are already loaded into the data registers.

Solution

```
INCLUDE   <P18F4321.INC>
ORG       0x100
MOVF      0x50, W     ; Move low byte of first data into WREG
ADDWF     0x60, F     ; Add low 8 bits, store result in 0x60
MOVF      0x51, W     ; Move high byte of first data into WREG
ADDWFC    0x61, F     ; Add high bytes with carry, store result in 0x61
SLEEP                 ; Halt
END
```

Example 6.11 Write a PIC18F assembly language program at address 0x100 to add four 8-bit numbers stored in consecutive data registers from high to low addresses starting at data register 0x78. Store the 8-bit result WREG. Assume that no carry is generated, due to the addition of two consecutive 8-bit numbers. Assume that data are already loaded into data memory addresses 0x75 through 0x78 with the first data byte at 0x78 and the last data byte at 0x75.

Solution

```
        INCLUDE   <P18F4321.INC>
        ORG       0x100
        MOVLW     4               ; Move WREG with 4
        MOVWF     0x50            ; Initialize 0x50 with loop count (4)
        MOVLW     0x00            ; Clear  WREG to store sum
        LFSR      0, 0x0078       ; Initialize pointer FSR0 with 0x0078
START   ADDWF     POSTDEC0, W     ; Add two bytes and store sum in WREG
        DECF      0x50, F         ; Decrement counter 0x50 by 1
        BNZ       START           ; Branch to START  if  Z is 0
        SLEEP                     ; Halt
        END
```

Example 6.12 Write a PIC18F assembly language program at 0x100 to subtract two 32-bit numbers as follows:

```
    [0x43] [0x42] [0x41] [0x40]
MINUS [0x75] [0x74] [0x73] [0x72]
      ---------------------------------------
    [0x43] [0x42]  [0x41] [0x40]
```

Assume data registers 0x40 through 0x43 contain the first 32-bit number while data registers 0x72 through 0x75 contain the second 32-bit number. Also, assume that data are already loaded into the data registers. Store the 32-bit result in data registers 0x40 (lowest byte) through 0x43 (highest byte). Do not use a loop.

Solution

```
        INCLUDE   <P18F4321.INC>
        ORG       0x100
        MOVF      0x72, W       ; Move byte into WREG and update pointer
        SUBWF     0x40, F       ; Subtract [WRFG] from a byte, store result
                                ; in data registers
        MOVF      0x73, W
        SUBWFB    0x41, F       ; Subtract [WREG] and carry from byte
        MOVF      0x74, W       ; and store rusult in data reg
        SUBWFB    0x42, F
        MOVF      0x75, W
        SUBWFB    0x43, F
        SLEEP                   ; Halt
        END
```

Example 6.13 Write a PIC18F assembly language program at address 0x100 to compute $(X^2 + Y^2)$ where X and Y are two 8-bit unsigned numbers stored in data registers 0x50 and 0x40 respectively. Store the 16-bit result in data registers 0x51 (high byte) and 0x50 (low byte). Assume X and Y are already loaded into data registers 0x50 and 0x40.

Solution

```
INCLUDE  <P18F4321.INC>
ORG      0x100
MOVF     0x40, W          ; Move X into WREG
MULWF    0x40             ; Multiply X by X, result in PRODH:PRODL
MOVFF    PRODL, 0x50      ; Save low byte of result in data reg 0x50
MOVFF    PRODH, 0x51      ; Save high byte of result in data reg 0x51
MOVF     0x40, W          ; Move Y into WREG
MULWF    0x40             ; Multiply Y by Y, result in PRODH:PRODL
MOVF     PRODL, W         ; Move PRODL into WREG
ADDWF    0x50, F          ; Add and store low byte of result in 0x50
MOVF     PRODH, W         ; Move high byte of X times X into WREG
ADDWFC   0x51, F          ; Add high bytes with carry. Store result in 0x51
SLEEP                     ; Halt
END
```

Example 6.14 Write a PIC18F assembly language program at address 0x100 to add two packed BCD bytes stored in data registers 0x20 and 0x21. Store the correct packed BCD result in WREG. Load packed BCD bytes 0x72 and 0x45 into data registers 0x20 and 0x21 respectively using PIC18F instructions. Note that data are arbitrarily chosen.

Solution

```
INCLUDE  <P18F4321.INC>
ORG      0x100
MOVL     W 0x72           ; Load first packed BCD data
MOVWF    0x20             ; into 0x20
MOVL     W 0x45           ; Load second packed BCD data
MOVWF    0x21             ; into 0x21
MOVF     0x20, W          ; Move 72H into WREG
ADDWF    0x21, W          ; Add and store binary result in WREG
DAW                       ; Convert WREG to correct packed
                          ; BCD byte
SLEEP                     ; Halt
END
```

6.3.3 Logic Instructions

The PIC18F logic instructions include logic AND, NOT (one's complement), OR, and Exclusive-OR operations. Table 6.7 lists PIC18F Logic instructions. Next, we explain the Logic instructions using the access bank and specifying F or W in place of 'd'.

- ANDLW data8 (ANDLW 0x8F) instruction ANDs the contents of WREG with the 8-bit literal (immediate data, data8). The result is placed in WREG. As an example, consider ANDLW 0x8F.

Prior to instruction execution: [WREG] = 0x72
After instruction execution:

```
[WREG] = 0x72 =   0111 0010
   AND  0x8F =   1000 1111
                 ---------
[WREG] =         0000 0010
```

N and Z flags are affected. Z = 0 (Result is nonzero) and N = 0 (Most Significant Bit of the result is 0). The status flags are affected in the same way after execution of other logic instructions such as OR, XOR, and NOT.

- ANDWF F, d, a (ANDWF 0x60, W or ANDWF 0x60, F) instruction ANDs the contents of WREG 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'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank.

  As an example, consider ANDWF 0x60, F.
  Prior to instruction execution: [0x60] = 0xFF, [WREG] = 0x01
  After instruction execution: [0x60] = 0x01, [WREG] = 0x01 (unchanged), Z = 0, N = 0
  The AND instruction can be used to perform a masking operation. If the bit value in a particular bit position is desired in data byte, the data can be logically ANDed with appropriate data to accomplish this. For example, the bit value at bit 2 of an 8-bit number 0100 1Y10 (where unknown bit value of Y is to be determined) can be obtained as follows:

```
          0 1 0 0 1 Y 1 0  -- 8-bit number
AND       0 0 0 0 0 1 0 0 -- Masking data
          -----------------
          0 0 0  0  0 Y 0 0 -- Result
```

TABLE 6.7 PIC18F Logic Instructions

Instruction	*Comment*
ANDLW data8	[WREG] AND 8-bit data → [WREG]
ANDWF F, d, a	[WREG] AND [F] → destination; see note for 'a' and 'destination'
COMF F, d, a	NOT [F] → destination; see note for 'a' and 'destination'
IORLW data8	[WREG] OR 8-bit data → [WREG]; Performs Inclusive OR or simply OR operation.
IORWF F, d, a	[WREG] OR [F] → destination; see note for 'a' and 'destination'. Performs inclusive OR or simply OR operation.
XORLW data8	[WREG] ⊕ 8-bit data → [WREG]
XORWF F, d, a	[WREG] ⊕ [F] → [destination] ; see note for 'a' and 'destination'

- All instructions in the above are executed in one cycle.
- The size of each instruction is one word.
- All instructions affect N and Z flags; other flags are not affected.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.
- For destination: d = 0 means that the destination is WREG while d = 1 means that the destination is file register F.

If the bit value Y at bit 2 is 1, then the result is nonzero (Flag Z=0); otherwise, the result is zero (Z=1). The Z flag can be tested using typical conditional branch instructions such as BZ (Branch if Z=1) or BNZ (Branch if Z = 0) to determine whether Y is 0 or 1. This is called masking operation. The AND instruction can also be used to determine whether a binary number is ODD or EVEN by checking the Least Significant Bit (LSB) of the number (LSB=0 for even and LSB=1 for odd).

ANDWF 0x60, W performs the same operation as ANDWF 0x60, F except the result is stored in WREG.

- COMF F, d, a (COMF 0x50, W or COMF 0x50, F) instruction complements (one's) the contents of register 'F'. If 'd' is '0', the result is stored in WREG. If 'd' is '1', the result is stored back in register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank.

  For example, consider COMF 0x50, W.
  Prior to instruction execution: [0x50] = 0x01, [WREG] = 0x57
  After instruction execution: [WREG] = 0xFE, [0x50] = 0x01 (unchanged),
  Z = 0, N = 1; No other flags are affected.
  Note that COMF 0x50, F is the same as COMF 0x50, F except that the result is stored in WREG.

- IORLW data8 (IORLW 0x7F) instruction ORes the contents of WREG with the 8-bit literal (immediate data, data8). The result is placed in WREG. As an example, consider IORLW 0x7F.

  Prior to instruction execution: [WREG] = 0x01
  After instruction execution: [WREG] = 0x7F (unchanged), Z = 0, N = 0; other flags are not affected.
  The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits. For example, a 1 can be inserted using the OR instruction at bit number 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits as follows:

```
        0 1 1 1 0 0 1 1 -- 8-bit number
OR      0 0 0 0 1 0 0 0 -- data for inserting a 1 at bit number 3
        -------------------
        0 1 1 1 1 0 1 1 -- Result
```

- IORWF F, d, a (IORWF 0x50, W or 0x50, F) instruction ORes [WREG] with register 'F'. If 'd' is '0', the result is placed in WREG. If 'd' is '1', the result is placed back in register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank .

  As an example, consider IORWF 0x50, W.
  Prior to instruction execution: [WREG] = 0xA2,[0x50] = 0x5D
  After instruction execution: [WREG] = 0xFF, [0x50] = 0x5D (unchanged), Z = 0, N = 0; other flags are not affected.

IORWF 0x50, F is the same as the IORWF 0x50, W except that the result is stored in 0x50.

The OR instruction can typically be used to insert a 1 in a particular bit position of a binary number without changing the values of the other bits.

For example, a 1 can be inserted using the OR instruction at bit number 3 of the 8-bit binary number 0 1 1 1 0 0 1 1 without changing the values of the other bits as follows:

```
    0 1 1 1 0 0 1 1 -- 8-bit number
OR  0 0 0 0 1 0 0 0 -- data for inserting a 1 at bit number 3
    -------------------
    0 1 1 1 1 0 1 1 -- Result
```

IORWF 0x50, F stores the result in 0x50.

- XORLW data8 (XORLW 0x02) instruction exclusive-ORes the contents of WREG with 8-bit literal (immediate data, data8). The result is placed in WREG.

  As an example, consider XORLW 0x02 .
  Prior to instruction execution: [WREG] = 0x42
  After instruction execution: [WREG] = 0x40, Z = 0, N = 0 ; no other flags are affected.

  The Exclusive-OR instruction can be used to find the ones complement of a binary number by XORing the number with all 1's as follows:

```
    0 1 0 1 1 1 0 0 - - 8-bit number
XOR 1 1 1 1 1 1 1 1 - - data
    --------------------------
    1 0 1 0 0 0 1 1 -- Result (Ones Complement of the 8-bit number 0 1 0 1 1 1 0 0)
```

- XORWF F, d, a (XORWF 0x42, W or XORWF 0x42, F) instruction exclusive ORes the contents of WREG with register 'F'. If 'd' is '0', the result is stored in WREG. If 'd' is '1', the result is stored back in the register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select bank .

  As an example, consider XORWF 0x42, W .
  Prior to instruction execution: [WREG] = 0xFF, [0x42] = 0xFF
  After instruction execution: [WREG] = 0x00, Z = 1, N = 0 ; No other flags are affected.

  Note that XORWF 0x42, F stores the result in 0x42.

6.3.4 Rotate Instructions

The PIC18F Rotate instructions are listed in Table 6.8. Next, we explain the Rotate instructions using the access bank and specifying F or W in place of 'd'.

Let us now explain the instructions in Table 6.8 using numerical examples.

- RLCF F, d, a (RLCF 0x40, W or RLCF 0x40, F) instruction rotates the contents of register 'F' one bit to the left through the Carry flag. If 'd' is '0', the result is placed in WREG. If 'd' is '1', the result is stored back in register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank.

  As an example, consider RLCF 0x40, W.
  Prior to instruction execution: [WREG] = 0xFF, [0x40] = 0xAF , C = 0
  After instruction execution: [WREG] = 0x5E, C = 1, Z = 0 (result in WREG after rotating is nonzero), N = 0 (most significant bit of result, 0x5E is 0); no other flags are affected.

  Note that RLCF 0x40, F stores the result in data register 0x40.

- RLNCF F, d, a (RLNCF 0x70, W or RLNCF 0x70, F) instruction rotates the contents of register 'F' one bit to the left. If 'd' is '0', the result is placed in WREG. If 'd' is '1', the result is stored back in register 'F'. If 'a' is '0', the access Bank is selected. If 'a' is '1', the BSR is used to select the bank .

  As an example, consider RLNCF 0x70, F.
  Prior to instruction execution: [WREG] = 0x89, [0x70] = 0x32
  After instruction execution: [0x70] = 0x64, [WREG] = 0x89 (unchanged), Z = 0 (result in 0x70 after rotating is nonzero) , N = 0 (most significant bit of result, 0x64 is 0) ; no other flags are affected.
  Note that RLNCF 0x70, W stores the result in WREG.
  The RLNCF instruction can be used to multiply an unsigned number by 2^n by shifting the number, *n* times to the left using a loop as long as a '1' is not shifted out of the most significant bit. In the above example, after shifting [0x70] once, the contents 0x32 of data register 0x70 are multiplied by 2. Hence, [0x70] = 0x64 after shifting.

- RRCF F, d, a (RRCF 0x30, W or RRCF 0x30, F) instruction rotates the contents of register 'F' one bit to the right through the Carry flag. If 'd' is '0', the result is placed in WREG. If 'd' is '1', the result is stored back in register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank .

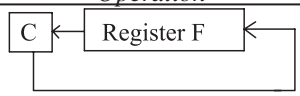
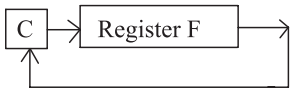
  As an example, consider RRCF 0x30, F.
  Prior to instruction execution: [WREG] = 0x91, [0x30] = 0x27 , C = 0
  After instruction execution: [0x30] = 0x13, [WREG] = 0x91 (unchanged), C = 1, Z = 0 (result, 0x13 is nonzero), N = 0 (most significant bit of result, 0x13 is 0) ; no other flags are affected.
  Note that RRCF 0x30, W stores the result in WREG.

- RRNCF F, d, a (RRNCF 0x60, W or RRNCF 0x60, F) instruction rotates the contents of register 'F' one bit to the right. If 'd' is '0', the result is placed in WREG. If 'd' is '1', the result is stored back in register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank.

  As an example, consider RRNCF 0x60, F.
  Prior to instruction execution: [WREG] = 0xB3, [0x60] = 0x28
  After instruction execution: [0x60] = 0x14, [WREG] = 0xB3 (unchanged), Z = 0 (result, 0x14 after rotating is nonzero), N = 0 (most significant bit of result, 0x14 is 0) ; no other flags are affected.
  Note that RRNCF 0x60, W stores the result in WREG.
  The RRNCF instruction can be used to divide an unsigned number by 2 by shifting the number *n* times to the right using a loop as long as a '1' is not shifted out of the least significant bit. This means that the remainder is discarded. Since the PIC18F does not have unsigned division instruction, the RRNCF can be used for this purpose. In the above example, after shifting [0x60] once, the contents, 0x28 of data register 0x60, are divided by 2. Hence, [0x60] = 0x14 after shifting.

Example 6.15 Write a PIC18F logic instruction to convert a 4-bit unsigned number in low 4 bits of WREG into an 8- bit unsigned number in WREG. Assume the 4-bit unsigned number is already loaded into WREG.

TABLE 6.8 PIC18F Rotate Instructions

Instruction	*Operation*
RLCF F, d, a	 Rotate register F one bit to the left through carry. See notes for 'd' and 'a'.
RLNCF F, d, a	 Rotate register F one bit to the left without carry. See notes for 'd' and 'a'.
RRCF F, d, a	 Rotate register F one bit to the left through carry. See notes for 'd' and 'a'.
RRNCF F, d, a	 Rotate register F one bit to the left without carry. See notes for 'd' and 'a'.

- All instructions in the above are executed in one cycle.
- The size of each instruction is one word.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.
- For destination: d = 0 means that the destination is WREG while d = 1 means that the destination is file register, F.
- RLCF and RRCF affect N, Z, and C flags according to the result while RLNCF and RRNCF affect N and Z flags based on the result.
- Note that the PIC18F does not have any shift instructions

Solution

```
ANDLW  0x0F
```

Note that ANDLW 0x0F instruction logically ANDs 8-bit data in WREG with 0x0F. This operation will clear the upper 4 bits to zero, and retain the lower 4 bits. Thus, the 4-bit unsigned number will be converted to an 8-bit unsigned number.

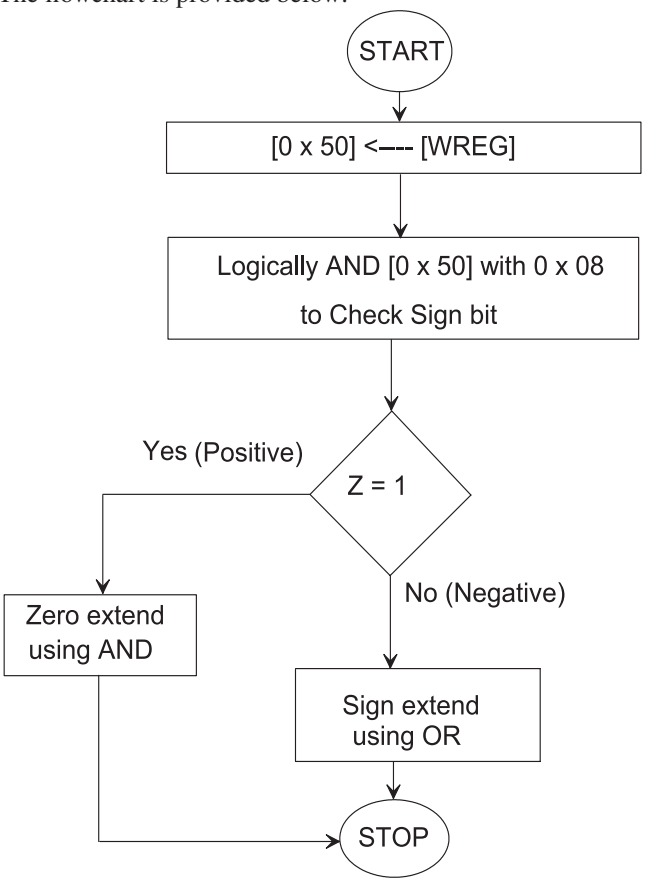
Example 6.16 Write a PIC18F assembly language program at address 0x100 to convert a 4-bit signed number stored in the low 4 bits of WREG into an 8- bit signed number in WREG. That is, if the 4-bit signed number is positive, zero-extend to 8 bits. On the other hand, if the 4-bit signed number is negative, sign-extend to 8 bits. Assume the 4-bit signed number is already loaded into WREG. Store result in WREG. Do not use any ROTATE instructions.

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Solution

(a) The flowchart is provided below:



(b) The PIC18F assembly language program is provided below:

```
            INCLUDE  <P18F4321.INC>
            ORG      0x100
            MOVWF    0x50        ; copy [WREG] into data register 0x50
            ANDLW    0x08        ; check bit 3 whether it is 0 (positive) or 1 (negative)
            BZ       POSITIVE    ; if Z = 1, the number is positive
            MOVF     0x50, W     ; else, the number is negative. Move [0x50] into W
            IORLW    0xF0        ; sign extend by inserting 1's in upper 4 bits of  W
            SLEEP                ; Halt. Result in WREG
POSITIVE    MOVF     0x50, W     ; the number is positive. Move [0x50] into WREG
            ANDLW    0x0F        ; the number is positive. hence zero extend upper 4
                                 ; bits
            SLEEP
            END
```

Example 6.17 Write a PIC18F assembly language program at address 0x100 to multiply an 8-bit unsigned number in data register 0x50 by 16. Store the 8-bit result in WREG. Do not use any multiplication instructions. Use ROTATE instruction. Assume that a '1' is not shifted out of the most significant bit each time after rotating to the left. Also, assume that the 8-bit unsigned number is already loaded into data register 0x40.
Write the program

(a) without using a loop (b) using a loop

Solution

(a) without using a loop

The following program will multiply [0x50] by 16 by shifting [0x50] four times to the left:

```
            INCLUDE  <P18F4321.INC>
            ORG      0x100
            BCF      STATUS, C   ; Clear Carry
            RLCF     0x50, F     ; Rotate [0x50] once  to left
            BCF      STATUS, C   ; Clear Carry
            RLCF     0x50, F     ; Rotate [0x50]  once to left
            BCF      STATUS, C   ; Clear Carry
            RLCF     0x50, F     ; Rotate [0x50] once  to left
            BCF      STATUS, C   ; Clear Carry
            RLCF     0x50, F     ; Rotate [0x50]  once to left
            MOVF     0x50, W     ; Save  result in WREG
FOREVER     GOTO     FOREVER     ; Stop
            END
```

(b) using a loop

The following program will multiply [0x50] by 16 by shifting [0x50] four times to the left in a loop:

```
            INCLUDE  <P18F4321.INC>
            ORG      0x100
COUNTER     EQU      0x70
            MOVLW    4              ; Initialize COUNTER with 4
BACK        BCF      STATUS, C      ; Clear Carry
            RLCF     0x50, F        ; Rotate [0x50] four times
            DECF     COUNTER, F     ; to left to multiply [0x50] by 16
            BNZ      BACK           ; branch to BACK if  Z = 0
            MOVF     0x50, W        ; Move result to WREG
FOREVER     GOTO     FOREVER        ; Stop
            END
```

Example 6.18 Write a PIC18F assembly language program at address 0x100 that will multiply an 8-bit unsigned number in data register 0x50 by 4 to provide an 8-bit product, and then, perform the following operations on the contents of data register 0x50 :

- set bits 0 and 3 to one without changing other bits in data register 0x50.
- clear bit 5 to zero without changing other bits in data register 0x50.
- one's complement bit 7 without changing other bits in data register 0x50.

Use only "Logic", and "Rotate" instructions. Do not use any multiplication or any other instructions. Assume data is already in data register 0x50. Store result in WREG. Assume that a '1' is not shifted out of the most significant bit each time after rotating to the left.

Solution

```
INCLUDE  <P18F4321.INC>
ORG      0x100
BCF      STATUS, C      ; Clear Carry
RLCF     0x50, F        ; Unsigned multiply [0x50] by 2, result in 0x50
BCF      STATUS, C      ; Clear Carry
RLCF     0x50, W        ; Unsigned multiply [0x50] by 4, result in W
IORLW    0x09           ; set bits 0 and 3 in WREG to one's
ANDLW    0xDF           ; clear bit 5  in WREG to zero
XORLW    0x80           ; ones complement  bit 7 in WREG
SLEEP                   ; Stop
END
```

As mentioned before, FINISH GOTO FINISH (unconditionally jumping to the same location) and the instruction SLEEP are equivalent to HALT instruction in other processors. Either can be used in the PIC18F as HALT in the assembly language program.

Example 6.19 Write a PIC18F assembly language program at address 0x100 to check whether an 8-bit signed number (x) in data register 0x60 is positive or negative. If the number is positive, then compute 16-bit value, $y1 = x^2$, and store The result in PRODH:PRODL. If the number is negative, then compute the 8-bit value, $y2 = 2x$. Store

the result in WREG. Do not use any logic instructions. Assume that the 8-bit number, x, is already loaded in data register 0x60.

Solution

```
            INCLUDE <P18F4321.INC>
            ORG     0x100
            MOVF    0x60, W         ; Move  x into WREG
            MOVFF   0x60, 0x70      ; Save x in 0x70
            RLCF    0x60, F         ; Rotate sign bit to carry to check whether 0 or 1
            BC      NEGATIVE        ; Branch if C = 1
            MULWF   0x60            ; Compute y1 and store in PRODH:PRODL
            GOTO    FINISH          ; Jump to FINISH
NEGATIVE    ADDWF   0x70, W         ; Compute y2 by adding x to itself
            SLEEP
            END
```

6.3.5 Bit Manipulation Instructions

The PIC18F has three bit manipulation instructions, and these are listed in Table 6.9. Next, we explain the Bit Manipulation instructions using the Access Bank and specifying F or W in place of 'd'.

- BCF F, b, a (BCF 0x50, 2) instruction clears the specified bit 'b' in register 'F' to zero. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank. As an example, consider BCF 0x50, 2.

  Prior to instruction execution: $[0x50] = 0x36 = 00110110_2$
  After Instruction execution: Bit 2 is cleared to 0. Hence, $[0x50] = 00110010_2 = 0x32$.
  The BCF instruction can be used to clear the carry flag to 0. For example, BCF STATUS, C will clear the carry flag in the status register to 0.

- BSF F, b, a (BSF 0x30, 5) instruction sets the specified bit 'b' in register 'F' to one. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank. As an example, consider BSF 0x30, 5.

  Prior to instruction execution: $[0x30] = 0x0F = 00001111_2$
  After instruction execution: Bit 5 is set to 1. Hence, $[0x30] = 00101111_2 = 0x2F$.
  The BSF instruction can be used to set the carry flag to 1. For example, BSF STATUS, C will set the carry flag in the status register to 1.

- BTG F, b, a (BTG 0x40, 2) instruction one's complements (toggles) the specified bit 'b' in register 'F'. If 'a' is '0', the Access Bank is selected. If 'a' is '1', the BSR is used to select the bank. As an example, consider BCF 0x40, 2.

  Prior to instruction execution: $[0x40] = 0x16 = 00010110_2$
  After instruction execution: Bit 2 is one's complemented from 1 to 0. Hence,
  $[0x40] = 00010010_2 = 0x12$.

Note that the BTG instruction can be used to toggle a specific bit in an I/O port. For example, BTG PORTB, 1 will toggle bit 1 of Port B. This may be useful sometimes in some I/O applications.

Example 6.20 Write a PIC18F assembly language program at address 0x100 that will multiply an 8-bit unsigned number in data register 0x50 by 4 to provide an 8-bit product,

TABLE 6.9 Bit Manipulation Instructions

Instruction	*Comment*
BCF F, b, a	Clear bit number 'b' to 0 in file register F
BSF F, b, a	Set bit number 'b' to 1 in file register F
BTG F, b, a	Toggle (one's complement) bit number 'b' in file register F

- All instructions in the above are executed in one cycle.
- The size of each instruction is one word.
- No flags are affected.
- b' can be from 0 to 7.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.

and then perform the following operations on the contents of data register 0x50 :

- set bits 0 and 3 to one without changing other bits in data register 0x50.
- clear bit 5 to zero without changing other bits in data register 0x50.
- one's complement bit 7 without changing other bits in data register 0x50.
- Use only "Rotate" and "Bit manipulation" instructions. Do not use any multiplication or any other instructions. Assume data is already in data register 0x50. Store the result in 0x50. Assume that a '1' is not shifted out of the most significant bit each time after rotating to the left.

This example is a repeat of Example 6.18, but uses "Bit manipulation instructions" instead of "Logic instructions".

Solution

```
INCLUDE  <P18F4321.INC>
ORG      0x100
BCF      STATUS, C
RLCF     0x50, F        ; Unsigned multiply [0x50] by 2
BCF      STATUS, C
RLCF     0x50, F        ; Unsigned multiply [0x50] by 4, result in W
BSF      0x50, 0        ; set bit 0 in [0x50] to one
BSF      0x50, 3        ; set bit 3 in [0x50] to one
BCF      0x50, 5        ; clear bit 5  in [0x50] to zero
BTG      0x50,7         ; ones complement  bit 7 in 0x50
SLEEP                   ; Stop
END
```

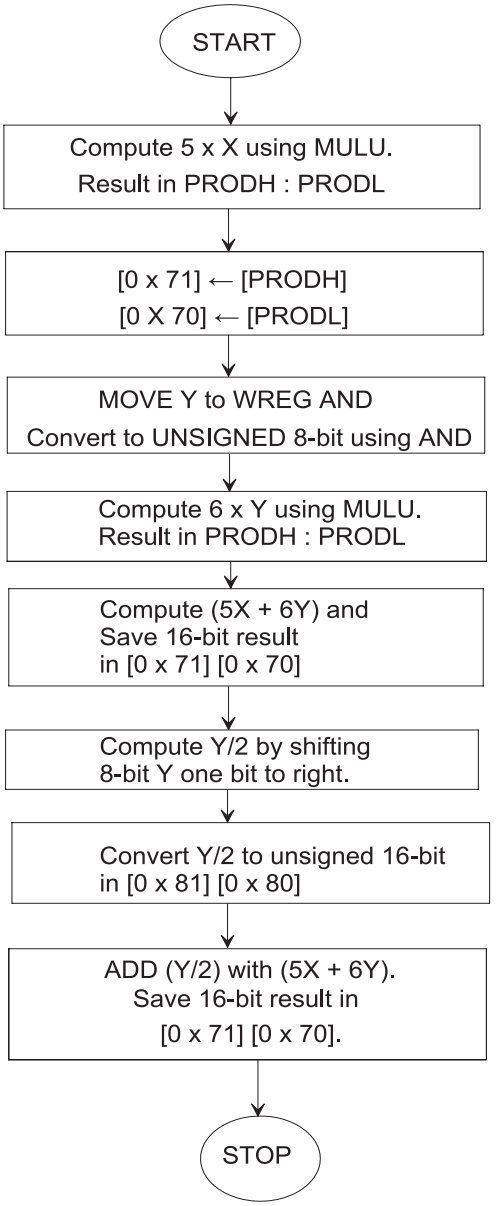
Example 6.21 Write a PIC18F assembly language program at address 0x100 that will perform : $5 \times X + 6 \times Y + [Y/2] \rightarrow$ [0x71][0x70] where X is an unsigned 8-bit number stored in data register 0x40 and Y is a 4-bit unsigned number stored in the upper 4 bits of data register 0x50. Discard the remainder of $Y/2$. Save the 16-bit result in 0x71 (upper byte) and in 0x70 (lower byte).

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Solution

(a) The flowchart is provided below:



(b) The PIC18F assembly language program is provided below:

```
        INCLUDE   <P18F4321.INC>
        ORG       0x100
        MOVF      0x40, W       ; MOVE X  TO WREG
        MULLW     5             ; COMPUTE UNSIGNED 16-BIT  5xX IN
                                ; PRODH:PRODL
        MOVFF     PRODH, 0x71   ; SAVE UPPER BYTE  OF 5xX  IN 0x71
        MOVFF     PRODL, 0x70   ; SAVE LOWER BYTE OF 5xX  IN 0x70
        SWAPF     0x50, W       ; MOVE Y TO LOW 4 BITS IN  WREG
        ANDLW     0x0F          ; CONVERT Y TO UNSIGNED 8-BIT IN WREG
        MOVWF     0x80          ; SAVE Y  FROM WREG TO 0x80
        MULLW     6             ; COMPUTE UNSIGNED 16-bit  6 x Y IN
                                ; PRODH:PRODL
        MOVFF     PRODL, W      ; MOVE  PRODL  INTO WREG
        ADDWF     0x70, F       ; ADD  LOW BYTES OF 5*X WITH 6*Y, SAVE
                                ; IN 0x70
        MOVFF     PRODH, W      ; MOVE  PRODH  INTO WREG
        ADDWFC    0x71, F       ; ADD  HIGH BYTES OF 5*X WITH 6*Y WITH
                                ; CARRY, AND SAVE IN 0x71
        ADDLW     0             ; CLEAR CARRY
        RRCF      0x80, F       ; COMPUTE  Y/2 , 8-BIT RESULT IN   0x80
        CLRF      0x81          ; CONVERT  Y/2 TO UNSIGNED 16-BIT IN
                                ; [0x81][0x80]
        MOVFF     0x80, W       ; MOVE LOW BYTE OF  Y/2 INTO WREG
        ADDWF     0x70, F       ; PERFORM 5 × X + 6 × Y + [Y/2] FOR LOW
                                ; BYTES, RESULT IN 0x70
        MOVFF     0x81, W       ; MOVE HIGH BYTE OF  Y/2 INTO WREG
        ADDWFC    0x71, F       ; PERFORM 5 × X + 6 × Y + [Y/2] FOR HIGH
                                ; BYTES PLUS CARRY , RESULT IN 0x71
FINISH  GOTO      FINISH        ; HALT
        END
```

QUESTIONS AND PROBLEMS

6.1 Write a PIC18F instruction sequence to implement the following C statement: c = a + b; Assume data registers 0x30, 0x40, and 0x50 store a, b, and c respectively.

6.2 Write a PIC18F instruction sequence to implement the following C statement: e = a + b + c - d; Assume data registers 0x30, 0x40, 0x50, 0x60, and 0x70 store a, b, c, d, and e respectively.

6.3 (a) Find the contents of data register 0x20 after execution of the MOVWF 0x20. Assume [WREG] = FFH prior to execution of this PIC18F MOVWF instruction.

(b) If [FSR0] = 0x0070, [FSR1] = 0x0060, [FSR2] = 0x0024, [WREG]= 0x2A, [0x24] = 0x0B, [0x60] = 0x43, and [0x70] = 0x57, what happens after execution of the PIC18F instruction: MOVWF INDF1?

6.4 Determine the contents of registers / memory locations affected by each of the following PIC18F instructions:

(a) MOVWF POSTDEC2

Assume the following data prior to execution of this MOVWF:

[FSR0] = 0x0050	[0x50] = 0x51
[FSR1] = 0x0025	[0x25] = $52
[FSR2] = 0x0075	[0x75] = 0x7F
[WREG] = 0xFE	

(b) MOVFF PLUSW1, 0x70

Assume the following data prior to execution of this MOVFF:
[FSR0] = 0x0020, [FSR1] = 0x0025, [FSR2] = 0x0028, [WREG] = 0x05, [0x20] = 0x05, [0x25] = 0x0A, [0x28] = 0xC2, [0x25] = 0x07, [0x2A] = 0x09, [0x2D] = 0x19

6.5 Find the machine code for the following PIC18F instruction sequence using Appendix D:

```
MOVLW   0x00
MOVWF   TRISB
NOP
MOVLW   0xAA
MOVWF   PORTB
SLEEP
```

6.6 Rewrite the following PIC18F instruction sequence with fewer instructions:

```
MOVLW   0x00
MOVWF   0x20
MOVLW   0xFF
MOVWF   0x22
```

6.7 Write a PIC18F instruction sequence that is equivalent to the following C code:

```
P- = Q;
```

6.8 Write a PIC18F assembley language program that is equivalent to the following C code:

```
if (x > 0)
  p = 0;
else
  p = 2p;
```

6.9 What is the content of WREG after execution of the following PIC18F instruction sequence?

```
MOVLW   0x33
ADDLW   0x77
DAW
```

6.10 Find two ways to clear [WREG] to 0 using:
(a) a single PIC18F instruction
(b) two PIC18F instructions

6.11 Using a single PIC18F instruction, clear the carry flag without changing the contents of any data registers, WREG, or other status flags.

6.12 Write the machine code for the following PIC18F instruction sequence:

```
        ORG     0x200
HERE    BRA     HERE
```

6.13 Write a PIC18F assembly language program at address 0x100 to add two 8-bit numbers (N1 and N2). Data register 0x20 contains N1. The low four bits of N2 are stored in the upper nibble of data register 0x21 while the high four bits of N2 are stored in the lower nibble of data register 0x21. Store result in data register 0x30.

6.14 Write a PIC18F assembly language program at address 0x100 to add two 24-bit data items in memory as shown in Figure P6.14. Store the result in data memory location addressed by 0x50. Do not use a loop. The operation with sample data is given by

```
        F1  91  B5
PLUS    07  A2  04
---------------------------------
        F9  33  B9
```

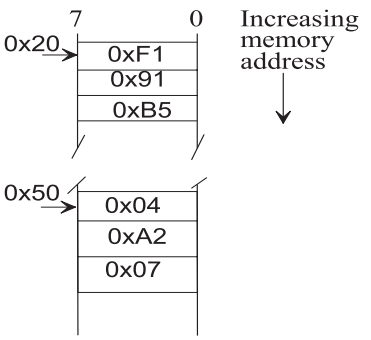


FIGURE P6.14

6.15 Write a PIC18F assembly language program to subtract two 16-bit numbers as follows:

```
        [0x40] [0x50]
MINUS   [0x20] [0x25]
        ---------------------
        [0x40] [0x50]
```

6.16 Write a PIC18F assembly language program at address 0x 0x100 to compute $\sum_{i=1}^{N} X_i Y_i$, where the X_i's and Y_i's are unsigned 8-bit numbers and $N = 4$. Store the 16-bit result in data registers 0x40 (low byte) and 0x41 (high byte). Assume that the X_i's are stored in data registers 0x50 through 0x53 while Y_i's are stored in data registers 0x70 through 0x73. Do not use a loop.

6.17 Write a PIC18F assembly program at address 0x50 to compute the following:

$$I = 6 \times J + (K/8)$$

where the data registers 0x31, 0x32, and 0x33 contain the 8-bit unsigned integers I, J and K. Store the 16-bit result into data registers [0x50] : [0x51]. Discard the remainder of $K/8$.

6.18 Write a PIC18F assembly language program at address 0x50 that will check whether the 8-bit unsigned number in WREG is odd or even. If the number is even, the program will clear all bits in data register 0x40 to 0's. On the other hand, if the number is odd, the program will all bits in data register 0x40 to 1's. Assume that the 8-bit numbers are already loaded into WREG.

6.19 Write a PIC18F assembly language program at address 0x70 to insert a '1' at bit 2 of WREG without changing the other bits if WREG contains a negative number. On the other hand, insert a '0' at bit 2 of WREG without changing the other bits if WREG contains a positive number. Assume that the 8-bit signed number is already loaded into WREG.

6.20 Write a PIC18F assembly language program at address 0x100 to check the parity of an 8-bit number in data register 0x70. If the parity is even, the program will store EE (hex) in data register 0x50. On the other hand, if the parity is odd, the program will store DD (hex) in data register 0x50.

6.21 Write a PIC18F assembly language at address 0x100 program to divide an unsigned 16-bit number by 2. Assume that the higher byte of the 16-bit number is already stored in data register [0x20], and lower byte in data register [0x21]. Discard remainder.

7

ASSEMBLY LANGUAGE PROGRAMMING WITH THE PIC18F: PART 2

In this chapter we provide the second part of the PIC18F's instruction set. Topics include Jump/branch, test/compare/skip, table read/write, subroutine, and system control instructions. Several assembly language programming examples using most of these instructions are provided. Finally, delay routines using PIC18F's instructions are covered.

7.1 PIC18F Jump/Branch instructions

These instructions include jumps and branches as listed in Table 7.1.

There is one unconditional JUMP such as GOTO k instruction, where 'k' is an address. Hence, the GOTO instruction uses "direct" or "absolute" addressing mode. There is also an unconditional branch such as BRA d instruction, where 'd' is a signed 11-bit offset (range -1024 decimal to +1023 decimal with 0 being positive) . Hence, this instruction uses "relative" addressing mode.

There are eight conditional branch conditions. They use "relative" addressing mode. For example, Consider Bcc d instruction where 'd' is an 8-bit signed offset (range -128 decimal to +127 decimal with 0 being positive) . Note that cc (condition code) in Bcc can be replaced by eight conditions providing eight instructions: BC, BNC, BZ, BNZ, BN, BNN, BOV, and BNOV. It should be mentioned that these instructions are applicable to signed numbers.

The instructions in Table 7.1 will now be discussed using numerical examples.

- GOTO k instruction unconditionally jumps to a 21-bit address (20-bit 'k') is loaded into the PC (bit 1 through bit 20) with the least significant bit (bit 0 of the PC) as 0. This will make the target address an even number. Note that the PIC18F instruction sizes are even multiple(s) of a byte (one or two words). The target address for the GOTO instruction must be an even number. Hence, the least significant bit of the PC is automatically fixed at 0. The 21-bit address with the GOTO instruction will allow the PIC18F to unconditionally jump to anywhere in the 2 Megabytes (2) of program memory. The "GOTO k" instruction is used to unconditional jump to any location in the program memory. Note that 'k' is 13-bit wide in the PIC18F4321 since its program address is 13-bit.
- Bcc d instruction (discussed in chapter 5) will branch if the condition cc is true, then the program. The 2's complement number '2 x d' is added to the PC. Since the PC will be incremented to fetch the next instruction, the new address will be PC + 2 + 2d. This instruction is then a two-cycle instruction. If the condition is false, then the next instruction is executed. Note that displacement 'd' is an 8-bit signed number.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

TABLE 7.1 PIC18F Jump/Branch instructions

Instruction	*Operation*
GOTO k	Unconditionally jumps to an address defined by the k. Uses direct or absolute mode.
Bcc d	If the condition cc is true, then (PC+2) + 2 x d → PC. The PC value is current instruction location plus 2. Displacement d is an 8-bit signed number.
BRA d	Branch always to (PC+2) + 2 x d, where PC value is current instruction location plus 2. d is a signed 11-bit number. This is an unconditional branch instruction with relative mode.

All instructions in the above except GOTO and BRA are executed in one cycle; GOTO and BRA are executed in two cycles. The size of each instruction except GOTO is one work; the size of GOTO is two words.

There are 8 conditions such as BC (Branch if Carry = 1), BNC (Branch if no Carry), BZ (Branch if result equals to zero, i.e., Z = 1), and BNZ (Branch if not equal, i.e., Z = 0), BN (Branch if negative i.e. N = 1), BNN (Branch if not negative i.e. N = 0), BOV (Branch if Overflow, i.e. OV = 1), BNOV (Branch if no Overflow, i.e. OV = 0).

In order to illustrate the concept of relative branching, the following example will be repeated from Chapter 5 for convenience. Hence, consider the PIC18F disassembled instruction sequence along with the machine code (all numbers in hex) provided below:

```
                                    1:          #INCLUDE<P18F4321.INC>
                                    2:          ORG       0x00
0000  0E02  MOVLW  0x2      3:      BACK        MOVLW     0x02
0002  0802  SUBLW  0x2      4:                  SUBLW     0x02
0004  E001  BZ     0x8      5:                  BZ        DOWN
0006  0E04  MOVLW  0x4      6:                  MOVLW     0x04
0008  0804  SUBLW  0x4      7:      DOWN        SUBLW     0x04
000A  E0FA  BZ  0           8:                  BZ        BACK
000C  0003  SLEEP           9:                  SLEEP
```

Note that all instructions, addresses, and data are chosen arbitrarily. The first branch instruction, BZ DOWN (line 5) at address 0x0004, has a machine code 0xE001. Upon execution of the instruction BZ (branch if Z-flag = 1), the PIC18F branches to label DOWN if Z = 1; the PIC18F executes the next instruction if Z = 0. The BZ instruction uses the relative addressing mode. This means that DOWN is a positive number (the number of steps forward relative to the current program counter) indicating a forward branch. The machine code 0xE001 means that the op-code for BZ is 0xE0 and the relative 8-bit signed offset value is 0x01 (+1). This is a positive value indicating a forward branch. Note that while executing BZ DOWN at address 0x0004, the PC points to address 0x0006 since the program counter is incremented by 2. This means that the program counter contains 0x0008. The offset 0x01 is multiplied by 2 and added to address 0x0006 to find the target branch address where the program will jump if Z = 1. The branch address can be calculated as follows:

```
  0x0006 =  0000 0000  0000 0110
+ 0x0002 =  0000 0000  0000 0010  (0x01 is multiplied by 2 , and sign-extended to 16 bits)
-----------------------------------------------
            0000 0000  0000 1000 = 0x0008
```

Hence, the PIC18F branches to address 0x0008 if Z = 1. This can be verified in the instruction sequence above.

Next, consider the second branch instruction, BZ BACK (line 8). Upon execution of BZ BACK, the PIC18F branches to label BACK if Z = 1; otherwise, the PIC18F executes the next instruction. The machine code of this instruction at address 0x000A is 0xE0FA, where 0xE0 is the op-code and 0xFA is the signed 8-bit offset value. The offset is represented as an 8-bit two's complement number. Since 0xFA is a negative number (-6), this is a backward jump. Note that while executing BZ BACK at address 0x000A, the PC points to address 0x000C since the program counter is incremented by 2. This means that the program counter contains 0x000C. The offset -6 is multiplied by 2, and then added to 0x000C to find the address value where the program will branch if Z = 1. The branch address is calculated as follows:

```
  0x000C = 0000 0000 0000 1100
+0xFFF4 = 1111 1111 1111 0100  (0xFA is multiplied by 2, and then sign-extended
                                to 16 bits)
-----------------------------------------------------
         1 0000 0000 0000 0000 = 0x0000
        ↗
Ignore final carry
```

The branch address is 0x0000, which can be verified in the instruction sequence above. As mentioned in Chapter 1, in order to add a 16-bit signed number with an 8-bit signed number, the 8-bit signed number must first be sign-extended to 16 bits. The two 16-bit numbers can then be added. Any carry resulting from the addition must be discarded. This will provide the correct answer.

- BRA d The BRA (Branch Always) instruction uses the relative addressing mode. As mentioned in Chapter 5, the BRA d instruction unconditionally branches to (PC + 2 + 2 x d) where offset 'd' is a signed 11-bit number specifying a byte range from -1024 decimal to + 1023 decimal with 0 being positive. Note that the PIC18F unconditional branch instruction "BRA d" uses relative mode and specifies 'd' as a signed 11-bit offset. All conditional branch instructions such as "BZ d" also uses relative mode but uses 'd' as a signed 8-bit offset.

7.2 PIC18F Test, Compare, and Skip instructions

Table 7.2 lists these instructions. Next, we explain the data movement instructions using the access bank and specifying F or W in place of 'd'.

- BTFSC F, b, a (BTFSC 0x40, 5) instruction tests the specified bit 'b' in the file register 'F', and skips the next instruction if the bit 'b' is 0. On the other hand, if bit 'b' is 1, the PIC18F executes the next instruction. Hence, the GOTO or BRA instruction is typically used after the BTFSC instruction. The BTFSC instruction is useful for conditional (polled) I/O. This topic is discussed later.

  Next, as an example, consider BTFSC 0x40, 5.

  Prior to execution of BTFSC 0x40, 5: [0x40] = F1H.

TABLE 7.2 PIC18F Test, Compare, and Skip instructions

Instruction	*Operation*
BTFSC F, b, a	Bit Test File register, Skip if Clear. If bit 'b' in register 'F' is '0', then the next instruction is skipped. If bit 'b' is '1', then the next instruction is executed.
BTFSS F, b, a	Bit Test File, Skip if Set. If bit 'b' in register 'F' is '1', then the next instruction is skipped. If bit 'b' is '0', then the next instruction is executed.
CPFSEQ F, a	Compare F with W, Skip if F = W. Compares the contents of data memory location 'F' to the contents of W by performing an unsigned subtraction. If 'F' = W, then the next instruction is skipped; else, the next instruction is executed.
CPFSGT F, a	Compare F with W, Skip if F > W. Compares the contents of data memory location 'F' to the contents of the W by performing an unsigned subtraction. If the contents of 'F' are greater than the contents of W, then the next instruction is skipped; else, the next instruction is executed.
CPFSLT F, a	Compare F with W, Skip if f < W. 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 next instruction is skipped; else, the next instruction is executed.
DECFSNZ F, d, a	Decrement F, Skip if Not 0. The contents of register 'F' are decremented by 1. If the result is not '0', then the next instruction is skipped; else, the next instruction is executed.
DECFSZ F, d, a	Decrement F, Skip if 0. The contents of register 'F' are decremented by 1. If the result is '0' ,then the next instruction is skipped; else, the next instruction is executed.
INCFSNZ F, d, a	Increment F, Skip if Not 0. The contents of register 'F' are incremented by 1. If the result is not '0', then the next instruction is skipped; else, the next instruction is executed.
INCFSZ F, d, a	Increment F, Skip if 0. The contents of register 'F' are incremented by 1. If the result is '0', then the next instruction is skipped; else, the next instruction is executed.
TSTFSZ F, a	Test F, Skip if 0. If 'F' = 0, then the next instruction is skipped; else, the next instruction is executed.

- All instructions in the above are executed in one to two cycles. No flags are affected. The size of each instruction is one word.
- a = 0 means that the data register is located in the access bank while a = 1 means that the contents of BSR specify the address of the bank.
- For destination: d = 0 means that the destination is WREG while d = 1 means that the destination is file register.

After execution of BTFSC 0x40, 5: Since bit 5 of [0x40] is 1, the BTFSC executes the next instruction.

The BTFSC instruction can be used to write the PIC18F assembly language instruction sequence for the following C segment:

```
if (x<0)
        y++ ;
```

```
else
        y--;
```

The PIC18F assembly language program can be written as follows:

```
        BTFSC   X, 7        ; Check sign bit (bit 7) of [X]. If  negative, increment [Y]
        BRA     NEG         ; If  [X] is positive, decrement [Y]
        DECF    Y           ; Increment [Y] if  [X] is negative
        BRA     NEXT
NEG     INCF    Y           ; Decrement [Y] if [X] is positive
NEXT            ----------  ;  Next instruction
```

- BTFSS F, b, a (BTFSS 0x70, 0) instruction tests the specified bit 'b' in the file register 'F', and skips the next instruction if the bit 'b' is '1'. On the other hand, if bit 'b' is '0', the PIC18F executes the next instruction. Like the BTFSC instruction, the GOTO or BRA instruction is typically used after the BTFSS instruction. The BTFSS instruction can be used for conditional (polled) I/O. This topic is discussed later.

  Next, as an example, consider BTFSS 0x70, 0.

  Prior to execution of BTFSS 0x70, 0 : [0x70] = 0xF1.

  After execution of BTFSS 0x70, 0: Since bit 0 of [0x70] is 1, the BTFSS skips the next instruction.

The same example for BTFSC, described in the last section, can be used to illustrate the BTFSS instruction. The assembly language program is written using the BTFSS instruction for the following C segment:

```
if (x<0)
y++ ;
else
y--;
```

The PIC18F assembly language instruction sequence using the BTFSS is provided below:

```
        BTFSS   X, 7        ; check sign bit (bit 7) of [X]. If  negative, increment [Y]
        BRA     POS         ; if  [X] is positive, decrement [Y]
        INCF    Y           ; increment [Y] if  [X] is negative
        BRA     NEXT
POS     DECF    Y           ; decrement [Y] if [X] is positive
NEXT    ---------           ; Next instruction
```

- PIC18F COMPARE instructions The CPFSEQ F, a instruction compares [F] with [WREG] by performing an unsigned subtraction, (both numbers are considered unsigned, skips the next instruction if [F] = [WREG]; if [F] ≠ [WREG], then the PIC18F executes the following instruction. The CPFSGT F, a, on the other hand, compares [F] with [WREG] by performing an unsigned subtraction, skips the next instruction if [F] > [WREG]; if [F] ≤ [WREG], then the PIC18F executes the next instruction. The CPFSLT F, a instruction compares [F] with [WREG] by performing an unsigned subtraction, skips the next instruction if [F] < [WREG]; if [F] ≥ [WREG], then the PIC18F executes the next instruction. Note that in all three cases, [F] and

[WREG] are considered as 8-bit unsigned (positive) numbers. The GOTO or BRA instruction is typically used after each of these COMPARE instructions. These instructions do not provide any result of subtraction and also, they do not affect any status flags.

In order to illustrate the use of one of the PIC18F COMPARE instructions, consider the following example. Suppose it is desired to find the number of matches for an 8-bit unsigned number in data register 0x80 with a data array (stored from low to high memory) of 50 bytes in memory pointed to by 0x50. Assume that data are already stored in memory. The following instruction sequence with CPFSEQ can be used :

```
        CLRF    0x40        ; Clear 0x40 to 0. Register 0x40 will hold
                            ;  the number of matches
        MOVLW   D'50'       ;  Move 50 into WREG
        MOVWF   0x20        ;  Initialize 0x20 with the array count, 50
        LFSR    0, 0x50     ; Initialize indirect pointer FSR0 with 0x50
        MOVF    0x80, W     ;  Move [0x80] to WREG
BACK    CPFSEQ  POSTINC0    ; Compare the number to be matched with [WREG]
        BRA     NOMATCH
        INCF    0x40        ; If there is a match, increment [0x40] by 1
NOMATCH DECF    0x20        ; Decrement [0x20] by 1
        BNZ     BACK        ; Go to BACK if Z is not 0
        ----------          ; Next instruction
```

Note that, in the above, CPFSEQ rather than SUBWF is used. This is because we are not interested in the subtraction result. Rather, we are interested in the number of matches. If SUBWF is used, one needs to load the number to be matched or the data byte from the array after each SUBWF; the subtraction result would erase the data Hence, CPFSEQ instead of SUBWF is ideal for the above example.

- DECFSNZ F, d, a (DECFSNZ 0x30, W or DECFSNZ 0x30, F) instruction decrements [F] by 1, and If the result is not '0', skips the instruction; else, the next instruction is executed. The DECFSZ , on the other hand, decrements [F] by 1, and if the result is '0', it skips the instruction; else, the next instruction is executed. The GOTO or BRA instruction is typically used after each of these instructions.

  Both instructions can be used to execute a certain loop 'n' times, where 'n' is an 8-bit number. This is another way of executing a loop without using the conditional branch instructions.

  For example, in order to execute a loop to obtain the 8-bit SUM (10 x A) by repeated addition, assume that the 8-bit unsigned number, 'A', is stored in register 0x70, and the SUM will be stored in 0x50. The following PIC18F instruction sequence using DECFSNZ will accomplish this:

```
        CLRF    0x50        ; Clear register 0x50 to 0 for SUM
        MOVLW   D'10'       ; Move 10 to WREG
        MOVWF   0x60        ; Initialize register 0x60 with 10
        MOVF    0x70,W      ; Move 'A' into WREG
REPEAT  ADDWF   0x50, F     ; Add 'A' 10 times, store result in 0x50
        DECFSNZ 0x60, F     ; Decrement counter, skip if reg 0x60 not 0
```

```
        GOTO    NEXT
        GOTO    REPEAT  ; Repeat addition until counter 0x60 is 0
NEXT    ----------      ; Next instruction
```

Using the DECFSZ , the above program to compute (10 x A) can be written as follows:

```
        CLRF    0x50    ; Clear register 0x50 to 0 for SUM
        MOVLW   D'10'   ; Move 10 to WREG
        MOVWF   0x60    ; Initialize register 0x60 with 10
        MOVF    0x70,W  ; Move 'A' into WREG
REPEAT  ADDWF   0x50, F ; Add 'A' 10 times, store result in 0x50
        DECFSZ  0x60, F ; Decrement counter, skip if 0
        GOTO    REPEAT  ; Repeat addition until counter 0x60 is 0
        ---------       ; Next instruction
```

- INCFSNZ or INCFSZ F, d, a (INCFSNZ or INCFSZ 0x40, W or 0x40, F) increments the contents of register 'F' by one, and skips the next instruction if the result is not 0; else, the next instruction is executed. The INCFSZ F, d, a , on the other hand, increments the contents of register 'F' by one, and skips the next instruction if the result is 0; else, the next instruction is executed. The GOTO or BRA instruction is typically used after each of these instructions.

  Both instructions can be used to execute a certain loop 'n' times, where 'n' is an 8-bit number. This is another way of executing a loop without using the conditional branch instructions.
  Consider the same examples of the last section using the DECFSNZ and DECFSZ instructions. As before, a loop will be executed to obtain the 8-bit SUM (10 x A) by repeated addition, assuming that the 8-bit unsigned number, 'A', is stored in register 0x70, and the SUM will be stored in 0x50. The INCFSNZ and INCFSZ will be used this time.

  The following PIC18F instruction sequence using INCFSNZ will accomplish this:

```
        CLRF    0x50    ; Clear register 0x50 to 0 for SUM
        MOVLW   0xF6    ; Move -10 to WREG
        MOVWF   0x60    ; Initialize register 0x60 with -10
        MOVF    0x70,W  ; Move 'A' into WREG
REPEAT  ADDWF   0x50, F ; Add 'A' 10 times, store result in 0x50
        INCFSNZ 0x60, F ; Increment counter, skip if reg 0x60 not 0
        GOTO    NEXT
        GOTO    REPEAT  ; Repeat addition until counter 0x60 is 0
NEXT    ----------      ; Next instruction
```

  Using the INCFSZ , the above program to compute (10 x A) can be written as follows:

```
        CLRF    0x50    ; Clear register 0x50 to 0 for SUM
        MOVLW   0xF6    ; Move -10 to WREG
        MOVWF   0x60    ; Initialize register 0x60 with -10
        MOVF    0x70,W  ; Move 'A' into WREG
```

```
REPEAT  ADDWF   0x50, F    ; Add 'A' 10 times, store result in 0x50
        INCFSZ  0x60, F    ; Increment counter, skip if counter is 0
        GOTO    REPEAT     ; Repeat addition until counter 0x60 is 0
        ---------          ; Next instruction
```

- TSTFSZ F, a (TSTFSZ 0x60) checks if [F] = 0, and skips the next instruction if it is zero (Z = 1); otherwise (Z = 0), the next instruction is executed. The TSTFSZ instruction can be used to check the contents of a register for 0 without using the conditional branch instruction. For example, a typical decrementing counter can be implemented using the conditional branch instruction such as BNZ as follows:

```
COUNTER EQU     0x40
        MOVLW   D'50'      ; Initialize loop counter with 50
        MOVWF   COUNTER
LOOP    DECF    COUNTER    ; Decrement COUNTER by 1
        BNZ     LOOP
                           ; Branch if [COUNTER] is not 0
        -------            ; Next instruction
```

The above loop can be implemented using the TSTFSZ instruction as follows:

```
COUNTER EQU     0x40
        MOVLW   D'50'      ; Initialize loop counter with 50
        MOVWF   COUNTER
LOOP    DECF    COUNTER    ; Decrement COUNTER by 1
        TSTFSZ  COUNTER    ; Test COUNTER for 0 and if not 0,
        GOTO    LOOP       ; go to LOOP. If [COUNTER] is 0, skip.
        -------            ; Next instruction
```

Example 7.1 Write a PIC18F assembly language program at 0x100 to subtract two 32-bit numbers as follows:

```
        [0x43] [0x42] [0x41] [0x40]
MINUS   [0x75] [0x74] [0x73] [0x72]
    ----------------------------------------
        [0x43] [0x42] [0x41] [0x40]
```

Assume data registers 0x40 through 0x43 contain the first 32-bit number while data registers 0x72 through 0x75 contain the second 32-bit number. Also, assume that data are already loaded into the data registers. Store the 32-bit result in data registers 0x40 (lowest byte) through 0x43 (highest byte). Use a loop. This is Example 6.12 using a loop.

Solution

```
        INCLUDE <P18F4321.INC>
        ORG     0x100
        MOVLW   4              ; Move WREG with 4
        MOVWF   0x50           ; Initialize 0x50 with loop count (4)
        LFSR 0, 0x0072         ; Initialize pointer FSR0 with 0x0072
        LFSR 1, 0x0040         ; Initialize pointer FSR1 with 0x0040
        ADDLW   0x00           ; Clear carry flag
START   MOVF    POSTINC0, W    ; Move byte into WREG and update pointer
        SUBWFB  POSTINC1, F    ; Subtract [WREG] and carry from byte,
```

```
                        ; store result in data register
DECFSZ   0x50, F        ; Decrement counter 0x50 by 1 and skip if [0x50] = 0
BNZ      START          ; Branch to START if Z is 0
SLEEP                   ; Halt
END
```

Note: In the above, DECFSZ is used instead of DECF. This is because DECF affects the carry flag while DECFSZ does not.

7.3 PIC18F Table Read/Write instructions

As mentioned before, the PIC18F program memory is 16 bits wide, while the PIC18F data memory space is 8 bits wide. Programs are stored in program memory with the data register contents defined using the assembler's DB directive. In order to execute a program requiring data, the data bytes stored in program memory using DB directive must be transferred to the specified data registers in data memory. Since the sizes of program memory and data memory are different, it would be difficult to accomplish this data transfer.

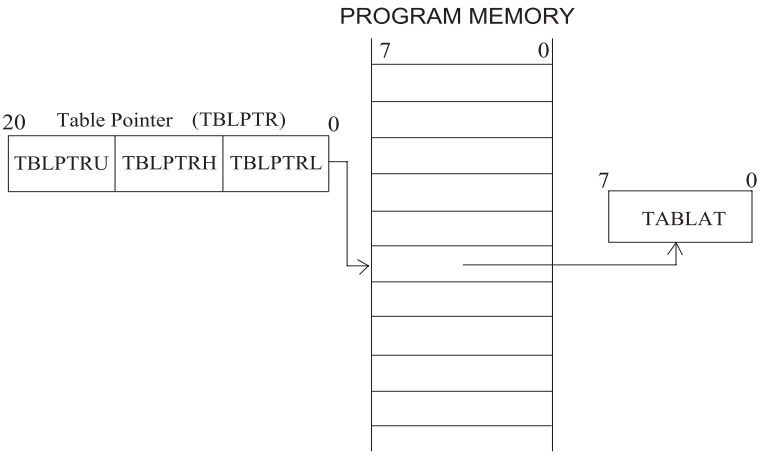


FIGURE 7.1 Table Read Operation (Instruction TBLRD*)

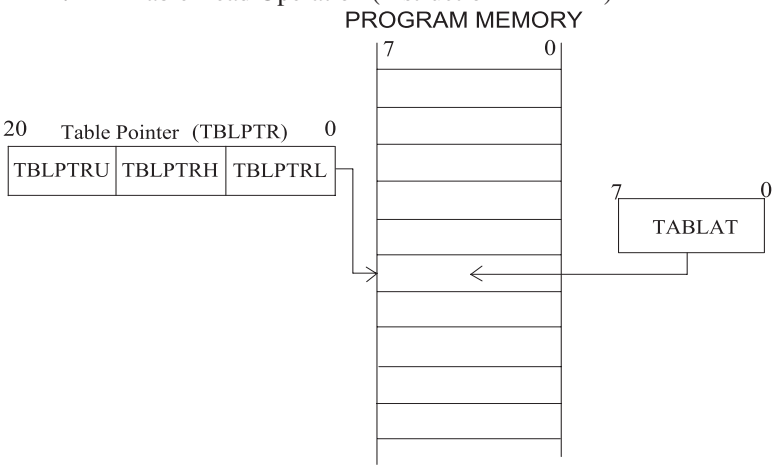


FIGURE 7.2 Table Write Operation (Instruction TBLWT*)

However, four table read and four table write instructions provided in the PIC18F facilitate transferring data between these two memory spaces through an 8-bit TABLAT (register called Table latch), and a 21-bit TBLPTR (pointer register called the Table pointer). The TBLPTR includes three registers, namely TABLPTRU (bits 20 through 16), TBLPTRH (bits 15 through 8), and TBLPTRL (bits 7 through 0). Since the PIC18F4321 uses 13-bit program address, the low 13 bits of the TBLPTR will be used as the pointer, and upper bits of TBLPTR will be 0's.

Two operations that allow the PIC18F to move bytes between the program memory and the data memory:

- Table Read (TBLRD)
- Table Write (TBLWT)

Table read operation retrieves data from program memory and places it into the data memory. Figure 7.1 shows the operation of a table read with program memory and data memory. The Table read operation is accomplished using the TBLRD instruction with four addressing modes (register indirect, postincrement, postdecrement, and predecrement).

Table write operation stores data from the data memory space into holding

TABLE 7.3 PIC18F Table Read/Write instructions

TBLRD*	Move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT.
TABLRD*+	Move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT, and then increment TBLPTR by 1.
TBLRD*-	Move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT, and then decrement TBLPTR by 1.
TBLRD+*	Increment TBLPTR by 1 and then move 8-bit data from program memory addressed by 21-bit TBLPTR into the 8-bit register TABLAT. .
TPLWT*	Move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR.
TBLWT*+	Move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR , and then increment TBLPTR by 1.
TBLWT*-	Move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR , and then decrement TBLPTR by 1.
TBLWT+*	Increment TBLPTR by 1 and then move 8-bit data from 8-bit register TABLAT into program memory addressed by 21-bit TBLPTR .

- All TBLRD and TBLWT instructions are executed in two cycles.
- The size of each instruction is one word.

registers in program memory. Figure 7.2 shows the operation of a table write with program memory and data memory. The Table write operation is accomplished using the TBLWT instruction with four addressing modes (register indirect, postincrement, postdecrement, and predecrement). Table operations work with data bytes . A table block containing data, rather than program instructions, is not required to be word aligned. Therefore, a table block can start and end at any byte address. Note that although the TBLPTR is 21-bit wide, the PIC18F4321 uses low 13 bits of the 21-bit address since the PIC18F4321 program memory address is 13-bit wide.

Table 7.3 lists the TBLRD and TBLWT instructions.

The table read/write instructions will now be explained in the following using numerical examples with similar data for each instruction:

- Consider TBLRD* instruction.

  Prior to execution of TBLRD* : [TBLPTR] = 0x000128, [TABLAT] = 0x24, [0x000128] = 0xF02.
  After execution of TBLRD*: [TABLAT] = 0xF2, [TBLPTR] = 0x000128 (unchanged), [0x000128] = 0xF2 (unchanged)

- Consider TBLRD*+ instruction.

  Prior to execution of TBLRD* +: [TBLPTR] = 0x000040, [TABLAT] = 0x24, [0x000040] = 0xF2
  After execution of TBLRD*+: [TABLAT] = 0xF2, [TBLPTR] = 0x000040, [0x000040] = 0xF2 (unchanged)

- Consider TBLRD*- instruction.

  Prior to execution of TBLRD*-: [TBLPTR] = 0x000052, [TABLAT] = 0x24, [0x000052] = 0xF2
  After execution of TBLRD*-: [TABLAT] = 0xF2, [TBLPTR] = 0x000051, [0x000052] = 0xF2 (unchanged)

- Consider TBLRD+* instruction.

  Prior to execution of TBLRD+*: [TBLPTR] = 0x0030, [TABLAT] = 0x24, [0x000031] = 0xF2
  After execution of TBLRD+*: [TABLAT] = 0xF2, [TBLPTR] = 0x000031, [0x000031] = 0xF2 (unchanged)

- Consider TBLWT* instruction.

  Prior to execution of TBLWT*: [TBLPTR] = 0x000120, [TABLAT] = 0x24, [0x000120] = 0x14
  After execution of TBLWT*: [0x000120] = 0x24, [TABLAT] = 0x24 (unchanged), [TBLPTR] = 0x000120 (unchanged)

- Consider TBLWT*+ instruction.

  Prior to execution of TBLWT*+: [TBLPTR] = 0x000030, [TABLAT] = 0x1F, [0x000030] = 0x02
  After execution of TBLWT*+: [0x000030] = 0x1F, [TBLPTR] = 0x000031, [TABLAT] = 0x1F (unchanged)

- Consider TBLWT*- instruction.

  Prior to execution of TBLWT*-: [TBLPTR] = 0x000318, [TABLAT] = 0x24, [0x000318] = 0xF2
  After execution of TBLWT*-: [0x000318] = 0x24, [TBLPTR] = 0x000317, [TABLAT] = 0x24 (unchanged)

- Consider TBLWT+* instruction.

  Prior to execution of TBLWT+*: [TBLPTR] = 0x000118, [TABLAT] = 0x24, [0x000119] = 0xF2
  After execution of TBLWT+*: [0x000119] = 0x24, [TBLPTR] = 0x000119, [TABLAT] = 0x24 (unchanged)

Example 7.2 Write a PIC18F assembly language program at address 0x100 to move the ASCII codes (30H through 39H) for BCD numbers 0 through 9 from program memory starting at address 0x200 (30H at address 0x200, 31H at 0x201, and so on) into data memory starting at address 0x40 (30H to be stored at address 0x40, 31H at 0x41, and so on).

Solution

```
            INCLUDE  <P18F4321.INC>
            ORG      0x100          ; #1 Starting address of program
COUNTER     EQU      0x20
            MOVLW    UPPER ADDR     ; #2 Move upper 5 bits (00H) of address
            MOVWF    TBLPTRU        ; #3 to TBLPTRU
            MOVLW    HIGH ADDR      ; #4 Move bits 15-8 (02H) of address
            MOVWF    TBLPTRH        ; #5 to  TBLPTRH
            MOVLW    LOW ADDR       ; #6 Move bits 7-0 (00H) of address
            MOVWF    TBLPTRL        ; #7 to TBLPTRL
            LFSR     0, 0x40        ; #8 Initialize FSR0 to 0x40 to be used as
                                    ; destination pointer in data memory
            MOVLW    D'10'          ; #9 Initialize COUNTER with 10
            MOVWF    COUNTER        ; #10 Move [WREG] into COUNTER
LOOP        TBLRD*+                 ; #11 Read data from program memory into
                                    ; TABLAT, increment TBLPTR by 1
            MOVF     TABLAT, W      ; #12 Move [TABLAT] into WREG
            MOVWF    POSTINC0       ; #13 Move W into data memory pointed to
                                    ; by FSR0, and then increment FSR0 by 1
            DECF     COUNTER, F     ; #14 Decrement COUNTER BY 1
            BNZ      LOOP           ; #15 Branch  if  Z = 0, else Stop
FINISH      BRA      FINISH         ; Stop
            ORG      0x200          ; Store ASCII codes in program memory
ADDR        DB       30H, 31H, 32H, 33H, 34H, 35H, 36H, 37H, 38H, 39H
            END
```

In the above program, the # sign along with the line number is placed before the comment in order to identify the specific line for explanation. Note that the directive DB used with the label ADDR at the end of the program stores the ASCII codes for BCD starting address 0x200 in the program memory.

TABLE 7.4 PIC18F Subroutine instructions

Instruction	*Operation*
CALL k, s	Call the subroutine at address k within the two Megabytes of program memory. First, return address (PC + 4) is pushed onto the return stack. If 's' = 1, the WREG, STATUS and BSR registers are also pushed into their respective shadow registers (internal to the CPU), WS, STATUSS and BSRS. If 's' = 0, these registers are unaffected (default). Then, the value 'k' is loaded into PC.
POP	Discards top of stack pointed to by SP and decrements STKPTR by 1.
PUSH	PUSHes or writes the PC onto the stack, and increments STKPTR by 1.
RETFIE	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. This instruction is normally used at the end of an interrupt service routine.
RETLW k	WREG is loaded with the eight-bit literal 'k'.The program counter is loaded from the top of the hardware stack (the return address).
RETURN s	Return from subroutine. The stack is popped and the top of the stack (TOS) 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, WREG, STATUS and BSR. If 's' = 0, these registers are not affected (default).

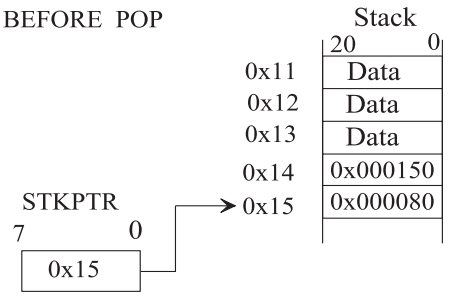


FIGURE 7.3 (a) PIC18F Hardware Stack with arbitrary data before execution of POP instruction

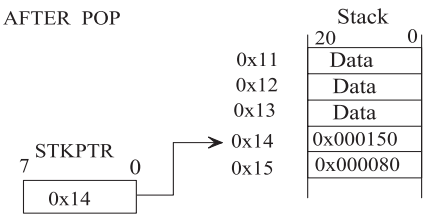


FIGURE 7. 3 (b) PIC18F Hardware Stack with arbitrary data after execution of POP instruction; address 0x15 is assumed to be free

Line #1 specifies the starting address of the program at 0x000200. Note that the programmer does not have to define the address, 0xFF5 for TABLAT. This is a predefined address (Special function register) by Microchip. The MPLAB assembler determines this internally. As mentioned before, the TBLPTR is divided into three registers as TBLPTRU (predefined address 0xFF8), TBLPTRH (predefined address 0xFF7), and TBLPTRL (predefined address 0xFF6). These are also Special function registers. The MPLAB assembler determines these addresses internally. Also, the MPLAB assembler identifies TBLPTRU as UPPER, TBLPTRH as HIGH, and TBLPTRL as LOW. Line #'s 2 through 7 initialize TBLPTR with the 21-bit address 0x000200.

Line #8 initializes data memory pointer FSR0 to 0x40. Line #'s 9 and 10 initialize COUNTER with 10. Line #11 reads a byte from program memory addressed by TBLPTR into TABLAT, and then increments TBLPTR by 1. Line #12 moves the contents of TABLAT into WREG. Line #13 moves [WREG] into the destination data memory address pointed to by FSR0, and then increments FSR0 by 1. Line #14 decrements [COUNTER] by 1. BNZ at line #15 checks the Z flag, and if Z = 0, the LOOP executed 10 times, and thus, the ASCII numbers 30H through 39H are transferred from program memory to data memory.

7.4 PIC18F Subroutine instructions

Table 7.4 lists PIC18F subroutine instructions. These include PUSH/POP and subroutine CALL/RETURN instructions. The subroutine instructions automatically use the "hardware stack" implemented by the manufacturer. The programmer can create "software stack" if needed for storing local variables. The "hardware stack" and "software stack" will be discussed in the next section in more detail.

- CALL and RETURN instructions are executed in two cycles.
- POP and PUSH are executed in one cycle.
- The size of each instruction except CALL is one word; the size of CALL instruction is two words.

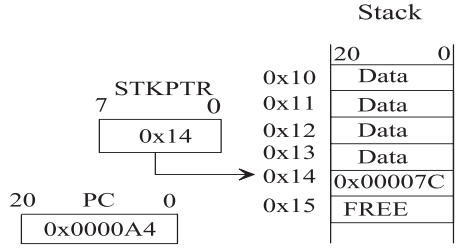


FIGURE 7.4 (a) PIC18F Hardware Stack with arbitrary data before execution of PUSH instruction

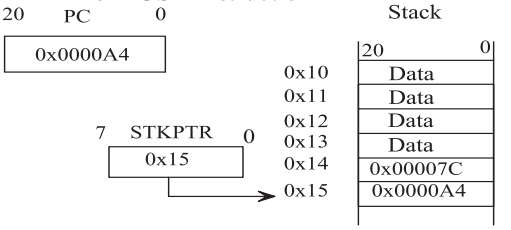


FIGURE 7.4 (b) PIC18F Hardware Stack with arbitrary data after execution of PUSH instruction

- POP instruction reads (pops) the TOS (Top Of Stack) value from the return stack and discards it; [STKPTR] is decremented by 1. The TOS value becomes the previous value that was pushed onto the return stack.

  As an example, consider the POP instruction with numerical data in the following:
  Prior to execution of the POP: [STKPTR] = 0x15, [0x15] = TOS (Top Of Stack) = 0x000080, Stack (1 level down), [0x14] = 0x000150
  After execution of the POP: [STKPTR] = 0x14, [0x14] = 0x000150
  Note that previous TOS (0x000080) is discarded, although 0x000080 is physically the content of 0x15. and previous Stack (1 level down) is the current TOS.
  Figures 7.3 (a) and (b) depict this.

- PUSH writes (pushes) PC+2 onto the top of the return stack; [STKPTR] is incremented by 1. The previous TOS value is pushed down on the stack.

  As an example, consider the PUSH instruction with numerical data in the following:
  Prior to execution of PUSH: [STKPTR] = 0x14, [0x14] = TOS (Top Of Stack) = 0x00007C, [PC+ 2] = 0x0000A4; that is the PUSH instruction is stored at address 0x0000A2.
  After execution of the PUSH: [STKPTR] = 0x15, [0x15] = TOS = 0x0000A4, Previous TOS (one level down), [0x14] = 0x00007C. Figures 7.4 (a) and (b) depict this.

- The "CALL k, s" with s = 0 (or CALL k) instruction is the simplest way of CALLing a subroutine; s = 0 is the default case. As an example, the CALL START instruction automatically pushes the current contents of the PC onto the stack, and loads PC with the label called START. Note that address START contains the starting address of the subroutine. The "RETURN s" instruction with s = 0 (or RETURN since s = 0 is default) pops the return address (PC pushed onto the stack by the CALL START instruction) from TOS, and loads PC with this address. Thus, control is returned to the main program, and program execution continues with the instruction next to the CALL START.

  Consider the following PIC18F program segment:

TABLE 7.5 PIC18F System control instructions

Instruction	*Operation*
CLRWDT	Clears watchdog timer to 0.
RESET	Resets all registers and flags to their 'hardware reset" values. The hardware RESET is performed upon activation of the PIC18F input pin. The RESET instruction provides software reset.
SLEEP	The PIC18F is put into sleep mode with the oscillator stopped.
NOP	No Operation

- All instructions in the above are exected in one cycle.
- The size of each instruction is one word.

Main Program		*Subroutine*	
—	SUB	—	; First instruction of subroutine
—		—	
—		—	
START CALL SUB		—	
—		—	
—		—	
—		RETURN	; Last instruction of subroutine

Here, the CALL SUB instruction in the main program calls the subroutine SUB. In response to the CALL instruction, the PIC18F pushes the current PC contents (START in this case) onto the stack and loads the starting address SUB of the subroutine into PC. After the subroutine is executed, the RETURN instruction at the end of the subroutine pops the address START from the stack into PC, and program control is then returned to the main program. Note that in the PIC18F MPLAB assembler, the starting address of the main program must be at a lower address than the starting address of the subaroutine.

7.5 PIC18F System Control instructions

The system control instructions are associated with the operation of the PIC18F. Table 7.5 lists these instructions.

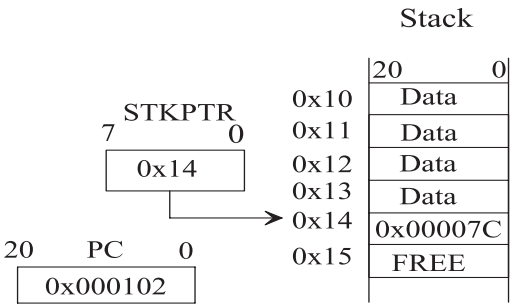


FIGURE 7. 5 (a) PIC18F Hardware Stack with arbitrary data before execution of CALL 0x000200 instruction

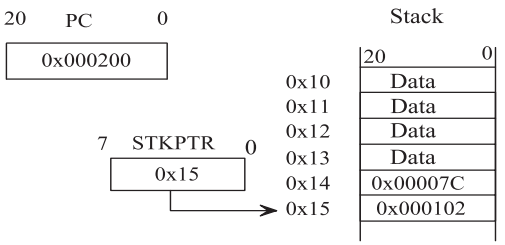


FIGURE 7. 5 (b) PIC18F Hardware Stack with arbitrary data after execution of CALL 0x000200 instruction

7.6 PIC18F Hardware vs. Software stack

As mentioned in Chapter 5, the PIC18F stack is a group of thirty one 21-bit registers to hold memory addresses. This stack (also called the "hardware stack") is neither part of data memory nor program memory. Note that the size of the stack (21-bit) is the same as the size of the PC (21-bit). The SP (Stack Pointer) is 5-bit wide in order to address 31 registers. In the PIC18F, after a POP (stack read), the SP is decremented by one while the SP is incremented by one after a PUSH (stack write). Also, the SP points to the last used address. Although with a 5-bit SP, 32 (2^5) registers are available, the PIC18F stack provides 31 registers with addresses 00001_2 through 11111_2.

In the PIC18F, these 31 registers are typically used to store return addresses after execution of the subroutine CALL instructions. Sometimes it may be necessary to save local variables before executing a subroutine CALL instruction. The 31 registers provided in the hardware stack may not be adequate. In that case, the user can create a "software stack" in the PIC18F using one or more of the three File Select Registers (FSRs) as the SP along with the registers in data memory. The programmer can implement a software stack using the POP instruction, and hence, will be able to properly manage the return stack. The PUSH instruction, on the other hand, allows implementing a software stack by modifying TOS and then pushing it onto the return stack.

The PIC18F software stack can be implemented by the programmer using data registers and FSR's (File Select Registers, FSR0-FSR2). The PIC18F uses one of the FSR's (referred to as FSRn with n = 0 to 2) as the software stack pointer, and supports software stack with the register indirect postincrement and predecrement addressing modes. In addition to the software stack pointers (FSRn), any bank of data registers can be used for

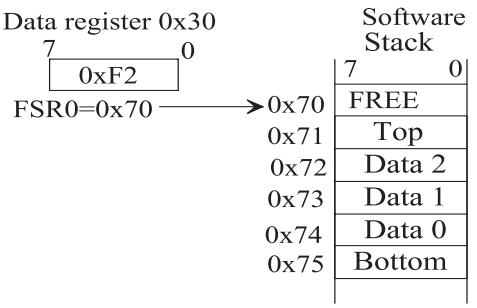


FIGURE 7.6 (a) PIC18F software stack with arbitrary data accessing stack from the top

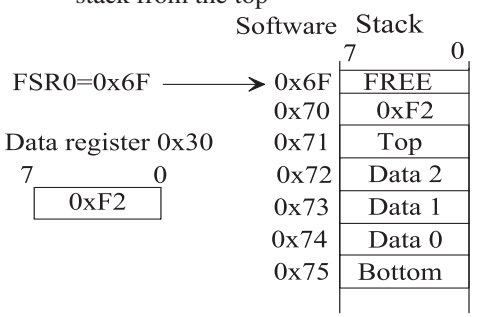


FIGURE 7.6 (b) PIC18F software stack with arbitrary data accessing stack from the top

the software stack. Subroutine CALLs, and interrupts automatically use the hardware stack pointer (STKPTR). As mentioned before, subroutine CALLs push the current PC onto the hardware stack; RETURN pops the PC from the hardware stack.

The PIC18F accesses the system stack from the top for operations such as subroutine calls or interrupts. This means that stack operations such as subroutine calls or interrupts access the hardware stack automatically from HIGH to LOW memory. As mentioned before, the low five bits of the STKPTR is used as the stack pointer for the hardware stack. Note that the STKPTR can be initialized using PIC18F MOVE instructions. For example, in order to load 0x14 into the STKPTR, the following PIC18F instruction sequence can be used:

```
MOVLW    0x14      ; Load 0x14 into WREG
MOVWF    STKPTR    ; Load [WREG] into STKPTR
```

Also, the STKPTR is incremented by 1 after a push and decremented by one after a pop. As an example, suppose that a PIC18F CALL instruction such as CALL 0x000200 located at [PC] = 0x000100 is executed; then, after execution of the subroutine call, the PIC18F will push the current contents of PC (0x000102) onto the hardware stack, and then load PC with 0x000200 (starting address of the subroutine specified in the CALL 0x000200 instruction). This is shown in Figures 7.5 (a) and (b). The RETURN instruction at the end of the subroutine will pop 0x000102 from the hardware stack into the PC and return control to the main program.

In the PIC18F, software stack can be created using appropriate addressing modes. Typical PIC18F memory instructions such as the MOVFF instruction can be used to access the stack. Also, by using one of the three FSRn (FSR0–FSR2) as software stack pointers, stacks can be filled from either HIGH to LOW memory or vice versa:

1. Filling a stack from HIGH to LOW memory (Top of the stack) is implemented with postdecrement mode for the push and preincrement mode for pop.

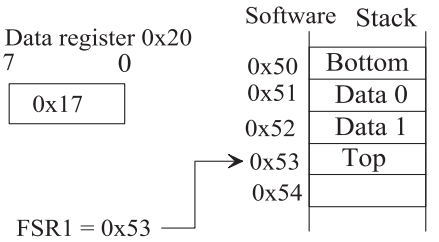


FIGURE 7.7 (a) PIC18F software stack with arbitrary data growing from HIGH to LOW memory before PUSH

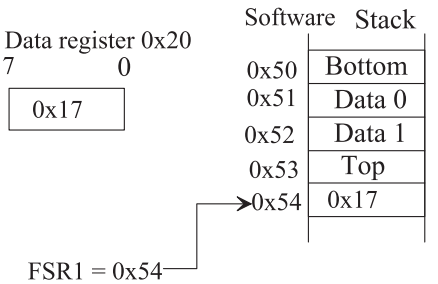


FIGURE 7.7 (b) PIC18F software stack with arbitrary data growing from HIGH to LOW memory after PUSH

2. Filling a stack from LOW to HIGH (Bottom of the stack) memory is implemented with preincrement for the push and postdecrement for pop.

The programmer can create a software stack growing from HIGH to LOW memory addresses using FSRn as the stack pointer. In this case, the stack is accessed from the top. To push the contents of a data register onto the software stack, MOVFF instruction with appropriate addressing modes can be used. For example, to push contents of a data register 0x30 using FSR0 as the stack pointer, the following PIC18F instruction sequence can be used:

```
LFSR    0, 0x0070          ; initialize FSR0 with 0x70 to be used as the SP
MOVFF   0x30, POSTDEC0     ; Push [0x30] to stack, decrement SP (FSR0) by 1
```

This is shown in Figures 7.6 (a) and (b). Figure 7.6 (a) shows the software stack with arbitrary data prior to execution of the above instructions. Figure 7.6 (b) shows the software stack with arbitrary data after execution of the above instructions. Note that the stack pointer, FSR0 in this case, is decremented by 1 after PUSH. Hence, the stack grows from HIGH to LOW addresses.

The 8-bit data 0xF2 can be popped from the stack into another data register 0x20, for example, using the MOVFF PREINC0, 0x20 instruction. Note that the stack pointer, FSR1 in this case, is incremented by 1 before POP.

Next, consider the stack growing from LOW to HIGH memory addresses in which the programmer also utilizes FSRn as the stack pointer. In this case, the stack is accessed from the bottom.

To push the 8-bit contents of a data register onto the software stack, MOVFF instruction with appropriate addressing modes can be used. For example, to push contents of a data register 0x20 using FSR1 as the stack pointer, the following PIC18F instruction sequence can be used:

```
LFSR    1, 0x0053          ; initialize FSR1 to 0x53 to be used as the SP
MOVFF 0x20, PREINC0        ; Increment SP (FSR1) by 1, Push [0x20] to stack.
```

This is shown in Figures 7.7 (a) and (b). Figure 7.7 (a) shows the software stack with arbitrary data prior to execution of the above instructions. Figure 7.7 (b) shows the software stack with arbitrary data after execution of the above instructions. Note that the stack pointer, FSR1 in this case, is incremented by 1 after PUSH.

The 8-bit data 0x17 can be popped from the stack into another data register 0x26, for example, using the MOVFF POSTDEC1, 0x26 instruction. Note that the stack pointer, FSR1 in this case, is decremented by 1 after POP.

Example 7.3 Write a PIC18F subroutine at address 0x100 to compute $Y = \sum_{i=1}^{N} X_i^2$. Assume the X_i's are 8-bit unsigned integers and $N = 4$. The numbers are stored in consecutive locations. Assume data register 0x40 points to the first element of the array for X_i's. The array elements are stored from LOW to HIGH memory addresses. The subroutine will store the 16-bit result (*Y)* in data memory registers 0x21 (high byte) and 0x20 (low byte), Also, write the main program at address 0x50 that will initialize STKPTR to 0x05, FSR0 to 0x0040, call the subroutine, compute *(Y*/4) by discarding the remainder, and then stop. Verify the correct operation of the programs using the MPLAB. Show screen shots as necessary.

Solution

```
        INCLUDE  <P18F4321.INC>
        ORG      0x50           ; Starting address of the main program
; LOAD FOUR ARBITRARILY CHOSEN DATA  INTO DATA MEM ADDR . 0x40 TO 0x43
        MOVLW    0x7E           ; Move 0x7E into WREG
        MOVWF    0x40           ; Move 0x7E into file register 0x40
        MOVLW    0x08           ; Move 0x08 into WREG
        MOVWF    0x41           ; Move 0x08 into file register 0x41
        MOVLW    0x23           ; Move 0x23 into WREG
        MOVWF    0x42           ; Move 0x23 into file register 0x42
        MOVLW    0x30           ; Move 0x30 into WREG
        MOVWF    0x43           ; Move 0x43 into file register 0x40
; INITIALIZE STKPTR, CALL SUBROUTINE, AND DIVIDE BY 4 BY RIGHT SHIFT TWICE
        MOVLW    0x05           ; Move 0x05 into WREG
        MOVWF    STKPTR         ; Load 0x05 into STKPTR
        LFSR     0,0x0040       ; Load File Select Register 0 with register 0x0040
        CALL     SQR            ; Call the function SQR
        BCF      STATUS, C      ; Clear the carry flag
        RRCF     0x21,F         ; Rotate right, or divide by 2
        BCF      STATUS, C      ; Clear the carry flag
        RRCF     0x20,F         ; Rotate right, or divide by 2
FINISH  GOTO     FINISH         ; Halt
        ORG      0x100          ; Starting address of the subroutine
SQR     MOVLW    0x00
        MOVWF    0x21           ; Clear register 0x21
        MOVWF    0x20           ; Clear register 0x20
        MOVLW    0x04
        MOVWF    0x60           ; Move 0x04 into register 0x60
BACK    MOVFF    INDF0, 0x50    ; Move the value addressed by FSR0 into
                                ; register 0x50. 0x50 is used as a holding register in
                                ; data memory Should not be confused with the
                                ; starting address 0x50 of the main program which
                                ; is in program memory of the PIC18F
        MOVF     POSTINC0, W    ; Move value addressed by FSR0  into WREG, and
                                ; then auto increment FSR0 by 1
        MULWF    0x50           ; Multiply WREG by 0x50, or X squared
        MOVF     PRODL, W       ; Move low byte of answer to WREG
        ADDWF    0x20, F        ; Sum with value in 0x20
        MOVF     PRODH, W       ; Move high byte of product to WREG
        ADDWFC   0x21, F        ; Sum with carry with value in 0x21
        DECFSZ   0x60, F        ; Decrement register 0x60 by one, skip next step if 0
        GOTO     BACK           ; Start over
        RETURN                  ; Return to main code
        END
```

Verification of the programs using MPLAB:

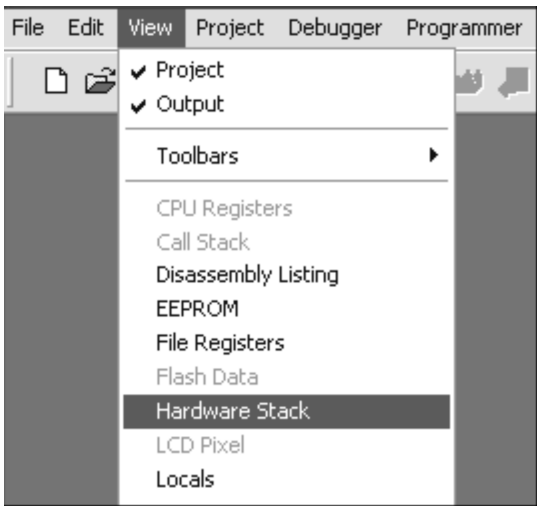
The following sample data are used:
[0x40] = 0x7E = 126 (decimal)
[0x41] = 0x08 = 8 (decimal)
[0x42] = 0x23 = 35 (decimal)
[0x43] = 0x30 = 48 (decimal)

$\sum_{i=1}^{N} X_i^2 = (126)^2 + (8)^2 + (35)^2 + (48)^2 = 15876 + 64 + 1225 + 2304 = 19469$ (decimal)

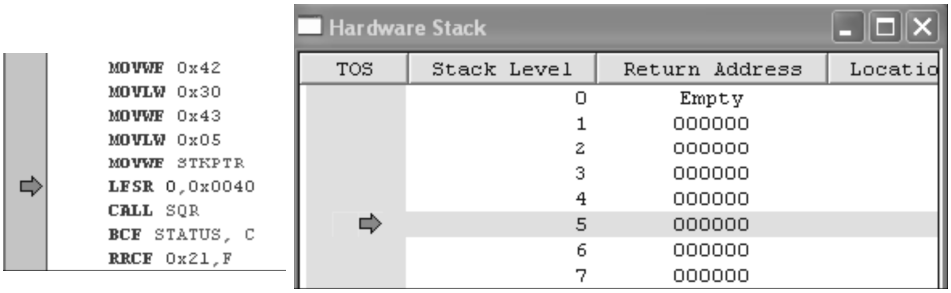
Hence, result = (19469)/4 = 4867.25 which is approximately 4867 (decimal) or 1303 (hex).

The following example will also demonstrate how the hardware stack on the PIC18F changes with the execution of the CALL and RETURN instructions.
The “”PIC18F Disassembly” function can be displayed from the “Disassembly Listing” option in the “View” menu as follows:

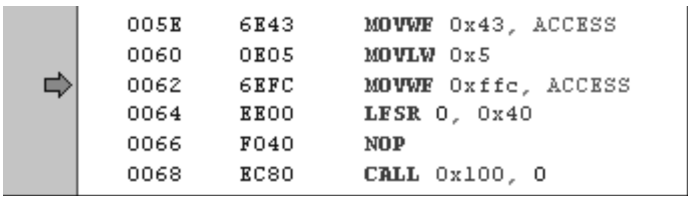
The “PIC18F hardware stack” can be displayed by selecting the “Hardware Stack” from the “View” menu as follows:



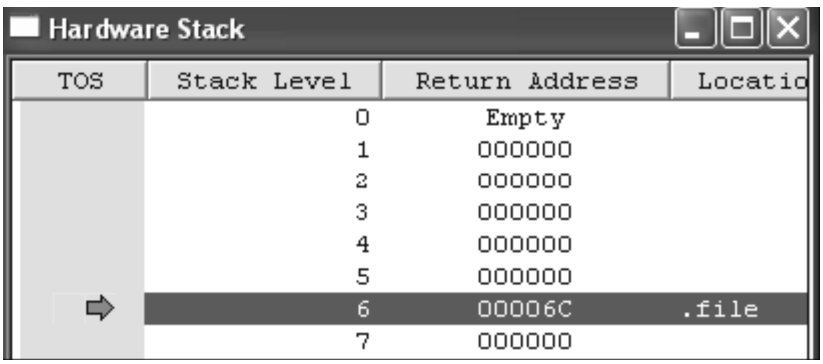
Next, each one of the instructions of the PIC18F assembly language main program is executed using the MPLAB debugger. After execution of the instruction “MOVLW STKPTR” in the main program, the value 0x05 is loaded into the STKPTR. The following screen shot displaying the contents of the hardware stack verifies this:



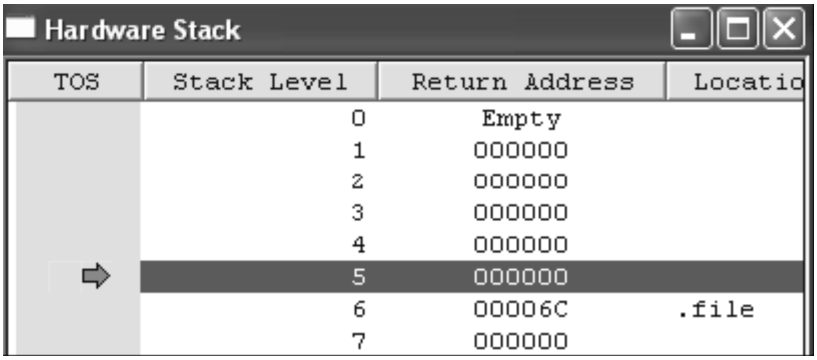
The next screen shot shows that "CALL SQR" instruction is located at address 0x0068 in the program memory. Since the CALL instruction is 4 bytes (two words) wide, the program counter will contain the address of the next instruction which is (0x0068 + 4 = 0x006C). The screen shot to verify this is provided below:



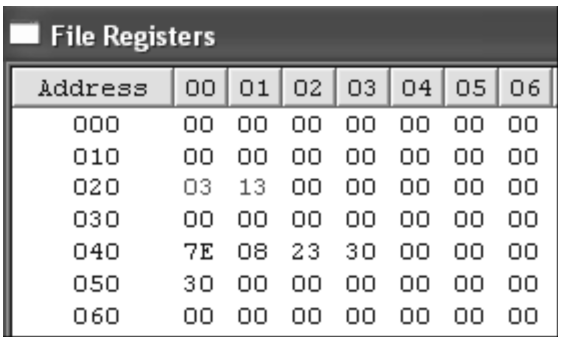
When the "CALL SQR" is executed, the following screen shot shows that the return address 00006C is PUSHed onto the hardware stack, and the STKPTR is incremented by 1 to contain 6 as follows:



After execution of the "RETURN" instruction at the end of the subroutine, the returned is popped from the hardware, and is placed in the Program Counter so that the program goes back to the main program. The STKPTR is then decremented by 1 to contain 5 as follows:



After execution of the main program, and the subroutine, the final answer (1303 hex) is stored in the file registers 0x21 (high byte) and 0x20 (low byte) as follows:



7.7 Multiplication and Division algorithms

As mentioned in chapter 1, an *Unsigned binary number* has no arithmetic sign, and therefore, it is always positive. Typical examples are your age or a memory address, which are always positive numbers. An 8-bit unsigned binary integer represents all numbers from 00_{16} through FF_{16} (0_{10} through 255_{10}).

A *signed binary number*, on the other hand, includes both positive and negative numbers. It is represented in the microcontroller in two's-complement form. For example, the decimal number +15 is represented in 8-bit two's-complement form as 00001111 (binary) or 0F (hexadecimal). The decimal number -15 can be represented in 8-bit two's-complement form as 11110001 (binary) or F1 (hexadecimal). Also, the most significant bit (MSB) of a signed number represents the sign of the number. For example, bit 7 of an 8-bit number represents the signs of the respective numbers. A "0" at the MSB represents a positive number; a "1" at the MSB represents a negative number. Note that the 8-bit binary number 11111111 is 255_{10} when represented as an unsigned number. On the other hand, 11111111_2 is -1_{10} when represented as a signed number.

As mentioned before, the PIC18F includes only unsigned multiplication instruction. The PIC18F instruction set does not provide any instructions for signed multiplication, unsigned and signed division instructions. These algorithms are covered in details in Chapter 4. A summary of the algorithms is provided in this section for convenience. The PIC18F assembly language programs using these algorithms are written in this section.

7.7.1 Signed Multiplication algorithm

Signed multiplication can be performed using various algorithms. A simple algorithm follows. Assume that M (multiplicand) and Q (multiplier) are in two's-complement form. Assume that M_n and Q_n are the most significant bits (sign bits) of the multiplicand (M) and the multiplier (Q), respectively. To perform signed multiplication, proceed as follows:

1. If $M_n = 1$, compute the two's complement of M.; else, keep M unchanged.
2. If $Q_n = 1$, compute the two's complement of Q; else, keep Q unchanged.
3. Multiply the n - 1 bits of the multiplier and the multiplicand using unsigned multiplication.
4. The sign of the result, $S_n = M_n \oplus Q_n$.
5. If $S_n = 1$, compute the two's-complement of the result obtained in step 3; else, keep result unchanged.

Next, consider a numerical example. Assume that M and Q are two's-complement numbers. Suppose that $M = 1100_2$ and $Q = 0111_2$. Because $M_n = 1$, take the two's-complement of $M = 0100_2$; because $Q_n = 0$, do not change Q. Multiply 0111_2 and 0100_2 using the unsigned multiplication method discussed before. The product is 00011100_2. The sign of the product $S_n = M_n \oplus Q_n = 1 \oplus 0 = 1$. Hence, take the two's-complement of the product 00011100_2 to obtain 11100100_2, which is the final answer: -28_{10}.

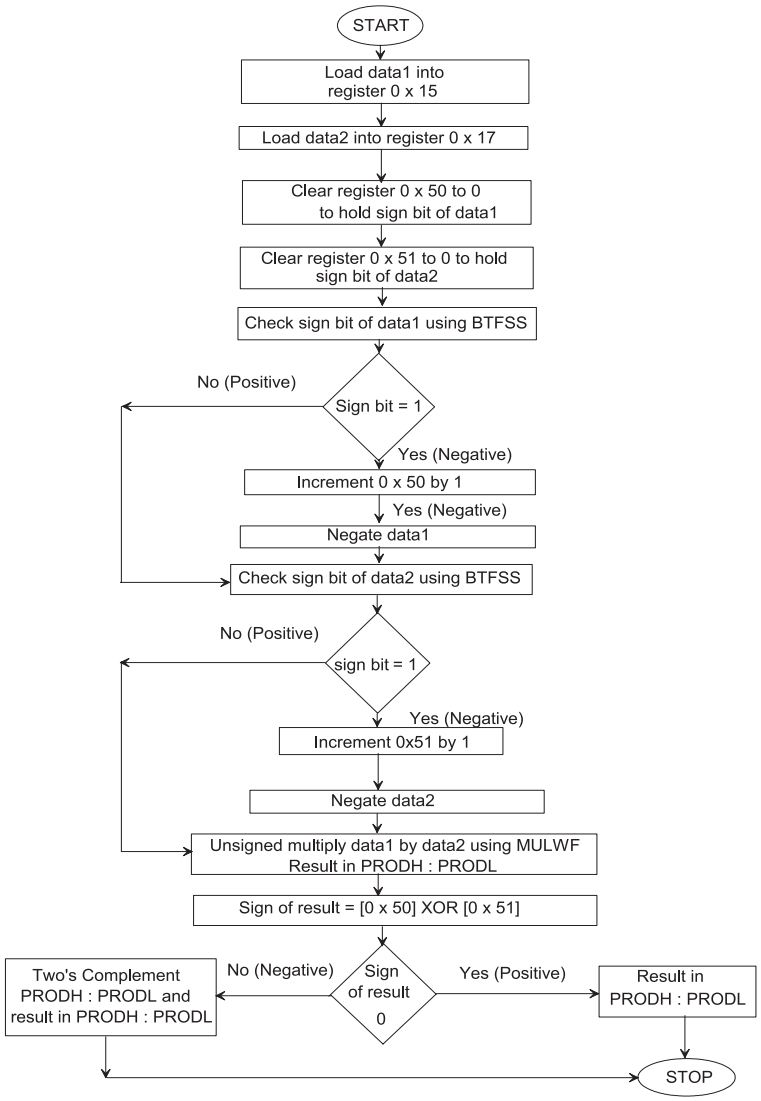
Example 7.4 Using the signed multiplication algorithm just described, it is desired to multiply two 8-bit signed numbers stored in data registers 0x15 and 0x17. Save the 16-bit result in PRODH:PRODL.

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Solution

(a)



(b)

```
            INCLUDE <P18F4321.INC>
            ORG     0x100
MULT1       EQU     0x15
MULT2       EQU     0x17
SIGN1       EQU     0X50
SIGN2       EQU     0X51
            MOVLW   0xFE        ; Load first 8-bit data (-2) in WREG
            MOVWF   MULT1       ; Save in MULT1
            MOVLW   0xFC        ; Load 2nd 8-bit data (-4) in WREG
            MOVWF   MULT2       ; Save in MULT2
            CLRF    SIGN1       ; Clear [SIGN1] to 0
```

```
                CLRF    SIGN2       ; Clear [SIGN2] to 0
; STEPS 1 AND 2 OF THE ALGORITHM OF  SECTION 7.7.1
                BTFSS   MULT1, 7    ; Check sign bit 7  for 1 for 1st #
                BRA     NEG         ; If sign = 0, branch to check sign of 2nd #
                INCF    SIGN1       ; Increment [SIGN1] if  sign of 1st# = 1
                NEGF    MULT1       ; and take 2's complement of [MULT1]
        NEG     BTFSS   MULT2, 7    ; Check sign bit 7  for 1 for 2nd #
                BRA     POSMUL      ; If both sign= 0, branch for unsigned mul
                INCF    SIGN2       ; Increment [SIGN2] if  sign of 2nd # = 1
                NEGF    MULT2       ; and take 2's complement of [MULT2]
; STEP 3 OF THE ALGORITHM OF  SECTION 7.7.1
        POSMUL  MOVF    MULT1, W    ; Move [MULT1] to WREG
                MULWF   MULT2       ; Unsigned product in PRODH:PRODL
                MOVF    SIGN1, W    ; Move [SIGN1] to WREG
                XORWF   SIGN2       ; Compute sign of the result
                BTFSS   SIGN2, 0    ; If sign of result is 0, result in
                BRA     FINISH      ; PRODH:PRODL and Stop
                COMF    PRODL       ; For negative result, take comp of PROD
; STEPS  4  AND 5  OF THE ALGORITHM OF  SECTION 7.7.1
                COMF    PRODH       ; Take 2's complement of PRODL
                MOVLW   1
                ADDWF   PRODL
                MOVLW   0
                ADDWFC  PRODH, F    ; Result in PRODH:PRODL in 2's comp
        FINISH  SLEEP
                END
```

7.7.2 Unsigned Division algorithm

The 8-bit by 8-bit unsigned division can be performed using the repeated subtraction algorithm. For example, consider dividing 7_{10} by 3_{10} as follows:

Dividend	Divisor	Subtraction Result	Counter
7_{10}	3_{10}	7 - 3 = 4	1
		4 - 3 = 1	1 + 1 = 2

Quotient = counter value = 2

Remainder = subtraction result = 1

Here, one is added to a counter whenever the subtraction result is greater than the divisor. The result is obtained as soon as the subtraction result is smaller than the divisor. The unsigned division algorithm can be summarized as follows:

First, load the dividend and divisor into the data registers, and initialize a counter to 0 to hold quotient (number of times divisor can be subtracted until subtraction result is less than the divisor). Data register storing the dividend will eventually contain the quotient (result of subtraction). The algorithm can be verified using numerical data.

1. Compare the dividend with the divisor for equality.
2. If equal**,** increment the counter by 1, and then perform (dividend - divisor). Store the subtraction result in the data register holding the dividend. If not equal, go to step 3.

3. Compare if dividend > divisor. If greater, increment the counter by 1 and then perform (dividend - divisor). Store the subtraction result in the data register holding the dividend.

4. Go to Step 1, and repeat steps 1 through 3 until the subtraction result is less than or equal to 0.

5. When the subtraction result in dividend is less than or equal to the divisor, go to halt. The counter will contain the quotient (number of times divisor can be subtracted until subtraction result is less than the divisor). The data register holding the dividend will contain the remainder (result of subtraction).

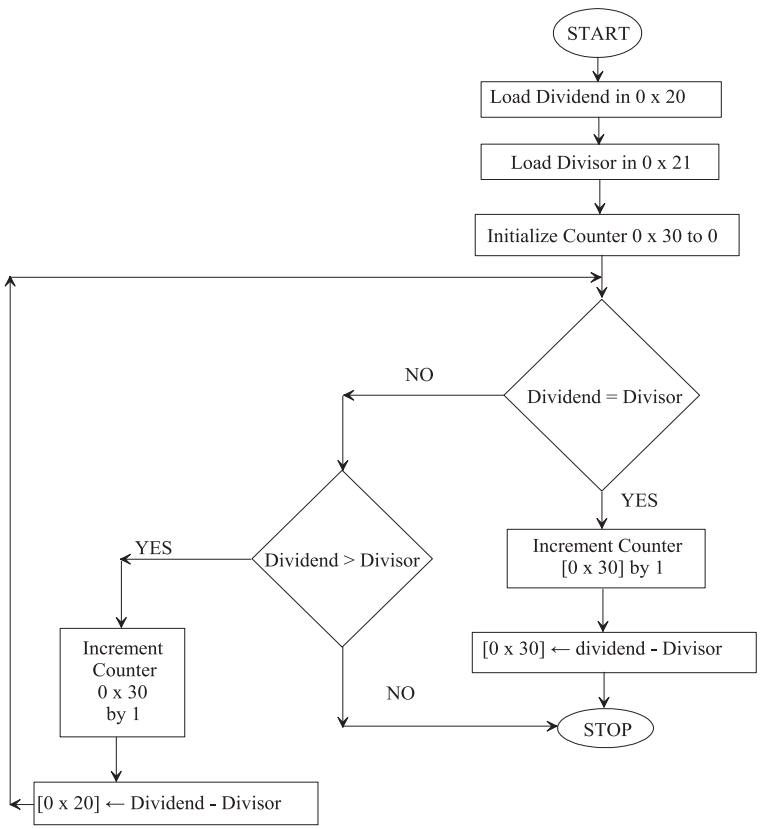
Example 7.5 Using the unsigned division algorithm just described, it is desired to divide an 8-bit unsigned number (dividend) stored in data register 0x20 by another 8-bit unsigned number (divisor) stored in data register 0x30. Save the 16-bit result in data registers 0x71 (remainder high byte) and 0x70 (quotient in low byte).

(a) Flowchart the problem.

(b) Convert the flowchart to PIC18F assembly language program starting at address 0x100.

Solution

(a)



```
(b)         INCLUDE <P18F4321.INC>
            ORG     0x100
DIVIDEND    EQU     0x20
DIVISOR     EQU     0x21
COUNTER     EQU     0x30
            MOVLW   16              ;  Dividend in WREG
            MOVWF   DIVIDEND        ; Store dividend in 0x20
            MOVLW   4               ; Divisor in WREG
            MOVWF   DIVISOR         ; Store divisor in 0x21
            CLRF    COUNTER         ; Clear Counter to 0
; STEPS 1 AND 2 OF THE ALGORITHM OF SECTION 7.7.2
BACK        CPFSEQ  DIVIDEND        ; If  dividend equals divisor, skip next instr.
            BRA     RESULT          ; If not equal, branch to RESULT
            INCF    COUNTER, F      ; Increment  [0x20] by 1
            SUBWF   DIVIDEND, F     ; Subtract divisor from dividend, result in 0x20
            BRA     FOREVER         ; Go to Halt
; STEPS 3 , 4 AND 5 OF THE ALGORITHM OF SECTION 7.7.2
RESULT      CPFSGT  DIVIDEND        ; If  dividend greater than divisor, skip next inst.
            BRA     FOREVER         ; Quotient in 0x30, Remainder in 0x20, halt
            INCF    COUNTER, F      ; Increment [0x20] by 1
            SUBWF   DIVIDEND, F     ; Subtract divisor from dividend, result in 0x20
            BRA     BACK            ; Repeat
FOREVER     GOTO    FOREVER         ; Halt
            END
```

7.7.3 Signed Division algorithm

The 8-bit by 8-bit signed division algorithm uses the equation for division:
Dividend = Quotient x Divisor + Remainder

Signed division can be performed using various algorithms. A simple algorithm follows. Assume that DV (Dividend) and DR *(*Divisor) are in two's-complement form. For the first case, perform unsigned division using repeated subtraction of the magnitudes without the sign bits. The sign bit of the quotient is determined as $DV_n \oplus DR_n$, where DV_n and DR_n are the most significant bits (sign bits) of the dividend (DV) and the divisor (DR) respectively. To perform signed division, proceed as follows:

1. If $DV_n = 1$, compute the two's complement of DV, else keep DV unchanged.
2. If $DR_n = 1$, compute the two's complement of DR, else keep DR unchanged.
3. Divide the n - 1 bits of the dividend by the divisor using unsigned division algorithm (repeated subtraction).
4. The sign of the Quotient,$Q_n = DV_n \oplus DR_n$. The sign of the remainder is the same as the sign of the dividend unless the remainder is zero.
5. If $Q_n = 1$, compute the two's-complement of the quotient obtained in step 3, else keep the quotient unchanged.

Example 7. 6 Write a PIC18F assembly language program at address 0x100 to divide an 8-bit signed number (dividend) in register 0x30 by another 8-bit signed number (divisor) in register 0x40. Use the signed division algorithm described in the last section.

Solution

```
            INCLUDE <P18F4321.INC>
            ORG     0x100
COUNTER     EQU     0x20
DIVIDEND    EQU     0x30
DIVISOR     EQU     0x40
SIGN1       EQU     0X50
SIGN2       EQU     0X51
            MOVLW   4               ; Load 8-bit data (+4) in WREG
            MOVWF   DIVIDEND        ; Save in DIVIDEND
            MOVLW   0xFE            ; Load 8-bit data (-2) in WREG
            MOVWF   DIVISOR         ; Save in DIVISOR
            CLRF    SIGN1           ; Clear [SIGN1] to 0
            CLRF    SIGN2           ; Clear [SIGN2] to 0
; STEPS 1 AND 2 OF THE ALGORITHM OF SECTION 7.7.3
            BTFSS   DIVIDEND, 7     ; Check sign bit 7  for 1 for 1st #
            BRA     NEG             ; If sign= 0, branch to check sign of 2nd#
            INCF    SIGN1           ; Increment [SIGN1] if  sign of 1st# = 1
            NEGF    DIVIDEND        ; take 2's complement of [DIVIDEND]
NEG         BTFSS   DIVISOR, 7      ; Check sign bit 7  for 1 for [DIVISOR]
            BRA     POSDIV          ; If both signs = 0, branch for unsigned division
            INCF    SIGN2           ; Increment [SIGN2] if  sign of 2nd# = 1
            NEGF    DIVISOR         ; and take 2's complement of [DIVISOR]
; STEP 3 OF THE ALGORITHM OF  SECTION 7.7.3
POSDIV      MOVF    DIVISOR, W      ; Load divisor into WREG
            CLRF    COUNTER         ; Clear Counter to 0
BACK        CPFSEQ  DIVIDEND        ; If  dividend equals divisor, skip next instr.
            BRA     RESULT1         ; If not equal, branch to RESULT
             INCF   COUNTER, F      ; Increment  [0x20] by 1
            SUBWF   DIVIDEND, F     ; Subtract divisor from dividend, result in 0x20
RESULT1     CPFSGT  DIVIDEND        ; If  dividend greater than divisor, skip next inst.
            BRA     RESULT          ; Quotient in 0x20, Remainder in 0x30
            INCF    COUNTER, F      ; Increment [0x20] by 1
            SUBWF   DIVIDEND, F     ; Subtract divisor from dividend, result in 0x30
            BRA     BACK            ; Repeat
; STEPS  4 AND 5 OF THE ALGORITHM OF  SECTION 7.7.3
RESULT      MOVF    SIGN1, W        ; Move [SIGN1] to WREG
            XORWF   SIGN2           ; Compute sign of the result
            BTFSS   SIGN2, 0        ; If sign of the quotient is 0, result in
            BRA     FINISH          ; 0x70 and Stop
            NEGF    0x20            ; For negative result, take comp of [0x20]
            BTFSS   SIGN1, 0        ; Check sign of  DIVIDEND
            BRA     FINISH          ; If plus, positive remainder in 0x30
            NEGF    0x30            ; If negative, negate remainder in 0x30
FINISH      SLEEP
            END
```

7.8 Advanced Programming Examples

In this section, more challenging assembly language programming examples using the PIC18F instruction set will be provided.

Example 7.7 Write a subroutine in PIC18F assembly language program at address 0x200 to find the n^{th} number (for example, n = 0 to 6) of the Fibonacci sequence. The subroutine will obtain the desired Fibonacci number using a lookup table stored starting at an address 0x150 in the program memory. Also, write the main program at address 0x100 which will transfer Fibonacci array from program memory stored at address starting at 0x200 to data memory stored starting at address 0x40, initialize STKPTR to 0x15, store a number (0 to 6) in WREG, initialize data pointer FSR1 to 0x40, call the sbroutine, and stop. The Fibonacci sequence for n = 0 to 6 is provided below:

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Solution

```
; MAIN  PROGRAM
                    INCLUDE  <P18F4321.INC>
                    ORG      0x100          ; Starting address of  the main program
          COUNTER   EQU      0x20
; FIBONACCI ARRAY TRANSFER FROM PROGRAM MEMORY TO DATA MEMORY
                    MOVLW    UPPER ADDR ; Move upper 5 bits (00H) of address
                    MOVWF    TBLPTRU    ;  to TBLPTRU
                    MOVLW    HIGH ADDR  ; Move bits 15-8 (01H) of address
                    MOVWF    TBLPTRH    ; to TBLPTRH
                    MOVLW    LOW ADDR   ; Move bits 7-0 (50H) of address
                    MOVWF    TBLPTRL    ; to TBLPTRL
                    LFSR     0, 0x40    ; Initialize FSR0 to 0x40 to be used as
                                        ; destination pointer in data memory
                    MOVLW    D'7'       ; Initialize COUNTER with 7
                    MOVWF    COUNTER    ; Move [WREG] into COUNTER
          LOOP      TBLRD*+             ; Read data from program memory into
                                        ; TABLAT, increment TBLPTR by 1
                    MOVF     TABLAT, W  ; Move [TABLAT] into WREG
                    MOVWF    POSTINC0   ; Move W into data memory addressed
                                        ; by FSR0, and then increment FSR0 by 1
                    DECF     COUNTER, F ; Decrement COUNTER BY 1
                    BNZ      LOOP       ; Branch  if  Z = 0
; INITIALIZE STKPTR, LOAD n, INITIALIZE DATA POINTER, CALL SUBROUTINE
                    MOVLW    0x15       ; Initialize STKPTR to 0x15
                    MOVWF    STKPTR
```

```
                MOVLW   4               ; Move n into WREG
                LFSR    1, 0x40         ; Load 0x40 into FSR0 to be used as pointer
                CALL    FIBNUM
FINISH          BRA     FINISH
; READ  THE FIBONACCI  NUMBER FOR n FROM  DATA MEMORY INTO 'W'
; USING MOVF WITH  INDEXED  ADDRESSING MODE
; SUBROUTINE
                ORG     0x200
      FIBNUM    MOVF    PLUSW1, W       ; Result in WREG
                RETURN                  ; Return to FINISH in main
                ORG     0x150
      ADDR      DB      1, 1, 2, 3, 5, 8, D'13'  ; Fibonacci numbers
                END
```

Example 7.8 Without using a lookup table, and the MOVF instruction with indexed addressing mode in the subroutine as in Example 7.7, write a subroutine in PIC18F assembly language at address 0x150 to find the n^{th} number (0 to 6) of the Fibonacci sequence. The subroutine will return the desired Fibonacci number in WREG based on 'n' stored by the main program. Also, write the main program at address 0x100 which will store the n^{th} number (0 to 6) in WREG, call the subroutine, and stop. The Fibonacci sequence for n = 0 to 6 is provided below:

n	Fib(n)
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Solution

This program can be written using the RETLW instruction which is ideal for returning the desired value using an operation alternate to using a table lookup with indexed addressing mode shown in Example 7.6. Note that the RETLW k loads the 8-bit immediate data k into WREG, and returns to the main program by loading the Program Counter with the address from the top of the hardware stack. The assembly language program is provided below:

```
                INCLUDE <PIC18F4321.INC>
                ORG     0x100           ; Main program
                MOVLW   0x10            ; Initialize STKPTR with  0x10
                MOVWF   STKPTR
                MOVLW   6               ; Load n into WREG
                CALL    FIBNUM          ; Call subroutine FIBNUM to find Fibonacci #
HERE            BRA     HERE            ; Halt
                ORG     0x150           ; Subroutine
FIBNUM          MULLW   2               ; PRODH:PRODL  2 x n, offset of RETLW table.
                                        ; 'n' is  multiplied by 2 since the  instruction size is
                                        ; word
```

```
MOVF    PRODL, W  ; Save low order 8 bits of the product in WREG
ADDWF   PCL       ; PCL = PCL + 2 x n
                  ; Fibonacci number table follows
RETLW   0x01
RETLW   0x01
RETLW   0x02
RETLW   0x03
RETLW   0x05
RETLW   0x08
RETLW   0x0D      ; 13 in decimal
END               ; WREG is loaded with the 8-bit literal k, and PC is
                  ; loaded with the return address from the top of the
                  ; stack
```

Example 7.9 Write a PIC18F assembly language program at address 0x200 to add two 16-bit numbers (N1 and N2), each containing two ASCII digits. The first 16-bit number (N1) is stored in two consecutive locations (from LOW to HIGH) in data memory with the low byte addressed by data reg 0x40, and the high byte addressed by data reg 0x41. The second 16-bit number (N2) is also stored in two consecutive locations (from LOW to HIGH) in data memory with the low byte addressed by data reg 0x50, and the high byte addressed by data reg 0x51. Store the packed BCD result in WREG.

Solution

Note that ASCII codes for decimal numbers 0 through 9 are 30H through 39H (See chapter 1).

Numerical Example: Assume [N1] = 3439H and [N2] = 3231H. The procedure for adding the two 16-bit ASCII numbers (N1 and N2) will be as follows:

1. Convert N1 and N2 to unpacked BCD numbers by masking off the high four bits using ANDWF instruction. This means that N1 = 0409H and N2 = 0201H.
2. Logically shift the high byte of N1 four times to the left so that the high byte will be converted from 04H to 40H. This is equivalent to swapping the low four bits (nibble) with the high four bits (nibble) using the SWAPF instruction. Logically OR this with the low byte of N1. Hence, N1 will be converted from unpacked BCD (0409H) to packed BCD 49H. Similarly, convert N2 from unpacked BCD (0201H) to packed BCD (21H).
3. Add (binary addition) the two packed BCD numbers (49H, 12H) using ADDWF instruction to obtain the following result:

```
First packed BCD byte     =  49H =  0100 1001
Second packed BCD byte = 21H =  0010 0001
                                 ----------------
Result after binary addition      0110  1010 (6AH)

4. Perform BCD correction on the binary result  0110  1010
Add 6 using DAW instruction                     0110
                                               ----------------
                                                0111 0000 = 70H
                                                      (Correct packed BCD result)
```

The PIC18F assembly language is provided below:

```
            INCLUDE  <P18F4321.INC>
            ORG      0x200
COUNTER     EQU      0x45
            MOVLW    0x39          ; #1 LOAD LOW BYTE OF N1 INTO 0x40
            MOVWF    0x40
            MOVLW    0x34          ; #2 LOAD HIGH BYTE OF N1 INTO 0x41
            MOVWF    0x41
            MOVLW    0x31          ; #3 LOAD LOW BYTE OF N2 INTO 0x50
            MOVWF    0x50
            MOVLW    0x32          ; #4 LOAD HIGH BYTE OF  N2 INTO 0x51
            MOVWF    0x51
            MOVLW    2             ; #5 INITIALIZE  COUNTER
            MOVWF    COUNTER
            LFSR     0, 0x40       ; #6  INITIALIZE FSR0 TO 0x40
            LFSR     1, 0x50       ; #7 INITIALIZE FSR1 TO 0x50
; STEP1: CONVERT N1 AND N2 TO UNPACKED BCD.
START       MOVLW    0x0F
            ANDWF    POSTINC0, F   ; #8 CONVERT  N1 TO UNPACKED BCD
            ANDWF    POSTINC1, F   ; #9 CONVERT  N2 TO UNPACKED BCD
            DECF     COUNTER, F    ; #10 DECREMENT  COUNTER BY 1
            BNZ      START         ; #11 BRANCH  IF NOT  ZERO
; UNPACKED BCD RESULT 0x41 (N1 HIGH BYTE), 0x40 (N1 LOW BYTE), 0x51
; (N2 HIGH BYTE), 0x50 (N2 LOW  BYTE)
; STEP2: CONVERT N1 AND N2 FROM  UNPACKED TO PACKED BCD
            SWAPF    0x41, W       ; #12 SWAP LOW NIBBLE OF 0x41 WITH
                                   ; HIGH NIBBLE AND STORE  IN WREG
            IORWF    0x40, F       ; #13 OR [W] WITH [0x40], PACKED BCD N1
            SWAPF    0x51, W       ; #14 SWAP LOW NIBBLE OF 0x51 WITH
                                   ; HIGH NIBBLE AND STORE IN WREG
            IORWF    0x50, W       ; #15 OR [WREG] WITH [0x50], PACKED BCD
                                   ; N2 IN WREG
; STEP3: PERFORM BINARY ADDITION BETWEEN N1 (PACKED BCD) WITH
; N2 (PACKED BCD) AND STORE RESULT IN WREG
            ADDWF    0x40, W       ; #16  BINARY RESULT IN WREG
; STEP4: ADJUST (BCD CORRECTION) [WREG] TO CONTAIN CORRECT PACKED BCD
            DAW                    ; #17 ADJUST THE RESULT TO CONTAIN
                                   ; CORRECT PACKED BCD
FINISH      BRA      FINISH        ; HALT
            END
```

Note: The above program will be explained in the following. Note that the # sign along with the line number is placed before each comment in order to explain the program. ASCII data to be added are assumed to be 3439H and 3231H. The purpose of the program is to convert the first number, ASCII 3439H, to unpacked BCD 0409H, and then to packed BCD 49H. Similarly, the second number, ASCII 3231H, to unpacked BCD 0201H, and then to packed BCD 21H. Finally, the two packed BCD numbers will be added in binary using PIC18F's ADDWF instruction, and then the result in WREG will be converted to

correct packed BCD using DAW.

Line #'s 1 through 4 initialize N1 and N2 so that [0x40] = 39H, [0x41] = 34H, [0x50] = 31H, [0x51] = 32H. Line #5 initializes COUNTER with loop count of 2 for converting the numbers from ASCII to unpacked BCD. Line #'s 6 and 7 initialize FSR0 and FSR1 with 0x40 and 0x50 respectively. Line #'s 8 through 11 convert the two bytes of ASCII codes in 0x41 (high byte) and 0x40 (low byte) into unpacked BCD in 0x41 (high byte) and 0x40 (low byte). Also, Line #'s 8 through 11 convert the ASCII numbers, N1 and N2, into their corresponding unpacked BCD bytes.

Line #'s 12 through 15 convert the unpacked BCD numbers (N1 and N2) into packed BCD bytes. This is done by swapping high unpacked bytes of N1 and N2 , and then ORing with the corresponding low unpacked bytes. Line #16 performs binary addition of the two packed BCD bytes (N1 and N2), and stores the binary result in WREG. The DAW instruction at Line #17 adjusts the contents of WREG to provide the correct packed BCD result.

7.9 PIC18F Delay Routine

Typical PIC18F software delay routines can be written by loading a "counter" with a value equivalent to the desired delay time, and then decrementing the "counter" in a loop using typically MOVE, DECREMENT, and conditional BRANCH instructions. For example, the following PIC18F instruction sequence can be used for a delay loop:

```
        MOVLW   COUNT
        MOVWF   0x20
DELAY   DECF    0X20, F
        BNZ     DELAY
```

Note that DECF in the above decrements the register 0x20 by one, and if Z =0, the program branches to DELAY and if Z = 1, the PIC18F executes the next instruction. The initial loop counter value of "COUNT" can be calculated using the machine cycles (Appendix D) required to execute the following PIC18F instructions:

```
MOVLW   (1 cycle)
MOVWF   (1 cycle)
DECF    (1 cycle)
BNZ     (2/1 cycles)
```

Note that the BNZ instruction requires two different execution times. BNZ requires 2 cycles when the PIC18F branches if Z = 0. However, the PIC18F goes to the next instruction and does not branch when Z = 1. This means that the DELAY loop will require 2 cycles for "(COUNT-1)" times, and the last iteration will take 1 cycle. The desired delay time can be obtained by loading register 0x20 with the appropriate COUNT value.

Assuming a 1-MHz default crystal frequency, the PIC18F's clock period will be 1μs. Note that the PIC18F divides the crystal frequency by 4. This is equivalent to multiplying the clock period by 4. Hence, each instruction cycle will be 4 microseconds. For a 100 microsecond delay, total cycles = $\frac{100\ micro\,\text{sec}}{4\ micro\,\text{sec}} = 25$. The BNZ in the loop will require 2 cycles for (COUNT - 1) times when Z = 0 and the last iteration will take 1 cycle

when no branch is taken (Z = 1). Thus, the total cycles including the MOVLW = Cycles for (MOVLW + MOVWF + (BNZ for Z = 0 + DCF) x (COUNT - 1)) + BNZ (Z = 1) = 1 + 1 + 3 × (COUNT - 1) + 1 = 25. Hence, COUNT = 8.3. Therefore, register 0x20 should be loaded with an integer value of 8 for an approximate delay of 100 microseconds.

Now, in order to obtain delay of one millisecond, the above DELAY loop of 100 microseconds can be used with an external counter. Counter value = $\frac{1\ milli\,\text{sec}}{100\ micro\,\text{secs}} = 10$. The following instruction sequence will provide an approximate delay of one millisecond:

```
        MOVLW   D'10'
        MOVWF   0x30      ; Initialize counter 0x30 for one millisecond delay
BACK    MOVLW   8
        MOVWF   0x20      ; Initialize counter 0x20 for 100 microsecond delay
DELAY   DECF    0X20, F ; 100 microsec delay
        BNZ     DELAY
        DECF    0X30, F
        BNZ     BACK
```

The above instruction sequence will provide an approximate delay of one millisecond. Note that execution times of certain instructions such as "MOVLW D'10'", "MOVWF 0x30", "DECF 0x30, F", and "BNZ BACK" are discarded since their execution times are negligible compared to one millisecond.

Example 7. 10 Assume 1 MHz PIC18F. Consider the following subroutine:

```
DELAY   MOVLW   D'100'
        MOVWF   0x20
LOOP    DECFSZ  0x20, F
        BRA     LOOP
        RETURN
```

(a) Calculate the time delay provided by the above subroutine.

(b) Calculate the counter value to be loaded into data register 0x20 for 1 ms delay.

Solution

(a) Each instruction in the above subroutine is executed in one cycle except for the DECFSZ instruction. DECFSZ is executed in one cycle if it does not skip, and two cycles if it skips (See Appendix D for instruction cycles).

Hence, total instruction cycles = Cycle for MOVLW + Cycle for MOVWF + (100-1) (Cycles for DECFSZ if it does not skip and BRA instructions) + (Cycle for DECFSZ if it skips) + Cycle for RETURN
= 1 + 1 + 99 (1 + 1) + 2 + 1
= 203

Since for the PIC18F, one instruction cycle = 4 clock cycles, total delay = (203) x 4 = 812 clock cycles. Also, for 1 MHz clock, each clock cycle is 1 μsec. Hence, total time delay = 812 μsec.

(b) Let n be the counter value. Hence, $(2 + 2 \times (n-1) + 3) \times 4 = 1000\ \mu$sec.
Note that 1 ms = 1000 μsec. Therefore, n = 123.5. Therefore, data register 0x20 should be loaded with 124 for an approximate delay of 1 ms.

QUESTIONS AND PROBLEMS

7.1 Write a PIC18F assembly language program at address 0x150 to subtract two 16-bit numbers as follows: [0x21][0x20] - [0x31][0x30]→[0x40][0x41], if [0x50] is odd. If [0x50] is even store 0's in [0x41] [0x40].

7.2 Write a PIC18F assembly language program at address 0x100 to add a 4-bit unsigned number stored in the high nibble of data register 0x30 with a 4-bit unsigned number stored in the low nibble of data register 0x30. Store the 8-bit result in 0x30.

7.3 Write a PIC18F assembly program at address 0x200 to multiply a 4-bit unsigned number in the low nibble of 0x30 by another 4-bit unsigned number in the high nibble of 0x30. Store the result in 0x31.

7.4 Write a PIC18F assembly language program at address 0x150 to convert temperature from Fahrenheit to Celsius using the equation: $C = [(F - 32)/9] \times 5$; assume that the temperature in Fahrenheit is 8-bit wide to be loaded into data register 0x20. Assume that the temperature is always positive. Store the 8-bit result in data register 0x21.

7.5 Write a PIC18F assembly language program at address 0x100 to find $X^2/128_{10}$ where X is an 8-bit unsigned number stored in data register 0x40. The program will then compute $X^2/128$ and then store the 8-bit quotient in data register 0x50. Discard the remainder.

7.6 Write a PIC18F assembly language program at address 0x200 to perform ($X^2 + Y^2$) where X is a signed 8-bit number and Y is an unsigned 8-bit number. Write subroutine for multiplication in PIC18F assembly as needed. Also, write the main program at address 0x100 in PIC18F assembly language that will initialize FSR0 to 0x0070, X and Y to arbitrary data, call the subroutines to compute ($X^2 + Y^2$), and then push the 8-bit result onto the software stack pointed to by FSR0.

7.7 Write a PIC18F assembly program at address 0x100 which is equivalent to the following C language segment:

```
sum = 0;
for (i = 0; i <= 9; i = i + 1)
sum = sum + x[i] * y[i];
```

Assume that the arrays, growing from LOW to HIGH addresses and, x[i] and y[i], contain unsigned 8-bit numbers already stored in memory starting at data memory addresses 0x20 and 0x30 respectively. Store the 8-bit result at address 0x50.

7.8 Write a PIC18F assembly program at address 0x150 to compare two strings of 10 ASCII characters. The first string is stored starting at 0x30. The second string is stored at location 0x50. The ASCII character in location 0x30 of string 1 will be compared with the ASCII character in location 0x50 of string 2, [0x31] will be compared with [0x51], and so on. Each time there is a match, store 0xEE onto the software stack; otherwise, store 0x00 onto the software stack. Initialize software stack pointer at 0x60.

7.9. Write a PIC18F assembly program at address 0x100 to divide a 9-bit unsigned number in the high 9 bits (bits 8-1 in bits 7-0 of register 0x30 and bit 0 in bit 7 of register 0x31) by 8_{10}. Store the result in registers 0x30 (Most significant bit of the result in bit 0 of 0x30) and 0x31 (Low 8 bits of the result). Discard the remainder.

7.10 Write a PIC18F assembly language program at address 0x200 that will check whether the 16-bit signed number in registers [0x31][0x30] is positive or negative. If the number is positive, the program will multiply the 16-bit unsigned number in [0x21][0x20] by 16, and provide a 16-bit result; otherwise, the program will set the byte in register 0x40 to all ones. Use only data movement, shift, bit manipulation, and program control instructions. Assume the 16-bit signed and unsigned numbers are already loaded into the data registers.

7.11 Assume that 30 8-bit packed BCD numbers are stored in data memory locations from 0x10 through 0x2D. Write a PIC18F assembly language program at address 0x100 to find how many of these numbers are divisible by 5, and save the result in data memory location 0x40.

7.12 Write a program at address 0x100 in PIC18F assembly language to add two 32-bit packed BCD numbers. BCD number 1 is stored in data registers starting from 0x20 through 0x23, with the least significant digit at register 0x23 and the most significant digit at 0x20. BCD number 2 is stored in data registers starting from 0x30 through 0x33, with the least significant digit at 0x33 and the most significant digit at 0x30. Store the result as packed BCD digits in 0x20 through 0x23.

7.13 Write a subroutine at address 0x100 in PIC18F assembly language program to find the square of a BCD digit (0 to 9) using a lookup table. The subroutine will obtain the desired result based on the BCD digit stored by the main program. The lookup table will store the square of the BCD numbers starting at program memory address 0x300. Also, write the main program at address 0x50 which will initialize STKPTR to 0x10, store the BCD digit (0 to 9) in WREG, call the subroutine, and stop. Use indexed addressing mode.

7.14 Write a subroutine in PIC18F assembly language program at address 0x100 to find the square of a BCD digit (0 to 9). The subroutine will return the desired result based on the BCD digit stored by the main program. Also, write the main program at address 0x20 which will initialize STKPTR to 0x12, store the BCD digit (0 to 9) in WREG, call the subroutine, and stop. Do not use indexed addressing mode.

7.15 Write a subroutine at address 0x150 in PIC18F assembly language to convert a 3-digit unpacked BCD number to binary using unsigned multiplication by 10, and addition. The most significant digit is stored in a memory location starting at register 0x30, the next digit is stored at 0x31, and so on. Store the 8-bit binary result *(N)* in register 0x50. Note that arithmetic operations for obtaining N will provide binary result. Use the value of the 3-digit BCD number,

$$\begin{aligned} N &= N2 \times 10^2 + N1 \times 10^1 + N0 \\ &= ((10 \times N2) + N1) \times 10 + N0 \end{aligned}$$

7.16 Write a subroutine in PIC18F assembly language at address 0x200 to compute

$$Z = \sum_{i=1}^{10} X_i$$

Assume the X_i's are unsigned 8-bit and stored in consecutive locations starting at 0x50. Assume FSR1 points to the X_i's. Also, write the main program in PIC18F assembly language at address 0x100 to perform all initializations (FSR1 to 0x30, STKPTR to 20 decimal), call the subroutine, and then compute *Z*/8 by discarding the remainder.

7.17 Write a PIC18F assembly language program to estimate the square root of an 8-bit integer number, P, using the algorithm provided in the following:

The sum of odd integers is always a perfect square. For example, $1 = 1^2$, $1+3 = 2^2$, $1+3+5 = 3^2$, and so on. Specifically, $\sum_{i=1}^{k} (2i-1) = k^2$. This property is useful in approximating the square root of an 8-bit unsigned number, P. For example, if P = 17, the square root of P can be estimated as follows:

Subtract 1 from P so that P becomes 16; since the subtraction went through, add 1 to a counter. Hence, the counter value is 1.

Subtract 3 from P so that P becomes 13; since the subtraction went through, add 1 to a counter. Hence, the counter value is 2.

Subtract 5 from P so that P becomes 8; since the subtraction went through, add 1 to a counter. Hence, the counter value is 3.

Subtract 7 from P so that P becomes 1; since the subtraction went through, add 1 to a counter. Hence, the counter value is 4. When 9 is subtracted from the existing value of P (P = 1), the result becomes negative (-8) meaning that the subtraction did not go through. The process terminates, and the integer square root approximation for 17 is 4.

7.18 Consider the following PIC18F DELAY subroutine:

```
DELAY   MOVLW   Q
        MOVWF   Q
LOOP1   MOVLW   100
        MOVWF   P
LOOP2   DECF    P, F
        BNZ     LOOP2
        DECF    Q, F
        BNZ     LOOP1
        RETURN
```

Assuming 1MHz PIC18F, determine the value of Q such that when this subroutine is called, a delay of 145.940 ms will be generated.

7.19 Consider the following loop statement in C language ;

```
for  (p = 80; p > 0; p--);    // a dummy loop with no statement
```

The above C loop can be represented using PIC18F assembly language in two ways:

Method 1: Using conventional way

```
        MOVWF   0x50
        MOVWF   P
LOOP    DECF    P, F
        BNZ     LOOP
```

Method 2: Using SKIP instruction

```
        MOVWF   0x50
        MOVWF   P
LOOP    DECFSZ  P, F
        BRA     LOOP
        DONE    -------
```

Assume 1 MHz PIC18F clock. Calculate the execution time of each delay loop (Method 1 and Method 2). Which delay loop will be executed faster?

7.20 Write a PIC18F assembly language instruction sequence that is equivalent to the following C code:

```
if (P <= q)
  p = p + 5;
else
  p = 10;
```

Assume p and q are unsigned numbers.

7.21 Repeat Problem 6.14 using a Loop.

7.22 Repeat Problem 6.16 using a Loop.

8

PIC18F PROGRAMMED I/O USING ASSEMBLY & C

In this chapter we describe PIC18F programmed I/O using assembly and C. Topics include PIC18F4321 pins and signals, clock and reset circuits, programmed I/O and Seven-segment displays.

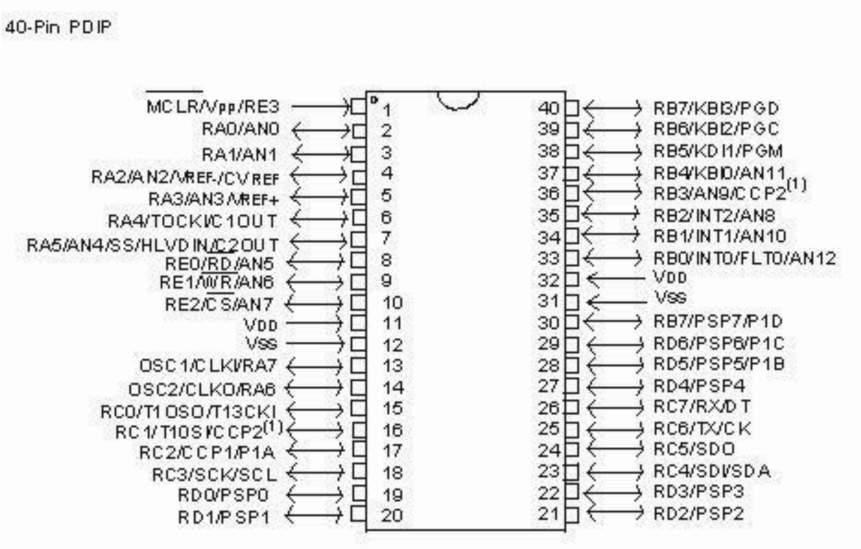
8.1 PIC18F Pins and Signals

The PIC18F4321 is contained in three types of packaging as follows:

- 40-pin Plastic Dual In-line Package (PDIP)
- 44-pin Quad Flat No-lead plastic package (QFN)
- 44-pin Thin plastic Quad Flat pack Package (TQFP)

Figure 8.1 shows the PIC18F4321 pin diagram for a PDIP. A brief description of all pins and signals for the PIC18F4321 contained in the 40-pin PDIP is provided in Table 8.1.

There are two VDD (Vcc) pins and two VSS (Ground) pins which are not shared (multiplexed) with other pins. The range of voltages for the VDD pins are from + 4.2V



Note 1: RB3 is the alternate pin for CCP2 multiplexing.

FIGURE 8.1 PIC18F4321 Pins and Signals

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

to +5.5V. However, the VDD pins are normally connected to +5V. The VSS pins are connected to ground. The maximum power dissipation for the PIC18F4321 is one watt. Note that multiple pins for power and ground are used in order to distribute the power to reduce noise problems at high frequencies.

All other 36 pins are multiplexed (shared) with other signals. There are 36 pins assigned to five I/O ports, namely Port A (8-bit, RA0-RA7), Port B (8-bit, RB0-RB7), Port C (8-bit, RC0-RC7), Port D (8-bit, RD0-RD7), and Port E (4-bit, RE0-RE3). These pins are multiplexed with other signals such as the clock/oscillator, reset, external interrupt, analog inputs, and CCP (Capture/Compare/Pulse Width Modulation). The PIC18F pins associated with clock, reset, and I/O will be discussed in the following.

8.1.1 Clock

The PIC18F4321 clock can be generated internally or externally. The internal clock can be obtained by an on-chip oscillator provided by Microchip. The external clock, on the other hand, can be generated by connecting external components such as a crystal and RC oscillator circuit at the OSC1 and OSC2 pins. When the internal clock is used, an internal oscillator generates the internal clock. This can be used as the microcontroller's clock source. This will eliminate the need for an external oscillator circuit on the OSC1

TABLE 8.1 PIC18F4321 Pinout description

Pin Number	Pin Name	Pin Type	Description
1	$\overline{\text{MCLR}}$	Input	Master Clear Reset ; active low input.
	Vpp	Input	Programming voltage input for the flash memory.
	RE3	Input	Digital input; port E bit 3.
2	RA0	Input/Output	Digital I/O; port A bit 0.
	AN0	Input	Analog input 0.
3	RA1	Input/Output	Digital I/O; port A bit 1.
	AN1	Input	Analog input 1.
4	RA2	Input/Output	Digital I/O; port A bit 2.
	AN2	Input	Analog input 2.
	VREF-	Input	A/D reference voltage (low) input
	CVREF	Output	Comparator reference voltage output
5	RA3	Input/Output	Digital I/O; port A bit 3.
	AN3	Input	Analog input 3.
	VREF+	Input	A/D reference voltage (high) input.
6	RA4	Input/Output	Digital I/O; port A bit 4.
	T0CKI	Input	Timer0 external clock input.
	C1OUT	Output	Comparator 1 output.
7	RA5	Input/Output	Digital I/O; port A bit 5.
	AN4	Input	Analog input 4.
	SS	Input	SPI slave select input
	HLVDIN	Input	High/Low-Voltage Detect input.
	C2OUT	Output	Comparator 2 output.
8	RE0	Input/Output	Digital I/O; port E bit 0.
	$\overline{\text{RD}}$	Input	Read control for Parallel Slave Port (see also $\overline{\text{WR}}$ and $\overline{\text{CS}}$ pins).
	AN5	Input	Analog input 5.
9	RE1	Input/Output	Digital I/O; port E bit 1.
	$\overline{\text{WR}}$	Input	Write control for Parallel Slave Port (see also $\overline{\text{RD}}$ and $\overline{\text{CS}}$ pins).
	AN6	Input	Analog input 6

TABLE 8.1 PIC18F4321 Pinout description (Continued)

Pin Number	Pin Name	Pin Type	Description
10	RE2	Input/Output	Digital I/O; port E bit 2.
	$\overline{CS}$	Input	Chip Select control for Parallel Slave Port (see related $\overline{RD}$ and $\overline{WR}$).
	AN7	Input	Analog input 7.
11	VDD	Power	Positive supply for logic and I/O pins.
12	VSS	Ground	Ground reference for logic and I/O pins.
13	OSC1	Input	Oscillator crystal input or external clock source input.
	CLKI	Input	External clock source input. Always associated with pin function OSC1.(See related OSC1/CLKI, OSC2/CLKO pins.)
	RA7	Input/Output	Digital I/O; port A bit 7.
14	OSC2	Output	Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. In RC, EC and INTIO modes, OSC2 pin outputs
	CLKO	Output	CLKO which has one-fourth the frequency of OSC1 and denotes the instruction cycle rate
	RA6	Input/Output	Digital I/O; port A bit 6.
15	RC0	Input/Output	Digital I/O; port C bit 0.
	T1OSO	Output	Timer1 oscillator analog output.
	T13CKI	Input	Timer1/Timer3 external clock input
16	RC1	Input/Output	Digital I/O; port C bit 1.
	T1OSI	Input	Timer1 oscillator analog input.
	CCP2	Input/Output	Capture 2 input/Compare 2 output/PWM 2 output; default assignment for CCP2 when Configuration bit, CCP2MX, is set.
17	RC2	Input/Output	Digital I/O; port C bit 2.
	CCP1	Input/Output	Capture 1 input/Compare 1 output/PWM 1 output.
	P1A	Output	Enhanced CCP1 output.
18	RC3	Input/Output	Digital I/O; port C bit 3.
	SCK	Input/Output	Synchronous serial clock input/output for SPI mode.
	SCL	Input/Output	Synchronous serial clock input/output for I^2C^{TM} mode
19	RD0	Input/Output	Digital I/O; port D bit 0.
	PSP0	Input/Output	Parallel Slave Port data.
20	RD1	Input/Output	Digital I/O; port D bit 1.
	PSP1	Input/Output	Parallel Slave Port data.
21	RD2	Input/Output	Digital I/O; port D bit2.
	PSP2	Input/Output	Parallel Slave Port data
22	RD3	Input/Output	Digital I/O; port D bit 3.
	PSP3	Input/Output	Parallel Slave Port data.

TABLE 8.1 PIC18F4321 Pinout description (Continued)

Pin Number	Pin Name	Pin Type	Description
23	RC4	Input/Output	Digital I/O.
	SDI	Input	SPI data in.
	SDA	Input/Output	IC data I/O.
24	RC5	Input/Output	Digital I/O; port C bit 5.
	SDO	Output	SPI data out.
25	RC6	Input/Output	Digital I/O; port C bit 6.
	TX	Output	EUSART asynchronous transmit.
	CK	Input/Output	EUSART synchronous clock (see related RX/DT).
26	RC7	Input/Output	Digital I/O; port C bit 7.
	RX	Input	EUSART asynchronous receive.
	DT	Input/Output	EUSART synchronous data (see related TX/CK).
27	RD4	Input/Output	Digital I/O; port D bit 4
	PSP4	Input/Output	Parallel Slave Port data
28	RD5	Input/Output	Digital I/O; port D bit 5.
	PSP5	Input/Output	Parallel Slave Port data.
	P1B	Output	Enhanced CCP1 output.
29	RD6	Input/Output	Digital I/O; port D bit 6.
	PSP6	Input/Output	Parallel Slave Port data.
	P1C	Output	Enhanced CCP1 output.
30	RD7	Input/Output	Digital I/O; port D bit 7.
	PSP7	Input/Output	Parallel Slave Port data.
	P1D	Output	Enhanced CCP1 output.
31	VSS	Ground	Ground reference for logic and I/O pins.
32	VDD	Power	Positive supply for logic and I/O pins.
33	RB0	Input/Output	Digital I/O; port b bit 0.
	INT0	Input	External interrupt 0.
	FLT0	Input	PWM Fault input for Enhanced CCP1.
	AN12	Input	Analog input 12.
34	RB1	Input/Output	Digital I/O; port B bit 1.
	INT1	Input	External interrupt 1.
	AN10	Input	Analog input 10.
35	RB2	Input/Output	Digital I/O; port B bit 2.
	INT2	Input	External interrupt 2.
	AN8	Input	Analog input 8.
36	RB3	Input/Output	Digital I/O; port B bit 3.
	AN9	Input	Analog input 9.
	CCP2	Input/Output	Capture 2 input/Compare 2 output/PWM 2 output; alternate assignment for CCP2 when Configuration bit CCP2MX is cleared.

TABLE 8.1 PIC18F4321 Pinout description (Continued)

Pin Number	Pin Name	Pin Type	Description
37	RB4	Input/Output	Digital I/O; port B bit 7.
	KBI0	Input	Interrupt-on-change pin.
	AN11	Input	Analog input 11.
38	RB5	Input/Output	Digital I/O; port B bit 5.
	KBI1	Input	Interrupt-on-change pin.
	PGM	Input/Output	Low-Voltage Programming enable pin.
39	RB6	Input/Output	Digital I/O; port B bit 6.
	KBI2	Input	Interrupt-on-change pin
	PGC	Input/Output	In-Circuit Debugger and programming clock pin.
40	RB7	Input/Output	Digital I/O; port B bit 7
	KBI3	Input	Interrupt-on-change pin.
	PGD	Input/Output	In-Circuit Debugger and ICSP programming data pin.

and/or OSC2 pins. Hence, it is convenient to use the internal oscillator.

The main disadvantage of the internal oscillator is the lack of precision and frequency stability. The baseline frequency depends on the values of the passive components used in building the internal oscillator circuit, and the tolerances of the values of the passive components are not particularly tight. Also, resistance and capacitance values are affected by ambient temperature. This means that changes in temperature lead to changes in frequency. Since the internal frequency is calibrated at the factory, many applications can tolerate the shortcomings of an internal oscillator. With older microcontrollers, the internal oscillator might have tolerance as bad as -20% to +20%. However, contemporary microcontrollers such as the PIC18F can provide tolerance of -1.5% to +1.5%. This is accurate enough for serial I/O including RS-232 and USB. Furthermore, the PIC18F uses a hardware circuit called the "Phase-Locked Loop" (PLL) to multiply the internal frequency to a higher frequency.

Upon power-on reset, the PIC18F4321 automatically operates at an internal clock frequency of 1-MHz (default). This means that with no crystal oscillator circuit connected to the OSC1 and OSC2 pins, the PIC18F4321 operates from an internal clock frequency of 1-MHz upon hardware reset.

Two internal oscillator configurations are available. They are INTIO1 mode and INTIO2 mode. These modes can be selected, for example, using the MPLAB C18 compiler's CONFIG directive (to be discussed later). In INTIO1 mode, the OSC2 pin outputs FOSC/4, while OSC1 functions as RA7 (bit 7 of Port A) for digital input and output. Note that OSC1 is multiplexed with RA7. On the other hand, in INTIO2 mode, OSC1 functions as RA7 and OSC2 functions as RA6 both for digital input and output. Note that OSC1 is multiplexed with RA7 while OSC2 is multiplexed with RA6.

There are three clock sources for the PIC18F4321. They are:

- Primary oscillators
- Secondary oscillators
- Internal oscillators

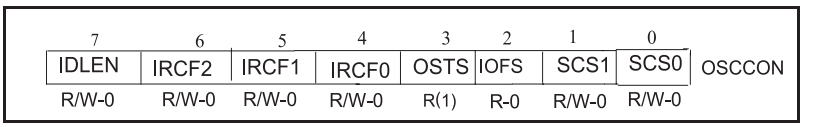
The primary oscillators require external circuits to be connected at the OSC1 and OSC2 pins. The external components such as a crystal and RC oscillator circuit can be connected to the OSC1 and OSC2 pins of the PIC18F4321. This will provide input to the

primary oscillators to generate the clock for the PIC18F4321 externally.

The secondary oscillators are those external sources not connected to the OSC1 or OSC2 pins. Instead, they are connected to the T1OSI (Timer1 Oscillator Input) and T1OSO (Timer1 Oscillator Output) pins of the PIC18F4321. The PIC18F4321 offers the Timer1 (one of the on-chip hardware timers) oscillator as a secondary oscillator.

The internal oscillators provide a clock to the PIC18F4321 without connecting any components such as crystal or RC circuits to its clock pins. The Primary, Secondary, or Internal oscillator can be selected by programming an internal register called the OSCCON (Oscillator Control) register (Figure 8.2).

The OSCCON (Oscillator Control) register controls several aspects of the device clock's operation, both in full power operation and in power-managed modes. Note that PIC18F power- managed mode consumes low power (nanowatts), and are suitable for low power applications including smoke detectors, hospital ID tags, and automobile ignition systems.



bit 7 **IDLEN:** Idle Enable bit
1 = Device enters an Idle mode when a SLEEP instruction is executed
0 = Device enters SLEEP mode when a SLEEP instruction is executed
bit 6-4 **IRCF2:IRCF0:** Internal Oscillator Frequency Select bits**(5)**
111 = 8 MHz (INTOSC drives clock directly)
110 = 4 MHz
101 = 2 MHz
100 = 1 MHz**(3)**
011 = 500 kHz
010 = 250 kHz
001 = 125 kHz
000 = 31 kHz (from either INTOSC/256 or INTRC directly)**(2)**
bit 3 **OSTS:** Oscillator Start-up Time-out Status bit**(1)**
1 = Oscillator Start-up Timer (OST) time-out has expired; primary oscillator is running
0 = Oscillator Start-up Timer (OST) 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**(4)**
1x = Internal oscillator block
01 = Secondary (Timer1) oscillator
00 = Primary oscillator
Note 1: Reset state depends on state of the IESO Configuration bit.
2: Source selected by the INTSRC bit (OSCTUNE<7>), see text.
3: Default output frequency of INTOSC on Reset.
4: Modifying the SCSI:SCSO bits will cause an immediate clock source switch.
5: Modifying the IRCF3:IRCF0 bits will cause an immediate clock frequency switch if the internal oscillator is providing the device clocks.
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

FIGURE 8.2 OSCCON (Oscillator Control) Register

The IDLEN bit controls whether the PIC18F4321 goes into Sleep mode or one of the Idle modes when the SLEEP instruction is executed. The IDLEN bit affects how the oscillator behaves in power managed modes.

Once the internal oscillator is selected (using "config OSC = INTIO2" in assembly or " #pragma config OSC = INTIO2" in C), the Internal Oscillator Frequency Select bits (IRCF2:IRCF0) in the OSCCON register select the frequency output of the internal oscillator to drive the PIC18F4321 clock. The three bits select one of eight different clock frequencies. If the internal oscillator is supplying the PIC18F4321 clock, changing the states of these bits will have an immediate change on the internal oscillator's output.

The internal oscillator start-up timer is a delay built into the PIC18F4321 to allow the external oscillator to stabilize. However, the internal oscillator will run the PIC18F4321 if it is selected as the main oscillator. The OSTS and IOFS are read-only and are set or reset by the device. The PIC18F4321 has the option to startup running off the internal oscillator until an external oscillator circuit is stabilized. This allows for a faster startup of the microcontroller with external oscillators. The OSTS bit is used to alert the software when the clock source has switched to the external primary oscillator.

The System Clock Select bits, SCS1:SCS0, select the clock source. The available clock sources are the primary clock, the secondary clock (Timer1 oscillator), and the internal oscillator. For example, SCS1SCS0 = 1x selects the internal clock.

As mentioned before, after power-on reset, the PIC18F4321 operates from an on-chip clock frequency of 1-MHz provided by Microchip, and no external crystal or clock circuit is required. This is called the default mode. In the default mode (after reset), the contents of the OSCCON register are 0x40 as follows:
IDIEN = 0, IRCF2 - IRCF0 = 100 = 1MHz, OSTS = 0, IOFS = 0, SCS1:SCS0 = 00.

However, the internal clock frequency can be changed within the range between 31 kHz and 8 MHz by writing appropriate data into the OSCCON register. For simplicity, the internal oscillator will be used in this book. Also, in order to change the frequency after reset, bits IRCF2-IRF0 in the OSCCON register will be changed while other bits will be loaded with 0's. For example, in order to change the internal frequency from 1-MHz (default mode) to 4-MHz, bits IRCF2 - IRCF0 in the OSCCON register must be 110. Hence, OSCCON register should be loaded with 0x60.

This can be accomplished by using the following PIC18F assembly code:

```
MOVLW 0x60; Set the internal clock to 4 MHz
MOVWF OSCCON
```

This is equivalent to the following C-code

```
OSCCON = 0x60;          // set the internal crystal to 4 MHz
```

Note that the value of SCS1:SCS0 = 10 or 11 (selecting the internal oscillator in the OSCCON register of Figure 8.2). This is because 'x' in SCS1:SCS0 = 1x (Figure 8.2) is don't care, and can be 0 or 1. Loading OSCCON register with 0x62 or 0x63 will also select the internal oscillator with 4 MHz. The statement config OSC = INTIO2 (for MPLAB PIC18F assembler) and the statement

```
# pragma config OSC = INTIO2
```

(for C18 compiler) will select the internal oscillator in the INTIO2 mode. These statements are sufficient enough to get the processor running using the internal oscillator regardless of the settings of the SCS1:SCS0 bits in the OSCCON register. The directives "config" and "#pragma config" will be covered in details later in this chapter.

The PIC18F4321 can be operated in several different oscillator configurations. The user can program the Configuration bits FOSC3:FOSC0 in the CONFIG1H (Configuration

Register) to select one of these configurations. Two of these configurations for the primary oscillator are provided below:

1. By connecting a crystal or ceramic oscillator at OSC1 and OSC2 pins, there are several choices for the crystal oscillator. One may consult the "PIC18F data sheet" for details.

If a crystal oscillator is used, a parallel resonant crystal rather than a series resonant crystal must be used. This is because series resonant crystals do not oscillate upon power-up. Figure 8.3 shows the connection of a crystal oscillator (parallel resonant) to the OSC1 and OSC2 pins of the PIC18F4321. The crystal frequency can vary from 4MHz to 25 MHz. The values of the capacitors can be between 15pF and 33pF. However, Microchip recommends 22pF for reliable operation with crystal frequencies.

2. By connecting an RC circuit at the OSC1 pin. The oscillator frequency divided by 4 is output at the OSC2 pin. Figure 8.4 illustrates this. An RC circuit connected to the OSC1 pin is used as the clock source. The RC-based clock is not that accurate compared to the crystal oscillator.

Using this configuration, the clock input at the OSC1 pin can be used as the CPU clock while the clock output (Fosc/4) at the OSC2 pin can be used for devices requiring Fosc/4 as the input clock. Since the OSC2 pin is multiplexed with PA6 (bit 6 of Port A), the OSC2 pin can also be used as an I/O pin.

In summary, to simplify using the PIC18F clock for performing meaningful experiments in a first course on "Micrcontrollers", the following suggestions should be followed:

1. Use the internal clock with a default value of 1-MHz.

2. Select the clock from the source INTIO1 or INTIO2 using the "config" directive for the MPLAB PIC18F assembler or the "#pragma config" directive for the C18 compiler.

3. If changing the default frequency is desired, then use bits 6-4 (IRCF2-IRCF0) in the OSCCON register to select the clock frequency within the allowable range. The other bits in the OSCCON register in the default mode are basically irrelevant. This makes changing the internal frequency using the OSCCON register in the default mode very easy and convenient.

8.1.2 PIC18F Reset

Upon initiating a reset, the PIC18F loads '0' into the program counter. Thus, the PIC18F reads the first instruction from the contents of address 0 in the program memory. Most registers are unaffected by a reset. However, WREG and STKPTR are cleared to zero. All TRISx (Data Direction Registers) registers are loaded with ones; that is, all I/O ports are configured as inputs.

The PIC18F4321 can be reset in several different ways. For simplicity, the two most commonly used RESET techniques are Power-on and Manual resets. These two resets will be discussed first. A summary of some of the other resets will then be provided.

Power-On Reset (POR) Upon power-up, a power-on reset pulse is generated internally whenever VDD rises above a certain threshold. This allows the PIC18F chip to start in the initialized state when VDD is adequate for operation. The reset circuit in Figure 8.5 provides a simple manual reset circuit with a pushbutton (manual) switch. When the VDD is connected to power (for example, +5VDC), the resistors in Figure 8.5 with the switch open will provide power-on reset. When the PIC18F exits the power-on reset condition

and starts normal operation, a program can be executed by pressing the pushbutton, and program execution can be restarted upon activation of the pushbutton.

Power-on Reset events are captured by the bit (bit 1 of RCON, Figure 8.6). The state of the bit is cleared to '0' whenever a POR occurs; bit = 1 indicates that a POR has not occurred.

Manual Reset Note that Figure 8.5 shows a simplified circuit for manual reset. The $\overline{\text{MCLR}}$ pin is normally HIGH. Upon activating the push button, the $\overline{\text{MCLR}}$ pin is driven from HIGH to LOW. The internal on-chip circuitry connected to the $\overline{\text{MCLR}}$ pin ensures

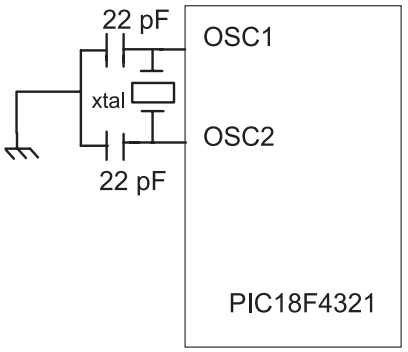


FIGURE 8.3 Typical Crystal Oscillator Circuit

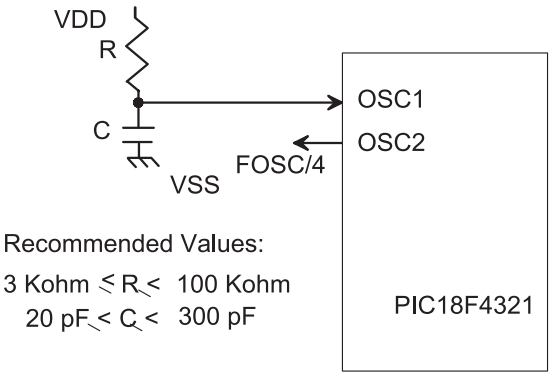


FIGURE 8.4 RC Oscillator

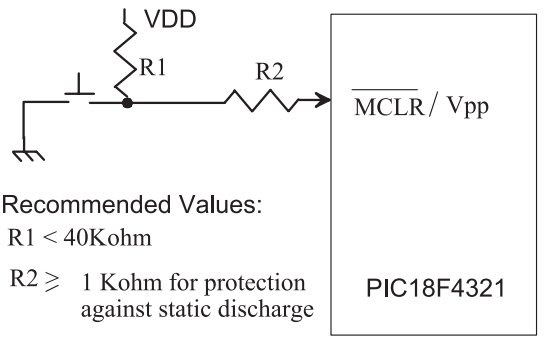


FIGURE 8.5 PIC18F Manual Reset Circuit

that the pin is LOW for at least 2 μsec (minimum requirement for reset). The PIC18F can be reset manually by the circuit of Figure 8.5. Note that this circuit has been implemented in the laboratory and the minimum timing requirement has been verified.

Other Resets Some of the other resets include:
Brown-out Reset (BOR)
Watchdog Timer (WDT) reset (during program execution)
RESET Instruction.

Note that other reset events are tracked through the RCON (Reset Control) register (Figure 8.6). The lower five bits of the register indicate that a specific reset event has occurred. In most cases, these bits can only be cleared by the event and must be set by the application after the event. The state of these flag bits, taken together, can be read to indicate the type of reset that just occurred. The RCON register also has control bits for setting interrupt priority (IPEN) to be discussed later, and software control of the BOR (Brown Out Reset).

Brown-out Reset. The PIC18F implements a BOR circuit that resets the PIC18F when the power supply voltage drops below a preprogrammed level (typically the level is 4.2V for a 5V supply). The C18 directive "#pragma config" can be used to set the BOR voltage level to 4.2V in C-language as follows:

```
#pragma config   BORV = 42      // Set BOR voltage to 4.2V
```

The PIC18F will not function properly if the power drops below the preprogrammed voltage. The idea behind BOR is that the processor will hold in reset when BOR occurs rather than trying to keep running unreliably. Thus, the PIC18F will resume program execution properly when the power goes above brownout threshold level. The BOR resets the PIC18F and then clears the bit in the RCON register. The BOR is useful when battery is used for the supply voltage.

In this book, for all I/O examples, a power supply rather than a battery will be used to provide the supply voltage and the BOR will be turned OFF. The C18 directive "#pragma config" can be used to accomplish this as follows:

```
#pragma config BOR = OFF        //  Turn OFF Brown Out Reset
```

Watchdog Timer (WDT) The PIC18F WDT is driven by an internal clock source. When the WDT is enabled via software using WDTCON (Watchdog Timer Control

7	6	5	4	3	2	1	0
IPEN	SBOREN	----------	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$

Bit 7 = IPEN (Interrupt Pending, 1 = enable, 0 = disable; to be discussed later) **Bit 6** = SBOREN (BOR software enable bit, 1 = enable, 0 = disable) **Bit 5** = Unimplemented (read as '0') **Bit 4** = $\overline{RI}$ (RESET instruction bit, 1 = RESET instruction executed, 0 = RESET instruction not executed) **Bit 3** = $\overline{TO}$ (Watchdog Timeout bit, 1 = upon power-up or execution of CLRWDT or SLEEP instruction, 0 = a watchdog timer timeout occurred) **Bit 2** = $\overline{PD}$ (Power- Down detection bit, 1 = upon power-up or execution of CLRWDT instruction, 0 = execution of SLEEP instruction. **Bit 1** = A $\overline{POR}$ (Power-on Reset status bit, 1 = A Power-on reset has not occurred, 0 = A Power-on reset has occurred) **Bit 0** = $\overline{BOR}$ (Brown-out Reset status bit, 1 = A Brown-out reset has not occurred, 0 = A Brown-out reset has occurred)

FIGURE 8. 6 RCON (RESET CONTROL) Register

register), the internal clock source is also enabled. The time delay provided by the WDT varies from 4 ms to 131.072 seconds. If the timer associated with the WDT is enabled and then times out after a specific amount of time during program execution, the PIC18F resets itself automatically and clears the bit in the RCON register. This "time out" may happen in certain situations such as if the program is "caught in an infinite loop" or if the program takes unexpectedly longer execution time. The WDT can be useful for debugging programs.

In this book, for all I/O examples, the WDT will be turned OFF. In C18 directive, "#pragma config" can be used to accomplish this as follows:

```
#pragma config  WDT = OFF        // Turn WDT OFF
```

RESET Instruction This is a software RESET. Upon execution of the RESET instruction, the PIC18F resets all registers and flags that are affected by a power-on Reset, and the bit in the RCON register is cleared to 0 indicating the reset instruction is not executed.

8.1.3 High Voltage and Low Voltage Programming (HVP and LVP)

There are two ways to program the PIC18F microcontroller. They are:

HVP (High Voltage Programming) This a is normal programming method. The downloading hardware interface such as the PICKit3 can be used for this purpose. Programming voltage of usually up to 13V is applied at the VPP pin during downloading. The assembled or compiled programs from a computer (Personal Computer or Laptop) can be downloaded into the PIC18F chip by connecting the PICKit3 between them**.**

LVP (Low Voltage Programming) LVP is basically the same as the HVP except the programming voltage is set to the normal supply voltage of 5V. The real benefit of using the LVP mode is that one can program several PIC18F on a board without having to program each one individually. One could daisy chain each extra PIC18F to a master PIC18F which would then program each one in turn. This is only possible since the Vpp signal is a normal logic level in LVP mode. Hence, this mode is a specialized operation and is not normal for standard use. Also, in the LVP mode, the PGM pin cannot be used for anything else such as RB5 since the PGM and RB5 are multiplexed. Hence, it is a good idea to disable the LVP mode by using the C18 directive "#pragma config" as follows:

```
#pragma config LVP = OFF        // Turn LVP OFF
```

8.1.4 "pragma config" and "config" directives

The C18 directive "#pragma config" is discussed in the previous sections. The equivalent directive in the PIC18F MPLAB assembler is the "config" directive. This directive tells the MPLAB PIC18F assembler that the assembler needs to do something specific to the hardware. Some of these functions include selecting an appropriate oscillator and turning ON or OFF certain features such as WDT (Watch Dog Timer), BOR (Brown Out Reset), and LVP (Low Voltage Programming). Since an internal oscillator will be used in all the I/O examples in this book, the following "config" commands are necessary for PIC18F assembly language I/O programs for the PIC18F chips to function properly**.** After downloading the assembled code into the PIC18F4321 on the breadboard using the PICKit3, the following configuration commands should be inserted after "#include <P18F4321.INC>" as follows:

```
include  <P18F4321.INC>
config  OSC  =  INTIO2          ; Select internal oscillator
config  WDT =  OFF              ; Watchdog Timer OFF
config  LVP   =  OFF            ; Low Voltage Programming OFF
config  BOR  =   OFF            ; Brown Out Reset OFF
```

Note that when the $\overline{\text{MCLR}}$ pin is activated for a hardware reset, the PGM is set to HIGH, the LVP (Low Voltage Programming) is enabled, and the PIC18F will enter the "Low Voltage Programming" mode. Since the PGM pin and the pin RB5 (bit 5 of Port B) are multiplexed, disabling the LVP mode frees this pin to be used as RB5 for digital I/O.

After downloading the assembled code into the PIC18F4321 on the breadboard using the PICKit3, the above configuration commands should be inserted after <p18f4321.inc> in the PIC18F assembly language program before the start of the program. This is because the clock source must be available before the program can run.

In order to accomplish the same with the C18 compiler for C-programs with I/O to work, the configuration commands using the directive "#pragma config" are necessary. After downloading the compiled C-code into the PIC18F4321 on the breadboard using the PICKit3, the following configuration commands should be inserted after #include <p18f4321.h> as follows:

```
#include  <p18f4321.h>
#pragma  config  OSC  =  INTIO2  // Select internal oscillator
#pragma  config  WDT =  OFF      // Watch Dog Timer OFF
#pragma  config  LVP   =  OFF    // Low Voltage Programming OFF
#pragma  config  BOR  =   OFF    // Brown Out Reset OFF
```

The program must follow the above statements. This is because the clock source must be available before the program can run.

For simplicity, the PIC18F4321 internal clock in the default mode will be used in this book to illustrate the various hardware aspects of the PIC18F4321.

8.1.5 A simplified setup for the PIC18F4321

Figure 8.7 shows a simplified setup for the PIC18F4321 microcontroller using the default clock of 1 MHz. Appendix H shows the hardware and software aspects of how to interface the PIC18F4321 to a Personal computer or a laptop using PicKit3. This setup can be used for performing inexpensive meaningful experiments in laboratories using a breadboard.

There are two pairs of pins on the PIC18F4321 that must be connected to power and the ground; pins 11 (VDD) and 32 (VSS) must be connected directly to +5V and, pins 12 and 31 connected directly to ground. Note that the operating voltage for VDD is between 4.2V and 5.5V.

The manual reset circuit is connected to pin 1 of the PIC18F4321 chip. When the pushbutton is activated, the PIC18F4321 is reset. This also allows for an easy way to restart a program in the PIC18F4321.

8.1.6 Downloading programs into the PIC18F4321 using the PICKit3 interface

The PICKit3 interface allows downloading assembled or compiled programs from a personal computer or a laptop into the program memory (Flash memory) of a PIC18F4321 inserted on a breadboard. This can be accomplished following the steps provided below:

1. First, the PIC18F4321-based hardware for a particular application can be designed and built on a breadboard.

2. Next, a PIC18F assembly or a C-program can be written to perform the specified task. In this book, all PIC18F assembly language programs are assembled using Microchip MPLAB PIC18F assembler. Appendix F provides a tutorial showing a step-by-step procedure to download the PIC18F assembler from Microchip website (www.microchip.com) and assemble a simple PIC18F assembly language program.

For the C-program, a tutorial or a step-by-step procedure (APPENDIX G) to download the MPLAB C18 v3.47 compiler from the Microchip website (www.microchip.com) and then to compile a simple C-program is shown. Microchip introduces newer versions of the C18 compiler frequently. One may download the C18 v3.47 from the archive or a newer version from the Microchip website.

3. The assembled or compiled program in a personal computer or a laptop can then be downloaded into the PIC18F4321 chip inserted on the breadboard using Microchip's PICKit3 interface. Note that one end of the PICKit3 is connected to the personal computer or laptop via USB; the other end of the PICKit3 is connected to the appropriate pins of the PIC18F4321 chip. A step-by-step procedure for accomplishing this is provided in Appendix H.

4. Finally, after successful downloading of the desired program, the PICKit3 can be disconnected from the breadboard and the computer. The PIC18F4321 chip on the breadboard can then execute the programs downloaded into its memory upon activating the RESET push-button. Thus, the desired task can be accomplished.

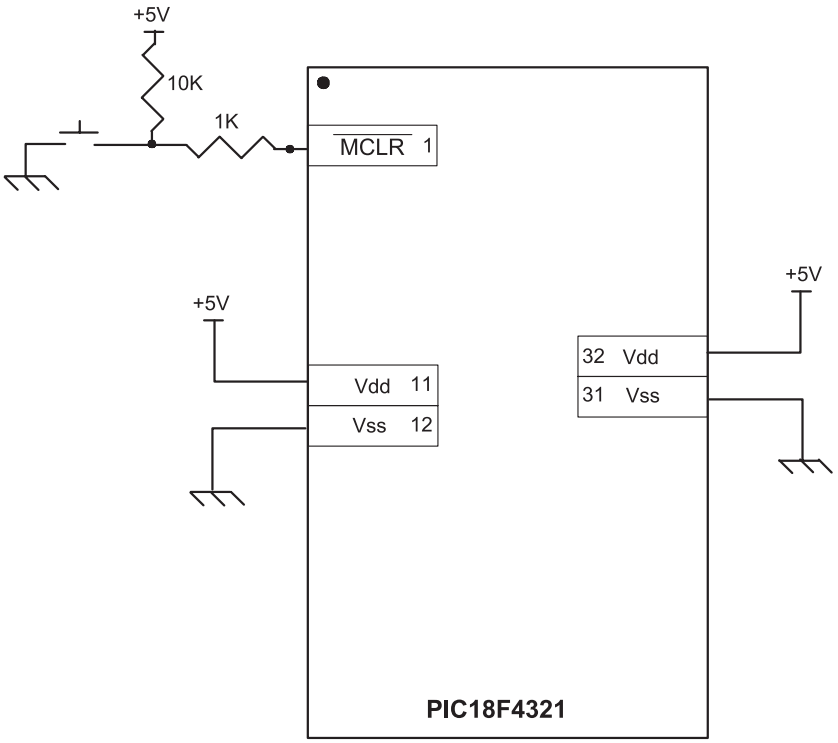


FIGURE 8.7 Simplified PIC18F4321 Setup

8.2 PIC18F4321 Programmed I/O

As mentioned before, a microcontroller communicates with an external device using programmed I/O via one or more registers (comprised of flip-flops) called *I/O ports*. Each bit in the port can be configured individually as either input or output. Each port can be configured as an input or output port by another register usually called the *Data Direction Register (DDR)*. The port contains the actual input or output data. The data-direction register is a register internal to the CPU, and can be used to configure the bits in the port as inputs or outputs.

Each bit in the port can be set up as an input or output, normally by writing a 0 or a 1 in the corresponding bit of the DDR. The PIC18F microcontroller makes an I/O port bit an input by writing a '1' in the corresponding bit in DDR. On the other hand, writing a '0' in a particular bit in DDR will configure the corresponding bit in the port as an output.

For example, if an 8-bit DDR in the PIC18F contains 0xCB (CB Hex), the corresponding port is defined as shown in Figure 8.8. In this example, because 0xCB (1100 1011) is stored in the data-direction register, bits 0, 1, 3, 6, and 7 of the port are set up as inputs, and bits 2, 4, and 5 of the port are defined as outputs. The microcontroller can then send output to external devices such as LEDs, connected to bits 2, 4, and 5, through a proper interface.

Similarly, the microcontroller can input the status of external devices, such as switches, through bits 0, 1, 3, 6, and 7. To input data from the input switches, the microcontroller inputs the complete byte, including the bits to which LEDs are connected. While receiving input data from an I/O port, however, the microcontroller places a value, probably 0, at the bits configured as outputs and the program must interpret them as "don't cares." At the same time, the microcontroller's outputs to bits configured as inputs are disabled.

I/O ports are addressed using either standard I/O or memory-mapped I/O techniques. Using *Standard I/O* or sometimes called *port I/O* (also called *isolated I/O* by Intel), the CPU outputs an internal signal such as the $M/\overline{IO}$ for memory and I/O units on the microcontroller chip. The CPU outputs a HIGH on $M/\overline{IO}$ to indicate to memory and the I/O that a memory operation is taking place. A LOW output from the CPU to $M/\overline{IO}$ indicates an I/O operation. Execution of an IN or OUT instruction makes the $M/\overline{IO}$ LOW, whereas memory-oriented instructions, such as MOVE, drive the $M/\overline{IO}$ to HIGH.

In standard I/O, the CPU uses the $M/\overline{IO}$ output signal to distinguish between I/O and memory. Intel microcontrollers such as the 8051 use standard I/O.

In *memory-mapped I/O*, the CPU does not probably have to use the $M/\overline{IO}$ pin. Instead, the CPU uses an unused address pin to distinguish between memory and I/O. The CPU uses a portion of the memory addresses to represent I/O ports. The I/O ports are mapped as part of the microcontroller's main memory addresses which may not exist

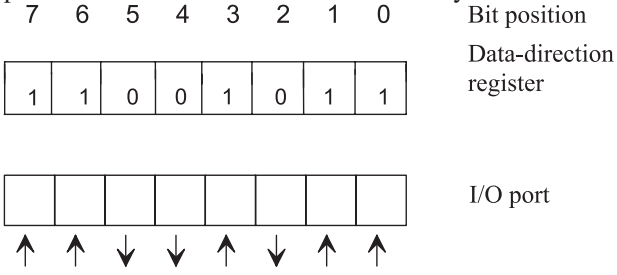


FIGURE 8.8 I/O port with the corresponding data-direction register.

physically, but are used by the microcontroller's memory-oriented instructions, such as MOVE, to generate the necessary control signals to perform I/O. The PIC18F uses memory-mapped I/O.

The PIC18F4321 contains five ports namely Port A (8-bit), Port B (8-bit), Port C (8-bit), Port D (8-bit) and Port E (4-bit). Most of the pins of the PIC18F4321 I/O ports are multiplexed with peripheral I/O pins on the chip. Note that peripheral I/O pins include signals to support on-chip peripheral I/O such as ADC and hardware timers. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin.

For simple I/O operation, three latches are associated with each I/O port bit. They are:

1. TRIS (Tristate) latch)
2. Input latch
3. Output latch

Writing a '1' in the TRIS latch will configure the corresponding bit in the port as an input. Writing a '0' at a particular bit in the TRIS latch will configure the corresponding bit in the port as an output.

A simplified model for a single pin of a generic I/O port is shown in Figure 8.9. This circuit does not include the peripheral functions that may be multiplexed to the I/O pin. The circuit in Figure 8.9 is a simplified version of the internal circuitry associated with an I/O port bit. For simplicity, other components connected to this circuit are not shown. This circuit will provide a basic understanding of how a port bit is configured as an input or an output. Note that Figure 8.9 shows three D latches, namely input latch, TRIS (Tristate) latch, and output latch.

Writing a '1' in the TRIS latch using the PIC18F instruction will make its Q output HIGH. This will enable the input buffer and disable the Output buffer. Thus the selected I/O pin of the I/O port will be configured as an input bit. Upon execution of an input instruction, the $\overline{RD}$ line will be LOW. The inverter at the bottom of Figure 8.9 will make EN line of the input latch HIGH. Hence, the I/O pin connected at the D input of the Input latch via the enabled Input buffer will be transferred to the selected data bus pin of the PIC18F.

Writing a '0' in the TRIS latch using the PIC18F instruction will make its Q output LOW. This will enable the Output buffer, and disable the Input buffer. Thus the selected I/O pin of the I/O port will be configured as an output bit. Upon execution of an output instruction, $\overline{WR}$ pin connected to the clock of the Output latch will transfer the selected data bus pin of the PIC18F to the I/O pin via the enabled Output buffer.

Table 8.2 shows a list of the PIC18F4321 I/O ports along with the associated TRISx (the generic name is DDRx) registers where 'x' is the port name (A, B, C, D, E). Note that these ports and data direction registers are mapped as data memory addresses by Microchip; these registers are called "Special Function Registers (SFR's)" in the PIC18F4321. These addresses are also included in the table for convenience. The MPLAB PIC18F assembler and C18 compiler automatically insert these addresses (binary equivalent of the hex values) in an assembly or a C-program when the programmer uses Port and TRIS names in the program. This means that PORT A through PORT E and TRISA through TRISE can be directly used in a program rather than their hex values. This is very convenient for the programmer.

Note that all bits of Ports A through D are available for general I/O operation. On the other hand, only three bits of Port E (bits 0, 1, 2) are available for general I/O.

The fourth pin (bit 3) of PORTE ($\overline{\text{MCLR}}$/VPP/RE3) is an input only pin. Its operation is controlled by the MCLRE Configuration bit in a special register called CONFIG3H (Configuration Register 3 High) . When selected as a port pin (MCLRE = 0), it functions as a digital input only pin; as such, it does not have TRIS associated with configuring RE3. Otherwise, it functions as the device's Master Clear input. In either configuration, RE3

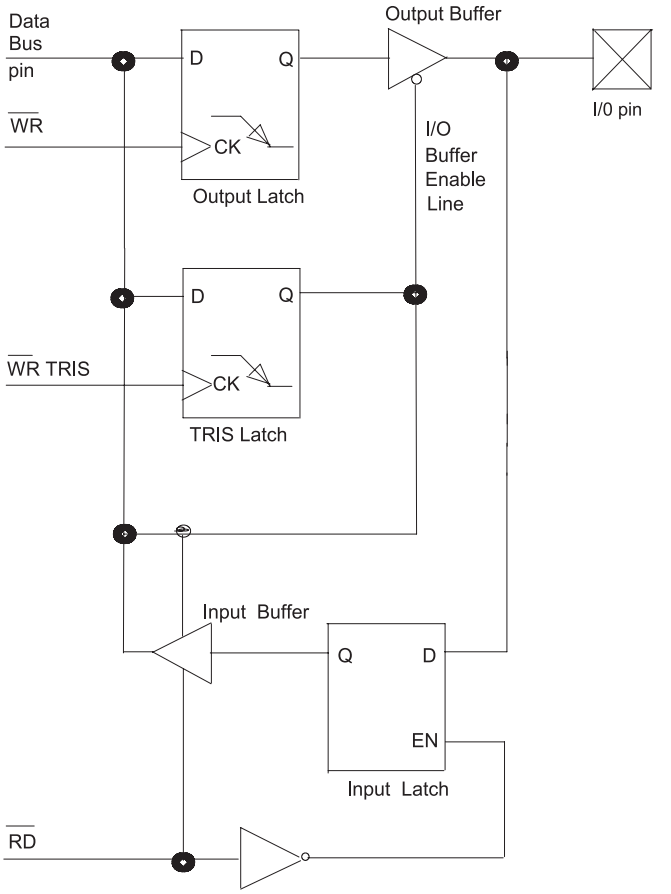


FIGURE 8.9 Generic I/O port operation (Simplified)

TABLE 8.2 PIC18F4321 I/O PORTS, TRISx REGISTERS, ALONG WITH ADDRESSES
(Upon RESET, all ports are configured as inputs)

Port Name	Size	Mapped SFR address	Comment
PORTA	8-bit	0xF80	PORTA
TRISA	8-bit	0xF92	Data Direction Register for PORTA
PORTB	8-bit	0xF81	PORTB
TRISB	8-bit	0xF93	Data Direction Register for PORTB
PORTC	8-bit	0xF82	PORTC
TRISC	8-bit	0xF94	Data Direction Register for PORTC
PORTD	8-bit	0xF83	PORTD
TRISD	8-bit	0xF95	Data Direction Register for PORTD
PORTE	3-bit	0xF84	PORTE
TRISE	3-bit	0xF96	Data Direction Register for PORTE

also functions as the programming voltage input (VPP) when programming the on-chip memory.

In the PIC18F, each bit in a port is assigned with a generic name Rxn where 'x' is A for Port A, B for Port B, C for Port C, D for Port D, and E for Port E, and 'n' is the bit number. This means that RA0 will indicate bit 0 of Port A. Similarly, in the PIC18F, each bit in a data direction register such as TRISx is assigned with a generic name TRISxn where 'x' is A for TRISA, B for TRISB, C for TRISC, D for TRISD, and E for TRISE, and 'n' is the bit number. As an example, TRISA0 will indicate bit 0 of the TRISA data direction register.

8.2.1 I/O instructions in PIC18F assembly

The PIC18F does not provide IN or OUT instructions for inputting from or outputting to ports. The PIC18F uses memory mapped I/O. Therefore, typical memory-oriented instructions such as MOVWF, MOVF, MOFFF can be used for inputting from or outputting to ports. As an example, consider the PIC18F instruction MOVF PORTD, W which will input the contents of PORTD into WREG. The MOVWF PORTC instruction, on the other hand, will output the contents of WREG into PORTC.

Data can also be outputted from one port to another. For example, the MOVFF PORTC, PORTD instruction will output the contents of PORTC to PORTD.

After configuring a bit in a port as an output bit using the corresponding TRISx register, the PIC18F bit-oriented instructions such as BSF and BCF can be used to output a '0' or a '1' to the specified bit. For example, the instruction BSF PORTD, RD6 will set bit 6 of PORTD to one; in other words, the PIC18F will output a '1' to bit 6 of port D. The instruction BCF PORTC, RC3 , on the other hand, will clear bit 3 of PORTC to zero; in other words, the PIC18F4321 will output a '0' to bit 3 of PORTC.

8.2.2 Configuring PIC18F4321 I/O ports using PIC18F assembly

As mentioned before, writing a '1' at a particular bit position in the TRISx register will make the corresponding bit in the associated port as an input. On the other hand, writing a '0' at a particular bit position in the TRISx register will make the corresponding bit in the associated port as an output. Upon reset all TRISx registers are automatically loaded with 1's, and hence, all ports will be configured as inputs.

Next, in order to illustrate how PIC18F4321 ports are configured using the associated TRISx registers, consider the following PIC18F instruction sequence:

```
MOVLW    0x34    ; Move 0x34 into WREG
MOVWF    TRISD   ; Configure PORT D
```

In the above instruction sequence, MOVLW loads WREG with 34 (hex), and then moves this data into TRISD (8-bit data direction register for PORTD) containing 34(Hex) (the corresponding port is defined as shown in Figure 8.10). In this example, because 34H (0011 0100) is written into TRISA, bits 0, 1, 3, 6, and 7 of the port are set up as outputs, and bits 2, 4, and 5 of the port are defined as inputs. The microcontroller can then send output to external devices, such as LEDs, connected to bits 0, 1, 3, 6, and 7 through a proper interface. Similarly, the PIC18F4321 can input the status of external devices, such as switches, through bits 2, 4 and 5. To input data from the input switches, the PIC18F4321 inputs the complete byte, including the bits to which output devices such as LEDs are connected. While receiving input data from an I/O port, however, the PIC18F4321 places a value, probably 0, at the bits configured as outputs and the program must interpret them as

"don't cares." At the same time, the PIC18F4321's outputs to bits configured as inputs are disabled.

The PIC18F instructions such as SETF and CLRF can be used to configure I/O ports. For example, to configure all bits in PORTC as inputs, and PORTD as outputs, SETF or CLRF instructions can be used as follows:

```
SETF    TRISC   ; Set all bits in TRISC to 1's and configure
                ; configure Port C as an input port.
CLRF    TRISD   ; Clear all bits in TRISD to 0's and configure
                ; configure PORTD as an output port
```

Also, a specific bit in a port can be configured as an input or as an output using PIC18F bit-oriented instructions such as BSF and BCF. For example, the instruction "BSF TRISD, 7" or "BSF TRISD, TRISD7" will make bit 7 of PORTD an input bit. On the other hand, "BCF TRISC, 1" or "BCF TRISC, TRISC1" will make bit 1 of Port C an output bit.

Note that configuring PORTA, PORTB, and PORTE is different than configuring PORTC and PORTD. This is because certain bits of PORTA, PORTB, and PORTE are multiplexed with analog inputs. For example, bits 0-3 and bit 5 of PORTA are multiplexed with analog inputs AN0 -AN4, bits 0-4 of PORTB are multiplexed with analog inputs AN8-AN12, and bits 0-2 of PORTE are multiplexed with analog inputs AN5-AN7 (Figure 8.1, PIC18F4321 pin diagram). When a port bit is multiplexed with an analog input, bits 0-3 of a register called ADCON1 (A/D Control Register 1) must be used to configure the port bit as an input. The other bits in ADCON1 are associated with the A/D converter.

Figure 8.11 shows the ADCON1 register along with the associated bits for digital I/O. When bits 0 through 3 of the ADCON1 register are loaded with 1111, the analog inputs (AN0- AN12) multiplexed with the associated bits of PORTA, PORTB, and PORTE (Bits 0, 1, 2) are configured as digital inputs. This will also make these port bits as inputs automatically; the corresponding TRISx registers are not required to configure the ports A, B, and E as inputs. However, for configuring these ports as outputs, the corresponding TRISx bits must be loaded with 0's. The following examples will illustrate this.

For example, the following instruction sequence will configure all 13 port bits multiplexed with AN0 - AN12 as input bits, and TRISx registers are not required:

```
MOVLW   0x0F        ; Move 0xF into WREG
MOVWF   ADCON1      ; Move WREG into ADCON1
```

Next, in order to configure bit 1 of PORTA, all bits of PORTB, and bit 0 of PORTE as outputs, the following instruction sequence can be used:

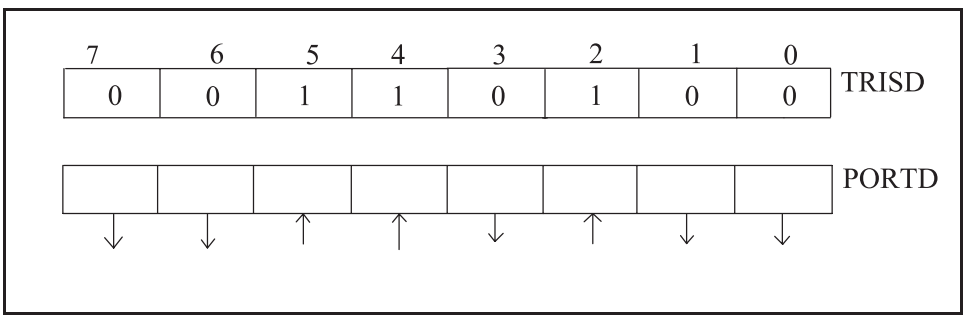


FIGURE 8.10 PORT D along with TRISD

```
MOVLW     0x0F            ; Move  0xF into WREG
MOVWF     ADCON1          ; Move WREG into ADCON1
BCF       TRISA, TRISA1   ; Configure bit 1 of PORTA as output bit
CLRF      TRISB           ; Configure PORTB as  an output port
BCF       TRISE, TRISE0   ; Configure bit 0 of PORTE as output bit
```

It should be mentioned that if a bit of an I/O port in the PIC18F family of microcontrollers such as the PIC18F4321 is multiplexed with an analog input, the bit must be configured as an input using the ADCON1 register; the same bit can be configured as an output using the corresponding bit in the associated TRISx registers. However, if a port bit is not multiplexed with an analog input, it should be configured as an input or an output using the associated TRISx registers.

8.2.3 Configuring PIC18F4321 I/O ports using C

It is relatively easier to configure Port C and Port D as an input or output port. Each port is associated with a corresponding data direction register called TRISx in the PIC18F4321. Table 8.2 lists Ports A through E along with their corresponding TRISx registers. A bit in a TRISx register is used to configure the corresponding bit in the associated port as an input or output. For example, writing a '1' at bit 0 of TRISC will make bit 0 of Port C an input bit. On the other hand, writing a '0' at bit 1 of TRISC will make bit 1 of Port C an output bit. Note that upon hardware reset, all ports are automatically configured as inputs.

The following C language statements will make all bits of Port C and Port D as inputs and outputs:

```
TRISC = 0xFF;          // Configure PORT C as an input port
TRISD = 0 ;            // Configure PORT D as an output port
```

The MPLAB C18 compiler provides a built-in feature for configuring a port bit. This allows the programmer to address a single bit in a port without changing the other bits in the port. This means that the PIC18F ports are bit addressable. For example, the C-statement "PORTCbits. RC0;" can be used to access bit 0 of PORTC. Note that 'bits' in "PORTCbits.RC0" must be lower-case for the C18 compiler. Each bit in the TRISx registers can be accessed in the same manner. As an example, the C-statement "TRISCbits. TRISC5 =0;" will configure bit 5 of PORTC as an output bit; a '1' can then be outputted to bit 5 of PORTC using the following C- statement "PORTCbits.RC5 = 1;".

Furthermore, the C18 directive "#define" can be used to declare a bit in a port. Note that "#define" is a preprocessor directive in C. It is used to define the preprocessor macros for the texts. For example, the name 'portbit' can be declared as bit 2 of PORTC as follows:

```
# define portbit PORTCbits.RC2     // Declare a bit (bit 2) of PORTC
```

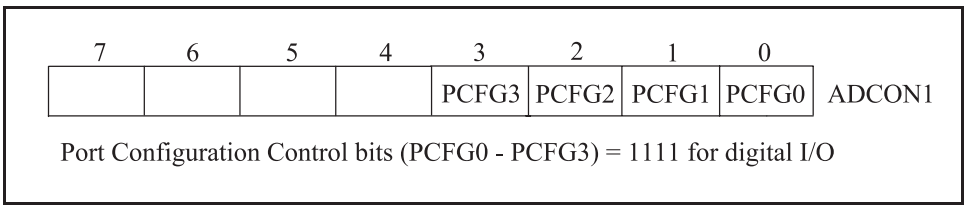


FIGURE 8.11 ADCON1 register for digital I/O

Next, bit 2 of PORTC can be configured as an output bit with the name 'portbit' using the following C-statements:

```
# define portbit PORTCbits.RC2    // Declare a bit (bit 2) of PORTC
TRISCbits.TRISC2 = 0 ;            // Configure bit 2 of PORTC as an output
```

Now, a '1' can be output to bit 2 of PORT C using the following statement:
portbit = 1;

Similarly, the statement "portbit = 0;" will output a '0' to bit 2 of PORTC
Next, bit 3 of PORTD can be configured as an input by writing a '1' at bit 3 of TRISD as follows:

```
# define switch PORTDbits.RD3    // Declare a bit (bit 3) of PORTD
TRISDbits. TRISD3 = 1;           // Configure bit 3 of PORTD as an input
```

Now, the name 'switch' (bit 3 of PORTD) can be used as an input bit in a C-program.

Note that the names 'portbit' and 'switch' in the above are chosen arbitrarily.

As mentioned before, configuring PORTA, PORTB, and PORTE is different than configuring PORTC and PORTD. This is because certain bits of PORTA, PORTB, and PORTE are multiplexed with analog inputs. For example, bits 0-3 and bit 5 of PORTA are multiplexed with analog inputs AN0 -AN4, bits 0-4 of PORTB are multiplexed with analog inputs AN8-AN12, and bits 0-2 of PORTE are multiplexed with analog inputs AN5-AN7 (Figure 8.1, PIC18F4321 pin diagram).

When a port bit is multiplexed with an analog input, bits 0-3 of ADCON1 (A/D Control Register 1) must be used to configure the port bit as input. The other bits in ADCON1 are associated with the A/D converter. Figure 8.11 shows the ADCON1 register along with the associated bits for digital I/O. When bits 0 through 3 of the ADCON1 register are loaded with 1111, the analog inputs (AN0- AN12) multiplexed with the associated bits of PORTA, PORTB, and PORTE (bits 0-2) are configured as digital I/O. This will also make these ports as inputs automatically; the corresponding TRISx registers are not required to configure the ports. This is because upon reset, TRISA, TRISB, and TRISE (0-2) are loaded with 1's.

However, for configuring these ports as outputs, the corresponding TRISx bits must be loaded with 0's. The following examples will illustrate this.

For example, the following C statement will configure all 13 port bits multiplexed with AN0 - AN12 as inputs:

```
ADCON1 = 0x0F ; // Configure 13 pins of ports A, B, and E as digital I/O
```

As mentioned before, TRISA, TRISB, and TRISE (bits 0-2) are automatically loaded with 1's after reset. Hence, PORTA, PORTB, and PORTE are automatically configured as inputs. This means that after reset, loading 0x0F into ADCON1 will make PORTA, PORTB, and PORTE (0-2) inputs. TRISA, TRISB, and TRISE are not required to configure these ports as inputs.

For configuring PORTA, PORTB, and PORTE as outputs, register ADCON1 must be loaded with 0x0F to make PORTA, PORTB, and PORTE digital I/O. Registers TRISA, TRISB, and TRISE can then be loaded with 0x00 to configure these ports as outputs.

This can be accomplished by the following C-statements:

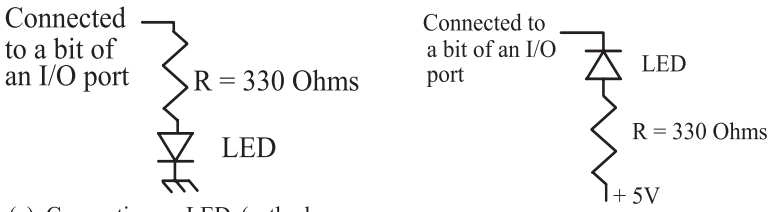
```
ADCON1 = 0x0F; // Configure ports A, B, and E as digital I/O
TRISA = 0;     // PORTA is output port
TRISB = 0;     // PORTB is output port
TRISE = 0;     // PORTE (bits 0-2) is output port
```

Also, to configure bit0 of PORTA as output, bit1 of PORTB as output and bit1 of PORTE as output, the following C-statements can be used:

```
ADCON1 = 0x0F;        // Configure ports A, B, and E as digital I/O
TRISAbits.TRISA0 = 0; // Bit0 of PORTA is output
TRISBbits.TRISB1 = 0; // Bit1 of PORTB is output
TRISEbits.TRISE1 = 0; // Bit1 of PORTE is output
```

8.2.4 Interfacing LED's (Light Emitting Diodes) and Seven-segment displays

The PIC18F sources and sinks adequate currents so that LEDs and seven-segment displays can be interfaced to the PIC18F without buffers (current amplifiers) such as 74HC244. An LED can be connected in two ways. Figures 8.12(a) and (b) show these configurations.



(a) Connecting an LED (cathode groumded) to an I/O port bit

(b) Connecting an LED (anode tied to 5V) to an I/O port bit

FIGURE 8.12 Interfacing LED to PIC18F

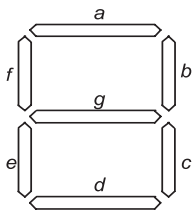


FIGURE 8.13 A seven-segment display

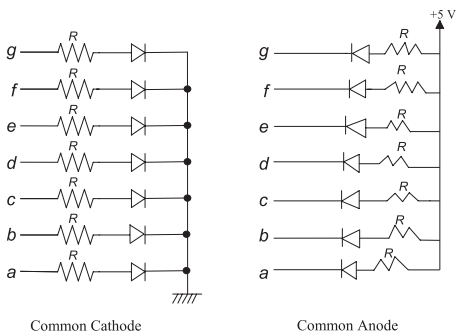


FIGURE 8.14 Seven-segment display configurations

In Figure 8.12(a), the PIC18F will output a HIGH to turn the LED ON; the PIC18F will output a 'LOW' to turn it OFF. In Figure 8.12(b), the PIC18F will output a LOW to turn the LED ON; the PIC18F will output a 'HIGH' to turn it OFF. Also, when an LED is turned on, a typical current of 10 mA flows through the LED with a voltage drop of 1.7V. Hence,

$$R = \frac{5 - 1.7}{10\ mA} = \frac{5 - 1.7}{10\ mA} = 330$$

As discussed in Chapter 3, a seven-segment display can be used to display, for example, decimal numbers from 0 to 9. The name "seven segment" is based on the fact that there are seven LEDs — one in each segment of the display. Figure 8.13 shows a typical seven-segment display.

Figure 8.14 shows two different seven-segment display configurations namely common cathode and common anode. Note that Figures 8.13 and 8.14 are redrawn from Chapter 3 for convenience. In Figure 8.13, each segment contains an LED. All decimal numbers from 0 to 9 can be displayed by turning the appropriate segment "ON" or "OFF". For example, a zero can be displayed by turning the LED in segment *g* "OFF" and turning the other six LEDs in segments *a* through *f* "ON." There are two types of seven segment displays. These are common cathode and common anode. In a common cathode arrangement, the microcontroller can be programmed to send a HIGH to light a segment and a LOW to turn it off. In a common anode configuration, on the other hand, the microcontroller can send a LOW to light a segment and a HIGH to turn it off. In both configurations, R = 330 ohms can be used.

Figure 8.15 shows a typical interface between the PIC18F4321 and a common cathode seven-segment display via PORT C. Each bit of PORT C is connected to a segment of the seven-segment display via a 330 ohm resistor. Note that the seven resistors are not shown in the figure. A common anode seven-segment display can similarly be interfaced to the PIC18F4321.

8.2.5 Programmed I/O examples using PIC18F assembly

In this section, programmed I/O examples using PIC18F assembly language are included. As mentioned before, in order for the programs to work properly, the following "config" statements should be included in all the programming examples after the "< include p18F4321.inc>"
statement as follows :

```
include <p18f4321.inc>
config  OSC  =  INTIO2   ; Select internal oscillator
config  WDT =  OFF       ; Watch Dog Timer OFF
config  LVP   =  OFF     ; Low Voltage Programming OFF
config  BOR  =   OFF     ; Brown Out Reset OFF
```

Note that in the above, the directive "include" includes the file "p18F4321.inc" in the assembly language program. The file "p18f4321.inc" contains standard library routines that are used in the P18f assembly language program. The above statement "include <p18f4321.inc>" instructs the MPLAB assembler to include the source code from the file "p18f4321.inc" into the program. The file p18f4321.inc is a PIC18F4321-specific file, and primarily contains definitions for the variables used to access the Special Function Registers (SFRs) of the PIC18F4321. For example, PORTA of the PIC18F4321 is mapped as a Special Function Register (SFR) with data memory address 0xF80. The "p18f4321" file contains this information. This means that the user can use PORTA in the C-program.

The compiler will insert the address 0xF80 during compilation whenever it encounters PORTA in the program.

The "config" directive in the above tells the MPLAB assembler to perform a specific action before assembly. This directive can be used to turn certain on-chip devices ON or OFF. For example, the statement "config WDT = OFF " placed after the "include" directive will turn the watchdog timer OFF.

Example 8.1 Write a PIC18F assembly language program at 0x100 to drive an LED connected to bit 0 of Port D based on a switch input to bit 0 of Port C as shown in Figure 8.16. If the switch is opened, turn the LED OFF; turn the LED ON if the switch is closed.

Solution

From Figure 8.16, since the cathode of the LED is connected to bit 0 of Port C, a '0' output from the PIC18F4321 will turn the LED ON, and a '1' will turn it OFF. The PIC18F sinks (for LOW output) adequate current to turn an LED OFF or ON without any buffer such as 74HC244; only a current limiting resistor R = 330 ohm is required.
The 0PIC18F assembly language program is provided below:

```
INCLUDE     <P18F4321.INC>
ORG         0x100
BSF         TRISC, TRISC0  ; Configure bit 0 of Port C as an input
```

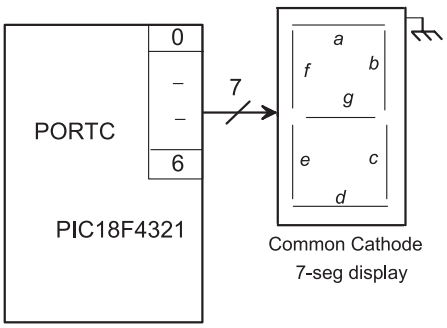


FIGURE 8.15 PIC18F4321 interface to a common cathode seven-segment display via PORT C

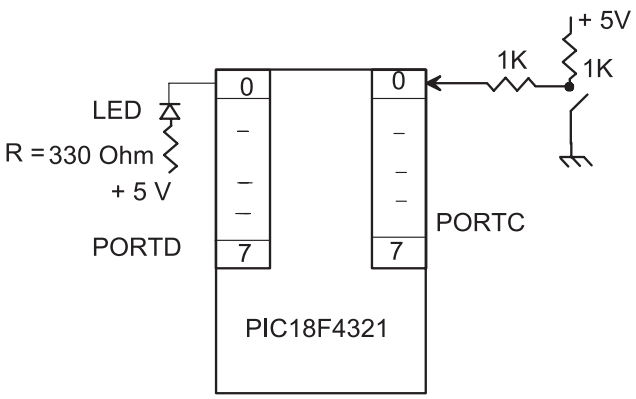


FIGURE 8.16 Figure for Example 8.1

```
        BCF     TRISD, TRISD0 ; Configure bit 0 of Port D as an output
START   MOVFF   PORTC, PORTD  ; Output switch input to LED
        BRA     START         ; Repeat
        END
```

In the above, since the switch and the LED data are aligned (both connected at bit 0 of the respective ports), the MOVFF PORTA, PORTB instruction directly inputs the switch input from bit 0 of PORTA and outputs to bit 0 of PORTB. Also, the infinite loop using BRA START will make the LOOP continuous. This means that after execution of the above program once, the LED will be turned ON and OFF automatically as soon as the switch is pressed. However, if SLEEP or FINISH BRA FINISH is placed at the end of the program, the program will be executed once upon activation of the manual reset. Each time the switch is pressed in Figure 8.16, the above program must be re-executed again by activating manual reset.

Example 8.2 A PIC018F4321 microcontroller is required to drive an LED connected to bit 7 of port C based on two switch inputs connected to bits 6 and 7 of port D. If both switches are equal (either HIGH or LOW), turn the LED ON; otherwise turn it OFF. Assume that a HIGH will turn the LED ON and a LOW will turn it OFF. Write a PIC18F assembly program at 0x200 to accomplish this.

Solution

```
        INCLUDE <P18F4321.INC>
        ORG     0x200
        BCF     TRISC, TRISC7 ; Configure bit 7 of PORTC as an output
        SETF    TRISD         ; Configure PORTD as an input port
BACK    MOVF    PORTD, W      ; Input PORTD into WREG
        ANDLW   0xC0          ; Retain bits 6 and 7
        BZ      LEDON         ; If both switches are LOW, turn the LED ON
        SUBLW   0xC0          ; If both switches are HIGH, turn the LED ON
        BZ      LEDON
        BCF     PORTC, RC7    ; Turn LED OFF
        BRA     DOWN
LEDON   BSF     PORTC, RC7    ; Turn LED ON
DOWN    BRA     BACK
        END
```

Example 8.3 The PIC18F4321 microcontroller shown in Figure 8.17 is required to input a BCD number (0-9) via four switches connected to bits 0 through 3 of PORTA, and then output the BCD digit to a common-anode seven segment display connected to bits 0 through 6 of PORTB. Write a PIC18F at address 0x70 to accomplish this by storing seven-segment codes in data memory starting at address 0x100.

Solution

To find the proper values for the display, a table (also called "Lookup table") containing the seven segment code for each BCD digit can be obtained from Figure 8.17 as follows:

	g	f	e	d	c	b	a		Hex:
0:	1	0	0	0	0	0	0	=	0x40
1:	1	1	1	1	0	0	1	=	0x79
2:	0	1	0	0	1	0	0	=	0x24
3:	0	1	1	0	0	0	0	=	0x30
4:	0	0	1	1	0	0	1	=	0x19
5:	0	0	1	0	0	1	0	=	0x12
6:	0	0	0	0	0	1	1	=	0x03
7:	1	1	1	1	0	0	0	=	0x78
8:	0	0	0	0	0	0	0	=	0x00
9:	0	0	1	1	0	0	0	=	0x18

Note that for a common-anode seven segment display, a '0' will turn a segment ON and a '1' will turn it OFF. Also, bit 7 of PORTB is assumed to be '0'. The above table must be stored in the data memory using MOVLW and MOVWF instructions or using the MPLAB assembler directive DB along with the PIC18F assembly language instruction TBLRD (Table Read) with appropriate addressing mode.

The PIC18F assembly language program is provided in the following:

```
INCLUDE <P18F4321.INC>
ORG      0x70
MOVLW    0x40              ; Store 7-segment codes
MOVWF    0x100
MOVLW    0x79
MOVWF    0x101
MOVLW    0x24
MOVWF    0x102
MOVLW    0x30
MOVWF    0x103
MOVLW    0x19
MOVWF    0x104
MOVLW    0x12
```

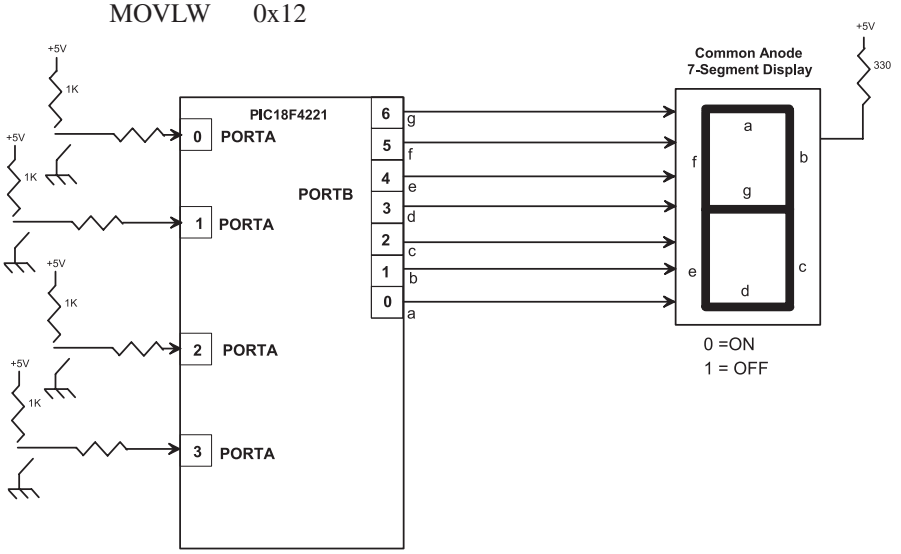


FIGURE 8.17 Figure for Example 8.3

```
        MOVWF   0x105
        MOVLW   0x03
        MOVWF   0x106
        MOVLW   0x78
        MOVWF   0x107
        MOVLW   0x00
        MOVWF   0x108
        MOVLW   0x18
        MOVWF   0x109
        MOVLW   0x0F            ; Configure PORTA as input
        MOVWF   ADCON1
        CLRF    TRISB           ; Configure PORTB as output
        LFSR    0, 0x100        ; Initialize FSR0 with the starting
                                ; address of lookup table
LOOP    MOVF    PORTA, W        ; Input switch data to the WREG
        ANDLW   B'00001111'     ; Mask the lower 4-bits
        MOVFF   PLUSW0, PORTB   ; Output BCD data to PORTB
        BRA     LOOP            ; Loop
        END
```

8.2.6 Programmed I/O examples using C Language

Before we go on, a summary of the basic characteristics of typical C-programs will be discussed in the following:

Preprocessor directives The C-preprocessor directives are part of the C compiler. The purpose of a preprocessor is to execute a specific command associated with the preprocessor. These preprocessors are not part of the C language. Instead, they are part of the C compiler software. Each preprocessor directive begins with the symbol #.

Before compiling a C-program, the C compiler first initiates execution of the preprocessor software module. This module then goes through the program looking for the '#' sign. If the symbol '#' is found, the preprocessor module performs the function specified by the command. The C compiler may then start compiling the program.

Typical preprocessor directives are "#include", "#define", and "pragma config". The "#include" preprocessor directive transfers texts (commands, comments, etc.) from another document into the program. For example, consider

```
#include <p18f4321.h>
```

In the above, the extension .h means header file. Note that the header file includes standard library functions that are used in the C program. The above statement instructs the compiler to include the source code from the file p18f4321.h into the program. The file p18f4321.h is a PIC18F4321-specific header file, and primarily contains definitions for the variables used to access the Special Function Registers (SFRs) of the PIC18F4321. For example, PORTA of the PIC18F4321 is mapped as a Special Function Register (SFR) with data memory address 0xF80. The header file contains this information. This means that the user can use PORTA in the C-program. The compiler will insert the address 0xF80 during compilation whenever it encounters PORTA in the program.

The "#define" preprocessor directive can be used to declare constants. For example, consider

```
#define y 4
```

The "#define" preprocessor will assign 4 to y. Constants defined with the preprocessor "#define" are textual replacements performed before the program is compiled.

The "#pragma config" preprocessor directive tells the compiler to perform a specific action before compilation. This directive can be used to turn certain on-chip devices ON or OFF. For example, the statement "#pragma config WDT = OFF " placed after the "#include" directive will turn the watchdog timer OFF.

In this section, programmed I/O examples using C language are included. As mentioned before, in order for the I/O programs to work properly, the following "# pragma config" statements should be included in all the programming examples after the "#include <p18f4321.h>" statement:

```
#pragma config OSC = INTIO2 // Select internal oscillator
#pragma config WDT = OFF    // Watch Dog Timer OFF
#pragma config LVP = OFF    // Low Voltage Programming OFF
#pragma config BOR = OFF    // Brown Out Reset OFF
```

main ( ) function All C-programs must include a "main ( )" function which can appear only once. The keyword "void" can optionally be placed within the brackets to clarify that there are no parameters. The statement,

```
main (void)
```

means no arguments are required for function call. The" void" type does not provide a result value to the caller. Hence, "main ( )" means that it does not return a value. Since "main" is classified as a function, all codes must be placed within curly brackets.

while (1) In microcontrollers including the PIC18F, after completion of program execution, there is no operating system to return to. Hence, an infinite "while (1)" loop is used to keep the microcontroller running and prevent it from exiting the "main( )". This will prevent attempts in executing codes at undefined memory locations. Note that the keyword "while" defines an iteration loop that continues or breaks based on a condition that is inserted inside the parentheses of the while statement. For example, the statements within the curly brackets in the following "while" loop

```
while ( x <2)
 {
   Statements
 }
```

are executed as long as the variable 'x' is less than 2 and the loop condition is true. Hence, before each iteration, the program determines whether the condition is 'TRUE" (represented by non-zero in C or "FALSE" (represented by zero in C). This means that "while (1)" with '1' inside the loop is always a TRUE condition. Hence, the loop will run forever. Therefore, while(1) is an infinite loop.

PIC18F data types Data types for C are discussed in Chapter 4. An overview of common PIC18F data types will be covered in this section. Common data types with the PIC18F are listed below:

1. unsigned char (8-bit). Since the PIC18F is an 8-bit microcontroller, this is normally used for positive variables (8-bit positive temperature) by the PIC18F.

2. signed char (8-bit): This is called "char" in the PIC18F since signed char is a default data type in the PIC18F. This data type can be used by an 8-bit signed temperature.

3. unsigned int (16-bit): Since memory addresses are unsigned (always positive), this data type can be used by the PIC18F4321 for program memory addresses. Note that the program memory address for the PIC18F4321 is 13-bit wide.

4. signed int (16-bit): Since memory contents are 16-bit wide in the PIC18F4321, this data type can be used to store 16-bit signed data in program memory.

Functions in C As mentioned in Chapter 4, a function in C is similar to subroutine in assembly. Hence, a function is a block of C codes that perform a particular task.

A typical example of a C-function is provided below. The following C-function does not return the result to the main; the reserved word "void" can be used in such a situation. A simple example for the C18 C-compiler for the PIC18F is provided in the following:

```
#include <p18f4321.h>
// Initializations
void CONVERT (void);
void main (void)
{
Statements
CONVERT ( ); // Function Call
}    // end of main
void CONVERT ( )
{ Body of the Function

}
```

Arrays in C As mentioned in Chapter 4, an array contains a number of the same type of data elements stored under a common name in consecutive memory locations. The same type of data elements means that data elements can be, for example, character (char) elements or integer (int) elements.

An array of BCD codes for a common-anode seven-segment display can be created using the C18 compiler of the PIC18F as follows:

```
include <pic18f4321>
   void main ( )
    {
unsigned char code [10] = {0x40, 0x79, 0x24, ox30, 0x19, 0x12, 0x03, 0x78,
                           0x00, 0x18};
     ---------
     ---------
     ---------
     }
```

In the above program, the array name is "code" with 10 (total number of elements) inside the square brackets and each element is listed within curly brackets. For example,

code [0] is element 0 with a value of 0x40, code [1] is element with a value of 0x79, and so on.

Suppose that the common-anode seven -segment display is connected to PORTA of the PIC18F4321 microcontroller. Now, in order to output a code for a particular digit, for example digit 1, the following C code can be used:

```
PORTA = code [1];
```

Delay routines in C Three ways of obtaining time delays using the PIC18F C18 compiler are:

- Using PIC18F on-chip timers
- Using a C-loop
- Using C18 delay functions such as the function Delay 10KTcyx ( )

We describe these in the following.

(a) Using PIC18F on-chip timers: Time delays using the PIC18F hardware timers will be discussed in Chapter 10.

(b) Using a C-loop: Time delays using "for" loop can be obtained as follows:

```
void delay(unsigned int itime)        // 2 ms delay
{
        unsigned int i,j;
        for(i=0; i<itime; i++)
                for(j=0; j<255;j++);
}
```

The above "delay" function in C will provide a time delay of 2 ms using an internal default clock of 1-MHz of the PIC18F4321. In order to determine that the "delay" function will provide a time delay of 2 ms, disassembly of the C-program is given below:

```
DELAY   MOVWF   0x20
LOOP1   MOVLW   D'255'   ; LOOP2 provides 2 ms delay with a count of 255
        MOVWF    0x21
LOOP2   DECFSZ  0X21
        GOTO    LOOP2
        DECFSZ  0x20
        GOTO    LOOP1
        RETURN
```

Using the 1-MHz clock along with PIC18F instruction execution times of Appendix D, it can be shown that the above PIC18F assembly language program will create a time delay of 2 ms. Also, the concepts are described in Chapter 7 associated with obtaining time delays using PIC18F assembly language.

Using the "delay" function in C of 2 ms, more delay time can be obtained by substituting values for itime. For example, with itime = 10, a time delay of 20 ms can be obtained using the C statement:

```
delay (10);
```

(c) Using C18 delay functions such as the function Delay 10KTCYx( ):

The C18 compiler contains a function called "Delay 10KTCYx( )" which allows the programmer to specify a time delay in multiples of 10,000 instruction cycles. This "Delay" function is included in the C18 compiler's header file "delays.h". Note that the instruction cycle is the time taken to execute the machine code of a PIC18F instruction. As mentioned before, the PIC18F performs one machine code instruction every four clock cycles. The details of the function name "Delay 10KTCYx( )" can be broken down as follows:

- Function "Delay" is used to provide a fixed time delay.
- "10K" means ten thousand.
- "TCY" means the instruction cycle, Tcy.
- "x" multiplied by the argument (number) specified in the bracket. The number in the bracket must be an 8-bit unsigned integer (0-255). Hence, the time delay created will be the number in brackets multiplied by 10,000 instruction cycles.

As an example, consider writing a C-program to generate a time delay of 10 sec with an internal clock of 4-MHz using the C18 function "Delay10KTCYx( )".

With OSC = 4Mhz, Timer frequency is (4MHz)/4 = 1MHz. Hence, period is 1μ*sec*. For 10 sec delay, Cycles = (10)/(10^{-6}) = 10^7 = 10,000,000. The C18 function "Delay10KTCYx(250)" will provide delays of 2,500,000 cycles. Performing a loop with the function "Delay10KTCYx(250)" four times will create a delay of 10 sec.

The C-program is provided below:

```
#include <P18F4321.h>
#include <delays.h>     // delay function library
{
unsigned char count;
void main()
   OSCCON = 0x60;  // 4Mhz
   for (count = 0; count < 4; count++)  // Use delay function 4 times
   Delay10KTCYx(250); // Delay in multiples of 10,000 instruction cycles.

}
```

Example 8.4 Assume PIC18F4321. Suppose that three switches are connected to bits 0-2 of PORTC and an LED to bit 6 of PORTD. If the number of HIGH switches is even, turn the LED on; otherwise, turn the LED off. Write a C language program to accomplish this.

Solution

The C language program is shown below:

```
#include <p18f4321.h>
#define    portc0  PORTCbits.RC0
#define    portc1  PORTCbits.RC1
#define    portc2  PORTCbits.RC2
void main (void)
{
     unsigned char mask = 0x07;              // data for masking off upper 5 bits of  PORTC
     unsigned char masked_in;
```

```
    unsigned char xor_bit;
    TRISC = 0xFF;                          // Configure PORTC as an input port
    TRISD = 0;                             // Configure PORTD as an output port
    while(1)
    {
            masked_in  =  PORTC & mask;          //  Mask input  bits
    if (masked_in = = 0)
            PORTD = 0x40;                  // For all low switches (even), turn led on
    else
            xor_bit = portc0 ^ portc1 ^ portc2; // xor input bits
    if (xor_bit = = 0)                     // for even # of high switches ,
            PORTD = 0x40;                  // turn led on
    else
            PORTD = 0;                     // for odd # of high switches, turn led off
    }
}
```

Example 8.5 Assume PIC18F4321. Suppose that it is desired to input a switch connected to bit 4 of PORTC, and then output it to an LED connected at bit 2 of PORTD. Write a C language program to accomplish this.

Solution

The following C code will accomplish this:

```
# include  <P18F4321.h>
# define portc_bitin  PORTCbits.RC4      // Declare a bit (bit 4) of  PORTC
# define portd_bitout  PORTDbits.RD2   // Declare a bit (bit 2) of  PORTD
void main  (void )
{
    TRISCbits.TRISC4 = 1;       // Configure bit 4 of PORTC as an input bit
    TRISDbits.TRISD2 = 0;       // Configure bit 2 of PORTD as an output bit
    while  (1)                  //  halt
          {
    portd_bitout = portc_bitin;   // output switch to LED
          }
}
```

Example 8.6 The PIC18F4321 microcontroller shown in Figure 8.18 is required to output a BCD digit (0 to 9) to a common-anode seven segment display connected to bits 0 through 6 of PORTB. The PIC18F4321 inputs the BCD number via four switches connected to bits 0 through 3 of PORTA. Write a C language program that will display a BCD digit (0 to 9) on the seven segment display based on the switch inputs. Use a lookup table for the seven-segment codes.

Solution

To find the proper values for the display, a table (also called "Lookup table") containing the seven segment code for each BCD digit can be obtained from Figure 8.18 as follows:

	g	f	e	d	c	b	a		Hex:
0:	1	0	0	0	0	0	0	=	0x40
1:	1	1	1	1	0	0	1	=	0x79
2:	0	1	0	0	1	0	0	=	0x24
3:	0	1	1	0	0	0	0	=	0x30
4:	0	0	1	1	0	0	1	=	0x19
5:	0	0	1	0	0	1	0	=	0x12
6:	0	0	0	0	0	1	1	=	0x03
7:	1	1	1	1	0	0	0	=	0x78
8:	0	0	0	0	0	0	0	=	0x00
9:	0	0	1	1	0	0	0	=	0x18

Note that for a common-anode seven segment display, a '0' will turn a segment ON and a '1' will turn it OFF. Also, bit 7 of PORTB is assumed to be '0'.

The C code is provided below:

```
#include <p18f4321.h>
 void main ()
{
unsigned char input;
unsigned char code[10] = {0x40, 0x79, 0x24, 0x30, 0x19, 0x12, 0x03, 0x78, 0x00, 0x18};
ADCON1 = 0x0F; // PORTA is input
TRISB = 0; // PORTB is output
while (1) {
        input = PORTA & 0x0F; // Mask off switch inputs
        PORTB = code [input];  // Output code to 7-segment display via PORTB
        }
}
```

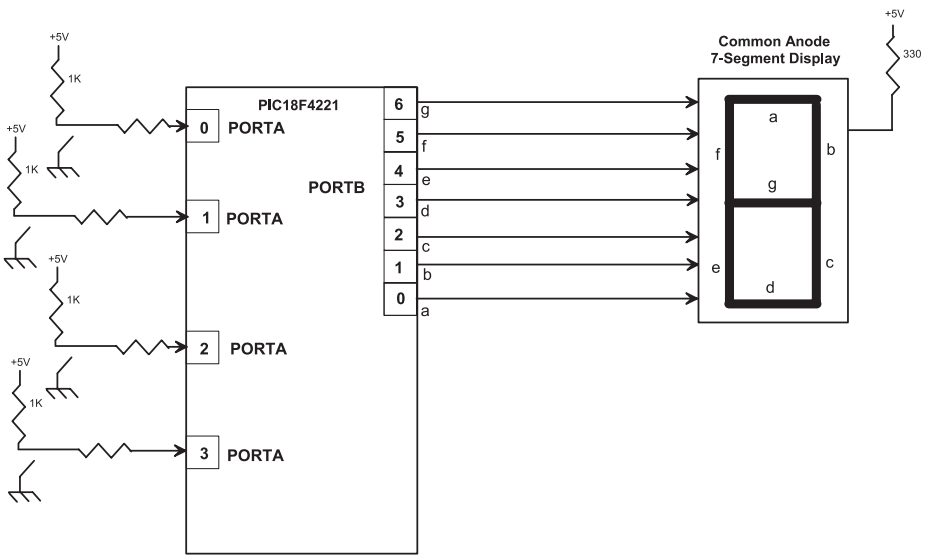


FIGURE 8.18 Figure for Example 8.6

In the above program, first the PORTB is set as an output port and PORTA is set as an input port. A variable ‘input’ is then declared. The program moves to an infinite ‘while’ loop where it will first take the input from the four switches via PORTA, and mask the first four bits. An unsigned char array code is set up in order to contain the seven-segment code for each decimal digit from 0 through 9. The input is used as the index to the array, and the corresponding 7-segment code is sent to PORTB.

QUESTIONS AND PROBLEMS

8.1 Identify the power and ground pins on the PIC18F4321. What is the purpose of using multiple power and ground pins?

8.2 What is the default clock frequency of the PIC18F4321?

8.3 How does the PIC18F obtain the address of the first instruction to be executed? What is this address?

8.4 What is the difference between Power-on reset and manual reset?

8.5 List the PIC18F4321 I/O ports along with their sizes.

8.6 Using PIC18F assembly language, configure:

(a) all bits of PORTC as inputs (b) all bits of PORTD as outputs

(c) bits 0 through 4 of PORTB as inputs (d) all bits of PORTA as outputs

8.7 Repeat Problem 8.6 using C.

8.8 Configure three bits (bits 0-2) of PORTE as outputs using:

(a) PIC18F assembly language (b) C language

8.9 Assume PIC18F4321. Suppose that three switches are connected to bits 0–2 of port C and an LED to bit 6 of port D. If the number of HIGH switches is even, turn the LED ON; otherwise, turn the LED OFF. Write a PIC18F assembly language program to accomplish this. Also, write a C-program to accomplish this using "if-else" construct. Assume that a '1' will turn the LED ON while a '0' will turn it OFF.

8.10 The PIC18F4321 microcontroller is required to drive the LEDs connected to bit 0 of ports C and D based on the input conditions set by switches connected to bit 1 of ports C and D. The I/O conditions are as follows:

- If the input at bit 1 of port C is HIGH and the input at bit 1 of port D is LOW, the LED at port C will be ON and the LED at port D will be OFF.
- If the input at bit 1 of port C is LOW and the input at bit 1 of port D is HIGH, the LED at port C will be OFF and the LED at port D will be ON.

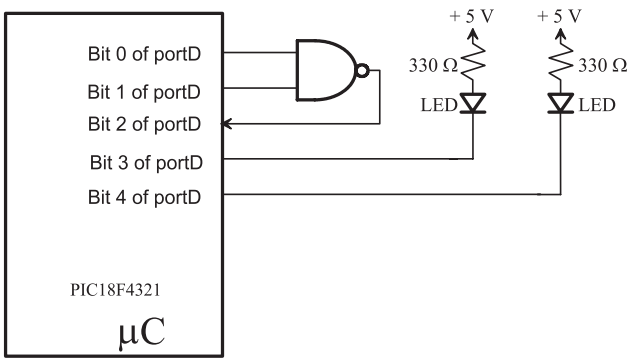


FIGURE P 8.11 (Assume that both LEDs are OFF initially.)

- If the inputs of both ports C and D are HIGH, turn ON both LEDs of Ports C and D. On the other hand, if the inputs of both ports C and D are LOW, turn OFF both LEDs of Ports C and D.
  Assume that a '1' will turn the LED ON while a '0' will turn it OFF.

(a) Write a PIC18F assembly language program to accomplish this.

(b) Write a C language program to accomplish this.

8.11 The PIC18F4321 microcontroller is required to test a NAND gate. Figure P8.11 shows the I/O hardware is needed to test the NAND gate. The microcontroller is to be programmed to generate the various logic conditions for the NAND inputs, input the NAND output, and turn the LED ON connected at bit 3 of port D if the NAND gate chip is found to be faulty. Otherwise, turn the LED ON connected at bit 4 of port D.

(a) Write a PIC18F assembly language program at address 0x100 to accomplish this.

(b) Write a C language program to accomplish this.

8.12 The PIC18F4321 microcontroller is required to add two three-bit numbers stored in the lowest three bits of data registers 0x20 and 0x21 and output the sum (not to exceed 9) to a common-cathode seven-segment display connected at port C as shown in Figure P8.12. Write a PIC18F assembly language main program at address 0x50 that will configure PORTC as output, initalize STKPTR to 0x10, perform the required addition, CALL a subroutine to obtain the 7-segment code for the result and then output to the display. Also, write a subroutine at address 0x70 that will return the 7-segment code based on the result.

8.13 The PIC18F4321 microcontroller (Figure P8.12) is required to add two three-bit numbers entered via DIP switches connected at bits 0-2 and bits 3-5 of Port D and output the sum (not to exceed 9) to a common-cathode seven-segment display connected to Port C as shown in Figure P8.12. Write a C language program to accomplish this by using a lookup table.

8.14 The PIC18F4321 microcontroller is required to input a number from 0 to 9 from an ASCII keyboard interfaced to it and output to an EBCDIC printer. Assume that the keyboard is connected to port C and the printer is connected to port D.

(a) Store the EBCDIC codes for 0 to 9 starting at an address 0x30, and use this lookup table to write a PIC18F assembly language program at address 0x100 to accomplish this.

(b) Write a C-program using a lookup table to accomplish this.

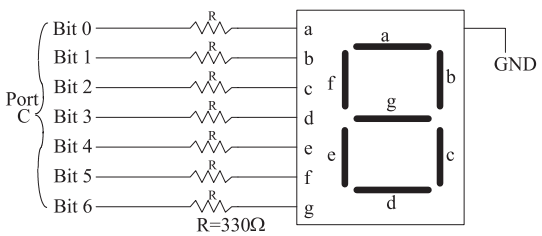


FIGURE P8.12

8.15 Write a C language program to implement a traffic light controller using the PIC18F4321 as follows:

Step 1: Make North-South light Green and East-West light Red for 10 seconds. Check to see if any waiting car is trying to go from east to west and vice versa. If there is a waiting car, go to step 2; otherwise, repeat this step.

Step 2: Make North-South light Yellow and East-West light Red for 2 seconds, and go to Step 3.

Step 3: Make North-South light Red and East-West light Green for 5 seconds, and then go to Step 4.

Step 4: Make North-South light Red, and East-West light Yellow for 2 seconds, and then go to Step 1.
Also, include provision for an emergency input. When this input asserted, a flashing RED light in both directions will be activated.

8.16 The PIC8F4321 microcontroller is required to output a BCD digit (0-9) to a common-anode seven-segment display connected to bits 0-6 of PORTD. The PIC18F4321 inputs the BCD number via four switches connected to bits 0-3 of PORTC. Write a PIC18F main program at address 0x40 that will intialize PORTC and PORTD, initialize STKPTR to 0x10, call a subroutine to obtain the seven-segment code for the BCD input and then output to the display via PORTD. Write the PIC18F Subroutine at address 0x70 that will return the seven-segment code using the RETLW instruction based on the BCD input digit.

9

PIC18F INTERRUPT I/O, LCD, AND KEYBOARD INTERFACING

In this chapter we describe the basics of polled and interrupt I/O. Topics include PIC18F polled and interrupt I/O using both assembly and C. Also, techniques associated with interfacing the PIC18F4321 to a typical LCD, such as the Hitachi HD44780, and a hexadecimal keyboard are covered.

9.1 Basics of Polled I/O vs. Interrupt I/O

There are typically two ways via which programmed I/O can be utilized. These are *unconditional I/O* and *conditional I/O*. The microcontroller can send data to an external device at any time using unconditional I/O. The external device must always be ready for data transfer. A typical example is when the microcontroller outputs a 7-bit code through an I/O port to drive a seven-segment display connected to this port. In conditional I/O, the microcontroller outputs data to an external device via *handshaking*. This means that data transfer occurs via the exchanging of control signals between the microcontroller and an external device. The microcontroller inputs the status of the external device to determine whether the device is ready for data transfer. Data transfer takes place when the device is ready. The flowchart in Figure 9.1 illustrates the concept of conditional programmed I/O or polled I/O.

The concept of polled I/O will now be demonstrated by means of data transfer between a generic microcontroller and an analog-to-digital (A/D) converter or ADC. As mentioned before, the names "A/D converter" and "ADC" will be used interchangeably in this book. Assume that this microcontroller does not have any on-chip A/D converter. Next, consider the A/D converter chip shown in Figure 9.2 . The A/D converter transforms an analog voltage V_x into an 8-bit binary output at pins D_7-D_0. A pulse at the START conversion pin initiates the conversion. This drives the BUSY signal to LOW. The signal stays LOW during the conversion process. The BUSY signal goes to HIGH as soon as the conversion ends. Note that a LOW on the OUTPUT ENABLE transfers the A/D converter's outputs.

The concept of polled I/O can be demonstrated by interfacing an external A/D converter chip to a the microcontroller chip. Figure 9.3 shows such an interfacing example. The user writes a program to carry out the conversion process. When this program is executed, the microcontroller sends a pulse to the START pin of the converter via bit 2 of port A. The microcontroller then checks the BUSY signal by inputting bit 1 of port A to determine if the conversion is completed. If the BUSY signal is HIGH (indicating the

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

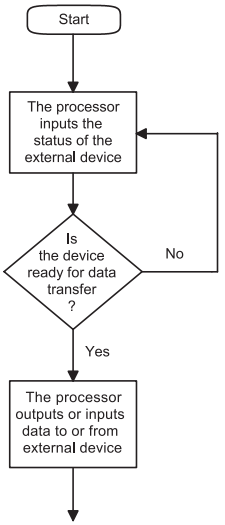


FIGURE 9.1 Flowchart for conditional programmed I/O

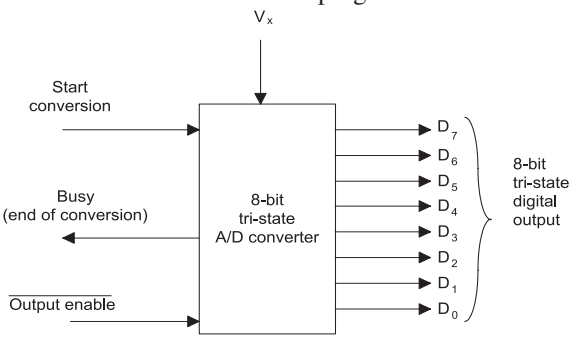


FIGURE 9.2 A/D converter

end of conversion), the microcontroller sends a LOW to the $\overline{\text{OUTPUT ENABLE}}$ pin of the A/D converter. The microcontroller then inputs the converter's D_0-D_7 outputs via port B. If the conversion is not completed, the microcontroller waits in a loop checking for the BUSY signal to go to HIGH. This is called "Conditional or Polled I/O". A disadvantage of polled I/O is that the microcontroller needs to check the status bit (BUSY signal for the A/D converter) by waiting in a loop. This type of I/O transfer is dependent on the speed of the external device (A/D Converter in this case). For a slow device, this waiting may slow down the microcontroller's capability of processing other data in the main program. The interrupt I/O technique is efficient in this type of situation.

Interrupt I/O is a device-initiated I/O transfer. The external device (A/D Converter chip in this case) is connected to a pin called the "interrupt (INT) pin" on the microcontroller chip as shown in Figure 9.4 When the microcontroller needs an I/O transfer with the A/D converter, a user-written program starts the A/D conversion via bit 7 of port A. The A/D converter chip performs conversion independent of the microcontroller. When the conversion is completed, the A/D converter outputs a HIGH on its BUSY pin. This will activate the interrupt pin of the microcontroller. The microcontroller completes the current instruction and normally saves the contents of the current program counter (RETURN address) and the staus register onto the stack.

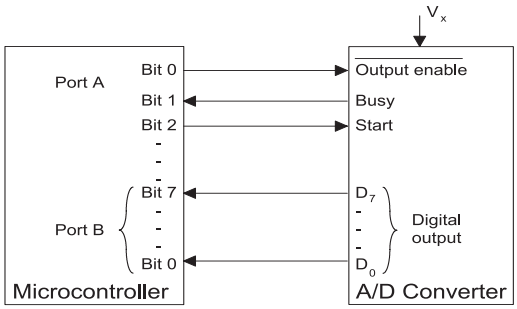


FIGURE 9.3 Interfacing an A/D converter to a generic microcontroller

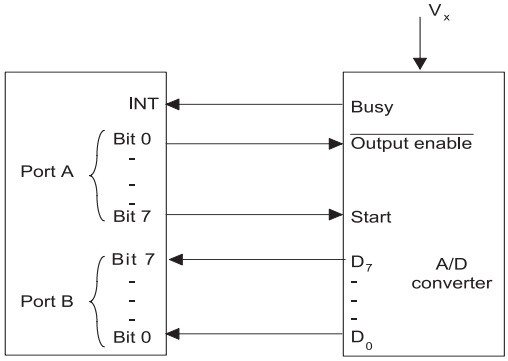


FIGURE 9.4 Microcomputer A/D converter interface via interrupt I/O

The microcontroller then automatically loads an address (normally specified by the microcontroller manufacturer) into the program counter to branch to a subroutine-type program called the "interrupt-service routine." This program is written by the user. The A/D converter chip wants the microcontroller to execute this program to input data. The last instruction of the service routine is a "RETURN from interrupt" which is typically similar in concept to the RETURN instruction used at the end of a subroutine. The RETURN from interrupt instruction pops the previously-saved program counter and the status register. The microcontroller then continues executing the main program at the right address. The RETURN from subroutine, on the other hand, pops only the PC to return to the main program. Note that the PIC18F pushes only the program counter before jumping to the service routine. However, the PIC18F return from interrupt instruction pops the saved program counter, and also enables interrupts.

The process of handling interrupt I/O is very similar in concept to subroutine CALL and RETURN instructions. The basic differences are summarized as follows:

1. Execution of a subroutine program is initiated using software by a microcontroller's 'CALL to subroutine' instruction in the main program. The interrupt, on the other hand, is initiated using hardware by activating an interrupt pin by an external device. In both cases, there are two programs: the main program and the subroutine program in the case of a subroutine whereas the main program and the interrupt service routine (program similar to a subroutine) in the case of the interrupt.

2. Execution of the subroutine CALL instruction, the microcontroller pushes only the current program counter containing the return address onto the stack while upon recognizing an interrupt, the microcontroller such as the PIC18F also pushes the

current program counter onto the stack. However, the instructions 'RETURN from subroutine' and 'RETURN from interrupt' are different. The PIC18F 'RETURN from subroutine' will pop the program counter while the PIC18F 'RETURN from interrupt' will pop the program counter and also enable the interrupts, and then return control to the main program at the right location.

Example 9.1 Assume that the PIC18F4321 microcontroller shown in Figure 9.5 is required to perform the following:

If Vx > Vy , turn the LED ON if the switch is open; otherwise, turn the LED OFF.

(a) Write a PIC18F assembly language program to accomplish the above by polling the comparator output via bit 0 of PORTB.

(b) Write a C program to accomplish the above by inputting the comparator output via bit 0 of PORTB.

Solution

(a) In this example, an LM339 comparator is connected to the PIC18F4321 in order to control when the LED will be turned ON or OFF. In Figure 9.5, when Vx > Vy the comparator will output a one, and the PIC18F4321 will turn the LED ON or OFF depending on the switch status.

If Vx < Vy, then the comparator will output a zero and the LED will be turned OFF. In the program, the ADCON1 register is used to configure RB0 and RB1 as inputs. The TRISD is used to make RD0 of PORTD a output.

The PIC18F assembly language program for Polled I/O is provided below:

```
        INCLUDE <P18F4321.INC>
        ORG     0               ; RESET VECTOR
        BRA     MAIN            ; JUMP TO MAIN
        ORG     0x100
MAIN    BCF     TRISD, TRISD0   ; CONFIGURE BIT 0 OF PORTD AS OUTPUT
        MOVLW   0x0F            ; CONFIGURE BITS 0 AND 1 OF PORTB
        MOVWF   ADCON1          ; AS DIGITAL INPUTS
BEGIN   BCF     PORTD, RD0      ; TURN LED OFF
CHECK   BTFSS   PORTB, RB0      ; CHECK IF COMPARATOR OUTPUT IS ONE
        BRA     BEGIN           ; WAIT IN LOOP UNTIL ONE
        BTFSS   PORTB, RB1      ; CHECK IF SWITCH IS OPEN
        BRA     BEGIN
        BSF     PORTD, RD0      ; IF SWITCH IS OPEN, TURN LED ON
```

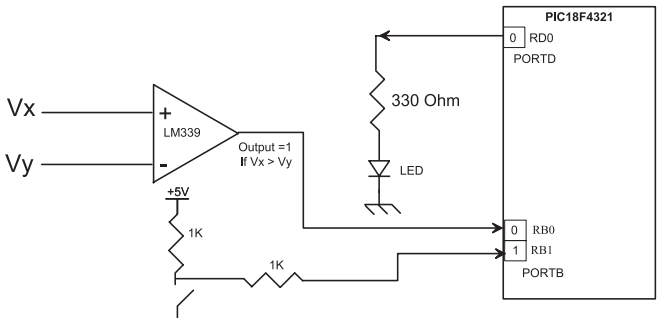


FIGURE 9. 5 Figure for Example 9.1

```
        BRA     CHECK
        END
```

In the above program, upon reset, the PIC18F starts executing the program at address 0x00 in program memory. After execution of the instruction BRA MAIN, the program will jump to address 0x100.

Next, PORTB and PORTD are configured. The instruction BCF PORTD, RD0 turns the LED OFF. In order to check whether the comparator is outputting a one, the instruction BTFSS PORTB, RB0 is used in the program. After execution of this instruction, if bit 0 of PORTB (comparator output) is 0, the next instruction, BRA BEGIN continues looping until the comparator outputs a one. However, if the comparator output is 1, the BTFSS PORTB, RB0 will skip the next instruction (BRA BEGIN).

(b) The C program is provided below:

```
#include <p18f4321.h>

void main (void)
{
TRISDbits.TRISD0 = 0;   // PORD is output
ADCON1 = 0x0F;          // Configure for PORTB to be digital input
PORTDbits.RD0 = 0;      // Turn LED OFF

while(1)
{
        PORTDbits.RD0 = 0;              // Turn LED OFF
        while(PORTBbits.RB0 = =1)       // While Vx > Vy
        {
                if(PORTBbits.RB1= =1)
                        PORTDbits.RD0 = 1;      // Turn LED ON
                else if (PORTBbits.RB1= = 0)
                        PORTDbits.RD0 = 0;      // Turn LED OFF
        }
}
}
```

In the above code, the register ADCON1 is used to configure PORTB. Within the infinite 'while' loop, the code checks to see when the comparator output is one indicating Vx > Vy. The LED is then turned ON or OFF based on the state of the switch.

As mentioned in Chapter 8, in order to implement an I/O program in PIC18F assembly such as Example 9.1(a) in the laboratory, the following configuration commands should be inserted using the assembler directive "config" in the PIC18F assembly language program after the "include <p18f4321.inc>" statement as follows:

```
include  <p18f4321.inc>
config  OSC  =  INTIO2   ; Select internal oscillator
config  WDT =  OFF       ; Watch Dog Timer OFF
config  LVP   =  OFF     ; Low Voltage Programming OFF
config  BOR  =   OFF     ; Brown Out Reset OFF
```

Similarly, in order to implement an I/O program using C such as Example 9.1(b) in the laboratory, the following configuration commands should be inserted in the C-program

using the directive "#pragma config" after #include <p18f4321.h> as follows:

```
#include <p18f4321.h>
#pragma config OSC = INTIO2 // Select internal oscillator
#pragma config WDT = OFF    // Watch Dog Timer OFF
#pragma config LVP = OFF    // Low Voltage Programming OFF
#pragma config BOR = OFF    // Brown Out Reset OFF
```

Note that the above applies to all programs in PIC18F assembly and C including programmed I/O, polled I/O, and interrupt I/O.

9.2 PIC18F Interrupts

The concept of interrupt is discussed in detail in the last section. PIC18F interrupts are basically divided into two types, namely external and internal interrupts. Figure 9.6 shows PIC18F interrupt structure.

External interrupts are initiated by external devices via the PIC18F's interrupt pins (INT0-INT2). *Internal interrupts,* on the other hand, are activated internally by on-chip peripheral devices (ADC, Hardware timers) for conditions such as interrupts due to "completion of on-chip A/D (Analog to Digital Converter) conversion" and "hardware timers reaching the programmed limits". These conditions are generated by PIC18F's on-chip peripheral devices.

9.2.1 Interrupt Procedure

Upon reset, the PIC18F operates in a mode called the "default" mode. In this mode, the PIC18F handles external interrupts using three external interrupt (INT0, INT1, INT2) pins. In this mode, interrupt priorities cannot be assigned. For simplicity, default interrupts will first be discussed. A detailed coverage of the PIC18F interrupts will then be explained.

When an interrupt is recognized after reset, the PIC18F completes execution of the current instruction, pushes the current contents of the program counter (address of the next instruction to be executed) onto the hardware stack automatically, and loads the program counter with an address (predefined by Microchip Technology) called the 'Interrupt address vector". The programmer writes a program called the "Interrupt Service Routine" at this address.

The RETFIE instruction is normally used at the end of the service routine. The PIC18F RETFIE 0 or simply RETFIE instruction pops the contents of the program counter previously pushed before going to the service routine, enables all interrupts, and returns control to the appropriate place in the main program.

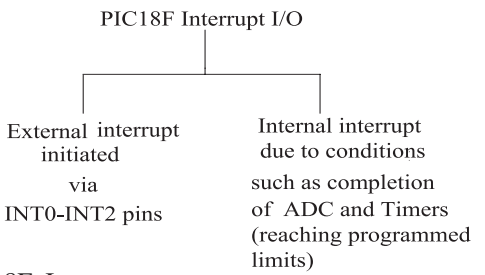


FIGURE 9.6 PIC18F Interrupts

(a) INTCON Register

7	6	5	4	3	2	1	0	
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0I F	INT0I F	RBIF	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

(b) INTCON3 Register

7	6	5	4	3	2	1	0	
INT2I P	INT1I P	---	INT2IE	INT1IE	---	INT2I F	INT1I F	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 interrupt
0 = Disables the INT2 interrupt
bit 3 **INT1IE:** INT1 External Interrupt Enable bit, 1 = Enables the INT1 interrupt
0 = Disables the INT1 interrupt
bit 2 **Unimplemented:** Read as '0'
bit 1 **INT2IF:** INT2 External Interrupt Flag bit, 1 = The INT2 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

FIGURE 9. 7 INTCON (Interrupt Control) and INTCON3 (Interrupt Control 3) registers

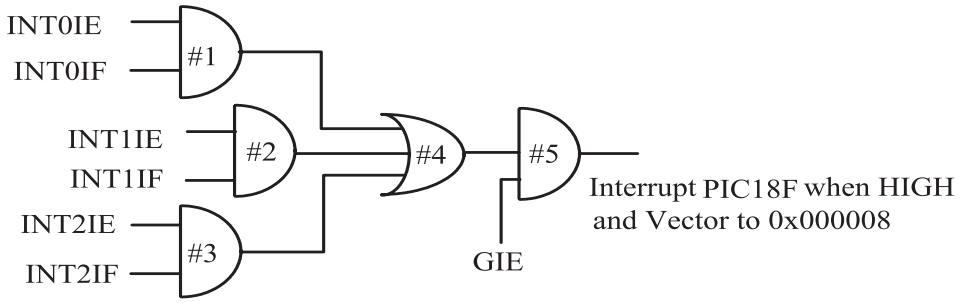


FIGURE 9.8 Simplified schematic for the PIC18F external interrupts for power-on reset

9.2.2 PIC18F Interrupt Types

The PIC18F4321 interrupts can be of two types. These are external interrupts initiated via PIC18F4321's interrupt pins, and internal interrupts initiated by internal peripheral devices such as on-chip ADC and on-chip timers. The PIC18F4321 is provided with three external interrupt pins. These pins are INT0, INT1, and INT2.

These interrupts are activated by the leading edge (LOW to HIGH) pulses in default mode. The PIC18F has the flexibility of changing the triggering levels to a trailing edge (HIGH to LOW) via programming a register (to be discussed later). For edge-triggered interrupts, in order for the PIC18F to recognize an interrupt such as INT0, the INT0 pin must be held LOW for at least two instruction cycles and HIGH for at least two instruction cycles. This means that if the internal clock of 1MHz is used, then each instruction cycle time is 4 microseconds. Hence, the PIC18F will recognize an interrupt if it is LOW for at least 8 microseconds (two instruction cycles) and HIGH for at least 8 microseconds (two instruction cycles).

The PIC18F4321 on-chip peripherals can generate internal interrupts. These peripheral interrupts will be covered later.

9.2.3 Programming the PIC18F External Interrupts

The concept of PIC18F external interrupts described in this section can be used to perform simple and meaningful experiments in laboratories.

As mentioned before, upon power-on reset, the PIC18F handles the three external interrupts (INT0, INT1, INT2) in a mode called the "default mode". The interrupt address vector (starting address of the service routine) for all three interrupts is the same address 0x00008 in the program memory. Each of the three external interrupts has an individual interrupt enable bit along with a corresponding flag bit located in a register. For example, the INT0 interrupt enable bit (INT0IE) and the INT0 interrupt flag bit (INT0IF) are located in a register called the INTCON register.

Each of the other two external interrupts (INT1 and INT2), on the other hand, have individual interrupt enable bits (INT1IE, INT2IE) along with the corresponding flag bits (INT1IF, INT2IF) in another register called the INTCON3 register. Figure 9.7 shows the INTCON and INTCON3 registers along with the associated interrupt bits. The other bits in these registers are specified for other functions such as timers, and will be discussed as these topics are covered.

Figure 9.8 depicts a simplified diagram showing the PIC18F external interrupt structure in default mode (branches to a single interrupt address vector 0x08). All PIC18F interrupts are disabled upon reset. However, before the PIC18F can accept external interrupts, certain initializations are necessary as follows (see Figure 9.8):

1. Clear INTxIF for each interrupt to 0. These flags will be set to ones automatically after occurance of interrupts. Note that INT0IF is located in the INTCON register while INT1IF and INT2IF are located in INTCON3 register.

2. Set INTxIE for each interrupt to 1. Note that INT0IE is located in the INTCON register while the INT1IE and INT2IE are contained in the INTCON3 register.

3. Set the GIE (Global Interrupt Enable) bit in the INTCON register to 1.

4. Finally, since the external interrupts (INT0- INT2) are multiplexed with analog inputs (AN8, AN10, and AN12), the ADCON1 register must be configured as digital input.

Next, consider INT0 in Figure 9.8. According to the steps listed above, the following initializations in PIC18F assembly language are necessary before the PIC18F will be able to accept external interrupt via its INT0 pin:

```
BCF     INTCON, INT0IF  ; Clear INT0IF to 0
BSF     INTCON, INT0IE  ; Set INT0IE to 1
BSF     INTCON, GIE     ; Set GIE to 1
MOVLW   0x0F            ; Configure INT0-INT2 as digital inputs
MOVWF   ADCON1
```

The C-equivalent of the above PIC18F assembly language will be:

```
INTCONbits.INT0IF = 0;      // Clear INT0IF to 0
INTCONbits. INT0IE = 1;     // Set INT0IE to 1
INTCONbits. GIE = 1;        // Set GIE to 1
ADCON1 = 0x0F;              // Configure INT0-INT2 as digital inputs
```

In Figure 9.8, the above PIC18F instruction sequence or C-statements will send a '0' to the output of AND gate #1. This is because INT0IF = 0, INT0IE = 1, and GIE = 1 ; note that INT1IE and INT2IE are zero after reset. This, in turn, will make the output of AND gate #5 as 0. As soon as the PIC18F is interrupted via the INT0 pin, the PIC18F completes the current instruction, pushes the current contents of the program counter (Return address) onto the stack, and then automatically sets the INT0IF to 1. Hence, the output of AND gate #5 will be '1' in Figure 9.8, and the PIC18F will load address 0x00008 into the program counter, and thus branches to the interrupt service routine.

The following PIC18F assembly instruction sequence will enable all external interrupts:

```
BCF     INTCON, INT0IF   ; INT0IF = 0
BCF     INTCON3, INT1IF  ; INT1IF = 0
BCF     INTCON3, INT2IF  ; INT2IF = 0
BSF     INTCON, INT0IE   ; Enable INT0
BSF     INTCON3, INT1IE  ; Enable INT1
BSF     INTCON3, INT2IE  ; Enable INT2
BSF     INTCON, GIE      ; Enable GIE
MOVLW   0X0F             ; Configure INT0-INT2
MOVWF   ADCON1           ; as digital inputs
```

The following C-language statements will enable all external interrupts:

```
INTCONbits.INT0IF=0;        // INT0IF = 0
INTCON3bits.INT1IF=0;       // INT1IF = 0
```

```
INTCON3bits.INT2IF=0;        // INT2IF = 0
INTCONbits.INT0IE=1;         // Enable INT0
INTCON3bits.INT1IE=1;        // Enable INT1
INTCON3bits.INT2IE=1;        // Enable INT2
INTCONbits. GIE = 1;         // Enable all inputs
ADCON1=0x0F;                 // Configure INT0-INT2 as digital inputs
```

The above is self-explanatory.

Note that before executing the RETFIE instruction (the last instruction of the interrupt service routine), the user must clear the INT0IF bit (assuming INT0 interrupt) using the PIC18F assembly instruction BCF INTCON, INT0IF to make sure that the INT0 interrupt is serviced only once, and not multiple number of times. The RETFIE instruction at the end of the service routine will automatically set the GIE bit (bit 7 in INTCON) to one enabling all interrupts, and will load the program counter from the hardware stack with the previously pushed value. Thus, the control is returned at the right place in the main program. Note that external interrupts INT1 and INT2 can be explained in a similar manner.

When an interrupt occurs and is recognized by the PIC18F4321, the PIC18F will automatically jump to interrupt address vector 0x08. Since program memory address is 0x18 assigned to the Low priority interrupt, there may not be enough locations at address 0x08 for large service routines. Hence, the PIC18F assembly language instruction GOTO should be used at address 0x08 to jump to another address for the interrupt service routine where adequate locations are available. The C18 compiler does not have an equivalent of the "GOTO" instruction. Hence, for a service routine called, "ISR", the PIC18F assembly statement, "GOTO ISR" must be inserted at address 0x08 (interrupt address vector in default mode) using the C18 directives _asm and _endasm. The first statement of the service routine should be #pragma interrupt ISR**.** The program will go to the service routine called ISR.

The PIC18F4321 contains ten registers which are used to control interrupt operation. These registers are:

- RCON (Figure 8.6)
- INTCON (Figure 9.7(a))
- INTCON2 (9.9)
- INTCON3 (Figure 9.7(b))
- PIR1, PIR2 (to be discussed in Chapter 10, Figures 10.8, 10.12)
- PIE1, PIE2 (to be discussed in Chapter 10, Figures 10.9, 10.13)

Registers RCON, INTCON, INTCON2, and INTCON3 are associated with external and port change interrupts. Hence, they are covered in this section. Registers PIR1, PIR2, PIE1 and PIE2 are used by peripheral interrupts. Hence, the functions of these registers will be discussed when these topics are covered in Chapter 10.

The PIC18F4321 interrupts can be classified into two groups: high priority interrupt levels and low priority interrupt levels. The high priority interrupt vector is at address 0x000008 and the low priority interrupt vector is at address 0x000018 in the program memory. High priority interrupt events will interrupt any low priority interrupts that may be in progress.

As mentioned before, upon power-on reset, the interrupt address vector is 0x000008 (default), and no interrupt priorities are available. Also, upon power-on reset, IPEN is automatically cleared to 0, and the PIC18F operates as a high-priority interrupt (single interrupt) system. Hence, the interrupt vector address is 0x000008. The IPEN bit (bit 7 of the RCON register) of the RCON register in Figure 8.6 can be programmed to assign interrupt priorities. During normal operation, the IPEN bit can be set to one by executing the "BSF RCON, IPEN" in PIC18F assembly or "RCONbits.IPEN = 1;" in C-language to assign priorities in the system.

When interrupt priority is enabled (IPEN = 1), there are two bits which enable interrupts globally. Setting the GIEH bit (bit 7 of INTCON register of Figure 9.7(a)) enables all interrupts that have the priority bit set (high priority). Setting the GIEL bit (bit 6 of INTCON register of Figure 9.7(a)) enables all low interrupts. When the interrupt flag, interrupt enable bit and appropriate global interrupt enable bit are set, the interrupt will vector immediately to address 0x000008 or 0x000018, depending on the priority bit setting. Individual interrupts can be disabled through their corresponding enable (INTxIE) bits.

From Figure 9.7(a), when IPEN of RCON register is 0, bits 6 and 7 of the INTCON register become GIE (Global Interrupt Enable) bit and PEIE (Peripheral Interupt Enable) bit. Setting or cleaning the GIE bit via programming will enable or disable all interrupts (INT0, INT1, INT2). On the other hand, setting or clearing the PEIE bit will enable or disable all peripheral interrupts.

Table 9.1 shows the three external interrupts of the PIC18F4321 along with the corresponding Interrupt Priority (IP) bit.

As shown in the table, since an IP (Interrupt Priority) is not assigned to INT0, it always has the high priority level. However, INT1 and INT2 can be programmed as high or low level priorities. When INT1 and INT2 are used as two interrupts in an application; INT0 is not used. For example, in order to program INT1 as a high priority interrupt and INT2 as a low priority interrupt, the following PIC18F assembly language instruction sequence can be used:

7	6	5	4	3	2	1	0	
$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	INTEDG0	-------	TMR0IP	-----	RBIP	INTCON2

bit 7 **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

FIGURE 9. 9 INTCON2 Register

TABLE 9.1 PIC18F4321 External interrupts along with Interrupt Priority (IP) bits

Interrupt Name	Interrupt Priority (IP) bit	Comment
INT0	Unassigned	Since no interrupt priority bit is assigned, INT0 always has the high priority level.
INT1	INT1IP	Bit 6 of INTCON3 register (Figure 9.7(b)); 1 = high priority, 0 = low priority
INT2	INT2IP	Bit 7 of INTCON3 register (Figure 9.7(b)); 1 = high priority, 0 = low priority

```
BSF     RCON, IPEN      ; Set IPEN to 1, enable interrupt level
BSF     INTCON, GIEL    ; Set low priority levels
BSF     INTCON, GIEH    ; Set high priority levels
BSF     INTCON3, INT1IP ; INT1 has high level
BCF     INTCON3, INT2IP ; INT2 has low level
```

The equivalent C-statements will be as follows:

```
RCONbits.IPEN = 1;          // Set IPEN to 1, enable interrupt level
INTCONbits.GIEL = 1;        // Set low priority levels
INTCONbits.GIEH = 1;        // Set high priority levels
INTCON3bits.INT1IP = 1;     // INT1 has high level
INTCON3bits.INT2IP = 0;     // INT2 has low level
```

Note that INT0 along with either INT1 or INT2 can be used as two interrupts in an application. Suppose INT0 and INT1 are used in a two-interrupt systgem. Since INT0 has always the high priority, the INT1 must be configured as the low priority. The following PIC18F assembly language instruction sequence will accomplish this:

```
BSF     RCON, IPEN      ; Enable priority interrupt
BSF     INTCON, GIEL    ; Enable global low interrupt
BSF     INTCON, GIEH    ; Enable global high interrupt
BCF     INTCON3, INT1IP ; Configure INT1 as low priority
```

The equivalent C-statements are as follows:

```
RCONbits.IPEN = 1;          // Enable priority interrupt
INTCONbits.GIEL = 1;        // Enable global low interrupt
INTCONbits.GIEIH = 1;       // Enable global high interrupt
INTCON3bits.INT1IP = 0;     // Configure INT1 as low priority
```

Finally, the "PORTB Interrupt-on-Change" is used in keyboard interfacing. The PIC18F4321 provides four interrupt-on-change pins (KB10 through KB13). These pins are multiplexed with bits RB4 through RB7 of PORTB. An input change (HIGH to LOW or LOW to HIGH) on one or more of these KB10-KB13 pins will set a single flag bit, RBIF (bit 0 of INTCON register) to one. Note that a single flag bit is assigned to all four interrupts.

The interrupt can be enabled/disabled by setting/clearing the single enable bit, RBIE (bit 3 of INTCON register). Interrupt priority for PORTB interrupt-on-change is determined by the value contained in the interrupt priority bit RBIP (bit 0 of INTCON2 register of Figure 9.9). Each bit associated with this interrupt can be set or cleared as appropriate via programming the corresponding register. A common application of this interrupt is to detect a keyboard actuation while interfacing a keyboard to the PIC18F.

9.2.4 Programming PIC18F external interrupts using assembly

In this section, topics such as the setting of interrupt triggering levels and other features associated with the PIC18F using assembly language will be covered.

Setting the triggering levels of INTn Pin Interrupts External interrupts on the INT0, INT1, and INT2 pins are edge-triggered. Upon power-on reset, each of these external interrupts (INT0, INT1, INT2) are activated by a rising edge pulse (LOW to HIGH). The PIC18F has the flexibility of changing the triggering levels for these interrupts to falling edge pulses (HIGH to LOW).

This can be accomplished by programming bits 4 through 6 of the INTCON2 register (Figure 9.9). For example, the instruction BCF INTCON2, INTEDG0 will change INT0 from a rising-edge triggered to falling-edge triggered interrupt.

Return from Interrupt Instruction The "RETFIE s" instruction is normally used at the end of the service routine. 's' can be 0 or 1. When s = 0, the PIC18F RETFIE 0 or simply RETFIE instruction pops the contents of the program counter previously pushed before going to the service routine, enables the global interrupt enable bit, and returns control to the appropriate place in the main program. For high priority interrupts only eight 16-bit words from addresses 0x000008 through 0x000017 are available for the service routine since the address 0x000018 is assigned to the low priority interrupt. This means that a small service routine can be written starting at address 0x08. However, for large service routines, the programmer may have to write a service routine by jumping to another location using the GOTO instruction.

9.2.5 Programming PIC18F external interrupts using C

MPLAB C18 compiler does not automatically place an interrupt service routine at user-defined interrupt address vector. Hence, the GOTO instruction is normally placed at the interrupt address vector 0x08 for transferring control to the interrupt service routine. As mentioned before, the C18 compiler does not have any feature to jump from one address to another. However, the C18 compiler has "_asm" and "_endasm" directives to place "GOTO" instruction at address 0x08 to jump to the interrupt service routine.

The PIC18F4321 provides four interrupt-on-change pins (KB10 through KB13). These pins are multiplexed with bits 4 through 7 of PORTB. An input change (HIGH to LOW or LOW to HIGH) on one or more of these interrupts sets a single flag bit, RBIF (bit 0 of INTCON register). Note that a single bit is assigned to all four interrupts. The interrupt can be enabled/disabled by setting/clearing a single enable bit, RBIE (bit 3 of INTCON register). Interrupt priority for PORTB interrupt-on-change is determined by the value contained in the interrupt priority bit, RBIP (bit 0 of INTCON2 register). This interrupt is typically used with keyboard interfacing. Further discussion on this topic is beyond the scope of this book.

Next, certain features associated with the C18 compiler will be discussed.

Specifying starting address of the service routine using the C18 compiler As mentioned before, using the MPLAB assembler, the programmer uses the ORG directive to specify the starting address of the service routine (interupt, address vector). Using the C18 compiler, the interrupt address vectors, must be defined and placed at the appropriate vector addresses using the "#pragma code" directive. For example, the programmer can use the statement "#pragma code begin = 0x08" to place begin at an address 0x08. Hence, the C statement "#pragma code int_vect = 0x08" will assign the address 0x08 to label "int_vect".

Assigning interrupt priorities using the C18 compiler The C18 compiler uses the directives "interrupt" and "interruptlow" to specify high and low priority interrupt levels. The programmers can use these directives which allow a program to branch automatically from the respective interrupt address vector to a different program to find the source of the interrupt (for multiple interrupts with priorities), and then to the appropriate service routine.

As mentioned before, the PIC18F supports up to two levels of interrupt priority. Most interrupts have a priority bit associated with the interrupt flag enable bits that assigns it to one of the two priority levels. Use of priority level is optional, and the PIC18F may be configured to use only one level priority. When two levels of interrupt priority are used, an interrupt of either priority level may interrupt the main program. However, only a High Priority Interrupt may interrupt a Low Priority Interrupt, and nothing may interrupt a High Priority Interrupt.

If only a single level of interrupts were used (IPEN bit in RCON register = 0), then it is only necessary to define the interrupt vector 0x00008 and a single interrupt service routine function with the directive "#pragma interrupt". All priority bit settings are ignored. The functions of GIEH and GIEL bits in the INTCON register become GIE (Global Interrupt Enable) and PEIE (Peripheral Interrupt Enable) bits, with the following functions:

INTCONbits.GIE	INTCONbits.PEIE	Interrupt Functions
0	0	No interrupts;all interrupts disabled since GIE = 0.
0	1	No interrupts;all interrupts disabled since GIE = 0.
1	0	Only interrupts in INTCONx enabled; all peripheral interrupts remain disabled
1	1	All interrupts including peripheral interrupts are enabled

A typical structure for interrupt programs using C The default interrupt INT0 with vector address 0x00008 is used in the following to illustrate the interrupt programs using C. Typical structures for the main program and the service routine (when the service routine requires more than eight words) are provided below:

```
#include  <P18F4321.h>
void       ISR (void);
#pragma code Int=0x08    // At interrupt code jumps here
void  Int(void)
{
_asm                      // Using assembly language
GOTO ISR
_endasm
}
#pragma code              //  Main program
void main( )
{
// Typically configure ports, enable INT0IE, clear INT0IF
while(1){                 // Wait in an infinite loop for the interrupt to occur
}
```

```
#pragma   interrupt  ISR
void ISR(void)              // Start of Interrupt service routine
        {
INTCONbits.INT0IF =0;  // clear INT0IF, and then write the service
        }
}
```

In the above, the "#pragma code" directive will place "Int" at the specific location 0x08 in program memory. The main program typically configures ports, enables interrupt, clears the interrupt flag bit and waits in an infinite loop for the interrupt to occur. When the interrupt occurs and is recognized by the PIC18F4321, the program will automatically jump to memory location 0x08. The PIC18F assembly language instruction GOTO must be used to jump to the interrupt service routine. The GOTO ISR can be used for this purpose. Also, note that the C18 interrupt keyword inserts the RETFIE instruction at the end of the service routine.

Example 9.2 ***Interrupt I/O*** Assume that the PIC18F4321 microcontroller shown in Figure 9.10 is required to perform the following:

If Vx > Vy , turn the LED ON if the switch is open; otherwise, turn the LED OFF.

(a) Write a PIC18F assembly language program to accomplish this using INT0 external interrupt. Use port A for the LED and switch as above. Write the main program at address 0x100 in PIC18F assembly language and the service routine at address 0x08. Connect INT0 pin to the output of the comparator.

The main program will initialize hardware stack pointer (STKPTR) to 0x10, configure PORTB and PORTD, clear INT0IF and enable GIE and INT0IE. Write the service routine in PIC18F assembly language which will clear the INT0F flag, input the switch, output to LED, and then return to the main program.

(b) Assume that the PIC18F4321 microcontroller shown in Figure 9.10 is required to perform the following:

If Vx > Vy , turn the LED ON if the switch is open; otherwise, turn the LED OFF. Write the main program in C language which will initialize PORTB and PORTD, and then wait for interrupt in an infinite loop. Also, write a service routine in C language.

Solution

(a) Figure 9.10 shows the relevant connections of the comparator to the PIC18F4321 using interrupt I/O. Note that the comparator output is connected to bit 0 of PORTB to be used as an INT0 pin.

In this example, by using ADCON1 register, bit 0 of PORTB can be configured as digital input to accept interrupt via INT0. The INT0IE bit of the INTCON register must be set to one in order to enable the external interrupt along with GIE to enable global interrupts.

The PIC18F assembly language program using external interrupt INT0 is provided in the following:

```
INCLUDE  <P18F4321.INC>
ORG      0                  ; RESET
BRA      MAIN_PROG
```

```
; MAIN PROGRAM
                ORG     0x00100         ; MAIN PROGRAM
MAIN_PROG       MOVLW   0x10            ; Initialize STKPTR to 0x10
                MOVWF   STKPTR
                BCF     TRISD,TRISD1    ; Configure bit 1 OF PORTD as output
                MOVLW   0x0F            ; Configure bit 0 of PORTB as INT0
                MOVWF   ADCON1          ; and bit 1 as input
                BCF     INTCON, INT0IF  ; Clear INT0IF flag
                BSF     INTCON, GIE     ; Enable global interrupts
                BSF     INTCON, INT0IE  ; Enable the external interrupt
WAIT            BRA     WAIT            ; Wait for interrupt
                BRA     MAIN_PROG       ; Repeat
; INTERRUPT SERVICE ROUTINE
                ORG     0x000008        ; Interrupt Address Vector
INT_SERV        MOVFF   PORTB, PORTD    ; Output switch status to turn LED ON/OF
                BCF     INTCON, INT0IF  ; Clear the external interrupt flag to avoid
                                        ; double interrupt
                RETFIE                  ; Enable interrupt and return
                END
```

(b) Using C:

In this example, an LM339 comparator is interfaced with the PIC18F4321 using C and external interrupts. An external interrupts allows the microcontroller to trigger an interrupt from a source outside the PIC18F4321 such as a comparator. As with the previous interrupt example, the code starts with the #pragma command which will place code fragments at specific locations in memory; when the interrupt is triggered, the microcontroller will automatically jump to memory location 0x000008 and finally to COMP_ISR.

In the main program, PORTD is configured as an output, and PORTB is configured as a digital input. The external interrupt flag and the global interrupt enable are enabled. The main program then waits in an infinite 'while' loop that turns the LED OFF until the comparator output is HIGH, interrupting the microcontroller. After recognizing the interrupt, the code will automatically jump to address 0x000008, and then jump to the service routine at COMP_ISR via the code at COMP_Int. Within the service routine, the code will continue to take the switch data from PORTB and output the data to the LED via PORTD. It will continue to do this as long as the comparator output stays HIGH. When

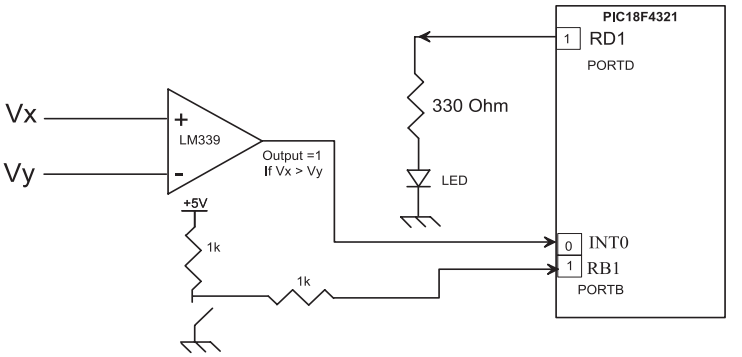


FIGURE 9.10 Figure for Example 9.2

Vx is lower than Vy, the comparator will output 0 and the code will return to the infinite 'while' loop and turn the LED OFF.
The PIC18F program using C is provided below:

```
#include <P18F4321.h>
void COMP_ISR (void);
#pragma code COMP_Int=0x08  //At interrupt code jumps here
void COMP_Int(void)
{
_asm              // Using assembly language
GOTO  COMP_ISR
_endasm
}
#pragma code
void main( )                                  // start of the main program
{
TRISD=0x00;                                   // PORTD is output
ADCON1=0x0F;                                  // Configure RB1 and INT0 as digital input
INTCONbits.INT0IE=1;                          // Enable external interrupt
INTCONbits.INT0IF=0;                          // Clear the external interrupt flag
INTCONbits.GIE=1;                             // Enable global interrupts
PORTD=0;                                        // Turn off LED;
while(1){   // wait in an infinite loop for the interrupt to occur
       PORTD=0;                               // LED is off
       }
}
#   pragma  interrupt COMP_ISR
void COMP_ISR(void)  //start of the Comparator interrupt service routine
       {
       INTCONbits.INT0IF=0;                   // Clear external interrupt flag
             while(PORTBbits.RB0= =1){ // Check if comparator is high
                   PORTD = PORTB;           // Move PORTB into PORTD
       }
}
```

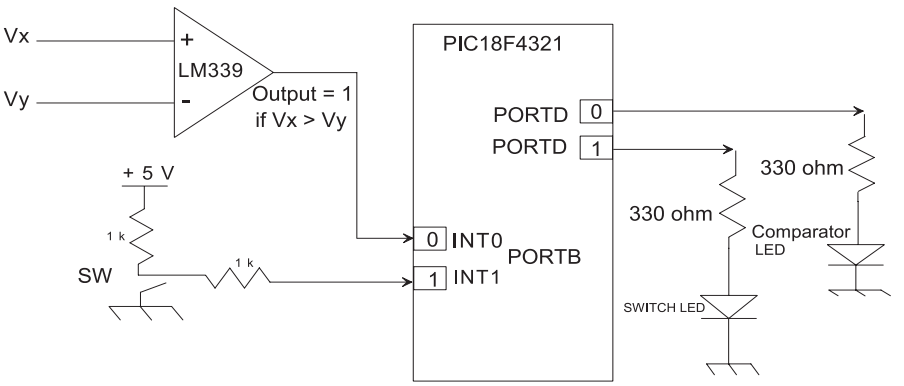


FIGURE 9.11 Figure for Example 9.3

Example 9.3 In Figure 9.11, if Vx > Vy, the PIC18F4321 is interrupted via INT0. On the other hand, opening the switch will interrupt the microcontroller via INT1. Note that in the PIC18F4321, INT0 has the higher priority than INT1.

(a) Write the main program in PIC18F assembly language at address 0x100 that will perform the following:

- Initialize STKPTR to 0x12
- Configure PORTB as interrupt inputs.
- Clear interrupt flag bits of INT0IF in INTCON reg and INT1IF in INTCON3 reg
- Configure PORTD (bits 0, 1) as outputs for LEDs
- Set INT1 as a low priority interrupt by clearing INT1IP bit in INTCON3 reg
- Enable priority interrupts by setting IPEN bit in RCON reg
- Enable global HIGH and LOW interrupts by setting GIEH and GIEL bits in INTCON reg
- Turn both LEDs at PORTD OFF.
- Wait in an infinite loop, and wait for one or both interrupts to occur.
- Turn both LEDs at PORTD OFF (Comparator LED at bit 0 of PORTD and Switch LED at bit 1 of PORTD)
- Wait in an infinite loop for one or both interrupts to occur.

Also, write a service routine in PIC18F assembly language at address 0x200 for the high priority interrupt (INT0) that will perform the following:
- Clear interrupt flag for INT0 (INT0IF)
- Check to see if the comparator output is still 1. If it is, turn the LED ON at bit 0 of PORTD**.**

Finally, write a service routine in PIC18F assembly language at address 0x300 for the low priority interrupt (INT1) that will perform the following:
- Clear interrupt flag for INT1 (INT1IF)
- Turn LED ON at bit 1 of PORTD

(b) In Figure 9.11, if Vx > Vy, the PIC18F4321 is interrupted via INT0. On the other hand, opening the switch will interrupt the microcontroller via INT1. Note that in the PIC18F4321, INT0 has the higher priority than INT1. Write the main program in C that will perform the following:

- Configure PORTB as interrupt inputs.
- Clear interrupt flag bits of INT0IF in INTCON reg and INT1IF in INTCON3 reg
- Configure PORTD (bits 0, 1) as outputs for LEDs
- Set INT1 as low priority interrupt by clearing INT1IP bit in INTCON3 reg
- Enable priority interrupts by setting IPEN bit in RCON reg
- Enable global HIGH and LOW interrupts by setting GIEH and GIEL bits in INTCON reg
- Turn both LEDs at PORTD OFF.
- Wait in an infinite loop, and wait for one or both interrupts to occur

Also, write a service routine in C-language for the high priority interrupt (INT0) that will check to see if the comparator output is still 1. If it is, turn LED at bit 0 of PORTD ON**.** If the comparator output is 0, return.

Finally, write a service routine in C language for the low priority interrupt (INT1) that will perform the following:
- Clear interrupt flag for INT1
- Check to see if the switch is still 1. If it is, turn LED at bit 1 of PORTD ON**.** If the switch input is 0, return.

Solution

This example will demonstrate the interrupt priority scheme of the PIC18F4321 microcontroller. With interrupt priority, the user has the option to have the interrupts declared as either low or high interrupts. If at anytime the low and high priority interrupts occur at the same time, the microcontroller will always service the high priority interrupt. In the above example, the comparator is set as the high priority and the switch is set as the low priority, so if both interrupts are triggered simultaneously, then only the LED associated with the comparator will be turned ON.

Note that the external interrupt INT0 can only be a high priority interrupt. When implementing a single interrupt, the interrupt service routine is written at address 0x08. On the other hand, when priority interrupts are enabled, the service routine for the high priority interrupt is written at address 0x08 while the service routine for the low priority interrupt is written at address 0x018.

In order to enable the second external interrupt, INT1, the register INTCON3 is configured. Also, INT1IE must be enabled, and INT1IF must be cleared to 0. Furthermore, the INT1IP bit in INTCON3 register that sets the priority of INT1 in the INTCON3 register must be cleared to 0 for low priority. Next, the IPEN bit in the RCON register that enables the interrupt priority functionality of the PIC18F4321 must be set to one. Finally, the GIEH and GIEL bits in the INTCON register must be set to one in order to enable global high and low interrupts.

(a) PIC18F assembly language program:

```
INCLUDE <P18F4321.INC>
; RESET
                ORG     0               ; Reset vector
                BRA     MAIN            ; Jump to main program
; HIGH PRIORITY INTERRUPT ADDRESS VECTOR
                ORG     0x0008          ; High priority interrupt
                BRA     HIGH_INT_ISR    ; Jump to service routine for the
                                        ; comparator
; LOW PRIORITY INTERRUPT ADDRESS VECTOR
                ORG     0x0018          ; Low priority interrupt
                BRA     LOW_INT_ISR     ; Jump to service routine for the
                                        ; comparator
                                        ; MAIN PROGRAM
                ORG     0x0100
MAIN            MOVLW   0x12            ; Initialize STKPTR to 0x12
                MOVWF   STKPTR
                CLRF    TRISD           ; PORTD is output
                MOVLW   0x0F            ; Configure ADCON1 to set up
                MOVWF   ADCON1          ; INT0 and INT1 as digital inputs
                BSF     INTCON, INT0IE  ; Enable the external interrupt INT0
                BSF     INTCON3,INT1IE  ; Enable the external interrupt INT1
                BCF     INTCON,INT0IF   ; Clear the INT0IF flag
                BCF     INTCON3,INT1IF  ; Clear the INT1IF flag
                BCF     INTCON3, INT1IP ; Configure INT1 as low priority
                BSF     RCON, IPEN      ; Enable priority interrupt
```

```
                BSF     INTCON, GIEH    ; Enable global high interrupts
                BSF     INTCON, GIEL    ; Enable global low interrupts
OVER            CLRF    PORTD           ; Turn both LED's  off
                BRA     OVER            ; Wait for interrupt

; SERVICE ROUTINE FOR HIGH PRIORITY
                ORG     0x200
HIGH_INT_ISR    BCF     INTCON,INT0IF   ; Clear the interrupt flag
CHECK           BTFSS   PORTB,RB0       ; Check to see if comparator output is
                                        ; still one
                RETFIE
                MOVLW   0x01            ; Turn on LED at bit 0 of PORTD
                MOVWF   PORTD
                BRA     CHECK
; SERVICE ROUTINE FOR LOW PRIORITY
                ORG     0x300
LOW_INT_ISR     BCF     INTCON3, INT1IF ; Clear the interrupt flag
CHECK1          BTFSS   PORTB,RB1       ; Check to see if switch is still one
                RETFIE
                MOVLW   0x02            ; Turn on LED at bit 1 of PORTD
                MOVWF   PORTD
                BRA     CHECK1
                END
```

(b) C-program

Implementing the above design, this example demonstrates the interrupt priority system of the PIC18F microcontroller. Using interrupt priority, the user has the option to have various interrupts assigned as either low priority or high priority interrupts. If a low priority interrupt and a high priority interrupt occur at the same time, the PIC18F will always service the high priority interrupt first.

In the above example, the high priority is assigned to the comparator while the switch is assigned with the low priority. Hence, if both interrupts were triggered at the same time, then the LED associated with the comparator would be turned ON first, and then the LED associated with the switch will be turned ON. Note that the external interrupt INT0 can only be a high priority interrupt. Hence, INT0 is connected to the comparator output while the switch is connected to INT1 since it has the low priority. At the end of the code provided below, it can be seen that there are two interrupt service routines, HP_COMP_ISR and LP_SWITCH_ISR, which are the high priority and low priority service routines.

The following code implements priority interrupts on the PIC18F using C-language:

```
#include <P18F4321.h>
void HP_COMP_ISR (void);
void LP_SWITCH_ISR(void);

#pragma code High_Priority_COMP_Int=0x08  //High interrupt code jumps here
void COMP_Int(void)
{
_asm  //Using assembly language
GOTO  HP_COMP_ISR  ( )
```

```
_endasm
}
#pragma code Low_Priority_SWITCH_Int=0x018  //Low interrupt code jumps here
void Switch_Int(void)
{
_asm  //Using assembly language
GOTO  LP_SWITCH_ISR ( )
_endasm
}
#pragma code
void main( )
{
TRISBbits.TRISB0=1;                    // Set bit 0 of PORTB as input
TRISBbits.TRISB1=1;                    // Set bit 1 of PORTB as input
TRISD=0x00;                            // PORTD is output
ADCON1=0x0F;                           // Configure PORTB to be digital input
INTCONbits.INT0IE=1;                   // Enable external interrupt INT0
INTCON3bits.INT1IE=1;                  // Enable external interrupt INT1
INTCONbits.INT0IF=0;                   // Clear INT0IF  interrupt flag bit for INT0
INTCON3bits.INT1IF=0;                  // Clear INT1IF interrupt flag bit for INT1
INTCON3bits.INT1IP=0;                  // Configure INT1 as low priority interrupt
RCONbits.IPEN=1;                       // Enable priority interrupts
INTCONbits.GIEH=1;                     // Enable global high priority interrupts
INTCONbits.GIEL=1;                     // Enable global low priority interrupts
PORTD=0;                               // Turn off LED;

while(1){
        PORTD=0;                       // LED is off
}

}                                      // End of main
#pragma  interrupt  HP_COMP_ISR
void HP_COMP_ISR(void){                // High priority interrupt service
        INTCONbits.INT0IF=0;           // Clear external interrupt flag
        while(PORTBbits.RB0==1)        // Check if comparator is high
{
              PORTD=0x01;              // Turn on LED
        }
}                                      // End of HP_COMP_ISR

#pragma  interruptlow  LP_SWITCH_ISR
void LP_SWITCH_ISR(void){              // Low priority interrupt service
        INTCON3bits.INT1IF=0;          // Clear external interrupt flag
        while(PORTBbits.RB1==1)        // Check if switch is still on
        {
        PORTD=0x02;                    // Turn on LED
        }

}                                      // End of LP_SWITCH_ISR
```

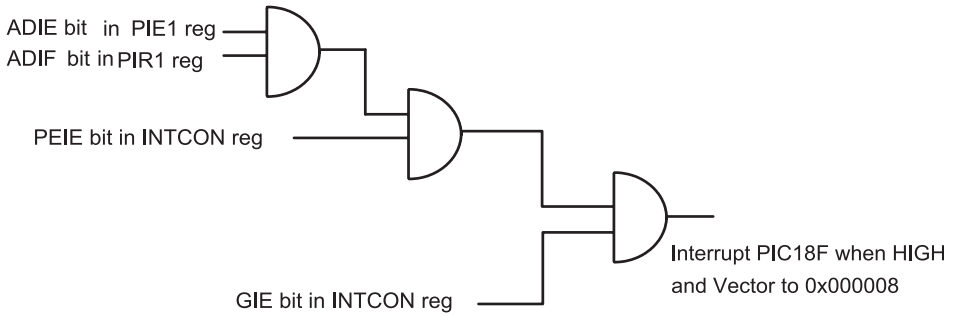


FIGURE 9.12 Interrupt-driven on-chip Peripheral device such as the ADC

9.2.6 Accessing PIC18F on-chip peripheral devices using Polled I/O vs. Interrupt I/O

There are several PIC18F on-chip peripheral I/O devices. Some of these on-chip peripherals include ADC (Analog to Digital) converter and Hardware timers. Each peripheral device is provided with a Peripheral Interrupt Flag such as the ADIF bit for the ADC. For polled I/O, this flag bit (ADIF for ADC) can be cleared to 0 first via programming. The PIC18F can then be programmed to start the ADC. Upon completion of the A/D conversion, the PIC18F automatically sets the ADIF bit to 1. Using polled I/O, the PIC18F can be programmed to wait in a loop for the ADIF to go to HIGH. For interrupt I/O, the PIC18F is provided with a few more peripheral interrupt bits in addition to ADIF and GIE bits. These bits are contained in certain PIC18F registers. Figure 9.12 shows a simplified diagram of a typical peripheral device such as the ADC along with these bits. As with polled I/O, the ADIF bit in the PIR1 in Figure 9.12 can be first cleared to 0. The ADIE, PEIE and GIE bits can then be set to ones by programming. Once the ADC is started by the PIC18F via programming, the ADIF is set to one indicating completion of A/D conversion. This will interrupt the PIC18F (Figure 9.12). The PIC18F will jump to interrupt service routine to input the ADC output.

The concept of polled and interrupt I/O associated with other on-chip peripheral devices can similarly be explained.

9.3 PIC18F Interface to a typical LCD (Liquid Crystal Display)

The seven-segment LEDs are easy to use, and can only display numbers and limited characters. An LCD is very useful for displaying numbers, and several ASCII characters along with graphics. Furthermore, the LCD consumes low power. Because of inexpensive price of the LCD these days, they have been becoming popular. The LCD's are widely used in copiers, fax machines, laser printers, networking equipment such as routers, and storage devices.

Figure 9.13 shows a pictorial view of Hitachi HD44780 LCD along with the pins. The Hitachi HD44780 is a dot matrix LCD manufactoured in several different configurations. Common sizes can display up to one 8-character line or two 8-character lines on a display screen. The HD 44780 contains 16 pins (Figure 9.13) which will be described in the following.

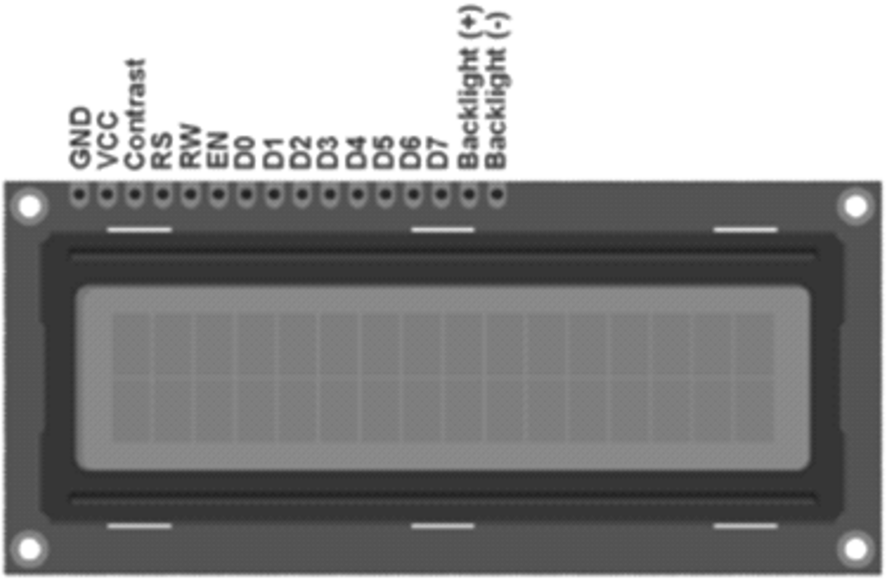


FIGURE 9.13 Hitachi HD44780 LCD and the pinout

-Pin 1 (GND): Connect to ground

-Pin 2 (VCC): Connect to +5V

-Pin 3 (Contrast): Connect to the middle terminal of a potentiometer (value between 2k and 6k); the other two terminals of the potentiometer should be connected to +5V and ground. The contrast of the LCD is changed by varying the value of the potentiometer. The potentiometer values vary from one LCD to another. In some cases, a potentiometer with a range from 10k to 20k is needed.

-Pin 4 (RS- Register Select): RS = 0 means commands while RS = 1 indicates data to be displayed on the LCD screen.

-Pin 5 (RW-Read Write): RW = 1 for reading data from the LCD registers while RW = 0 for writing data into the LCD registers.

-Pin 6 (EN-Enable): 8-bit data or command is latched on the data pins (D0-D7) when EN goes from HIGH to LOW for at least 450 nsec.

-Pins 7-14 (D0-D7): Eight data pins

-Pin 15 (Backlight+), Pin 16 (Backlight-): Connect Backlight+ to 5V and Backlight- to the ground. The HD44780 is a character LCD, and displays texts only. This display is provided with LED backlight. The nominal operating voltage for LED backlights is 5V at full brightness.

The HD44780 is an ASCII character-based alphanumeric LCD display and provides text displays along with LED backlight. It contains 16 pins. The VCC pin is connsected to +5V and the VSS pin is connected to ground. The contrast pin is used to control the brightness of the display. The contrast pin is connected to a potentiometer with a value between 2k and 6k. The eight data pins (D0-D7) are used to input data and commands for displaying the desired message on the screen.

The three control pins, EN, R/$\overline{W}$, and RS, allow the user to let the display know what kind of data is sent. The EN pin latches the data from the D0-D7 pins into the LCD display. Data on D0-D7 pins will be latched on the trailing edge (high-to-low) of the EN pulse. The EN pulse must be at least 450 ns wide; no delay routine is needed because instructions to send a '1' and then a '0' will accomplish this. The R/$\overline{W}$ (read/write) pin, allows the user to either write to the LCD or read data from the LCD. In this example, the R/$\overline{W}$ pin will always be zero since only a string of data is written to the LCD. The R/$\overline{W}$ pin is set to one for reading data from the LCD.

The command or data can be outputted to the LCD in two ways. One way is to provide time delays for a few milliseconds (10ms is used in Example 9.4) before outputting the next command or data. The second approach utilizes a busy flag in a register internal to the LCD to determine whether the LCD is free for the next data or command. For example, in order to display characters one at a time, the LCD must be read by outputting a HIGH on the R/$\overline{W}$ pin. The busy flag can be checked to ensure whether the LCD is busy or not before outputting another string of data. Note that the busy flag can thus be used instead of using time delays.

Finally, the RS (Register Select) pin is used to determine whether the user is sending command or data. The LCD contains two 8-bit internal registers. They are command register and data register. When RS = 0, the command register is accessed, and typical LCD commands such as clear cursor left (hex code 0x04) can be used. Table 9.2 shows a list of some of the commands. Note that the busy flag is bit 7 of the LCD's command register. The busy bit can be read by outputting 0 to RS pin, 1 to R/$\overline{W}$ pin, and a leading edge (LOW to HIGH) pulse to the EN pin.

When attempting to send data or commands to the LCD, the user must make sure that the values of EN, R/$\overline{W}$, and RS are correct along with appropriate timing. For example, in order to send the 8-bit command code to the LCD, a PIC18F assembly or a C- language program can be written to perform the following steps:
-output the command value to the PIC18F4321 I/O port that is connected to the LCD's D0- D7 pins.
- Send a '0' to RS pin and a '0' to R/$\overline{W}$ pin.
- Send a '1' and then a '0' to the EN pin to latch the LCD's D0-D7 code.
For sending data to the LCD, the PIC18F assembly or C-program can be written to perform the following steps:
- Ouput data from PORTD of the PIC18F4321 to LCD's D0-D7 pins.
- Send a '1' to RS pin and a '0' to the R/$\overline{W}$ pin.
- Send a '1' and then a '0' to the EN pin to latch the data.

EXAMPLE 9.4 Figure 9.14 shows the PIC18F4321's interface to the Hitachi HD44780 LCD display. It is desired to display the phrase "Switch Value:" along with the numeric BCD value (0 through 9) of the four switch inputs. Four switches are connected to bits 0 through 3 of PORTC. The D0-D7 pins of the LCD are connected to bits 0 through 7 of PORTD. The RS, R/$\overline{W}$, and EN pins of the LCD are connected to bits 0, 1, and 2 of PORTB of PIC18F4321.

Use time delay rather than the busy bit before outputting the command or data to the LCD. Also, assume 1-MHz default crystal frequency for the PIC18F4321.

(a) Write a PIC18F assembly language program to accomplish this.

(b) Write a C language program to accomplish this.

Solution

(a) Three subroutines are used: one for outputting command code, one for delay, and one for LCD data. Since subroutines are used, the hardware STKPTR is initialized in the main program with an arbitrarily chosen value of 0x10. PORTB and PORTD are configured as output ports, and PORTC is set up as an input port.
The PIC18F assembly language program is provided below:

```
        INCLUDE <P18F4321.INC>
        ORG     0x100       ; Start of the MAIN program

MAIN    MOVLW   0x10        ; Initialize STKPTR with arbitrary value  of 0x10
        MOVWF   STKPTR
        CLRF    TRISD       ; PORTD is output
        CLRF    TRISB       ; PORTB is output
        SETF    TRISC       ; PORTC is input
        CLRF    PORTB       ; rs=0 rw=0 en=0
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
        MOVLW   0x0C        ; Display on, Cursor off
        CALL    CMD
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
        MOVLW   0x01
        CALL    CMD         ; Clear Display
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
        MOVLW   0x06        ; Shift Cursor to the right
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
        MOVLW   0x80        ; Move cursor to the start of the first line
        CALL    CMD
        MOVLW   D'5'        ; 10 ms delay
        CALL    DELAY
        MOVLW   A'S'        ; Send ASCII S
        CALL    LCDDATA
        MOVLW   A'w'        ; Send ASCII w
        CALL    LCDDATA
        MOVLW   A'i'        ; Send ASCII i
        CALL    LCDDATA
        MOVLW   A't'        ; Send ASCII t
        CALL    LCDDATA
        MOVLW   A'c'        ; Send ASCII c
        CALL    LCDDATA
        MOVLW   A'h'        ; Send ASCII h
        CALL    LCDDATA
        MOVLW   A' '        ; Send ASCII space
        CALL    LCDDATA
        MOVLW   A'V'        ; Send ASCII V
```

TABLE 9.2 Typical LCD commands along with 8-bit codes in hex

Hex	Command
0x01	Clear the screen
0x02	Return home
0x04	Shift cursor to left
0x06	Shift cursor to right
0x05	Shift display to right
0x07	Shift display to left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor blinking
0x10	Shift cursor position to the left
0x14	Shift cursor position to the right
0x80	Move cursor to the start of the first line

```
        CALL    LCDDATA
        MOVLW   A'a'        ; Send ASCII a
        CALL    LCDDATA
        MOVLW   A'l'        ; Send ASCII l
        CALL    LCDDATA
        MOVLW   A'u'        ; Send ASCII u
        CALL    LCDDATA
        MOVLW   A'e'        ; Send ASCII e
        CALL    LCDDATA
        MOVLW   A':'        ; Send ASCII :
        CALL    LCDDATA
AGAIN   MOVF    PORTC, W    ; Move switch value to WREG
        ANDLW   0x0F        ; Mask lower 4 bits
        IORLW   0x30        ; Convert to ASCII data by ORing with 0x30
        CALL    LCDDATA     ; Display switch value on screen
        MOVLW   0x10        ; Shift cursor to left
        CALL    CMD
```

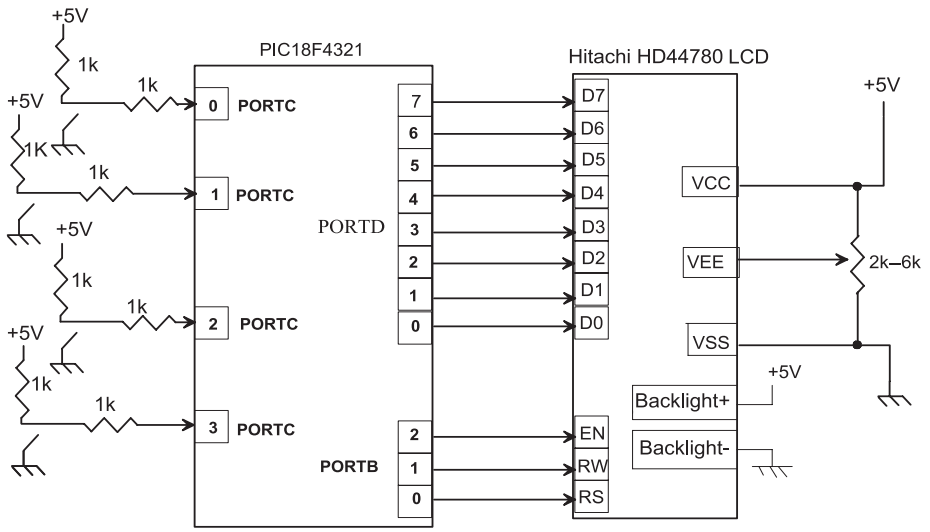


FIGURE 9.14 Figure for Example 9.4

```
        BRA     AGAIN
CMD     MOVWF   PORTD     ; Command is sent to PORTD
        MOVLW   0x04
        MOVWF   PORTB     ; rs=0 rw=0 en=1
        MOVLW   D'5'      ; 10 ms delay
        CALL    DELAY
        CLRF    PORTB     ; rs=0 rw=0 en=0
        RETURN
LCDDATA MOVWF   PORTD     ; Data sent to PORTD
        MOVLW   0x05      ; rs=1 rw=0 en=1
        MOVWF   PORTB
        MOVLW   D'5'      ; 10 ms delay
        CALL    DELAY
        MOVLW   0x01
        MOVWF   PORTB     ; rs=1 rw=0 en=0
        RETURN
DELAY   MOVWF   0x20
LOOP1   MOVLW   D'255'    ; LOOP2 provides 2 ms delay with a count of 255
        MOVWF   0x21
LOOP2   DECFSZ  0X21
        GOTO    LOOP2
        DECFSZ  0x20
        GOTO    LOOP1
        RETURN
        END
```

In the above, the following loop for the 2 ms delay is used:

```
LOOP1   MOVLW   D'255'    ; LOOP2 provides 2 ms delay with a count of 255
        MOVWF    0x21
LOOP2   DECFSZ  0X21
        GOTO    LOOP2
```

For a 1-MHz default crystal frequency, the PIC18F clock period will be 1 μsec. Hence, each instruction cycle will be 4 microseconds. For 2 ms delay, total cycles = (2 ms)/(4 μsec)= 500. The DECFSZ in the loop will require 2 cycles for (COUNT - 1) times when Z = 0 and the last iteration will take 1 cycle when skip is taken (Z = 1). Hence, the value of count will approximately be 255 (decimal) discarding execution times of certain instructions. Therefore, register 0x21 should be loaded with an integer value of 255 for an approximate delay of 2 ms.

As an example, let us consider the code for outputting a command code such as the command "move cursor to the beginning of the first line" to the LCD. From Table 8.3, the command code for this is 0x80. In the above program, the code MOVLW 0x80 moves 0x80 into WREG. The CALL CMD calls the subroutine called CMD. The CMD subroutine first outputs the command code 0x80 to PORTD using MOVWF PORTD. Since PORTD is connected to LCD's D0-D7 pins, this data will be available to be latched by the LCD. The following few lines of the code of the CMD subroutine are for outputting 0's to RS and R/$\overline{W}$ pins, and a trailing edge (1 to 0) pulse to EN pin along with a delay of 10 ms. Hence, the LCD will latch 0x80, and the cursor will move to the start of the first line. Note that an external counter of 5 is loaded into a register 0x21 with a 2 ms inner loop for LOOP2 is used for the 10 ms delay. Typical delays should be 10 to 30 milliseconds. Also, 1-MHz

default crystal frequency for the PIC18F4321 is assumed. The program then returns to the main program.

The first few lines of the main program at address MAIN perform initializations. Next, in order to display 'S', the MOVLW D'5' moves 5 (decimal) into WREG, and CALL DELAY provides a 10 ms delay using this value in the routine. After executing the DELAY routine, MOVLW A'S' moves the 8-bit ASCII code for S into WREG. The instruction CALL LCDDATA calls the subroutine called LCDDATA. The MOVWF PORTD instruction in this subroutine outputs the ASCII code for S into the D0-D7 pins of the LCD via PORTD. The next few instructions in the LCDDATA subroutine outputs 1 to the RS pin, 0 to the R/$\overline{W}$ pin, and a trailing edge (1 to 0) pulse to EN pin along with delay so that the LCD will latch ASCII code for S, and will display S on the screen.

Similarly, the program logic in the PIC18F assembly program for outputting other characters and switch input data can be explained.

(b) The C-program for the LCD is provided below:

```
#include <P18F4321.h>
#define DELAYTIME 5
void cmd(unsigned char);
void data(unsigned char);
void delay(unsigned int);
void main(void)
{
   unsigned char input, output,i;
    unsigned char tstr[13] = {'S','w',
   'i','t','c','h',' ', 'V','a','l','u','e',':'};      // Switch input
   TRISD = 0x00;                                       // configure PORTD as output
   ADCON1 = 0x0F;                                      // PORTB as digital I/O
   TRISB = 0;                                          // configure PORTB as output
   TRISC = 0xFF;                                       // configure PORTC as input
   PORTB = 0x00;                                       // rs = 0 rw=0 en = 0
   delay(DELAYTIME);
   cmd(0x0C);                                          // display on, cursor off
   delay(DELAYTIME);
   cmd(0x01);                                          // clear display
   delay(DELAYTIME);
   cmd(0x06);                                          // shift cursor to the right
   delay(DELAYTIME);
   cmd(0x80);                                          // move the cursor to the start of the first line
   delay(DELAYTIME);
   for(i = 0; i < 13;i++)
        data(tstr[i]);                                 // output "switch input:"
   while(1)                                            // Wait
   {
      input = PORTC&0x0F;                              // mask the four switch inputs
      output = 0x30|input;                             // convert it to ASCII code by ORing with 30H
      data(output);                                    // output the number
      delay(DELAYTIME);                                // 10ms delay
      cmd(0x10);                                       // shift cursor left one, goes back to the position
   }
```

```
}                                       // end main
void cmd(unsigned char value)
{
        PORTD = value;                  // command is sent to PORT D
        PORTB = 0x04;                   // rs= 0 rw=0 en=1
        delay(DELAYTIME);               // delay 10ms
        PORTB = 0x00;                   // rs=0 rw = 0 en =0
}

void data(unsigned char value)
{
        PORTD = value;                  // data is sent to PORTD
        PORTB = 0x05;                   // rs=1 rw=0 en=1
        delay(DELAYTIME);               // 10ms delay
        PORTB = 0x01;//rs=1 rw=0 en=0
}
void delay(unsigned int itime)
{
        unsigned int i,j;               // 2 ms delay, Can be verified from the
        for(i=0; i<itime; i++);         // PIC18F assembly program for LCD
          for(j = 0; j<255;j++);
}
```

9.4 Interfacing PIC18F4321 to a hexadecimal keyboard and a seven-segment display

In this section we describe the basics of interfacing the PIC18F4321 microcontroller to a hexadecimal keyboard and a seven-segment display.

9.4.1 Basics of Keyboard and Display Interface to a Microcontroller

A common method of entering programs into a microcontroller is via a keyboard. An inexpensive way of displaying microcontroller results is by using seven-segment displays. The main functions to be performed for interfacing a keyboard are:

- Detect a key actuation.
- Debounce the key.
- Decode the key.

Let us now elaborate on keyboard interfacing concepts. A keyboard is arranged in rows and columns. Figure 9.15 shows a 2×2 keyboard interfaced to a typical microcontroller such as the PIC18F4321. In Figure 9.15, the columns are normally at a HIGH level. A key actuation is sensed by sending a LOW (closing the diode switch) to each row one at a time via RC0 and RC1 of port C. The two columns can then be inputted via RD2 and RD3 of PORTD to see whether any of the normally HIGH columns are pulled LOW by a key actuation. If so, the rows can be checked individually to determine the row in which the key is down. The row and column code for the key pressed can thus be found.

The next step is to debounce the key. *Key bounce* occurs when a key is pressed or released; it bounces for a short time before making the contact. When the bounce occurs, it

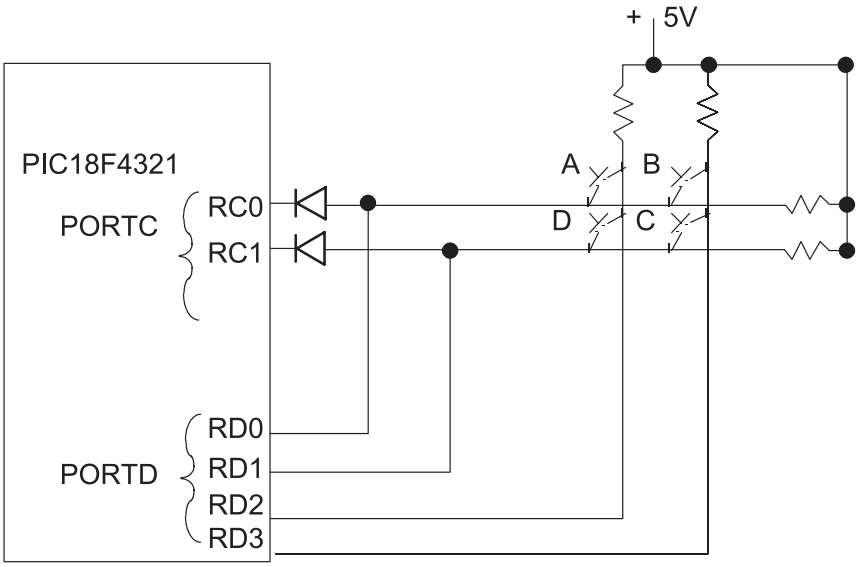


Figure 9.15 A 2x2 keyboard interfaced to the PIC18F4321

may appear to the microcontroller that the same key has been actuated several times instead of just once. This problem can be eliminated by reading the keyboard after about 20 ms and then verifying to see if it is still down. If it is, the key actuation is valid. The next step is to translate the row and column code into a more popular code, such as hexadecimal or ASCII. This can easily be accomplished by a program. Certain characteristics associated with keyboard actuation must be considered while interfacing to a microcontroller. Typically, these are two-key lockout and *N*-key rollover. The two-key lockout ensures that only one key is pressed. An additional key depressed and released does not generate any codes. The system is simple to implement and most often used. However, it might slow down the typing because each key must be released fully before the next one is pressed down. On the other hand, the *N*-key rollover will ignore all keys pressed until only one remains down.

Now let us elaborate on the interfacing characteristics of typical displays. The following functions are typically performed for displays:

- Output the appropriate display code.
- Output the code via right entry or left entry into the displays if there is more than one display.

These functions can easily be realized by a microcontroller program. If there is more than one display, the displays are typically arranged in rows. A row of four displays is shown in Figure 9.16. In the figure, one has the option of outputting the display code via right entry or left entry. If the code is entered via right entry, the code for the least significant digit of the four-digit display should be outputted first, then the next-digit code, and so on. The program that outputs to the displays are so fast that visually all four digits will appear on the display simultaneously. If the displays are entered via left entry, the most significant digit must be outputted first and the rest of the sequence is similar to that of the right entry.



FIGURE 9.16 A row of four displays.

Two techniques are typically used to interface a hexadecimal display to the microcontroller: nonmultiplexed and multiplexed. In nonmultiplexed methods, each hexadecimal display digit is interfaced to the microcontroller via an I/O port. Figure 9.17 illustrates this method. BCD-to-seven-segment conversion is done in software. The microcontroller can be programmed to output to the two display digits in sequence. However, the microcontroller executes the display instruction sequence so fast that the displays appear to the human eye at the same time. Figure 9.18 illustrates the multiplexing method of interfacing the two hexadecimal displays to the microcontroller. In the multiplexing scheme, appropriate seven-segment code is sent to the desired displays on seven lines common to all displays. However, the display to be illuminated is grounded. Some displays, such as the Texas Instrument's TIL 311, have an on-chip decoder. In this case, the microcontroller is required to output 4 bits (decimal) to a display.

9.4.2 PIC18F4321 Interface to a Hexadecimal Keyboard and a Seven-Segment Display

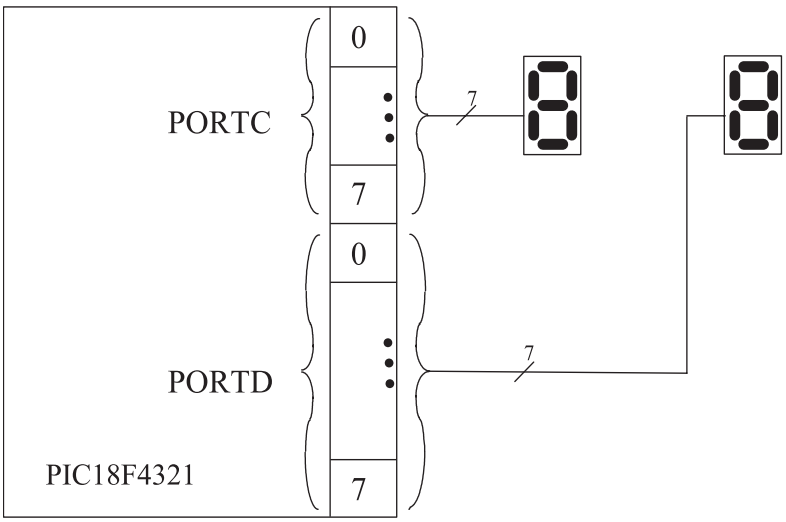


FIGURE 9.17 Nonmultiplexed hexadecimal displays.

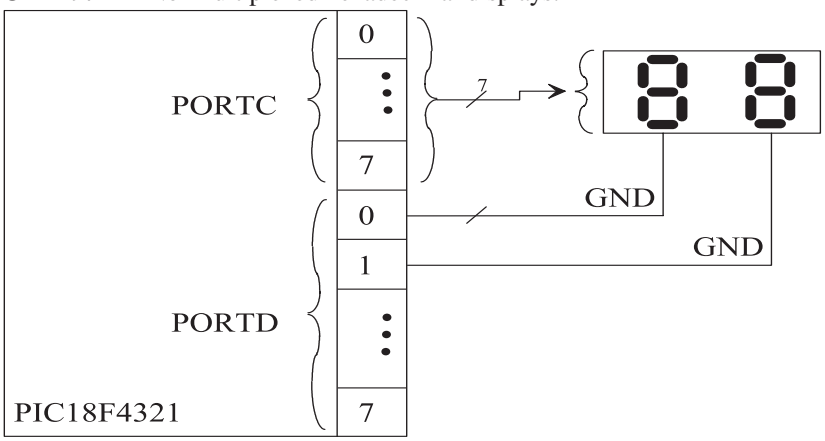


FIGURE 9.18 Multiplexed hexadecimal displays.

In this section, the basic concepts associated with interfacing a hexadecimal keyboard along with a seven-segment display to the PIC18F4321 is provided in a simplified manner. The PIC18F4321 microcontroller is designed to display a hexadecimal digit (0-F) entered via a hexadecimal keypad (16 keys). The user will push one of the hex digits (0 to F) using the keys on the hexadecimal keyboard. The PIC18F4321 will input this data via PORTD, and output via PORTB to a seven-segment display. Figure 9.19 shows the hardware schematic.

Three 8-bit I/O ports (PORTB, PORTC, PORTD) of the PIC18F4321 are used in the design. Ports B, C, and D are configured as follows:

- PORTB is configured as an output port to display the key(s) pressed.
- PORTC is configured as an output port to output zeros to the rows to detect a key actuation
- PORTD is configured as an input port to receive the row–column code.

The PIC18F4321 default crystal frequency of 1 MHz is assumed. Debouncing is provided to avoid unwanted oscillation caused by the opening and closing of the key contacts. To ensure stability of the input signal, a delay of 20 ms is used for debouncing the input.

Using a 2 ms delay routine from the previous section as the inner loop, the following subroutine can be used for a delay routine of 20 ms:

```
DELAY  MOVLW   D'10'
       MOVWF   0x20
LOOP1  MOVLW   D'255'     ; LOOP2 provides a 2 ms delay with a count of 255
       MOVWF   0x21
LOOP2  DECFSZ  0X21
       GOTO    LOOP2
       DECFSZ  0x20
       GOTO    LOOP1
       RETURN
```

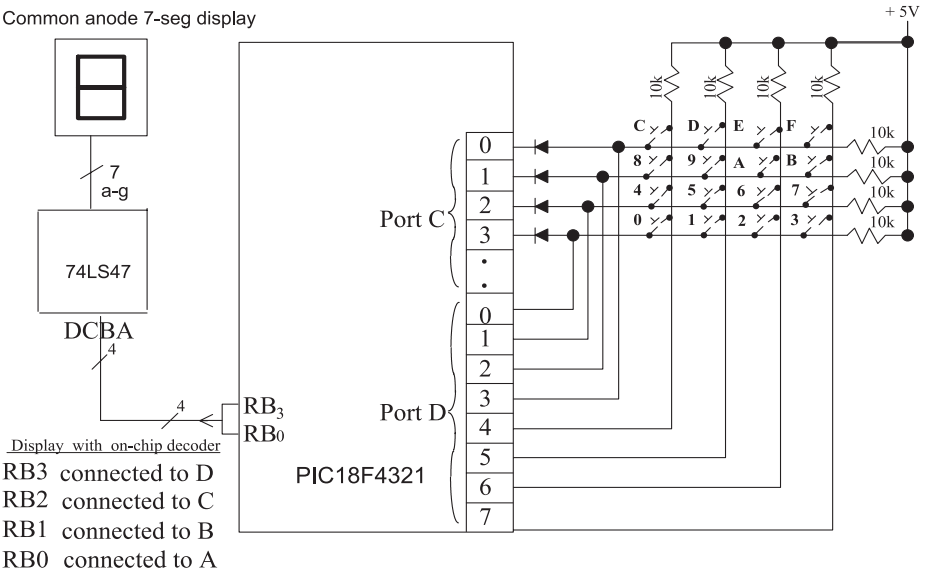


FIGURE 9. 19 PIC18F4321 interface to keyboard and display for the assembly program



FIGURE 9. 20 74LS47 display for hex digits 0-F

PIC18F assembly language program for the keyboard/display interface A PIC18F assembly program is written for the keyboard/display interface. The program scans all 16 keys for key actuation. As soon as a key actuation is detected, the program will debounce the key using the DELAY routine, and then determine the key pressed using a decode table stored in memory. The decode table contains common anode seven-segment codes for the hexadecimal digits from 0 to F.

The 74LS47 common-anode decoder is used for the display. It has four inputs (D, C, B, A with D as the most significant bit and A as the least significant bit), and seven outputs (a-g). The 7447 will display BCD numbers 0-9 and symbols for hex digits A-F as shown in Figure 9.20.

The PIC18F assembly language program written at address 0x100 for interfacing the PIC18F4321 to a hexadecimal keyboard and a seven-segment display is provided below. Seven-segment codes for 0-F are stored into data memory of the PIC18F4321 using the MOVLW instruction. Note that assembler directive DB could have been used to store the seven-segment codes in the program memory, and then transfer them to data memory using the TBLRD*+ instruction. Note that to explain the program, line numbers are included with the comments in the following:

```
        INCLUDE<P18F4321.INC>
        ORG     0x100;          #1 Start of the program
OPEN    EQU     0xF0;           #2 Assign 0xF0 to OPEN
        MOVLW   0x77;           #3 Start of the Code
        MOVWF   0x50;           #4 Code for F
        MOVLW   0xB7;           Code for E
        MOVWF   0x51
        MOVLW   0xD7;           Code for D
        MOVWF   0x52
        MOVLW   0xE7;           Code for C
        MOVWF   0x53
        MOVLW   0x7B;           Code for B
        MOVWF   0x54
        MOVLW   0xBB;           Code for A
        MOVWF   0x55
        MOVLW   0xDB;           Code for 9
        MOVWF   0x56
        MOVLW    0xEB;          Code for 8
        MOVWF   0x57
        MOVLW   0x7D;           Code for 7
        MOVWF   0x58
        MOVLW   0xBD;           Code for 6
        MOVWF   0x59
        MOVLW   0xDD;           Code for 5
```

```
                MOVWF   0x5A
                MOVLW   0xED;           Code for 4
                MOVWF   0x5B
                MOVLW   0x7E;           Code for 3
                MOVWF   0x5C
                MOVLW   0xBE;           Code for 2
                MOVWF   0x5D
                MOVLW   0xDE;           Code for 1
                MOVWF   0x5E
                MOVLW   0xEE;           # 5 End of Code. Code for 0
                MOVWF   0x5F;
; Perform initializations
                CLRF    TRISB;          # 6 Configure PORTB as an output port
                CLRF    TRISC;          # 7 Configure PORTC as an output port
                SETF    TRISD;          # 8 Configure PORTD as an input port
                MOVLW   0x5;            # STKPTR is initialized with arbitrary value
                MOVWF   STKPTR;         since subroutine DELAY is used later
                MOVLW   0;               # 9 Initialize display with 0
                MOVWF   PORTB
; Detect a key actuation, debounce it, decode, and display
SCAN_KEY        MOVLW   0;              # 10 Output 0s to rows of the keyboard
                MOVWF   PORTC
                MOVLW   OPEN;           # 11 Move 0xF0 to 0x30
                MOVWF   0x30
KEY_OPEN        MOVF    PORD,W;         # 12 Read PORTD into WREG
                SUBWF   0x30,W;         # 13 Are keys opened?
                BNZ     KEY_OPEN;       # 14 Repeat if closed
                CALL    DELAY;          # 15 Debounce for 20 ms
KEY_CLOSE       MOVF    PORTD, W;       # 16 Read PORTD into WREG
                SUBWF   0x30,W;         # 17 Are all keys closed?
                BZ      KEY_CLOSE;      # 18 Repeat if opened
                CALL    DELAY;          # 19 Debounce again for 20 ms
                SETF    0x35;           # 20 Set 0x35 contents to all 1's
                BCF     STATUS,C;       # 21 Clear Carry flag
NEXT_ROW        RLCF    0x35,F;         # 22 Set up row mask
                MOVFF   0x35,0x36;      # 23 Save row mask in 0x36
                MOVFF   0x35,PORTC;     # 24 Output 0 to a row
                MOVF    PORTD, W;       # 25 Read PORTD into WREG
                MOVWF   0x31;           # 26 Save row/column codes in 0x31
                MOVLW   0xF0;           Move data for masking
                ANDWF   0x31,W;         # 27 Mask row code
                CPFSEQ  0x30;           # 28 is column code affected?
                BRA     DECODE;         # 29 if affected, row found 0x31 has row
                                        ; and column code
                MOVFF   0x36,0x35;      # 30 Restore row mask in 0x35
                BCF     STATUS,C;       # 31 Clear carry flag to 0
                GOTO    NEXT_ROW;       # 32 Check next row
DECODE          MOVLW   D'16';          # 33 Initialize 0x32 with 16 decimal since there
```

```
        MOVWF   0x32;           # 34 are 16 hex digits
        MOVWF   0x33;           Move 16 to 0x33
        DECF    0x33,F;         Decrement 0x33 by 1 to contain hex digits F to 0
        LFSR    0, 0x50;        # 35 Initialize FSR0 with 0x50
        MOVF    0x31,W;         # 36 Move row code to WREG
SEARCH  CPFSEQ  POSTINC0;       # 37 Compare and skip if equal
        BRA     SEARCH1;        # 38 Load if not found
        MOVFF   0x33, PORTB;    # 39 Get character along with LOW enable
        BRA     NEXT1;          # 40 Branch to NEXT1
SEARCH1 MOVFF   0x31, W
        DECF    0x33,F;         Decrement 0x33
        DECF    0x32,F;         Decrement 0x32
NEXT1   GOTO    SCAN_KEY;       #41 Branch to SEARCH if not 0
DELAY   MOVLW   D'10';          Return to scan another key
        MOVWF   0x20;           20 ms delay routine
LOOP1   MOVLW   D'255';         LOOP2 provides 2 ms delay
        MOVWF   0x21
LOOP2   DECFSZ  0x21;
        GOTO    LOOP2;
        DECFSZ  0x20;
        GOTO    LOOP1;
        RETURN;
        END
```

In the program, a decode table for keys 0 through F are loaded into data memory using the MOVLW instruction starting at address 0x50 (chosen arbitrarily). The codes for the hexadecimal numbers 0 through F are obtained by inspecting Figure 9.19. For example, consider key 9. When key 9 is pressed and if a LOW is outputted by the program to bit 0 of PORTC, the second row and second column of the keyboard will be LOW. This will make the content of PORTD:

Bit Number:	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	
Data:	1	1	0	1	1	0	1	1	$= DB_{16}$

Thus, a code of DB_{16} is obtained at PORTD when key 9 is pressed. Diodes are connected at the 4 bits (bits 0-3) of PORTC. This is done to make sure that when a 0 is output by the program to one of these bits (row of the keyboard), the diode switch will close and will generate a LOW on that row.

Now, if a key is pressed on a particular row that is LOW, the column connected to this key will also be LOW. This will enable the programmer to obtain the appropriate key code for each key. Next, the assembly language program will be explained using some of the line numbers included in the comment field.

Line #1 is the starting address of the program at 0x100. This address is chosen arbitrarily. Line #2 equates label OPEN to data 0xF0. This is because when all keys are up (no keys are pushed) and 0's are outputted to the rows via PORTC in Figure 9.19, data input at PORTD will be 11110000 (0xF0). Note that bits 0 -3 of PORTD are connected to rows and bits 4-7 of PORTD are connected to columns of the keyboard.

Line #3 and #4 store code for F, and are the start of the codes while Line #5 is the end of codes.

Line #'s 6 through 8 configure ports and initializes STKPTR . Line #9 initialize the seven-segment display by outputting 0.

Line #'s 10 through 14 check to see if any key is pushed. This is done by outputting 0's to all rows via PORTC, and then inputting PORTD. If all keys are open, data at PORTD will be 0xF0. Hence, 0xF0 stored in data memory address 0x30 is subtracted from data at PORTD in WREG. If Z = 0, the program waits in a loop with label KEY_OPEN until a key is pushed. When a key is closed, Z = 1, and the program comes out of the loop. Line #15 CALLs the DELAY routine to debounce the key by providing a delay of 20 ms.

Line #'s 16 through 18 detect a key closure. The program inputs PORTD into WREG and, subtracts 0xF0 stored in 0x30 from [WREG]. If Z = 1, the program waits in a loop with label KEY_CLOSE until a key is closed. If Z = 0, the program leaves the loop. Line #19 CALLs the DELAY routine to debounce as soon as a key closure is detected. It is necessary to determine exactly which key is pressed. This is accomplished by outputting a '0' to a row while outputting 1's to the other three rows. Hence, a sequence of row-control codes (0xFE, 0xFD, 0xFB, and 0xF7, where the upper 4 bits 'F' are don't cares in this case) are outputted via PORTC. Line #'s 20 through 22 initialize 0x35 to all 1's, clear the C-bit to 0, and rotate [0x35] through carry once to the left to contain the appropriate row control code.

For example, after the first RLCF in line #22, 0x35 will contain 11111110 (0xFE). Note that the low 4 bits are the row-control code (the upper 4 bits are don't cares) for the first pass in the loop, labeled NEXT_ROW. Line #'s 23 and 24 output this data to PORTC to make the top row of the keyboard zero. The row–column code is inputted via PORTD to determine if the column code changes corresponding to each different row code. Line #'s 25 and 26 input PORTD into 0x31 via WREG.

Lines 27 through 32 make the low 4 bits 0's and retain the upper 4 bits. If the column code is not 0xF0 (changed), the input key is identified. The program then goes through a lookup table to match the row–column code saved in 0x33. If the code is found, the corresponding index value, which equals the input key's value (0-9, and symbols for A-F), is displayed. However, if no key in the top row is pushed, a 0 is outputted to the second row, and the process continues. The program is written such that it will scan continuously for an input key and update the display for each new input.

Suppose that key 9 is pushed when the program branches to DECODE at line #33. Line #'s 33 through 35 initialize data memory addresses 0x32, and 0x33 with 16 (total number of hex digits), decrement [0x33] by 1 which will initially hold 9, and will eventually contain the BCD digit to be displayed. Line #35 will load the starting address 0x50 of the decode table into FSR0 to be used as an indirect pointer. Line #36 moves the row code 0xDB (for 9) saved in data memory address 0x31 (Line #26) into WREG. The instruction "CMPSEQ POSTINC0" at line 37 will compare the code for 9 at address 0x50 (starting address of the table) with [WREG]. Since it is assumed that '9' is pushed, there will be a match, and the instruction at line #40 will be skipped. The instruction "MOVFF 0x33, PORTB" at line #39 will be executed which will output 1001 to the DCBA inputs of the 7447 via PORTB, and the digit 9 will be displayed on the seven-segment display. The instruction "BRA NEXT1" at line #40 will branch to label NEXT1 (Line #41) which, in turn, will go back, and repeat the process.

C language program for the keyboard/display interface Figure 9.21 shows the hex keyboard interface to the PIC18F4321. Comments are included in the program to explain the program logic. Since the 74LS47 is used for the display, digits 0-9 will be displayed

properly while A-F will be displayed according to Figure 9.20. For example, if the key 'F' is pushed, then all LED segments will be turned OFF.

The C-program for the PIC18F4321-Keyboard interface is provided below:

```
#include <P18F4321.h>
#define TRUE 1
#define FALSE 0
#define setbit(var,bit)((var) |= (1<<(bit)))          // #1 set a bit var = var | (1 <<bit)
#define clearbit (var,bit)((var) &= ~(1<<(bit)))   // #2 clear a bit var = var & ~(1<<bit)
#define testbit(var,bit)((var) & (1<<(bit)))         // #3 test a bit return var & (1<<bit)
unsigned char row=0x0F;                                  // #4 all rows are initially high
  int i=0, j=0;
unsigned char Matrix[4][4] =                          // #5 create a matrix to store the
                                                              // corresponding Hex key numbers
{
{0x0F, 0x0E, 0x0D, 0x0C},
{0x0B, 0x0A, 0x09, 0x08},
{0x07, 0x06, 0x05, 0x04},
{0x03, 0x02, 0x01, 0x00}};
void delay_ms(unsigned int time);                  // #6 delay function to prived 2ms delay
void main()
{
OSCCON = 0x60;                                          // 4MHz clock
ADCON1 = 0x0F;                                         // #8 AN0-AN12 all digital input
TRISB = 0x00;                                 // #9 configure PORTB as output for 7 seg decoder
TRISC = 0x00;                                 // #10 configure PORTC as output for matrix keyboard
TRISD = 0xFF;                                 // #11 configure PORTD as input for key pressed
PORTB = 0x00;                                 // #12 output initially set to 0 at PORTB
while (1) {
PORTC = 0x00;                                 // #13 output initially set to 0 at PORTC
while(PORTD! = 0x0F);                         // #14 wait in while loop if a key is pressed
while(PORTD = = 0x0F);                        // #15 wait in while loop if no keys are pressed
for(i = 0; i<4; i++)                          // #16 Scan for the keyboard FOR COLUMN
{
for(j = 0; j<4;j++)                           // #17 Scan FOR ROWS
 {
clearbit(row,j);                              // #18 Clear a bit for corresponding row j
PORTC = row;                                  // #19 Output a low for that row
if(testbit(PORTD,i)= = 0)                     // #20 Detect if a key is pressed in column i
  {
  delay_ms(20);                               // #21 Delay for 20ms
   if(testbit(PORTD,i) ==  0)                 // #22 Check again for debouncing
   {
    PORTB = Matrix[j][i];                     // #23 output result from lookup table to PORTB
   }
   }
   setbit(row,j);                             // #24 Set the zero bits back to row variable
 }
```

```
 }
}
}
void delay_ms(unsigned int time)    // #25 2ms delay function
 {
 unsigned int i, j;
 for(i=0; i<time; i++)
 for(j=0; j<255; j++);
}
```

In Figure 9.21, the four rows and four columns of the hex keyboard are connected to the PIC18F4321 via PORTC and PORTD respectively. The top row (row 0) of the keyboard is connected to RC0, next row (row 1) to RC1, next row (row 2) to RC2, and next row (row 3) to RC3. On the other hand, the rightmost column (column 0) is connected to RD3, next column to the left (column 1) to RD2, next column to the left (column 2) to RD1, and next column to the left (column 3) to RD0. In the figure, the columns of the keyboard connected to +5V, are normally HIGH. The key matrix at line #5 in the program is arranged differently than the Keyboard in Figure 9.21. For example, in the program Key 'F' is located at row 0 and column 0. Hence, key 'F' is located on the extreme right of the top row of the key matrix.

In order to detect a key actuation, 0's are outputted on the four rows using "PORTC = 0x00;" (Line # 13). If a key is pushed, that key will place a '0' on the normally HIGH column connected to it. Thus, the pushed key will make one of the columns connected to it '0', and the other three columns will be HIGH.

Line #18 clears a bit for row j, and line #19 outputs a LOW on that row. Line #20 detects if a key is pushed in column i connected to that row. Once the key is found, it is debounced (Line #21). With i and j, the key is obtained from the key matrix [j][i], and then outputted to PORTB on the seven segment display (Line #23).

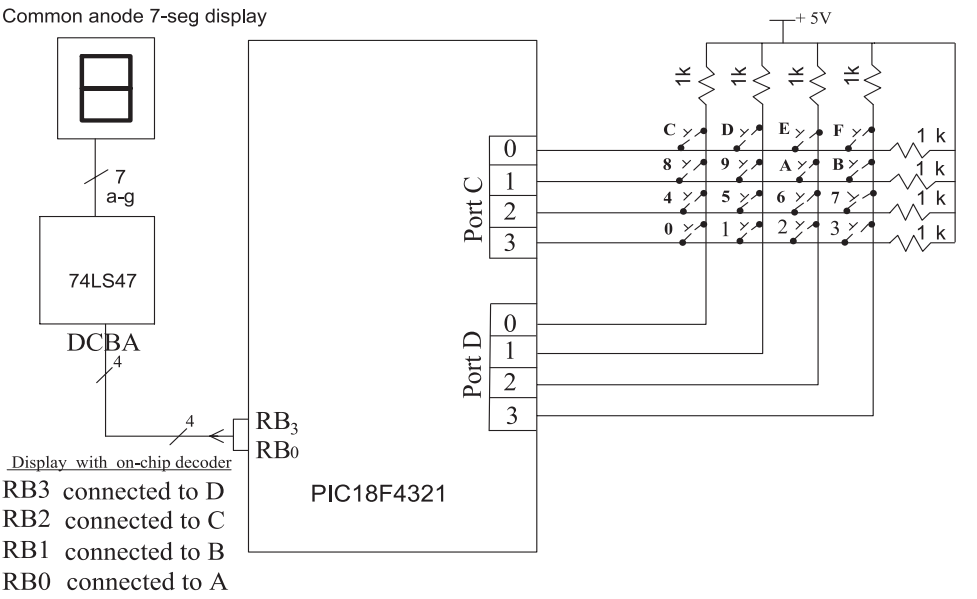


FIGURE 9.21 PIC18F4321 interface to keyboard and display for the C-program

As an example, consider pressing key number 2. Note that '2' is located at row 3 (j = 3) and at column 1 (i = 1). In the program, j = 0 and i = 0 initially. The "clearbit" function (Line #18) clears a bit, and then "PORTC = row;" (Line #19) outputs a '0' on the row. This means that for j = 0 and i = 0, PORTC outputs 0xE (1110). The "testbit" function (Line #20) checks whether the input at PORTD is 0. If it is 0, the key is found. The key is debounced (Line #21) and then outputted to PORTB. The pushed key will thus be displayed on the seven-segment display. In case of pressed key '2', the key is not found yet. Hence, the process continues until j = 3 and i =1.

When j = 3 and i = 1, the "clearbit" function at line #18 clears a bit for row 3 (j = 3) and outputs 0x7 (0111) on low four bits of PORTC. When the "if" statement with the "testbit" function is executed (Line #20), column 1 (i = 1) is obtained. The coordinate with j = 3 (row 3) and i = 1 (column 1) provides the pressed key '2'. The key is then debounced and outputted to PORTB. Hence, the seven segment display will display '2'.

QUESTIONS AND PROBLEMS

9.1 What is the interrupt address vector upon Power-on reset?

9.2 Identify the PIC18F4321 external interrupts as maskable or nonmaskable.

9.3 What are the interrupt address vectors for high priority and low priority interrupts?

9.4 What is the priority levels for INT0 through INT2 external interrupts of the PIC18F4321 upon power-on reset?

9.5 (a) Write PIC18F instruction sequence in PIC18F assembly language to set interrupt priority of INT1 as the high level, and interrupt priority for INT2 level as low level.

(b) Write C language program to set interrupt priority of INT1 as the high level, and interrupt priority for INT2 level as low level.

9.6 What is the interrupt priority level of INT0?

9.7 What are the triggering levels (rising or falling edge) of INT0 - INT2 upon power-on reset? Write a PIC18F assembly language program to activate the triggering level of INT0 by rising edge, and, the INT1 and INT2 interrupts by falling edge.

9.8 How many Interrupt-on-Change pins are provided on the PIC18F4321? Are they activated by rising or falling edge?

9.9 In Figure P9.9, the PIC18F4321 is required to turn on an LED connected to bit 1 of PORTD if the comparator output connected to bit 0 of PORTC is one (Vx > Vy); otherwise, the LED will be turned off.

(a) Write a PIC18F assembly language program at address 0x200 to accomplish this using conditional or polled I/O.

(b) Write a C-program to accomplish this.

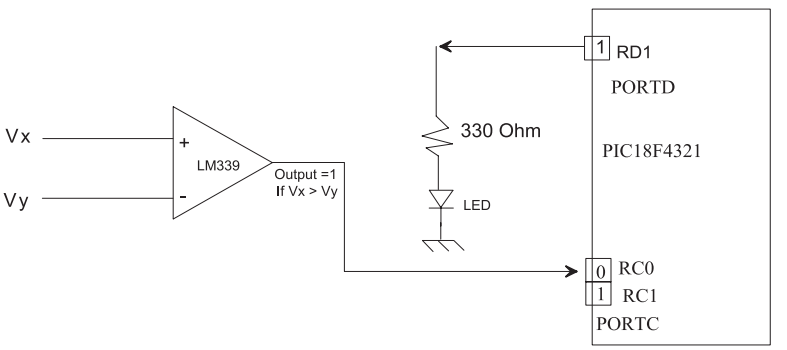


FIGURE P9.9

9.10 (a) Repeat Problem 9.9 using Interrupt I/O by connecting the comparator output to INT1. Note that RB1 is also multiplexed with INT1. Write the main program at 0x80 and the interrupt service routine at 0x150 in PIC18F assembly language. The main program will configure the I/O ports, enable interrupt INT1, initialize STKPTR to 0x10, turn the LED OFF, and then wait for interrupt. The interrupt service routine will turn the LED ON and return to the main program at the appropriate location so that the LED is turned ON continuously until the next interrupt.

(b) Repeat (a) using C except initialization of STKPTR is not requiered and also starting addresses for the main and the service routine do not need to be assigned.

9.11 In Figure P9.11, if $V_M > 12$ V, turn an LED ON connected at bit 3 of PORTA. If $V_M < 11$ V, turn the LED OFF. Using ports, registers, and memory locations as needed and INT0 interrupt:

(a) Draw a neat block diagram showing the PIC18F4321 microcontroller and the connections to ports in the diagram shown in Figure P9.11.

(b) Write the main program at 0x150 and the service routine at 0x200 in PIC18F assembly language. The main program will initialize the ports and wait for an interrupt. The service routine will accomplish the task and stop.

(c) Repeat (b) using C-language. Note that starting addresses for the main program and the service routine as in the assembly language are not needed.

9.12 What is the purpose of RS, R/$\overline{W}$, and EN pins on the Hitachi HD44780 LCD? Describe briefly how these signals are used to display data on the LCD via D0-D7 pins.

9.13 Assume the PIC18F4321- Hitachi HD44780 LCD interface of Figure 9.14.

(a) Write a program in PIC18F assembly language program to display the phrase "PIC18F" on the LCD as soon as the four input switches connected to PORT C are all HIGH.

(b) Repeat (a) using C.

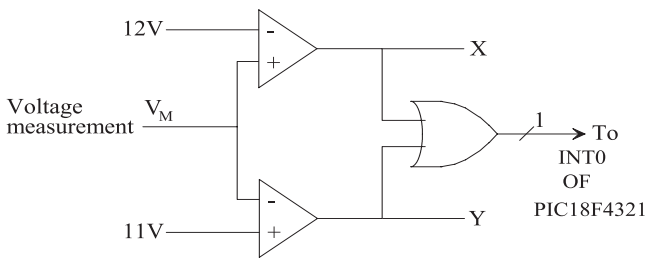


FIGURE P9.11

9.14 What are the factors to be considered for interfacing a hex keyboard to a microcontroller?

9.15 It is desired to implement a PIC18F4321-based system as shown in Figure P9.15. The system will scan a hex keyboard with 16 keys, and drive three seven-segment displays. The PIC18F4321 will input each key pressed, and scroll them in from the right side of the displays, and keep scrolling as each key is pressed. The leftmost digit is just discarded. The system continues indefinitely. Write a C language program to accomplish the above.

9.16 Design a PIC18F4321-based system (Figure P9.16) as follows: The system will drive two seven-segment digits and monitor two key switches. The system will start displaying 00. If the increment key is pressed, it will increment the display by one. Similarly, if the decrement key is pressed, the display will be decremented by one. The display will go from 00 to 99, and vice versa. Write a C language program to accomplish the above. Use a 4MHz internal clock. Connect the increment Switch to INT1 (RB1) and the decrement Switch to INT2 (RB2). Configure INT1 as high priority and INT2 as Low Priority. Connect high 7-segment display via RC3-RC0 of PORTC while Low 7-segment display via RD3-RD0 of PORTD. Use two 7447's for the displays.

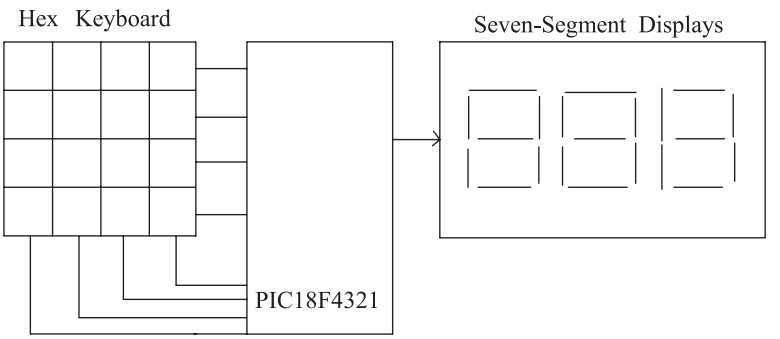


FIGURE P9.15

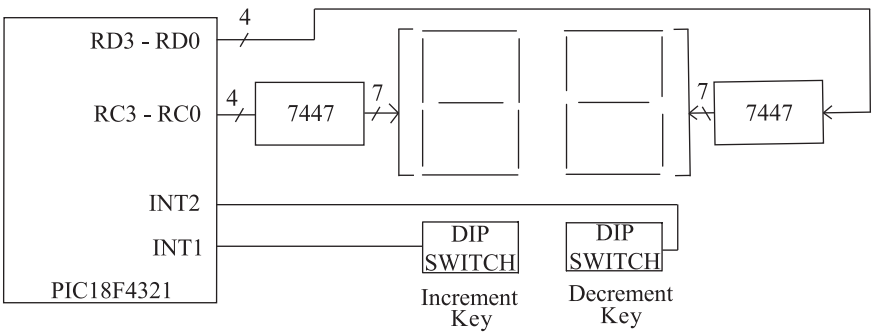


FIGURE P9.16

10
PIC18F TIMERS AND ANALOG INTERFACE

In this chapter we describe basics of programming the PIC18F on-chip peripherals such as timers and ADC using both assembly and C. Topics include PIC18F hardware timers and ADC (Analog to Digital Converter). Finally, an external DAC (Digital to Analog Converter) is interfaced to the PIC18F, and programming the DAC using C is included.

10.1 PIC18F Timers

The PIC18F4321 microcontroller includes four timers, namely, Timer0 (8-bit or 16-bit), Timer1 (16-bit), Timer2 (8-bit), and Timer3 (16-bit). These timers can be used to generate time delays using on-chip hardware. Because these timers are on-chip hardware devices, they accurately keep time in the background while other functions are performed by the PIC18F4321.

The basic hardware inside each of these timers is a register that can be incremented or decremented at the rising or falling edge of a clock. The register can be loaded with a count for a specific time delay. Time delay is computed by subtracting the initial starting count from the final count in the register, and then multiplying the subtraction result by the clock frequency.

These timers can also be used as event counters. Note that an event counter is basically a register with the clock replaced by an event such as a switch. The counter is incremented or decremented whenever the switch is activated. Thus, the number of times the switch is activated (occurrence of the event) can be determined.

Timer0 can operate as an 8-bit timer or a 16-bit timer. Timer0 utilizes the internal clock when used as a timer, and an external clock when used as a counter. Timer1 can only be used as a 16-bit timer or a 16-bit counter; it cannot be used as an 8-bit timer or 8-bit counter.

Like Timer0, Timer1 uses the internal clock when used as a timer, and the external clock when used as a counter. Timer2 can be used as an 8-bit timer using the internal clock. It cannot be used as a counter. Finally, Timer3 can operate as a 16-bit timer or a 16-bit counter. It cannot be used as an 8-bit timer or an 8-bit counter.

In this section, topics related to how Timer0 through Timer3 can be used as timers to provide time delays will be covered. Use of these timers as counters will not be provided. Timer1, Timer2, and Timer3 are used for the CCP (Capture/Compare/Pulse Width Modulation) operation. For example, the PIC18F4321 utilizes Timer1 or Timer3 for the capture and compare modes, and Timer2 for the PWM mode. Note that Timer0 is not used by the CCP module. These topics will be discussed later in this chapter.

In summary, the timer is basically a binary counter (Timer register) which can be programmed to count clock pulses . Once the timer register reaches the maximum value

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

(0xFF for 8-bit, 0xFFFF for 16-bit), it will roll back to 0 setting up an overflow flag to one at the next cycle, and also generate an interrupt if enabled. In the following, the basic features of the PIC18F timers will be discussed.

Polling Timer flags Timer0 counts up to a maximum value of 0xFF when it operates as an 8-bit timer and then overflows to 0. Also, Timer0 or Timer1 or Timer3 counts up to a maximum value of 0xFFFF while operating as a 16-bit timer, and then overflows to 0.

The PIC18F4321 Timer2 contains an 8-bit timer register (TMR2) and an 8-bit period register (PR2). Timer2 overflows to 0 for TMR2-to-PR2 match. The overflow is indicated automatically by the PIC18F4321 by setting a flag bit (TMRxIF bit where x = 0, 1, 2, 3) associated with each of these timers. This flag bit can be polled in software to provide time-related functions. All four timers include the flag bit.

The PIC18F4321 timers along with their basic features are summarized in TABLE 10.1.

Timer interrupts The problem with polling the timer flag bit is that the PIC18F4321 has to wait in a loop checking for the TMRxIF bit to be HIGH. In this case, the PIC18F4321 will not be able to do any other task. Hence, timer interrupts can be used. This concept is similar to the advantages of using interrupt I/O over polled I/O.

Since timers are considered as peripheral devices, a single peripheral interrupt (PEIE) bit is provided in the INTCON register for all four timers. Each timer (TMRx where x = 0,1,2,3) is also provided with an interrupt enable bit (TMRxIE where x = 0,1,2,3) and an interrupt flag bit (TMRxIF where x = 0,1,2,3).

Next, in order for obtaining timer-driven interrupts with any of the four timers (Timer0 through Timer3), initializations of certain registers are necessary. For example, consider Figure 10.1 for Timer0. The following bits associated with Timer0 can be initialized via programming as follows: TMR0IF = 0, TMR0IE = 1, PEIE = 1, GIE = 1. This can be accomplished using PIC18F assembly as follows

```
BCF     INTCON, TMR0IF
BSF     INTCON, TMR0IE
BSF     INTCON, PEIE
BSF     INTCON, GIE
```

The equivalent C-code is provided below:

TABLE 10.1 PIC18F4321 Timers with basic features

Timer	Size	Register (Enable and Flag bits)	Counter register
Timer0	8-bit or 16-bit	TMR0IE, TMR0IF in INTCON	TMR0L for 8-bit TMR0H:TMR0L for 16-bit
Timer1	16-bit	TMR1IE in PIE1 TMR1IF in PIR1	TMR1H:TMR1L
Timer2	8-bit	TMR2IE in PIE1 TMR2IF in PIR1	TMR2
Timer3	16-bit	TMR3IE in PIE2 TMR3IF in PIR2	TMR3H:TMR3L

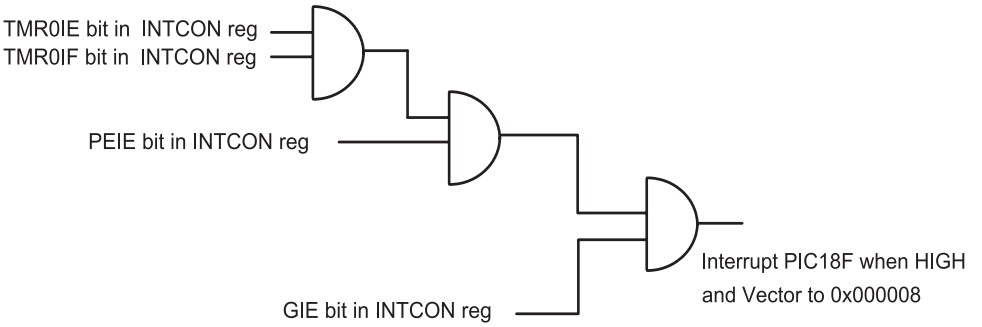


FIGURE 10.1 Timer0 Interrupt

```
INTCONbits.TMR0IF = 0;
INTCONbits.TMR0IE = 1;
INTCONbits.PEIE = 1;
INTCONbits.GIE = 1;
```

As soon as Timer0 interrupts the PIC18F4321, the microcontroller completes execution of the current instruction and loads PC with address 0x000008 (assuming default mode). A service routine can be written at this address to perform timer-related functions. Note that interrupt-driven timers using Timer1 through Timer3 work in the same manner.

Prescaler and Postscaler Timers are provided with prescaler and postscaler options which can be programmed via the timer's control registers. The prescaler option divides the PIC18F's clock by an integer, and reduces the clock period. This reduction in clock period provides the timer with higher time delays. For example, consider internal clock of 4 MHz. Since the PIC18F timer works with a divide-by-4 crystal, the timer frequency = (4 MHz/4) = 1 MHz. Using a prescaler value of 1:4, Timer clock frequency = (1MHz/4) = 0.25 MHz = 250 kHz. Hence, a lower clock frequency will increase the clock period providing a higher time delay. Note that prescaler means that the timer clock frequency is divided before the frequency is used by the timer. In the PIC18F4321, Timer0, Timer1, and Timer3 are provided with prescalers. These timers do not have postscaler option.

The postscaler also divides the clock by an integer after the timer is used. This clock is then used to slow down setting of the timer interrupt flag bit. In the PIC18F4321, the Timer2 is provided with both prescaler and postscaler options. As will be shown later, Timer2 prescaler option can be used to obtain more time delays while the Timer2 postscaler option can be programmed to delay TMR2IF (Timer Interrupt Flag) generation.

10.1.1 Timer0

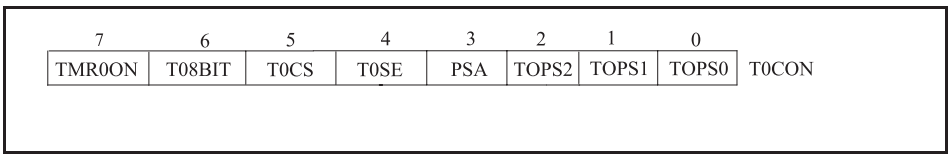
The Timer0 can operate as a timer or as a counter in 8-bit or 16-bit mode. The Timer0 uses the internal clock when used as a timer, and external clock (T0CK1) when used as a counter.

Timer0 as a timer The Timer0 can be used as a timer by setting the TMR0ON (bit 7 of T0CON register of Figure 10.2) to 1 using the PIC18F assembly instruction "BSF T0CON, TM0ON" or the C-equivalent "T0CONbits.TMR0ON=1;". After the Timer0 is started, it counts up by incrementing the contents of the register (TMR0L for 8-bit timer mode or TMR0H:TMR0L for 16-bit timer mode) by 1 at each instruction cycle. The TMR0L counts up until the TMR0L reaches 0xFF in the 8-bit mode. The TMR0IF (interrupt on overflow)

flag bit in the INTCON register is set to 1 when the TMR0L rolls over from 0xFF to 0x00. In the 8-bit mode, only the TMR0L register is used; the TMR0H register is not used, and contains a value of 0.

Similarly, in the 16-bit, after the Timer0 is started, the TMR0H:TMR0L register pair counts up until the TMR0H:TMR0L reaches 0xFFFF in the 16-bit timer mode. The TMR0IF (interrupt on overflow) flag bit in the INTCON register is set to 1 when the TMR0H:TMR0L rolls over from 0xFFFF to 0x0000. The timer can be stopped in either 8-bit or 16-bit mode by clearing the TMR0ON (bit 7 of T0CON register of Figure 10.2) to 0 using the PIC18F assembly instruction "BCF T0CON, TM0ON" or the C-code "T0CONbits.TMR0ON =0;". The T0CON register shown in Figure 10.2 controls all aspects of the Timer0 operation, including the prescale selection. Timer0 is both readable and writable.

Timer0 as a counter The Timer0 can be configured as a counter by setting the T0CS bit (bit 5 in the T0CON register of Figure 10.2) to 1. This will enable the PIC18F4321 to use



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 = External clock connected to RA4/T0CKI pin (pin 6; Timer0 external clock input)
0 = Internal clock from crystal oscillator (divide by 4; crystal frequency can vary from 4-MHz to 25 MHz)

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

FIGURE 10. 2 T0CON (TIMER0 Control) Register

an external clock connected to the T0CK1 pin. The T0SE bit (bit 4 of the T0CON register can then be cleared to 0 to increment the Timer0 register to increment on the rising edge of the clock or set to one to increment the Timer0 register on the falling edge of the clock.

Timer0 block diagrams Figure 10.3 and Figure 10.4 show the simplified block diagrams of the Timer0 module in 8-bit and 16-bit modes respectively. In Figure 10.3 TOSE (bit 4 of T0CON register of Figure 10.2) is Exclusive-ORed with the T0CK1 clock pin of the PIC18F4321. The output of the Exclusive-OR gate is selected by T0CS (bit 5 of T0CON of Figure 10.2). When T0CS is 0, the internal oscillator (Fosc/4) is used (T0CS=0), and the Timer0 operates as a timer; otherwise, the external clock is selected (T0CS=1) and the Timer0 operates as a counter. The Timer0 bypasses the prescaler if PSA bit (bit 3 of T0CON register of Figure 10.2) is 1; otherwise, the prescaler is selected (PSA = 0), and is specified by TOPS2:TOPS0 (T0CON of Figure 10.2). The next block provides a two cycle (Tosc) delay. This is because when control data is written to T0CON register, the increment operation is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the timer register called TMR0. Example 10.1 illustrates this.

An interrupt on overflow indicated by TMR0IF (bit 2 of INTCON register in Figure 9.7) is set to one if the TMR0L rolls over from 0xFF to 0x00.

The block diagram for the Timer0 16-bit mode of Figure 10.4 can similarly be explained.

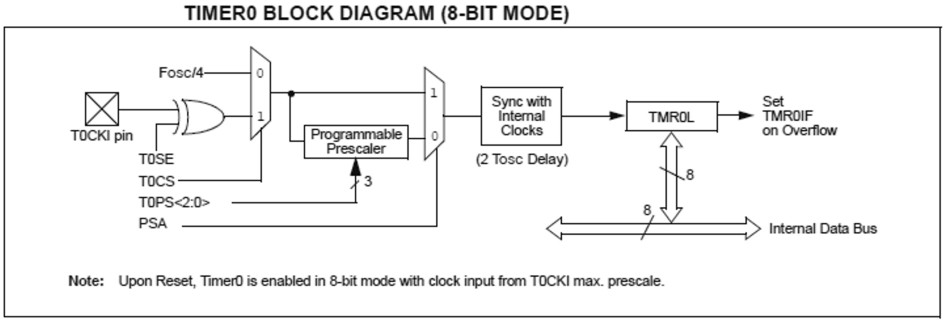


FIGURE 10. 3 TIMER0 Block Diagram (8-bit Mode)

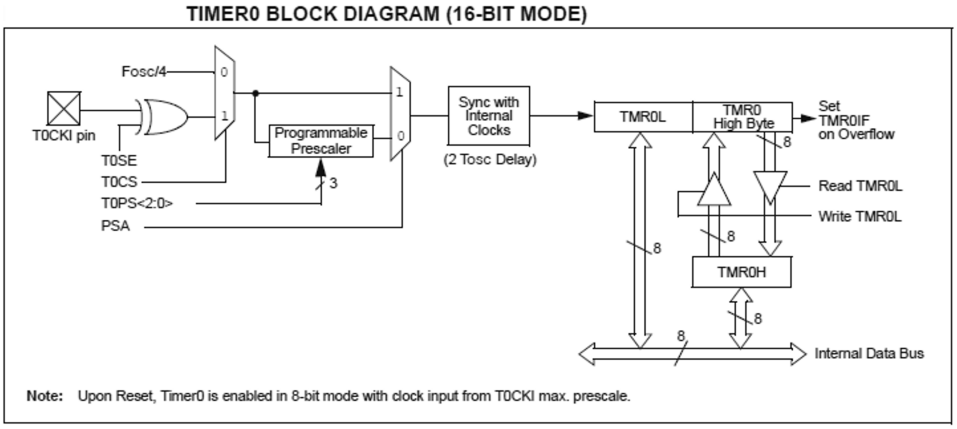


FIGURE 10. 4 TIMER0 Block Diagram (16-bit Mode)

Timer0 Read/Write in 16-Bit Mode Two 8-bit registers (TMR0H and TMR0L) are used to hold the 16-bit value in this mode. The register pair TMR0H: TMR0L is used as a 16-bit register in the 16-bit mode. The 8-bit high byte TMR0H is latched (buffered). It is not the actual high byte of Timer0 in 16-bit mode and is not directly readable nor writable (refer to Figure 10.4). Since TMR0H is not the actual high byte register of Timer0, one should initialize (write to) TMR0H before TMR0L to avoid any errors. Note that the upper 8-bit value of the timer is stored in the latched register, and loaded into actual TMR0H when TMR0L is loaded.

Similarly, a write to the high byte of Timer0 must also take place through the TMR0H Buffer register. The high byte is updated with the contents of TMR0H when a write occurs to TMR0L. This allows all 16 bits of Timer0 to be updated at once.

Prescaler An 8-bit counter is available as a prescaler for the Timer0. The value of the Prescaler is set by the PSA (bit 3 of T0CON), and T0PS2:T0PS0 bits (bits 0 through 3 of T0CON) which determine the prescaler assignment and prescale ratio. Clearing the PSA bit assigns the prescaler to the Timer0. When it is assigned, prescale values from 1:2 through 1:256 in power-of-2 increments are selectable. This is shown in Figure 10.2.

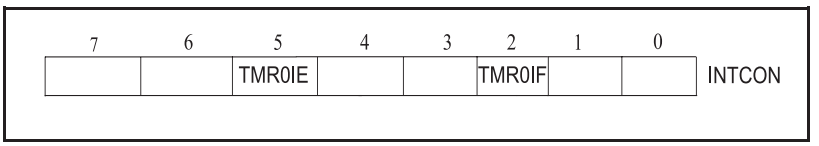
When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g.,in assembly, CLRF TMR0, MOVWF TMR0, BSF TMR0) will clear the prescaler count. The prescaler assignment is fully under software control and can be changed "on-the-fly" during program execution.

Timer0 Interrupt and Timer0 Flag bits As mentioned before, the Timer0 counts up in increments of one from 0x00 to 0xFF for 8-bit mode and from 0x0000 to 0xFFFF for 16-bit mode. Hence, any value between 0x00 and 0xFF can be loaded into TMR0L register for 8-bit mode. For 16-bit mode, any value between 0x0000 and 0xFFFF can be loaded into the 16-bit register TMR0H:TMR0L. An extra clock cycle is required when Timer0 rolls over from 0xFF to 0x00 in 8-bit mode or from 0xFFFF to 0x0000 in 16-bit mode.

The INTCON register (Figure 9.7) shows the TMR0IE and TMR0IF bits. The TMR0 interrupt can be generated internally (if enabled by setting TMR0IE to 1) using "INTCONbits.TMR0IE = 1;" in C. This is used for interrupt driven Timer0.

When the TMR0 register overflows from 0xFF to 0x00 in 8-bit mode, or from 0xFFFF to 0x0000 in 16-bit mode, this overflow sets the TMR0IF flag bit in the INTCON register to 1 (Figure 10.5). The TMR0IF can be polled to obtain time delays using Timer0.

As mentioned before, for interrupt driven Timer0, the following C-code will clear TMR0IF to 0, set PEIE to 1, TMR0IE to 1, GIE to 1, and then start Timer0:



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

FIGURE 10. 5 INTCON Register with the TMR0IE and TMR0IF bits

```
INTCONbits.TMR0IF = 0;
INTCONbits.PEIE = 1;          // enable peripheral interrupt
INTCONbits.TMR0IE = 1;        // Set TMR0IE to 1
INTCONbits.GIE = 1;           // Enable global interrupt
T0CONbits.TMR0ON = 1;         // Start TMR0
```

Once TMR0 overflows (counts from 0xFF to 0x00 for 8-bit or from 0xFFFF to 0x0000 for 16-bit), the PIC18F4321 sets the TMR0IF bit to one in the INTCON register. The PIC18F4321 is then interrupted. The microcontroller completes execution of the current instruction, saves the program counter (PC) internally, and automatically loads the PC with 0x00008 where a service routine can be written. If enough locations are not available, a branch instruction can be placed at this interrupt address vector to branch to another address so that adequate memory locations are available to write the service routine to perform the desired task.

Interrupt priority for Timer0 is determined by the value contained in the interrupt priority bit, TMR0IP (bit 2 of INTCON2, Figure 9.9)**.** The PIC18F4321 can use the Timer0 interrupt as a high priority interrupt (vector address 0x000008) by setting TMR0IP to 1 using the C-code: "INTCON2bits.TMR0IP = 1;".

The PIC18F4321 can use the timer interrupt as a low priority interrupt (vector address 0x000018) by clearing TMR0IP to 0 using the Code: INTCON2bits.TMR0IP = 0;. The TMR0IF bit must be cleared via programming in the Interrupt Service Routine. Note that the PIC18F instruction "BCF INTCON, TMR0IF" will clear the timer interrupt flag bit.

In the 8-bit mode, the TMR0IF bit is set to one when the timer value in 8-bit register TMR0L is incremented from 0xFF to 0x00 (overflow). In the 16-bit mode, the TMR0IF bit is set to one when the timer value in the 16-bit register TMR0H:TMR0L is incremented from 0xFFFF to 0x0000 (overflow). An extra clock is required when Timer0 rolls over from 0xFF to 0x00 in 8-bit mode or from 0xFFFF to 0x0000 in 16-bit mode.

Example 10.1 Assuming 4 MHz crystal oscillator, calculate the time delay for the following PIC18F instruction sequence:

```
MOVLW    0xD4
MOVWF    T0CON      ; Initialize T0CON with 0xD4
MOVLW    0x80       ; Load 8-bit timer with count 0x80
MOVWF    TMR0L
```

Solution

The above PIC18F instruction sequence loads 0xD4 into the T0CON register. Note that 0xD4 = 11010100. Hence, from Figure 10.2, the T0CON register can be drawn with the binary data as shown in Figure 10.6. Comparing data of Figure 10.6 with Figure 10.2, the following results are obtained: TMR0ON = 1 meaning TIMER0 is ON, T08BIT = 1 meaning 8-bit timer, T0CS = 0 meaning internal instruction clock, PSA = 0 meaning

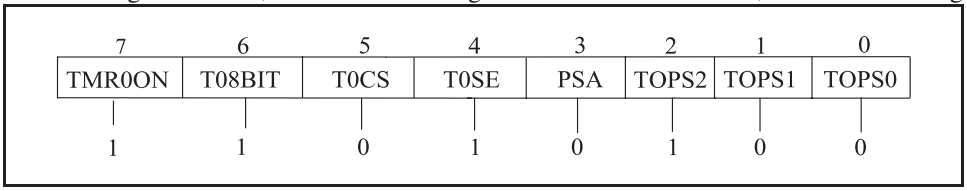


FIGURE 10.6 T0CON register with binary data 11010100

prescaler enabled, TOPS2 TOPS1 TOPS0 = 100 meaning 1:32 prescale value.

Clock Period = 1/(4 MHz) = 0.25, Instruction cycle clock period = 4 x 0.25 μsec = 1 μsec . Since the prescaler multiplies the Instruction cycle clock period by the prescaler value, Time Delay = (Instruction cycle clock perod) x (Prescaler value) x (Counter value)

= (1 μsec) x (32) x (128) = 4096 μsec = 4.096 ms

Note that, in the above, Counter value = 0x80 = 128 in decimal. This value determines the desired time delay. Also, the last two instructions MOVLW and MOVWF account for the two instruction cycles, during which the increment operation is inhibited before writing to the TMR0L register.

Example 10.2 Using Timer0 in 16-bit mode, write a PIC18F assembly language program to obtain a time delay of 1 ms. Assume an 8-MHz crystal, and a prescale value of 1:128. As mentioned in the last section, TMR0H must be initialized prior to TMR0L to avoid any errors since the contents of TMR0 are changed after Timer0 is started.

Solution

Since the timer works with a divide by 4 crystal, the timer frequency = (8MHz)/4 = 2 MHz.

Instruction cycle clock period = (1/2 MHz) = 0.5 μsec.

The bits in register T0CON of Figure 10.2 are as follows:

TMR0ON(bit 7) = 0, T08BIT (bit 6) = 0, T0CS (bit 5) = 0, PSA (bit 3) = 0,

TOPS2 TOPS1 TOPS0 = 110 for a prescale value of 1: 128. Hence, the T0CON register will be initialized with 0x06.

Time delay = Instruction cycle x Prescale value x Count

Hence, Count = (1 ms) / (0. 5 μsec x 128) = 15.625 which can be approximated to an integer value of 16 (0x0010). The timer counts up from an initialized value to 0xFFFF, and then rolls over (increments) to 0000H. The number of counts for rollover is (0xFFFF - 0x0010) = 0xFFEF.

Note that an extra cycle is needed for the roll over from 0xFFFF to 0x0000, and the TMR0IF flag is then set to 1. Because of this extra cycle, the total number of counts for roll over = 0xFFEF + 1 = 0xFFF0.

The following PIC18F assembly language program will provide a time delay of 1 ms:

```
        INCLUDE   <P18F4321.INC>
        MOVLW     0x70              ;8MHz internal clock
        MOVWF     OSCCON
        MOVLW     0x06              ; Initialize T0CON
        MOVWF     T0CON
        MOVLW     0xFF              ; Initialize TMR0H first with 0xFF
        MOVWF     TMR0H
        MOVLW     0xF0              ; Initialize TMR0L next
        MOVWF     TMR0L
        BCF       INTCON, TMR0IF    ; Clear Timer0 flag bit
        BSF       T0CON, TMR0ON     ; Start Timer0
BACK    BTFSS     INTCON, TMR0IF    ; Check Timer0 flag bit for 1
        GOTO      BACK              ; Wait in loop
        BCF       T0CON, TMR0ON     ; Stop Timer0
FINISH  BRA       FINISH            ; Halt
        END
```

Example 10.3 Using Timer0 in 16-bit mode, write a C language program to obtain a time delay of 1 ms. Assume an 8-MHz crystal, and a prescale value of 1:128. As mentioned in the last section, one should initialize TMR0H before TMR0L to avoid any errors.

Solution

Since the timer works with a divide by 4 crystal, the timer frequency = (8MHz)/4 = 2 MHz. Instruction cycle clock period = (1/2 MHz) = 0.5 μsec.

The bits in register T0CON of Figure 10.2 are as follows:
TMR0ON(bit 7) = 0, T08BIT (bit 6) = 0, T0CS (bit 5) = 0, Since internal clock is used, T0SE = x (don't care), assumed 0 here. Hence, T0SE (bit 4) = 0, PSA (bit 3) = 0, TOPS2 TOPS1 TOPS0 = 110 for a prescale value of 1: 128. Hence, the T0CON register will be initialized with 0x06.

Time delay = Instruction cycle x Prescale value x Count
Hence, Count = (1 ms) / (0.5 μsec x 128) = 15.625 which can be approximated to an integer value of 16 (0x0010). The timer counts up from an initialized value to 0xFFFF, and then rolls over (increments) to 0000H. The number of counts for rollover is (0xFFFF - 0x0010) = 0xFFEF.

Note that an extra cycle is needed for the roll over from 0xFFFF to 0x0000, and the TMR0IF flag is then set to 1. Because of this extra cycle, the total number of counts for rollover will be 0xFFF0.

The following C language program will provide a time delay of 1 ms:

```
#include<p18f4321.h>
void main(void)
    {
        OSCCON=0x70;                    // 8MHz
        T0CON=0x06;                     // Initialize T0CON
        TMR0H=0xFF;                     // Initialize TMR0H first with 0xFF
        TMR0L=0xF0;                     // Initialize TMR0L next
        INTCONbits.TMR0IF=0;            // Clear Timer0 flag bit
        T0CONbits.TMR0ON=1;             // Start Timer0
        while(INTCONbits.TMR0IF==0);// Wait for Timer0 flag bit to be 1
        T0CONbits.TMR0ON=0;             // Stop Timer0
        While(1);                       // Halt
    }
```

Example 10.4 Using Timer0 in 16-bit mode, write a C language program to toggle an LED connected to bit 0 of PORTC after 10 seconds. Assume a 4-MHz crystal, a leading edge clock, and a prescale value of 1:256. Assume a '0' will turn the LED OFF while a '1' will turn it ON.

(a) Use polled I/O

(b) Use interrupt I/O

Solution

(a) Polled I/O Since the timer works with a divide by 4 crystal, the timer frequency = (4MHz)/4 = 1 MHz.
Instruction cycle clock period = (1/1 MHz) = 1 μsec.

The bits in register T0CON of Figure 10.2 are as follows:
TMR0ON(bit 7) = 0, T08BIT (bit 6) = 0, T0CS (bit 5) = 0, T0SE (bit 4) = 0, PSA (bit 3) = 0, and TOPS2 TOPS1 TOPS0 = 111 for a prescale value of 1: 256. Hence, the T0CON register will be initialized with 0x07.

Time delay = Instruction cycle x Prescale value x Count

Hence, Count = (10 ms) / (1 μsec x 256) = 39062.5. For a 16-bit, total counts in decimal will be 65,535. Hence, Timer0 needs to count up from an initialized value of 65535- 39062.5 = 26472.5 which can be approximated to 26472 in decimal or 6768 in hex Hence, Timer0 needs to count from an initialized value to 0xFFFF (65535 in decimal), and then rollover (increments) to 0000H.

Note that an extra cycle is needed for the rollover from 0xFFFF to 0x0000, and the TMR0IF flag is then set to 1. Because of this extra cycle, the number of counts for the rollover is 0x6769 (0x6768 + 1) cycles which means that the total number of counts for rollover will be 0x6769. Hence, TMR0H = 67, TMR0L = 69.

```
//    C-program using polled I/O:
#include <P18F4321.h>
unsigned char count;
void T0Delay();
void main()
{
     OSCCON = 0x60;                 // 4MHz
     TRISC = 0x00;                  // Port C output
     PORTCbits.RC0 = 0;             // turn LED OFF

T0Delay();                          // Use Timer0
     PORTCbits.RC0 = 1;             // turn LED ON
}
void T0Delay()
{
T0CON = 0x07;                       // 16-bit, 1:256 prescale, internal clock
     TMR0H = 0x67;
     TMR0L = 0x69;
     INTCONbits.TMR0IF = 0;         // clear timer flag
     T0CONbits.TMR0ON = 1;          // start Timer0
     while (!INTCONbits.TMR0IF);    // polling, wait until timer finishes counting
     T0CONbits.TMR0ON = 0;          // Stop Timer0
}
```

(b) Interrupt I/O For interrupt I/O, we will have to loop the complete program. Although all other instructions in the loop will be repeated, execution times of these instructions are negligible compared to 10 seconds, and can be discarded.

```
// using C
#include <P18F4321.h>
unsigned char count;
void LEDON();
#pragma code T0ISR=0x08
void T0ISR()
```

```
{
    _asm
    GOTO LEDON
    _endasm
}
#pragma code

void main()
{
    OSCCON = 0x60;              // 4MHz
    TRISC = 0x00;               // Port C output
    PORTCbits.RC0 = 0;          // LED off initially
    INTCONbits.PEIE = 1;        // enable peripheral interrupt
    INTCONbits.GIE = 1;         // enable global interrupt
    T0CON = 0x07;               // 16-bit, 1:256 prescale, internal clock
    TMR0H = 0x67;
    TMR0L = 0x69;
    INTCONbits.TMR0IE = 1;      // enable Timer0 interrupt
    INTCONbits.TMR0IF = 0;      // clear Timer0 interrupt
    T0CONbits.TMR0ON = 1;       // start Timer0
    while (1);                  // wait until interrupt occurs
}

#pragma interrupt LEDON
void LEDON()
{
    PORTCbits.RC0 = 1;          // LED on
    INTCONbits.TMR0IF = 0;      // clear Timer0 flag
}
```

10.1.2 Timer1

The Timer1 can be used as a 16-bit timer or a counter. It consists of two 8-bit registers, namely TMR1H and TMR1L. The TMRIF flag in the PIR1 register goes to HIGH when the Timer1 overflows from 0xFFFF to 0x0000. An extra cycle is required when the Timer1 rolls over from 0xFFFF to 0x0000.

Timer1 is controlled through the T1CON Control register (Figure 10.7). It also contains the Timer1 Oscillator Enable bit (T1OSCEN). Timer1 can be enabled or disabled by setting or clearing the TMR1ON (bit 0 of T1CON) control bit.

Timer1 Operation Timer1 can operate as a timer, a synchronous counter, or an asynchronous counter. The operating mode is determined by the clock select bit, TMR1CS (bit 1 of T1CON). When TMR1CS is cleared to 0, Timer1 operates as a timer using the internal clock, and increments on every internal instruction cycle (Fosc/4). When the TMR1CS bit is set to 1, Timer1 increments on every rising edge of the Timer1 external clock input or the Timer1 oscillator, if enabled. Note that the on-chip crystal oscillator circuit can be enabled by setting the Timer1 Oscillator Enable bit, T1OSCEN (bit 3 of T1CON). The oscillator is a low-power circuit rated for 32 kHz crystals. Finally, the Timer1 is enabled by setting the TMR1ON (bit 1 of T1CON register) to 1.

Timer1 interrupts The TMR1 register pair (TMR1H:TMR1L) increments from 0x0000 to 0xFFFF and rolls over to 0x0000. The Timer1 interrupt, if enabled, is generated on overflow which is latched in an interrupt flag bit, TMR1IF (bit 0 of PIR1), shown in Figure 10.8. The TMR1 overflow interrupt bit can be enabled or disabled by setting or clearing the Timer1 Interrupt Enable bit, TMR1IE (bit 0 of PIE1), shown in Figure 10.9. The other bits in the PIR1 and PIE1 registers contain the individual flag and enable bits for the peripheral interrupts.

Example 10. 5 Write a PIC18F assembly language program to provide a delay of 1 ms using Timer1 with an internal clock of 4 MHz. Use the 16-bit mode of Timer1 and the prescaler value of 1:4.

Solution

For a 4 MHz clock, each instruction cycle = 4 x (1/4 MHz) = 1 μsec
Total instruction cycles for 1 ms delay = $(1 \times 10^{-3} / 10^{-6}) = 1000$
With the prescaler value of 1:4, instruction cycles = 1000 / 4 = 250
Number of Counts for rollover = $65535_{10} - 250_{10} = 65285_{10}$ = 0xFF05
An extra cycle is required for rollover from 0xFFFF to 0x0000 which sets the TMR1IF to 1.
Hence, the total number of counts = 0xFF05 + 1 = 0xFF06
Therefore, TMR1H must be loaded with 0xFF, and TMR1L with 0x06.

The PIC18F assembly language program for one ms delay is provided below:

```
        INCLUDE   <P18F4321.INC>
        MOVLW     0x60              ; 4MHz
        MOVWF     OSCCON
        MOVLW     0xA0              ; 16-bit mode, 1:4 prescaler, Timer1 OFF
        MOVWF     T1CON             ; Load into T1CON register
        MOVLW     0xFF              ; Initialize TMR1H with 0xFF
        MOVWF     TMR1H
        MOVLW     0x06              ; Initialize TMR1L with 0x06
        MOVWF     TMR1L
        BCF       PIR1, TMR1IF      ; Clear Timer1 overflow flag in PIR1
        BSF       T1CON, TMR1ON     ; Turn Timer1 ON
BACK    BTFSS     PIR1, TMR1IF      ; If TMR1IF=1, skip next instruction to halt
        GOTO      BACK
HERE    BRA       HERE              ; Halt
```

Note that the external loop can be used with the above 1 ms delay routine as the inner loop to obtain higher time delays.

Example 10.6 Using Timer1, and write a C language program to turn on an LED connected to bit 0 of PORTC after 10 seconds. Assume a 4-MHz crystal, and a prescale value of 1:8. Assume a '0' will turn the LED OFF while a '1' will turn it ON.

(a) Use polled I/O

(b) Use interrupt I/O

Solution

(a) Polled Timer Since the timer works with a divide by 4 crystal, the timer frequency = (4MHz)/4 = 1 MHz.

Instruction cycle clock period = (1/1 MHz) = 1 μs .

T1CON register will be initialized with 0x30 for an internal clock and 1:8 prescale.

Time delay = Instruction cycle x Prescale value x (Count+1)

Since Timer1 has a maximum prescaler of 8, maximum time delay = (1 μsec)x(8)x(65536) = 0.5 seconds (approximately)

Hence, an external counter is required. External counter value = (10sec/0.5)=20

```
// C-program for Timer1 polled I/O
#include <P18F4321.h>
unsigned char count;
void T1Delay();

void main()
{
  OSCCON = 0x60;                // 4MHz
   TRISC = 0x00;                // Port C output
```

7	6	5	4	3	2	1	0	
RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	T1CON

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 = Device clock is derived from Timer1 oscillator
0 = Device 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

bit 2 **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 RC0/T1OSO/T13CKI (on the rising edge)
0 = Internal clock (FOSC/4)

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

FIGURE 10.7 T1CON (Timer1 Control) Register

```
   PORTCbits.RC0 = 0;              // Turn LED OFF
   T1Delay();                      // Use Timer1
   PORTCbits.RC0 = 1;              // Turn LED ON
}
void T1Delay()
{
T1CON = 0x30;                      // 16-bit, 1:8 prescale, internal clock
   for (count = 0; count < 20; count++)
   {
      TMR1H = 0;
```

7	6	5	4	3	2	1	0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

bit 7 **PSPIF:** Parallel Slave Port Read/Write Interrupt Flag bit(1)
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred
Note 1: This bit is unimplemented on 28-pin devices and will read as '0'

bit 6 **ADIF:** A/D Converter Interrupt Flag bit
1 = An A/D conversion completed (must be cleared in software)
0 = An A/D conversion is not complete

bit 5 **RCIF:** EUSART Receive Interrupt Flag bit
1 = The EUSART receive buffer, RCREG is full (cleared when RCREG is read)
0 = The EUSART receive buffer is empty

bit 4 **TXIF:** EUSART Transmit Interrupt Flag bit
1 = The EUSART transmit buffer, TXREG is empty (cleared when RCREG is read)
0 = The EUSART transmit buffer is full

bit 3 **SSPIF:** Master Synchronous Serial Port Interrupt Flag bit
1 = The transmission/reception is complete (must be cleared in software)
0 = Waiting to transmit/receive

bit 2 **CCP1IF:** CCP1 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode.

bit 1 **TMR2IF:** TMR2-to-PR2 Match Interrupt Flag bit
1 = A TMR2-toPR2 match occurred (must be cleared in software)
0 = No TMR2-toPR2 match occurred

bit 0 **TMR1IF:** TMR1 Overflow Interrupt Flag bit,
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

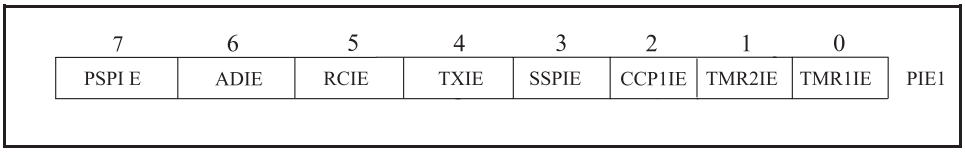
FIGURE 10.8 PIR1 (Peripheral Interrupt Request) Register1

```
        TMR1L = 0;
        PIR1bits.TMR1IF = 0;          // clear timer flag
        T1CONbits.TMR1ON = 1;         // start Timer0
        while (!PIR1bits.TMR1IF);     // polling, wait until timer finishes counting
        T1CONbits.TMR1ON = 0;         // Stop Timer0
    }
}
```

(b) Interrupt-driven Timer For interrupt I/O, we will have to loop the complete program.

Although all other instructions in the loop will be repeated, execution times of these instructions are negligible compared to 10 seconds, and can be discarded.

```
// C-program
#include <P18F4321.h>
```



bit 7 **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt

bit 6 **ADIE:** A/D Converter Interrupt Enable bit
1 = Enables the A/D interrupt
0 = Disables the A/D interrupt

bit 5 **RCIE:** EUSART Receive Interrupt Enable bit
1 = Enables the EUSART receive interrupt
0 = Disables the EUSART receive interrupt

bit 4 **TXIE:** EUSART Transmit Interrupt Enable bit
1 = Enables the EUSART transmit interrupt
0 = Disables the EUSART transmit interrupt

bit 3 **SSPIE:** Master Synchronous Serial Port Interrupt Enable bit
1 = Enables the MSSP interrupt
0 = Disables the MSSP interrupt

bit 2 **CCP1IE:** CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE:** TMR2-to-PR2 Match Interrupt Enable bit
1 = Enables the TMR2-to-PR2 match interrupt
0 = Disables the TMR2-to-PR2 match interrupt

bit 0 **TMR1IE:** TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

FIGURE 10.9 PIE1 (Peripheral Interrupt Enable) Register 1

```
unsigned char count = 0;            // loop count
void LEDON();

#pragma code T1ISR=0x08
void T1ISR()
{
   _asm
   GOTO LEDON
   _endasm
}
#pragma code

void main()
{
  OSCCON = 0x60;                    // 4MHz
   TRISC = 0x00;                    // Port C output
   PORTCbits.RC0 = 0;               // LED off initially

   INTCONbits.PEIE = 1;             // enable peripheral interrupt
   INTCONbits.GIE = 1;              // enable global interrupt
   T1CON = 0x30;                    // 16-bit, 1:8 prescale, internal clock
   TMR1H = 0;
   TMR1L = 0;
   PIE1bits.TMR1IE = 1;             // enable Timer1 interrupt enable
   PIR1bits.TMR1IF = 0;             // clear Timer1 flag
   T1CONbits.TMR1ON = 1;            // start Timer1
   while (1);                       // wait until interrupt occurs
}

#pragma interrupt LEDON
void LEDON()
{
   if (count == 19)                 // loop count from 0 to 19
      PORTCbits.RC0 = 1;            // LED on
   else
   {
      count++;
      PIR1bits.TMR1IF = 0;          // clear Timer1 flag
   }
}
```

As mentioned before, in order to implement an I/O program in PIC18F assembly such as Example 10.5 in the laboratory, the following configuration commands should be inserted using the assembler directive "config" in the PIC18F assembly language program after the "include <p18f4321.inc>" statement as follows:

```
include <p18f4321.inc>
config OSC = INTIO2;  // Select internal oscillator
config WDT = OFF;     // Watch Dog Timer OFF
config LVP = OFF;     // Low Voltage Programming OFF
config BOR = OFF;     // Brown Out Reset OFF
```

Similarly, in order to implement an I/O program using C such as Example 10.6 in the laboratory, the following configuration commands should be inserted in the C-program using the directive "#pragma config" after #include <p18f4321.h> as follows:

```
#include <p18f4321.h>
#pragma config OSC = INTIO2 // Select internal oscillator
#pragma config WDT = OFF    // Watch Dog Timer OFF
#pragma config LVP = OFF    // Low Voltage Programming OFF
#pragma config BOR = OFF    // Brown Out Reset OFF
```

Note that the above applies to all programs in PIC18F assembly and C including polled timers, interrupt-driven timers, polled ADC, and interrupt-driven ADC.

10.1.3 Timer2

Timer2 contains an 8-bit timer register and 8-bit period register (TMR2 and PR2). The Timer2 can only be programmed as a timer (not as a counter) with prescale values of 1:1, 1:4, and 1:16, and postscale values of 1:1 through 1:16.

The module is controlled through the T2CON register shown in Figure 10.10. The T2CON register enables or disables the timer and configures the prescaler and postscaler. Timer2 can be shut off by clearing the control bit, TMR2ON (bit 2 of T2CON), to minimize power consumption.

Timer2 Operation In normal operation, the PR2 register is initialized to a specific value, and the 8-bit timer register (TMR2) is incremented from 0x00 on each internal clock (FOSC/4). A 4-bit counter/prescaler on the clock input gives direct input, divide-by-4, and divide-by-16 prescale options. These are selected by the prescaler control bits, T2CKPS1:T2CKPS0 (bits 1, 0 of T2CON). The value of TMR2 is compared with the 8-bit period register, PR2, on each clock cycle. When the two values match, the Timer2 outputs a HIGH on the TMR2IF flag in the PIR1 register, and also resets the value of TMR2 to 0x00 on the next cycle.The output frequency is divided by a counter/postscale value (1:1 to 1:16) as specified in the T2CON register. Note that the interrupt is generated and the TMR2IF flag bit in the PIR1 register (Figure 10.9) is set to 1 indicating, the match between TMR2 and PR2 registers. The TMR2IF bit must be cleared to 0 using software.

Timer2 Interrupt In order for the PIC18F4321 to generate a Timer2 interrupt, the following initializations are necessary:

1. Set PEIE and GIE bits in the INTCON register to one.
2. Set TMR2IE bit in the PIE1 register to one.
3. Clear TMR2IF in the PIR1 register to 0.
4. Clear TMR2 register to 0.
5. Initialize PR2 and T2CON registers with appropriate data.
6. Turn the Timer2 ON by programming the T2CON register.
7. After the Timer2 is started, TMR2 is incremented every cycle. As soon as the TMR2 value matches PR2, the PIC18Fsets the TMR2IF in the PIR1 register to one, and then clears TMR2 to 0 in the next cycle. Since IN20IE is one, the PIC18F4321 completes execution of the current instruction, and then branches to address 0x08 where the service routine is executed.

Example 10.7 Write a PIC18F assembly language program using Timer2 to turn ON an LED connected at bit 0 of PORT D after 10 sec. Assume an internal clock 4 MHz, a prescaler value of 1:16, and a postscaler value of 1:16.

Solution

For a 4 MHz clock, each instruction cycle = 4 x 1/(4MHz) = 1 μsec. TMR2 is incremented every 1 μsec. When the TMR2 value matches with the value in PR2, the value in TMR2 is cleared to 0 in one instruction cycle. Since the PR2 is 8-bit wide, we can have a maximum counter value of 255. Let us calculate the delay with this PR2 value.

Delay = (Instruction cycle) x (Prescale value) x (Postscale value) x (Counter value + 1)
= (1 μsec) x (16) x (16) (255 + 1)
= 65.536 ms

Note that in the above, one is added to the Counter value since an additional clock is needed when it rolls over from 0xFF to 0x00, and sets the TMR2IF to 1.
External counter value for a 10 sec delay using 65.536 ms as the inner loop = (10 sec)/(65.536 ms) which is approximately 153 in decimal.

The PIC18F assembly language is provided below:

```
            INCLUDE   <P18F4321.INC>
EXT_CNT     EQU       0x50
            MOVLW     0x60              ; 4MHz
            MOVWF     OSCCON
            BCF       TRISD, TRISD0     ; Configure bit 0 of PORT D as an output
            BCF       PORTD,  RD0       ; Turn LED OFF
            MOVLW     0x7A              ; 1:16 prescaler, 1:16 postscaler Timer1 off
            MOVWF     T2CON             ; Load into T2CON register
            MOVLW     0x00              ; Initialize TMR2 with 0x00
            MOVWF     TMR2
            MOVLW     D'153'            ; Initialize EXT_CNT with 153
            MOVWF     EXT_CNT
LOOP        MOVLW     D'255'            ; Load PR2 with 255
            MOVWF     PR2
            BCF       PIR1, TMR2IF      ; Clear Timer2 interrupt flag in PIR1
            BSF       T2CON, TMR2ON     ; Set TMR2ON bit in T2CON to start timer
BACK        BTFSS     PIR1, TMR2IF      ; If TMR2IF=1, skip next instruction
            GOTO      BACK
            DECF      EXT_CNT
            BNZ       LOOP
            BSF       PORTD, RD0        ; Turn LED ON
            BCF       T2CON, TMR2ON     ; Turn off Timer2
FINISH      GOTO      FINISH            ; Halt
            END
```

In the above program, the execution times associated with some of the instructions such as MOVLW D'153',MOVWF EXT_CNT, DECF EXT_CNT, and BNZ LOOP are discarded. These execution times are very small compared to a 10 sec delay.

Example 10.8 Using Timer2, write a C language program to turn on an LED connected to bit 0 of PORTC after 10 seconds. Assume a 4-MHz crystal.

(a) Use polled I/O

(b) Use interrupt I/O

Solution

(a) Polled I/O

Since the timer works with a divide by 4 crystal, timer frequency = (4MHz)/4 = 1 MHz. Instruction cycle clock period = 1/(1 MHz) = 1 μsec. T2CON register will be initialized with 0x7A for an internal clock and 1:16 prescale , 1:16 postscale.

Maximum Time delay = Instruction cycle x Prescale value x Postscale value x (Count+1)
= (1 μsec) x (16) x (16) x (255 +1)
= 0.06 (approximately)
External counter = (10sec) / (0.06) = 167 (approximately)

```
// Timer2 polled I/O C-program
#include <P18F4321.h>
unsigned char count;
void T2Delay();

void main()
{
   OSCCON = 0x60;                  // 4MHz
   TRISC = 0x00;                   // Port C output
    PORTCbits.RC0 = 0;             // Turn LED OFF
   T2Delay();                      // Use Timer2
   PORTCbits.RC0 = 1;              // Turn  LED ON
}
```

7	6	5	4	3	2	1	0	
---	T2OUTPS3	T2OUTPS2	T2OUTPS1	T2OUTPS0	TMR2ON	T2CKPS1	T2CKPS0	T2CON

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

bit 6-3 **T2OUTPS3:T2OUTPS0:** 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

FIGURE 10.10 T2CON (Timer2 Control) Register

```
void T2Delay()
{ T2CON = 0x7A;                    // 1:16 prescale, 1:16 postscale
   for (count = 0; count < 167; count++)
   {
      TMR2 = 0;
      PR2 = 0xFF;
      PIR1bits.TMR2IF = 0;         // clear Timer2 flag
      T2CONbits.TMR2ON = 1;        // start Timer0
      while (!PIR1bits.TMR2IF);    // polling, wait until timer finishes counting
      T2CONbits.TMR2ON = 0;        // Stop Timer0
   }
}
```

(b) Interrupt I/O For interrupt I/O, we will have to loop the complete program.

Although all other instructions in the loop will be repeated, execution times of these instructions are negligible compared to 10 seconds, and can be discarded.

```
// C-program
#include <P18F4321.h>
unsigned char count = 0;           // loop count
void LEDON();
#pragma code T2ISR=0x08
void T2ISR()
{
   _asm
   GOTO LEDON
   _endasm
}
#pragma code
void main()
{
  OSCCON = 0x60;                   // 4MHz
   TRISC = 0x00;                   // Port C output
   PORTCbits.RC0 = 0;              // LED off initially
   INTCONbits.PEIE = 1;            // enable peripheral interrupt
   INTCONbits.GIE = 1;              // enable global interrupt
   T2CON = 0x7A;                   // 16-bit, 1:8 prescale, internal clock
   TMR2 = 0;
   PR2 = 0xFF;
   PIE1bits.TMR2IE = 1;            // enable Timer2 interrupt
   PIR1bits.TMR2IF = 0;            // clear Timer2 flag
   T2CONbits.TMR2ON = 1;           // start Timer2
   while (1);                      // stay here wait for interrupt occurs then loop
}

#pragma interrupt LEDON
void LEDON()
{
   if (count == 166)               // loop count = 167
```

```
      PORTCbits.RC0 = 1;          // LED on
   else
   {
      count++;
      PIR1bits.TMR2IF = 0;        // clear Timer2 flag
   }
}
```

10.1.4 Timer3

Timer3 can be used as a 16-bit timer or a 16-bit counter. Although Timer3 consists of two 8-bit registers, namely TMR3H (high byte) and TMR3L (low byte), it can only be programmed in 16-bit mode. The Timer3 module is controlled through the T3CON register (Figure 10.11). Some of the bits of the T3CON register are associated with the CCP module. This topic will be discussed in chapter 11.

7	6	5	4	3	2	1	0	
RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	$\overline{\text{T3SYNC}}$	TMR3CS	TMR3ON	T3CON

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 6,3 **T3CCP2:T3CCP1:** Timer3 and Timer1 to CCPx Enable bits
1x = Timer3 is the capture/compare clock source for the CCP modules
01 = Timer3 is the capture/compare clock source for CCP2;
Timer1 is the capture/compare clock source for CCP1
00 = Timer1 is the capture/compare clock source for the CCP modules

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 2 **T3SYNC:** Timer3 External Clock Input Synchronization Control bit
(Not usable if the device 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

FIGURE 10.11 T3CON (Timer3 Control) Register



bit 7 **OSCFIF:** Oscillator Fail Interrupt Flag bit
1 = Device oscillator failed, clock input has changed to INTOSC (must be cleared in software)
0 = Device clock operating

bit 6 **CMIF:** Comparator Interrupt Flag bit
1 = Comparator input has changed (must be cleared in software)
0 = Comparator input has not changed
bit 5 **Unimplemented:** Read as '0'

bit 4 **EEIF:** Data EEPROM/Flash Write Operation Interrupt Flag bit
1 = The write operation is complete (must be cleared in software)
0 = The write operation is not complete or has not been started

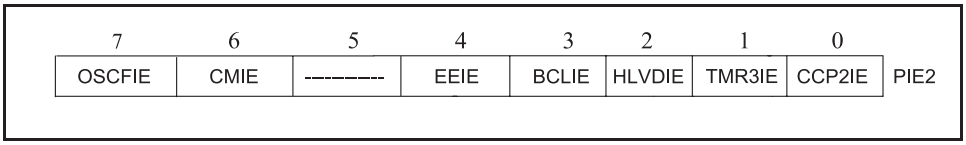
bit 3 **BCLIF:** Bus Collision Interrupt Flag bit
1 = A bus collision occurred (must be cleared in software)
0 = No bus collision occurred

bit 2 **HLVDIF:** High/Low-Voltage Detect Interrupt Flag bit
1 = A high/low-voltage condition occurred; direction determined by VDIRMAG bit (HLVDCON<7>)
0 = A high/low-voltage condition has not occurred

bit 1 **TMR3IF:** TMR3 Overflow Interrupt Flag bit
1 = TMR3 register overflowed (must be cleared in software)
0 = TMR3 register did not overflow

bit 0 **CCP2IF:** CCP2 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode.

FIGURE 10.12 PIR2 (Peripheral Interrupt Request) Register 2



bit 7 **OSCFIE:** Oscillator Fail Interrupt Enable bit
1 = Enabled
0 = Disabled

bit 6 **CMIE:** Comparator Interrupt Enable bit
1 = Enabled
0 = Disabled

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

bit 4 **EEIE:** Data EEPROM/Flash Write Operation Interrupt Enable bit
1 = Enabled
0 = Disabled

bit 3 **BCLIE:** Bus Collision Interrupt Enable bit
1 = Enabled
0 = Disabled

bit 2 **HLVDIE:** High/Low-Voltage Detect Interrupt Enable bit
1 = Enabled
0 = Disabled

bit 1 **TMR3IE:** TMR3 Overflow Interrupt Enable bit
1 = Enabled
0 = Disabled

bit 0 **CCP2IE:** CCP2 Interrupt Enable bit
1 = Enabled
0 = Disabled

FIGURE 10.13 PIE2 (Peripheral Interrupt Enable) Register 2

Timer3 Operation Timer3 can operate in one of three modes, namely timer, synchronous counter, and asynchronous counter. The operating mode is determined by the clock select bit, TMR3CS (bit 1 of T3CON, Figure 10.11). When TMR3CS is cleared to 0, Timer3 increments on every internal instruction cycle (FOSC/4). When the bit is set to 1, Timer3 increments on every rising edge of the Timer1 external clock input or the Timer1 oscillator, if enabled. We will us Timer3 mode by selecting an internal clock with TMR3CS bit = 0.

Timer3 Interrupt The TMR3 register pair (TMR3H:TMR3L) increments from 0x0000 to 0xFFFF and overflows to 0x0000. The Timer3 interrupt, if enabled, generates an overflow and is latched in an interrupt flag bit, TMR3IF (bit 1 of PIR2, Figure 10.12). This interrupt can be enabled or disabled by setting or clearing the Timer3 Interrupt Enable bit, TMR3IE (bit 1 of PIE2, Figure 10.13).

Example 10.9 Using Timer3, write a C language program to turn on an LED connected to bit 0 of PORTC after 10 seconds. Assume a 4-MHz crystal, and a prescale value of 1:8. Assume a '0' will turn the LED OFF while a '1' will turn it ON.

(a) Use polled I/O

(b) Use interrupt I/O

Solution

(a) Using polled I/O Since the timer works with divide by 4 crystal, timer frequency = (4MHz)/4 = 1 MHz. Instruction cycle clock period = (1/1 MHz) = 1 μsec. T3CON register will be initialized with 0x30 for an internal clock, a 16-bit timer, and a 1:8 prescale.

Maximum Time delay = Instruction cycle x Prescale value x Postscale value x (Count +1)

= (1 μsec) x (8) x (65535 +1)

= 0.5 (approximately)

Hence, External counter = (10s) / (0.5) = 20

```
// Timer3 Polled I/O
#include <P18F4321.h>
unsigned char count;
void T3Delay();

void main()
{
   OSCCON = 0x60;                    // 4MHz
   TRISC = 0x00;                     // Port C output
   PORTCbits.RC0 = 0;                // Turn LED OFF
  T3Delay();                         // Use Timer3
   PORTCbits.RC0 = 1;                // Turn  LED ON
}

void T3Delay()
{ T3CON = 0x30;                      // 16-bit, 1:8 prescale, internal clock
   for (count = 0; count < 20; count++)
   {
      TMR3H = 0;
      TMR3L = 0;
      PIR2bits.TMR3IF = 0;           // clear Timer3 flag
```

```
    T3CONbits.TMR3ON = 1;          // start Timer3
    while (!PIR2bits.TMR3IF);      // polling, wait until timer finishes counting
    T3CONbits.TMR3ON = 0;          // Stop Timer3
  }
}
```

(b) Interrupt I/O For interrupt I/O, we will have to loop the complete program.

Although all other instructions in the loop will be repeated, execution times of these instructions are negligible compared to 10 seconds, and can be discarded.

```
// C-program
#include <P18F4321.h>
unsigned char count = 0;          // loop count
void LEDON();

#pragma code T3ISR=0x08
void T3ISR()
{
  _asm
  GOTO LEDON
  _endasm
}
#pragma code

void main()
{
  OSCCON = 0x60;                  // 4MHz
  TRISC = 0x00;                   // Port C output
  PORTCbits.RC0 = 0;              // LED off initially

  INTCONbits.PEIE = 1;            // enable peripheral interrupt
  INTCONbits.GIE = 1;             // enable global interrupt
  T3CON = 0x30;                   // 16-bit, 1:8 prescale, internal clock
  TMR3H = 0;
  TMR3L = 0;
  PIE2bits.TMR3IE = 1;            // enable Timer3 interrupt
  PIR2bits.TMR3IF = 0;            // clear Timer3 interrupt
  T3CONbits.TMR3ON = 1;           // start Timer3
  while (1);                      // stay here, wait until interrupt occurs
}

#pragma interrupt LEDON
void LEDON()
{
  if (count == 19)                // loop count 20
    PORTCbits.RC0 = 1;            // LED on
  else
  {
    count++;
    PIR2bits.TMR3IF = 0;          // clear Timer3 interrupt
  }
```

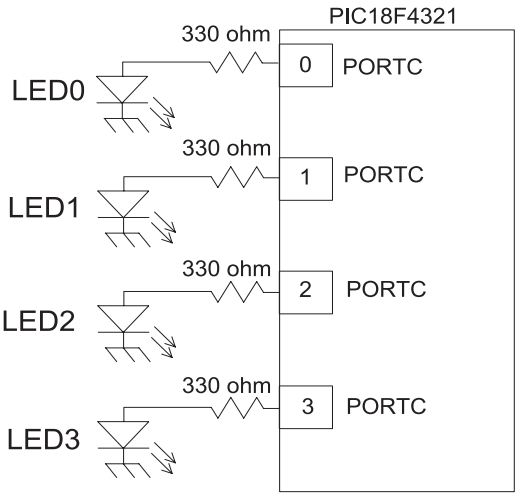


FIGURE 10.14 Figure for Example 10.10

}

Example 10.10 Four LEDs are connected at bits 0-3 of PORTC of PIC18F4321 in Figure 10.14. Write a C-program to flash the LEDs ON or OFF based on time delay provided by Timer3 by polling the TMR3IF flag. Use maximum delay provided by Timer3, and a TMR3 prescaler of 1:4. Use 1MHz default clock.

Solution

The Program is required to do the following:

1. Configure PORTC as an output, outputs 0's to LEDs to turn them OFF.
2. Configure T3CON for one 16-bit operation, 1:4 prescaler, and then turn Timer3 OFF.
3. Initialize TIMR3H:TMR3L to 0:0.
4. Clear TMR3IF to 0
5. Start Timer3, and then wait in a 'while' loop for the TMR3IF to become HIGH.
6. As soon as Timer3 counts up to 0xFFFF (providing maximum time delay), and overflows to 0x0000, the TMR3IF becomes 1.
7. Stop Timer3.
8. Re-initialize TMR3H:TMR3L to 0:0 for the next round.
9. Invert PORTC so that all four LEDs will be turned ON the first time through the loop.
10. Clear TMR3IF to 0.
11. The process continues.

The following C-code will accomplish the above:

```
#include <p18f4321.h>
void main (void)
{
    TRISC   = 0x00;                    // PortC is output
    PORTC = 0x00;
    T3CON = 0xA0;                      // Timer3 OFF with prescaler of 4
```

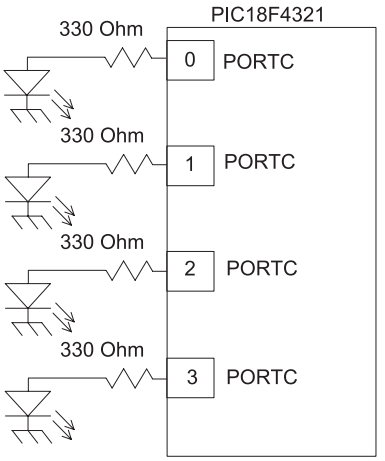


FIGURE 10.15 Figure for Example 10.11

```
    TMR3H = 0x00;                       // TMR3H initialized to 0
    TMR3L = 0x00;                       // TMR3L initialized to 0
    PIR2bits.TMR3IF=0;                  // Clear TMR3IF to 0
    while(1)
    {
        T3CONbits.TMR3ON=1;             // Turn on TMR3
        while(PIR2bits.TMR3IF= =0);     // Wait for TMR3IF to be 1
        T3CONbits.TMR3ON=0;             // Stop TMR3
        TMR3H=0x00;                     // Re-initialize TMR3H to 0
        TMR3L=0x00;                     // Re-initialize TMR3L to 0
        PORTC= ~PORTC;                  // Invert PORTC output
        PIR2bits.TMR3IF=0;              // Clear TMR3IF to 0
    }
}
```

Example 10.11 Four LEDs are connected at bits 0-3 of PORTC of the PIC18F4321 in Figure 10.15. Use 1MHz default clock. Write a C-program to do the following:

1. Turn all LEDs OFF.
2. Turn LED0 ON using time delay provided by interrupt-driven Timer0. Use maximum time delay with a prescaler value of 1:4.
3. Turn LED1 ON using time delay provided by interrupt-driven Timer1. Use maximum time delay with a prescaler value of 1:4.
4. Turn LED2 ON using time delay provided by interrupt-driven Timer2. Use maximum value for PR2 with a prescaler value of 1:4 and a postscaler value of 1:16.
5. Turn LED3 ON using time delay provided by interrupt-driven Timer3. Use maximum time delay with a prescaler value of 1:4.
6. Repeat the process.

Solution

The C-program is provided below:

```
#include <p18f4321.h>
void TMR0_ISR (void);
void TMR1_ISR (void);
```

```
void TMR2_ISR (void);
void TMR3_ISR (void);
#pragma interrupt check_int
void check_int(void)                    // Check whitch interrupt flag is triggered
{
        if ( INTCONbits.TMR0IF = = 1)
         TMR0_ISR( );
        if ( PIR1bits.TMR1IF = =1)
         TMR1_ISR ( );
        if ( PIR1bits.TMR2IF = =1)
         TMR2_ISR ( );
        if ( PIR1bits.TMR3IF = =1)
         TMR3_ISR ( );
}
#pragma code TMR_Int = 0x08             // At interrupt, code jumps here
void TMR_Int (void)
{
        _asm                            // Using assembly language
        GOTO check_int
        _endasm
}
#pragma code                            // End of code

void main (void)
{
        TRISC = 0x00;                   // PortC is output
        PORTC = 0x00;                   // All LEDs OFF
        T0CON = 0x21;                   // Timer0 OFF, 8-bit with prescaler of 4
        TMR0L = 0x00;                   // Value placed in lower 8 bits of TMR0
        INTCONbits.TMR0IF = 0;          // Clear TMR0 interrupt flag
        INTCONbits.TMR0IE = 1;          // Enable TMR0 interrupt
        T1CON = 0xA0;                   // Timer1 OFF, 16-bit with prescaler of 4
        TMR1H = 0x00;                   // TMR1H = 0
        TMR1L = 0x00;                   // TMR1L = 0
        PIR1bits.TMR1IF = 0;            // Clear TMR1 interrupt flag
        PIE1bits.TMR1IE = 0;            // Enable TMR interrupt
        T2CON = 0x 79;                  // Timer2 OFF, prescaler of 4 and postscaler of 16
        TMR2 = 0x00;                    // Value placed in Timer2
        PR2 = 0x FF;                    // Value to compare with Timer2
        PIR1bits.TMR2IF = 0;            // Clear TMR2 interrupt flag
        PIE1bits.TMR2IE = 1;            // Enable TMR2 interrupt
        T3CON = 0xA0;                   // Timer3 OFF, 16-bit with prescaler of 4
        TMR3H = 0x00;                   // TMR3H = 0;
        TMR3L = 0x00;                   // TMR3L = 0;
        PIR2bits.TMR3IF = 0;            // Clear TMR3 interrupt flag
        PIE2bits.TMR3IE = 1;            // Enable TMR3 interrupt

        INTCONbits.GIE = 1;             // Enable global interrupts
        INTCONbits.PEIE = 1;            // Enable peripheral interrupts
```

```
        T0CONbits. TMR0ON = 1;      // Turn on TMR0
        while(1);                   // Do nothing while timers are counting
}
void TMR0_ISR (void)
{
        T0CONbits.TMR0ON = 0;       // Turn off TMR0
        PORTC = 0x01;               // Light ON LED connected to PORTC bit 0
        TMR0H = 0x00;               // TMR0H = 0
        TMR0L = 0x00;               // TMR0L = 0
        INTCONbits.TMR0IF = 0;      // Clear TMR0 interrupt flag
        T1CONbits.TMR1ON = 1;       // Turn on TMR1
}
void TMR1_ISR (void)
{
        T1CONbits.TMR1ON = 0;       // Turn off TMR1
        PORTC = 0x02;               // Light LED connected to PORTC bit 1
        TMR1H = 0x00;               // TMR1H = 0
        TMR1L = 0x00;               // TMR1L = 0
        PIR1bits.TMR1IF = 0;        // Clear TMR1 interrupt flag
        T2CONbits.TMR2ON = 1;       // Turn on TMR2
}
void TMR2_ISR (void)
{
        T2CONbits.TMR2ON = 0;       // Turn off TMR2
        PORTC = 0x04;               // Light  LED connected to PORTC bit 2
        TMR2 = 0x00;                // Value placed in Timer2
        PIR1bits.TMR2IF = 0;        // Clear TMR2 interrupt flag
        T3CONbits.TMR3ON = 1;       // Turn on TMR3
}
void TMR3_ISR (void)
{
        T3CONbits.TMR3ON = 0;       // Turn off TMR3
        PORTC = 0x08;               // Light LED connected to PORTC bit 3
        TMR3H = 0x00;               // TMR3H = 0;
        TMR3L = 0x00;               // TMR3L = 0;
        PIR2bits.TMR3IF = 0;        // Clear TMR3 interrupt flag
        T0CONbits.TMR0ON =1;        // Turn on TMR0
}
```

10.2 Analog Interface

Analog interface is provided as an on-chip feature with a microcontroller. This interface typically includes on-chip A/D (Analog to Digital) and D/A (Digital to Analog) converters. This facilitates interfacing of analog signals such as temperature, flow, and pressure with the microcontroller. As mentioned in Chapter 1, A/D and D/A converters (also called ADCs and DACs) are widely used for performing data acquisition and control. The ADC (also called A/D converter) converts an analog voltage to a binary number. The DAC (also

called D/A converter), on the other hand, converts a binary number into an analog voltage or current. The terms "ADC" or "A/D converter" and "DAC" or "D/A converter" will be used interchangeably in this discussion.

Separate A/D and D/A converter chips are commercially available. These chips are typically used in microprocessor-based applications. The PIC18F4321 includes an on-chip A/D. However, an external D/A chip needs to be interfaced if digital-to-analog conversion is desired.

10.2.1 PIC18F on-chip ADC (A/D Converter)

The PIC18F on-chip ADC provides outputs of 8-bit or 10-bit. Typical ADC characteristics include Resolution, Reference voltage, ADC channels, Acquisition time, and ADC clock. We will now briefly describe them in the following.

Resolution Resolution is one of the most important characterstics of ADCs. The resolution specifies how accurately the ADC converts the analog voltage. For example, if an analog voltage (also called 'Reference Voltage' to be discussed later) is between 0 and 5 volt DC, then an 8-bit ADC will convert the input voltage into 8-bit binary number, and can measure it accurately up to 19 mV ($5/2^8 = 5/(256) = 19$ mV). This means that an 8-bit ADC cannot distinguish the difference between 1 mV and 18 mV. A 10-bit ADC, on the other hand, can convert an analog voltage from 0 to 5V DC into a 10-bit binary number with an accuracy of 4.8 mV ($5/2^{10} = 5/(1024) = 4.8$ mV). Hence, the 10-bit ADC provides higher resolution. The PIC18F on-chip ADC is 8-bit or 10-bit, and can be selected via programming.

Reference voltage Reference voltage specifies the minimum and maximum voltages of the analog input voltage. Two reference voltage inputs, Vref+ (pin5 on the PIC18F4321) and Vref- (pin 4 on the PIC18F4321) are included in the PIC18F4321. Vref+ is connected to the maximum voltage while Vref- is connected to the minimum voltage. The PIC18F4321 converts the differential voltage (Vref+ - Vref-) into 8-bit or 10-bit number. The values of Vref+ and Vref- are selected by programming a register in the PIC18F (to be discussed later). Common values of Vref+ and Vref- are +5VDC and 0V. When 8-bit ADC is selected via programming, 5V will be converted to 255 in decimal (FFH) and 0V to 0.

ADC channels The ADC hardware module is connected to several channels via a multiplexer. The multiplexer allows multiple analog inputs to be connected to ADC for conversion. Each channel can be connected to an analog voltage to be converted into binary by the ADC. One of these channels can be selected via programming, and the multiplexer connects the selected channel to the ADC module. The PIC18F includes 13 channels (AN0 through AN12).

Acquisition time When an ADC channel is selected, the voltage at this channel is used to charge an internal capacitor. It takes a certain time for the capacitor to get completely charged to the voltage at the selected channel. This charging time is called "acquisition time". The PIC18F ADC acquisition time can be programmed by the user. Once the acquisition time is reached, the PIC18F disconnects the selected channel from the source, and then starts the conversion. The acquisition time depends on factors such as source impedance and Vdd (Drain voltage).

ADC clock The ADC clock that is used to perform the conversion is called the period of the ADC clock (T_{AD}). This is the same as the time required to convert one bit.

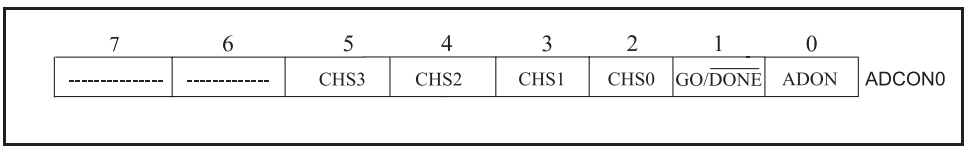
The PIC18F4321 contains an on-chip Analog-to-Digital converter (ADC) module with 13 channels (AN0-AN12). An analog input can be selected as an input on one of these

13 channels, and can be converted to a corresponding 8-bit or 10-bit digital number. Three 8-bit control registers (ADCON0 through ADCON2) along with two 8-bit data registers (ADRESH, ADRESL) are used in the ADC conversion process. Since these five registers are special function registers with predefined addresses in the PIC18F4321 (Appendix E), the programmer does not need to use their numerical addresses. The register names ADCON0, ADCON1, ADCON2, ADRESH, and ADRESL can be used while writing a PIC18F assembly or a C program using the MPLAB assembler or C18 compiler.

ADCON0 register The ADCON0 register can be loaded with data to select an analog input channel, start the conversion, check whether the ADC conversion is complete, and enable or disable the ADC module.

For example, consider the ADCON0 register shown in Figure 10.16. The ADCON0 register can be programmed to select one of 13 channels using bits CHS3 through CHS0 (bits 5 through 2). The conversion can be started by setting the GO/$\overline{\text{DONE}}$ (bit 1) to 1. Once the conversion is completed, this bit is automatically cleared to 0 by the PIC18F4321. For example, writing 01H into ADCON0 will select channel 0 (CHS3 CHS2 CHS1 CHS0 = 0000), make ADC idle (GO/$\overline{\text{DONE}}$ = 0), and then enable the ADC module.

This can be accomplished using the following PIC18F assembly language instruction sequence:



bit 7-6 **Unimplemented:** Read as '0'
bit 5-2 **CHS3:CHS0:** Analog Channel Select bits
0000 = Channel 0 (AN0)
0001 = Channel 1 (AN1)
0010 = Channel 2 (AN2)
0011 = Channel 3 (AN3)
0100 = Channel 4 (AN4)
0101 = Channel 5 (AN5)
0110 = Channel 6 (AN6)
0111 = Channel 7 (AN7)
1000 = Channel 8 (AN8)
1001 = Channel 9 (AN9)
1010 = Channel 10 (AN10)
1011 = Channel 11 (AN11)
1100 = Channel 12 (AN12)
1101 = Unimplemented
1110 = Unimplemented
1111 = Unimplemented

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

bit 0 **: ADON :** ADC On bit
1 = ADC module is enabled
0 = ADC module is disabled

FIGURE 10.16 ADCON0 (ADC Control Register0)

```
MOVLW     01H          ; Load 01H into ADCON0
MOVWF     ADCON0
```

The following C statement will also accomplish this:

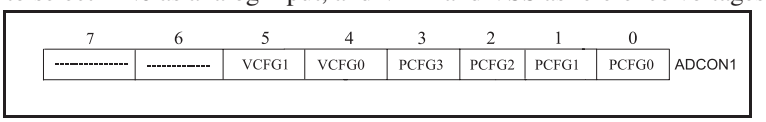
```
ACON0 = 01H;
```

After initializations of ADCON1 and ADCON2 (to be discussed later), the ADC can be started by setting the "GO/ $\overline{\text{DONE}}$" bit (bit 1 in ADCON0) to one. During the conversion process, GO/ $\overline{\text{DONE}}$ = 1. Upon ADC completion, this bit is cleared to 0. Both polled and interrupt I/O can be used to obtain the converted 8-bit or 10-bit binary value into ADRESH:ADRESL register pair (to be discussed later).

ADCON1 register The ADCON1 register can be initialized to specify reference voltages, and configure AN0 through AN12 pins as analog inputs or digital I/O.

The ADCON1 register, shown in Figure 10.17, configures the functions of the port pins as Analog (A) input or Digital (D) I/O. The table shown in Figure 10.17 shows how the port bits are defined as analog or digital signals by programming the PCFG3 through PCFG0 (bits 3 through 0) of the ADCON1 register. Note that in the PIC18F4321, AN0 through AN12 signals are multiplexed (shared) with other signals such as digital I/O.

For example, the following PIC18F assembly language instruction sequence can be used to select AN0 as analog input, and VDD and VSS as reference voltages:



bit 7-6 **Unimplemented:** Read as '0'
bit 5 **VCFG1:** Voltage Reference Configuration bit (VREF- source)
1 = VREF- (AN2)
0 = VSS
bit 4 **VCFG0:** Voltage Reference Configuration bit (VREF+ source)
1 = VREF+ (AN3)
0 = VDD
bit 3-0 **PCFG3:PCFG0:** A/D Port Configuration Control bits

bit 7-6 **Unimplemented**: Read as '0'
bit 5 **VCFG1:** Voltage Regerence Configuration bit (VREF-source)
1 = VREF- (AN2)
0 = VSS
bit 4 **VCFG0:** Voltage Reference Configuration bit (VREF + source)
1 = VREF + (AN3)
0 = VDD

bit 3-0 (decimal) in Column 1: **PCFG3:PCFG0:**A/D Port Configuration Control bits

	AN12	AN11	AN10	AN9	AN8	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0
0	A	A	A	A	A	A	A	A	A	A	A	A	A
1	A	A	A	A	A	A	A	A	A	A	A	A	A
2	A	A	A	A	A	A	A	A	A	A	A	A	A
3	D	A	A	A	A	A	A	A	A	A	A	A	A
4	D	D	A	A	A	A	A	A	A	A	A	A	A
5	D	D	D	A	A	A	A	A	A	A	A	A	A
6	D	D	D	D	A	A	A	A	A	A	A	A	A
7	D	D	D	D	D	A	A	A	A	A	A	A	A
8	D	D	D	D	D	D	A	A	A	A	A	A	A
9	D	D	D	D	D	D	D	A	A	A	A	A	A
10	D	D	D	D	D	D	D	D	A	A	A	A	A
11	D	D	D	D	D	D	D	D	D	A	A	A	A
12	D	D	D	D	D	D	D	D	D	D	A	A	A
13	D	D	D	D	D	D	D	D	D	D	D	A	A
14	D	D	D	D	D	D	D	D	D	D	D	D	A
15	D	D	D	D	D	D	D	D	D	D	D	D	D

A = Analog input D = Digital I/O

FIGURE 10.17 ADCON1 (A/D Control Register 1)

```
MOVLW     0x0D;     Move 0DH to WREG
MOVWF     ADCON1;   Move WREG to ADCON1
```

This is equivalent to the C statement:

```
ADCON1 = 0x0D;
```

From Figure 10.17, initializing ADCON1 with 0DH will mean that reference voltages are selected as follows: Vref+ = VDD and Vref- = VSS since bits 4 and 5 of the ADCON1 register are 00. If VDD and VSS are connected to 5V and ground, then Vref+ and Vref- will be 5V and 0V respectively. Also, since bits 3 through 0 are 0xD (13 decimal), pins AN0 and AN1 of the PIC18F4321 are configured as analog inputs and hence, analog inputs can be connected to these pins. On the other hand, pins AN2 through AN12 of the PIC18F4321 are configured as digital I/O. This can be verified using the table in Figure 10.17.

ADCON2 The ADCON2 register, shown in Figure 10.18, configures the ADC clock source, sets up acquisition time, and specifies justification**.** After conversion, the binary output of the ADC is placed in a 16-bit register (two 8-bit register pair) ADRESH:ADRESL.

The ADFM bit (bit 7) in the ADCON2 register is used to specify whether the converted result is "Left Justified" or "Right Justified". By clearing the ADFM bit in ADCON2 register, the converted result is interpreted as "Left Justified". The left justified format provides an 8-bit result in the ADRESH register; the contents of ADRESL are discarded. On the other hand, the converted result can be interpreted as "Right Justified" by setting the ADFM bit in the ADCON2 register to one. The right justified format provides the 10-bit converted result in ADRESH:ADRESL with the upper two bits in bits 1, 0 of the ADRESH and lower eight bits in ADRESL; the upper six bits of ADRESH are discarded.

Bits 3 through 5 of the ADCON2 register are used to select ADC acquisition time. This can be selected by multiplying T_{AD} by 2, 4, 6, 8, 12, 16, and 20. According to the PIC18F4321 data sheet, a minimum value for acquisition time (T_{acq}) of 2.4 μsec is recommended.

Bits 0 through 2 of the ADCON2 register are used to specify the ADC clock. In the PIC18F, the ADC requires 11 T_{AD} to perform a 10-bit conversion. This can be obtained from the CPU clock (F_{OSC}) after dividing it by 2, 4, 8, 16, 32, 64.

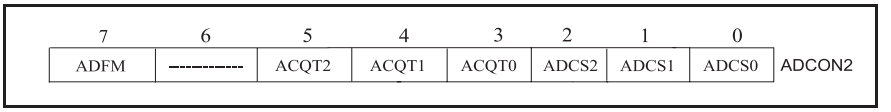
According to Microchip PIC18F4321 data sheet, for correct A/D conversion, T_{AD} must be as short as possible, but greater than the minimum requirement of 0.7 microseconds.

As an example, if a 1-MHz default clock is used on the PIC18F4321, T_{OSC}= 1/(1MHz) = 1 microsecond. Using a prescaler of 2,

T_{AD} = 2 x T_{OSC}= 2 x 1 μsec = 2 μsec which is more than the minimum requirement of 0.7 microsecond. Note that dividing the frequency F_{osc} by a constant is equivalent to multiplying the period T_{osc} by the same constant.

The division factor can be calculated from the above example for other frequencies. Next, acquisition time (T_{acq}) will be calculated. Acquisition time can be selected in the PIC18F via programming to one of the following values: $2T_{AD}$, $4T_{AD}$, $6T_{AD}$, $8T_{AD}$, $12T_{AD}$, $16T_{AD}$, and $20T_{AD}$. If $2T_{AD}$ is used , then T_{acq} = 2 x T_{AD}= 2 x 2 μsec = 4 μsec which is more than the minimum value of 2.4 μsec**.** Hence, the timing requirements T_{AD} and T_{acq} are satisfied.

Figure 10.19 shows a simplified block diagram of the PIC18F4321 on-chip A/D converter. The figure shows how 13 channels (AN0 through AN12) are connected to the PIC18F4321 through a 13x1 multiplexer. One of these 13 channels are selected via



bit 7 **ADFM:** ADC Result Format Select bit
1 = Right justified; 10-bit result in low two bits of ADRESH and in all eight bits of ADRESL
0 = Left justified; 8-bit result in ADRESH, and the contents of ADRESL are ignored. Used for 8-bit conversion.
bit 6 **Unimplemented:** Read as '0'
bit 5-3 **ACQT2:ACQT0:** ADC Acquisition Time Select bits
111 = 20 TAD
110 = 16 TAD
101 = 12 TAD
100 = 8 TAD
011 = 6 TAD
010 = 4 TAD
001 = 2 TAD
000 = 0 TAD**(1)**
bit 2-0 **ADCS2:ADCS0:** A/D Conversion Clock Select bits
111 = FRC (clock derived from A/D RC oscillator)**(1)**
110 = FOSC/64
101 = FOSC/16
100 = FOSC/4
011 = FRC (clock derived from A/D RC oscillator)**(1)**
010 = FOSC/32
001 = FOSC/8
000 = FOSC/2
Note 1: If the ADC FRC clock source is selected, a delay of one TCY (instruction cycle) is added before the ADC clock starts. This allows the SLEEP instruction to be executed before starting a conversion.

FIGURE 10.18 ADCON2 (ADC Control Register 2)

programming bits CHS3 through CHS0 of the ADCON0 register. For example, writing 0100 at bits CHS3 through CHS0 of the ADCON0 register will connect the analog voltage connected to AN4 to the PIC18F4321 on-chip ADC for conversion. Also, programming bits VCFG1 and VCFG0 of the ADCON1 register will select the reference voltages.

Polled vs. Interrupt-driven ADC The A/D conversion can be performed using polled I/O or interrupt I/O. Using polled I/O, the GO/$\overline{\text{DONE}}$ bit in the ADCON0 register can be polled in a loop in order to check for completion of the ADC conversion. The following PIC18F assembly language instruction sequence can be used to accomplish this:

```
        BSF     ADCON0, GO;  Start ADC conversion
CONV    BTFSC   ADCON0, GO;  Wait until ADC conversion is done
        BRA     CONV
        ------
        ------
```

The above PIC18F is equivalent to the following C-code:

```
ADCON0bits.GO = 1;                  // Start ADC
while (ADCON0bits.DONE ==1); // Wait until ADC conversion is DONE
-----
-----
```

Note that in the above C-code, "GO" in the statement, "ADCON0bits.GO = 1;" is used to start the ADC conversion while "DONE" in the statement, "while (ADCON0bits.DONE ==1);" rather than " while (ADCON0bits.GO ==1);" is used for polling the GO/$\overline{\text{DONE}}$ bit (bit 1 of ADCON0). Both statements will work with the C18 compiler since they have the same bit number in ADCON0. The "while (ADCON0bits.DONE ==1);" is

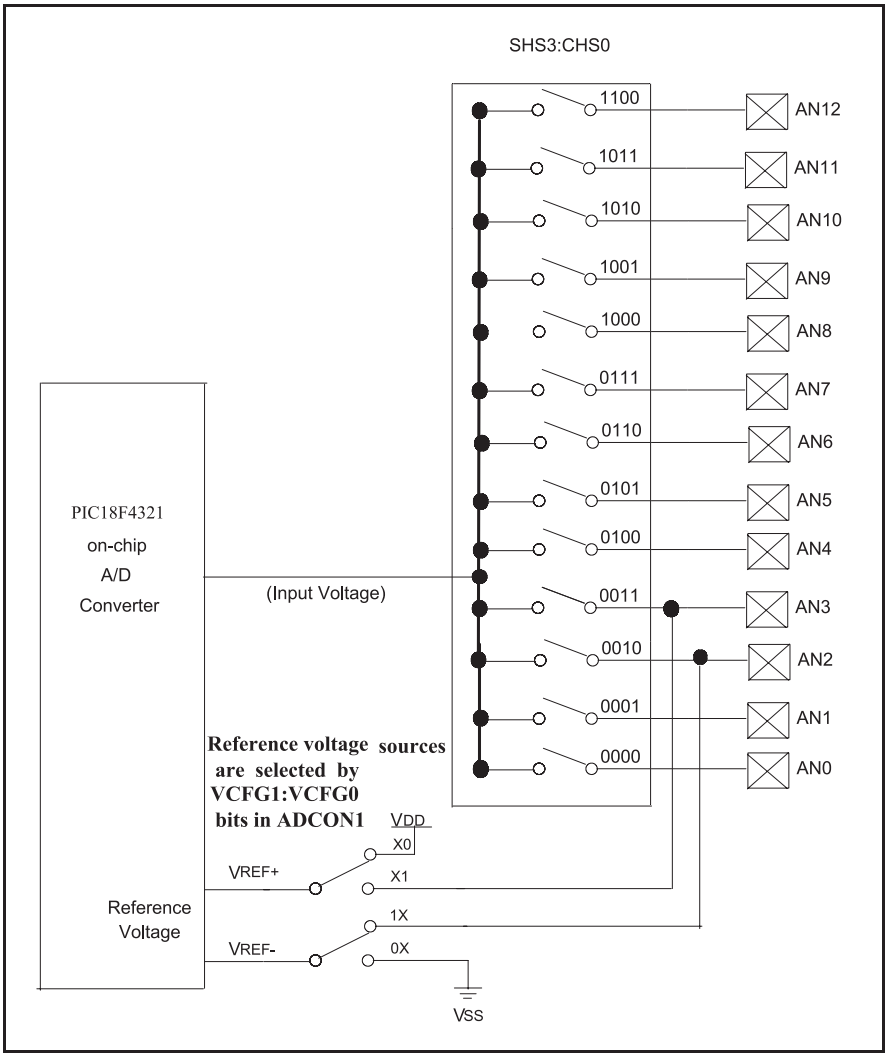


FIGURE 10.19 Block diagram of the PIC18F4321 A/D

used for polling in order to indicate that "wait until ADC conversion is DONE". This is used for better readability.

Figure 10.20 shows a simplified diagram for interrupt-driven ADC.

For interrupt I/O, from Figure 10.20, the following steps need to be followed via programming**:**

1. Set the ADIE bit in the PIE1 register to one. This will enable the ADC interrupt.
2. Clear the ADIF bit in the PIR1 register to zero. Note that the ADIF bit is automatically set to one by the processor after interrupt occurs upon completion of the A/D conversion..
3. Set the PEIE bit in the INTCON register to one. This will enable the peripheral interrupt. Note that ADC is considered as a peripheral device.

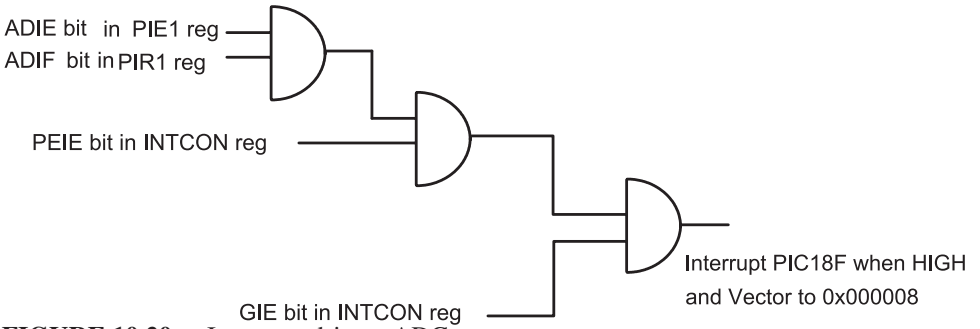


FIGURE 10.20 Interrupt-driven ADC

4. Set the GIE bit in the INTCON register to one.
5. Start the A/D conversion by setting the GO/$\overline{\text{DONE}}$ bit in the ADCON0 register to one.
6. Wait for interrupt to occur, indicating completion of A/D conversion.

The above steps can be accomplished using the following PIC18F assembly language instruction sequence:

```
        BSF PIE1 , ADIE       ; Enable ADC interrupt
        BCF PIR1, ADIF        ; Clear ADC interrupt flag bit
        BSF INTCON, PEIE      ; Enable PIC18F interrupts
        BSF INTCON, GIE       ; Enable global interrupt
        BSF ADCON0, GO        ; Start ADC
WAIT    BRA WAIT              ; Wait for conversion to complete
```

This is equivalent to the following C-language statements:

```
PIE1bits.ADIE = 1;      // Set ADIE bit in PIR1 register to one
PIR1bits.ADIF = 0;      // Clear ADIF bit in PIR1 register to zero
INTCONbits.PEIE = 1;    // Enable peripheral interrupt
INTCONbits.GIE  = 1;    // Enable global interrupt
ADCON0bits.GO = 1;      // Start A/D conversion
while (1);              // Wait for conversion to complete
```

As soon as A/D conversion is completed, the ADIF bit in the PIR1 register is automatically set to one. The PIC18F4321 completes execution of the current instruction, and then branches to address 0x08 (high priority interrupt address vector upon power-on reset). Note that the low priority can be assigned to ADC interrupt by clearing the ADIP (ADC interrupt priority bit) in the IPR1 register (Page 100 of the PIC18F4321 data sheet---can be downloaded from the Internet using Google) to zero. In that case, the interrupt address vector will be 0x018.

Converting the 8-bit contents of ADRESH into Voltage Note that the maximum decimal value that can be accommodated in 8 bits of ADRESH is 255_{10} (FF_{16}). Hence, the maximum voltage of 5V will be equivalent to 255_{10}. This means that 1 volt = 51 (decimal); this factor of 51 will be used to convert the 8-bit ADC output in the ADRESH into analog voltage. Suppose that the decimal equivalent of the 8-bit ADC output in ADRESH register is *X*, the integer part of the voltage is *Y* (after converting to Voltage), and the fractional part (after converting to Voltage) is *Z*. Since *X* is 8-bit, its maximum value in decimal is 255. Hence, 5V is equivalent to 255 in decimal.

Therefore, voltage equivalent of $X = 5$ x (X/255)
$= X/51$
= Quotient + Remainder

Note that Quotient is the integer part, Y of the voltage. The fractional part Z can be calculated from the Remainder as follows: $Z =$ (Remainder/51) x 10 which is approximately (Remainder/5). This will provide the result up to one digit after the decimal point.

For example, suppose that the decimal equivalent of the 8-bit output of A/D is 200. $Y = 200/51 = 3$ with Remainder 47. Hence, $Z = (47/5) = 9$ with remainder 2. This will provide a voltage of 3.9. However, if more fractional digits are desired, they can be obtained as follows. For example, the second fractional digit can be obtained from (2/51) x 100 = 4 (approximately). The final result will be 3.94.

Example 10.12 A PIC18F4321 microcontroller shown in Figure 10.21 with 1MHz internal clock is used to implement a voltmeter to measure voltage in the range 0 to 5 V and display the result in two decimal digits: one integer part and one fractional part. Using both polled I/O and interrupt I/O,

(a) Write a PIC18F assembly language program to accomplish this.

(b) Write a C language program to accomplish this.

Solution

In order to design the voltmeter, the PIC18F4321 on-chip A/D converter will be used. Three registers ADCON0-ADCON2 need to be configured. In ADCON0, analog input AN0 is designated as the analog signal to be converted. Hence, CHS3-CHS0 bits (bits 5-2) are programmed as 0000 to select channel 0 (AN0). The ADCON0 register is also used to enable the A/D, start the A/D, and then check the "End of conversion" bit. In the PIC18F assembly language program provided below, the ADCON0 is loaded with 0x01 which will select AN0, and enable A/D.

The reference voltages (0V and 5V) are chosen by programming the ADCON1 register. In this example, V_{DD} (by clearing bit 4 of ADCON1 to 0), and V_{SS} (by clearing bit 5 of ADCON1 to 0) will be used. Note that V_{DD} and V_{SS} already connected to the

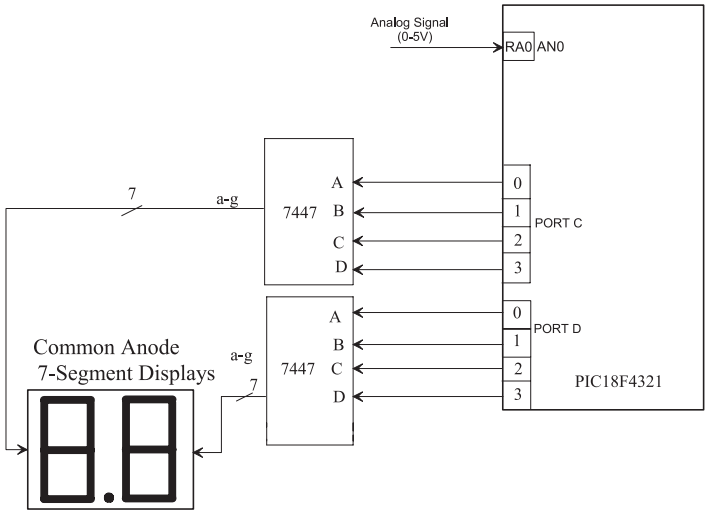


FIGURE 10.21 Figure for Example 10.12

PIC18F4321. The ADCON1 register is also used to configure AN0 (bit 0 of PORT A) as an analog input by writing 1101 (13 decimal in Figure 10.17) at PCFG3-PCFG0 (bits 3-0 of ADCON1). Note that there are several choices to cofigure AN0 as an analog input. In the program, the ADCON1 is loaded with 0x0D which will select VSS and VDD as reference voltage sources, and AN0 as analog input.

In the program, the ADCON2 is loaded with 0x08 which will provide the 8-bit result left justified, select 2 TAD, and select Fosc/2. Note that proper selection of T_{AD} and T_{acq} is discussed in the last section.

The ADCON2 is used to set up the acquisition time, conversion clock, and also if the result is to be left or right justified. In this example, an 8-bit result is assumed. The A/D result is configured as left justified, and therefore, the 8-bit register ADRESH will contain the result. The contents of ADRESL are ignored. As discussed in the last section, the contents of ADRESH are divided by 51 to obtain the integer part of the voltage. The fractional part of the voltage can be obtained by dividing the remainder by 5.

The final result will be in decimal with one integer digit (Integer part) followed by one digit after decimal point (Fractional part). Both integer and fractional parts of the result will be outputted to two 7447's (BCD to 7-segment decoder) in order to display them on two seven-segment displays arranged in a row as shown in Figure 10.21. Note that both integer and fractional parts are in BCD (0-9) since the voltage is from 0 to 5V.

(a)

Using polled I/O, the PIC18F assembly language program for the voltmeter is provided in the following:

```
                INCLUDE   <P18F4321.INC>
D0              EQU       0x30          ; contains data for right (fractional) 7-seg
D1              EQU       0x31          ; contains data for left (integer) 7-seg
ADCONRESULT     EQU       0x34          ; contains 8-bit A/D result
                ORG       0x100         ; starting address of the program
                MOVLW     0x10          ; Initialize STKPTR to 0x10 (arbitrary value)
                MOVWF     STKPTR        ; since subroutines are used
                CLRF      TRISC         ; Configure PortC  as output
                CLRF      TRISD         ; Configure PortD  as output
                SETF      TRISA         ; Configure PortA  as input
                MOVLW     0x01
                MOVWF     ADCON0        ; Select AN0 for input and enable ADC
                MOVLW      0x0D
                MOVWF     ADCON1        ; Select VDD and VSS as reference
                                        ; voltages and AN0 as analog input.
                MOVLW     0x08
                MOVWF     ADCON2        ; Select Left justified, 2TAD and Fosc/2
START           BSF       ADCON0, GO          ; Start A/D conversion
INCONV          BTFSS     ADCON0, DONE        ; Wait until A/D conversion is done
                BRA       INCONV
                MOVFF     ADRESH,ADCONRESULT  ; Move ADRESH of result
                                              ; into ADCONRESULT register
                CALL      DIVIDE        ; Call the divide subroutine
                CALL      DISPLAY       ; Call display subroutine
                BRA       START
```

```
DIVIDE      CLRF    D0              ; Clears  D0
            CLRF    D1              ; Clears D1
            MOVLW   D'51'           ; #1 Load 51 into WREG
EVEN        CPFSEQ  ADCONRESULT     ; #2
            BRA     QUOTIENT        ; #3
            INCF    D1, F           ; #4
            SUBWF   ADCONRESULT, F ; #5
QUOTIENT    CPFSGT  ADCONRESULT     ; #6 Checks if ADCONRESULT
                                    ; still greater than 51
            BRA     DECIMAL         ; #7
            INCF    D1, F           ; #8 increment D1 for each time
                                    ; ADCONRESULT is greater than 51
            SUBWF   ADCONRESULT, F ; #9 Subtract 51 from
                                    ; ADCONRESULT
            BRA     EVEN            ; #10
DECIMAL     MOVLW   0x05            ; #11
REMAINDER   CPFSGT  ADCONRESULT     ; #12 Checks if ADCONRESULT
                                    ; greater than 5
            BRA     DIVDONE         ; #13
            INCF    D0, F           ; #14 Increment D0 each
            SUBWF   ADCONRESULT, F  ; #15 Subtract 5
                                    ; from ADCONRESULT
            BRA     REMAINDER
DIVDONE     RETURN                  ; #16
DISPLAY     MOVFF   D1, PORTC       ; #17 Output D1  on integer 7-seg

            MOVFF   D0, PORTD       ; #18 Output D0  on fractional 7-seg
            RETURN
            END
```

In the above, since the PIC18F does not have any unsigned division instruction, a subroutine called DIVIDE is written to perform unsigned division using repeated subtraction. In the DIVIDE subroutine, the output of the A/D contained in the ADCONRESULT register is subtracted by 51. Each time the subtraction result is greater than 51, the contents of register D1 (address 0x31) is incremented by one, which would yield the integer part of the answer. Once the contents of the ADCONRESULT reaches a value below 51, the remainder part of the answer is determined. This is done by subtracting the number in ADCONRESULT subtracted by 5. Each time the subtraction result is greater than 5, register D0 (address 0x30) is incremented by one. Finally, the integer value is placed in D1 and the remainder part is placed in D0. Now the only task left is to display the result on the 7-segment display.

The # symbol along with a number in the comment field is used in some of the lines in the above program in order to explain the program logic. Line#1 moves 51 (decimal) into WREG. The CPFSEQ at Line#2 compares the A/D's 8-bit result in ADCONRESULT with 51 for equality. Suppose that the analog input voltage at AN0 is one volt, which is 51 in decimal. Since [WREG] = [ADCONRESULT] = 51, the program branches to line #4, and increments [D1] by 1, storing 1 in D1. The SUBWF ADCONRESULT, F at Line #5 subtracts [WREG] from [ADCONRESULT] and stores the result in ADCONRESULT.

Since the subtraction result is 0 in this case, '0' is stored in ADCONRESULT. The CPFSGT ADCONRESULT instruction at Line #6 compares [WREG] with [ADCONRESULT] to check whether [ADCONRESULT] > [WREG]. Since [WREG] = 51 and [ADCONRESULT] = 0, the program executes BRA DECIMAL at Line #7, and branches to label DECIMAL at Line #11 where 5 is moved into WREG.

The CPFSGT ADCONRESULT at Line #12 is then executed to check whether [ADCONRESULT] > [WREG]. Since [WREG] = 5 and [ADCONRESULT] = 0, the program executes BRA DIVDONE at Line #13, and branches to label DIVDONE at Line #16 where the RETURN instruction is executed. The program returns to the "CALL DISPLAY"-- one instruction after "CALL DIVIDE". The program pushes the address of the next instruction "BRA START" onto the hardware stack, and executes the subroutine called DISPLAY (Line #17). The instruction "MOVFF D1, PORTC" at Line #17 outputs [D1] = 0x01 to PORTC. Hence, '1' is displayed on the integer display.

Note that the BCD number '1' of the integer part is contained in low four bits of D1 which are output to the DCBA inputs of the integer 7447 to display a '1' on the left (integer) 7-segment display. The instruction "MOVFF D0, PORTD" outputs [D0] = 0x00 to PORTD, and a '0' is displayed on the right (fractional) display. Outputting to integer and fractional displays using instructions in sequence are executed so fast by the PIC18F that the displays appear to human eyes at the same time. Finally, "1.0" indicates that 1.0 volt is displayed on the two seven-segment displays.

Next, suppose that the decimal value contained in the ADCONRESULT (A/D converter's output) is 200 (decimal) which is equivalent to 3.9 volts. Line#1 moves 51 (decimal) into WREG. The CPFSEQ at Line#2 compares the A/D's 8-bit result in ADCONRESULT with 51 for equality. Since [WREG] = 51, and [ADCONRESULT] = 200, the program executes the instruction "BRA QUOTIENT" at line #3, and branches to Line #6. The CPFSGT ADCONRESULT instruction at Line #6 compares [WREG] with [ADCONRESULT] to check whether [ADCONRESULT] > [WREG]. Since [WREG] = 51 and [ADCONRESULT] = 200, the program branches to Line #8, and increments [D1] by 1, and stores the result in D1. The instruction "SUBWF ADCONRESULT, F" at Line #9 is then executed, and [WREG] is subtracted from [ADCONRESULT], and the result is stored in ADCONRESULT. Since [WREG] = 51 and [ADCONRESULT] = 200, the subtraction result 149 will be stored in ADCONRESULT. The instruction "BRA EVEN" at Line #10 is executed next. The program branches to label EVEN at line #2 to execute the instruction "CPFSEQ ADCONRESULT", and the loop is repeated until the result of subtraction in ADCONRESULT is less than 51. This will happen in this case when [D1] = 3, and [ADCONRESULT] = 47 = subtraction result of "[ADCONESULT]- [WREG]" after going through the loop 3 times. As soon as [ADCONRESULT] < [WREG], the instruction "BRA DECIMAL' at Line #7 is executed where the fractional part '9' is determined in the same manner as the last example. The rest of the logic is very similar to the last example. Finally, "3.9" will be displayed on the two seven-segment displays.

Using interrupt I/O, the PIC18F assembly language program for the voltmeter is provided in the following:

```
                INCLUDE <P18F4321.INC>
                D0 EQU 0x30
                D1 EQU 0x31
ADCONRESULT     EQU 0x34
                ORG 0x00; Reset
```

```
                BRA MAIN_PROG    ; Main Program
                ORG 0x100
MAIN_PROG       MOVLW 0x10       ; Initialize STKPTR
                MOVWF STKPTR     ; Move 10H to STKPTR
                CLRF TRISC       ; Configure PortC to be output
                CLRF TRISD       ; Configure PortD to be output
                MOVLW 0x01       ; Move 01H to WREG
                MOVWF ADCON0     ; Select AN0 for input and enable A/D
                MOVLW 0x0D       ; Set Vdd and Vss as reference voltages
                MOVWF ADCON1     ; and AN0 as analog input
                MOVLW 0x08       ; Select Left justify, 2TAD, FOSC/2
                MOVWF ADCON2
                BSF PIE1 , ADIE  ; Enable ADC interrupt
                BCF PIR1, ADIF   ; Clear ADC interrupt flag bit
                BSF INTCON, PEIE ; Enable PIC18F interrupts
                BSF INTCON, GIE
                BSF ADCON0, GO   ; Start ADC
WAIT            BRA WAIT         ; Wait for conversion to be complete
                BRA MAIN_PROG    ; Return to Main Program
                                 ; Interrupt Service Routine
                ORG 0x0008       ; Interrupt address vector
                BCF PIR1,ADIF    ; Clear ADC interrupt flag bit
                MOVFF ADRESH, ADCONRESULT      ; Store ADRESH
                CALL DIVIDE      ; Calls divide subroutine
                CALL  DISPLAY    ; Calls display subroutine
                RETFIE           ; Enable interrupt and return to main
DIVIDE          CLRF D0          ; Clears D0
                CLRF D1          ; Clears D1
                MOVLW D'51'      ; Load 51 into WREG
EVEN            CPFSEQ ADCONRESULT; Compare with 51 for equality
                BRA QUOTIENT     ; Branch to QUOTIENT
                INCF D1, F       ; Increment [D1] by 1, store in D1
                SUBWF ADCONRESULT, F      ; Subtract [WREG] from
                                          ; [ADCONRESULT],
                                 ; store in ADCONRESULT
QUOTIENT CPFSGT ADCONRESULT  ; Checks if ADCONRESULT is still greater than 51
                BRA DECIMAL      ; Branch to DECIMAL
                INCF D1, F       ; Increment D1 for each time
                                 ; ADCONRESULT is greater than 51
                SUBWF ADCONRESULT, F      ; Subtract 51 from ADCONRESULT
                BRA EVEN         ; Branch to EVEN
DECIMAL         MOVLW 0x05       ; Move 5 into WREG
REMAINDER CPFSGT ADCONRESULT  ; Checks if ADCONRESULT is greater
                                 ; than 5
                BRA DIVDONE      ; Branch to DIVDONE
                INCF D0, F       ; Increment D0
                SUBWF ADCONRESULT, F      ; Subtract 5 from ADCONRESULT
                BRA REMAINDER    ; Branch to REMAINDER
```

```
DIVDONE     RETURN              ; Execute RETURN instruction
DISPLAY     MOVFF D1, PORTC     ; Output D1 on integer 7-seg
            MOVFF D0, PORTD     ; Output D0 on fractional 7-seg
            RETURN
            END
```

(b)

Using polled I/O, the C- program for the Voltmeter is provided in the following:

```
#include <P18F4321.h>
unsigned int result; //Initialize variable
void main ( )
{
TRISC = 0;                      // Configure  Port C as output
TRISD = 0;                      // Configure PORT D as output
ADCON0 = 0x01;                  // Select   channel 0 for AN0 and enable ADC module
ADCON1 = 0x00;                  // Select reference voltages  0V and 5V, enable AN0
ADCON2 = 0x08;                  // Left justified, 8-bit result in ADRESH, 2TAD, FOSC/2
ADCON0bits.GO = 1;              // Start the ADC
while(ADCON0bits.DONE == 1); // Wait until conversion is complete
result = ADRESH;                // Move the converted  8-bit data into result
PORTC = result/51;
PORTD = (result%51)/5;
ADCON0bits.GO = 1;              // Start A/D converter again
}
```

Using interrupt I/O, the C- program for the Voltmeter is provided in the following :

```
#include <p18f4321.h>
unsigned int result;
void convert (void);
#pragma code ADCINT=0x08  // At interrupt, code jumps here
void ADCINT (void)
{ _asm

GOTO  convert
_endasm
}
#pragma code // End code
void main()
{
TRISC = 0;                      // Configure Port C as output
TRISD = 0;                      // Configure Port D as output
PIE1bits.ADIE = 1;              // enable ADC interrupt
PIR1bits.ADIF = 0;              // clear ADC interrupt flag
INTCONbits.PEIE = 1;            // enable peripheral interrupt
INTCONbits.GIE = 1;             // enable global interrupt
ADCON0 = 0x01;                  // Select   channel 0 for AN0 and enable ADC module
ADCON1 = 0x00;                  // Select reference voltages  0V and 5V, enable AN0
ADCON2 = 0x08;                  // Left justified, 8-bit result in ADRESH, 2TAD, FOSC/2
```

```
ADCON0bits.GO = 1;          // Start A/D conversion
while(1);                   // Wait for interrupt
}
#pragma interrupt convert
void convert (void)
{
PIR1bits.ADIF = 0;          // Clear ADIF to 0
result = ADRESH;
PORTC = result/51;
PORTD = (result%51)/5;
ADCON0bits.GO = 1;          // Start A/D conversion again
}
```

10.2.2 Interfacing an external D/A (Digital to Analog) Converter using C

As discussed before, most microcontrollers such as the PIC18F4321 do not have any on-chip D/A converter (or sometimes called DAC). Hence, an external D/A converter chip is interfaced to the PIC18F4321 to accomplish this function. Some microcontrollers such as the Intel/Analog Devices 8051 include an on-chip D/A converter. In order to illustrate the basic concepts associated with interfacing a D/A coverter to the PIC18F, a typical D/A converter such as the Maxim MAX5102 is interfaced to the PIC18F4321.

Example 10.13 Assume the block diagram of Figure 10.22. Write a C program that will input eight switches via PORTD of the PIC18F4321, and output the byte to D0-D7 input pins of the MAX5102 D/A converter. The microcontroller will send appropriate signals to the $\overline{WR}$ and A0 pins so that the D/A converter will convert the input byte to an analog voltage between 0V and 5V, and output the converted voltage on its OUTA pin. Use 1-MHz internal clock.

Solution

The steps for writing a PIC18F C language program for the D/A converter interface of Figure 10.22 is provided in the following:

1. Configure PORTB and PORTC as outputs, and PORTD as an input.
2. Output a LOW to A0 Pin of the D/A via bit 1 of PORTB to select OUTA.

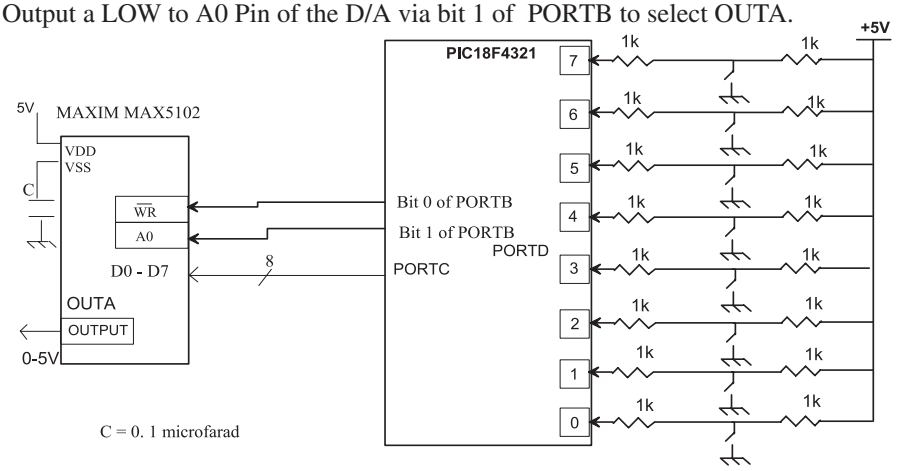


FIGURE 10.22 Figure for Example 10.13

3. Output a LOW to $\overline{WR}$ pin of the D/A via bit 0 of PORTB.
4. Input the switches via PORTD, and output to PORTC.
5. Output a HIGH to $\overline{WR}$ pin of the D/A via bit 0 of PORTB to latch an 8-bit input data for converting to analog voltage. No delay is needed since the program will be written to input one byte of data from the switches.

The C language program is provided below:

```
#include <p18f4321.h>
void main (void)
{
        TRISB=0x00;          // Configure PORTB as output
        TRISC=0x00;          // Configure PORTC as output
        TRISD=0xFF;          // Configure PORTD as input
        PORTBbits.RB1=0;     // Clear A0 to 0 to select OUTA
        PORTBbits.RB0=0;     // Output LOW on  bit 0 of PORTB
        PORTC=PORTD;         // Input switches, output to PORTD
        PORTBbits.RB0=1;     // Latch data for conversion
        while(1);            // Halt
}
```

QUESTIONS AND PROBLEMS

10.1 Find the contents of T0CON register to program Timer0 in 8-bit mode with 1:16 prescaler using the external clock, and inrementing on negative edge.

10.2 Write programs in PIC18F assembly and C:

(a) To initialize Timer0 as an 8-bit timer to provide a time delay with a count of 100. Assume a 4 MHz internal clock with a prescaler value of 1:16.

(b) To generate a square wave with a period of 4 ms on bit 0 of PORTC using a 4 MHz crystal. Use Timer0.

(c) To generate a square wave with a period of 4 ms on bit 7 of PORTD using a 4 MHz crystal. Use Timer1.

(d) To turn an LED ON connected at bit 0 of PORTC with a PR2 value of 200 microseconds. Assume a 4-MHz crystal and a TMR2 prescaler and postscaler of 1:1.

(e) To generate a square wave on pin 3 of PORTC with a 4 ms period using Timer3 in 16-bit mode with a prescaler value of 1:8. Use a 4 MHz crystal.

10.3 Design a PIC18F4321-based digital clock. The clock will display time in hours, minutes, and seconds. Write a C program to accomplish this. Use 1MHz internal clock.

10.4 Using the C18 delay function "10KTCYx()," write a C-program to toggle an LED connected to bit0 of PORTC after 10 sec. Use a 4MHz internal clock.

10.5 Assume PIC18F4321 with Fosc = 1 MHz. Consider the following C program:

```
#include <p18f4321.h>
void    delay(void);
void    main(void){
                TRISBbits.TRISB=0;
                while (1){
                        PORTBbits.RB4^=1;
                        delay( );
                }
}
void    delay( ) {
                T0CON=0x01;
                TMR0H=0xc2;
                TMR0L=0xF7;
                T0CONbits.TMR0ON=1;
                while(INTCONbits.TM0IF = = 0)
                                ;
                T0CONBits.TMR0ON = 0;
                INTCONbits.TMR0IF = 0;
}
```

(a) What type of signal is generated on the RB4 pin? What is its frequency?

(b) What is the frequency of the signal if T0CON is initialized with 0x42, and TMR0H = 0xC2 is deleted from the program?

10.6 Assume PIC18F4321 with 4MHz crystal. For the ADC conversion, calculate T_{AD} with a prescaler of 4 and T_{acq} with prescaler of 2 in the ADCON2 register and then verify whether the conversion is valid.

10.7 Find the contents of ADCON0, ADCON1 and ADCON2 registers to start the ADC with following selections:

- AN5 as analog input.
- $V^{+}_{ref} = V_{dd}$, $V^{-}_{ref} = V_{ss}$
- Fosc/4 , $4T_{AD}$
- Result is left justified
- ADC module is on

10.8 Write a PIC18F assembly language program that will convert an analog signal (0-5V DC) connected to AN9 into a 10-bit number using the PIC18F ADC, divide this 10-bit number by 4, and store the 8-bit result in ADRESL. Discard the remainder. Use 1MHz internal clock.

10.9 Design and develop hardware and software for a PIC18F4321-based system (Figure P10.9) that would measure, compute, and display the Root-Mean-Square (RMS) value of a sinusoidal voltage. The system is required to:

(a) Sample a 5V (zero-to-peak voltage), 60-Hz sinusoidal voltage 128 times.

(b) Digitize the sampled value using the on-chip ADC of the PIC18F4321 along with its interrupt upon completion of conversion signal.

(c) Compute the RMS value of the waveform using the formula, RMS Value = $[SQRT(\sum_{i=1}^{N}((X_n^2)]/N$, where X_n's are the samples and N is the total number of samples**.** Display the RMS value on two seven-segment displays (one for the integer part, and the other for the fractional part).

Design all necessary interfaces. Use one MHz internal clock.

Write a C language program to accomplish the above.

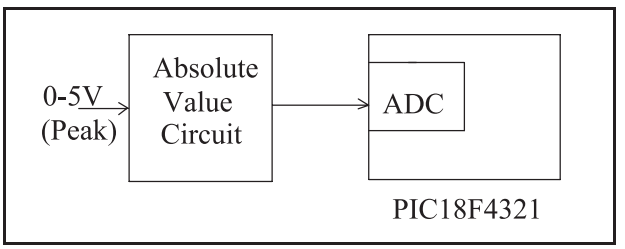


FIGURE P10.9

10.10 *Capacitance meter.* Consider the RC circuit of Figure P10.10. The voltage across the capacitor is Vc (t) = k $e^{-t/RC}$. In one-time constant RC, this voltage is discharged to the value k/e. For a specific value of R, value of the capacitor, C = T/R, where T is the time constant that can be counted by the PIC18F4321 using one MHz internal clock. Design the hardware and software for the PIC18F4321 to charge a capacitor by using a pulse to a voltage of up to 10V peak voltage via an amplifier. The PIC18F4321 will then stop charging the capacitor, measure the discharge time for one time constant, and compute the capacitor value.

(a) Draw a hardware schematic.

(b) Write a C program to accomplish the above.

10.11 Design a PIC18F4321-based system to measure the power absorbed by a 2k resistor (Figure P10.11). The system will input the voltage (V) across the 2k resistor, convert it to an 8-bit input using the PIC18F4321's on-chip A/D converter, then compute the power using V^2/R.

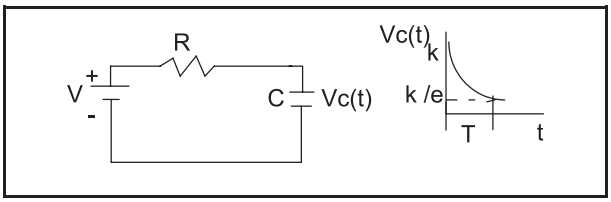


FIGURE P10.10

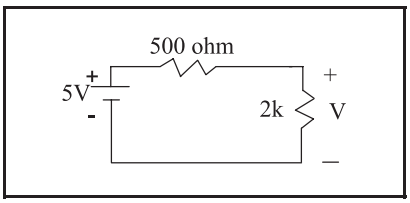


FIGURE P10.11

11
PIC18F CCP AND SERIAL I/O

In this chapter we describe the basics of programming the PIC18F on-chip peripherals using both assembly and C. Topics include PIC18F CCP (Capture/Compare/PWM) and Serial I/O modules.

11.1 PIC18F CCP (Capture/Compare/PWM (Pulse Width Modulation) Module

The CCP module is implemented in the PIC18F4321 as an on-chip feature to provide measurement and control of time based pulse signals. Timer1 through Timer3 are used for this purpose. Timer0 is not used by the CCP module. Any reset will turn the CCP module off.

The PIC18F4321 contains two CCP (Capture/Compare/PWM) modules namely, CCP1 and CCP2. The CCP1 module of the PIC18F4321 is implemented as a standard CCP with enhanced PWM capabilities for better DC motor control. Hence, the CCP1 module in the PIC18F4321 is also called ECCP (Enhanced CCP). The CCP2 module is provided with standard capture, compare, and PWM features. Note that CCP1 and CCP2 are two modules. Hence, one can have CCP1 set up for PWM and CCP2 for capture or compare (or vice versa) since they are two independent modules. It may not be possible to use one CCP module (such as the CCP1) to do capture (or compare) and PWM at the same time. This is because, although the timers may be different (Timer1, Timer3), other logic functions are shared. The CCP1 and CCP2 modules will be referred to as CCPx in the following discussion.

The CCP module can be programmed to one of the following modes:

- Capture mode (CCPx pin to be configured as input via programming)
- Compare mode (CCPx pin to be configured as output via programming)
- PWM mode (CCPx pin to be configured as output via programming)

The PIC18F Capture mode causes the contents of a 16-bit timer (Timer1 or Timer3) to be written in a 16-bit register (CCPRxH:CCPRxL provided with the CCPx module) upon detecting an n^{th} rising or falling edge of a pulse. Capture mode can be used to measure the length of time elapsed between two events. Typical events include detecting every falling edge or every rising edge of a periodic waveform connected to the CCPx pin. By capturing two consecutive rising edges, and then subtracting their values, the period of a waveform can be obtained. Thus, the capture mode can be used to measure the period of an incoming periodic waveform.

The PIC18F Compare mode is very similar to the timer function of a stopwatch. Note that a stopwatch is loaded with a preset time determined by the user. Upon starting the stopwatch, it counts down from the preset value to 0. The PIC18F Compare mode works in a similar manner except that it counts from 0 to the preset value. The PIC18F Compare

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

mode can be used to turn ON or OFF a device connected to the CCPx pin after a specified amount of time. This can be accomplished by outputting a signal (for example, a HIGH or a LOW by programming the CCPxCON rgister) as soon as the specified timer value (Timer1 or Timer3) matches with the 16-bit register value (CCPRxH:CCPRxL). Hence, using the Compare mode, the PIC18F can be programmed to output a signal on the CCPx pin after a certain time delay. As an example, the Compare mode can be used in a chemical plant to turn a pump OFF after filling a tank for 50 seconds.

The PIC18F PWM (Pulse Width Modulation) mode can be used to generate a square wave on the CCPx pin with a user-specified frequency and duty cycle. Note that duty cycle is defined as the percentage of the time the pulse is high in a clock period. Also, as the duty cycle of a PWM signal increases, the average voltage and power provided by the PWM will increase. Because of this feature, the PWM mode can be used in applications such as dimming LEDs and controlling the speed of a brushless DC electric motor. This means that LEDs will be brighter as the PWM duty cycle increases while the speed of the motor will increase as the PWM duty cycle is increased. On the other hand, LEDs will be dimmer with lower PWM duty cycle while the motor will run at a slower speed with lower PWM duty cycle.

Both polled I/O and interrupt I/O can be used with the CCP module. Examples for the CCP1 module using polled I/O will be provided in this section. Note that the CCPxIF flag bit in the PIR1 register is monitored in a wait loop for polled I/O for the Capture and Compare modes while the TMR2IF flag bit in the T2CON register is checked in wait loop for the PWM mode.

11.1.1 CCP Registers

Each CCP module is provided with an 8-bit control register (CCPxCON) shown in Figure 11.1. The CCPxCON (x = 1 or 2) can be used to select one of the three modes namely, Compare or Capture or PWM.

Each CCP module also contains a 16-bit data register (CCPRx). The 16-bit data register, in turn, is comprised of two 8-bit registers: CCPRxL (low byte) and CCPRxH (high byte). This 16-bit data register can operate as a 16-bit Capture register, a 16-bit Compare register or an 8-bit PWM register (CCPRxL) holding the 8-bit decimal part of the duty cycle.

11.1.2 CCP modules and associated timers

The CCP modules utilize Timers 1, 2 or 3, depending on the mode selected. Timer1 and Timer3 are available to modules in Capture and Compare modes, while Timer2 is available for modules in PWM mode. The assignment of a particular timer to a module is determined by the Timer to CCP enable bits in the T3CON register (Figure 10.11). Table 11.1 summarizes the timer assignments to each one of these modes in the CCP1 module.

11.1.3 PIC18F4321 Capture mode

In Capture mode, the CCPRxH:CCPRxL register pair captures the 16-bit value of the TMR1 or TMR3 registers when an event (such as every rising or falling edge) occurs on the corresponding CCPx pin. The event is selected by the mode select bits, CCPxM3:CCPxM0 (bits 3-0 of CCPxCON, Figure 11.1). When a capture is made, the CCPx interrupt flag bit such as CCP1IF in the PIR1 register (Figure 10.8) is set to one; it must be cleared to zero in software. If another capture occurs before the value in register CCPRx is read, the old captured value is overwritten by the new captured value; hence, the old captured value should be saved.

TABLE 11.1 Assignment of Timers for the PIC18F4321 CCP1 module

CCP mode selected	Timer
Capture mode	Timer1 or Timer3
Compare mode	Timer1 or Timer3
PWM mode	Timer2

In Capture mode, the CCPx pin of the PIC18F4321 should be configured as an input by setting the corresponding TRISx direction bit. Also, the timers that are to be used with the capture mode (Timer1 or Timer3) must be running in Timer mode. The timer (Timer1 or Timer3) to be used with each CCP module is selected using the T3CON register (Figure 10.11).

In summary, the following steps can be used to program the PIC18F4321 in capture mode to determine the period of a waveform (assume CCP1; similar procedure for CCP2):

1. Load the CCP1CON register (Figure 11.1) with appropriate data for capture mode.
2. Configure CCP1 as an input pin using the TRISC register. Note that the CCP1 pin is multiplexed with the RC2 pin.
3. Select Timer1 or Timer3 for the capture mode by loading appropriate data into T3CON register (Figure 10.11). T1CON is not used for Timer1 selection. Instead, only T3CON is used for selecting Timer1 or Timer3.



bit 7-6 **Unimplemented:** Read as '0'

bit 5-4 **DCxB1:DCxB0**: PWM Duty Cycle bit 1 and bit 0 for CCP Module x
Capture mode:
Unused.
Compare mode:
Unused.
PWM mode:
These bits are the two Lower two bits (bit 1 and bit 0) of the 10-bit PWM duty cycle. The higher eight bits (DCx9:DCx2) of the duty cycle are found in CCPRxL.

bit 3-0 **CCPxM3:CCPxM0**: CCPx Module Mode Select bits
0000 = Capture/Compare/PWM disabled (resets CCP module)
0001 = Reserved
0010 = Compare mode, toggle output on match (CCPxIF bit is set)
0011 = 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: initialize CCP pin low; on compare match, force CCP pin high (CCPxIF bit is set)
1001 = Compare mode: initialize CCP pin high; on compare match, force CCP pin low (CCPxIF bit is set)
1010 = Compare mode: generate software interrupt on compare match (CCPxIF bit is set, CCP pin reflects I/O state)
1011 = Compare mode: trigger special event, reset timer, start A/D conversion on CCPx match (CCPxIF bit is set)
11xx = PWM mode

FIGURE 11.1 CCPxCON Register

4. Clear the CCP1 flag: bit CCP1IF for CCP1 (Register PIR1 of Figure 10.8) or bit CCP2IF for CCP2 (Register PIR2 of Figure 10.12), after a capture so that the next capture can be made.
5. Clear CCPR1H and CCPR1L to 0.
6. Check CCP1IF flag in PIR1 and wait in a loop until CCP1IF is 1 for the first rising edge. As soon as the first rising edge is detected, start Timer1 using T1CON (or Timer3 using T3CON).
7. Save CCPR1H and CCPR1L in data memory such as REGX and REGY.
8. Clear CCP1IF to 0.
9. Check CCP1IF flag in PIR1 and wait in a loop until CCP1IF is 1 for the second rising edge. As soon as the second rising edge is detected, stop Timer1 (or Timer3).
10. Disable capture by clearing CCP1CON register.
11. Perform 16-bit subtraction: [CCPR1H:CCPR1L] - [REGX:REGY] and store the result in [REGX:REGY].
12. A 16-bit result in register pair [REGX:REGY] will contain the period of the incoming waveform in terms of the number of clock cycles.

Typical applications of the capture mode include:
- measurement of the pulse width of an unknown periodic signal by capturing the subsequent leading (rising) or trailing (falling) edges of a pulse.
- measurement of the period of a signal by capturing two subsequent leading or trailing edges.
- measurement of duty cycle. Note that the duty cycle is defined as (t1/T) x 100 where t1 is the fraction of the time the signal is HIGH in a period, T.

Example 11.1 Assume PIC18F4321.

(a) Write a PIC18F assembly language program at address 0x200 to measure the period (in terms of the number of clock cycles) of an incoming periodic waveform connected at CCP1 pin. Store the result in registers 0x21 (high byte) and 0x20 (low byte). Use Timer3, and capture mode of CCP1.

(b) Write a C language program to measure the period (in terms of the number of clock cycles) of an incoming periodic waveform connected to the CCP1 pin. Use Timer3, and the capture mode of CCP1.

Solution

(a) The PIC18F assembly language program is provided below:

```
        INCLUDE   <P18F4321.INC>
        ORG       0x200
        MOVLW     B'00000101'      ; Select capture mode rising edge
        MOVWF     CCP1CON
        BSF       TRISC, CCP1      ; Configure CCP1 pin as input
        MOVLW     B'01000000'      ; Select TIMER3 as clock source for capture
        MOVWF     T3CON            ; Select TIMER3 internal clock, 1:1 prescale
                                   ; TIMER3 OFF
        CLRF      CCPR1H           ; Clear CCPR1H to 0
        CLRF      CCPR1L           ; Clear CCPR1L to 0
        BCF       PIR1, CCP1IF     ; Clear CCP1IF
```

```
WAIT    BTFSS    PIR1, CCP1IF       ; Wait for the first rising edge
        GOTO     WAIT
        BSF      T3CON, TMR3ON      ; Turn Timer3 ON
        MOVFF    CCPR1L, 0x20       ; Save CCPR1L in 0x20 at 1st rising edge
        MOVFF    CCPR1H, 0x21       ; Save  CCPR1H in 0x21 at 1st rising edge
        BCF      PIR1, CCP1IF       ; Clear CCP1IF
WAIT1   BTFSS    PIR1, CCP1IF       ; Wait for next rising edge
        GOTO     WAIT1
        BCF      T3CON, TMR3ON      ; Turn OFF  Timer3
        CLRF     CCP1CON            ; Disable capture
        MOVF     0x20, W            ; Move 1st Low byte to WREG
        SUBWF    CCPR1L, F          ; Subtract WREG from 2nd low byte
                                    ; Result in 0x20
        MOVF     0x21, W            ; Move 1st HIGH byte to WREG
        SUBWFB   CCPR1H, F          ; Subtract WREG with borrow
                                    ; from 2nd high byte, result in 0x21
        SLEEP                       ; Halt
        END
```

(b)
```
#include<p18f4321.h>
void main(void)
{
    unsigned char FIRST_CCPR1L, FIRST_CCPR1H, HIGH_BYTE, LOW_
BYTE;
    CCP1CON=0x05;                 // Select capture mode rising edge
    TRISCbits.TRISC2=1;           // Configure RC2/CCP1/P1A pin as input
    T3CON=0x40;                   // Select TIMER3 as clock source for capture
    CCPR1H=0x00;                  // Clear CCPR1H to 0
    CCPR1L=0x00;                  // Clear CCPR1L to 0
    PIR1bits.CCP1IF=0;            // Clear CCP1IF

    while(PIR1bits.CCP1IF==0);    // Wait for the first rising edge

    T3CONbits.TMR3ON=1;           // Turn Timer3 ON
    FIRST_CCPR1L=CCPR1L;          // Save CCPR1L in FIRST_CCPR1L at 1st
                                  // rising edge
    FIRST_CCPR1H=CCPR1H;          // Save CCPR1H in FIRST_ at 1st rising edge
    PIR1bits.CCP1IF=0;            // Clear CCP1IF

    while(PIR1bits.CCP1IF==0);    // Wait for next rising edge

    T3CONbits.TMR3ON=0;           // Turn OFF  Timer3
    CCP1CON=0x00;                 // Disable capture
    LOW_BYTE=CCPR1L−FIRST_CCPR1L;         // Low byte of result
    HIGH_BYTE=CCPR1H−FIRST_CCPR1H;        // High byte of result

    while(1);                      // Halt
}
```

As mentioned before, in order to implement an I/O program in PIC18F assembly such as Example 11.1(a) in the laboratory, the following configuration commands should be

inserted using the assembler directive "config" in the PIC18F assembly language program after the "include <p18f4321.inc>" statement as follows:

```
include <p18f4321.inc>
config OSC = INTIO2          ; Select internal oscillator
config WDT = OFF             ; Watch Dog Timer OFF
config LVP = OFF             ; Low Voltage Programming OFF
config BOR = OFF             ; Brown Out Reset OFF
```

Similarly, in order to implement an I/O program using C, such as Example 11.1(b) in the laboratory, the following configuration commands should be inserted in the C-program using the directive "#pragma config" after #include <p18f4321.h> as follows:

```
#include <p18f4321.h>
#pragma config OSC = INTIO2  // Select internal oscillator
#pragma config WDT = OFF     // Watch Dog Timer OFF
#pragma config LVP = OFF     // Low Voltage Programming OFF
#pragma config BOR = OFF     // Brown Out Reset OFF
```

Note that the above applies to all programs in PIC18F assembly and C including polled and interrupt-driven CCP, polled and interrupt-driven SPI and I^2C.

11.1.4 PIC18F4321 Compare mode

In Compare mode, the 16-bit CCPRx (CCPR1H:CCPR1L for CCP1 or CCPR2H: CCPR2L for CCP2) register value is constantly compared to the value in either the TMR1 or TMR3 register. When a match occurs, the CCPx pin of the PIC18F4321 can be:

- driven high
- driven low
- toggled (high-to-low or low-to-high; complemented)
- remain unchanged

The action on the pin is based on the value of the mode select bits (CCPxM3:CCPxM0) in CCPxCON register (Figure 11.1). As soon as a match occurs, the CCPx flag bit, CCPxIF, is set to one. The user must configure the CCPx pin as an output by clearing the appropriate TRIS bit. Timer1 or Timer3 must be running in Timer mode.

Typical applications of the compare mode include activating an event (such as turning a pump ON or OFF) after a certain time delay, generating a pulse train or a waveform with a specific duty cycle.

The following steps can be used to program the PIC18F4321 in compare mode:

1. Load the CCPxCON (Figure 11.1) with appropriate data for compare mode. Configure the CCPx pin of PORTC as an output. Note that CCPx pins are multiplexed with certain bits of PORTC.
2. Load the CCPR1H:CCPRIL (or CCPR2H:CCPR2L) register pair with appropriate values.
3. Select Timer1 (or Timer3) as the clock source in the Compare mode by loading appropriate data into T3CON register. T1CON is not used to select the clock source for Compare mode. Instead, T3CON is used to select Timer1 or Timer3 as the clock source.

4. Initialize Timer1H:Timer1L (or Timer3H:Timer3L) to 0.
5. Clear CCP1IF in PIR1 (or CCP2IF in PIR2) to 0.
6. Start Timer1 using T1CON (or Timer3 using T3CON).
7. Wait in a loop until the CCP1IF (or CCP2IF) is HIGH.
8. As soon as a match occurs (CCP1IF or CCP2IF HIGH), stop timer1 using T1CON (or Timer3 using T3CON), and the programmed bit value (HIGH or LOW or Toggled or Unchanged) will be outputted to the CCPx pin.

Example 11.2 Assume PIC18F4321.

(a) Write a PIC18F assembly language program at address 0x100 that will toggle the CCP1 pin after a time delay of one ms. Use Timer3, and compare mode of CCP1. Use 8MHz internal clock.

(b) Write a C language program that will toggle the CCP1 pin after a time delay of one ms. Use Timer3 and Compare mode of CCP1. Use 8MHz internal clock.

Solution

With an 8-MHz internal crystal, Fosc = 8 MHz. Since Timer3 uses Fosc/4. Timer clock frequency = Fosc/4 = 2 MHz. Hence, clock period of Timer3 = 0.5 μsec. Counter value = (1 ms)/(0.5 μsec) = 2000_{10} = 0x07D0. Hence, CCPR1H :CCPR1L should be loaded with 0x07D0 for the PIC18F4321 compare mode.

(a) The PIC18F assembly language program is provided below:

```
        INCLUDE   <P18F4321.INC>
        ORG       0x100
        MOVLW     0x70            ; 8MHz internal clock
        MOVWF     OSCCON
        MOVLW     0x02            ; Select compare mode, toggle CCP1 pin
        MOVWF     CCP1CON         ; on match
        BCF       TRISC, CCP1     ; Configure CCP1 pin as output
        MOVLW     0x40            ; Select TIMER3 as clock source for
                                  ; compare
        MOVWF     T3CON           ; Select TIMER3 internal clock, 1:1 prescale
                                  ; TIMER3 OFF
        MOVLW     0x07            ; Load CCPR1H with 0x07
        MOVWF     CCPR1H
        MOVLW     0xD0            ; Load CCPR1L with 0xD0
        MOVWF     CCPR1L
        CLRF      TMR3H           ; Initialize TMR3H to 0
        CLRF      TMR3L           ; Initialize TMR3L to 0
        BCF       PIR1, CCP1IF    ; Clear CCP1IF
        BSF       T3CON, TMR3ON   ; Start Timer3
WAIT    BTFSS     PIR1, CCP1IF    ; Wait in a loop until CCP1IF is 1. CCP1 pin
        BRA       WAIT            ; toggles when match occurs
        BCF       T3CON, TMR3ON   ; Stop Timer3
HERE    BRA       HERE            ; Halt
        END
```

(b) The C language program is provided below:

```
#include<p18f4321.h>
void main(void)
{
        OSCCON = 0x70;              // 8MHz internal clock
        CCP1CON=0x02;               // Select compare mode, toggle CCP1 pin
                                    // on match
        TRISCbits.TRISC2=0;         // Configure CCP1 pin as output
        T3CON=0x40;                 // Select TIMER3 as clock for compare,
                                    // 1:1 prescale
        CCPR1H=0x07;                // Load CCPR1H with 0x07
        CCPR1L=0xD0;                // Load CCPR1L  with 0xD0
        TMR3H=0;                    // Initialize TMR3H to 0
        TMR3L=0;                    // Initialize TMR3L to 0
        PIR1bits.CCP1IF=0;          // Clear CCP1IF
        T3CONbits.TMR3ON=1;         // Start Timer3

        while(PIR1bits.CCP1IF==0); // Wait in a loop until CCP1IF is 1

        T3CONbits.TMR3ON=0;         // Stop Timer3

        while(1);                   // Halt
}
```

Example 11.3 Write a C-program to generate a waveform with a 100 ms period and a 75% duty cycle on the CCP1 pin of the PIC18F4321. Use Compare mode, Timer3, and one MHz crystal.

Solution

With 1-MHz OSC, Frequency = Fosc/4 = 0.25MHz
Period = 1/0.25MHz = 4 microseconds, Waveform period is 100 milliseconds

With 75% HIGH during a period, the waveform will be HIGH for 75 ms, LOW for 25 ms.

Hence, with 1:1 prescaler, Counter value = (75milliseconds)/(4 microseconds) = 18,750 (decimal) or 0x493E. Therefore, 0x493E should be loaded into CCPR1H:CCPR1L for 75% HIGH during a period.

Also, with 25% LOW during a period and with a 1:1 prescaler, Counter value = (25milliseconds)/(4 microseconds) = 6250 (decimal) or 0x186A. Hence, 0x186A should be loaded into CCPR1H:CCPR1L for 25% LOW during a period.

The C-program is provided below:

```
void main(void)
{
    CCP1CON=0x02;               // Select compare mode, toggle CCP1 pin on match
    TRISCbits.TRISC2=0;         // Configure CCP1 pin as output
    T3CON=0x40;                 // Select TIMER3 as clock for compare, 1:1 prescale
    while(1)
    {
      //--LOW:
```

```
        CCPR1H=0x18;            // Load CCPR1H
        CCPR1L=0x6A;            // Load CCPR1L
        TMR3H=0;                // Initialize TMR3H to 0
        TMR3L=0;                // Initialize TMR3L to 0
        PIR1bits.CCP1IF=0;      // Clear CCP1IF
        T3CONbits.TMR3ON=1;     // Start Timer3
        while(PIR1bits.CCP1IF==0);      // Wait in a loop until CCP1IF is 1
        T3CONbits.TMR3ON=0;     // Stop Timer3
        //--HIGH:
        CCPR1H=0x49;            // Load CCPR1H
        CCPR1L=0x3E;            // Load CCPR1L
        TMR3H=0;                // Initialize TMR3H to 0
        TMR3L=0;                // Initialize TMR3L to 0
        PIR1bits.CCP1IF=0;      // Clear CCP1IF
        T3CONbits.TMR3ON=1;     // Start Timer3
        while(PIR1bits.CCP1IF==0);      // Wait in a loop until CCP1IF is 1
        T3CONbits.TMR3ON=0;     // Stop Timer3
    }
}
```

11.1.5 PIC18F4321 PWM (Pulse Width Modulation) mode

In Pulse-Width Modulation (PWM) mode, the CCPx pin can be configured as an output to generate a periodic waveform with a specified frequency, and a 10-bit (8-bit integer part and 2-bit fractional part) duty cycle. The PWM duty cycle is specified by writing the upper eight bits (integer part) into the CCPRxL register, and the lower two bits (fractional part) of the CCPxCON register (bits 5 and 4). Timer2 is used for the PWM mode. The PWM period is specified by writing to the 8-bit PR2 register in the CCP module.

From the PIC18F4321 data sheet, the PWM period can be calculated using the following formula:

PWM Period = [(PR2) + 1] x 4 x Tosc x (TMR2 Prescale Value)

where Tosc = (1/Fosc), Fosc is the crystal frequency, TMR2 Prescale Value can be initialized as 1, 4, or 16 using the T2CON register.

Hence, PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1

Note that PWM frequency (Fpwm) is defined as 1/[PWM period].

The following equation is used to calculate the PWM duty cycle in time:

PWM Duty Cycle = (CCPxL:CCPxCON<5:4>) × Tosc × (TMR2 Prescale Value).

As mentioned before, the duty cycle is defined as the percentage of the time the pulse is high in a clock period. Also, the upper eight bits in the CCPRxL is the decimal part of the duty cycle while bits 5 and 4 of the CCPxCON register contain the fractional part of the duty cycle. For example, consider a 25% duty cycle. Since duty cycle is a fraction of the PR2 register value, decimal value for the duty cycle with a PR2 value of 30 is 7.5 (0.25 x 30). Hence, the 8-bit 00000111 must be loaded into CCPRxL, and 2-bit 10 (0.5 decimal) must be loaded for DCxB1 and DCxB0 bits in the CCPxCON register (Figure 11.1). This will assign a duty cycle of 7.5 to the PWM waveform to be generated at the CCPx pin.

The CCP1 PWM output waveform with the specified duty cycle in CCPR1L:CCPICON register is generated as follows: CCPR1L is copied to CCPR1H. Two bits of CCP1CON <5:4> are latched internally to provide a 10-bit duty cycle. Also, an 8-bit TMR2 value is concatenated with a two-bit internal latch to create a 10-bit duty cycle.

After TMR2 is started from 0, the CCP1 pin goes to HIGH to indicate the start of a cycle. The CCPR1H and two-bit latch values are compared with TMR2 and two-bit latch values for a match. As soon as the match occurs, the CCP1 pin goes to LOW. At this point, the waveform will be HIGH for the duration specified by the duty cycle. TMR2 keeps incrementing. As soon as the contents of TMR2 and PR2 match, CCP1 pin is driven to HIGH, and TMR2 is cleared to 0. This completes a cycle, and another cycle is then started. The same process continues for subsequent cycles. This is depicted in Figure 11.2.

The following procedure should be followed when configuring the CCP module for PWM operation:

1. The PR2 register should be initialized with the PWM period.
2. Load the PWM duty cycle by writing the 8-bit integer value into the CCPRxL register, and the two-bit fractional part into bits 5, 4 of CCPxCON (Figure 11.1). Also, disable PWM mode by programming the CCPxCON register.
3. Configure the CCPx pin an output by clearing the appropriate TRISCx bit.
4. Select the Timer2 prescale value, then enable Timer2 by writing to T2CON.
5. Enable PWM mode by writing appropriate data into CCPxCON register.
6. Initialize TMR2 register to 0.
7. Turn Timer2 ON by writing appropriate data into T2CON register.
8. Clear TMR2IF flag in the PIR1 register (Figure 10.8) to 0.
9. Wait in a loop for the TMR2IF flag to become HIGH. As soon as TMR2IF is HIGH, the PWM waveform is generated at the CCPx pin with the specified duty cycle.
10. This will complete a cycle.

PWM waveform can be generated by polling the TMR2IF in the PIR1 register (Figure 10.8). The following PIC18F assembly language instruction sequence and C-code will poll the TMR2IF flag bit in the PIR1 register by clearing TMR2IF to 0, turning the TMR2 On, and then waiting in a loop for the TMR2IF bit to become HIGH indicating completion of a cycle:

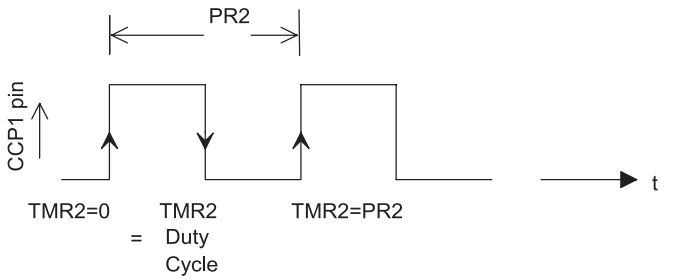


FIGURE 11.2 CCP1 waveform in PWM mode

In PIC18F assembly:

```
REPEAT  BCF     PIR1, TMR2IF      ; Clear TMR2IF flag to 0
BACK    BSF     T2CON, TMR2ON     ; Turn Timer2 ON
        BTFSS   PIR1, TMR2IF      ; wait until completion of a cycle
        GOTO    BACK
        GOTO    REPEAT            ; Repeat
```

In C:

```
PIR1bits.TMR2IF=0;                // Clear TMR2IF flag to 0
T2CONbits.TMR2ON=1;               // Turn Timer2 ON
while(PIR1bits.TMR2IF==0);        // Wait until TMR2IF = 1
```

Example 11.4 (a) Write a PIC18F assembly language program at address 0x100 to generate a 4 KHz PWM with a 50% duty cycle on the CCP1 pin of the PIC18F4321. Assume 4 MHz crystal.

(b) Write a C language program to generate a 4 KHz PWM with a 50% duty cycle on the CCP1 pin of the PIC18F4321. Assume 4 MHz crystal.

Solution

PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1
PR2 = [(4 MHz)/(4 x 4 KHz x 1)] - 1 assuming Prescale value of 1
PR2 = 249. With 50% duty cycle = 0.5 x 249 = 124.5. Hence, the CCPR1L register will be loaded with 124, and bits DC1B1:DC0B0 (CCP1CON register) with 10 (binary).

(a) The PIC18F assembly language program is provided below:

```
        INCLUDE  <P18F4321.INC>
        ORG      0x100
        MOVLW    0x60             ; 4MHz internal clock
        MOVWF    OSCCON
        MOVLW    D'249'           ; Initialize PR2 register
        MOVWF    PR2
        MOVLW    D'124'           ; Initialize CCPR1L
        MOVWF    CCPR1L
        MOVLW    0x20             ; CCP1 OFF,
        MOVWF    CCP1CON          ; DC1B1:DC0B0=10
        BCF      TRISC, CCP1      ; Configure CCP1 pin as output
        CLRF     T2CON            ; 1:1 prescale, Timer2 OFF, no postscale
        MOVLW    0x2C             ; PWM mode
        MOVWF    CCP1CON
        CLRF     TMR2             ; Clear Timer2 to 0
BACK    BCF      PIR1, TMR2IF     ; Clear TMR2IF to 0
        BSF      T2CON, TMR2ON    ; Turn Timer2 ON
WAIT    BTFSS    PIR1, TMR2IF     ; Wait until TMR2IF is HIGH
        GOTO     WAIT
        BRA      BACK
        END
```

In the above program, the value of TMR2 is compared to the period register, PR2, on each clock cycle. When the two values match, a comparator generates a match signal as the timer output. This signal also resets the value of TMR2 to 0x00 on the next cycle. In the above program, the last instruction BRA BACK branches to the label where the TMR2IF flag in the PIR1 is cleared to 0. The program does not have to go back to clear TMR2 to 0 since the TMR2 is automatically cleared after each match.

(b) The C language program is provided below:

```
#include<p18f4321.h>
void main(void)
{
    OSCCON =0x60;                 // 4MHz internal clock
    PR2=249;                      // Initialize PR2 register
    CCPR1L=124;                   // Initialize CCPR1L
    CCP1CON=0x20;                 // CCP1 OFF, DC1B1:DC0B0=10
    TRISCbits.TRISC2=0;           // Configure CCP1 pin as output
    T2CON =0;                     // 1:1 prescale, Timer2 OFF, no postscale
    CCP1CON=0x2C;                 // PWM mode
    TMR2=0;                       // Clear Timer2 to 0

    while(1)
    {
    PIR1bits.TMR2IF=0;            // Clear TMR2IF to 0
    T2CONbits.TMR2ON=1;           // Turn Timer2 ON
    while(PIR1bits.TMR2IF==0);    // Wait until TMR2IF is HIGH
    }

}
```

11.2 DC Motor Control

As mentioned before, the duty cycle of a PWM signal is directly proportional to the average voltage and power provided by the PWM. Because of this feature, the PWM mode can be used in applications such as dimming LEDs and controlling the speed of a brushless DC electric motor. This means that LEDs will be brighter as the PWM duty cycle is increased. Similarly, the speed of a DC motor will increase as the PWM duty cycle is increased. On the other hand, LEDs will be dimmer with a lower PWM duty cycle while the motor will run at a slower speed.

In earlier days, voltage regulator circuits were used to control the speed of a DC motor. But voltage regulators dissipate a lot of power. Hence, the PIC18F in the PWM mode is used to control the speed of a DC motor. In this scheme, power dissipation is significantly reduced by turning the driving voltage to the motor ON and OFF.

Sometimes, it is desirable to change direction of rotation of the DC motor. This can be accomplished by reversing the direction of the motor via software by interfacing a device called H-Bridge to an I/O port of the PIC18F. Note that the speed of the motor, on the other hand, can be controlled using the PWM mode and by connecting the DC motor to the CCP1 pin of the PIC18F4321. The basic concepts associated with the DC motor control using the PIC18F4321's PWM mode will be illustrated in Example 11.5.

Microcontrollers such as the PIC18F4321 are not capable of outputting the required large current and voltage to control a typical DC motor. Hence, a driver such as the CNY17F Optocoupler is needed to amplify the current and voltage provided by the PIC18F's output, and provide appropriate levels for the DC motor. One of the many useful applications for using a PWM signal is its ability to control a device such as a motor.

In Pulse-Width Modulation (PWM) mode, the CCPx pin can be configured as an output to generate a periodic waveform with a specified frequency, and a 10-bit duty cycle. TMR2 is used for the PWM mode. The PWM period is specified by writing to an 8-bit internal register called PR2 in the CCP module. Note that PWM waveform can be generated using timers. However, it is easier to produce PWM waveform using the CCP module.

Example 11.5 It is desired to change the speed of a DC motor by dynamically changing its pulse width using a potentiometer connected to bit 0 of PORTB (Figure 11.3 (a)). Note that the PWM duty cycle is controlled by the potentiometer. The PIC18F4321 will input the potentiometer voltage via its on-chip A/D converter as eight bits, generate an 800-Hz PWM waveform on the CCP1 pin, and then change the speed of the motor as the potentiometer voltage is varied. Use a 4 a MHz crystal and a TMR2 prescaler value of 16. Ignore the fractional part of the duty cycle. Note that the circuit in Figure 11.3 (b) can be used instead of the optocoupler CNY17F.

Using interrupt-driven ADC:

(a) write a PIC18F assembly language program to accomplish this.

(b) write a C language program to accomplish this.

Solution

PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1
= [(4 MHz)/(4 x 800x 16)] - 1
= 77.125
Hence, PR2 will be 77 approximately.

After converting the potentiometer voltage into 8-bit binary, the ADRESH will contain the converted data. Note that the 8-bit ADRESH register can accommodate a maximum value of 255. The contents of ADRESH can be moved to CCPR1L to represent the decimal portion of the duty cycle. In order for the duty cycle to be in the range of 0 to 77, the contents of ADRESH must be divided by 3 (255/77 = 3 approximately). This will ensure that the decimal portion of the duty cycle is between 0 and 77. The higher the voltage across the potentiometer, the higher the value of ADRESH/3. This will generate a PWM waveform with a higher duty cycle, and thus the motor will run faster.

(a) The PIC18F assembly language program is provided below:

```
INCLUDE <P18F4321.INC>
ORG     0x000
GOTO    MAIN
ORG     0x0008          ; Program will go to CHECK_INT
                        ; at interrupt
GOTO    ADIF_ISR        ; Go to Interrupt Service Routine
ORG     0x70            ; Start of the main program
```

```
MAIN        MOVLW   0x60
            MOVWF   OSCCON          ; Setting the internal clock to 4MHz
            MOVLW   0x10            ; Initialize STKPTR
            MOVWF   STKPTR          ; since interrupt is used
            MOVLW   0x00
            MOVWF   TRISC           ; Make PORTC output
            MOVLW   0x02
            MOVWF   T2CON           ; Configure Timer2 with prescale 16 and
                                    ; no postscale
            CLR     TMR2
            BCF     PIR1, TMR2IF    ; Clears Timer2  flag
            MOVLW   0x31            ; Use AN12 as ADC
            MOVWF   ADCON0
            MOVLW   0x00
            MOVWF   ADCON1          ; Enable pins for analog input
            MOVLW   0x08
            MOVWF   ADCON2          ; 2 TAD and Fosc/2, Left justified
            MOVLW   0x0C
            MOVWF   CCP1CON         ; Select PWM mode
            MOVLW   D'77'           ; Set period of PWM signal
            MOVWF   PR2
START       BSF     ADCON0,GO       ; Start the ADC
WAIT        BRA     WAIT            ; Wait for ADC to interrupt
            BRA     START           ; Start ADC again
            ORG     0x200           ; start of the Service routine
ADIF_ISR    MOVFF   ADRESH, 0x20    ; Move value in ADRESH to 0x20
                                    ; Divide by 3 using repeated addition
            MOVLW   0x03            ; Move 3 into the WREG
            CLRF    0x21            ; Clear value in 0x21
DIVIDE      CPFSGT  0x20            ; Compare the value to 3 skip if greater than
            BRA     FINISHED        ; Division is done
            INCF    0x21, F         ; Increment 0x21
            SUBWF   0x20            ; Subtract 3 from 0x20
            BRA     DIVIDE          ; Subtract again
FINISHED    MOVFF   0x21, CCPRIL    ; Move final value into CCPRIL
            BCF     PIR1, TMR2IF    ; Clear Timer2 flag
            BSF     T2CON, TMR2ON   ; Turn on Timer2
AGAIN       BTFSS   PIR1, TMR2IF    ; Wait until cycle is over
            BRA     AGAIN
            RETFIE                  ; Return to start ADC again
            END
```

(b) The C-program is provided below:

```
#include <p18f4321.h>
void ADINT_ISR (void);
#pragma interrupt check_int
void check_int(void)
{    ADINT_ISR();
```

```
}
#pragma code ADC_INT=0x08    // At interrupt, code jumps here
void ADC_Int (void)
{    _asm
     GOTO check_int
     _endasm
}
#pragma code                 // End of Code
void main ()
  {      OSCCON=0x60;        // 4 MHz
         PIE1bits.ADIE = 1;  // enable ADC interrupt
         PIR1bits.ADIF = 0;  // clear ADC interrupt flag
         INTCONbits.PEIE = 1; // enable peripheral interrupt
         INTCONbits.GIE = 1; // enable global interrupt
         TRISC=0;            // CCP1 output
         T2CON=0x02;         // Configure Timer2 with prescale 16
         PIR1bits.TMR2IF=0;  // Clear Timer2 interrupt flag
```

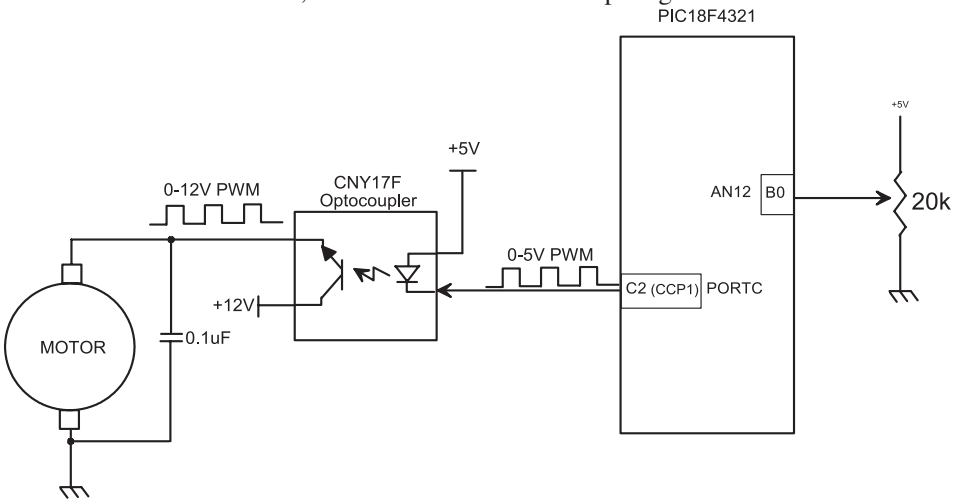


FIGURE 11.3(a) Figure for Example 11.5

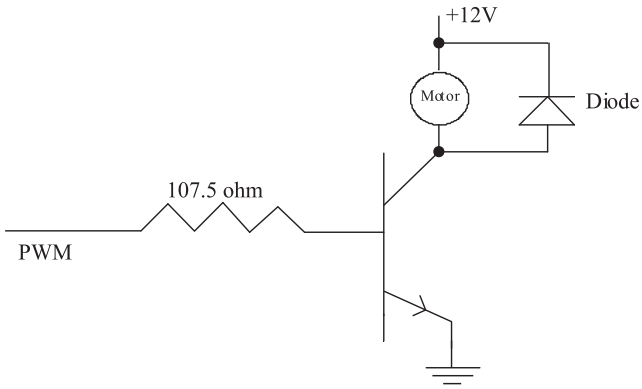


FIGURE 11.3(b) Circuit to be used instead of the Optocoupler

```
        ADCON0=0x31;            // AN12
        ADCON1=0x00;            // Enable analog pins
        ADCON2=0x08;            // 2 TAD and Fosc/2, Left justified8-bit in ADRESH
        CCP1CON=0x0C;           // PWM enable
        PR2=77;
        TMR2 = 0;
        ADCON0bits.GO = 1;

      while (1);                  // Wait for interrupt
}
      void ADINT_ISR (void)
      {
            PIR1bits.ADIF = 0;
            CCPR1L=ADRESH/3;
            PIR1bits.TMR2IF=0;          // Clear Timer2 interrupt
            T2CONbits.TMR2ON=1;

       while(PIR1bits.TMR2IF==0);
       ADCONbits.GO=1;
}
```

11.3 Serial Interface

In various instances, it is desirable to transmit binary data from one microcontroller to another. In such situations, data can be transmitted using either parallel or serial transmission techniques. In parallel transmission, each bit of the binary data is transmitted over a separate line. This means that an 8-bit data will be transmitted over eight lines. Parallel data transmission is feasible for short distances. It is not cost effective for transferring data between two systems located at hundreds of feet. Hence, serial transmission is normally used between two systems such as two computers located at long distances.

For long distances, telephone lines can be used for data transmission. A peripheral device called "modem" can be used for this purpose. Note that modem stands for modulator/demodulator. The modem at the transmitting end transforms (modulates) binary data at the transmitting digital device, such as a microcontroller into audio signals, while another modem for the receiving end transforms (demodulates) audio signals to binary data for the receiving digital device, such as another microcontroller.

In serial transmission, only one line is used to transmit the complete binary data bit by bit. Hence, the transmitting device such as a microcontroller must convert parallel data into a string of serial bits using a parallel-in-serial-out shift register. The receiving device, such as another microcontroller, must convert data from serial to parallel using a serial-in-parallel-out shift register. Data are usually sent starting with the least significant bit. In order to differentiate among various bits, a clock signal is used. Serial data transmission can be divided into two types: synchronous and asynchronous. We will now briefly describe them.

11.3.1 Synchronous Serial Data Transmission

The basic feature of synchronous serial data transmission is that data are transmitted or received based on a clock signal. After deciding on a specific rate of data transmission, commonly known as "baud rate" (bits per second), the transmitting device

sends a data bit at each clock pulse. In order to interpret data correctly, the receiving device must know the start and end of each data unit. Therefore, in synchronous serial data transmission, the receiver must know the number of data units to be transferred. Also, the receiver must be synchronized with data boundaries. Usually, one or two SYNC characters (a string of bits) are used to indicate the start of each synchronous data stream. The data unit normally contains error bits such as parity. In some transmissions, the least significant bit is used as a parity bit. The synchronous usually waits in a "hunt" mode while looking for data. As soon as it matches one or more SYNC characters based on the number of SYNC characters used, the receiver starts interpreting the data. In synchronous serial transmission, the transmitting device needs to send data continuously to the receiving device. However, if data are not ready to be transmitted, the transmitter will be padded with SYNC characters until data are available.

As mentioned before, in synchronous serial transfer, the receiver must know the number of SYNC characters used and the number of data units to be transferred. Once the receiver matches the SYNC characters, it receives the specified number of data units, and then goes into a "hunt" mode for matching the SYNC pattern for the next data.

11.3.2 Asynchronous Serial Data Transmission

In this type of data transfer, the transmitting device does not need to be synchronized to the receiving device. The transmitting device can send one or more data units when it has data ready to be sent. Each data unit must be formatted. In other words, each data unit must contain "start" and "stop" bits, indicating the beginning and the end of each data unit. The interface circuits between the transmitting device and the receiving device must perform the following functions:

1. Convert an 8-bit parallel data unit from the transmitting device into serial data for transmitting them to the receiving device.
2. Convert serial data from the receiving device into parallel data for sending them back to the transmitting device assuming two-way (full duplex) transmission.

Each data unit can be divided into equal time intervals, called "bit intervals". A data bit can be either HIGH or LOW during each bit interval. For example, an 8-bit data will have eight bit intervals. Each data bit will correspond to one of the eight bit intervals.

The format for asynchronous serial data typically contains the following information:

1. A LOW START bit.
2. 5-8 bits, denoting the actual data being transferred.
3. An optional parity bit for either odd or even parity.
4. 1, 1½, or 2 STOP bits having HIGH levels. Note that 1½ STOP bits mean a HIGH level with a duration of 1.5 times the bit interval.

11.3.3 Basics of SPI and I²C

Serial I/O is typically fabricated as an on-chip module in microcontrollers. This will facilitate interfacing microcontrollers with other microcontrollers or peripheral devices. Several protocol (rules) standards for serial data transmission have been introduced over the years. Two such standards include SPI (Serial Peripheral Interface) developed by Motorola in the late 80's, and I²C (Inter-Integrated Circuit) developed by Philips in 1982. Both protocols are based on synchronous serial data transmission. Also, both SPI and I²C protocols are used for communication between ICs (Integrated Circuits) with on-board

peripherals. They are intended for data transmission from a master to or from one or more slave devices over short distances.

The main purpose of the SPI is to replace parallel interfaces by avoiding routing of parallel buses in a PCB (Printed Circuit Board). The SPI protocol is based on the principle that a bit from an 8-bit shift register can be shifted out on a single pin and a bit can be shifted into another pin. SPI protocol can also be used for data transfer between the CPU (master) and slave devices such as flash memory and ADC.

The original purpose of the I^2C protocol was to easily interface a CPU to peripheral chips in a TV. Peripheral devices in an embedded system are typically connected to a microcontroller using memory-mapped I/O. This requires a lot of wiring on the PCB (Printed Circuit Board). In order to refrain from using microcontroller pins and avoid additional circuits, I^2C protocol can be used to interface the microcontroller to all the peripherals. This will also reduce wiring and thus, implementation cost.

SPI uses four wires; one for clock, one for data out, one for data in, and one for SS or CS (Slave Select or Chip Select to enable a slave device). I^2C, on the other hand, uses two wires; one for clock and one for data. Data transfer using SPI is faster (up to 25MHz) while data transfer using I^2C is slower (100KHz to 400KHz). As far as the software is concerned, SPI and I^2C perform communication in a similar manner. In both protocols, the master generates the clock and data read or written at the rising or falling edges of the clock. However, SPI protocol is based on a single master and one or more slaves while I^2C protocol is based on multi-master with a single or multiple slaves.

Both polled and interrupt I/O can be used for data transfer using SPI and I^2C modes. However, polled I/O will be used in this discussion to illustrate the basic concepts.

11.4 PIC18F Serial I/O

The PIC18F4321 contains an on-chip Master Synchronous Serial Port (MSSP) module which is a serial interface, useful for communicating with other peripheral or microcontroller devices. The MSSP module can operate in either SPI or I^2C mode. We will cover the PIC18F SPI and I^2C modes in this section.

11.4.1 PIC18F SPI mode

The PIC18F SPI primarily uses four pins of the PIC18F4321. They are SCK (Serial Clock), SDI (Serial Data Input) , SDO (Serial Data Output) and $\overline{SS}$ (Slave Select). The $\overline{SS}$ pin is provided for applications requiring multiple slave devices. The $\overline{SS}$ pin may not be used for a single slave in the system. In summary, SCK, SDI, SDO, and $\overline{SS}$ pins are provided on the PIC18F to support SPI.

Figure 11.4 shows a block diagram interfacing a PIC18F master with a single PIC18F slave. Since SPI is a master/slave protocol, the master is the only device which will control the clock (SCK). Data transfers between the master and the slave are controlled by the SCK clock. This means that sending and reading of data are controlled by SCK.

SDI carries the data into the Master while SDO carries the data sent out from the Master. $\overline{SS}$ is similar to a chip select signal. $\overline{SS}$ must be used when more than one slave is present. However, the pin can be disabled via programming for a single slave. This will be illustrated using an example later in this section. For simplicity, a master with a single slave will be covered in this section. Coverage of a master with multiple slaves is beyond the scope of this book.

Figure 11.5 shows a simplified diagram of SPI Master/Slave connection between

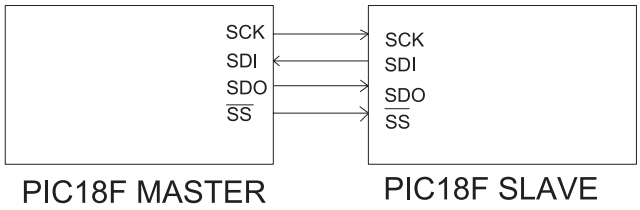


FIGURE 11. 4 PIC18F Master interface to a single slave PIC18F

two PIC18F4321's along with relevant registers. SSPSR is the shift register of the SPI module. It shifts data in and data out of the master or the slave. Data travels in a loop to the next shift register. Data is shifted out on the SDO pin of one device and into the SDI pin of the other device. Once a byte of data has been transferred between the two devices, it is copied to the SSPBUF register. The SSPBUF is then read by the user program.

The master initiates the data transfer while the slave waits for the clock from the master before the transfer occurs. The master determines when an SPI transfer can take place. In order to accomplish this, the master transmits SCK. The slave waits for the SCK signal and uses it when processing the SPI data.

PIC18F4321 registers in SPI mode SPI is implemented in the PIC18F microcontroller using a hardware module called the Master Synchronous Serial Port (MSSP).
The MSSP module uses four registers for SPI mode operation. These are:

- MSSP Control Register 1 (SSPCON1)
- MSSP Status Register (SSPSTAT)
- Serial Receive/Transmit Buffer Register (SSPBUF)
- MSSP Shift Register (SSPSR) – Not directly accessible

Figures 11.6 and 11.7 show the SSPCON1 and SSPSTAT registers respectively. The SSPCON1 and SSPSAT are control and status registers in SPI mode of operation. The SSPCON1 can be used to select the master or slave mode using the SSPM3-SSPM0 bits. The SSPEN bit of the SSPCON1 register is set to one to turn on the SSP module for using it for SPI. The SSP module must be left ON for the entire time the SSP module is in use. SSPEN can be cleared to 0 to disable or reset the SSP module.

The SSMP3-SSMP0 (bits 3-0) can be used to select the master or slave mode. The clock polarity is selected using the CKP bit. The purposes of WCOL and SSPOV in the SSPCON1 will be described later in this section.

Both SPI and I²C modes use the SSPSTAT register. This register can be used to select the SPI mode (master or slave) using the SMP (data SaMPle Timing) bit, SPI clock using the CKE (ClocK Edge select) bit, and the Buffer Full (BF) bit. These three bits in the SSPSTAT control SPI.

SMP must be maintained clear for the slave. In master mode, SMP = 1 means a sample occurs at the end of data output while SMP = 0 indicates a sample occurs in the middle of data output. The SMP bit controls when data received from the SDI line is sampled, relative to when it is sent out on the SDO line.

CKP along with CKE controls one of four SPI modes used for all SPI transfers. Four CKP CKE modes are 00, 01, 10, and 11. Most common modes are 00 and 11. Note that CKP = 0 and CKE = 0 mean that data transfer occurs at the LOW to HIGH clock. On the other hand, when CKP = 1 and CKE = 1, data transfer occurs at the HIGH to LOW clock.

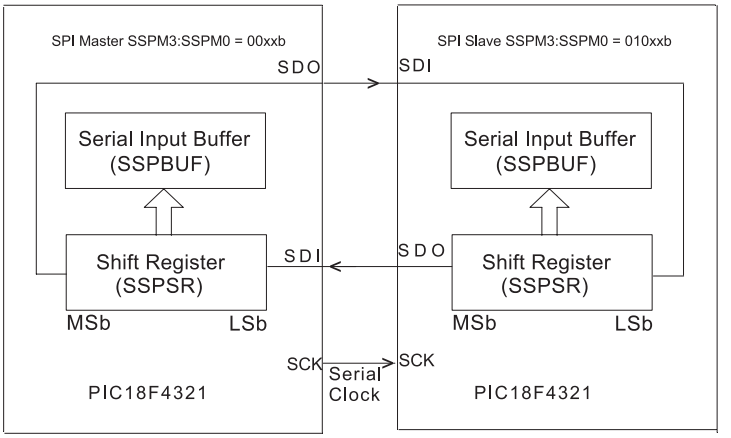


FIGURE 11.5 SPI Master/Slave interface between two PIC18F4321's along with relevant registers

BF (Buffer Full) = 1 means that the SSPBUF register contains data that has not yet been read. SSBUF register contains data received via SPI. The data should be read before any more data is written or received. This is true whether the device is a master or a slave.

The SSPOV (SSP Overflow) bit is set to one if SSPBUF is not read before another byte of data is transferred. After an occurrence of SSPOV = 1, indicating an overflow, the module must be reset to clear this condition. Toggling the SSPEN bit will reset the SSP module.

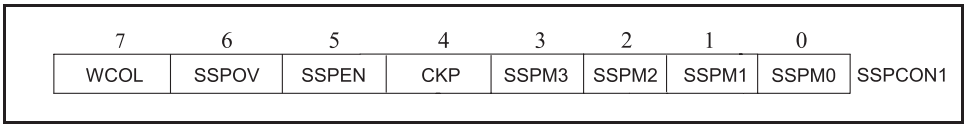
Bits 1-5 of the SSPSTAT are used in the I^2C mode.

Operation When initializing the SPI, several options need to be specified. This is done by programming the appropriate control bits (bits 0-5 of SSPCON1 and bits 7-6 of SSPSTAT). These control bits allow the following to be specified:

- Master mode (SCK is the clock output)
- Slave mode (SCK is the clock input)
- Clock Polarity (Idle state of SCK)
- Data Input Sample Phase (middle or end of data output time)
- Clock Edge (output data on rising/falling edge of SCK)
- Clock Rate (Master mode only)
- Slave Select mode (Slave mode only)

The MSSP consists of a transmit/receive shift register (SSPSR) and a buffer register (SSPBUF). The SSPSR shifts the data in and out of the device, the most significant bit (MSB) first. The SSPBUF holds the data that was written to the SSPSR until the received data is ready. Once the 8 bits of data have been received, that byte is moved to the SSPBUF register. Then, the Buffer Full detect bit, BF (bit 0 of SSPSTAT), and an interrupt flag bit (not discussed here) are set to 1. Data transfer can be performed by polling BF or via interrupt. However, data transfer using BF is considered in this section.

User software must clear the WCOL bit so that it can be determined if the following write(s) to the SSPBUF register completed successfully. When the application software is



bit 7 **WCOL:** Write Collision Detect bit (Transmit mode only)
1 = The SSPBUF register is written while it is still transmitting the previous word (must be cleared in software)
0 = No collision

bit 6 **SSPOV:** Receive Overflow Indicator bit
SPI Slave mode:
1 = A new byte is received while the SSPBUF register is still holding the previous data. In case of overflow, the data in SSPSR is lost. Overflow can only occur in Slave mode. The user must read the SSPBUF, even if only transmitting data, to avoid setting overflow (must be cleared in software).
0 = No overflow
Note: In Master mode, the overflow bit is not set since each new reception (and transmission) is initiated by writing to the SSPBUF register.

bit 5 **SSPEN:** Synchronous Serial Port Enable bit
1 = Enables serial port and configures SCK, SDO, SDI and SS as serial port pins
0 = Disables serial port and configures these pins as I/O port pins
Note: When enabled, these pins must be properly configured as input or output.

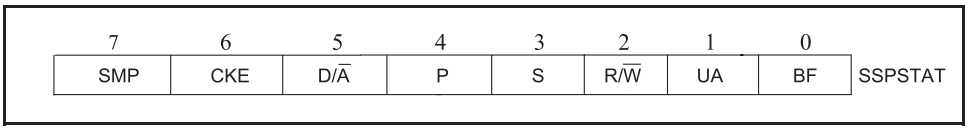
bit 4 **CKP:** Clock Polarity Select bit
1 = Idle state for clock is a high level
0 = Idle state for clock is a low level

bit 3-0 **SSPM3:SSPM0:** Synchronous Serial Port Mode Select bits
0101 = SPI Slave mode, clock = SCK pin, SS pin control disabled, SS can be used as I/O pin
0100 = SPI Slave mode, clock = SCK pin, SS pin control enabled
0011 = SPI Master mode, clock = TMR2 output/2
0010 = SPI Master mode, clock = FOSC/64
0001 = SPI Master mode, clock = FOSC/16
0000 = SPI Master mode, clock = FOSC/4
Note: Bit combinations not specifically listed here are either reserved or implemented in I2C™ mode only.

FIGURE 11.6 SSPCON1 (MSSP CONTROL) Register 1 in SPI mode

expecting to receive valid data, the SSPBUF should be read before the next byte of data to transfer is written to the SSPBUF. The Buffer Full bit, BF (bit 0 of SSPSTAT), indicates when SSPBUF has been loaded with the received data (transmission is complete). When the SSPBUF is read, the BF bit is cleared. This data may be irrelevant if the SPI is only a transmitter. Generally, the MSSP interrupt is used to determine when the transmission/reception has completed. The SSPBUF must be read and/or written. If the interrupt method is not going to be used, then software polling can be done to ensure that a write collision does not occur.

Enabling SPI I/O To enable the serial port, MSSP Enable bit, SSPEN (bit 5 of SSPCON1) must be set to one. To reset or reconfigure SPI mode, clear the SSPEN bit to 0, reinitialize the SSPCON registers, and then set the SSPEN bit to one. This configures the SDI, SDO, and SCK pins as serial port pins. For the pins to behave as the serial port function, the data direction bits (in the TRIS register) must be appropriately programmed



bit 7 **SMP:** Sample bit
SPI Master mode:
1 = Input data sampled at end of data output time
0 = Input data sampled at middle of data output time
SPI Slave mode:
SMP must be cleared when SPI is used in Slave mode.

bit 6 **CKE:** SPI Clock Select bit
1 = Transmit occurs on transition from active to Idle clock state
0 = Transmit occurs on transition from Idle to active clock state
Note: Polarity of clock state is set by the CKP bit (SSPCON1<4>).

bit 5 **D/A:** Data/Address bit
Used in I2C™ mode only.

bit 4 **P:** Stop bit
Used in I2C mode only. This bit is cleared when the MSSP module is disabled, SSPEN is cleared.

bit 3 **S:** Start bit
Used in I2C mode only.

bit 2 **R/W:** Read/Write Information bit
Used in I2C mode only.

bit 1 **UA:** Update Address bit
Used in I2C mode only.

bit 0 **BF:** Buffer Full Status bit (Receive mode only)
1 = Receive complete, SSPBUF is full
0 = Receive not complete, SSPBUF is empty

FIGURE 11.7 SSPSTAT (MSSP Status Register) in SPI mode

as follows:

- SDI is automatically controlled by the SPI module.
- SDO pin is multiplexed with bit 5 of Port C. Hence, in order to configure SDO as an output pin, bit 5 of TRISC must be cleared to 0.
- SCK (Master mode) pin is multiplexed with bit 3 of Port C. Hence, in order to configure SCK as an output pin, bit 3 of TRISC must be cleared to 0.
- SCK (Slave mode) pin must be configured as an output pin. Hence, bit 3 of TRISC must be set to one.

In Figure 11.5, the master PIC18F4321 initiates the data transfer by sending the SCK signal. The master can initiate the data transfer at any time because it controls the SCK. The master determines when the slave will broadcast data by the software protocol. Data is shifted out of both shift registers on their programmed clock edge and latched on the opposite edge of the clock. Both processors should be programmed to the same Clock

Polarity (CKP), then both controllers would send and receive data at the same time.

Note that the SSPCON1 of the master, when initialized with 20H, will enable the SSP module for SPI mode (SSPEN = 1), select LOW idle state for the clock (CKP = 0), and select SPI master mode with the clock of FOSC/4 (SSPM3:SSPM0 = 0000). For the slave SSPCON1, on the other hand, when initialized with 25H, keep the SSP module enabled, select the same LOW idle state as the master, and select SPI slave mode with the $\overline{SS}$ pin disabled for a single slave application.

As an example, initializing master SSPSTAT with 40H will select the SPI master mode, data transmission from HIGH to LOW clock (CKE = 1), and SSPBUF empty (BF = 0). During data transmission, the BF bit in the SSPSTAT register (master or slave) can then be polled by waiting in a loop until BF = 1. This will ensure proper data transfer between the master and the slave. Example 11.6 illustrates this. Note that Example 11.6 has been successfully implemented in the laboratory. This example demonstrates the basic concepts associated with PIC18F SPI in a very simplified manner.

Example 11.6 Figure 11.8 shows a block diagram for interfacing two PIC18F4321 in SPI mode. One of the microcontrollers is the master while the other is the slave. The master PIC18F4321 will input four switches via bits 0-3 of PORTB (Switch SW0 connected to bit 0 of Port B, Switch SW1 to bit 1, SW2 to bit 2, and SW3 to bit 3), and then transmit the 4-bit data using its SDO pin to the slave's SDI pin. The slave PIC18F4321 will output this data to four LED's (LED0 connected to bit 0 of Port D, LED1 to bit 1, LED 2 to bit 2, and LED 3 to bit 3), and turn them ON or OFF based on the switch inputs. For example, LED0 will be turned ON if SW0 is one (Open) and will be turned OFF if SW0 is zero (closed), LED1 will be turned ON if SW1 is one (Open) and will be turned OFF if SW1 is zero (closed), and so on.

(a) Write a PIC18F assembly language program at 0x70 for the master PIC18F4321 that will configure PORTB, initialize STKPTR to 5, initialize SSPSTAT and SSPCON1, input switches, and call a subroutine called SERIAL_WRITE to place this data into its SSPBUF register.

Also, write a PIC18F assembly language program at 0x100 for the slave PIC18F4321 that will configure PORTD, initialize SSPSTAT and SSPCON1 registers, input data from its SDI pin, place it in the slave's SSPBUF, and then output to the LED's.

(b) Repeat (a) using C language. The master PIC18F4321 will configure PORTB, initialize SSPSTAT and SSPCON1, input switches, and place this data into its SSPBUF register.

Also, write a C language program for the slave PIC18F4321 that will configure PORTD, initialize SSPSTAT and SSPCON1 registers, input data from its SDI pin, place it in the slave's SSPBUF, and then output to the LED's.

Solution

(a) The PIC18F assembly language programs for the master PIC18F4321 and the slave PIC18F4321 can be written using the following steps as the guidelines:

Master PIC18F4321

1. Configure PORTB as input with SDO and SCK as outputs.
2. Select CKE (SPI clock select bit) using the master's SSPSTAT register.

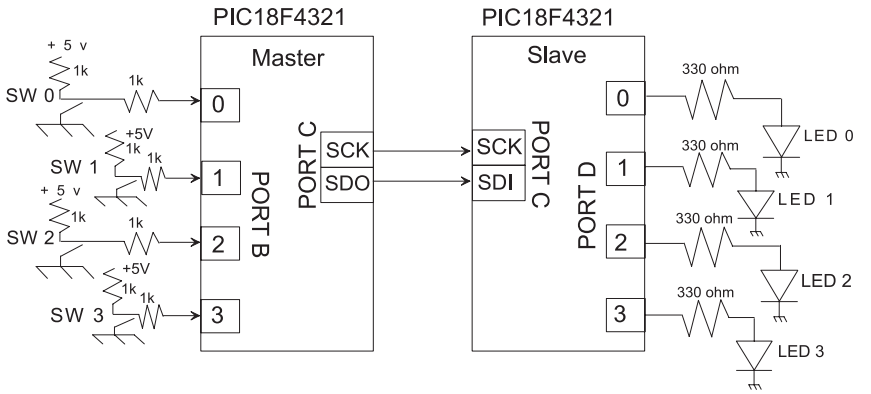


FIGURE 11.8 Figure for Example 11.6 using SPI mode

3. Enable serial functions and select the master mode with a clock such as fosc/4 using the CCPCON1 register.
4. Input switches into WREG, and then CALL a subroutine called SERIAL_WRITE to move switch data into the master's Serial Buffer register (SSPBUF).
5. Wait in a loop, and check whether the BF (Buffer Full) bit in the master's SSPSTAT register is 1, indicating completion of transmission.
6. As soon as BF = 1, the program returns from the subroutine and branches to Step 4.

Slave PIC18F4321

1. Initialize SDI and SCK pins as inputs and PORTD as an output. Note that the SCK is controlled by the master, and therefore, is configured as an input by the slave.
2. Select CKE -same as the master CKE (high to low clock in this example) -using the slave's SSPSTAT register.
3. Enable serial functions, disable the $\overline{SS}$ pin, and select slave mode using the slave's SSPCON1 register.
4. Wait in a loop and check whether BF = 1 in the slave's SSPSTAT register.
5. If BF = 0, wait. However, if BF = 1, output the contents of the slave's Serial Buffer register (SSPBUF) to slave's PORTD.
6. Go to Step 5.

The PIC18F assembly language programs for the master and the slave microcontrollers are provided below:

```
                ;Program for the master PIC18F4321

                INCLUDE <P18F4321.INC>
                ORG     0x00          ; Reset
                GOTO    MAIN
                ORG     0x70
MAIN            BCF     TRISC, RC5    ; Configure RC5/SD0 as output
                BCF     TRISC, RC3    ; Configure RC3/SCK as output
                MOVLW   0x0F
                MOVWF   ADCON1        ; Make PORTB digital input
```

```
                MOVLW   5               ; Initialize STKPTR to 5 since subroutine
                MOVWF   STKPTR          ; called SERIAL_WRITE is used in the
                                        ; program
                MOVLW   0x40
                MOVWF   SSPSTAT         ; Set data transmission on high to
                                        ; low clock
                MOVLW    0x20
                MOVWF    SSPCON1        ; Enable serial functions and set to
                                        ; master device, and  Fosc/4
GET_DATA        MOVF    PORTB,W         ; Move switch value to WREG
                CALL    SERIAL_WRITE    ; Call SERIAL_WRITE function
                BRA     GET_DATA
SERIAL_WRITE    MOVWF   SSPBUF          ; Move switch value to serial buffer
WAIT            BTFSS   SSPSTAT, BF     ; Wait until transmission is complete
                BRA     WAIT
                RETURN
                END
        ; Program for the slave PIC18F4321
                INCLUDE <P18F4321.INC>
                ORG     0x00            ; Reset
                GOTO    MAIN
                ORG     0x100
MAIN            BSF     TRISC, RC4      ; Configure RC4/SDI  as input
                BSF     TRISC, RC3      ; Configure RC3/SCK as input
                CLRF    TRISD           ; Configure PORTD as output
                MOVLW   0x40
                MOVWF   SSPSTAT         ; Set data transmission on high to
                                        ; low clock
                MOVLW   0x25
                MOVWF   SSPCON1         ; Enable serial functions and
                                        ; set to the slave
WAIT            BTFSS   SSPSTAT, BF     ; Wait until transmission is
                                        ; complete (BF=1)
                BRA     WAIT            ; If BF=0, wait
                MOVFF   SSPBUF, PORTD   ; Output serial buffer data to
                                        ; PORTD LEDs
                BRA     WAIT
                END
```

Let us now explain the above program. First, consider the master PIC18F4321. The CKE bit (bit 6) in the SSPSTAT is set to one so that data transmission will occur from an active to an idle (HIGH to LOW) clock. Next, the register SSPCON1 is configured in order to set up the parameters for serial transmission. The bit SSPEN (bit 5) in the SSPCON1 is set to HIGH in order to enable the three pins, namely SCK, SDO, and SDI. Writing 0000 to bits 3-0 of the SSPCON1 register define the master mode operation with a clock of Fosc/4.

The following PIC18F instructions accomplish this:

```
MOVLW   0x20
MOVWF   SSPCON1     ; Enable serial functions and set to master
                    ; and Fosc/4
```

Next, consider the PIC18F assembly language program for the slave; the four bits (bits 3-0) of the slave's SSPCON1 are initialized with 0101. This will place the microcontroller in the slave mode, and also, the $\overline{SS}$ pin will be disabled since there is only one serial device in this example. Note that the $\overline{SS}$ pin is required if multiple slave devices are used. Also, the SCK pin will be used as the clock.

Let us now briefly explain the program logic. The assembly language program for the master, the PIC18F4321, will first perform all initializations, input the switches, and place it in the WREG. The program will then call a subroutine called SERIAL_WRITE. The subroutine moves the switch inputs into the SSPBUF register. As soon as the serial parameters for the master such as the SCK clock pin is set up, data is automatically transmitted to the slave device. Once all the data has been written, the BF bit (bit 0) in the SSPSTAT register of the master microcontroller will go to HIGH indicating completion of transmission.

The program for the slave microcontroller waits in a loop until the BF flag in its SSPSTAT register goes to HIGH, indicating that the transmission is completed. The switch values from the slave's SSPBUF register are outputted to the LEDs connected at PORTD using the MOVFF instruction as follows:

```
MOVFF   SSPBUF, PORTD   ; Move serial buffer value to PORTD.
```

First, the master and the slave microcontrollers need to be downloaded with the above assembled programs using PICKit3. Then, upon hardware reset**,** the master PIC18F4321 will jump to address 0x70 (arbitrarily chosen address) while the slave PIC18F4321 will jump to address 0x100 (arbitrarily chosen address). Note that both microcontrollers do not need to be reset at the same time. Also, both the master and the slave start executing the respective programs independently. The master microcontroller performs initializations, moves switch data input continuously, and waits in the GET_DATA loop. The slave microcontroller also performs initializations, and then waits in the WAIT loop until BF = 1. As soon as the serial communication is established between the master and the slave, the master transmits the contents of SSPBUF via its SDO pin to the slave's SDI pin using the SCK clock. The switch data is transferred to the slave's SSPBUF register. After completion of the transfer, the slave's BF bit in the SSPSTAT register becomes 1. The slave then outputs this data to the LEDs via PORTD.

(b)

C language programs for the master PIC18F4321 and the slave PIC18F4321 can be written using the following steps as the guidelines:

Master PIC18F4321

1. Configure PORTB as input with SDO and SCK as outputs.
2. Select CKE (SPI clock select bit) using the master's SSPSTAT register.
3. Enable serial functions and select the master mode with a clock such as fosc/4 using the CCPCON1 register.
4. Place switch values into the master's Serial Buffer register (SSPBUF).
5. Wait in a loop, and check whether BF (Buffer Full) bit in the master's SSPSTAT register is 1, indicating completion of transmission.

6. As soon as BF = 1, the program returns from the subroutine, and branches to Step 4.

Slave PIC18F4321

1. Initialize SDI and SCK pins as inputs and PORTD as an output. Note that the SCK is controlled by the master, and therefore, is configured as an input by the slave.
2. Select slave CKE same as the master CKE (high to low clock in this example) using the slave's SSPSTAT register.
3. Enable serial functions, disable the $\overline{SS}$ pin, and select slave mode using the slave's SSPCON1 register.
4. Wait in a loop, and check whether BF = 1 in the slave's SSPSTAT register.
5. If BF = 0, wait. However, if BF = 1, output the contents of the slave's Serial Buffer register (SSPBUF) to slave's PORTD to turn LEDs ON or OFF based on the switches.

// C-code for the master SPI PIC18F4321 device:

```
#include <p18f4321.h>
void SPI_out(unsigned char);

void main (void)
{
unsigned char output;
TRISCbits.TRISC5 = 0;          // Configure SDO as output
TRISCbits.TRISC3 =0;           // Configure SCK as output
ADCON1=0x0F;                   // Configure PORTB to be digital input
SSPSTAT= 0x40;                 // Transmission occurs on high to low clock
SSPCON1 = 0x20;                // Enable serial functions and set as master device
        while(1){
        output = PORTB;        // Move switch value to output
        SPI_out(output);       // Send variable 'output' to SPI_out
        }
}
void SPI_out(unsigned char SPI_data)
{
        SSPBUF = SPI_data;     // Place switch value into the serial buffer
        while (SSPSTATbits.BF == 0);   // Wait for transmission to finish
}
        // C-code for the slave PIC18F4321 device:

#include <p18f4321.h>
void main (void)
{
TRISCbits.TRISC4 = 1;          // Configure SDI as input
TRISCbits.TRISC3 =1;           // Configure SCK as input
TRISD=0x00;                    // ConfigurePORTD as output for LEDs
SSPSTAT= 0x40;                 // Transmission occurs on high to low clock
SSPCON1 = 0x25;                // Enable serial functions and disable the slave device
        while(1){
```

```
        while (SSPSTATbits.BF == 0);    // Wait for transmission to finish
        PORTD=SSPBUF;                   // Move serial buffer to PORTD
        }
}
```

11.4.2 PIC18F I²C (Inter-Integrated Circuit) mode

As mentioned before, I²C protocol was introduced by Philips during the 80's. This serial I/O was originally developed to interface IC (Integrated Circuit) chips on a PCB (Printed Circuit Board). I²C is synchronous and is based on master/slave operation. This protocol uses two pins: one for clock and one for data. They are SCL (Serial Clock pin for I²C mode multiplexed with bit 3 of Port C) and SDA (Serial Data pin for I²C mode multiplexed with bit 4 of Port C). There is no need for Slave Select ($\overline{SS}$) like the SPI mode. Any number of masters and slaves can be connected using these two signal lines, SCL and SDA. Both signals are bidirectional. Figure 11.9 shows the block diagram of a single master interfaced with a single slave.

The I²C protocol specifies that in a multimaster/multislave system, the master that initiates a data transfer on the bus is considered as the bus master with all the slave devices connected to the bus as the slaves. SCL and SDA are provided with open drain outputs (pull-up resistors are needed) so that multiple devices can be connected to these lines. Microchip recommends the following pull-up resistor values based on data transfer rate: 4.7 k for data rate less than 100 kbps, 2.2k for data rate of 100 kbps, and 1k for data rate of 400 kbps. Note that bps stands for bits per second.

Each slave is identified by a unique 7-bit address with data divided into bytes. Note that addressing is done in hardware in SPI while addressing is done in I²C in software. Data transfer in I²C uses a few control bits. They typically include a START , READ/WRITE, and STOP signals. In order to transfer data using I²C, the active master sends a START condition. All slave devices connected to the bus will receive this signal, and will wait for the incoming data. Next, the master will send a 7-bit address for the slave (to be selected) along with a read/write signal bit. Upon receipt of this 8-bit information, each slave will compare the master's transmitted 7-bit address with its own 7-bit address. The selected slave with the matched address will respond by sending an ACKNOWLEDGE signal. Upon receipt of this ACKNOWLEDGE signal from the selected slave, the master sends or receives data based on the read/write signal. The master will send a STOP condition and will release the bus. The slaves will then wait for the next data transfer.

As mentioned before, the I²C bus includes a number of conditions. These conditions typically include "START of a data transfer, "STOP" a transfer, and "ACKNOWLEDGE" a transfer. We will now briefly describe these conditions.

A START condition indicates that a device would like to transfer data on the I²C bus. In the START condition, SDA is pulled LOW while SCL is HIGH. The PIC18F takes care of the timing details. However, the programmer needs to tell the PIC18F when the START condition is desired, and then check for completion of the transfer.

A STOP condition indicates that a device has completed its transfer on the I²C bus, and would like to release the bus. Upon releasing the bus, other devices may use the bus for data transmission. The indication of a STOP condition is the release of the SCL line followed by the release of the SDA line. Note that releasing a line means that the line floats HIGH. Once the STOP condition is completed, both SCL and SDA will be HIGH. This is considered to be an "Idle state". Once a bus is idle, a START condition can be used to transfer data.

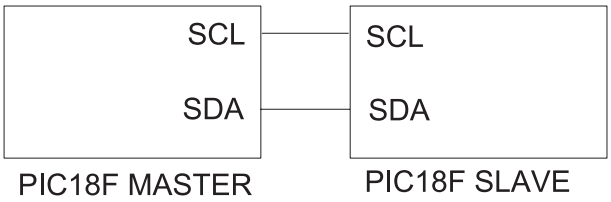


FIGURE 11.9 Master-Slave Interface in I²C mode

Data is sent on the SDA line and the clock is generated on the SCL line. Data is only considered valid when SCL is HIGH. Data is allowed to change when SCL is LOW. Eight bits of data are sent on the bus.

Finally, consider "ACK" (Acknowledge) condition. The ACK condition is used to acknowledge completion of the transfer of a byte by driving the SDA line LOW during the 9th clock pulse of SCL. Nine bits are transferred over the bus as follows. First, eight bits are transferred using the clock, and then the device receiving data will take over the bus for one bit. If this bit is LOW, the device is sending an ACK signal.

The PIC18F I²C uses the same Master Synchronous Serial Port (MSSP) as SPI. Unlike the SPI, the I²C protocol uses only two pins. They are pin 18 Serial Clock (SCL) and pin 23 Serial Data (SDA).

Figure 11.10 shows the timing diagram for the "I²C write transmission sequence" between the PIC18F Master and the Slave. As mentioned before, the I²C protocol contains only two lines: one for data and one for clock. The Master will take control of the communication by using the clock. The master decides when to start transferring data or when to stop the transmission. However, the slave can hold the clock when more data needs to be sent using a feature "clock stretching". This can be accomplished by programming a PIC18F register to be discussed later in this section.

In the figure, the master will drive the SDA line to LOW while maintaining the SCL line HIGH in order to inform the slave of the start of the transmission process. This is called the START condition. For the STOP condition, the SDA line is toggled to HIGH while SCL is maintained HIGH.

The I²C interface can have more than one slave device. Therefore, in the data transfer sequence, the first 8 bit of information includes a 7-bit address of the slave along with one bit for the R/$\overline{W}$ bit (R/$\overline{W}$ = 0 in the figure since this is write transmission). The master identifies the selected slave from this information. After receiving this 8-bit information, the slave sends an ACK signal to the master acknowledging receipt of this information. The master then sends the actual data to the slave. Upon receipt of this 8-bit data, the slave will send ACK signal to the master, and the process continues until the slave receives the STOP condition. The programmer can generate the START, STOP, and ACK signals by programming certain registers in the PIC18F. This is basically transparent to the

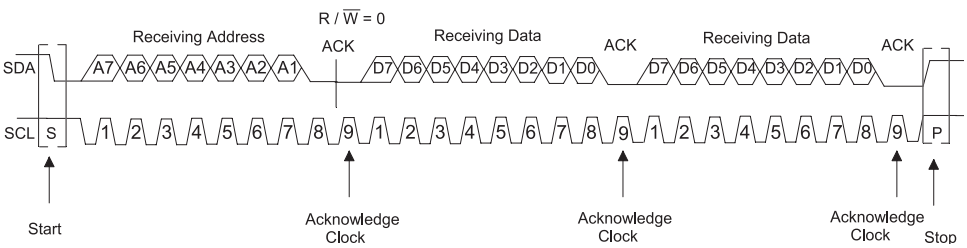


FIGURE 11.10 I²C write transmission sequence

programmer. Figure 11.11 shows the MSSP block diagram for the I²C.

SSPSR is the shift register used for shifting data in or out. SSPBUF is the register to which data bytes are written to or read from. SSPADD register holds the slave device address when the MSSP is configured in I²C Slave mode. When the MSSP is configured in Master mode, the lower seven bits of SSPADD act as the Baud Rate Generator reload value. When SSPSR receives a complete byte, it is transferred to SSPBUF and an interrupt is set. A write to SSPBUF will write to both SSPBUF and SSPSR.

PIC18F MSSP Registers for I²C mode The following registers included in the MSSP module are used for the I²C mode:

- SSPSTAT (Figure 11.12)
- SSPCON1 (Figure 11.13)
- SSPCON2 (Figure 11.14)
- SSPADD (Figure 11.15)

SSPSTAT register (Figure 11.12) Upper two bits (SMP and CKE) are read/write and other bits are read-only.

- BF (Buffer Full): If this is 1, it means the SSPBUF is loaded with 8-bit. This bit will be cleared if the SSPBUF is read.
- R/$\overline{W}$ (Read/Write): This bit tells when the Master is reading from or writing to the slave device.
- S (Start): If we set this bit, the START condition will occur.
- P (Stop): If we set this bit, the STOP condition will occur.
- CKE (Clock Edge) enables (synchronizes with) or disables the SMBus. Note that the SMBus (System Management Bus) is a serial protocol based on I²C. The SMBus is a two-wire bus commonly used in computer motherboards for monitoring voltage and temperature. SMBus only goes up to 100kHz.
- SMP (Slew rate control): Enabled (SMP cleared to 0) for 400kHz and disabled (SMP set to 1) for 100kHz. Note that slew rate control is primarily an EMI (ElectroMagnetic Interference) control feature. When using slow-speed along with long-wire IC setups,

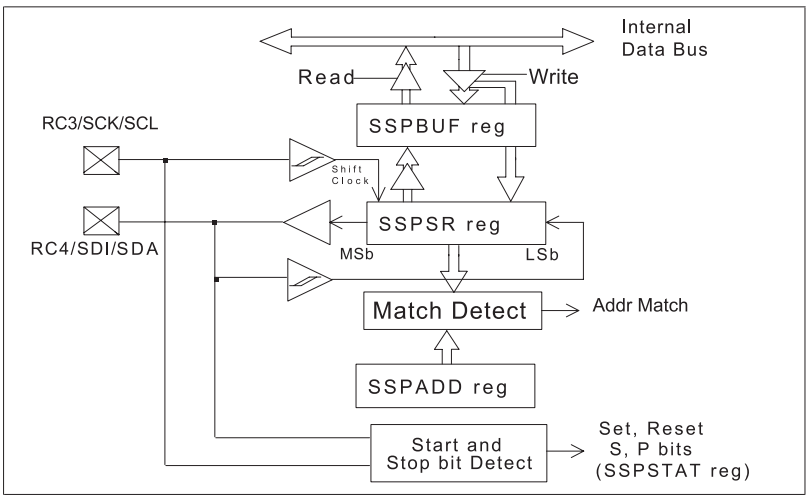
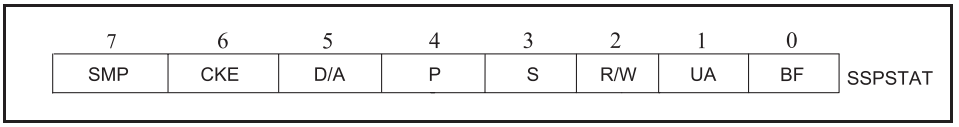


FIGURE 11.11 MSSP Block Diagram (I²C)



bit 7 **SMP:** Slew Rate Control bit
In Master of Slave mode:
1 = Slew rate control disabled for Standard Speed mode (100 kHz and 1 MHz)
0 = Slew rate control enabled for High-Spee mode (400 kHz)

bit 6 **CKE:** SMBus Select bit
In Master or Slave mode:
1 = Enable SMBus specific inputs
0 = Disabel SMBus specific inputs

bit 5 **D/$\overline{A}$:** Data/$\overline{\text{Address}}$ bit
In Master mode:
Reserved.
In Slave mode:
1 = Indicates that the last byte received or transmitted was data
0 = Indicates that the last byte received or transmitted was address

bit 4 **P:** Stop bit
1 = Indicates that a Stop bit has been detected last
0 = Stop bit was not detected last
Note: This bit is cleared on Reset and when SSPEN is cleared.

bit 3 **S:** Start bit
1 = Indicates that a Start bit has been detected last
0 = Start bit was not detected las
Note: This bit is cleared on Reset and when SSPEN is cleared.

bit 2 **R/$\overline{W}$**: Read/$\overline{\text{Write}}$ Information bit (I^2C mode only)
In Slave mode:
1 = Read
0 = Write
Note: This bit hold the R/$\overline{W}$ bit information following the last address match. This bit is only valid from the address match to the next Start bit, Stop bit or not ACK bit.
In Master mode:
1 = Transmit is in progress
0 = Transmit is not in progress
Note: ORing this bit with SEN, RSEN, PEN, RCEN or ACKEN will indicate if the MSSP is in Active mode.

bit 1 **UA:** Update Adrress bit (10-bit Slave mode only)
1 = Indicates that the user needs to update the address in the SSPADD register
0 = Address does not need to be updated

bit 0 **BF:** Buffer Full Status bit
In Transmit mode:
1 = SSPBUF is full
0 = SSPBUF is empty
In Receive mode:
1 = SSPBUF is full (does not include the ACK and Stop bits)
0 = SSPBUF is empty (does not include the ACK and Stop bits)

FIGURE 11.12 SSPSTAT: MSSP STATUS REGISTER (I^2C^{TM} MODE)

enabling the slew rate control can provide additional distance of bus length with appropriate pull-ups. In our case, SMP will be set to one to disable.

SSPCON1 register (FIGURE 11.13) The device will be selected as Master or Slave using the SSPCON1 register. This register also enables or diables the MSSP module

WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
bit 7							bit 0

bit 7 **WCOL:** Write Collision Detect bit
In Master Trransmit mode:
1 = A write tothe SSPBUF register was attemted while the I²C conditions were not valid for trasmission to be started (must be cleared in software
0 = No collision
In Slave Transmit mode:
1 = The SSPBUF register is written while it is still trasmittting the previous word (must be cleared in software)
0 = No collision
In Slave Transmit mode:
1 = The SSPBUF register is written while it is still transmitting the previous word (must be cleared in software)
0 = No collision
In Receive mode (Master or Slave modes):
This is a "don't care" bit.

bit 6 **SSPOV:** Recieive Overflow Indicator bit
In Receive mode:
1 = A byte is reeived while the SSPBUF register is still holding the previous byte (must be cleared in software)
0 = No overflow
In Transmit mode:
This a "don't care" bit in Transmit mode.

bit 5 **SSPEN:** Master Synchronous Serial Port Enable bit
1 = Enables the serial port and configures th SDA and SCL pins as the serial port pins
0 = Disables serial port and configures these pins as I/O port pins
Note: When enabled, the SDA and SCL pins must be properly configured as inputs.

bit 4 **CKP:** SCK Release Control bit
In Slave mode:
1 = Release clock
0 = Holds clock low (clock stretch), used to ensure data setup time
In Master mode:
Unused in this mode.

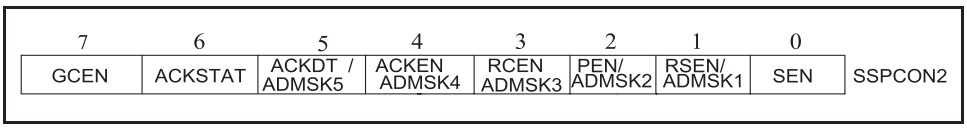
bit 3-0 **SSPM3:SSPM0:** Master Synchronous Serial Port Mode Select bits
1111 = I²C Slave mode, 10-bit address with Start and Stop bit interrups enabled
1110 = I²C Slave mode, 7-bit address with Start and Stop bit interrups enabled
1011 = I²C Firmware Controlled Master mode (slve Idle)
1000 = I²C Master mode, clock = Fosc/(4 * (SSPADD + 1))
0111 = I²C Slave mode, 10-bit address
0110 =I²C Slave mode, 7-bit address

FIGURE 11.13 SSPCON1: MSSP CONTROL REGISTER 1 (I2C™ MODE)

for I²C and releases or holds SCK for the slave.

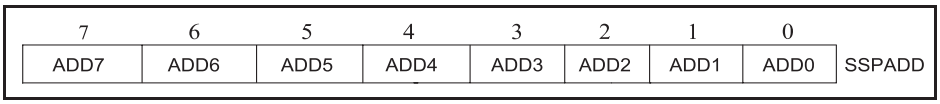
SSPCON2 register (FIGURE 11.14):

- provides acknowledge status bit for the master (received or not received) for the master
- initiates START or STOP conditions for the master
- provides clock stretching. Note that IC devices can slow down communication by clock stretching (that is, by stretching SCL). During an SCL low phase, any IC device on the bus may additionally hold down SCL clock to prevent it to rise high again, enabling them to slow down the SCL clock rate or to stop I²C communication for a while. If SEN is enabled (SSPCON2<0> = 1), RC3/SCK/SCL will be held low (clock stretch) following each data transfer. The clock must be released by setting bit, CKP (SSPCON1<4>).



bit 7 **GCEN:** General Call Enable bit (Slave mode on)
1 = Enable interrupt when a general call address (0000h) is received in the SSPSR
0 = General call address disabled
bit 6 **ACKSTAT:** Acknowledge Status bit (Master Transmit mode only)
1 = Acknowledge was not received from slave 0 = Acknowledge was received from slave
bit 5 **ACKDT/ADMSK5:** Acknowledge Data bit
In Master Receive mode: 1 = Not Acknowledge 0 = Acknowledge
Note: Value that will be transmitted when the user initiates an Acknowledge sequence at
the end of a receive. In Slave mode: 1 = Address masking of ADD5 enabled 0 = Address masking of ADD5 disabled
bit 4 **ACKEN/ADMSK4:** Acknowledge Sequence Enable bit
In Master Receive mode:(1) 1 = Initiate Acknowledge sequence on SDA and SCL pins and transmit ACKDT data bit.
Automatically cleared by hardware.0 = Acknowledge sequence Idle In Slave mode:
1 = Address masking of ADD4 enabled 0 = Address masking of ADD4 disabled
bit 3 **RCEN/ADMSK3:** Receive Enable bit
In Master Receive mode:(1) 1 = Enables Receive mode for I²C 0 = Receive Idle In Slave mode:
1 = Address masking of ADD3 enabled 0 = Address masking of ADD3 disabled
bit 2 **PEN/ADMSK2:** Stop Condition Enable bit
In Master mode:(1)
1 = Initiate Stop condition on SDA and SCL pins. Automatically cleared by hardware. 0 = Stop condition Idle
In Slave mode:1 = Address masking of ADD2 enabled 0 = Address masking of ADD2 disabled
bit 1 **RSEN/ADMSK1:** Repeated Start Condition Enable bit
In Master mode:1 = Initiate Repeated Start condition on SDA and SCL pins. Automatically cleared by hardware.
0 = Repeated Start condition In Slave mode (7-bit Address mode): 1 = Address masking of ADD1 enabled 0 = Address masking of ADD1 disabled
In Slave mode (10-bit Address mode): 1 = Address masking of ADD1 and ADD0 enabled
0 = Address masking of ADD1 and ADD0 disabled
bit 0 **SEN:** Start Condition Enable/Stretch Enable bit (1)
In Master mode:
1 = Initiate Start condition on SDA and SCL pins. Automatically cleared by hardware. 0 = Start condition Idle
In Slave mode:1 = Clock stretching is enabled for both slave transmit and slave receive (stretch enabled)
0 = Clock stretching is disabled Note that I2C devices can slow down communication by clock stretching (that is, by stretching SCL). During an SCL low phase, any I2C device on the bus may additionally hold down
SCL clock to prevent it to rise high again, enabling them to slow down the SCL clock rate or to stop I2C

FIGURE 11.14 SSPCON2: MSSP CONTROL REGISTER 2 (I2C™ MODE)



bit 7-0 **ADD<7:0>:** MSSP Address bits
Note 1: MSSP Address register in 1²C Slave mode. MSSP Baud Rate register in I²C Master mode.

FIGURE 11.15 SSPADD: MSSP ADDRESS REGISTER

SSPADD register (FIGURE 11.15): For the slave device, SSPADD = slave address. For the master device, the 8-bit value of SSPADD will determine the clock frequency in the SCK line. This is called Baud rate. The baud rate formula is:
Baud rate = FOSC/(4*(SSPADD +1))
SSPADD = (FOSC/ (4*Baud rate))-1

Example 11.7 Figure 11.16 shows a block diagram for interfacing two PIC18F4321 in I²C mode. One of the microcontrollers is the master while the other is the slave. The master PIC18F4321 will input four switches via bits 0-3 of PORTB (Switch SW0 connected to bit 0 of Port B, Switch SW1 to bit 1, SW2 to bit 2 and SW3 to bit 3), and then transmit the 4-bit data using its SDA pin to the slave's SDA pin. The slave PIC18F4321 will output this data to four LED's (LED0 connected to bit 0 of Port D, LED1 to bit 1, LED 2 to bit 2, and LED 3 to bit 3), and turn them ON or OFF based on the switch inputs. For example, LED0 will be turned ON if SW0 is one (Open) and will be turned OFF if SW0 is zero (closed), LED1 will be turned ON if SW1 is one (Open) and will be turned OFF if SW1 is zero (closed), and so on.

(a) Write a PIC18F assembly language program at 0x70 for the master PIC18F4321 that will configure PORTB, initialize STKPTR to 5, initialize SSPSTAT and SSPCON1, input switches, and call a subroutine called SERIAL_WRITE to place this data into its SSPBUF register.
Also, write a PIC18F assembly language program at 0x100 for the slave PIC18F4321 that will configure PORTD, initialize SSPSTAT and SSPCON1 registers, input data from its SDA pin, place it in the slave's SSPBUF, and then output to the LED's.

(b) Repeat (a) using C language. The master PIC18F4321 will configure PORTB, initialize SSPSTAT and SSPCON1, input switches, and place this data into its SSPBUF register.
Also, write a C language program for the slave PIC18F4321 that will configure PORTD, initialize SSPSTAT and SSPCON1 registers, input data from its SDA pin, place it in the slave's SSPBUF, and then output to the LED's.

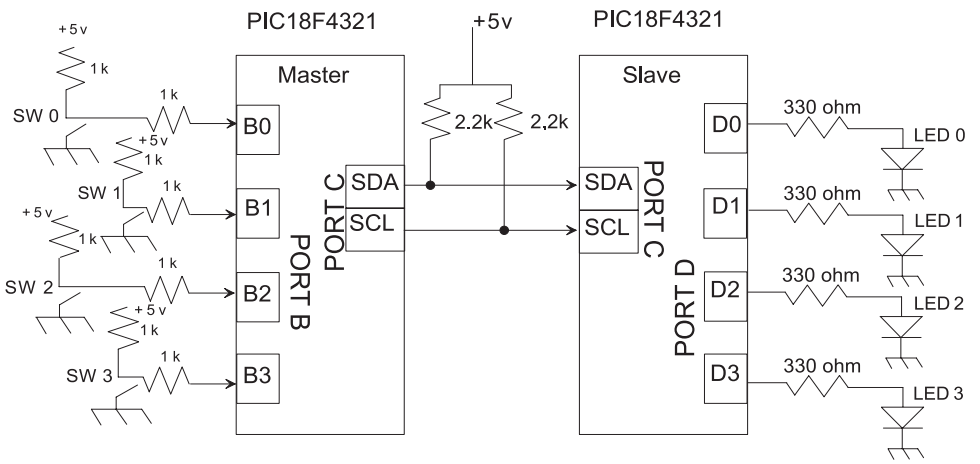


FIGURE 11.16 Figure for Example 11.7

Solution

Master:

1. Configure PORTB as input for switches. Also, configure Master SDA and SCL as inputs. See "Note" (bit 5, SSPCON1, Figure 11.13)
2. Initialize SSPSTAT = 0x80, SSPCON1 = 0x38, SSPADD = 0x01, SSPCON2 = 0x00

- SSPSTAT (Figure 11.12) = 0x80 means SMP = 1 (Slew rate control OFF), CKE = 0 (Disable SMBus). Since other bits are read-only, they are don't cares initially and assumed 0's.
- SSPCON1 (Figure 11.13) = 0x38 means WCOL = 0 (No collision), SSPOV = 0 (Don't care mode in transmit mode; assumed 0), SSPEN = 1 (Enables serial port and configures SDA and SCL), CKP = 1 (Unused in master mode; assumed 1 arbitrarily), SSPM3:SSPM0 = 1000 (I^2C master mode).

  Since SSPEN = 1 in this example, SDA and SCL pins are configured as inputs. Hence, configure SDA (I^2C data pin multiplexed with RC4) and SCL (I^2C clock multiplexed with RC3) as inputs using PORTC using TRISC register.
- SSPADD (Figure 11.15) = 0x01. SSPADD is the baud rate register and its contents are computed from SSPADD = (FOSC/ (4*Baud rate))-1. With FOSC = 1MHz and Baud rate = 100 kHz, SSPADD is 1.5. Hence, '1' is loaded into SSPADD.
- SSPCON2 (Figure 11.14) = 0x00. See details in the figure for SSPCON2.

3. Write functions that will tell the Master to wait for idle, trigger start condition, send stop condition, and write information to the slave.
4. In the main, set up ADCON1, TRISB, and run the function in a loop with sequence: start, write address, write data, stop.

Slave:

1. Configure PORTD as an output for LEDs. Configure SDA and SCL as inputs in PORTC.
2. Configure SSPSTAT, SSPCON1, SSPADD. In this case, SSPSTAT = 0x80, SSPCON1 = 0x36, and SSPADD = slave address.

- SSPSTAT = 0x80 means SMP = 1 (Slew rate control OFF), CKE = 0 (Disable SMBus). The other bits are read-only.
- SSPCON1 = 0x36 means WCOL = 0 (No collision), SSPOV = 0 (Don't care mode in transmit mode; assumed 0), SSPEN = 1 (Enables serial port and configures SDA and SCL), CKP = 1 (release clock), SSPM3:SSPM0 = 0110 (I^2C Slave mode).
- SSPADD contains slave address.
- SSPCON2 = 0x01 to enable clock stretching.

3. Check for information sent from the master. To know whether the master is sending address or data, the slave will check at bits S, D/$\overline{A}$, R/ $\overline{W}$ and BF in the SSPSTAT register. Note that S=1, R/$\overline{W}$ = 0, and BF = 1 will indicate that the master is sending information and the SSPBUF contains address or data that needs to be read or cleared. Next, the D/ $\overline{A}$ bit can be checked:
   D/$\overline{A}$ = 0: The master is sending address; ignore by doing the dummy read to clear the SSPBUF.

D/$\overline{A}$ = 1: The master is sending data; output SSPBUF data to PORTD

4. In the main, make infinite loop to keep reading data from Master.

(a)

```
; PIC18F assembly language program for the Master

INCLUDE <P18F4321.INC>

SLAVE_ADDR      EQU   0x30              ; 7-bit Slave address (arbitrarily chosen)
                                        ; + Write bit
                ORG 0x00
                GOTO MAIN
                ORG 0x70
MAIN            BSF TRISC, RC3          ; Configure RC3/SCL as input
                BSF TRISC, RC4          ; Configure RC4/SDA as input
                MOVLW  5
                MOVWF  STKPTR           ; Initialize STKPTR since subroutines used
                MOVLW 0x0F
                MOVWF ADCON1            ; ADCON1 = 0x0F, make PORTB as an input
                MOVLW 0x38
                MOVWF SSPCON1           ; I2C master mode
                CLRF SSPCON2            ; Reset/clear SSPCON2
                MOVLW 1
                MOVWF SSPADD            ; Baudrate 100kHz with 1MHz clock
                MOVLW 0x80
                MOVWF SSPSTAT           ; Slew rate control disable
START           CALL IDLE_WAIT          ; Wait for idle
                BSF SSPCON2, SEN        ; Send start condition
WRITE_ADDR      CALL IDLE_WAIT          ; Wait for idle
                MOVLW SLAVE_ADDR        ; Load slave address to WREG
                CALL MASTER_WRITE       ; Send Address to slave
WRITE_DATA      CALL IDLE_WAIT          ; Wait for idle
                MOVF PORTB, W           ; Move switch value to WREG
                CALL MASTER_WRITE
STOP            CALL IDLE_WAIT          ; Wait for idle
                BSF SSPCON2, PEN        ; Send stop condition
                BRA START               ; LOOPING
MASTER_WRITE    MOVWF SSPBUF            ; Move switch value to Serial buffer
                RETURN
IDLE_WAIT       BTFSC  SSPSTAT, R_W     ; Check if the slave is reading
                BRA      IDLE_WAIT
                RETURN
                END
; PIC18F assembly language for the Slave
<INCLUDE <P18F4321.INC>

SLAVE_ADDR      EQU    0x30
DUMMY           EQU    0x60             ; dummy place to clear SSPBUF
                ORG    0x00
                GOTO  MAIN
```

```
                ORG 0x70
MAIN            BSF TRISC, RC3          ; Configure RC3/SCL as input
                BSF TRISC, RC4          ; Configure RC4/SDA as input
                MOVLW 0xF0
                MOVWF TRISD             ; PORT D outputs
                MOVLW SLAVE_ADDR        ; Slave address to WREG
                MOVWF SSPADD            ; SSPSTAT = slave address
                MOVLW 0x36
                MOVWF SSPCON1           ; I2C slave mode
                MOVLW 0x80
                MOVWF SSPSTAT           ; Slew rate control disable
                MOVLW 0x01
                MOVWF SSPCON2           ; Enable clock stretching
CHECK           BTFSS SSPSTAT, S        ; Check if S = 1
                BRA CHECK
                BTFSC SSPSTAT, R_W      ; Check if R/nW = 0
                BRA CHECK
                BTFSS SSPSTAT, BF       ; Check if BF = 1
                BRA CHECK
                BTFSS SSPSTAT, D_A      ; Check if D/nA = 1
                BRA DUMMY_READ
                MOVFF SSPBUF, PORTD     ; PORTD = SSPBUF
                BSF SSPCON1, CKP        ; SSPCON1.CKP = 1, release clock
                BRA CHECK               ; Looping
DUMMY_READ      MOVFF SSPBUF, DUMMY     ; DUMMY = SSPBUF, clear SSPBUF
                BSF SSPCON1, CKP        ; Release clock
                BRA CHECK               ; Looping
                END
```

(b)

```
//C-program for the Master

#include <P18F4321.h>
#include <stdio.h>
#define SLAVE_ADDRESS 0x30
void I2C_Master_Init(unsigned long bRate)
{
        TRISCbits.RC3 = 1;              // SCL as input
        TRISCbits.RC4 = 1;              // SDA as input
        SSPCON1 = 0x38;                 // I2C master mode
        SSPCON2 = 0x00;
        SSPADD = (1000000/(4*bRate))-1;   // I2C Baud Rate formula with 1MHz OSC
        SSPSTAT = 0x80;                 // slew rate control disable for 100khz i2c BaudRate
}
void I2C_Master_Wait()
{
        while ((SSPSTATbits.R_W)); // wait for idle, not writing
}
void I2C_Master_Start()
```

```
{
        I2C_Master_Wait();
        SSPCON2bits.SEN = 1;
}
void I2C_Master_Stop()
{
        I2C_Master_Wait();
        SSPCON2bits.PEN = 1;
}
void I2C_Master_Write(unsigned char writeData)
{
        I2C_Master_Wait();
        SSPBUF = writeData;
}
void main()
{
        ADCON1 = 0x0F;              // PORTB as input
        I2C_Master_Init(100000);    // 100kHz baud rate
        while(1)
        {
        I2C_Master_Start(); //Start condition
        I2C_Master_Write(SLAVE_ADDRESS);    // 7 bit address + Write
        I2C_Master_Write(PORTB);  // Write data
        I2C_Master_Stop();          // Stop condition
        }
```

QUESTIONS AND PROBLEMS

11.1 What is the main purpose of:
(a) Capture mode
(b) Compare mode
(c) PWM mode
Indicate whether the PIC18F4321 CCPx pin is configured as input or output for each of these modes.

11.2 Identify timer assignments for each of the CCPx modes in the PIC18F4321.

11.3 Assume PIC18F4321. Write a PIC18F assembly language program that will measure the period of a periodic pulse train on the CCP1 pin using the capture mode. The 16-bit result will be performed in terms of the number of internal (Fosc/4) clock cycles, and will be available in the TMR1H:TMR1L register pair. Use 1:1 prescale value for Timer1. Also, write a C-program to accomplish this.

11.4 Assume PIC18F4321. Write a PIC18F assembly language program at address 0x200 that will generate a square wave on the CCP1 pin using the Compare mode. The square wave will have a period of 20 ms with a 50% duty cycle. Use Timer1 internal clock of 4MHz with 1:2 prescale value.

11.5 Write a PIC18F assembly language program at 0x100 to generate a 16 kHz PWM with a 75% duty cycle on the CCP1 pin of the PIC18F4321. Assume 4 MHz crystal.

11.6 Assume PIC18F4321. Write a C language program that will generate a square wave on the CCP1 pin using the Compare mode. The square wave will have a period of 20 ms with a 50% duty cycle. Use Timer1 with a 4MHz internal clock and 1:2 prescale value.

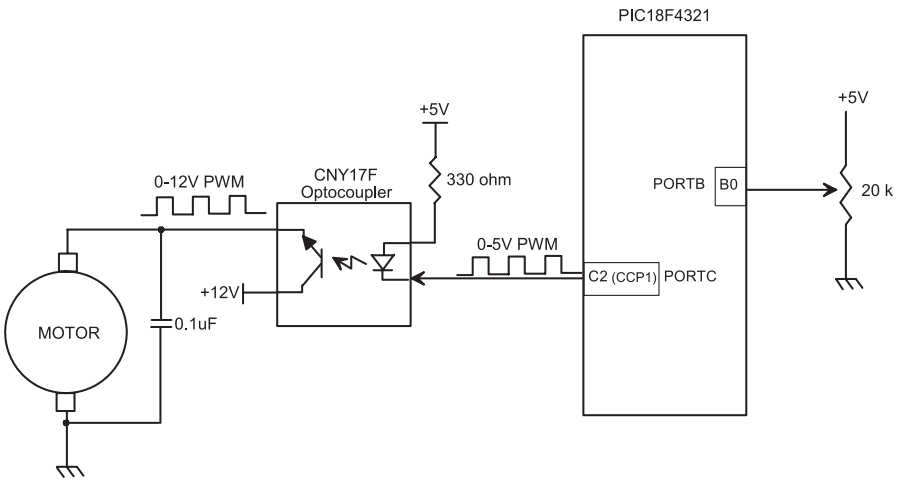


FIGURE P11.8

11.7 Write a C language program to generate a 16 kHz PWM with a 75% duty cycle on the CCP1 pin of the PIC18F4321. Assume 4 MHz crystal.

11.8 It is desired to change the speed of a DC motor by dynamically changing its pulse width using a potentiometer connected at bit 0 of PORTB (Figure P11.8). Note that the PWM duty cycle is controlled by the potentiometer. The PIC18F4321 will input the potentiometer voltage via its on-chip A/D converter as eight bits, generate 800Hz PWM waveform on the CCP1 pin, and will then change the speed of the motor as the potentiometer voltage is varied. Use a 4MHz crystal and a TMR2 prescaler value of 16. Ignore the fractured part of the duty cycle. Using polled ADC:

(a) Write a PIC18F assembly language program to accomplish this.
(b) Write a C-language program to accomplish this.

11.9 Figure P11.9 shows a simplified diagram interfacing the PIC18F4321 to a DC motor via the CNY17F Optocoupler. The purpose of this example is to control the speed of a DC motor by inputting two switches connected at bit 0 and bit 1 of PORTD. The motor will run faster or slower based on the switch values (00 or 11), but will not provide any measure of the exact RPM of the motor. When both switches are closed (00), a PWM signal at the CCP1 pin of the PIC18F4321 with 50% duty cycle will be generated. When both switches are open (11), a PWM signal at the CCP1 pin of the PIC18F4321 with 75% duty cycle will be generated. Otherwise, the motor will stop, and the program will wait in a loop. If switches are closed (00), the motor will run using the 4kHz PWM pulse with 50% duty cycle. If both switches are open (11), the motor will run using the same PWM pulse at a faster speed with a duty cycle of 75%. The program will first perform initializations, and wait in a loop until the switches are 00 or 11. Assume 4MHz crystal.

(a) Write a PIC18F assembly language program at address 0x100 to accomplish this.

(b) Write a C-program to accomplish this.

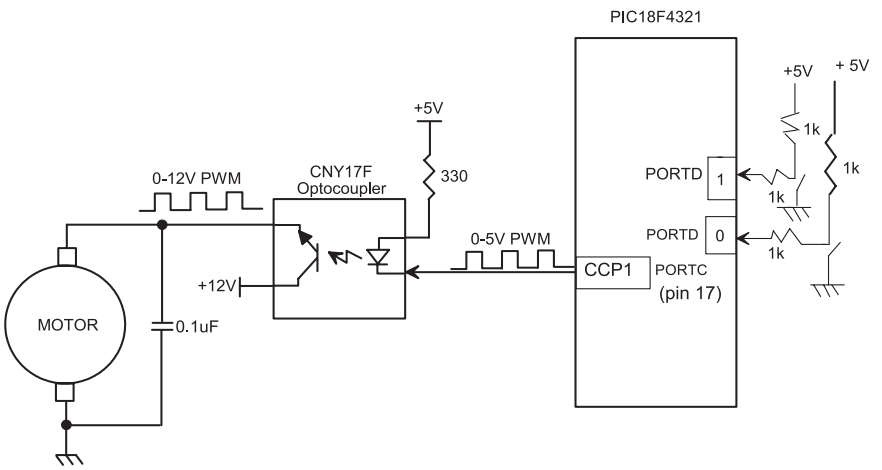


FIGURE P11.9

11.10 Summarize the basic differences between SPI and I^2C modes relating them to the PIC18F.

11.11 Assume that two PIC18F4321's are interfaced in the SPI mode. A switch is connected to bit 0 of PORTD of the master PIC18F4321 and an LED is connected to bit 5 of PORTB of the slave PIC18F4321. Write C language programs to input the switch via the master, and output it to the LED of the slave PIC18F4321. If the switch is open, the LED will be turned ON while the LED will be turned OFF if the switch is closed.

11.12 Repeat Problem 11.11 using I^2C mode.

APPENDIX A: ANSWERS TO SELECTED PROBLEMS

Chapter 2

2.1 Microprocessor: CPU on a single chip.
A typical microcontroller contains the CPU, memory, I/O, timers, A/D converter-all on a single chip.

2.4 (a) sign = 0, carry = 0, zero = 0, overflow = 0.
(d) sign = 1, carry = 0, zero = 0, overflow = 1.

2.5 PUSH means writing to the stack.
POP means reading from the stack.

2.6 (a) 20BE
(b) (20BE) = 05, (20BF) = 02.

2.7 To load the program counter with the address of the first instruction to be executed.

2.14 Instruction fetch and execution are overlapped.

Chapter 3

3.2 $2^{24} = 2^{10} \times 2^{10} \times 2^{4} = 16$ MB

3.5 (a) 16,384
(b) 128 chips
(c) 4 bits

3.7 (a) 20
(b) 6 x 64 decoder

3.8 14 unused address pins Available.
Maximum Directly Addressable Memory = 16 Megabytes

3.10 Memory Chip #1 EC00H - EDFFH
Memory Chip #2 F200H - F3FFH

3.11 (a) ROM Map: 0000H - 07FFH
RAM Map: 2000H - 27FFH

3.16 Using standard I/O, the microcontroller uses an output pin such as M/$\overline{IO}$ pin to distinguish between memory and I/O. Also, the microcontroller uses IN and OUT instructions to perform I/O operation in standard I/O.
Using memory-mapped I/O, the microcontroller uses an unused address pin to distinguish between memory and I/O. The ports are mapped as memory locations. Memory-oriented instructions are used for performing I/O operation.
The PIC18F uses only memory-mapped I/O.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

3.17 Memory-mapping provides the physical addresses for the microcomputer's main memory while memory-mapped I/O maps port addresses into memory locations.

3.20 Interrupt address vector is the starting address of the service routine.

Chapter 4

4.2 Yes.

4.3 No.

4.7 Use the following identities: $a \oplus a$ and $a \oplus 0 = a$ and $(a \oplus b) \oplus a = b$

4.8 Product = $0000\ 0000\ 0000\ 0100_2$

4.9 Quotient = -8, Remainder = -1. The sign of the remainder is the same as the sign of the dividend unless remainder is zero.

Chapter 5

5.2 Flash memory.

5.4 SRAM.

5.9 (a) PC contains addresses of instructions in program memory whereas the FSRs point indirectly to data memory.

5.10 0152H.

5.17 (a) Implied mode.

5.17 (b) Literal mode.

Chapter 6

6.1
```
MOVF    0x30, W
ADDWF   0x40, W
MOVWF   0x50
```

6.3 (a) [0x20] = FFH

6.6
```
CLRF    0x20
SETF    0x22
```

6.10 (a) MOVLW 0

6.13 Assume N1 and N2 are already loaded into registers 0x20 and 0x21 respectively.
```
INCLUDE     <PIC18F4321.INC>
ORG         0x100
SWAPF       0x21, F         ; Swap nibbles of N2 in 0x21
MOVF        0x20, W         ; Move [0x20] into WREG
ADDWF       0x21, W         ; Add [WREG] with [0x21], store result in WREG
MOVWF       0x30            ; Store result in 0x30
SLEEP
END
```

6.21 Since the remainder can be discarded, unsigned division can be accomplished by logically shifting the 16-bit unsigned number, 0x0124 once to the right.

```
        INCLUDE     <PIC18F4321.INC>
        ORG         0x100
        BCF         STATUS, C       ; Clear Carry flag to 0
        RRCF        0x20, F         ; Right shift [0x20][0x21] once
        RRCF        0x21, F
        SLEEP
        END
```

Chapter 7

7.2 Assume data is already loaded into 0x30.

```
        INCLUDE     <PIC18F4321.INC>
        ORG         0x200
        MOVFF       0x30, 0x40      ; Copy data in 0x40
        SWAPF       0x30, F         ; Move data into low 4 bits
        MOVLW       0x0F            ; Move mask data into WREG
        ANDLW       0x30, F         ; One unsigned 8-bit data in 0x30
        ANDLW       0x40, W         ; Another unsigned data in WREG
        MULWF       0x30            ; unsigned multiply data
                                    ; Since result will be 8-bit maximum,
                                    ; PRODL will contain result
        MOVWF       PRODL, 0x31     ; Result in 0x31
        SLEEP                       ; HALT
        END
```

7.7 Assume arrays x[i] and y[i] are already loaded into 0x20 and 0x30 respectively.

```
        INCLUDE     <PIC18F4321.INC>
        ORG         0x100
        CLRF        0x50            ; Clear sum to 0
        LFSR        0, 0x0020       ; Load 0x0020 into FSR0
        LFSR        1, 0x0030       ; Load 0x0030 into FSR1
        MOVLW       D'10'           ; Move 10 (decimal) into counter 0x90
        MOVWF       0x90
LOOP    MOVF        POSTINC0, W     ; Move x[i] into WREG, Increment pointer
        MULWF       POSTINC1        ; unsigned multiply in x[i] * y[i]
        MOVF        PRODL, W        ; Move 8-bit product to WREG
        ADDWF       0x70, F         ; sum in 0x70
        DECF        0x90, F         ; Decrement counter by 1
        BNZ         LOOP            ; Repeat if Z = 0
        SLEEP                       ; 0x70 contains the result and Halt
        END
```

7.11 A number P is divisible by 5 if its least significant packed BCD digit is either 0 or 5.

```
        INCLUDE <18F4321.INC>
        ORG     0x100
        CLRF    0x40          ; Clear  counter 0x40  to  0
        MOVLW   0x0F          ; Load (starting address - 1)  into 0x60
        MOVWF   0x60
        LFSR    0, 0x10       ; Initialize FSR0 with 0x10
BACK    INCF    0x60, F
        MOVLW   0x0F          ; Load  mask data into WREG
        ANDWF   INDF0, F      ; Retain low 4 bits of data in F
        MOVLW   0x00          ; Load 0  into WREG
        SUBWF   POSTINC0, F   ; Subtract 0 from [F],  Increment FSR0
        BZ      DOWN          ; If Z = 1, divisible by 5, increment [0x40]
        MOVLW   0x05          ; Else, load 5 into WREG
        SUBWF   INDF0, F      ; Subtract 5 from [0x60]
        BZ      DOWN
        BRA     DOWN1
DOWN    INCF    0x40, F
DOWN1   MOVLW   0x2D          ; Load last address into WREG
        CPFSEQ  0x60          ; If [0x60] = 0x2D, skip
        BRA     BACK
        SLEEP
        END
```

7.18 Q = 120

Chapter 8

8.2 1 MHz

8.6 (a) SETF PORTC

8.6 (b) CLRF PORTD

8.7 (a) TRISC = 0xFF;

8.7 (b) TRISD = 0;

8.9

```
        INCLUDE   <PIC18F4321.INC>
        SETF      PORTC          ; Configure  PORTC as an input port
        BCF       TRISD, TRISC6  ; Configure bit 6 of PORTD as output
        BCF       PORTD, RD6     ; Turn LED  OFF
BACK    MOVF      PORTC, F       ; Input PORTC
        MOVLW     0x07
        ANDWF     PORTC, F       ; Retain low three bits
        MOVLW     0x00           ; Check for no high switches, 0 is an even number
        SUBWF     PORTC, W
        BZ        LED            ; If no high switches, turn LED ON
        MOVLW     0x03           ; Check for two high switches
        SUBWF     PORTC, W       ; If  two high switches, turn LED ON
```

```
        MOVLW   0x05        ; Check for two high switches
        SUBWF   PORTC, W
        BZ      LED         ; If  two high switches, turn LED ON
        MOVLW   0x06        ; Check for two high switches
        SUBWF   PORTC, W
        BZ      LED         ; If  two high switches, turn LED ON
FINISH  BRA     BACK        ; GO BACK
LED     BSF     PORTD, RD6  ; Turn LED  ON
        BRA     FINISH
        END
```

8.14 (b)

```
        #include <P18F4321.h>
// Declare lookup_table for EBCDIC printer
unsigned char
lookup_table[10]={0xF0,0xF1,0xF2,0xF3,0xF4,0xF5,0xF6,0xF7,0xF8,0xF9};
void main()
{
unsigned char ASCII_value, temp; //Declare variables
TRISC=0xFF; // PortC is input
TRISD=0x00; // PortD is output
ASCII_value=PORTC; // Move ASCII value from PORTC to
//variable ASCII_value
temp=ASCII_value & 0x0F; // Mask lower 4 bits
PORTD=lookup_table[temp]; // Output based on lower 4 bits of the ASCII input
}
```

Chapter 9

9.1 0x08

9.2 The PIC18F4321 does not have any nonmaskable interrupt.
The PIC18F4321 maskable interrupts are: INT0, INT1, INT2.

9.5

```
INTCONbits.INT1IF =0;
INTCON3bits.INT1IE = 1;
INTCON3bits.INT2IF = 0;
INTCON3bits.INT2IE = 1;
INTCON3bits.INT1IP = 1;
INTCON3bits.INT2IP = 0;
RCONbits.IPEN = 1;
INTCONbits.GIEH = 1;
INTCONbits.GIEL = 1;
```

9.12 The three control pins, EN, R/$\overline{W}$, and RS allow the user to let the display know what kind of data is sent. The EN pin latches the data from the D0-D7 pins into the LCD display. Data on D0-D7 pins will be latched on the trailing edge (high-to-low) of the EN pulse. The EN pulse must be at least 450 ns wide. The R/$\overline{W}$ (read/write) pin, allows the user to either write to the LCD or read data from the LCD.

9.14 The main functions to be performed for interfacing a keyboard are:
.. Sense a key actuation.
.. Debounce the key.
.. Decode the key.

Chapter 10

10.1 Bit 7: Set to 0 so that TMR0 is off
Bit 6: Set to 1 in order to enable the 8-bit mode of TMR0
Bit 5: Set to 1 so that an external crystal oscillator can be used
Bit 4: Set to 1 so the timer will increment when the clock is going from high to low (negative edge).
Bit 3: Set to 0 in order to enable the prescaler function
Bit 2-0: Set to 011 to enable a 1:16 prescaler
Hence, T0CON=0x73

10.2 ((b)

```
#include <p18f4321.h>
void main(void)
    {
      OSCCON = 0x60;            // 4 MHz   internal clock
      TRISC=0x00;               // PortC is output
      T0CON=0x08;               // Timer0 is 16-bit no prescaler
      TMR0L=0xA0;               // Value placed in lower 8 bits of TMR0
      TMR0H=0x0F;               // Value placed in upper 8-bits of TMR0
      INTCONbits.TMR0IF=0;      // Clear TMR0  flag
      T0CONbits.TMR0ON=1;       // Turn on TMR0
      while(1)
      {
        while(INTCONbits.TMR0IF==0);
        T0CONbits.TMR0ON=0;                 // Turn off TMR0
        PORTCbits.RC0=~PORTCbits.RC0; // Change output of square wave
        TMR0L=0xA0;                         // Value placed in lower 8 bits of TMR0
        TMR0H=0x0F;                         // Value placed in upper 8-bits of TMR0
        INTCONbits.TMR0IF=0;                // Clear TMR0 flag
        T0CONbits.TMR0ON=1;                 // Turn on TMR0
      }

    }
```

10.2 (c)

```
        INCLUDE  <P18F4321.INC>
        ORG      0x70
        BCF      TRISD, RD7        ; Bit 7 of PORTD is output
        MOVLW    0x60              ; 4MHz internal clock
        MOVWF    OSCCON
        MOVLW    0xC8
        MOVWF    T1CON             ; Setup Timer1 as 16-bit no prescaler
        MOVLW    0xA0
        MOVWF    TMR1L             ; Value placed in TMR1L
```

```
        MOVLW   0x0F
        MOVWF   TMR1H           ; Value place in TMR1H
        BCF     PIR1, TMR1IF    ; Clear Timer1 interrupt flag
        BSF     PIE1, TMR1IE    ; Enable Timer1 interrupt
LOOP    BSF     T1CON, TMR1ON   ; Turn on Timer1
WAIT    BTFSS   PIR1, TMR1IF    ; Wait until TMR1IF occurs
        BRA     WAIT
        BCF     T1CON, TMR1ON   ; Turn Timer1 off
        COMF    PORTD, RD7      ; 1's complement of bit 7 of  PORTD
        MOVLW   0xA0
        MOVWF   TMR1L           ; Value placed in TMR1L
        MOVLW   0x0F
        MOVWF   TMR1H           ; Value placed in TMR1H
        BCF     PIR1, TMR1IF    ; Clear Timer1 interrupt flag
        BSF     T1CON, TMR1ON   ; Turn on Timer1
        BRA     LOOP
        END
```

10.5 (a) A square wave with 50% duty cycle or a symmetrical square wave will be generated on the pin RB4.

Chapter 11

11.1 (a) Capture mode can be used to measure the period of a periodic waveform. CCPx pin is configured as an input for the PIC18F capture mode.

11.2 PWM mode uses Timer2.

11.3 C- program:

```
#include <P18F4321.h>
void main()
{
    TRISC=1;                    // PORTC is input
    CCP1CON=0x05;               // Capture mode, event on rising edge
    T1CON=0xC8;                 // Internal clock, no prescale
    TMR1L=0;                    // Clear TMR1L register
    TMR1H=0;                    // Clear TMR1H register
    PIR1bits.CCP1IF=0;          // Clear CCP1 interrupt flag
    while(PIR1bits.CCP1IF==0); // Wait for first rising edge
    T1CONbits.TMR1ON=1;         // Turn on TMR1
    PIR1bits.CCP1IF=0;          // Clear CCP1 interrupt flag
    while(PIR1bits.CCP1IF==0); // Wait for second rising edge
    T1CONbits.TMR1ON=0;         // Turn off TMR1
    while(1);                   // Period is found in registers CCPR1L and CCPR1H
}
```

11.5

PR2 = [(Fosc)/(4 x Fpwm x TMR2 Prescale Value)] - 1
PR2 = [(4 MHz)/(4 x 16 kHz x 1)] - 1 assuming Prescale value of 1
PR2 = 61.5 = 61 (approximately). With 75% duty cycle = 0.75 x 61,5 = 46.125. Hence, the CCPR1L register will be loaded with 46. Since, 125 (decimal) = .001 (binary), bits DC1B1:DC0B0 of the CCP1CON register will be loaded with 00.

```
;The PIC18F assembly language program is provided below
        INCLUDE <P18F4321.INC>
        ORG     0x100
        MOVLW   0x60            ; 4MHz internal clock
        MOVWF   OSCCON
        MOVLW   D'61'           ; Initialize PR2 register
        MOVWF   PR2
        MOVLW   D'46'           ; Initialize CCPR1L
        MOVWF   CCPR1L
        MOVLW   0x00            ; CCP1 OFF,
        MOVWF   CCP1CON         ; DC1B1:DC0B0=00
        BCF     TRISC, CCP1     ; Configure CCP1 pin as output
        CLRF    T2CON           ; 1:1 prescale, Timer2 OFF
        MOVLW   0x0C            ; PWM mode
        MOVWF   CCP1CON
        CLRF    TMR2            ; Clear Timer2 to 0
BACK  BCF       PIR1, TMR2IF    ; Clear TMR2IF to 0
        BSF     T2CON, TMR2ON   ; Turn Timer2 ON
WAIT BTFSS      PIR1, TMR2IF    ; Wait until TMR2IF is HIGH
        GOTO    WAIT
        BRA     BACK
        END
```

APPENDIX B: GLOSSARY

Absolute Addressing: This addressing mode specifies the address of data with the instruction.

Accumulator: Register used for storing the result after most ALU operations; available with 8-bit microcontrollers.

ADC: See Analog-to-Digital (A/D) Converter.

Address: A unique identification number (or locator) for source or destination of data. An address specifies the register or memory location of an operand involved in the instruction.

Addressing Mode: The manner in which a microcontroller determines the effective address of source and destination operands in an instruction.

Address Register: A register used to store the address (memory location) of data.

Address Space: The number of storage location in a microcontroller's memory that can be directly addressed by the CPU. The addressing range is determined by the number of address lines on the CPU.

American Standard Code for Information Interchange (ASCII): An 8-bit code commonly used with microcontrollers for representing alphanumeric codes. Decimal numbers 0 through 9 are represented by 30 (Hex) through 39 (Hex) in EBCDIC.

Analog-to-Digital (A/D) Converter: Transforms an analog voltage into its digital equivalent. The PIC18F microcontroller contains on-chip A/D converter.

Arithmetic and Logic Unit (ALU): A digital circuit which performs arithmetic and logic operations on two n-bit numbers.

Assembler: A program that translates an assembly language program into a machine language program.

Assembly Language: A type of microcontroller programming language that uses a semi-English-language statement.

Asynchronous Operation: The execution of a sequence of steps such that each step is initiated upon completion of the previous step.

Base address: An address that is used to convert all relative addresses in a program to absolute (machine) addresses.

Baud Rate: Rate of data transmission in bits per second.

Big Endian: This convention is used to store a 16-bit number such as 16-bit data in two bytes of memory locations as follows: the low memory address stores the high byte while the high memory address stores the low byte. The Motorola/Freescale HC11 8-bit microcontroller follows the big Indian format.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

Binary-Coded Decimal (BCD): The representation of 10 decimal digits, 0 through 9, by their corresponding 4-bit binary number.

Bit: An abbreviation for a binary digit. A unit of information equal to one of two possible states (one or zero, on or off, true or false).

Branch: The branch instruction allows the computer to skip or jump out of program sequence to a designated instruction either unconditionally or conditionally (based on conditions such as carry or sign).

Breakpoint: Allows the user to execute the section of a program until one of the breakpoint conditions is met. It is then halted. The designer may then single step or examine memory and registers. Typically breakpoint conditions are program counter address or data references. Breakpoints are used in debugging assembly language programs.

Buffer: A temporary memory storage device deigned to compensate for the different data rates between a transmitting device and a receiving device (for example, between a CPU and a peripheral). Current amplifiers are also referred to as buffers.

Bus: A collection of wires that interconnects computer modules.

Bus Arbitration: Bus operation protocols (rules) that guarantee conflict-free access to a bus. Arbitration is the process of selecting one respondent from a collection of several candidates that concurrently request service.

Bus Cycle: The period of time in which a microcontroller carries out read or write operations.

CCP: CCP stands for Capture, Compare, and PWM (Pulse Width Modulation). The CCP module is implemented in the PIC18F4321 as an on-chip feature to provide measurement and control of time based pulse signals. The capture mode can be used to measure the period of an incoming periodic waveform connected to the PIC18F's CCP pin.. The Compare mode can be used to turn ON a device such as a pump, connected to the PIC18F's CCP pin, after a specified amount of time. As an example, the Compare mode can be used in a chemical plant to turn a pump OFF after filling a tank for 50 seconds. The PWM (Pulse Width Modulation) mode can be used to generate a square wave on the PIC18F's CCP pin with a user-specified frequency and duty cycle.

Central Processing Unit (CPU): The brain of a computer containing the ALU, register section, and control unit.

Chip: An Integrated Circuit (IC) package containing digital circuits.

CISC: Complex Instruction Set Computer. The Control unit is designed using microprogramming. Contains a large instruction set. Difficult to pipeline compared to RISC.

Clock: Timing signals providing synchronization among the various components in a microcontroller. Analogous to heart beats of a human being.

CMOS: Complementary MOS. Dissipates low power, offers high density and speed compared to TTL.

Compiler: A program which translates the source code written in a high-level programming language into machine language that is understandable to the microcontroller.

Computer: The basic blocks of a computer are the central processing unit (CPU), the memory, and the input/output (I/O).

Condition Code Register: Contains information such as carry, sign, zero, and overflow based on ALU operations.

Control Unit: Part of the CPU; its purpose is to translate or decode instructions read (fetched) from the main memory into the Instruction Register.

Data: Basic elements of information represented in binary form (that is, digits consisting of bits) that can be processed or produced by a microcontroller. Data represents any group of operands made up of numbers, letters, or symbols denoting any condition, value, or state. Typical microcontroller operand sizes include: a byte (8 bits), or a word, which typically contains 2 bytes (16-bits).

Debugger: A program that executes and debugs the object program generated by the assembler or compiler. The debugger provides a single stepping, breakpoints, and program tracing.

Decoder: A chip, when enabled, selects one of 2^n output lines based on n inputs.

Digital to Analog (D/A) Converter: Converts binary number to analog signal.

Diode: Two terminal electronic switch.

Directly Addressable Memory: The memory address space in which the microcontroller can directly execute programs. The maximum directly addressable memory is determined by the number of the microcontroller's address pins.

DRAM: See Dynamic RAM.

Duty Cycle: The duty cycle of a periodic waveform is defined as the percentage of the time the pulse is high in a clock period.

Dynamic RAM: Stores data as charges in capacitors and therefore, must be refreshed since capacitors can hold charges for a few milliseconds. Hence, requires refresh circuitry.

EAROM (Electrically Alterable Read-Only Memory): Same as EEPROM or E^2PROM. Can be programmed one line at a time without removing the memory from its sockets. This memory is also called read-mostly memory since it has much slower write times than read times.

Editor: A program that produces an error-free source program, written in assembly or high-level languages.

EEPROM or **E^2PROM:** Same as EAROM (see EAROM).

Effective Address: The final address used to carry out an instruction. Determined by the addressing mode.

EPROM (Erasable Programmable Read-Only Memory): Can be programmed and erased all programs in an EPROM chip using ultraviolet light. The chip must be removed from the circuit board for programming.

Exclusive-OR: The output is 0, if inputs are same; otherwise; the output is 1.

Extended Binary-Coded Decimal Interchange Code (EBCDIC): An 8-bit code sometimes used with computers for representing alphanumeric codes. Normally used by IBM. Decimal numbers 0 through 9 are represented by F0 (Hex) through F9 (Hex) in EBCDIC.

Firmware: Microprogram is sometimes referred to as firmware to distinguish it from hardwired control (purely hardware method).

Flag(s): An indicator, often a single bit, to indicate some conditions such as trace, carry, zero, and overflow.

Flash Memory: Utilizes a combination of EPROM and EEPROM technologies. Used in cellular phones and digital cameras. Also, used to hold program memory on the PIC18F microcontroller.

Flip-Flop: One-bit memory.

Gate: Digital circuits which perform logic operations.

Handshaking: Data transfer via exchange of control signals between the microprocessor and an external device.

Hardware: The physical electronic circuits (chips) that make up the microcontroller.

Hardwired Control: Used for designing the control unit using all hardware.

Harvard CPU architecture: The CPU uses separate instruction and data memory units along with separate buses for instructions and data.

HCMOS: High speed CMOS. Provides high density and consumes low power.

Hexadecimal Number System: Base-16 number system.

High-Level Language: A type of programming language that uses a more understandable human-oriented language such as C.

HMOS: High-density MOS reduces the channel length of the NMOS transistor and provides increased density and speed in VLSI circuits.

I^2C (Inter-Integrated Circuit): I^2C is synchronous and is based on master/slave operation. This protocol uses two pins; one for clock and one for data. Any number of masters and slaves can be connected. The I^2C protocol specifies that in a multimaster/multislave system, the master that initiates a data transfer on the bus is considered as the bus master with all the slave devices connected to the bus as the slaves.

Immediate Address: An address that is used as an operand by the instruction itself.

Implied Address: An address is not specified, but is contained implicitly in the instruction.

Index: A number (typically 8-bit signed or 16-bit unsigned) is used to identify a particular element in an array (string). The index value typically contained in a register is utilized by the indexed addressing mode.

Indexed Addressing: The effective address of the instruction is determined by the sum of the address and the contents of the index register. Used to access arrays.

Index Register: A register used to hold a value used in indexing data, such as when a value is used in indexed addressing to increment a base address contained within an instruction.

Indirect Address: A register holding a memory address to be accessed.

Instruction: Causes the microcontroller to carry out an operation on data. A program contains instructions and data.

Instruction Cycle: The sequence of operations that a microcontroller has to carry out while executing an instruction.

Instruction Register (IR): A register storing instructions.

Instruction Set: Lists all the instructions that the microcontroller can execute.

Internal Interrupt: Activated internally by exceptional conditions such as completion of A/D conversion.

Interpreter: A program that executes a set of machine language instructions in response to each high-level statement in order to carry out the function.

Interrupt I/O: An external device can force the microcontroller to stop executing the current program temporarily so that it can execute another program known as the interrupt service routine.

Interrupts: A temporary break in a sequence of a program, initiated externally or internally, causing control to jump to a routine, which performs some action while the program is stopped.

I/O (Input/Output): Describes that portion of a microcontroller that exchanges data between the microcontroller system and an external device.

I/O Port: A register that contains control logic and data storage used to connect a microcontroller to external peripherals.

Inverting Buffer: Performs NOT operation. Current amplifier.

Keyboard: Has a number of push button-type switches configured in a matrix form (rows x columns).

Keybounce: When a mechanical switch opens or closes, it bounces (vibrates) for a small period of time (about 10-20 ms) before settling down.

Large-Scale Integration (LSI): An LSI chip contains 100 to 1000 gates.

LCD: Liquid Crystal Display. Displays numbers, and several ASCII characters along with graphics. Furthermore, the LCD consumes low power. Because of inexpensive price of the LCD these days, they have been becoming popular. The LCD's are widely used in notebook computers.

LED: Light Emitting Diode. Typically, a current of 10 ma to 20 ma flows at 1.7v to 2.4v drop across it.

Little Endian: This convention is used to store a 16-bit number such as 16-bit data in two bytes of memory locations as follows: the low memory address stores the low byte while the high memory address stores the high byte. The PIC18F microcontroller follows the little-endian format.

Loops: A programming control structure where a sequence of microcontroller instructions are executed repeatedly (looped) until a terminating condition (result) is satisfied.

Machine Code: A binary code (composed of 1's and 0's) that a microcontroller understands.

Machine Language: A type of microntroller programming language that uses binary or hexadecimal numbers.

Macroinstruction: Commonly known as an instruction; initiates execution of a complete microprogram. Example includes assembly language instructions.

Macroprogram: The assembly language program.

Mask: A pattern of bits used to specify (or mask) which bit parts of another bit pattern are to be operated on and which bits are to be ignored or "masked" out. Uses logical AND operation.

Mask ROM: Programmed by a masking operation performed on the chip during the manufacturing process; its contents cannot be changed by user.

Maskable Interrupt: Can be enabled or disabled by executing typically the interrupt instructions.

Memory: Any storage device which can accept, retain, and read back data.

Memory Access Time: Average time taken to read a unit of information from the memory.

Memory Address Register (MAR): Stores the address of the data.

Memory Cycle Time: Average time lapse between two successive read operations.

Memory Map: A representation of the physical locations within a microcontroller's addressable main memory.

Memory-Mapped I/O: I/O ports are mapped as memory locations, with every connected device treated as if it were a memory location with a specific address. Manipulation of I/O data occurs in "interface registers" (as opposed to memory locations); hence there are no input (read) or output (write) instructions used in memory-mapped I/O.

Microcode: A set of instructions called "microinstructions" usually stored in a ROM in the control unit of a microcontroller's CPU to translate instructions of a higher-level programming language such as assembly language programming.

Microcomputer: Consists of a microprocessor, a memory unit, and an input/output unit.

Microcontroller: Typically includes a CPU, memory, I/O, timer, A/D (Analog to Digital) converter in the same chip.

Microinstruction: Some microcontrollers have an internal memory called control memory. This memory is used to store a number of codes called microinstructions. These microinstructions are combined to design the instruction set of the microcontroller.

Microprocessor: CPU on a single chip. The Central Processing Unit (CPU) of a microcomputer.

Microprogramming: Some microcontrollers use microprogramming to design the instruction set. Each instruction in the Instruction register initiates execution of a microprogram stored typically in ROM inside the control unit to perform the required operation.

Multiplexer: A hardware device which selects one of n input lines and produces it on the output.

Nested Subroutine: A commonly used programming technique in which one subroutine calls another subroutine.

Nibble: A 4-bit word.

Non-inverting Buffer: Input is same as output. Current amplifier.

Nonmaskable Interrupt: Occurrence of this type of interrupt cannot be ignored by microcontroller and even though interrupt capability of the microcontroller is disabled. Its effect cannot be disabled by instruction.

Non-Multiplexed: A non-multiplexed microcontroller pin that assigns a unique function as opposed to a multiplexed microcontroller pin defining two functions on time-shared basis.

Object Code: The binary (machine) code into which a source program is translated by a compiler, assembler, or interpreter.

Ones Complement: Obtained by changing 1's to ' 0's, and 0's to 1's of a binary number.

One-Pass Assembler: This assembler goes through the assembly language program once and translates the assembly language program into a machine language program. This assembler has the problem of defining forward references. See Two-Pass Assembler.

Op Code (Operation Code): Part of an instruction defining the operation to be performed.

Operand: A datum or information item involved in an operation from which the result is obtained as a consequence of defined addressing modes. Various operand types contain information, such as source address, destination address, or immediate data.

Operating System: Consists of a number of program modules to provide resource management. Typical resources include CPU, disks, and printers.

Page: Some microcontrollers, divide the memory locations into equal blocks. Each of these blocks is called a page and contains several addresses.

Parallel Operation: Any operation carried out simultaneously with a related operation.

Parallel Transmission: Each bit of binary data is transmitted over a separate wire.

Parity: The number of 1's in a word is odd for odd parity and even for even parity.

Peripheral: An I/O device capable of being operated under the control of a CPU through communication channels. Examples include disk drives, keyboards, CRT's, printers, and modems.

Personal Computer: Low-cost, affordable microcomputer normally used by an individual for word processing and Internet applications.

Physical Address Space: Address space is defined by the address pins of the microcontroller.

Pipeline: A technique that allows a microcontroller processing operation to be broken down into several steps (dictated by the number of pipeline levels or stages) so that the individual step outputs can be handled by the microcontroller in parallel. Often used

to fetch the processor's next instruction while executing the current instruction, which considerably speeds up the overall operation of the microcontroller. Overlaps instruction fetch with execution.

Pointer: A storage location (usually a register within a microcontroller) that contains the address of (or points to) a required item of data or subroutine.

Polled Interrupt: A software approach for determining the source of interrupt in a multiple interrupt system.

POP Operation: Reading from the top or bottom of stack.

Port: A register through which the microcontrollers communicate with peripheral devices.

Primary or Main Memory: Storage that is considered internal to the microcontroller. The microcontroller can directly execute all instructions in the main memory. The maximum size of the main memory is defined by the number of address pins in the CPU.

Processor Memory: A set of CPU registers for holding temporary results when a computation is in progress.

Program: A self-contained sequence of computer software instructions (source code) that, when converted into machine code, directs the computer to perform specific operations for the purpose of accomplishing some processing task. Contains instructions and data.

Program Counter (PC): A register that normally contains the address of the next instruction to be executed in a program.

Programmed I/O: The microcontroller executes a program to perform all data transfers between the microcontroller system and external devices.

PROM (Programmable Read-Only Memory): Can be programmed by the user by using proper equipment. Once programmed, its contents cannot be altered.

Protocol: A list of data transmission rules or procedures that encompass the timing, control, formatting, and data representations by which two devices are to communicate. Also known as hardware "handshaking", which is used to permit asynchronous communication.
PUSH Operation: Writing to the top or bottom of stack.

Random Access Memory (RAM): A read/write memory. RAMs (static or dynamic) are volatile in nature (in other words, information is lost when power is removed).

Read-Only-Memory (ROM): A memory in which any addressable operand can be read from, but not written to, after initial programming. ROM storage is nonvolatile (information is not lost after removal of power).

Reduced Instruction Set Computer (RISC): A simple instruction set is included. The RISC architecture maximizes speed by reducing clock cycles per instruction. The control unit is designed using hardwired control. Easier to implement pipelining.

Register: A high-speed memory usually constructed from flip-flops that are directly accessible to the CPU. It can contain either data or a specific location in memory that stores word(s) used during arithmetic, logic, and transfer operations.

Register Indirect: Uses a register which contains the address of data.

Relative Address: An address used to designate the position of a memory location in a routine or program.

RISC: See Reduced Instruction Set Computer.

Routine: A group of instructions for carrying out a specific processing operation. Usually refers to part of a larger program. A routine and subroutine have essentially the same meaning, but a subroutine could be interpreted as a self-contained routine nested within a routine or program.

SDRAM: Synchronous DRAM. This chip contains several DRAMs internally. The control signals and address inputs are sampled by the SDRAM by a common clock.

Secondary Memory Storage: An auxiliary data storage device that supplements the main (primary) memory of a computer. It is used to hold programs and data that would otherwise exceed the capacity of the main memory. Although it has a much slower access time, secondary storage is less expensive. Examples include floppy and hard disks.

Sequential Circuit: Combinational circuit with memory.

Serial Transmission: Only one line is used to transmit the complete binary data bit by bit.

Seven-Segment LED: Contains an LED in each of the seven segments.Can display numbers.

Signed Number A signed binary number, on the other hand, includes both positive and negative numbers. It is represented in the microcontroller in two's-complement form. For example, the decimal number +15 is represented in 8-bit two's-complement form as 00001111 (binary) or 0F (hexadecimal). The decimal number -15 can be represented in 8-bit two's-complement form as 11110001 (binary) or F1 (hexadecimal). Also, the most significant bit (MSB) of a signed number represents the sign of the number. For example, bit 7 of an 8-bit number, bit 15 of a 16-bit number, and bit 31 of a 32-bit number represent the signs of the respective numbers. A "0" at the MSB represents a positive number; a "1" at the MSB represents a negative number.

Single-Chip Microcomputer: Microcomputer (CPU, memory, and input/output) on a chip.

Single-chip Microprocessor: Microcomputer CPU (microprocessor) on a chip.

Single Step: Allows the user to execute a program one instruction at a time and examine contents of memory locations and registers.

Software: consists of a collection of programs that contain instructions and data for performing a specific task in a microcontroller.

Source Code: The assembly language program written by a programmer using assembly language instructions. This code must be translated to the object (machine) code by the assembler before it can be executed by the microcontroller.

SPI (Serial Peripheral Interface): The main purpose of the SPI is to replace parallel interfaces by avoiding routing of parallel buses in a PCB (Printed Circuit Board). The SPI protocol is based on the principle that a bit from an 8-bit shift register can be shifted out on

a single pin and, a bit can be shifted into another pin. SPI protocol can also be used for data transfer between the CPU (master) and slave devices such as flash memory and ADC.

SRAM: See Static RAM.

Stack: An area of read/write memory typically used by a microcontroller during subroutine calls or occurrence of an interrupt.The microcontroller saves in the stack the contents of the program counter before executing the subroutine or program counter contents and other status information before executing the interrupt service routine. Thus, the microcontroller can return to the main program after execution of the subroutine or the interrupt service routine. The stack is a last in/first out (LIFO) read/write memory (RAM) that can also be manipulated by the programmer using PUSH and POP instructions.

Stack Pointer: A register used to address the stack.

Standard I/O: Utilizes a control pin on the CPU typically called the $M/\overline{IO}$ pin, in order to distinguish between input/output and memory; IN and OUT instructions are used for input/output operations.

Static RAM: Also known as **SRAM**. Stores data in flip-flops; does not need to be refreshed. Information is lost upon power failure unless backed up by battery.

Status Register: A register which contains information concerning the flags in a microcontroller.

Subroutine: A program carrying out a particular function and which can be called by another program known as the main program. A subroutine needs to be placed only once in memory and can be called by the main program as many times as the programmer wants.

Synchronous Operation: Operations that occur at intervals directly related to a clock period.

Tracing: Allows single stepping. A dynamic diagnostic technique permits analysis (debugging) of the program's execution.

Tristate Buffer: Has three output states: logic 0, 1, and a high-impedance state. This chip is typically enabled by a control signal to provide logic 0 or 1 outputs. This type of buffer can also be disabled by the control signal to place it in a high-impedance state.

Two's Complement: The two's complement of a binary number is obtained by replacing each 0 with a 1 and each 1 with a 0 and adding one to the resulting number.

Two-Pass Assembler: This assembler goes through the assembly language program twice. In the first pass, the assembler assigns binary addresses to labels. In the second pass, the assembly program is translated to the machine language. No problem with forward branching.

Unsigned Number: An *Unsigned binary number* has no arithmetic sign, therefore, are always positive. Typical examples are your age or a memory address, which are always positive numbers. An 8-bit unsigned binary integer represents all numbers from 00 through FF (0 through 255 in decimal).

Very Large Scale Integration (VLSI): a VLSI chip contains more than 1000 gates. More commonly, a VLSI chip is identified by the number of transistors rather than the gate count.

von Neumann (Princeton) CPU architecture: uses a single memory unit and the same bus for accessing both instructions and data.

Word: The bit size of a microcontroller refers to the number of bits that can be processed simultaneously by the basic arithmetic and logic circuits of the CPU. A number of bits taken as a group in this manner is called a word.

APPENDIX C: PIC18F INSTRUCTION SET (Alphabetical Order)

Instruction	*Example*	*Operation*
ADDLW data8	ADDLW 0x07	[WREG] + 0x07 t [WREG]
ADDWF F, d, a	ADDWF 0x20, W	[WREG] + [0x20] → [WREG]
	ADDWF 0x20, F	[WREG] + [0x20] → [0x20]
ADDWFC F, d, a	ADDWFC 0x40, W	[WREG] + [0x40] + Carry → [WREG]
	ADDWFC 0x40, F	[WREG] + [0x40] + Carry → [0x40]
ANDLW data8	ANDLW 0x02	[WREG] AND 0x02 → [WREG]
ANDWF F, d, a	ANDWF 0x30, W	[WREG] AND [0x30] → [WREG]
	ANDWF 0x30, F	[WREG] AND [0x30] → [0x30]
BC d8	BC START	Branch to START if C = 1 where START is an 8-bit signed #
BCF F, b, a	BCF 0x30, 2	Clear bit number 2 to 0 in data register 0x30, store result in 0x30
	BCF STATUS, C	Clear the Carry Flag to 0 in the status register.
BN d8	BN START	Branch to START if N = 1 where START is an 8-bit signed #
BNC d8	BNC START	Branch to START if C = 0 where START is an 8-bit signed #
BNN d8	BNN START	Branch to START if N = 0 where START is an 8-bit signed #
BNOV d8	BBNOV START	Branch to START if OV = 0 where START is an 8-bit signed #
BNZ d8	BNZ START	Branch to START if Z = 0 where START is an 8-bit signed #
BOV d8	BOV START	Branch to START if OV = 1 where START is an 8-bit signed #
BRA d8	BRA START	Branch always to START where START is an 8-bit signed #
BSF F, b, a	BSF 0x20, 7	Set bit number 7 to 1 in data register 0x20, store result in 0x20.
	BSF STATUS, C	Set the Carry Flag to 1 in the status register.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

Instruction	*Example*	*Operation*
BTFSC F, b, a	BTFSC 0x50, 3	If bit number 3 in data register 0x50 is 0, skip the next instruction; otherwise, the next instruction is executed.
BTFSS F, b, a	BTFSC 0x40, 0	If bit number 0 in data register 0x40 is 1, skip the next instruction; otherwise, the next instruction is executed.
BTG F, b, a	BTG 0x20, 2	Invert (ones complement) bit number 2 in data register 0x20.
BZ d8	BZ START	Branch to START if Z = 1 where START is an 8-bit signed #
CALL k, s	CALL BEGIN	This is a two-word instruction. The simplest way to CALL a subroutine is when s = 0 (default); pushes current program counter (PC+4) which is also the return address , and loads PC with BEGIN which is the starting address of the subroutine.
CLRF F, a	CLRF 0x40	Clear the contents of data register to 0.
CLRWDT	CLRWDT	Reset the watchdog timer.
COMF F, d, a	COMF 0x30, F	One's complement each bit of [0x30], and store the result in 0x30.
	COMF 0x30, W	One's complement each bit of [0x30], and store the result in WREG.
CPFSEQ F, a	CPFSEQ 0x30	Unsigned comparison. If [0x30] =[WREG], skip the next instruction; else, the next instruction is executed.
CPFSGT F, a	CPFSGT 0x50	Unsigned comparison. If [0x50] >[WREG], skip the next instruction; else, the next instruction is executed.
CPFSLT F, a	CPFSLT 0x60	Unsigned comparison. If [0x60] <[WREG], skip the next instruction; else, the next instruction is executed.
DAW	DAW	Decimal Adjust [WREG] resulting from earlier addition of two packed BCD digits providing correct packed BCD result.
DECF F, d, a	DECF 0x20, W	Decrement [0x20] by 1, and store result in WREG.
	DECF 0x20, F	Decrement [0x20] by 1, and store result in 0x20.

Instruction	*Example*	*Operation*
DECFSZ F, d, a	DECFSZ 0x30, W	Decrement [0x30] by 1, and store result in WREG. If [WREG] = 0, skip the next instruction; else, execute the next instruction.
	DECFSZ 0x30, F	Decrement [0x30] by 1, and store the result in 0x30. If [0x30] = 0, skip the next instruction; else, execute the next instruction.
DECFSNZ F, d, a	DECFSNZ 0x50,W	Decrement [0x50] by 1, and store result in WREG. If [WREG] ≠ 0, skip the next instruction; else, execute the next instruction.
	DECFSNZ 0x50, F	Decrement [0x50] by 1, and store the result in 0x50. If [0x50] ≠ 0, skip the next instruction; else, execute the next instruction.
GOTO k	GOTO START	Unconditional branch to address START.
INCF F, d, a	INCF 0x20, W	Increment [0x20] by 1, and store result in WREG.
	INCF 0x20, F	Increment [0x20] by 1, and store result in 0x20.
INCFSZ F, d, a	INCFSZ 0x30, W	Increment [0x30] by 1, and store result in WREG. If [WREG] = 0, skip the next instruction; else, execute the next instruction.
	INCFSZ 0x30, F	Increment [0x30] by 1, and store the result in 0x30. If [0x30] = 0, skip the next instruction; else, execute the next instruction.
INCFSNZ F, d, a	INCFSNZ 0x50, W	Increment [0x50] by 1, and store result in WREG. If [WREG] ≠ 0, skip the next instruction; else, execute the next instruction.
	INCFSNZ 0x50, F	Increment [0x50] by 1, and store the result in 0x50. If [0x50] ≠ 0, skip the next instruction; else, execute the next instruction.
IORLW k	IORLW 0x54	The contents of WREG are logically ORed with 0x54, and the result is stored in WREG.

Instruction	*Example*	*Operation*
IORWF F, d, a	IORWF 0x50, W	The contents of WREG are logically ORed with the contents of 0x50, and the result is stored in WREG.
	IORWF 0x50, F	The contents of WREG are logically ORed with the contents of 0x50, and the result is stored in 0x50.
LFSR F, k	LFSR 0, 0x0080	Load 00H into FSR0H, and 80H into FSR0L.
MOVF F, d, a	MOVF 0x30, W	The contents of 0x30 are loaded into WREG.
	MOVF 0x30, F	The contents of 0x30 are copied into 0x30.
MOVFF Fs, Fd	MOVFF 0x50,0x60	Move [0x50] into [0x60]. The contents of 0x50 are unchanged.
MOVLB k	MOVLF 0x04	Load BSR with 04H.
MOVLW k	MOVLW 0x21	Load WREG with 21H.
MOVWF F, a	MOVWF 0x50	Move the contents of WREG into 0x50.
MULLW k	MULLW 0xF1	[WREG] x F1H → [PRODH] [PRODL]; unsigned multiplication.
MULWF F, a	MULWF 0x30	[WREG] x [0x30] → [PRODH] [PRODL] ; unsigned multiplication.
NEGF F, a	NEGF 0x20	Negate the contents of 0x20 using two's complement.
NOP	NOP	No Operation.
POP	POP	Discard top of stack pointed by SP, and decrement PC by 1.
PUSH	PUSH	Push or write PC onto the stack, and increment SP by 1.
RCALL n	RCALL START	Relative subroutine CALL. One-word instruction. Pushes PC+2 onto the hardware stack. START is an 11-bit signed number. Jumps to a subroutine located at an address (PC +2) + 2 x START.
RESET	RESET	Reset all registers and flags that are affected by a $\overline{\text{MCLR}}$ reset.
RETFIE	RETFIE	Return from Interrupt.
RETLW k	RETLW k	WREG is loaded with the 8-bit literal k, and PC is loaded with the return address from the top of stack.
RETURN	RETURN	Return from subroutine.

Instruction	Example	Operation
RLCF F, d, a	RLCF 0x20, W	Rotate [0x20] once to the left through Carry. Store result in WREG.
	RLCF 0x20, F	Rotate [0x20] once to the left through Carry. Store result in register 0x20.
RLNCF F, d, a	RLNCF 0x30, W	Rotate [0x30] once to the left without Carry. Store result in WREG.
	RLNCF 0x30, F	Rotate [0x30] once to the left without Carry. Store result in register 0x30.
RRCF F, d, a	RRCF 0x50, W	Rotate [0x50] once to the right through Carry. Store result in WREG.
	RRCF 0x50, F	Rotate [0x50] once to the right through Carry. Store result in register 0x50.
RRNCF F, d, a	RRNCF 0x60, W	Rotate [0x60] once to the right without Carry. Store result in WREG.
	RRNCF 0x60, F	Rotate [0x60] once to the right without Carry. Store result in register 0x60.
SETF F, a	SETF 0x30	The contents of 0x30 are set to 1's.
SLEEP	SLEEP	Enter Sleep mode.
SUBFWB F, d, a	SUBFWB 0x20, W	[WREG] –[0x20] – Carry → [WREG]
	SUBFWB 0x20, F	[WREG] –[0x20] – Carry → [0x20]
SUBLW k	SUBLW 0x05	[0x05] – [WREG] → [WREG]
SUBWF F, d, a	SUBWF 0x50, W	[0X50] – [WREG] → [WREG]
	SUBWF 0x50, F	[0X50] – [WREG] → [0x50]
SUBWFB F, d, a	SUBWFB 0x32, W	[0x32] – [WREG] – Carry → [WREG]
	SUBWFB 0x32, F	[0x32] – [WREG] – Carry → [0x32]
SWAPF F, d, a	SWAPF 0x30, W	The upper and lower 4 bits of register 0x30 are exchanged. The result is stored in WREG.
	SWAPF 0x30, F	The upper and lower 4 bits of register 0x30 are exchanged. The result is stored in register 0x30.
TBLRD	TBLRD	Table Read.
TBLWT	TBLWT	Table Write.
TSTFSZ F, a	TSTFSZ 0x50	If [0x50] = 0, skip the next instruction; else, execute the next instruction.
XORLW k	XORLW 0xF2	[WREG] XOR F2H → [WREG]
XORWF F, d, a	XORWF 0x30, W	[WREG] XOR [0x30] → [WREG]
	XORWF 0x30, F	[WREG] XOR [0x30] → [0x30]

APPENDIX D: PIC18F INSTRUCTION SET — DETAILS

24.0 INSTRUCTIONSETSUMMARY

PIC18F4321 family devices incorporate the standard set of 75 PIC18 core instructions, as well as an extended set of 8 new instructions, for the optimization of code that is recursive or that utilizes a software stack. The extended set is discussed later in this section.

24.1 Standard Instruction Set

The standard PIC 18 instruction set adds many enhancements to the previous PIC® instruction sets, while maintaining an easy migration from these PIC instruction sets. Most instructions are a single program memory word (16 bits), but there are four instructions that require two program memory locations.

Each single-word in struction is a 16-bit word divided into an opcode, which specifies the instruction type and one or more operands, which further specify the operation of the instruction.

The instruction set is highly orthogonal and is grouped into four basic categories:

- **Byte-oriented** operations
- **Bit-oriented** operations
- **Literal** operations
- **Control** operations

The PIC18 instruction set summary in Table 24-2 lists **byte-oriented**, **bit-oriented**, **literal,** and **control** operations. Table 24-1 shows the opcode fiel d descriptions.

Most **byte-oriented** instructions have three operands:

1. The file register (specified by 'f')
2. The destination of the result (specified by 'd')
3. The accessed memory (specified by 'a')

The file register designator 'f' specifies which file register is to be used by the instruction. 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.

All **bit-oriented** instructions have three operands:

1. The file register (specified by 'f')
2. The bit in the file register (specified by 'b')
3. The accessed memory (specified by 'a')

The bit field designator 'b' selects the number of the bit affected by the operation, while the file register designator 'f' represents the number of the file in which the bit is located.

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 FSR register to load the literal value into (specified by 'f')
- No operand required (specified by '—')

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')
- The mode of the table read and table write instructions (specified by 'm')
- No operand required (specified by '—')

All instructions are a single word, except for fo ur double-word i nstructions. The se in structions w ere made double-word to contain the required information in 32 bits. In the second word, the 4 MSbs are '1's. If this second word is ex ecuted as an instruction (by itself), it will execute as a `NOP`.

All sin gle-word instructions are executed in a single instruction cycle, unless a conditional test is true or the program counter is changed as a result of the instruction. In these cases, the execution takes two instruction cycles, with the additional instruction cycle(s) executed as a `NOP`.

The double-word instructions execute in two instruction cycles.

One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1s. If a conditional test is true, or the program counter is changed as a result of an i nstruction, t he i nstruction execution time is 2s. Two-word branch instructions (if true) would take 3s.

Figure 24-1 shows the general formats that the instructions can have. All examples use the convention 'nnh' to represent a hexadecimal number.

The Instruction Set Summary, shown in Table 24-2, lists the standard in structions recognized by the Microchip MPASM™ Assembler.

Section 24.1.1 "Standard Instruction Set" provides a description of each instruction.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

TABLE 24-1: OPCODE FIELD DESCRIPTIONS

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.
C, DC, Z, OV, N	ALU Status bits: **C**arry, **D**igit **C**arry, **Z**ero, **Ov**erflow, **N**egative.
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 (00h to FFh) or 2-bit FSR designator (0h to 3h).
fs	12-bit Register file address (000h to FFFh). This is the source address.
fd	12-bit Register file address (000h to FFFh). This is the destination address.
GIE	Global Interrupt Enable bit.
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.
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.
PD	Power-down bit.
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)
TBLPTR	21-bit Table Pointer (points to a Program Memory location).
TABLAT	8-bit Table Latch.
TO	Time-out bit.
TOS	Top-of-Stack.
u	Unused or unchanged.
WDT	Watchdog Timer.
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.
zs	7-bit offset value for indirect addressing of register files (source).
zd	7-bit offset value for indirect addressing of register files (destination).
{ }	Optional argument.
[text]	Indicates an indexed address.
(text)	The contents of text.
[expr]<n>	Specifies bit n of the register indicated by the pointer expr.
→	Assigned to.
< >	Register bit field.
∈	In the set of.
italics	User defined term (font is Courier).

FIGURE 24-1: GENERAL FORMAT FOR INSTRUCTIONS

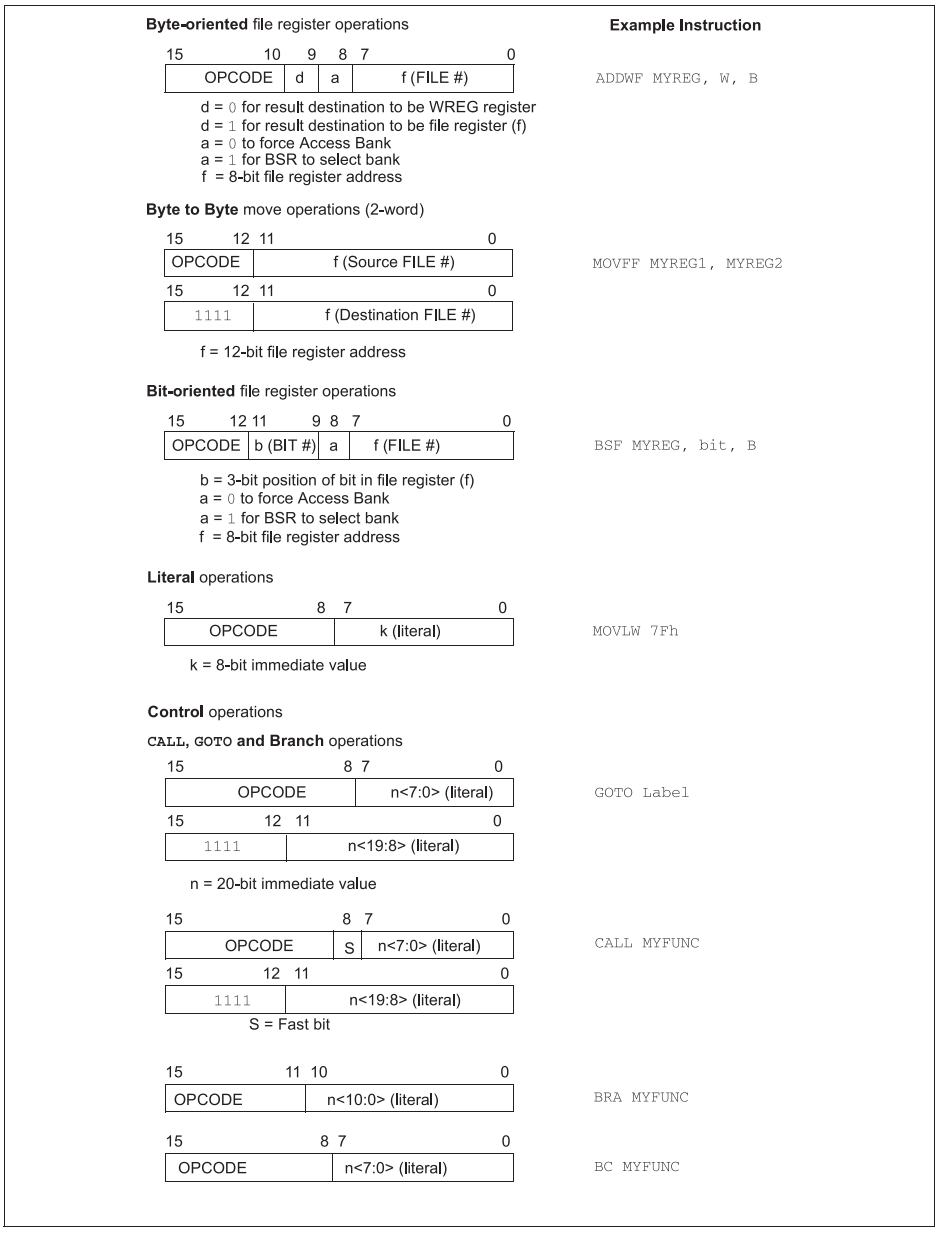


TABLE 24-2: PIC18FXXXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
BYTE-ORIENTED OPERATIONS									
ADDWF	f, d, a		1	0010	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	Add WREG and CARRY bit to f	1	0001	01da	ffff	ffff	Z, N	1,2
CLRF	f, a	AND WREG with f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Clear f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Complement f	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	Compare f with WREG, skip <	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, Skip if 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Decrement f, Skip if Not 0	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, Skip if 0	1 (2 or 3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Increment f, Skip if Not 0	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Inclusive OR WREG with	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	fs, fd	f Move f	2	1100	ffff	ffff	ffff	None	
		Move fs (source) to 1st word fd (destination) 2nd word		1111	ffff	ffff	ffff		
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	1, 2
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	
RLCF	f, d, a	Rotate Left f through Carry	1	0011	01da	ffff	ffff	C, Z, N	1, 2
RLNCF	f, d, a	Rotate Left f (No Carry)	1	0100	01da	ffff	ffff	Z, N	
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	1, 2
SUBFWB	f, d, a	Subtract f from WREG with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	
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	

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 two-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.

TABLE 24-2: PIC18FXXXX INSTRUCTION SET (CONTINUED)

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
BIT-ORIENTED 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, b, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2
CONTROL OPERATIONS									
	n		1 (2)	1110 0010	nnnn	nnnn			
BC	n	Branch if Carry	1 (2)	1110 0110	nnnn	nnnn		None	
BN	n	Branch if Negative	1 (2)	1110 0011	nnnn	nnnn		None	
BNC	n	Branch if Not Carry	1 (2)	1110 0111	nnnn	nnnn		None	
BNN	n	Branch if Not Negative	1 (2)	1110 0101	nnnn	nnnn		None	
BNOV	n	Branch if Not Overflow	1 (2)	1110 0001	nnnn	nnnn		None	
BNZ	n	Branch if Not Zero	1 (2)	1101 0100	nnnn	nnnn		None	
BOV	n	Branch if Overflow	2	1110 0nnn	nnnn	nnnn		None	
BRA	n	Branch Unconditionally	1 (2)	1110 0000	nnnn	nnnn		None	
BZ	n, s	Branch if Zero	2	1111 110s	kkkk	kkkk		None	
CALL		Call subroutine 1st word		1111 kkkk	kkkk	kkkk		None	
		2nd word		0000 kkkk	0000	0100		TO, PD	
CLRWDT	—	Clear Watchdog Timer	1	0000 0000	0000	0111		C	
DAW	—	Decimal Adjust WREG	1	1110 kkkk	kkkk	kkkk		None	
GOTO	n	Go to address 1st word	2	1111 kkkk	kkkk	kkkk			
		2nd word		0000 0000	0000	0000		None	
NOP	—	No Operation	1	1111 xxxx	xxxx	xxxx		None	
NOP	—	No Operation	1	0000 0000	0000	0110		None	
POP	—	Pop top of return stack (TOS)	1	0000 0000	0000	0101		None	
PUSH	—	Push top of return stack (TOS)	2	0000 1nnn	nnnn	nnnn		All	
RCALL	—	Relative Call	1	1101 0000	1111	1111		GIE/GIEH, PEIE/GIEL	
RESET		Software device Reset	2	0000 0000	0001	000s		None	4
RETFIE	n	Return from interrupt enable	1	0000 0000	kkkk	kkkk		None	
RETLW	s	Return with literal in WREG	2	0000 1100	0001	001s		TO, PD	
RETURN	k	Return from Subroutine	2	0000 0000	0000	0011			
SLEEP	s	Go into Standby mode	1	0000 0000					
	—			0000					

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 two-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.

TABLE 24-2: PIC18FXXXX INSTRUCTION SET (CONTINUED)

Mnemonic, Operands		Description	Cycles	16-Bit Instruction Word MSb			LSb	Status Affected	Notes
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 FSR(f)	2	1110	1110	00ff	kkkk	None	
		1st word Move literal to		1111	0000	kkkk	kkkk		
MOVLB	k	BSR<3:0> Move literal to	1	0000	0001	0000	kkkk	None	
MOVLW	k	WREG Multiply literal with	1	0000	1110	kkkk	kkkk	None	
MULLW	k	WREG Return with literal in	1	0000	1101	kkkk	kkkk	None	
RETLW	k	WREG Subtract WREG from	2	0000	1100	kkkk	kkkk	None	
SUBLW	k	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	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 two-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.

24.1.1 STANDARD INSTRUCTION SET

ADDLW **ADD Literal to W**

Syntax: ADDLW k

Operands: $0 \le k \le 255$

Operation: $(W) + k \rightarrow W$

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 15h`

Before Instruction
W = 10h
After Instruction
W = 25h

ADDWF **ADD W to f**

Syntax: 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `ADDWF REG, 0, 0`

Before Instruction
W = 17h
REG = 0C2h
After Instruction
W = 0D9h
REG = 0C2h

Note: All PIC18 instructions may take an optional label argument preceding the instruction mnemonic for use in symbolic addressing. If a label is used, the instruction format then becomes: {label} instruction argument(s).

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

Syntax: 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `ADDWFC REG, 0, 1`

Before Instruction
Carry bit = 1
REG = 02h
W = 4Dh
After Instruction
Carry bit = 0
REG = 02h
W = 50h

ANDLW **AND Literal with W**

Syntax: ANDLW k

Operands: $0 \le k \le 255$

Operation: (W) .AND. k → W

Status Affected: N, Z

Encoding:

0000	1011	kkkk	kkkk

Description: The contents of W are ANDed 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 05Fh`

Before Instruction
W = A3h
After Instruction
W = 03h

ANDWF **AND W with f**

Syntax: ANDWF f {,d {,a}}

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

Operation: (W) .AND. (f) → dest

Status Affected: N, Z

Encoding:

0001	01da	ffff	ffff

Description: The contents of W are ANDed 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `ANDWF REG, 0, 0`

Before Instruction
W = 17h
REG = C2h
After Instruction
W = 02h
REG = C2h

BC **Branch if Carry**

Syntax: 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 5`

Before Instruction
PC = address (HERE)
After Instruction
If Carry = 1;
PC = address (HERE + 12)
If Carry = 0;
PC = address (HERE + 2)

BCF **Bit Clear f**

Syntax: 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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 0`

Before Instruction
FLAG_REG = C7h
After Instruction
FLAG_REG = 47h

BN **Branch if Negative**

Syntax: 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)`

BNC **Branch if Not Carry**

Syntax: 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)`

BNN **Branch if Not Negative**

Syntax: 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)`

BNOV **Branch if Not Overflow**

Syntax: 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)

BNZ **Branch if Not Zero**

Syntax: 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)

BRA **Unconditional Branch**

Syntax: 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)

BSF **Bit Set f**

Syntax: 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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 1`

Before Instruction
FLAG_REG = 0Ah
After Instruction
FLAG_REG = 8Ah

BTFSC **Bit Test File, Skip if Clear**

Syntax: BTFSC f, b {,a}

Operands: $0 \le f \le 255$
$0 \le b \le 7$
$a \in [0,1]$

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh).
See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 0
FALSE   :
TRUE    :
```
Before Instruction
PC = address (HERE)
After Instruction
If FLAG<1> = 0;
PC = address (TRUE)
If FLAG<1> = 1;
PC = address (FALSE)

BTFSS **Bit Test File, Skip if Set**

Syntax: BTFSS f, b {,a}

Operands: $0 \le f \le 255$
$0 \le b < 7$
$a \in [0,1]$

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh).
See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 0
FALSE   :
TRUE    :
```
Before Instruction
PC = address (HERE)
After Instruction
If FLAG<1> = 0;
PC = address (FALSE)
If FLAG<1> = 1;
PC = address (TRUE)

BTG **Bit Toggle f**

Syntax: BTG f, b {,a}

Operands: $0 \le f \le 255$
$0 \le b < 7$
$a \in [0,1]$

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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `BTG PORTC, 4, 0`

Before Instruction:
PORTC = 0111 0101 [75h]
After Instruction:
PORTC = 0110 0101 [65h]

BOV **Branch if Overflow**

Syntax: BOV n

Operands: $-128 \le n \le 127$

Operation: if Overflow bit is '1'
$(PC) + 2 + 2n \rightarrow PC$

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)

BZ **Branch if Zero**

Syntax: 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)

CALL **Subroutine Call**

Syntax: 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, 1`

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

CLRF **Clear f**

Syntax: CLRF f {,a}

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

Operation: $000h \rightarrow f$
$1 \rightarrow Z$

Status Affected: Z

Encoding:

0110	101a	ffff	ffff

Description: Clears the contents of the specified register.
If 'a' is '0', the Access Bank is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: CLRF FLAG_REG, 1

Before Instruction
FLAG_REG = 5Ah
After Instruction
FLAG_REG = 00h

CLRWDT **Clear Watchdog Timer**

Syntax: CLRWDT

Operands: None

Operation: $000h \rightarrow$ WDT,
$000h \rightarrow$ WDT postscaler,
$1 \rightarrow \overline{TO}$,
$1 \rightarrow \overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

0000	0000	0000	0100

Description: CLRWDT instruction resets the Watchdog Timer. It also resets the postscaler of the WDT. Status bits, $\overline{TO}$ and $\overline{PD}$, are set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	No operation

Example: CLRWDT

Before Instruction
WDT Counter = ?
After Instruction
WDT Counter = 00h
WDT Postscaler = 0
$\overline{TO}$ = 1
$\overline{PD}$ = 1

COMF **Complement f**

Syntax: COMF f {,d {,a}}

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

Operation: $(\bar{f}) \rightarrow$ 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example:
```
COMF    REG, 0, 0
```
Before Instruction
REG = 13h
After Instruction
REG = 13h
W = ECh

CPFSEQ **Compare f with W, Skip if f = W**

Syntax: CPFSEQ 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	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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 0
NEQUAL  :
EQUAL   :
```
Before Instruction
PC Address = HERE
W = ?
REG = ?
After Instruction
If REG = W;
PC = Address (EQUAL)
If REG ≠ W;
PC = Address (NEQUAL)

CPFSGT **Compare f with W, Skip if f > W**

Syntax: 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 the 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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 0
NGREATER  :
GREATER   :
```
Before Instruction
PC = Address (HERE)
W = ?
After Instruction
If REG > W;
PC = Address (GREATER)
If REG ≤ W;
PC = Address (NGREATER)

CPFSLT **Compare f with W, Skip if f < W**

Syntax: CPFSLT 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	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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (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, 1
NLESS   :
LESS    :
```
Before Instruction
PC = Address (HERE)
W = ?
After Instruction
If REG < W;
PC = Address (LESS)
If REG ≥ W;
PC = Address (NLESS)

DAW **Decimal Adjust W Register**

Syntax: 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> + DC > 9] or [C = 1] then
(W<7:4>) + 6 + DC → W<7:4>;
else
(W<7:4>) + DC → W<7:4>

Status Affected: C

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.

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 = A5h
C = 0
DC = 0
After Instruction
W = 05h
C = 1
DC = 0

Example 2:

Before Instruction
W = CEh
C = 0
DC = 0
After Instruction
W = 34h
C = 1
DC = 0

DECF **Decrement f**

Syntax: 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `DECF CNT, 1, 0`

Before Instruction
CNT = 01h
Z = 0
After Instruction
CNT = 00h
Z = 1

DECFSZ **Decrement f, Skip if 0**

Syntax: DECFSZ f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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    DECFSZ  CNT, 1, 1
        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)

DCFSNZ **Decrement f, Skip if Not 0**

Syntax: DCFSNZ f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 1, 0
ZERO    :
NZERO   :
```

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

GOTO **Unconditional Branch**

Syntax: GOTO k

Operands: $0 \le k \le 1048575$

Operation: $k \rightarrow PC<20:1>$

Status Affected: None

Encoding:

1st word (k<7:0>)	1110	1111	k_7kkk	$kkkk_0$
2nd word(k<19:8>)	1111	$k_{19}kkk$	kkkk	$kkkk_8$

Description: GOTO allows an unconditional branch anywhere within 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)

INCF **Increment f**

Syntax: INCF f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default). If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: INCF CNT, 1, 0

Before Instruction
CNT = FFh
Z = 0
C = ?
DC = ?
After Instruction
CNT = 00h
Z = 1
C = 1
DC = 1

INCFSZ **Increment f, Skip if 0**

Syntax: INCFSZ f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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    INCFSZ  CNT, 1, 0
NZERO   :
ZERO    :
```

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

INFSNZ **Increment f, Skip if Not 0**

Syntax: INFSNZ f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 1, 0
ZERO
NZERO
```

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

IORLW **Inclusive OR Literal with W**

Syntax: 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 ORed 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 35h`

Before Instruction
W = 9Ah
After Instruction
W = BFh

IORWF **Inclusive OR W with f**

Syntax: IORWF f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `IORWF RESULT, 0, 1`

Before Instruction
RESULT = 13h
W = 91h
After Instruction
RESULT = 13h
W = 93h

LFSR **Load FSR**

Syntax: LFSR f, k

Operands: $0 \le f \le 2$
$0 \le k \le 4095$

Operation: $k \rightarrow FSRf$

Status Affected: None

Encoding:

1110 1111	1110 0000	00ff $k_7$kkk	k_{11}kkk kkkk

Description: The 12-bit literal 'k' is loaded into the File Select Register pointed to by 'f'.

Words: 2

Cycles: 2

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k' MSB	Process Data	Write literal 'k' MSB to FSRfH
Decode	Read literal 'k' LSB	Process Data	Write literal 'k' to FSRfL

Example: `LFSR 2, 3ABh`

After Instruction
FSR2H = 03h
FSR2L = ABh

MOVF **Move f**

Syntax: MOVF f {,d {,a}}

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

Operation: $f \rightarrow dest$

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 '0', the result is placed in W. If 'd' is '1', 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `MOVF REG, 0, 0`

Before Instruction
REG = 22h
W = FFh
After Instruction
REG = 22h
W = 22h

MOVFF **Move f to f**

Syntax: 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.

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 = 33h
REG2 = 11h
After Instruction
REG1 = 33h
REG2 = 33h

MOVLB **Move Literal to Low Nibble in BSR**

Syntax: MOVLW k

Operands: $0 \le k \le 255$

Operation: $k \rightarrow BSR$

Status Affected: None

Encoding:

0000	0001	kkkk	kkkk

Description: The eight-bit literal 'k' is loaded into the Bank Select Register (BSR). The value of BSR<7:4> always remains '0', regardless of the value of k_7:k_4.

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 = 02h
After Instruction
BSR Register = 05h

MOVLW **Move Literal to W**

Syntax: 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 5Ah`

After Instruction

W = 5Ah

MOVWF **Move W to f**

Syntax: MOVWF f {,a}

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

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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `MOVWF REG, 0`

Before Instruction

W = 4Fh
REG = FFh

After Instruction

W = 4Fh
REG = 4Fh

MULLW **Multiply Literal with W**

Syntax: MULLW k

Operands: $0 \le k \le 255$

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 0C4h`

Before Instruction
W = E2h
PRODH = ?
PRODL = ?
After Instruction
W = E2h
PRODH = ADh
PRODL = 08h

MULWF **Multiply W with f**

Syntax: MULWF f {,a}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `MULWF REG, 1`

Before Instruction
W = C4h
REG = B5h
PRODH = ?
PRODL = ?
After Instruction
W = C4h
REG = B5h
PRODH = 8Ah
PRODL = 94h

NEGF **Negate f**

Syntax: NEGF f {,a}

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

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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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 [3Ah]
After Instruction
REG = 1100 0110 [C6h]

NOP **No Operation**

Syntax: 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.

POP **Pop Top of Return Stack**

Syntax: 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 = 0031A2h
Stack (1 level down) = 014332h

After Instruction
TOS = 014332h
PC = NEW

PUSH **Push Top of Return Stack**

Syntax: 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

Example:

```
PUSH
```

Before Instruction
TOS = 345Ah
PC = 0124h

After Instruction
PC = 0126h
TOS = 0126h
Stack (1 level down) = 345Ah

RCALL **Relative Call**

Syntax: 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)

RESET **Reset**

Syntax: 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

RETFIE **Return from Interrupt**

Syntax: 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

RETLW **Return Literal to W**

Syntax: 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 = 07h
After Instruction
W = value of kn

RETURN **Return from Subroutine**

Syntax: 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 (TOS) 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 Instruction:
PC = TOS

RLCF **Rotate Left f through Carry**

Syntax: RLCF f {,d {,a}}

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

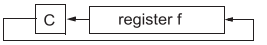
Operation: (f<n>) → dest<n + 1>,
(f<7>) → C,
(C) → dest<0>

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.



Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: RLCF REG, 0, 0

Before Instruction
REG = 1110 0110
C = 0
After Instruction
REG = 1110 0110
W = 1100 1100
C = 1

RLNCF **Rotate Left f (No Carry)**

Syntax: RLNCF f {,d {,a}}

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

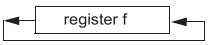
Operation: (f<n>) → dest<n + 1>,
(f<7>) → dest<0>

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.



Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: RLNCF REG, 1, 0

Before Instruction
REG = 1010 1011
After Instruction
REG = 0101 0111

RRCF **Rotate Right f through Carry**

Syntax: RRCF 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>) → 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.



Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: RRCF REG, 0, 0

Before Instruction
REG = 1110 0110
C = 0
After Instruction
REG = 1110 0110
W = 0111 0011
C = 0

RRNCF **Rotate Right f (No Carry)**

Syntax: RRNCF f {,d {,a}}

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

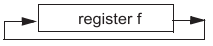
Operation: (f<n>) → dest<n – 1>,
(f<0>) → dest<7>

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).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.



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, 0, 0
Before Instruction
    W    =  ?
    REG  =  1101 0111
After Instruction
    W    =  1110 1011
    REG  =  1101 0111
```

SETF **Set f**

Syntax: SETF f {,a}

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

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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example:
```
SETF    REG, 1
Before Instruction
    REG  =  5Ah
After Instruction
    REG  =  FFh
```

RRNCF **Rotate Right f (No Carry)**

Syntax: RRNCF f {,d {,a}}

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

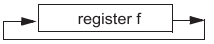
Operation: (f<n>) → dest<n – 1>,
(f<0>) → dest<7>

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).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.



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, 0, 0
```
Before Instruction
W = ?
REG = 1101 0111
After Instruction
W = 1110 1011
REG = 1101 0111

SETF **Set f**

Syntax: SETF f {,a}

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

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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example:
```
SETF    REG, 1
```
Before Instruction
REG = 5Ah
After Instruction
REG = FFh

SLEEP **Enter Sleep mode**

Syntax: SLEEP

Operands: None

Operation: 00h → WDT,
0 → WDT postscaler,
1 → $\overline{TO}$,
0 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

0000	0000	0000	0011

Description: The Power-Down status bit ($\overline{PD}$) is cleared. The Time-out status bit ($\overline{TO}$) is set. Watchdog Timer and its postscaler are cleared.
The processor is put into Sleep mode with the oscillator stopped.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No operation	Process Data	Go to Sleep

Example: SLEEP

Before Instruction
$\overline{TO}$ = ?
$\overline{PD}$ = ?
After Instruction
$\overline{TO}$ = 1†
$\overline{PD}$ = 0

† If WDT causes wake-up, this bit is cleared.

SUBFWB **Subtract f from W with Borrow**

Syntax: SUBFWB f {,d {,a}}

Operands: 0 ≤ f ≤ 255
d ∈ [0,1]
a ∈ [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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example 1: SUBFWB REG, 1, 0
Before Instruction
REG = 3
W = 2
C = 1
After Instruction
REG = FF
W = 2
C = 0
Z = 0
N = 1 ; 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

SUBLW **Subtract W from Literal**

Syntax: SUBLW k

Operands: $0 \le k \le 255$

Operation: $k - (W) \rightarrow W$

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   02h
Before Instruction
    W   =   01h
    C   =   ?
After Instruction
    W   =   01h
    C   =   1    ; result is positive
    Z   =   0
    N   =   0
```

Example 2:
```
SUBLW   02h
Before Instruction
    W   =   02h
    C   =   ?
After Instruction
    W   =   00h
    C   =   1    ; result is zero
    Z   =   1
    N   =   0
```

Example 3:
```
SUBLW   02h
Before Instruction
    W   =   03h
    C   =   ?
After Instruction
    W   =   FFh  ; (2's complement)
    C   =   0    ; result is negative
    Z   =   0
    N   =   1
```

SUBWF **Subtract W from f**

Syntax: SUBWF f {,d {,a}}

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

Operation: $(f) - (W) \rightarrow dest$

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example 1:
```
SUBWF   REG, 1, 0
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, 0, 0
Before Instruction
    REG   =   2
    W     =   2
    C     =   ?
After Instruction
    REG   =   2
    W     =   0
    C     =   1    ; esult is rero     z
    Z     =   1
    N     =   0
```

Example 3:
```
SUBWF   REG, 1, 0
Before Instruction
    REG   =   1
    W     =   2
    C     =   ?
After Instruction
    REG   =   FFh  ;(2's complement)
    W     =   2
    C     =   0    ; result is negative
    Z     =   0
    N     =   1
```

SUBWFB **Subtract W from f with Borrow**

Syntax: 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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  =  19h     (0001 1001)
    W    =  0Dh     (0000 1101)
    C    =  1
After Instruction
    REG  =  0Ch     (0000 1011)
    W    =  0Dh     (0000 1101)
    C    =  1
    Z    =  0
    N    =  0       ; result is positive
```

Example 2:
```
SUBWFB  REG, 0, 0
Before Instruction
    REG  =  1Bh     (0001 1011)
    W    =  1Ah     (0001 1010)
    C    =  0
After Instruction
    REG  =  1Bh     (0001 1011)
    W    =  00h
    C    =  1
    Z    =  1       ; result is zero
    N    =  0
```

Example 3:
```
SUBWFB  REG, 1, 0
Before Instruction
    REG  =  03h     (0000 0011)
    W    =  0Eh     (0000 1101)
    C    =  1
After Instruction
    REG  =  F5h     (1111 0100)
                    ; [2's comp]
    W    =  0Eh     (0000 1101)
    C    =  0
    Z    =  0
    N    =  1       ; result is negative
```

SWAPF **Swap f**

Syntax: SWAPF f {,d {,a}}

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

Operation: (f<3:0>) → dest<7:4>,
(f<7:4>) → 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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever f ≤ 95 (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example:
```
SWAPF   REG, 1, 0
Before Instruction
    REG  =  53h
After Instruction
    REG  =  35h
```

TBLRD **Table Read**

Syntax: 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 = 55h
TBLPTR = 00A356h
MEMORY (00A356h) = 34h
After Instruction
TABLAT = 34h
TBLPTR = 00A357h

Example 2:

```
TBLRD   +* ;
```

Before Instruction
TABLAT = AAh
TBLPTR = 01A357h
MEMORY (01A357h) = 12h
MEMORY (01A358h) = 34h
After Instruction
TABLAT = 34h
TBLPTR = 01A358h

TBLWT T able Write

Syntax: 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

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)

TBLWT Table Write (Continued)

Example 1: `TBLWT *+;`

Before Instruction
TABLAT = 55h
TBLPTR = 00A356h
HOLDING REGISTER (00A356h) = FFh
After Instructions (table write completion)
TABLAT = 55h
TBLPTR = 00A357h
HOLDING REGISTER (00A356h) = 55h

Example 2: `TBLWT +*;`

Before Instruction
TABLAT = 34h
TBLPTR = 01389Ah
HOLDING REGISTER (01389Ah) = FFh
HOLDING REGISTER (01389Bh) = FFh
After Instruction (table write completion)
TABLAT = 34h
TBLPTR = 01389Bh
HOLDING REGISTER (01389Ah) = FFh
HOLDING REGISTER (01389Bh) = 34h

TSTFSZ **Test f, Skip if 0**

Syntax: TSTFSZ f {,a}

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

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 is selected.
If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

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, 1
NZERO   :
ZERO    :
```
Before Instruction
PC = Address (HERE)
After Instruction
If CNT = 00h,
PC = Address (ZERO)
If CNT ≠ 00h,
PC = Address (NZERO)

XORLW **Exclusive OR Literal with W**

Syntax: 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 XORed 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   0AFh
```
Before Instruction
W = B5h
After Instruction
W = 1Ah

XORWF **Exclusive OR W with f**

Syntax: XORWF f {,d {,a}}

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

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 is selected. If 'a' is '1', the BSR is used to select the GPR bank (default).
If 'a' is '0' and the extended instruction set is enabled, this instruction operates in Indexed Literal Offset Addressing mode whenever $f \le 95$ (5Fh). See **Section 24.2.3 "Byte-Oriented and Bit-Oriented Instructions in Indexed Literal Offset Mode"** for details.

Words: 1

Cycles: 1

Q Cycle Activity:

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

Example: `XORWF REG, 1, 0`

Before Instruction
REG = AFh
W = B5h
After Instruction
REG = 1Ah
W = B5h

APPENDIX E: PIC18F4321 SPECIAL FUNCTION REGISTERS

SPECIAL FUNCTION REGISTERS

The Special Function Registers (SFRs) are registers used by the CPU and peripheral modules for controlling the desired operation of the device. These registers are implemented as static RAM. SFRs start at the top of data memory (FFFh) and extend downward to occupy the top half of Bank 15 (F80h to FFFh). A list of these registers is given in Table E-1.

TABLE E-1: SPECIAL FUNCTION REGISTER MAP FOR PIC18F4321 FAMILY DEVICES

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(1)	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2(1)	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2(1)	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2(1)	FBCh	CCPR2H	F9Ch	—(2)
FFBh	PCLATU	FDBh	PLUSW2(1)	FBBh	CCPR2L	F9Bh	OSCTUNE
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—(2)
FF9h	PCL	FD9h	FSR2L	FB9h	—(2)	F99h	—(2)
FF8h	TBLPTRU	FD8h	STATUS	FB8h	BAUDCON	F98h	—(2)
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	ECCP1DEL(3)	F97h	—(2)
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCP1AS(3)	F96h	TRISE(3)
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD(3)
FF4h	PRODH	FD4h	—(2)	FB4h	CMCON	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	HLVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—(2)
FF0h	INTCON3	FD0h	RCON	FB0h	SPBRGH	F90h	—(2)
FEFh	INDF0(1)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—(2)
FEEh	POSTINC0(1)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—(2)
FEDh	POSTDEC0(1)	FCDh	T1CON	FADh	TXREG	F8Dh	LATE(3)
FECh	PREINC0(1)	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD(3)
FEBh	PLUSW0(1)	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—(2)	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—(2)
FE7h	INDF1(1)	FC7h	SSPSTAT	FA7h	EECON2(1)	F87h	—(2)
FE6h	POSTINC1(1)	FC6h	SSPCON1	FA6h	EECON1	F86h	—(2)
FE5h	POSTDEC1(1)	FC5h	SSPCON2	FA5h	—(2)	F85h	—(2)
FE4h	PREINC1(1)	FC4h	ADRESH	FA4h	—(2)	F84h	PORTE(3)
FE3h	PLUSW1(1)	FC3h	ADRESL	FA3h	—(2)	F83h	PORTD(3)
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	ADCON2	FA0h	PIE2	F80h	PORTA

Note 1: This is not a physical register.
2: Unimplemented registers are read as '0'.
3: This register is not available on 28-pin devices.

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

APPENDIX F: TUTORIAL FOR ASSEMBLING AND DEBUGGING A PIC18F ASSEMBLY LANGUAGE PROGRAM USING THE MPLAB

The PIC18F assembly language programs in this book are assembled using Microchip MPLAB assembler. Appendix F provides a tutorial showing step-by-step procedure to download the MPLAB assembler from Microchip's website (www.microchip.com), and assemble a simple PIC18F assembly language program using MPLAB IDE v8.50. Microchip introduces newer versions of the MPLAB assembler frequently. One may download the MPLAB IDE v8.50 from the archive from Microchip website.

Assembling PIC18F assembly language program using MPLAB

First, download MPLAB IDE v8.50 from the Microchip website as follows:

1. Go to www.microchip.com.
2. Type in MPLAB IDE v8.50 in the "search" and hit return.
3. Click on MPLAB IDE v8.50.

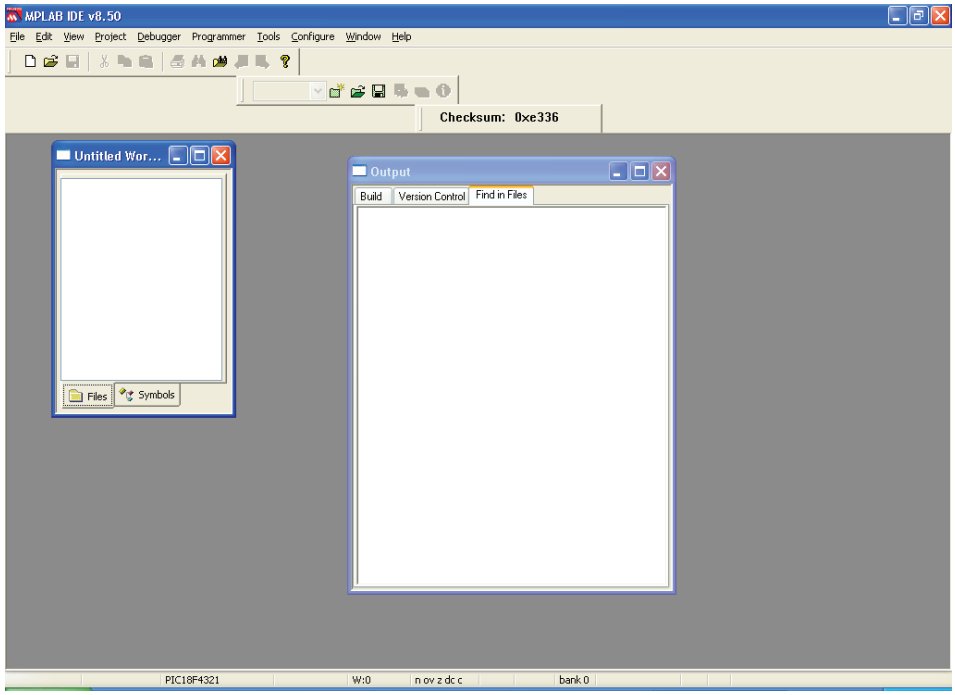
After installing and downloading the program, you will see the following icon on your desktop:



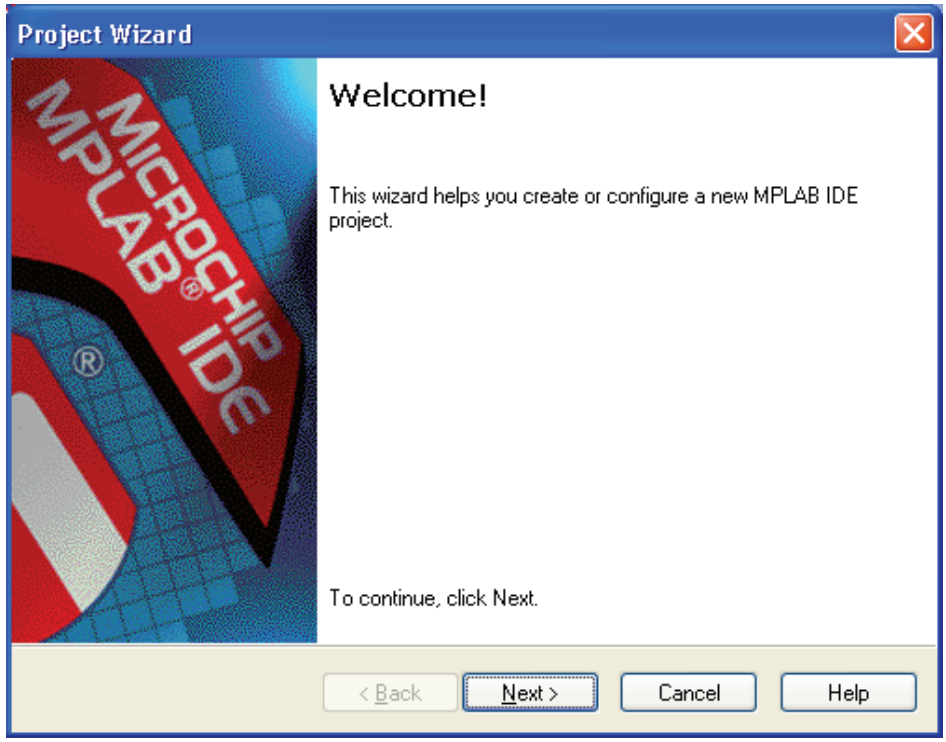
MPLAB IDE v8.50.lnk

Double click (right) on the MPLAB icon and wait until you see the following screen:

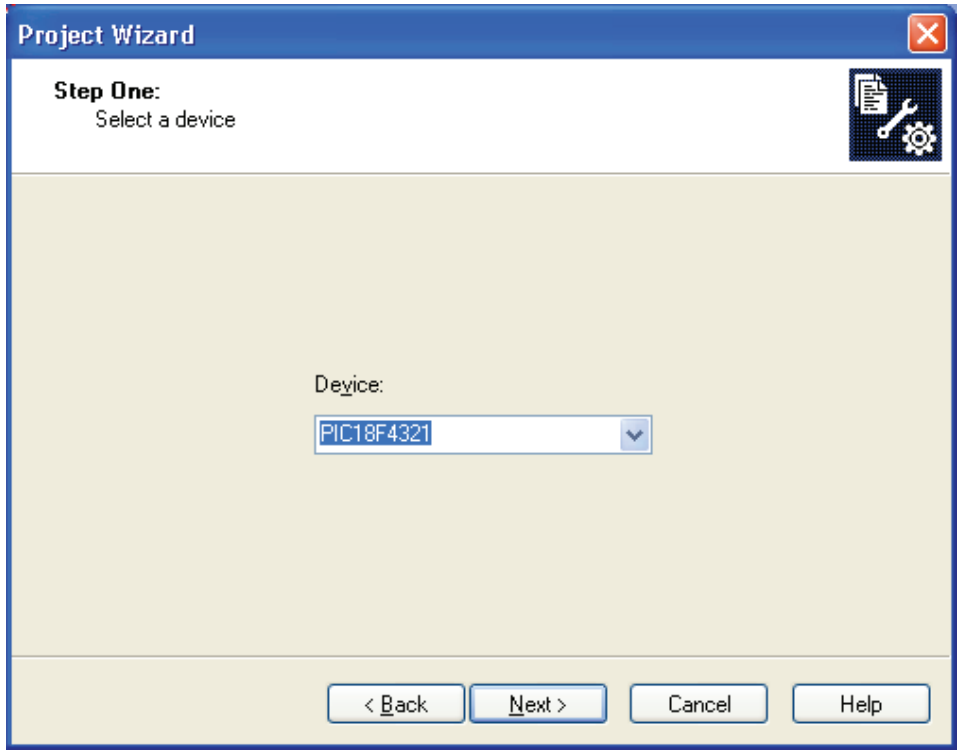
© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.



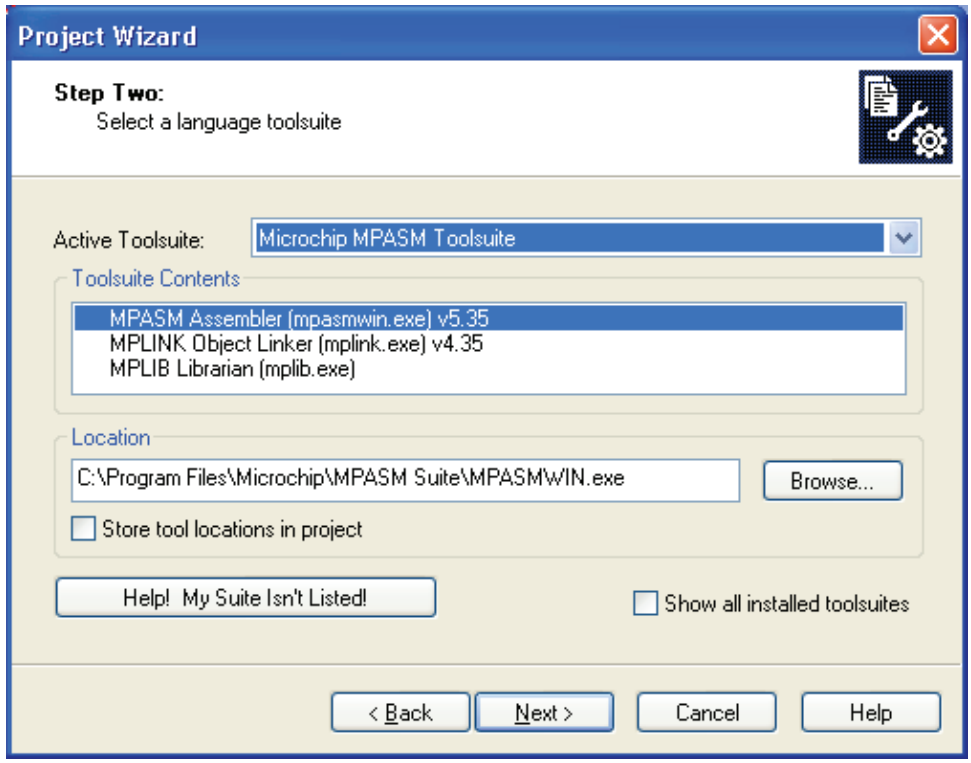
Next, click on ‘Project’ and then ‘Project Wizard’; the following screen will appear:



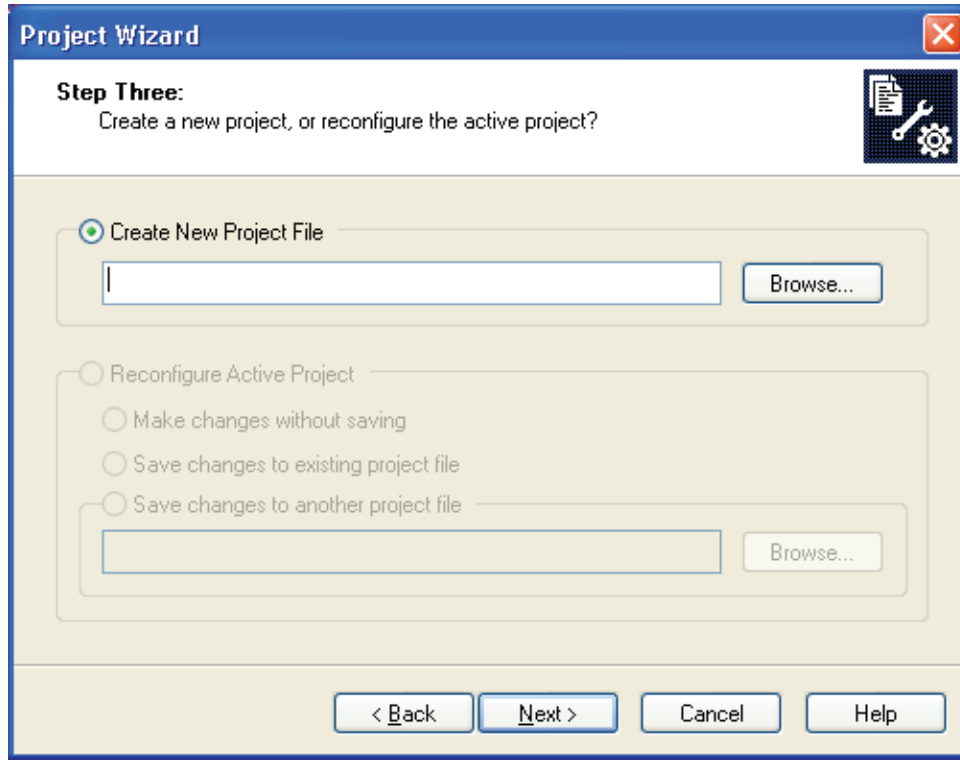
Click Next; the following screen shot will be displayed:



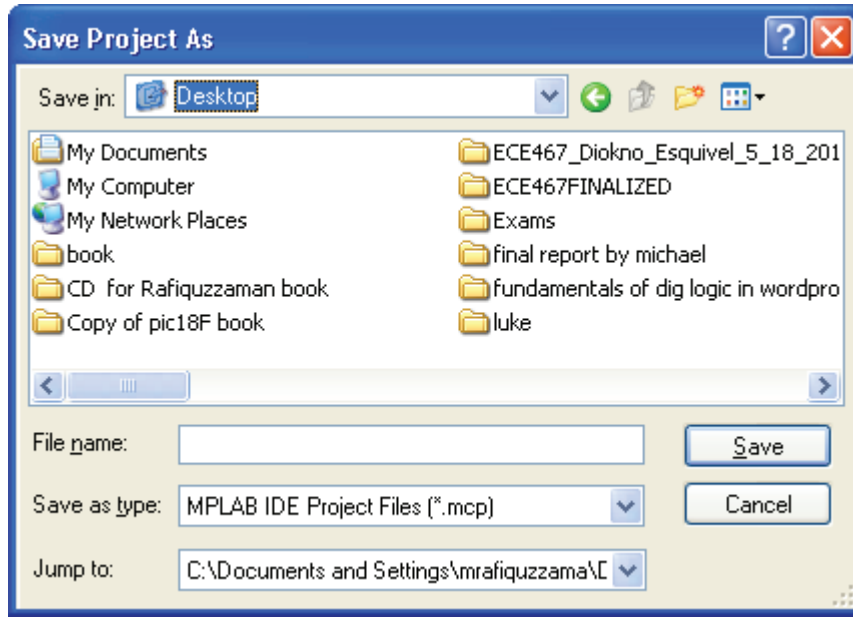
Select the device PIC18F4321, hit Next, and wait; the following will be displayed:



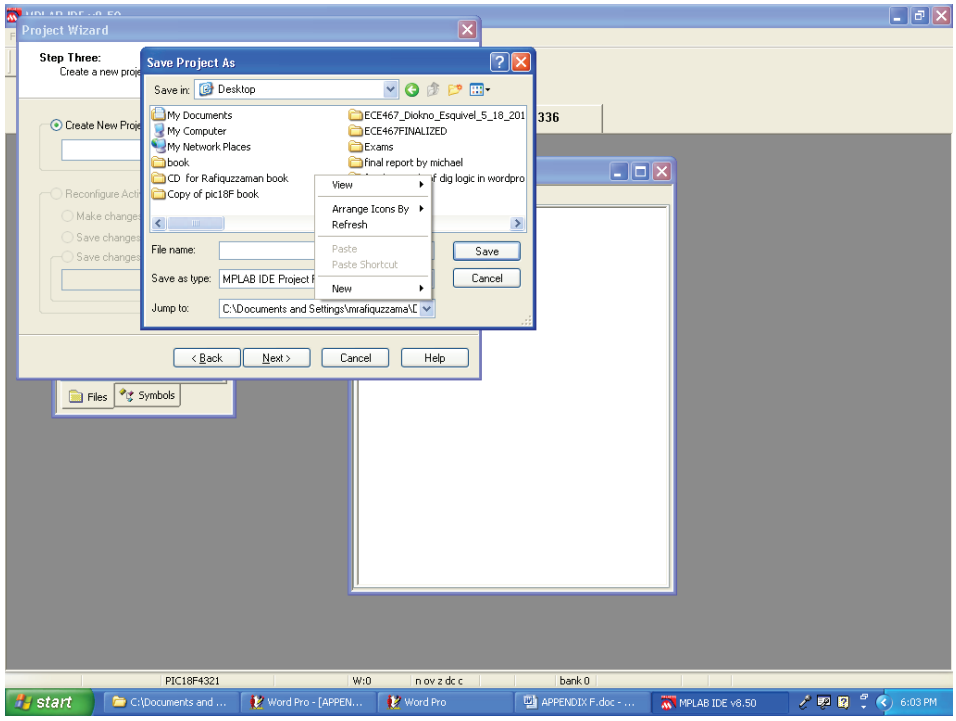
In the 'Active Toolsuite', select 'Microchip MPASM Toolsuite', and click Next; the following will be displayed:



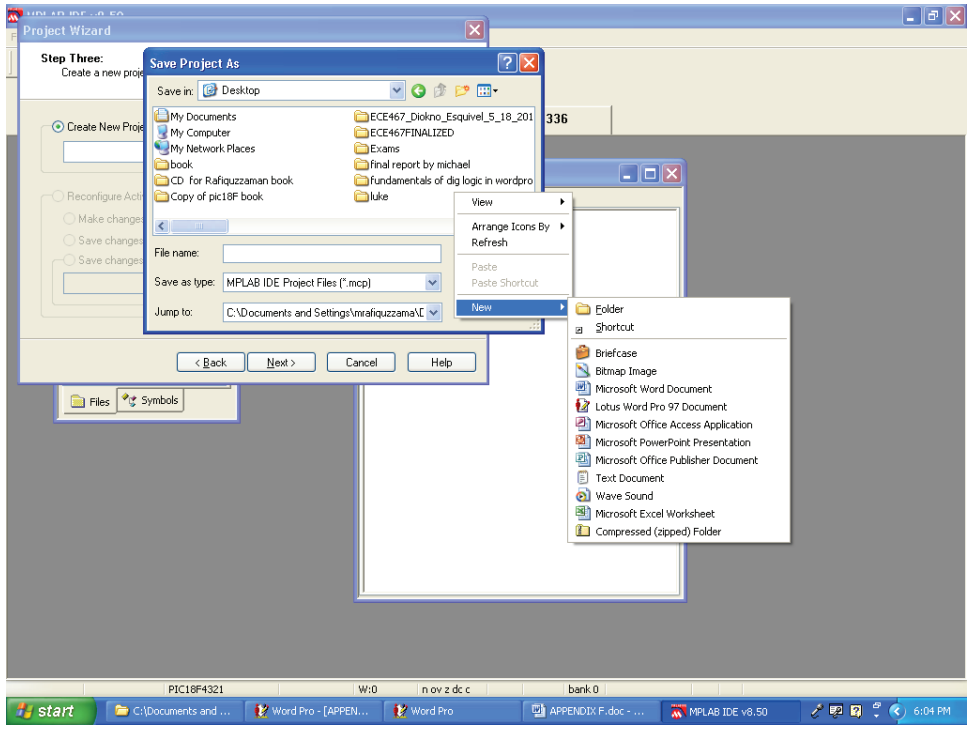
Select a location where all project contents will be placed. For this example, the folder will be placed on the desktop (arbitrarily chosen). Go to the desktop directory, make a new folder, and name the folder. In order to do this, click on 'Browse', and select desktop:



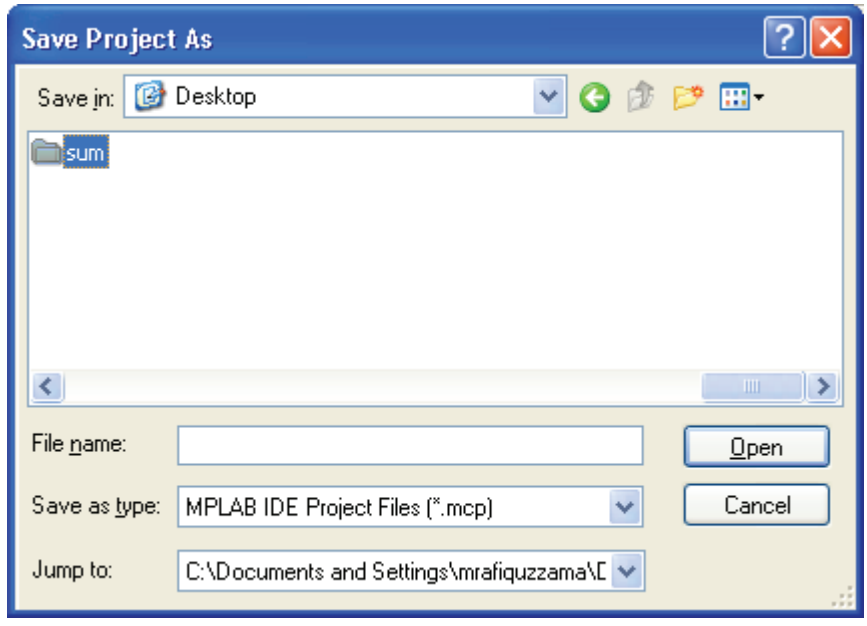
Next, create a new folder by clicking on the icon (second yellow icon from right on top row) or by right clicking on the mouse on the above window; see the following screen:



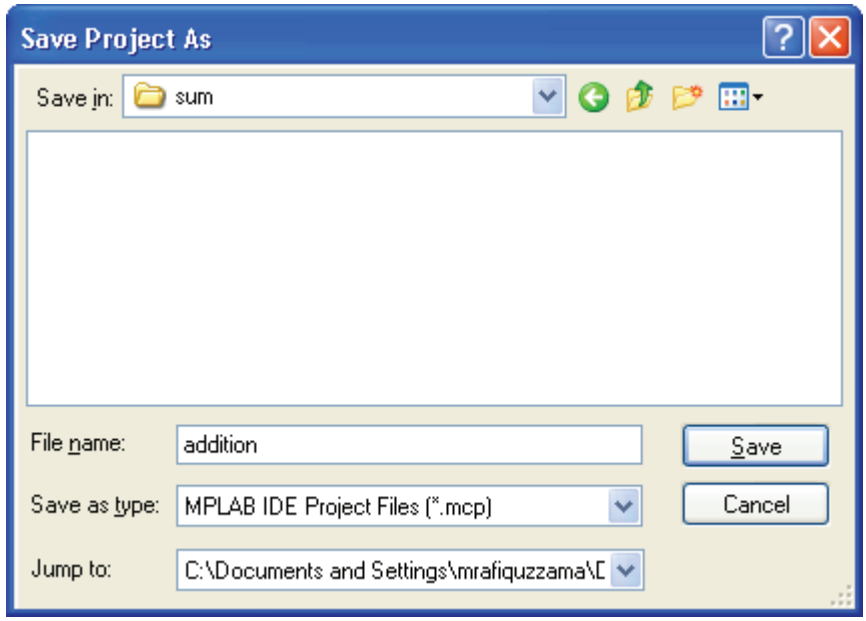
Next, click on ‘New’ to see the following:



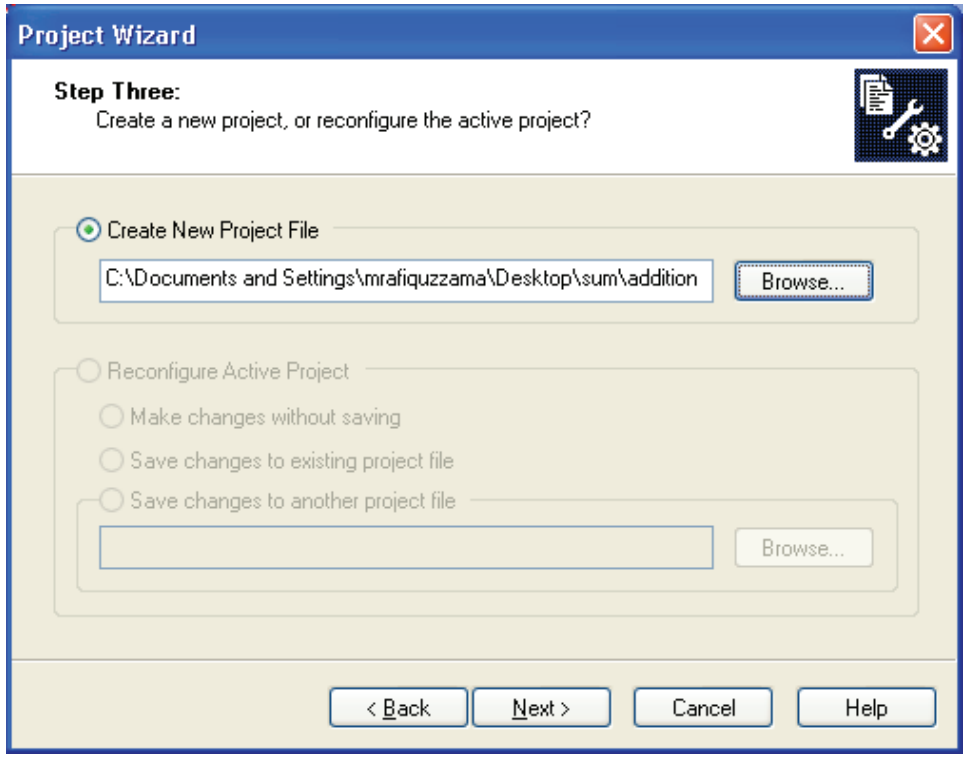
Click on folder, name it 'sum' (arbitrarily chosen name), and see the following :



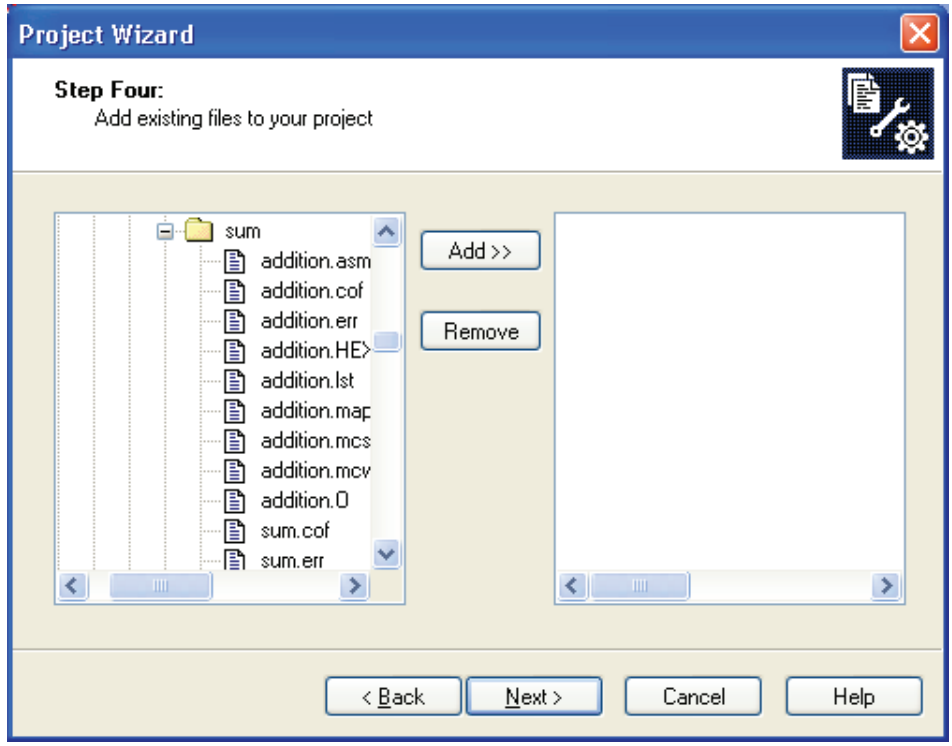
Next, type in a file name such as addition (arbitrarily chosen name), and see the following on the screen:



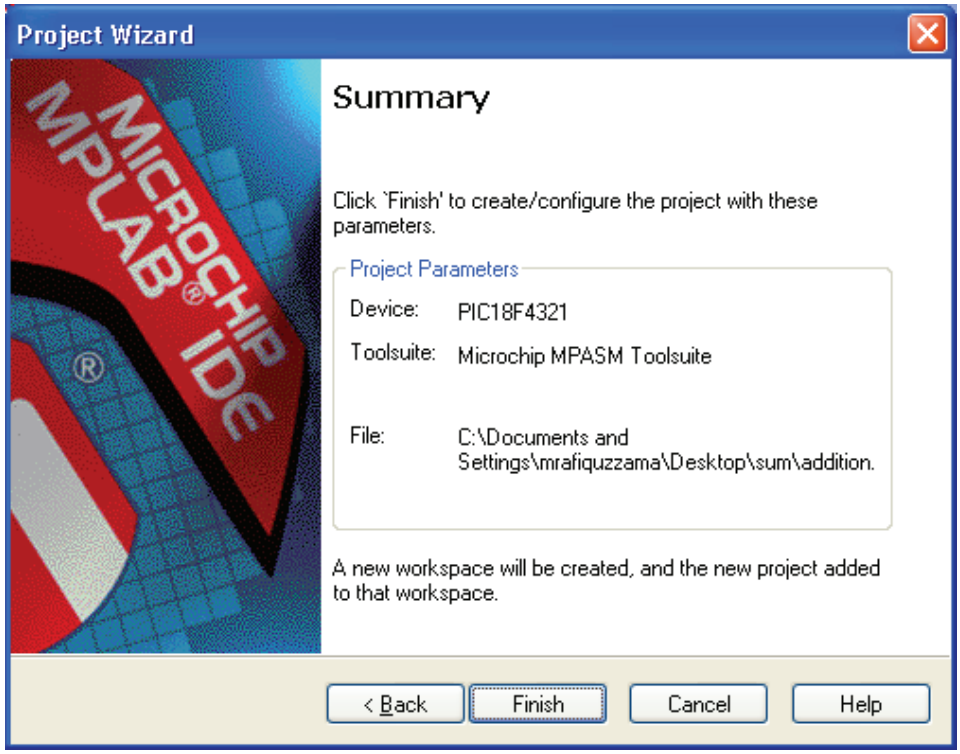
Next, click on Save; the following screen will appear:



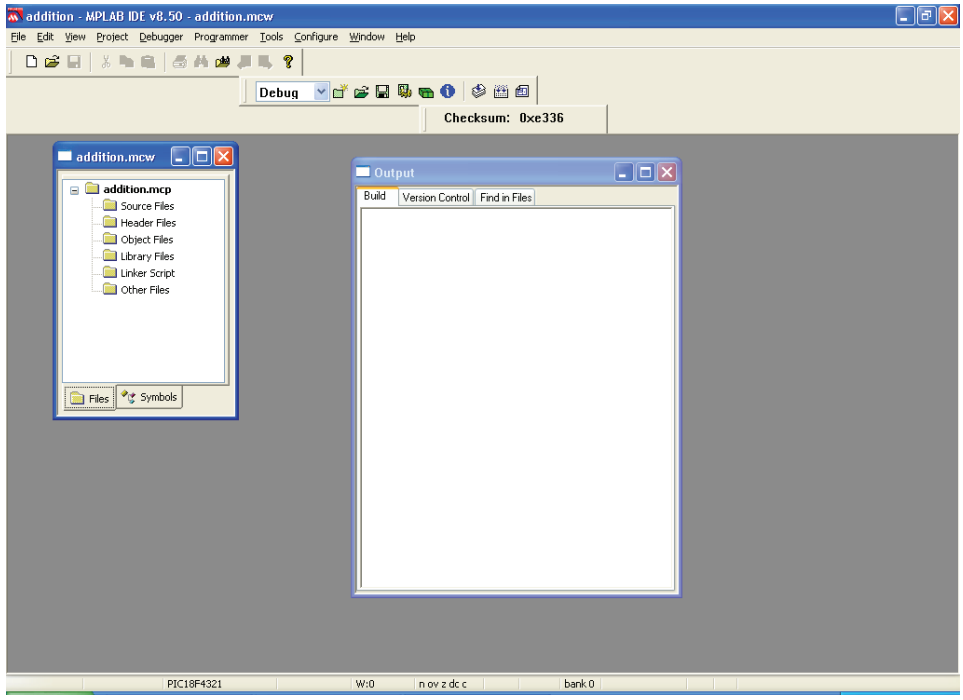
Click on Next, and see the following:



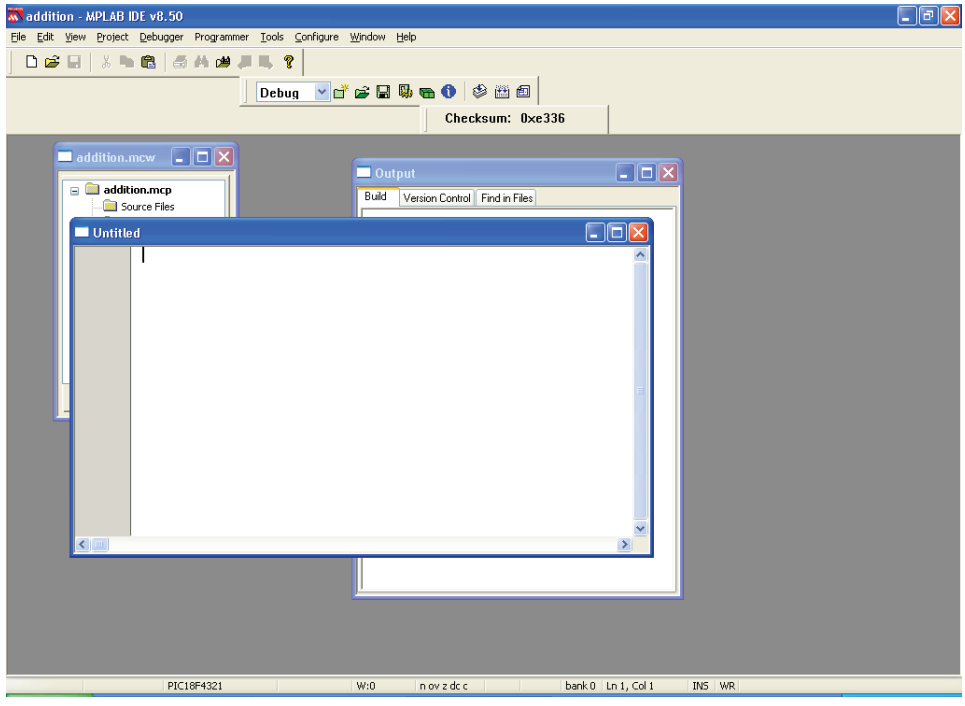
Click on Next, and see the following:



Click on Finish, and see the following:



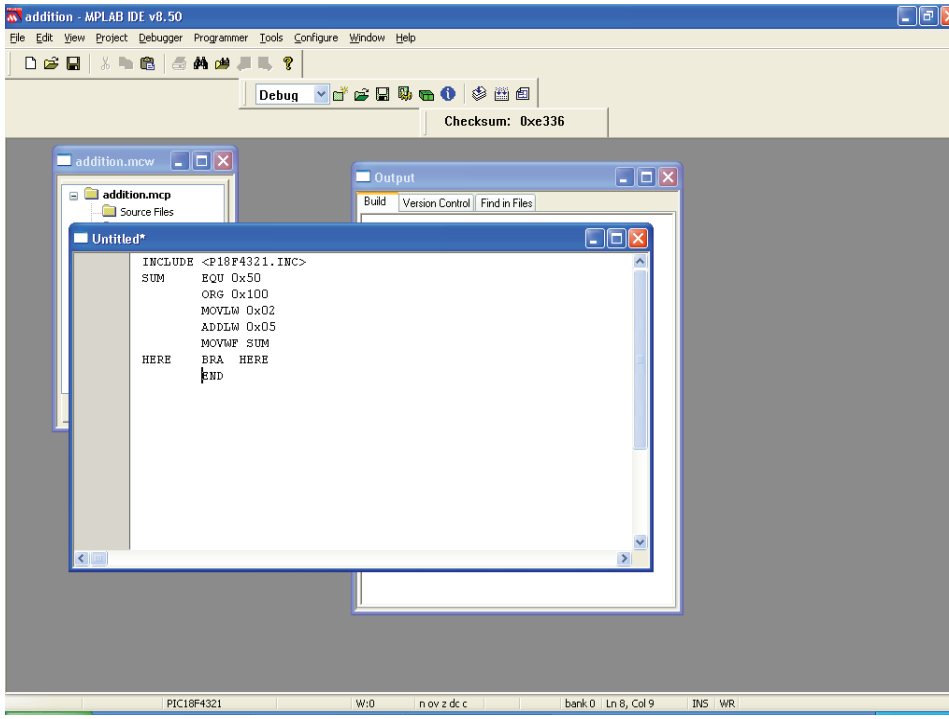
Click on File, and then New to see the following:



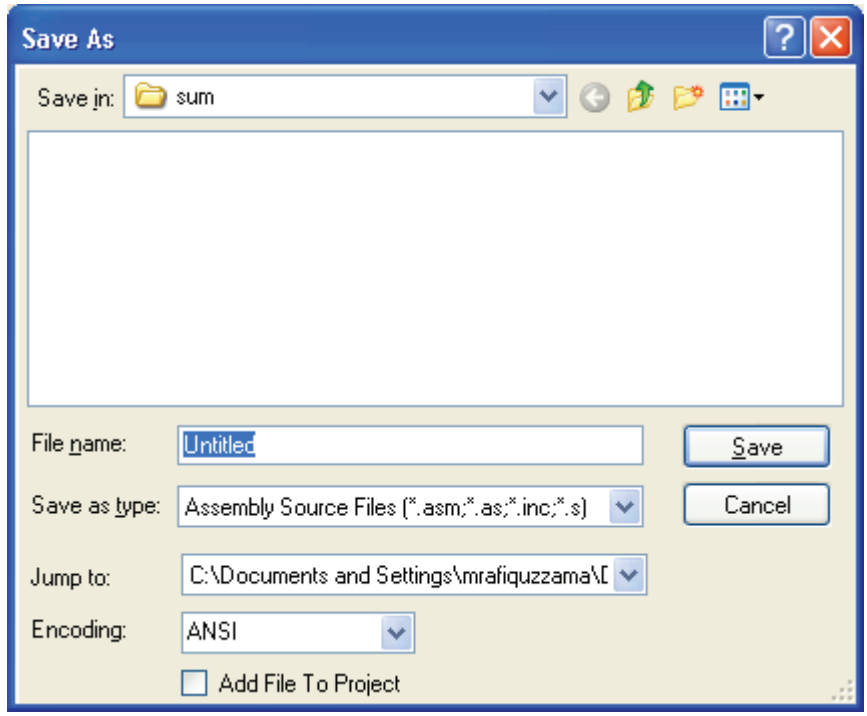
Type in the program you want to assemble. The following addition program is entered:

```
INCLUDE   <P18F4321.INC>
SUM       EQU     0x50
          ORG     0x100
          MOVLW   0x02
          ADDLW   0x05
          MOVWF   SUM
HERE      BRA     HERE
END
```

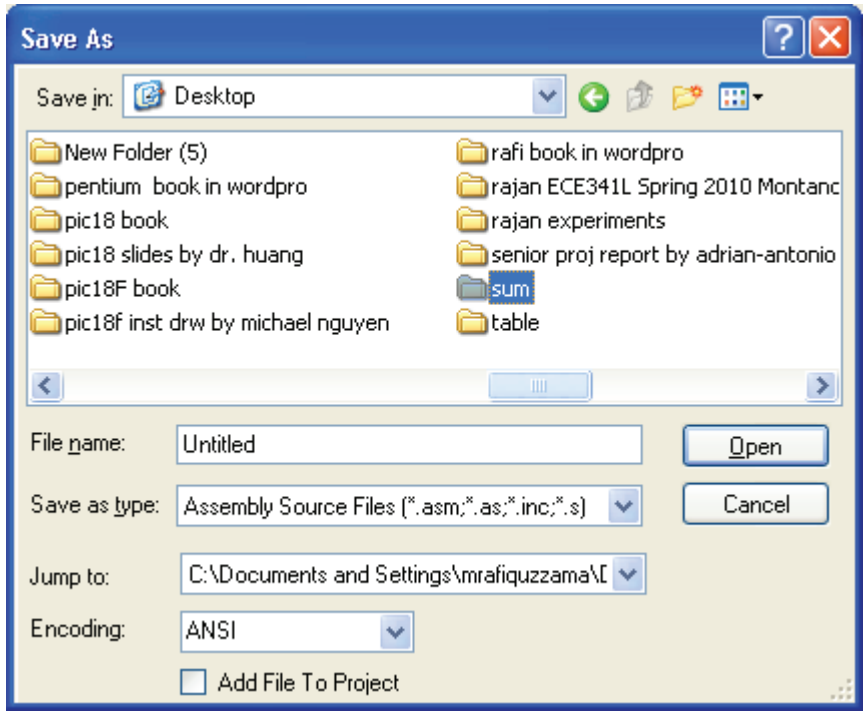
After entering the program, see the following:



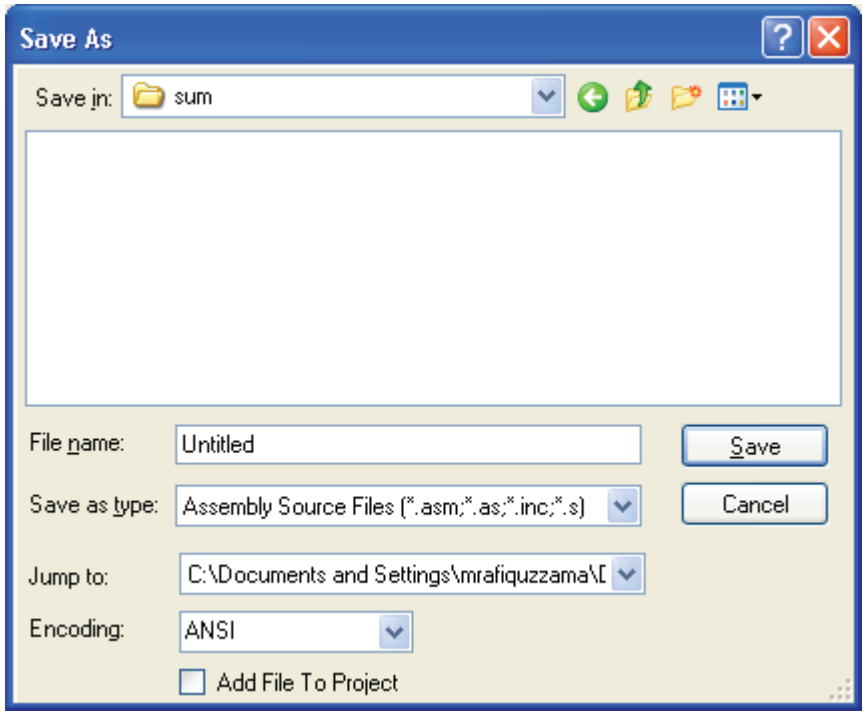
Next, click on File, and Save as, and see the following:



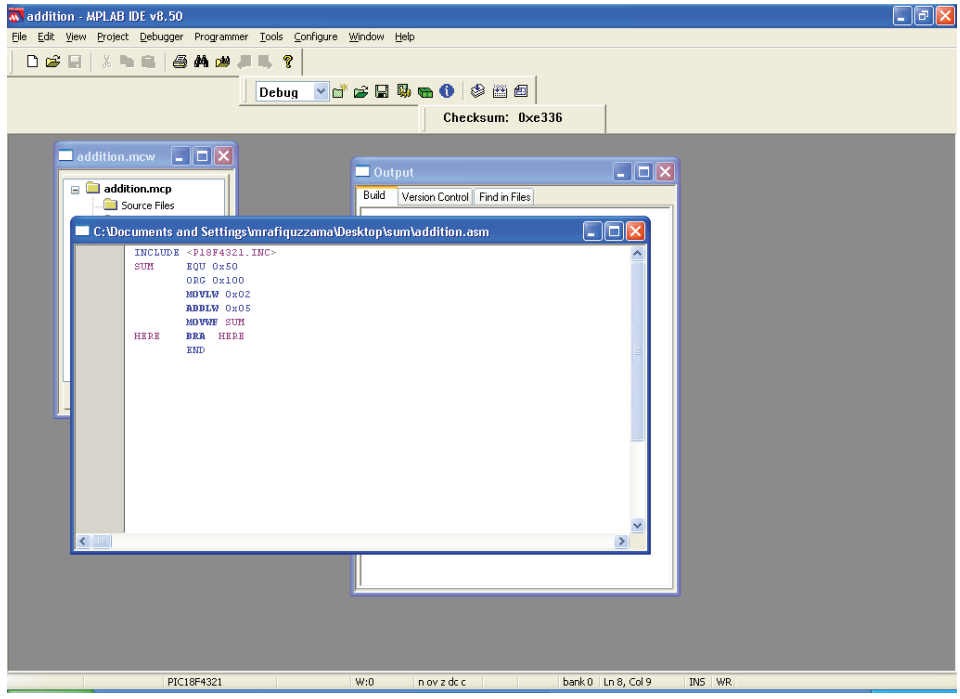
Make sure you scroll up to desktop, and then click on sum (the folder that was created before), and see the following:



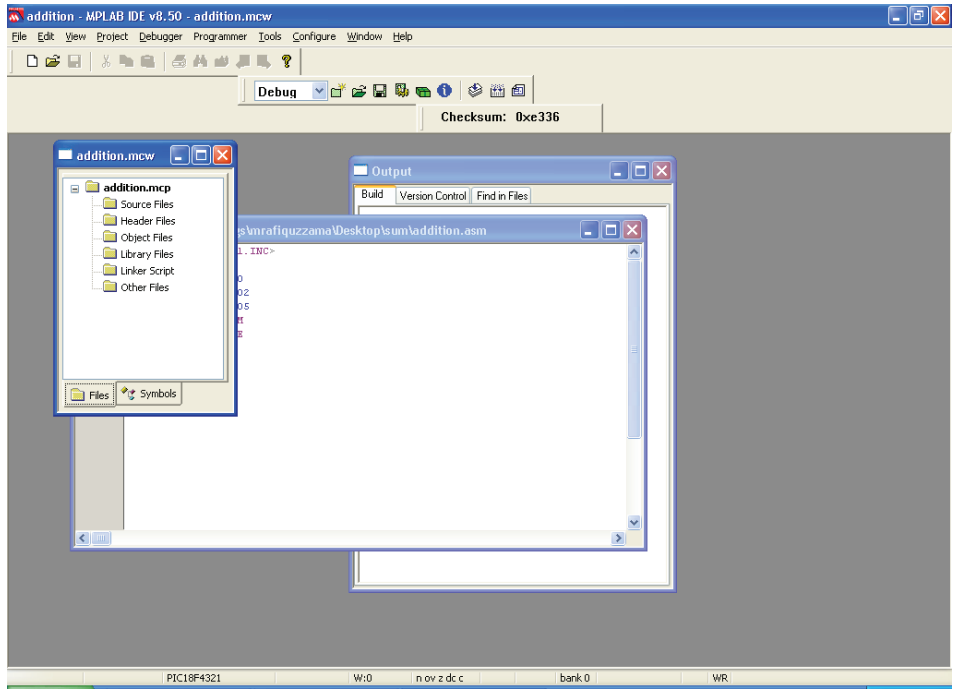
Next, double click (left) on sum to see the following:



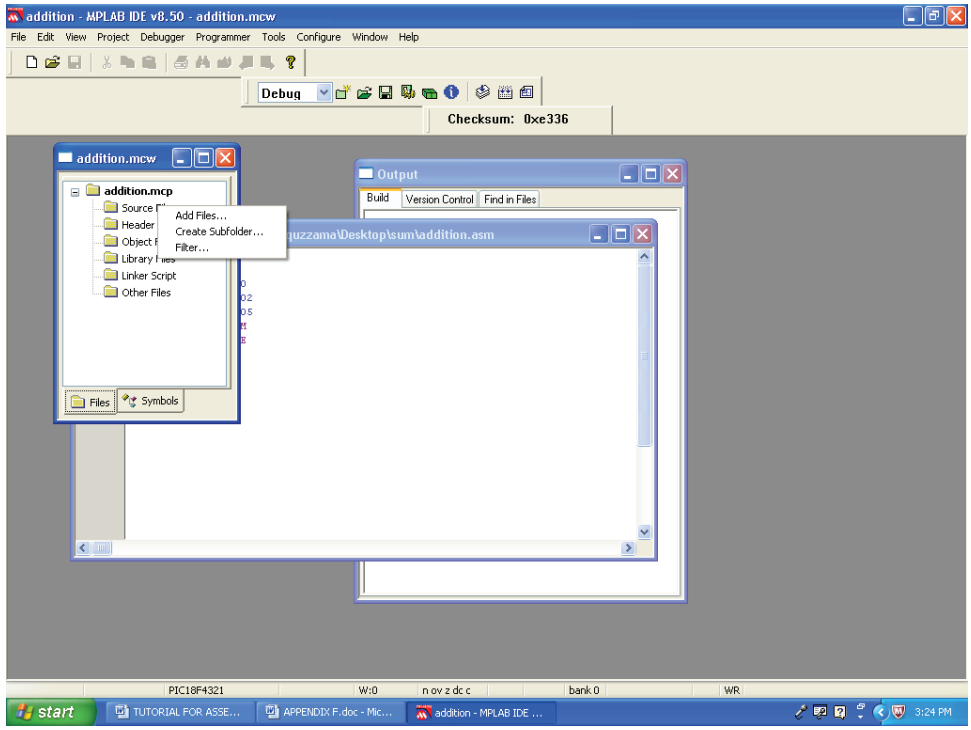
Delete Untitled, and enter the same file name 'addition' with .asm extension as File name. Click on save, and see the following screen shot (notice the display changes color):



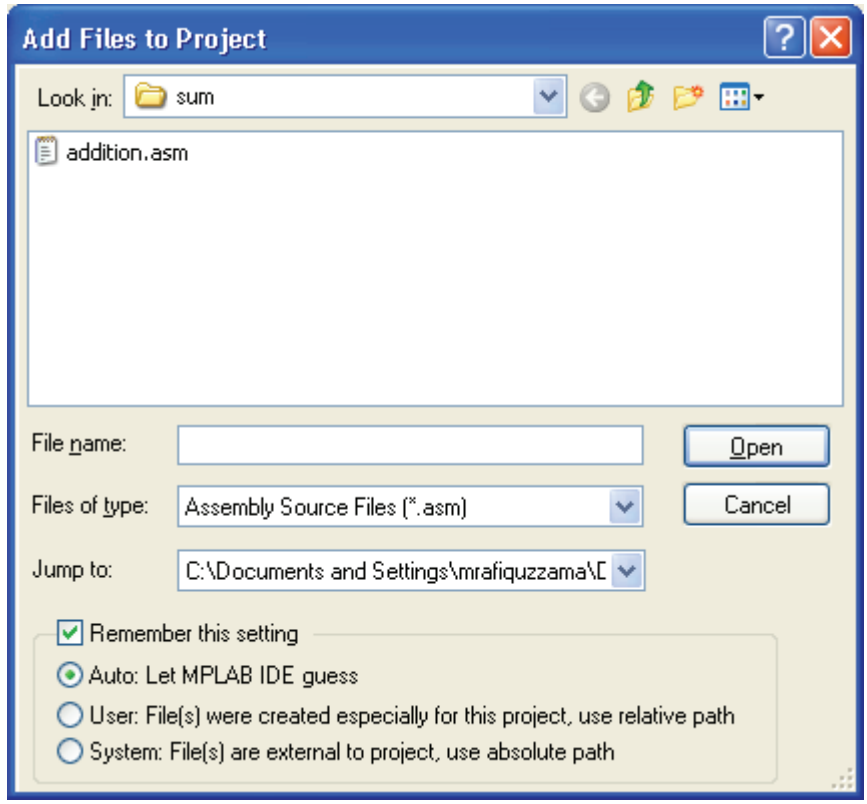
Next, highlight by clicking on the top (blue) section of addition.asm.mcp, and see the following:



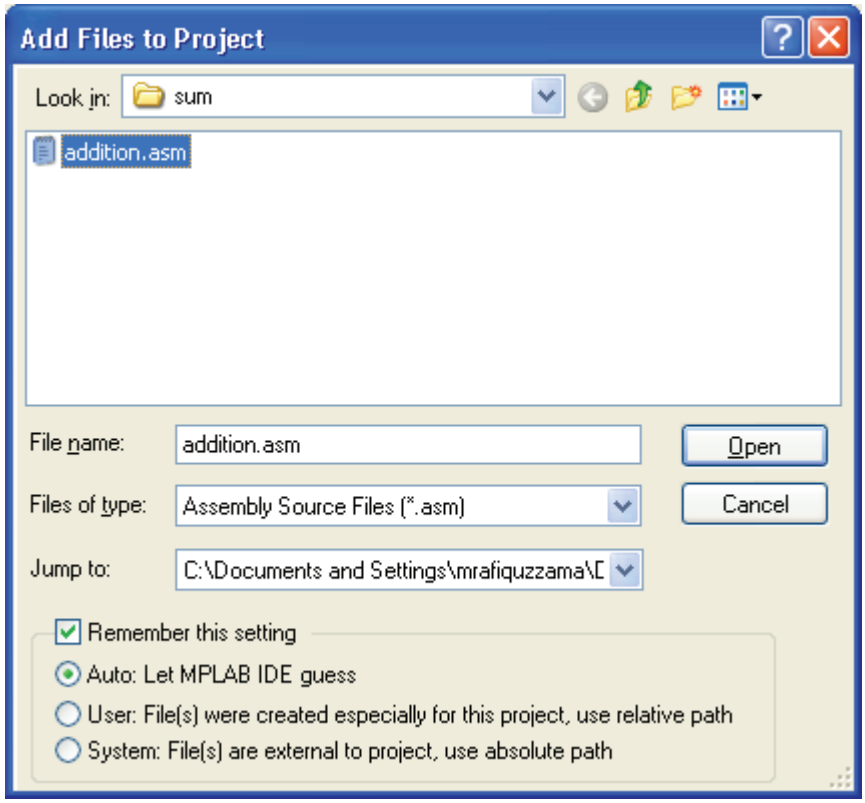
Right click on Source Files to see the following:



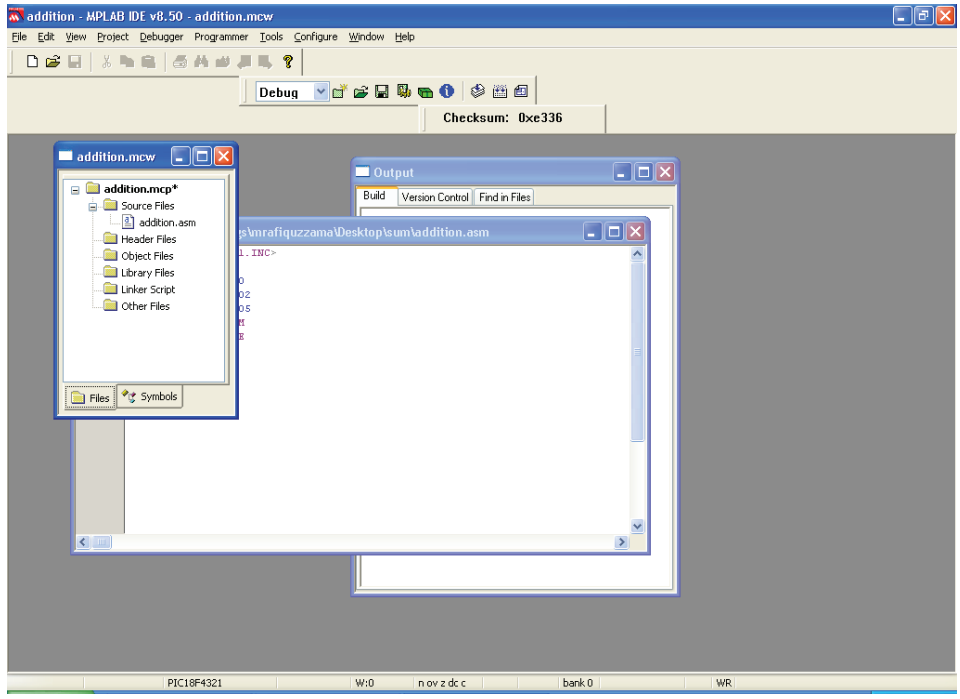
Click on Add files to see the following:



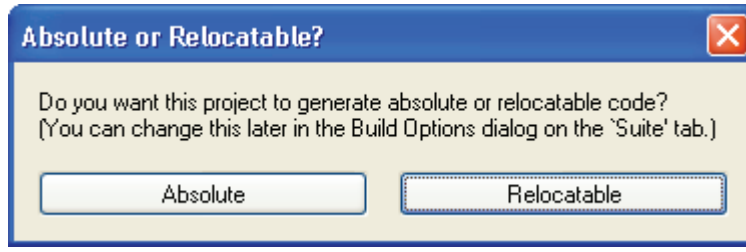
Click on addition.asm to see the following screen shot:



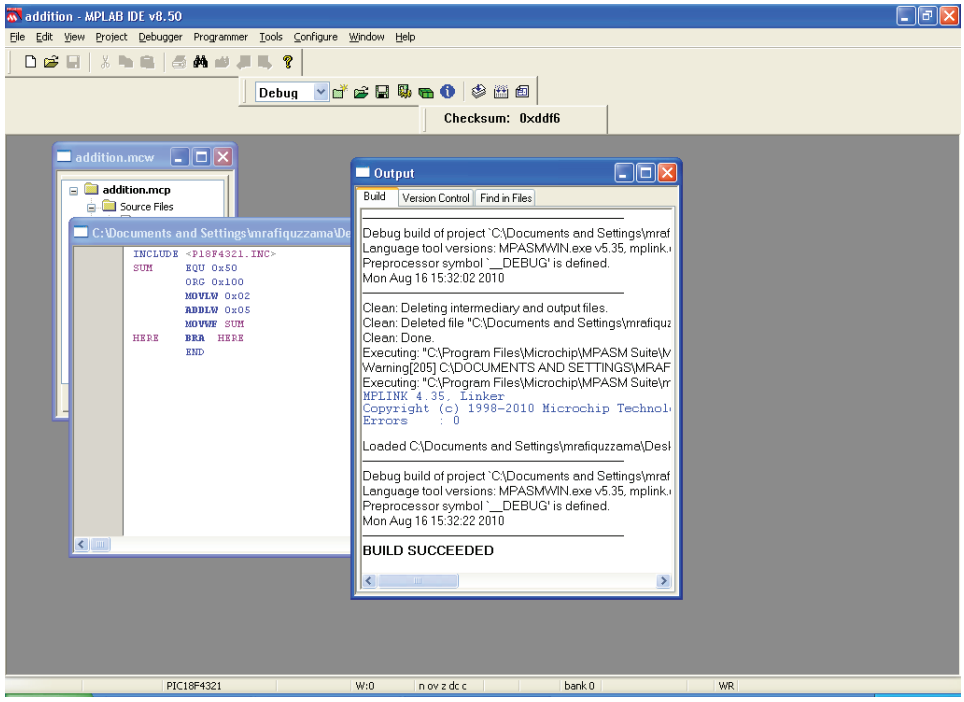
Click Open to see the following:



Note that addition.asm is listed under Source Files. Next, click on Project and then build all (or only the 'Build All' icon, second icon on top right of the Debug toolbar), and see the following:

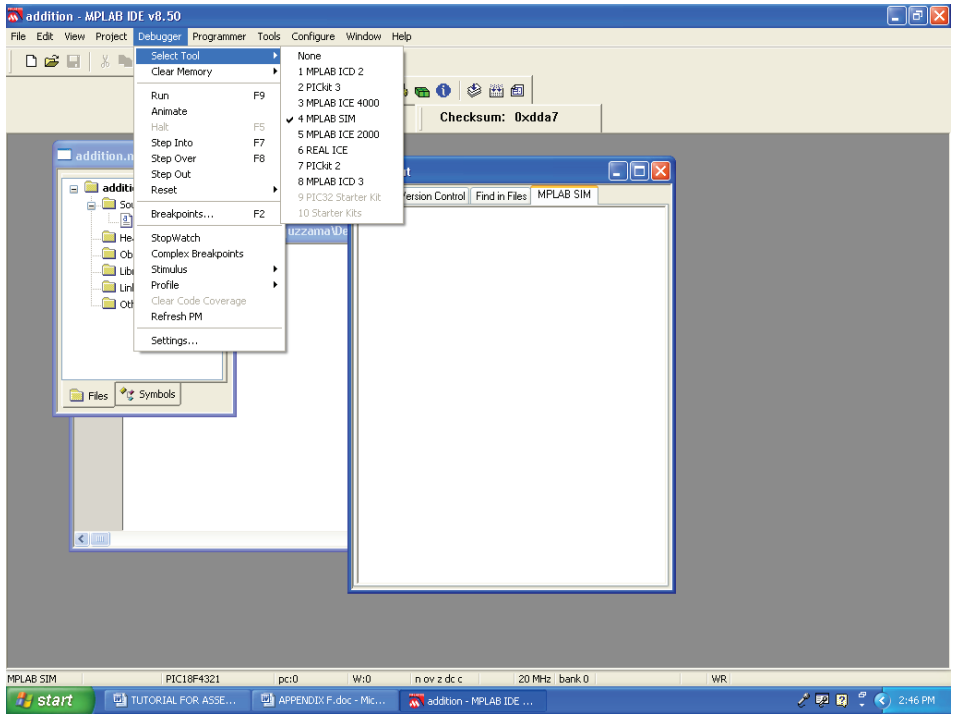


Next, click on Absolute to see the following:

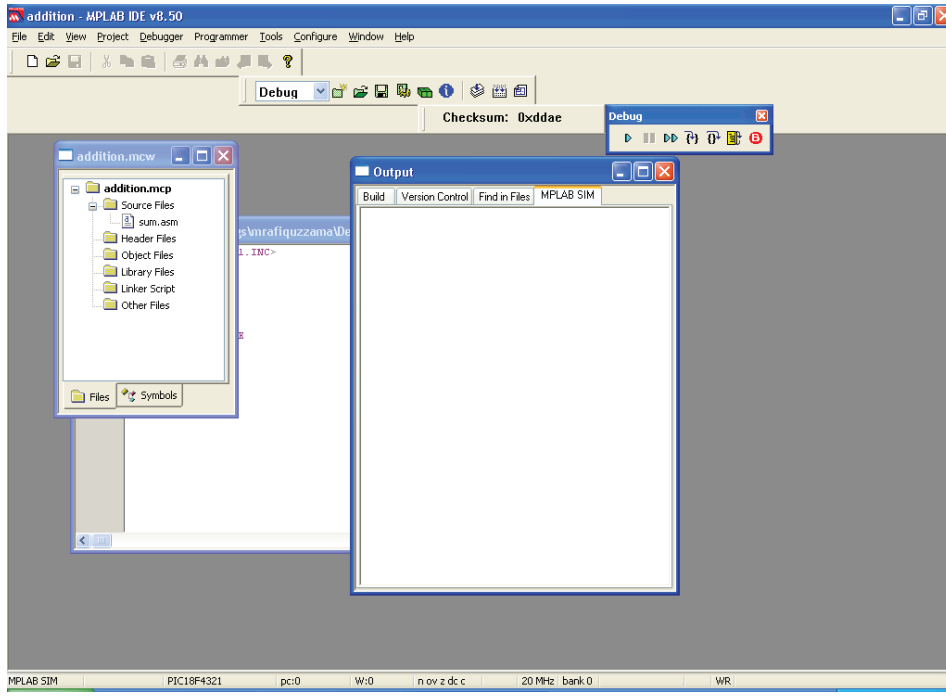


This means that the assembling the program is successful. Next the result will be verified using the debugger.

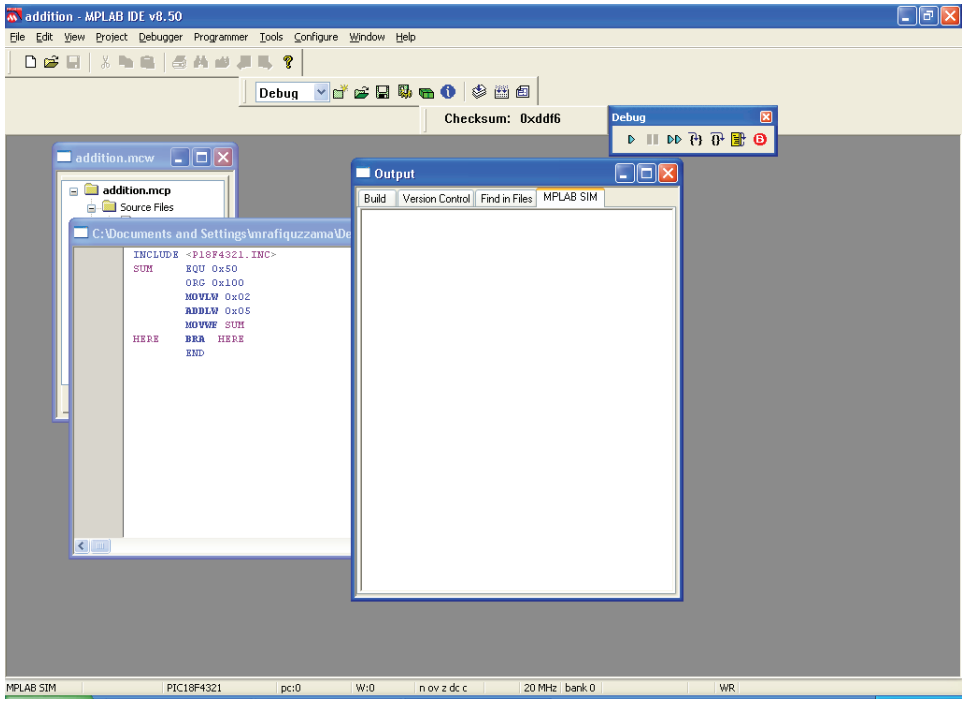
Click on Debugger, Select Tool, and then MPLAB SIM to see the following display:



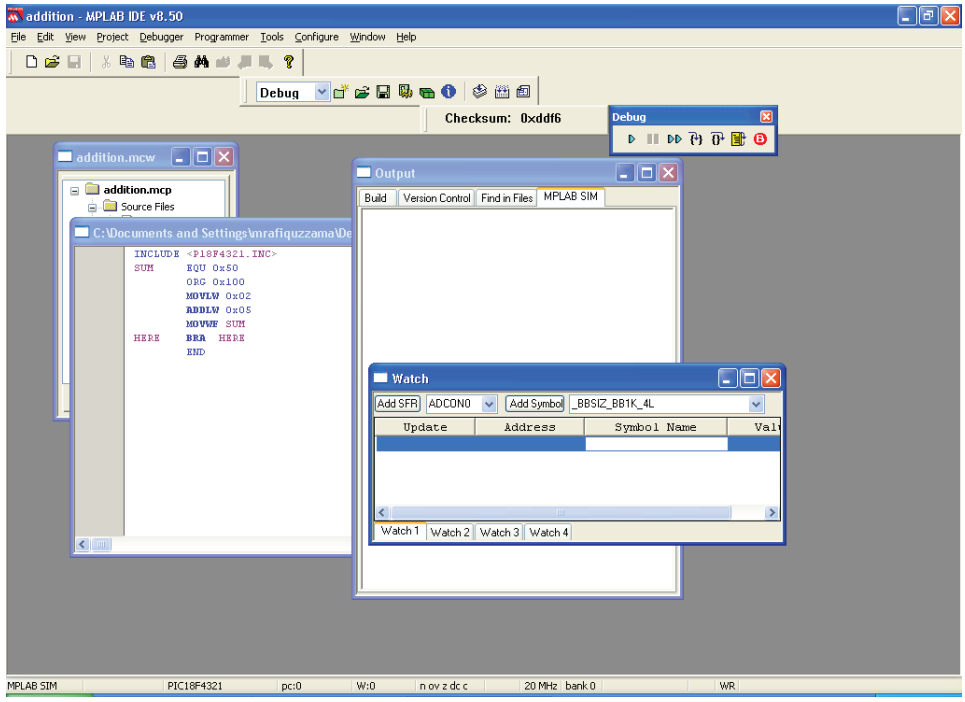
Click on MPLAB SIM to see the following:



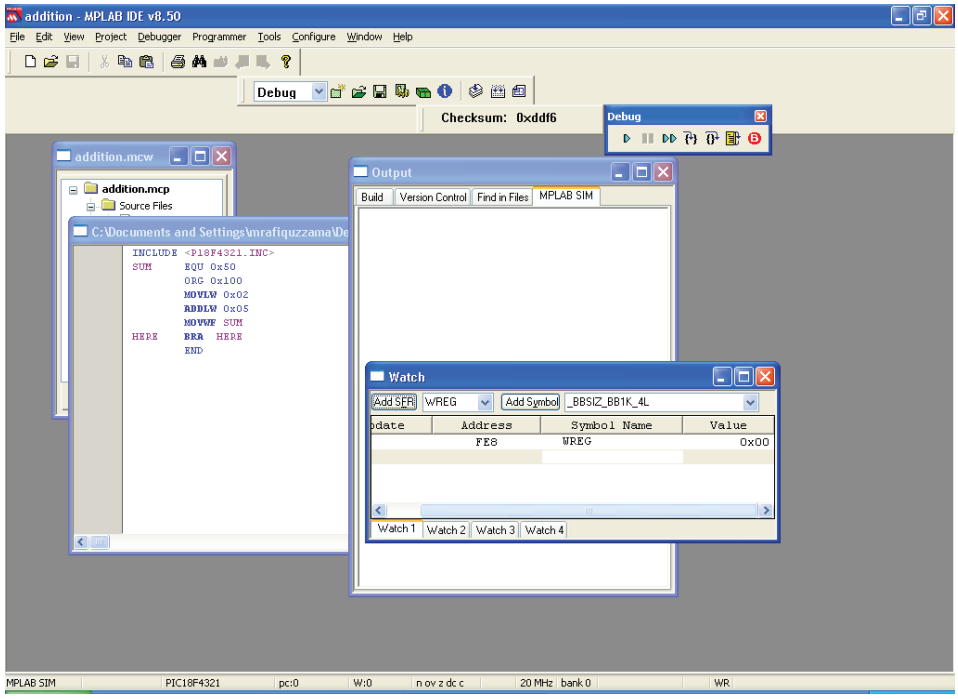
Click on View, toolbars, and Debug to see the following display with Debug toolbar:



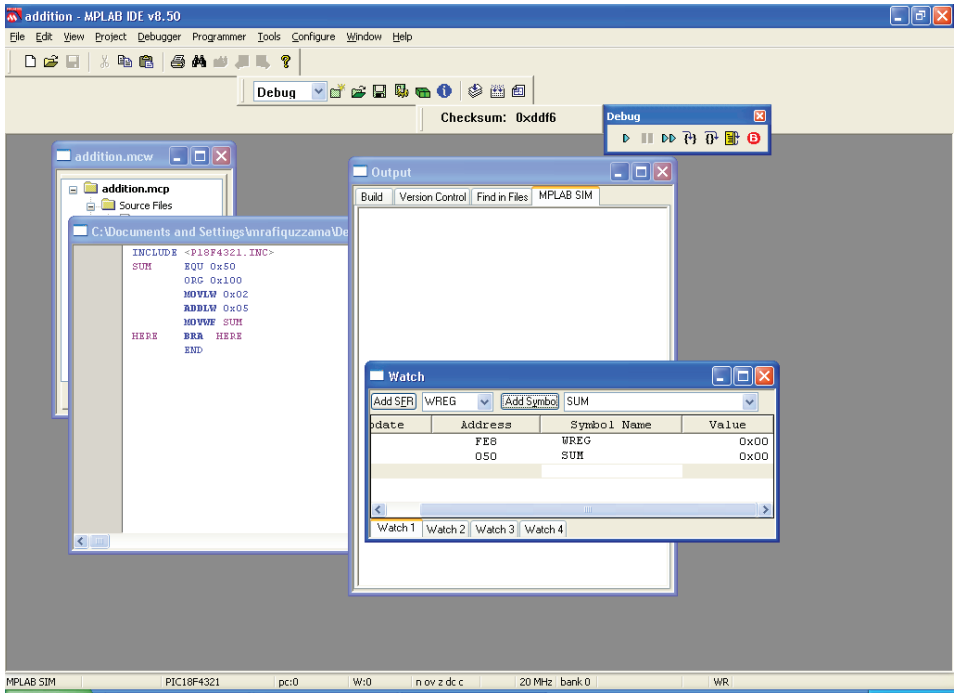
Next, click on View, and then watch to see the following:



On the Watch list, you can now include WREG and SUM to monitor their contents. For example, to add WREG, scroll down to WREG by using the arrow beside ADCON0, and then click on Add SFR to see the following display:

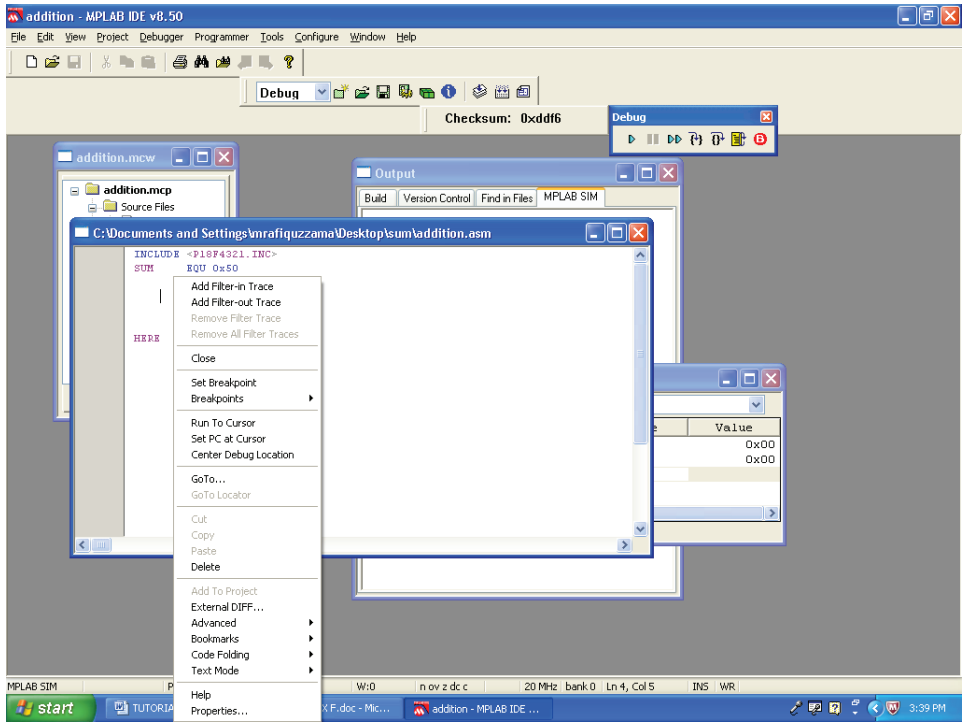


Next, scroll down using arrow beside Add Symbol, select SUM, and then click on Add Symbol to see the following display:

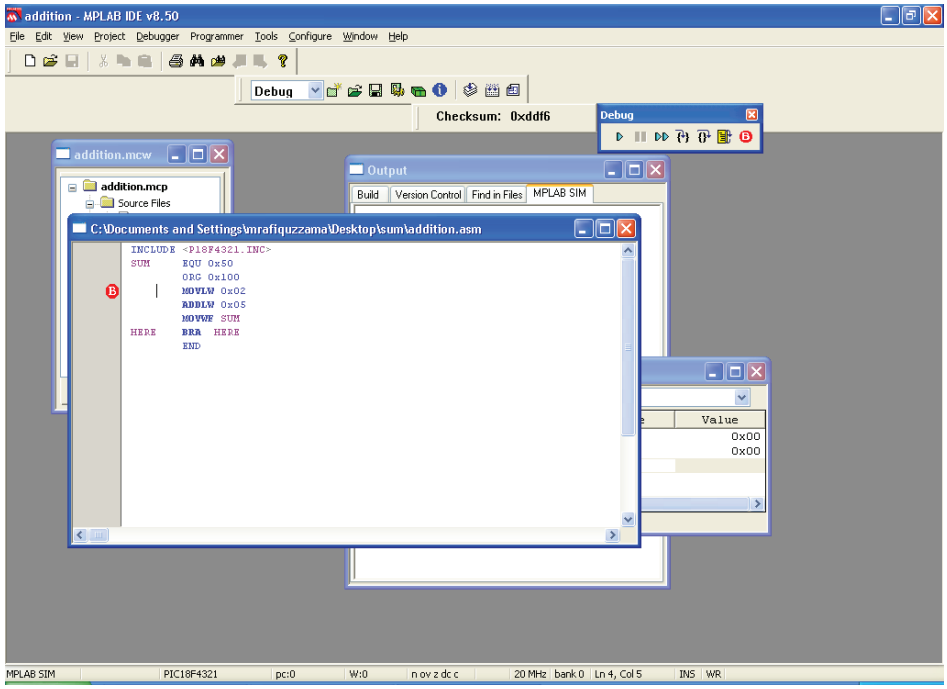


Note that SUM with address 0x50 along with contents is displayed.

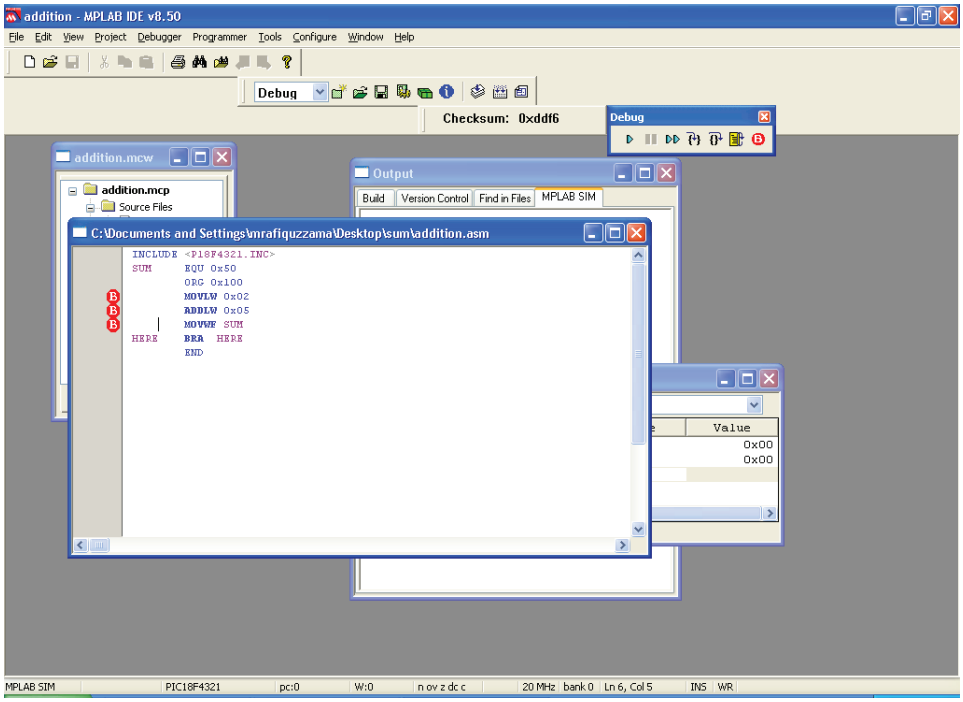
In order to enter breakpoint for MOVLW 0x02, right click beside MOVLW to see:



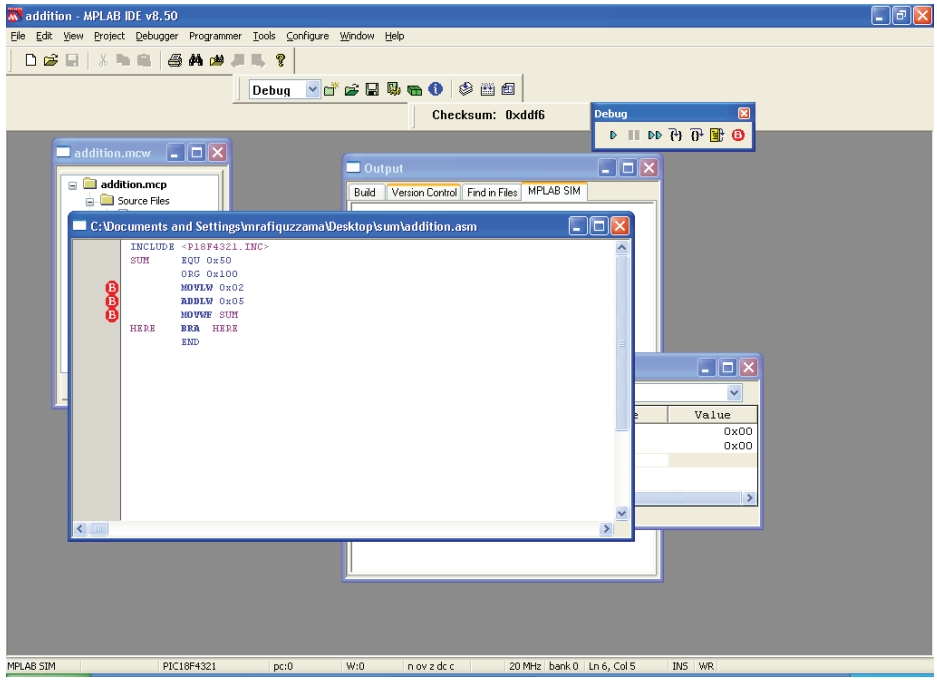
Click on set breakpoint to see the following:



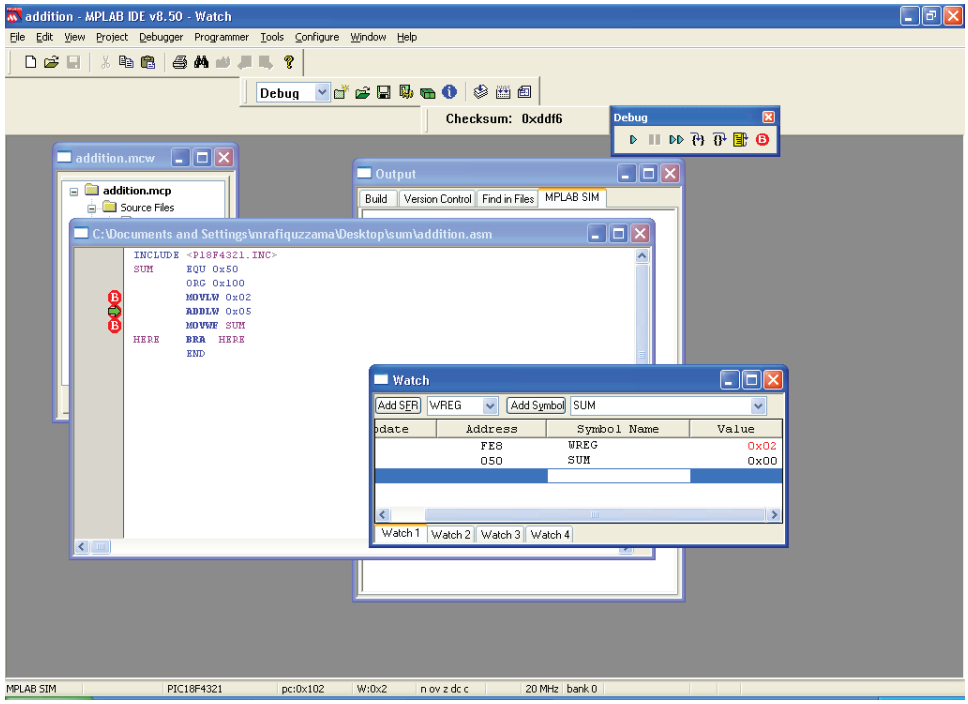
B in red means breakpoint is inserted for MOVLW0x02. Similarly, breakpoints for ADDLW 0x05 and MOVWF SUM can be inserted, and the following will be displayed:



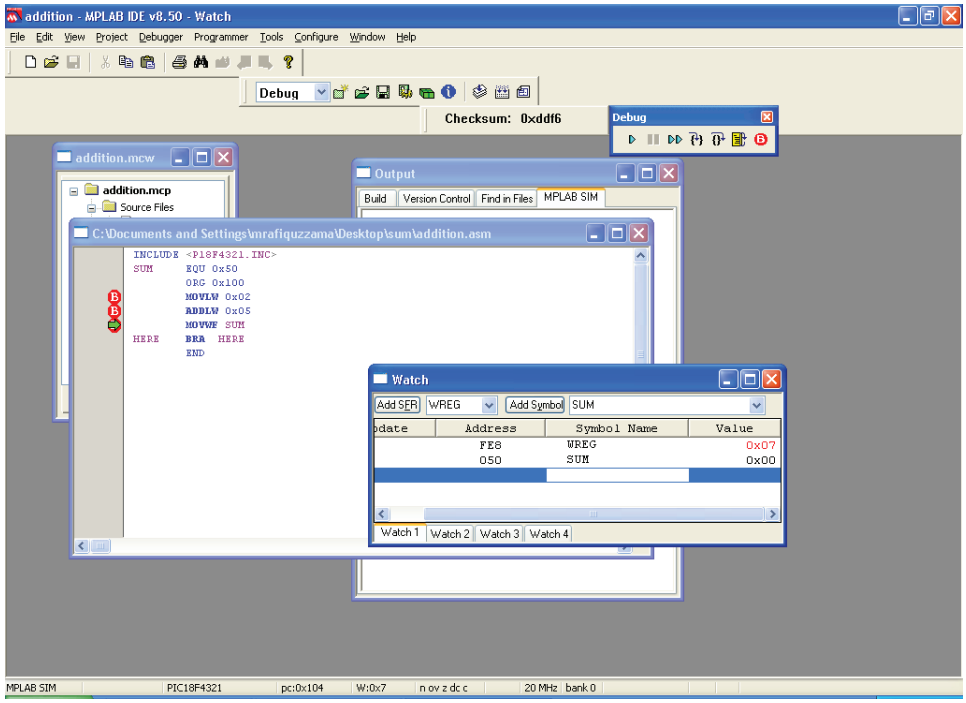
Next, locate the Debug Toolbar. If, for some reason, Debug toolbar is missing, go to view, select Toolbars, click on Debug, and see the following:



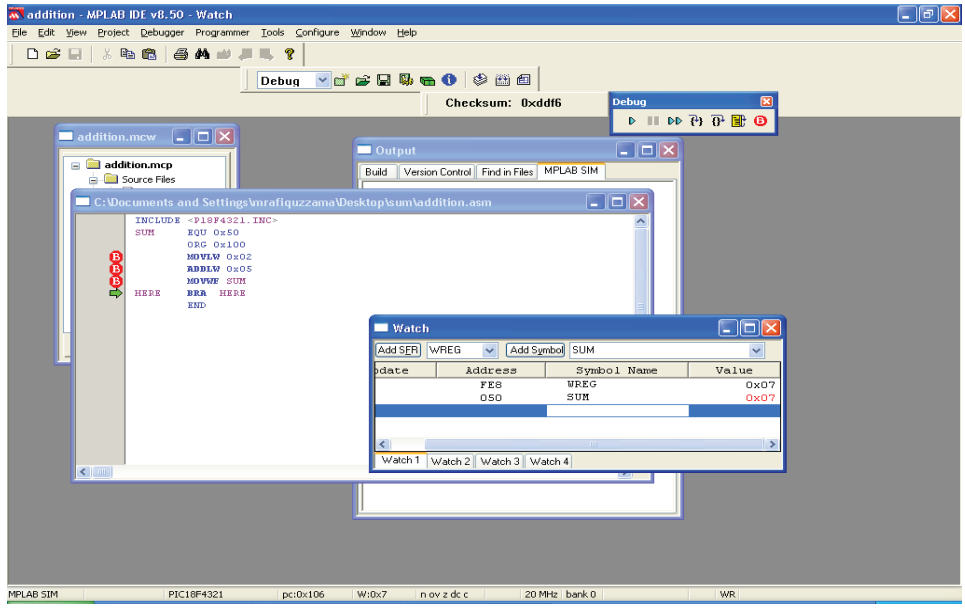
Click on the reset (first icon from right on the Debug toolbar), move the cursor to left of MOVLW 0x02, click on run (green arrow on left on the Debug toolbar) on Debug toolbar, and see that WREG is loaded with 0x02:



Next, click on run to execute ADDLW 0x05, and see that the result of addition0x07 is loaded into WREG as follows:



Finally, click on Animate (double green arrow on the Debug tool) to execute MOVWF SUM to see that the result 0x07 is stored in SUM (address 0x50) as follows:



The debugging is now complete.

APPENDIX G: TUTORIAL FOR COMPILING AND DEBUGGING A C-PROGRAM USING THE MPLAB

The C-programs in this book are compiled using Microchip's C18 compiler. Appendix G provides a tutorial showing step-by-step procedure to download the C18 compiler from Microchip's web site (www.microchip.com), and compile a simple C-program using the MPLAB C18 compiler v3.47. Microchip introduces newer versions of the C18 compiler frequently. One may download the C18 v3.47 from the archive or a newer version from Microchip website.

Compiling a C- language program using MPLAB

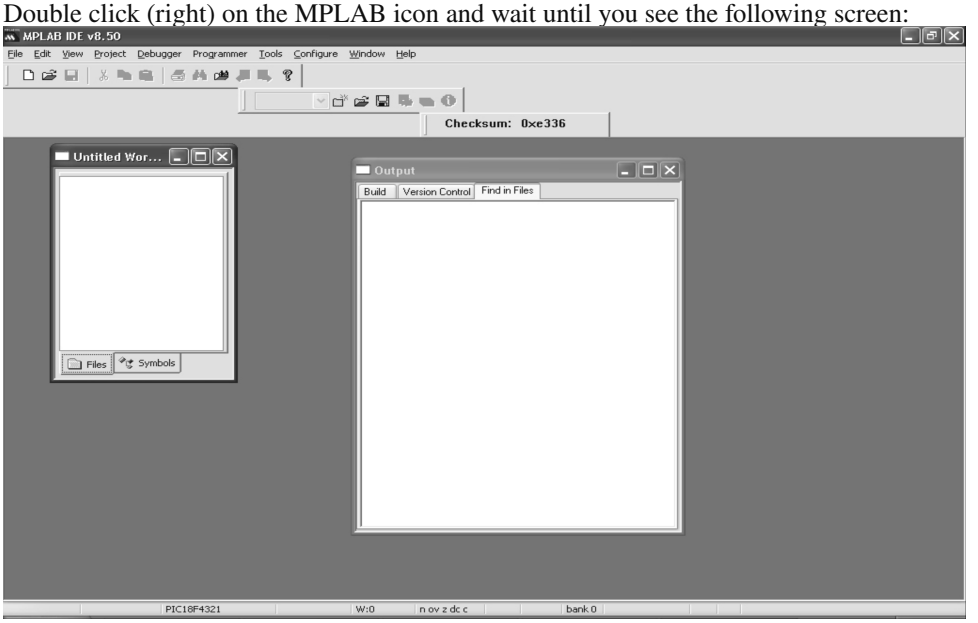
First download MPLAB v3.47 from Microchip website as follows:

1. Go to www.microchip.com.
2. Type in C18 v3.47 in the "Search" and hit return.
3. Click on MPLAB C for PIC18 v3.47 in lite mode.

After installing and downloading the program, you will see the following icon on your desktop:

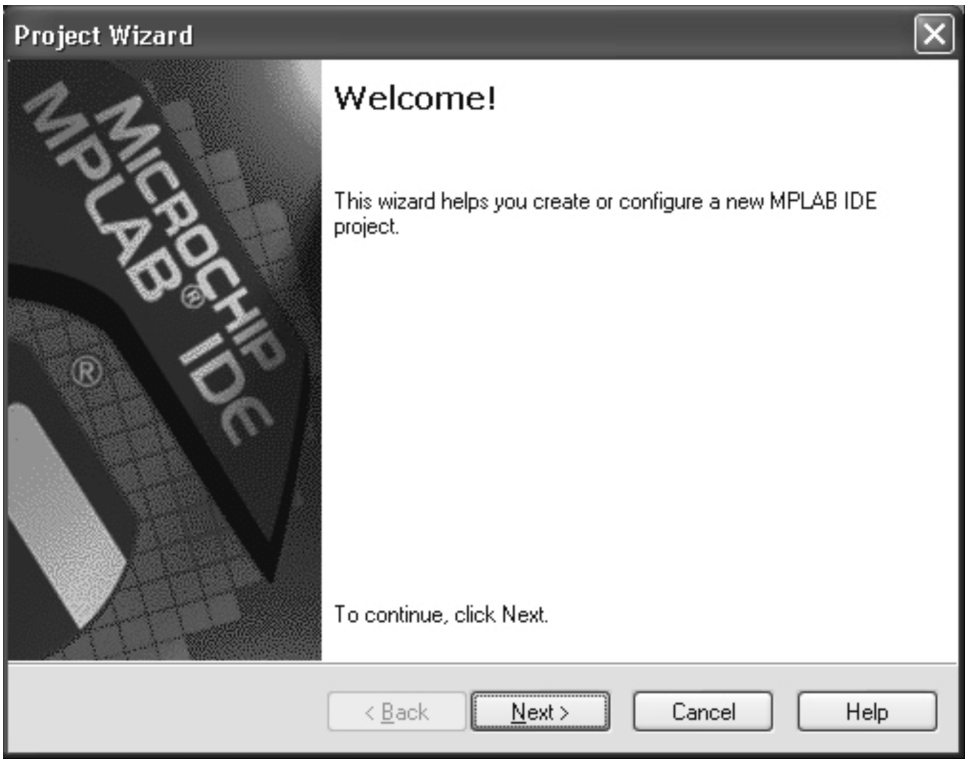


Double click (right) on the MPLAB icon and wait until you see the following screen:



© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

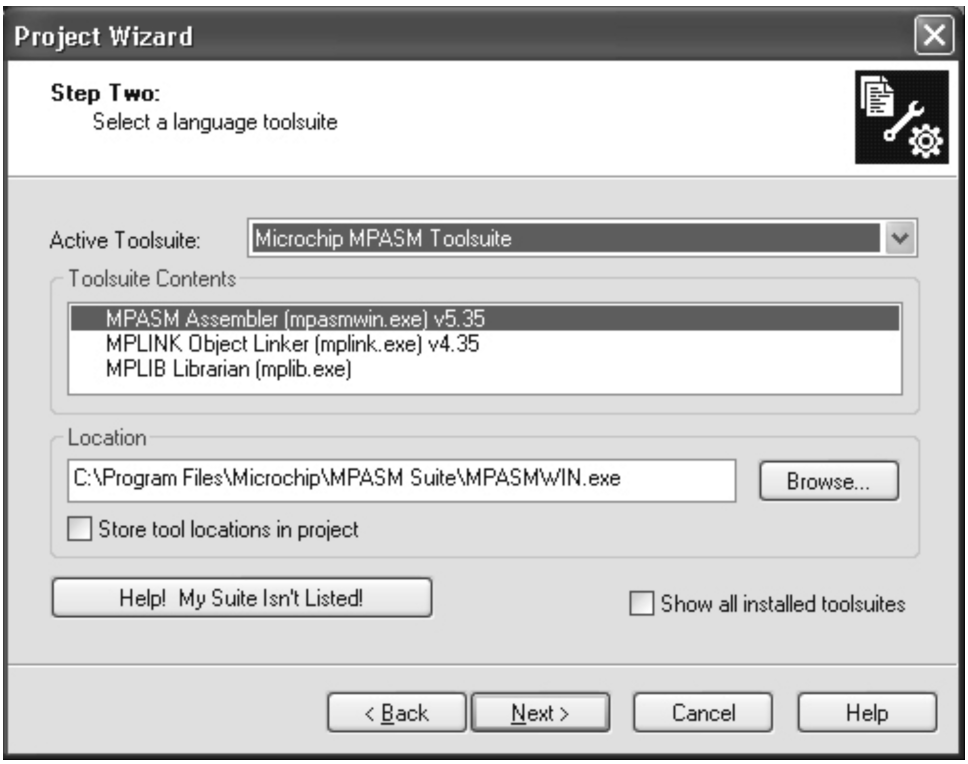
Next, click on ‘Project’ and then ‘Project Wizard’, the following screen will appear:



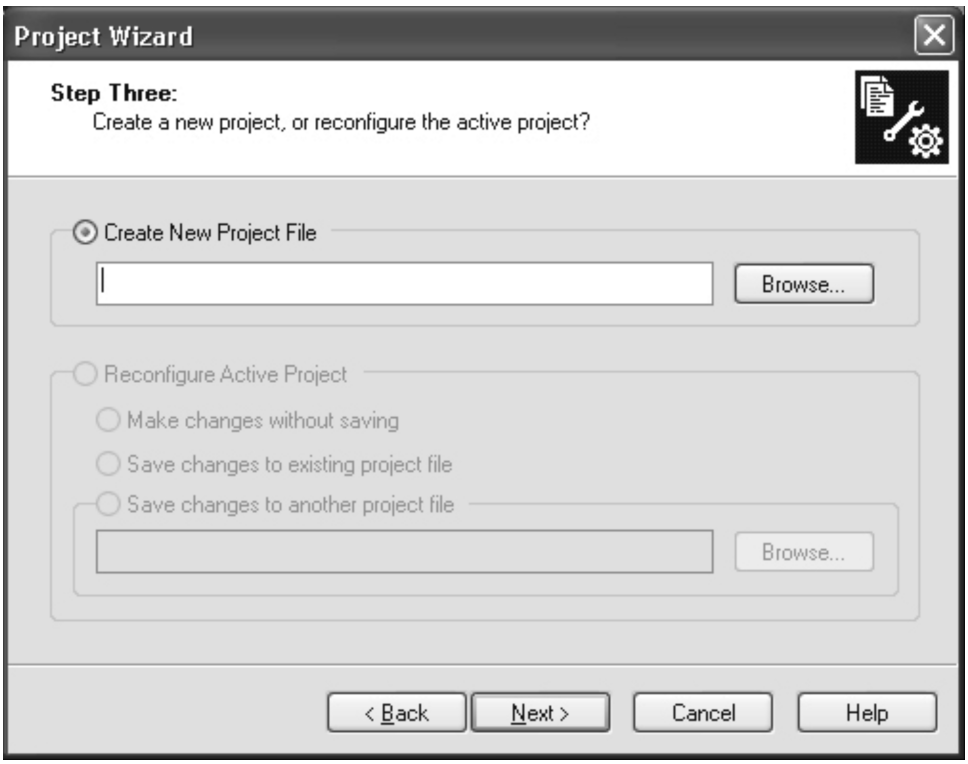
Click Next, the following screen shot will be displayed:



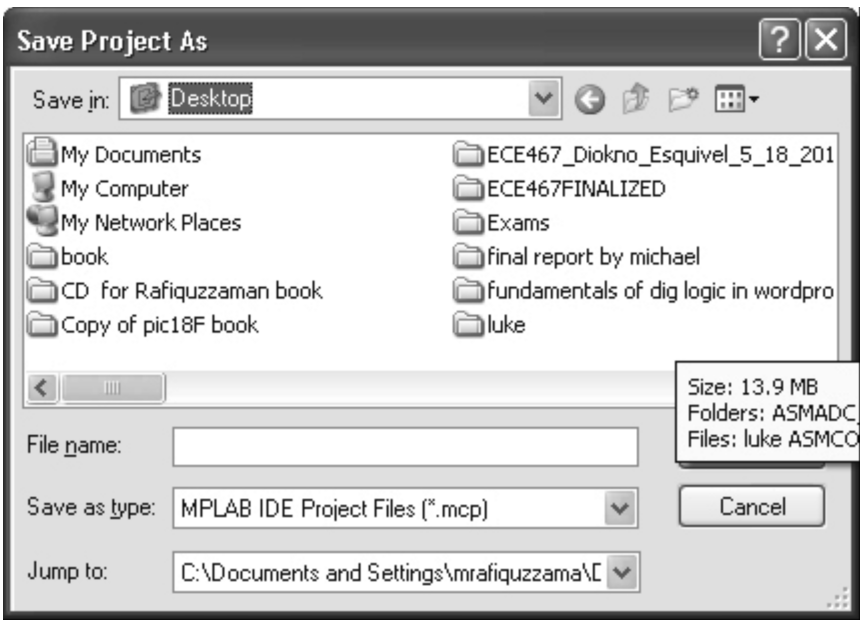
Select the device PIC18F4321, hit Next, and wait, the following will be displayed:



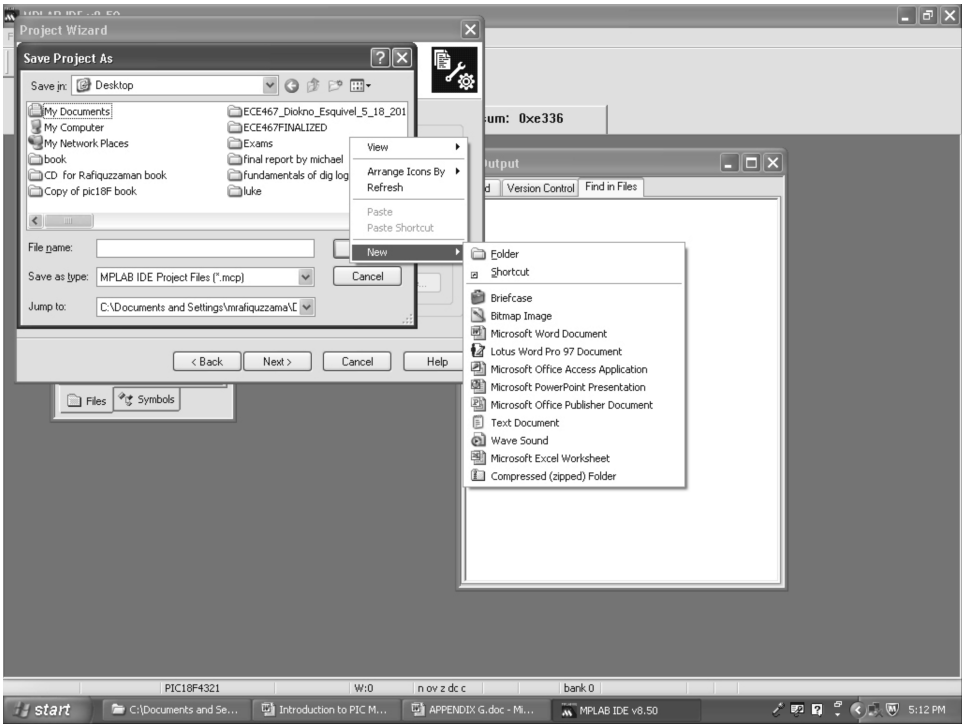
In the 'Active Toolsuite', select 'Microchip C18 Toolsuite', and click Next, the following will be displayed:



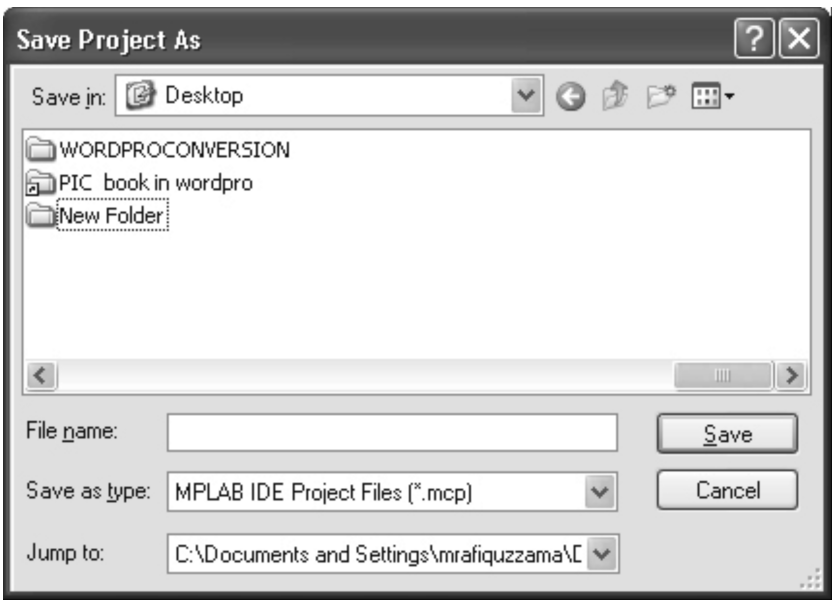
Select a location where all project contents will be placed. For this example, the folder will be placed on the desktop (arbitrarily chosen). Go to the desktop directory, make a new folder, and name the folder. In order to do this, Click on 'Browse', select desktop:



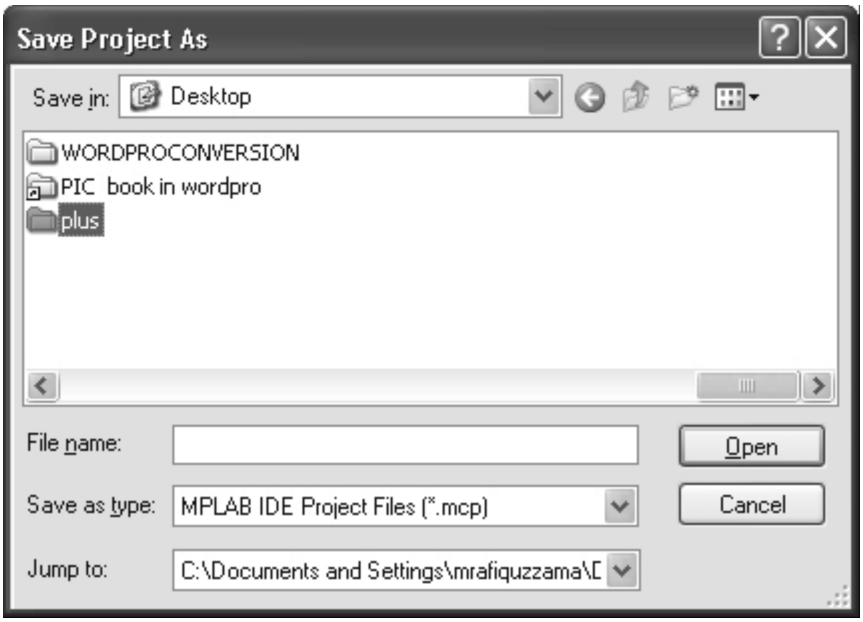
Next, create a new folder by clicking on the icon (second yellow icon from right on top row) or by right clicking on the mouse on the above window, and then go to New to see the following screen:



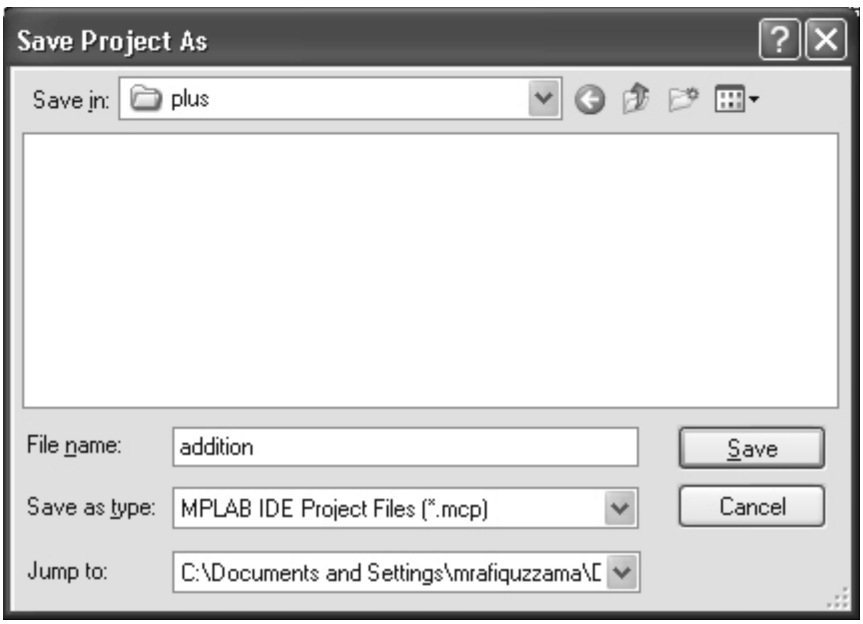
Click on Folder to see the following:



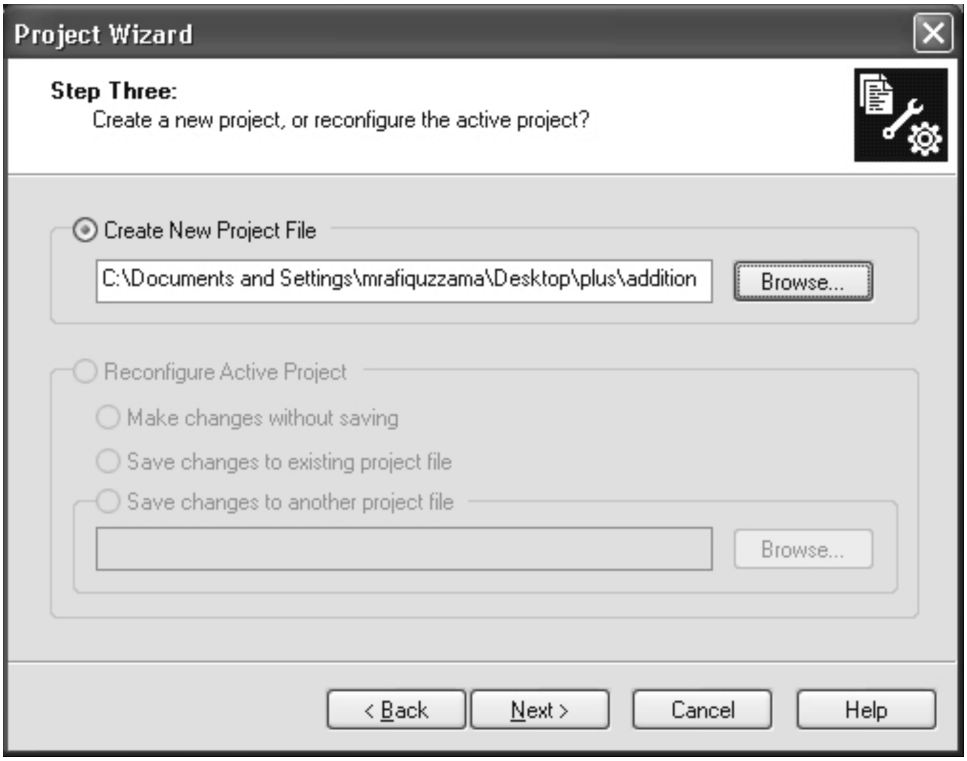
Click on folder, name it ‘plus’ (arbitrarily chosen name) and see the following:



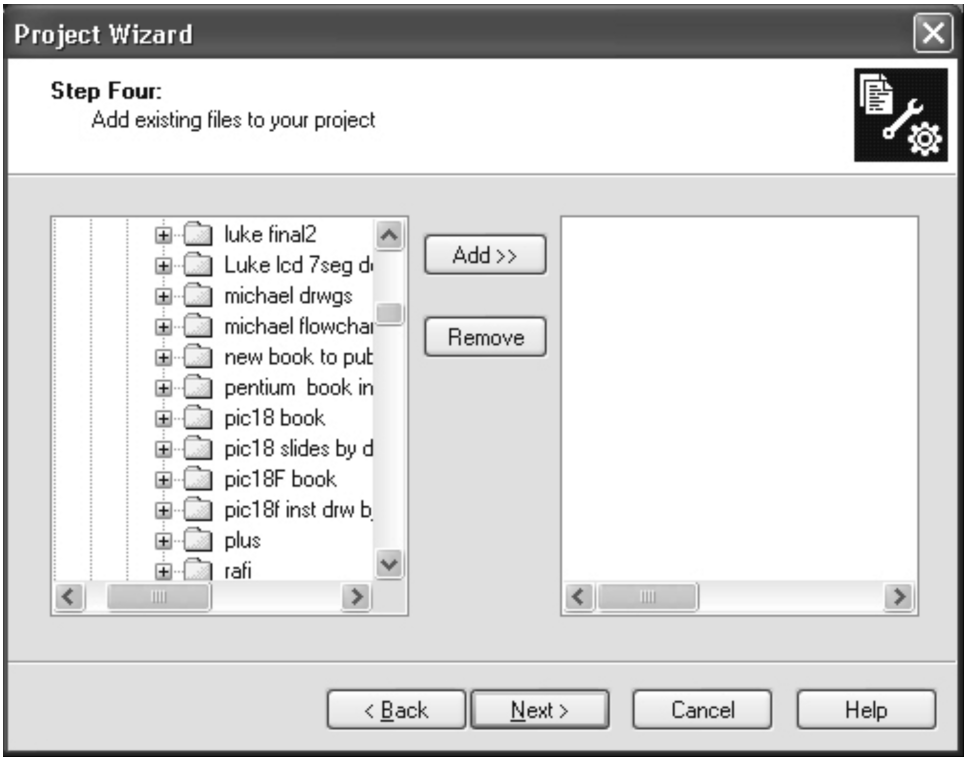
File name ‘addition’ is arbitrarily chosen . Type in the File name to see the following:



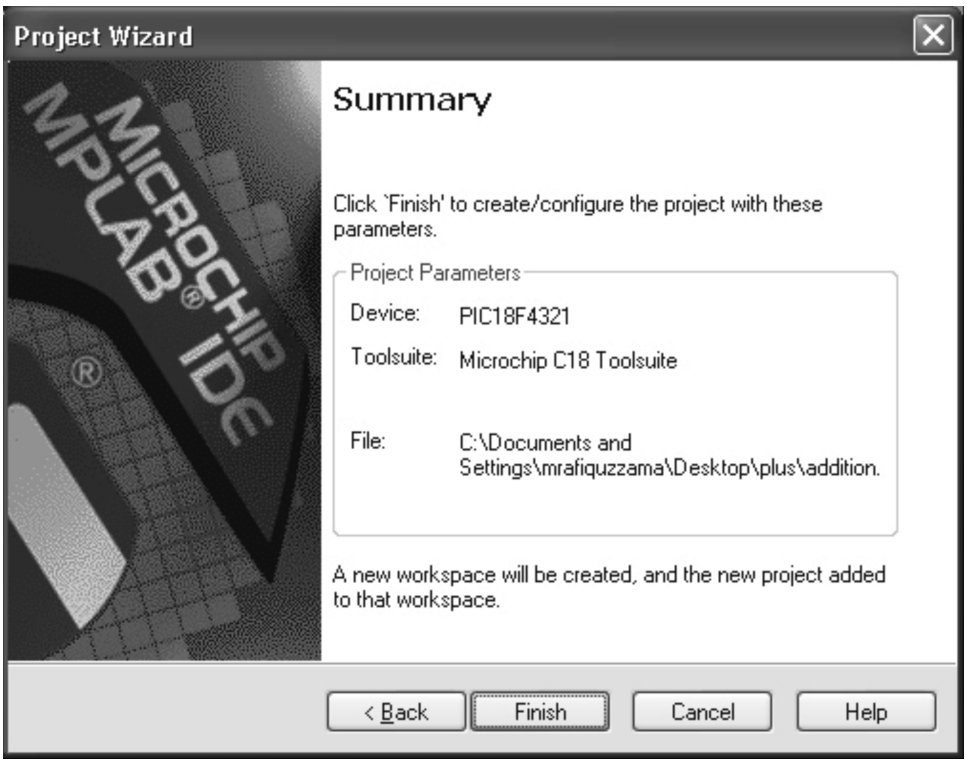
Next, click on Save, the following screen will appear:



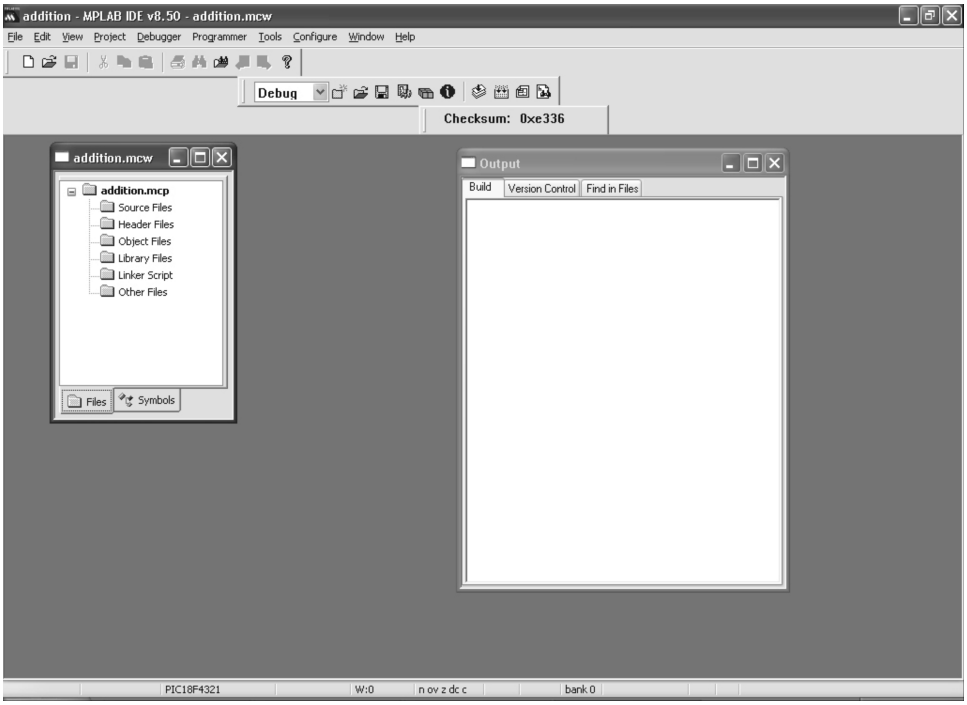
Click on Next, and see the following:



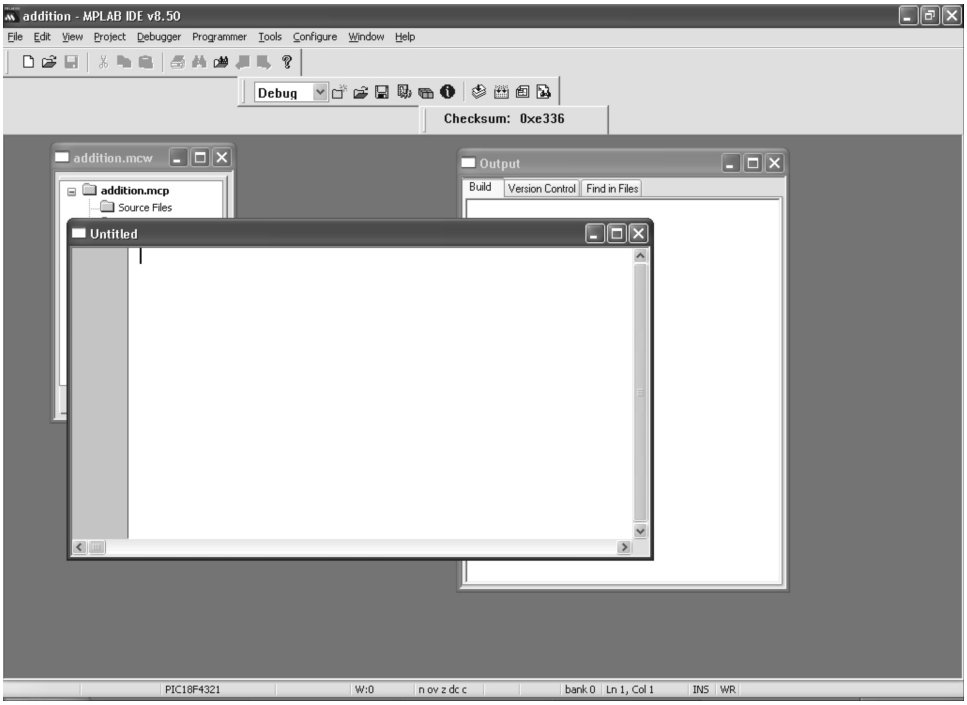
Click Next to see the following:



Click on Finish, and see the following:



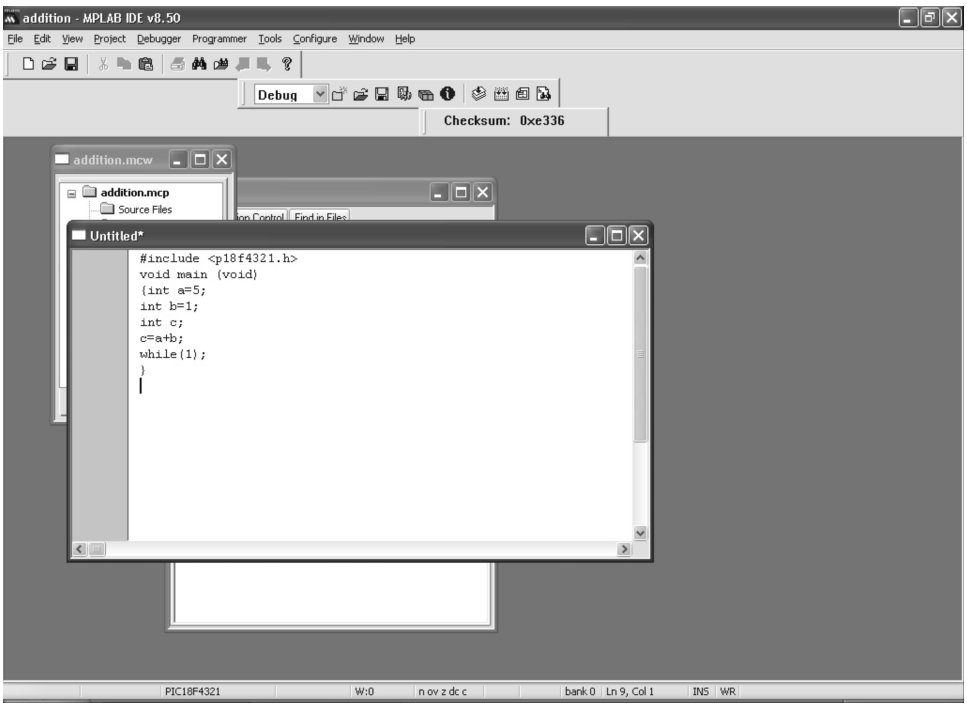
Click on File, and then New to see the following:



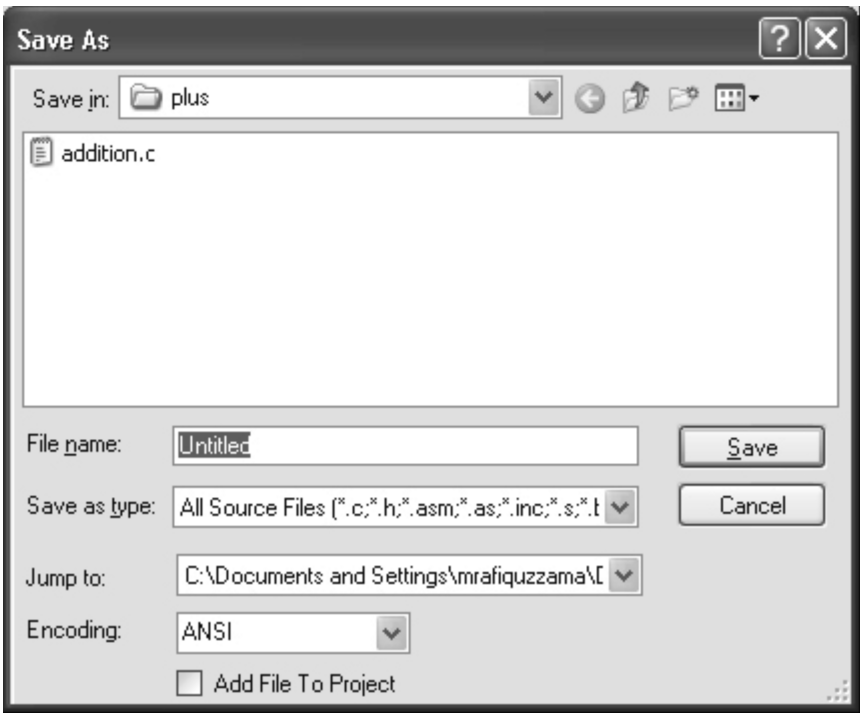
Type in the program you want to compile. The following addition program is entered:

```
#include <p18f4321.h>
void main (void)
{int a=5;
int b=1;
int c;
c=a+b;
while(1);
}
```

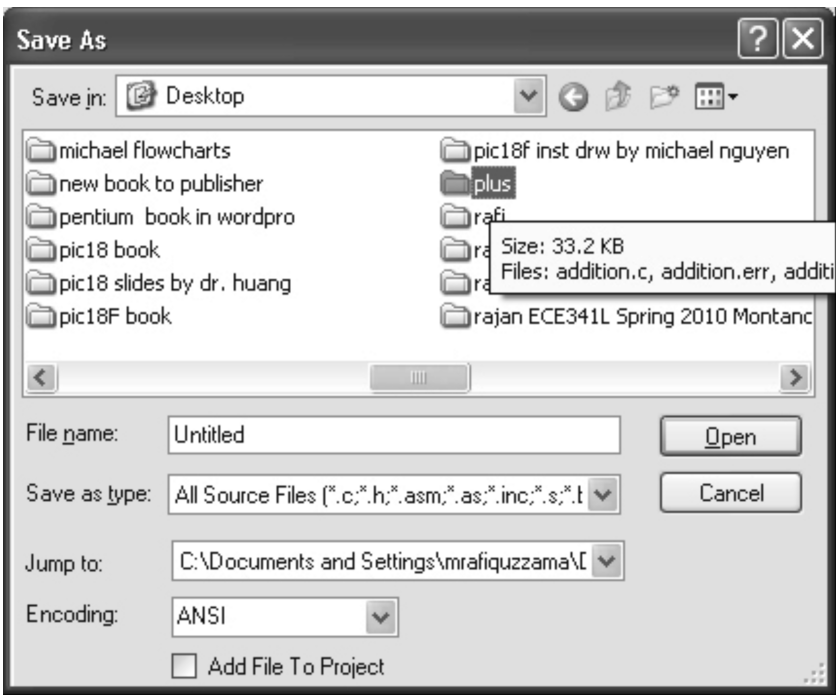
After entering the program, see the following:



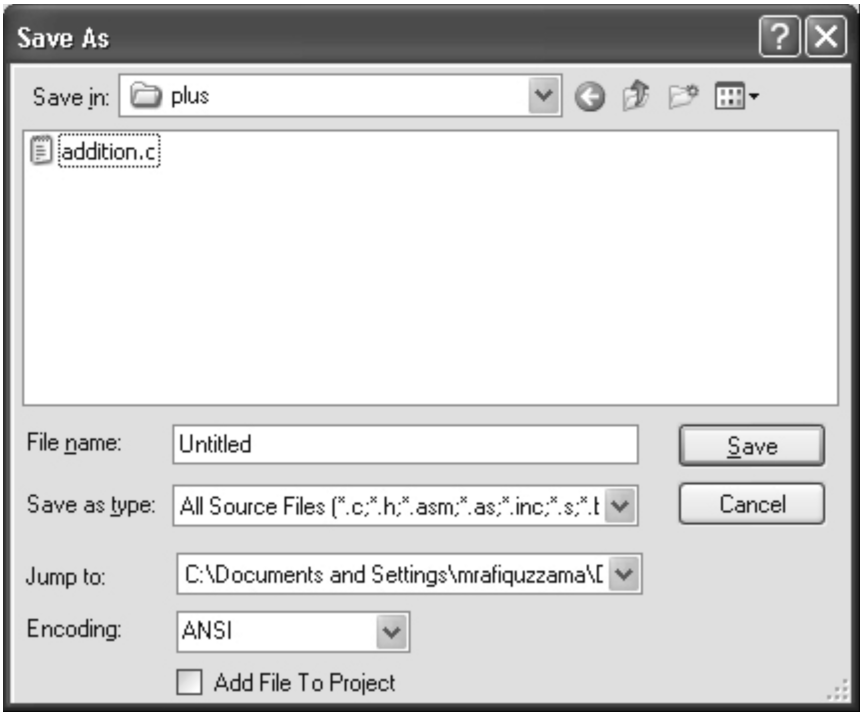
Next, click on File, and then Save as to see the following screen shot:



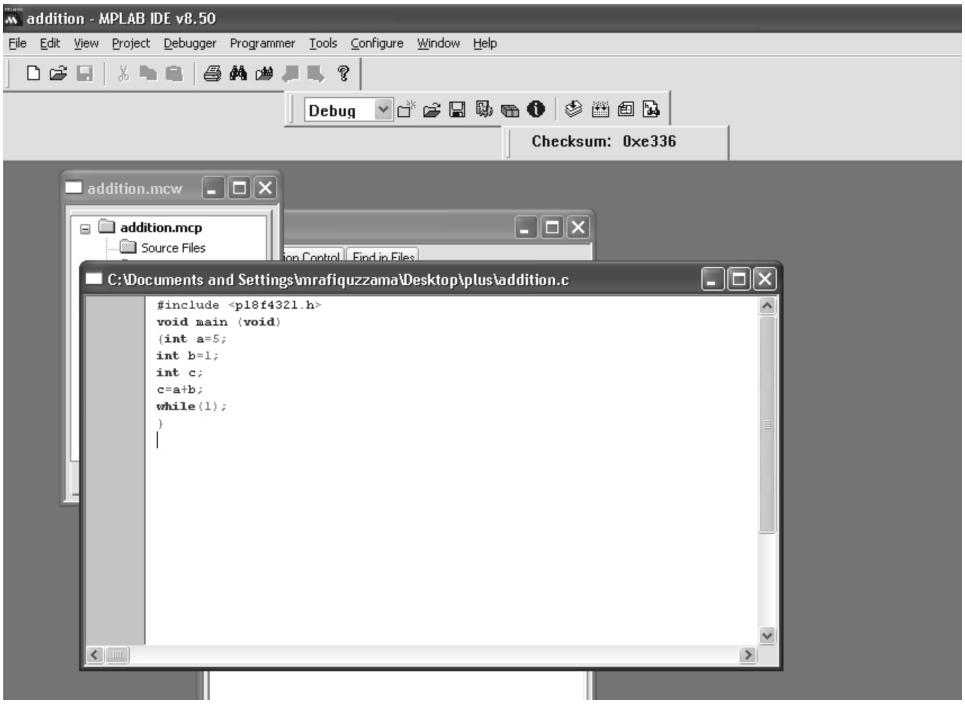
Make sure you scroll up to desktop, and then click on plus (the folder which was created before), and see the following:



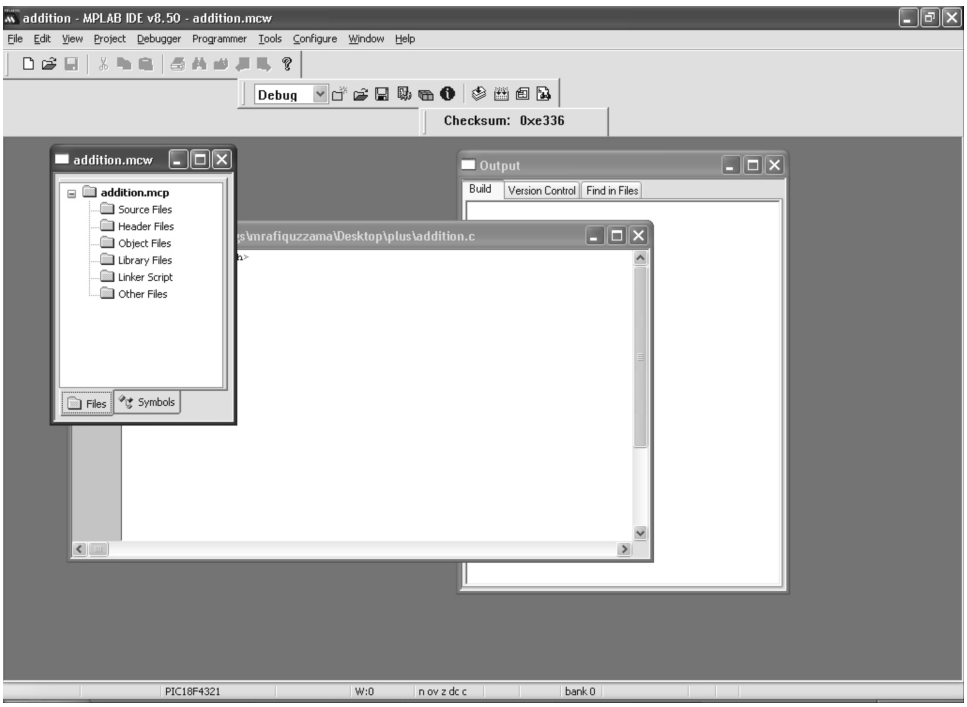
Next, double click (left) on plus to see the following:



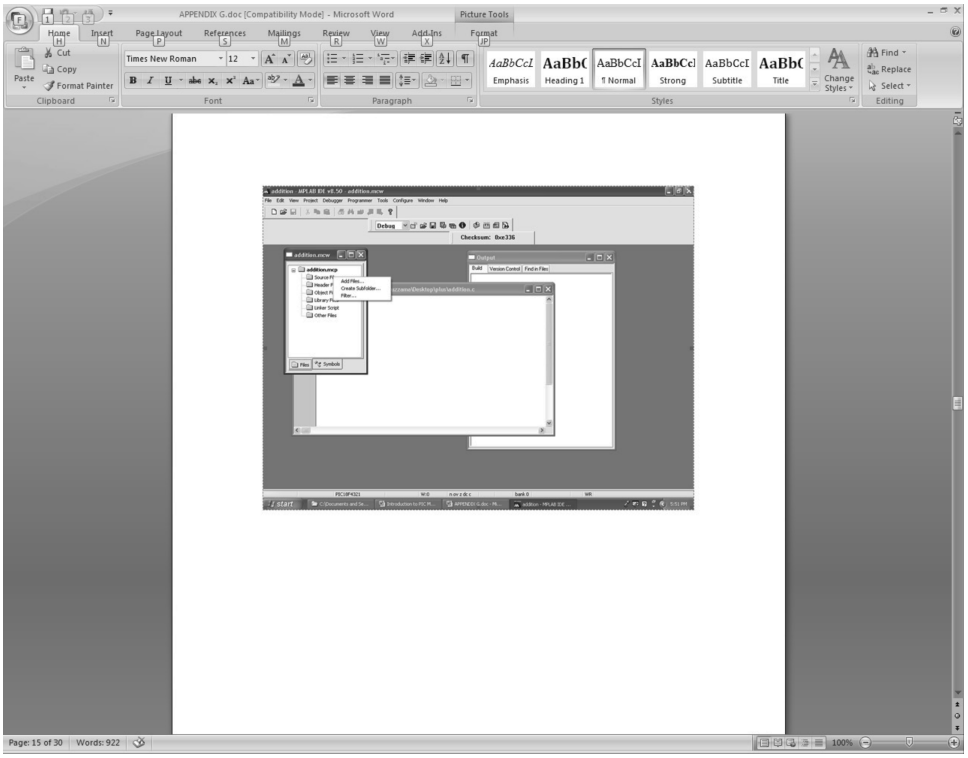
Delete Untitled, enter the same file name ‘addition’ with .c extension as File name. Click on save, and see the following screen shot (notice the display changes color):



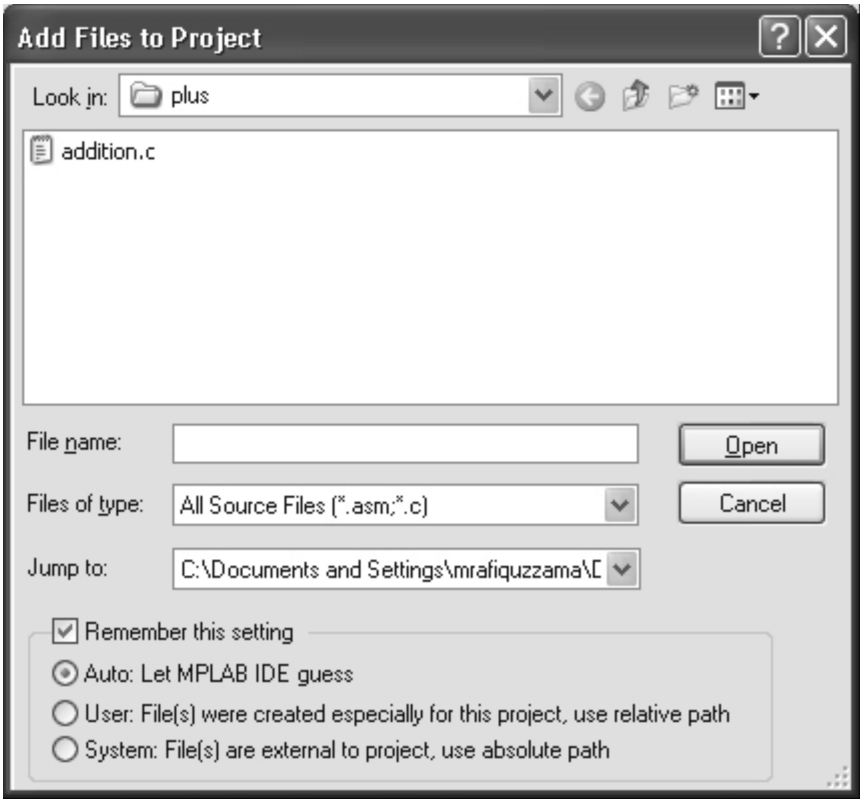
Next highlight by clicking on the top (blue) section of addition.mcw, and see the following:



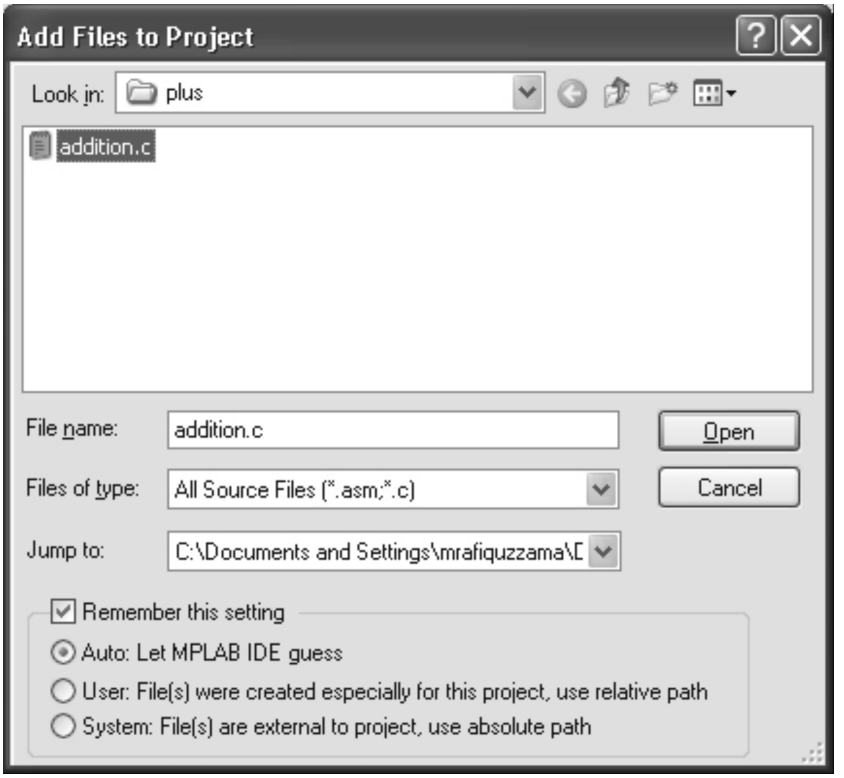
Right click on Source Files to see the following:



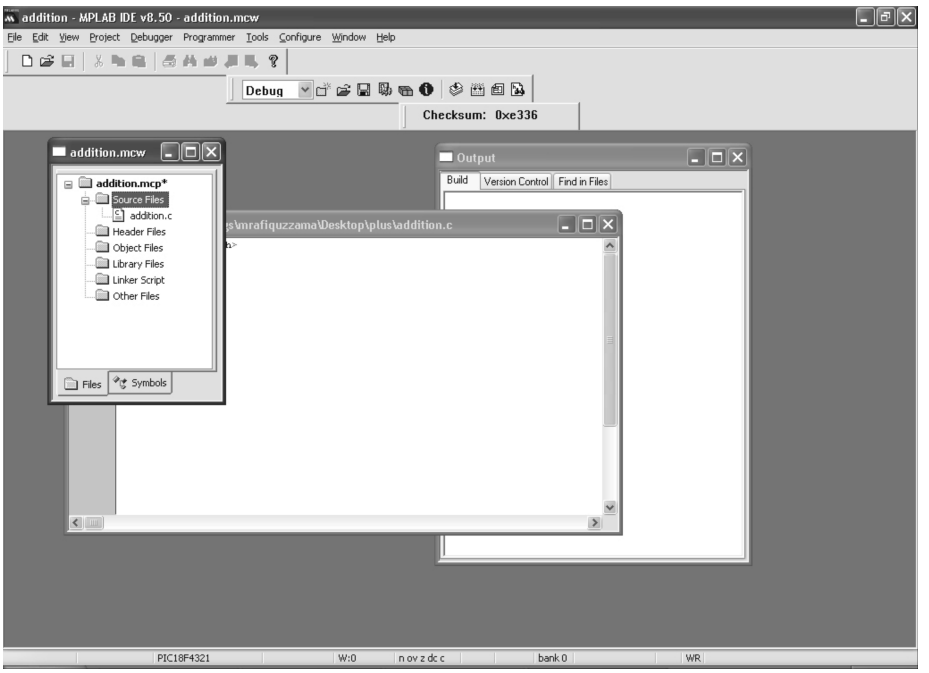
Click (left) on Add files to see the following:



Click once (left) on addition.c on the window to see the following:



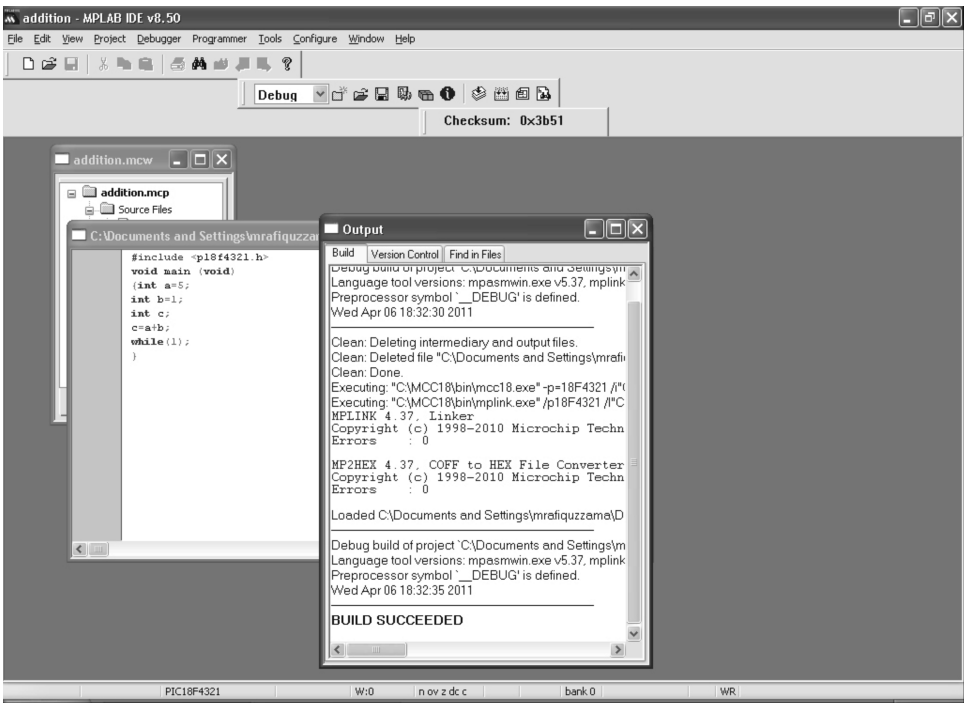
Click Open to see the following:



Next, do the following:

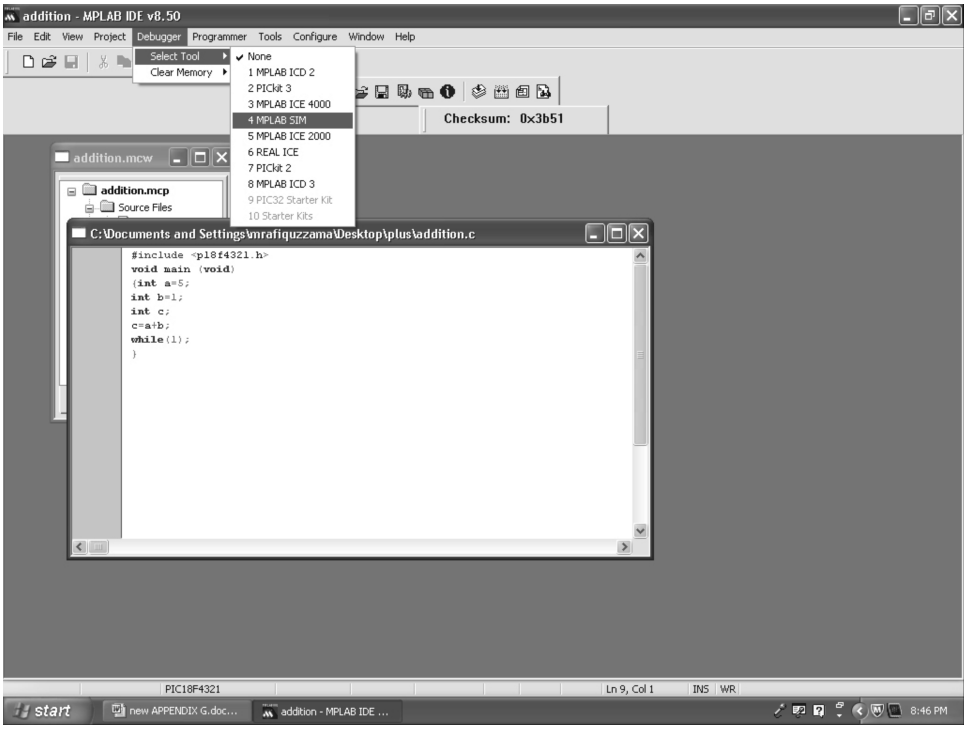
- Click on project, Click on Build options and then project
- Scroll down on output directory to Linker.Script Search Path, Select new
- Click on … (three dots on the extreme right)
- Select C:\MCC18, bin, and then LKR, Click OK
- Click on project, Click on Build options and then project
- Scroll down on output directory to Library Search Path, Select new
- Click on … (three dots on the extreme right)
- Select C:\MCC18, and then lib, Click OK
- Click on project, Click on Build options and then project
- Scroll down on output directory to Include Search Path, Select new
- Click on … (three dots on the extreme right)
- Select C:\MCC18, and then h, Click OK

Note that addition.c is listed under Source Files. Next, click on Project and then build all (or only the 'Build All' icon, third icon on top right of the Debug toolbar), and see the following:

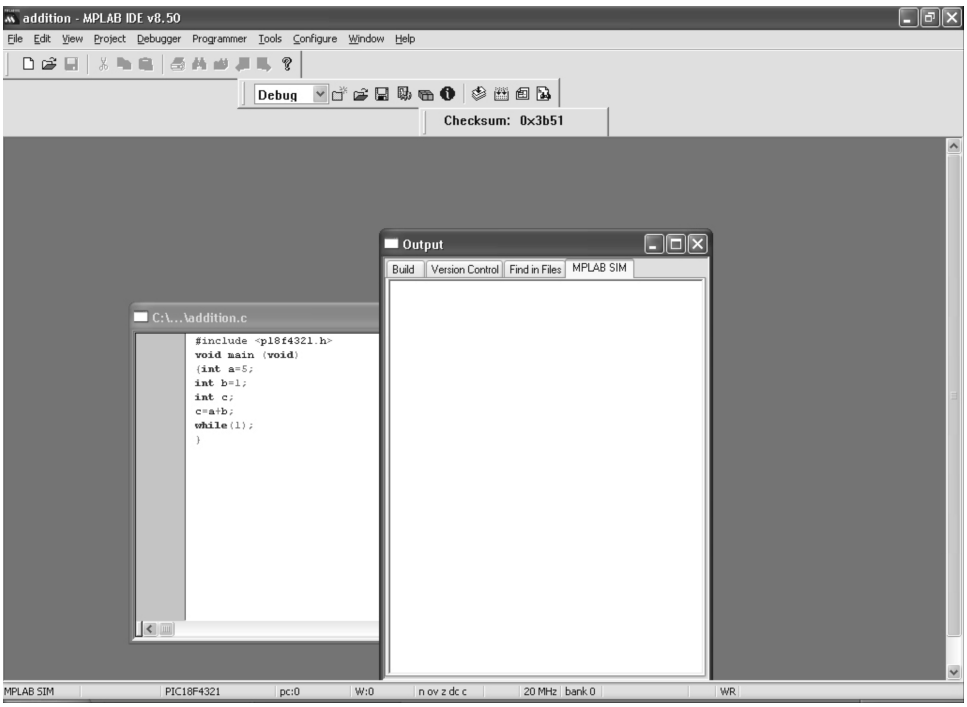


This means that the compiling the C program is successful. Next the result will be verified using the debugger.

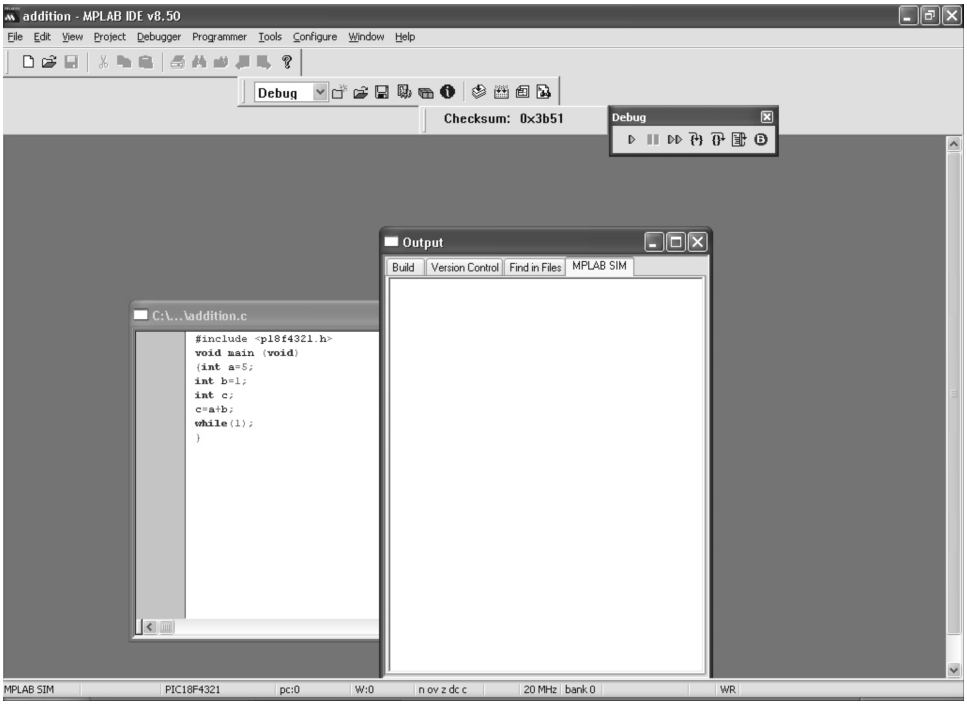
Click on Debugger, Select Tool, and then MPLAB SIM to see the following display:



Click on MPLAB SIM to see the following:

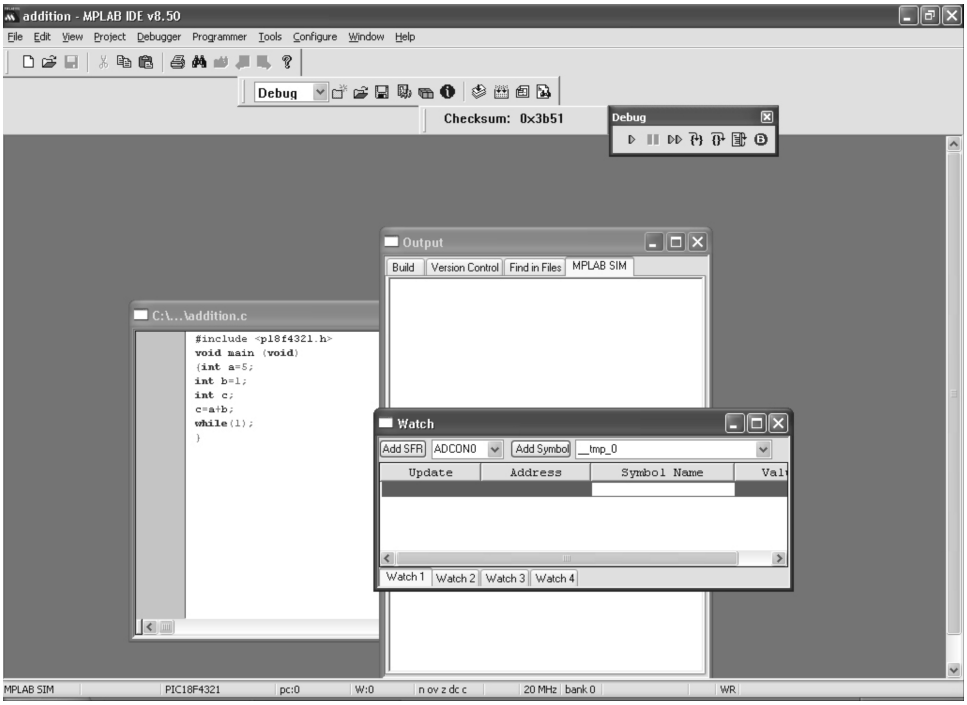


Click on View, toolbars, and Debug to see the following display with Debug toolbar:

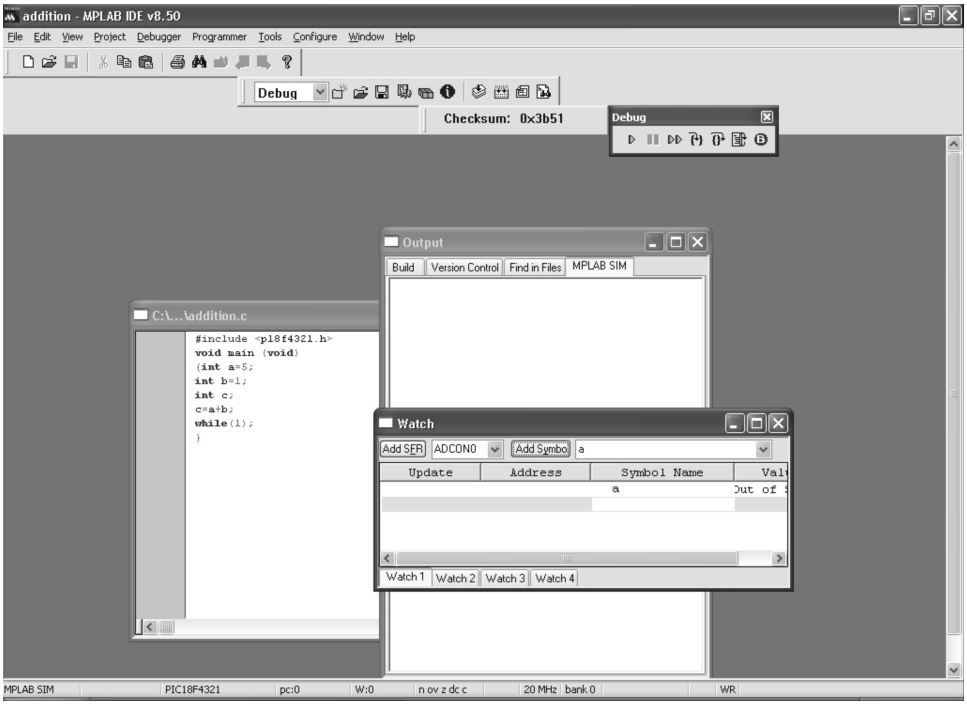


In the above, locate the Debug Toolbar. If, for some reason, Debug toolbar is missing, go to view, select Toolbars, click on Debug.

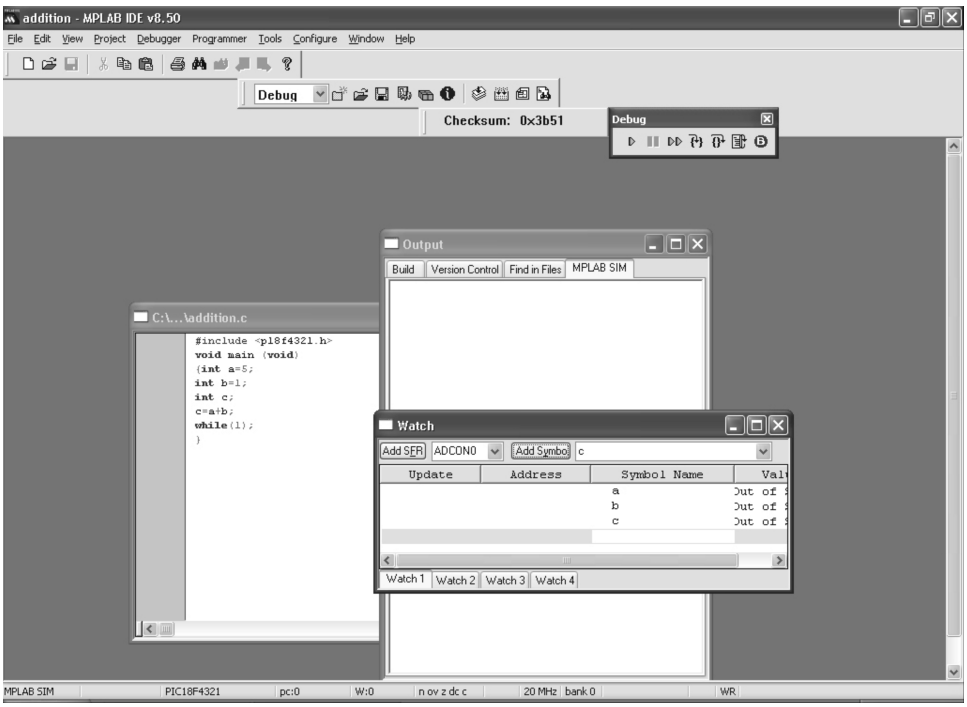
Next, click on View, and then watch to see the following:



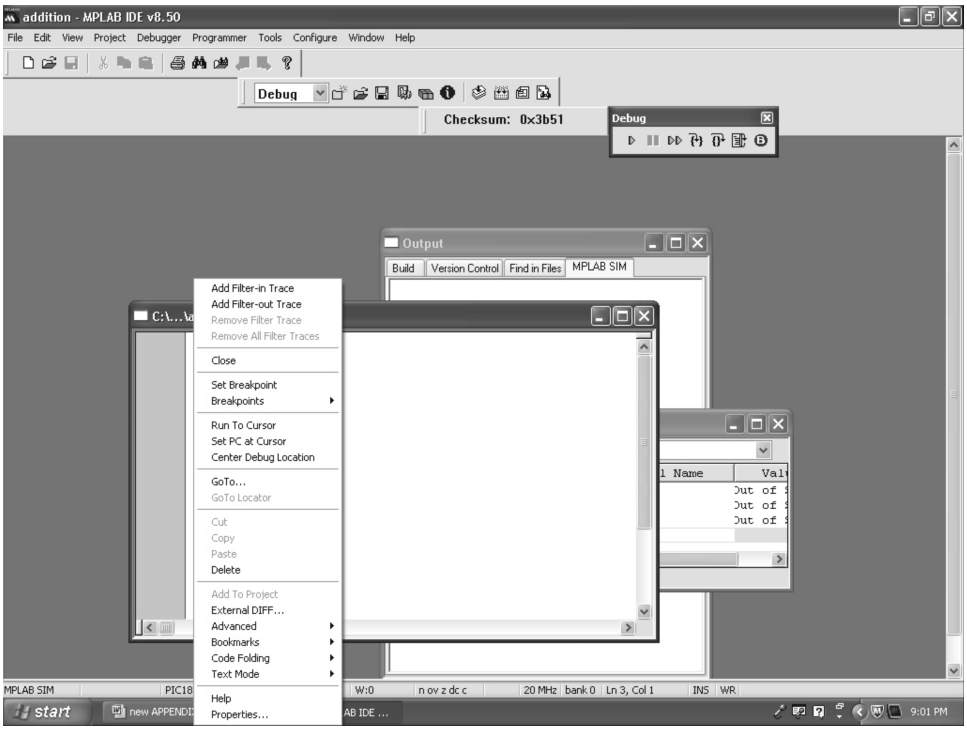
On the Watch list, you can now include locations a, b, c to monitor their contents. For example, to add 'a', simply select 'a' by scroll down using the arrow on the Add Symbol window, and then click on Add Symbol to see the following display:



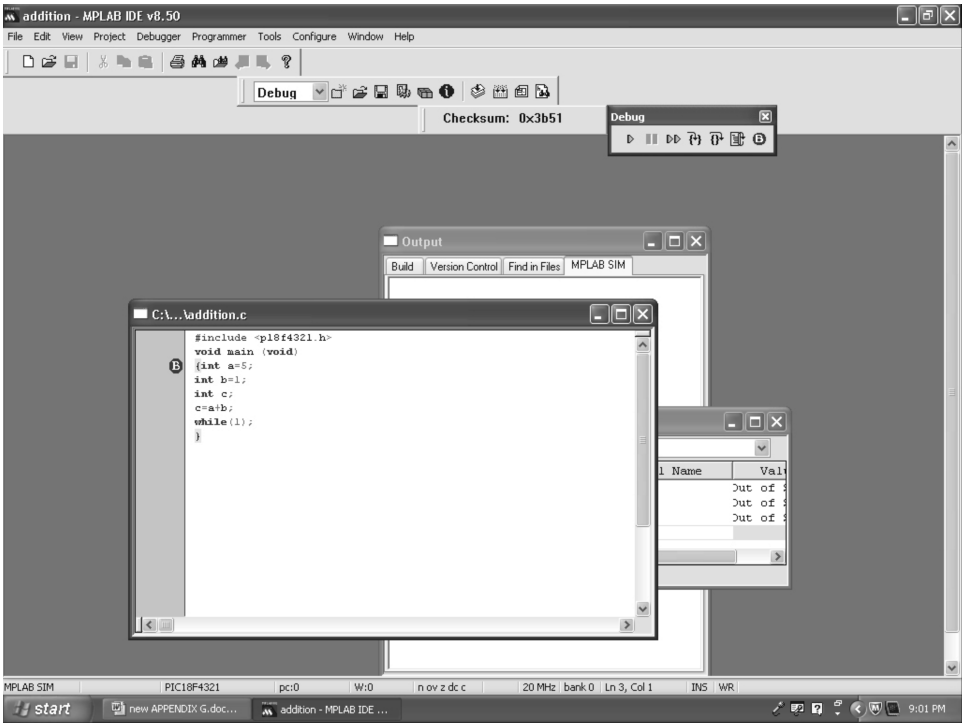
See that ‘a’ is displayed on the watch window. Similarly, display ‘b’ and ‘c’, and see the following screen shot:



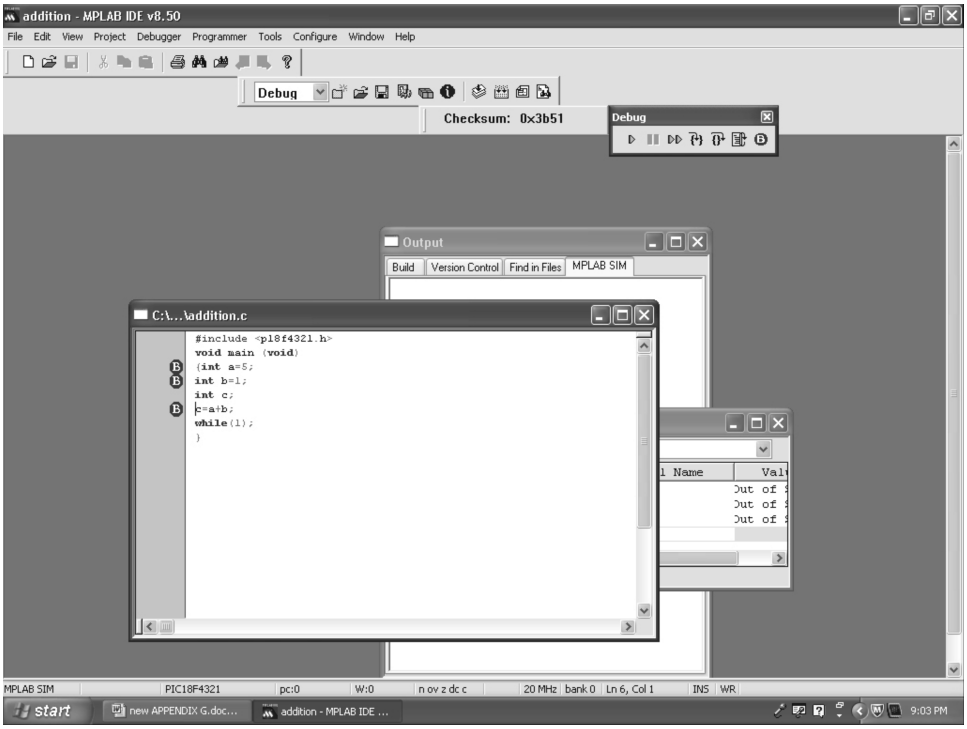
Next, insert breakpoints. Three breakpoints will be inserted for this program. One for int a = 5, one for int b = 2, and one for c = a+b. To insert a breakpoint, move the cursor to the left of the line where breakpoint is to be inserted. For example, to insert a breakpoint at int a =5, move cursor to the left of the line, click (right) and see the following display:



Next, click on Set Breakpoint to see the following:

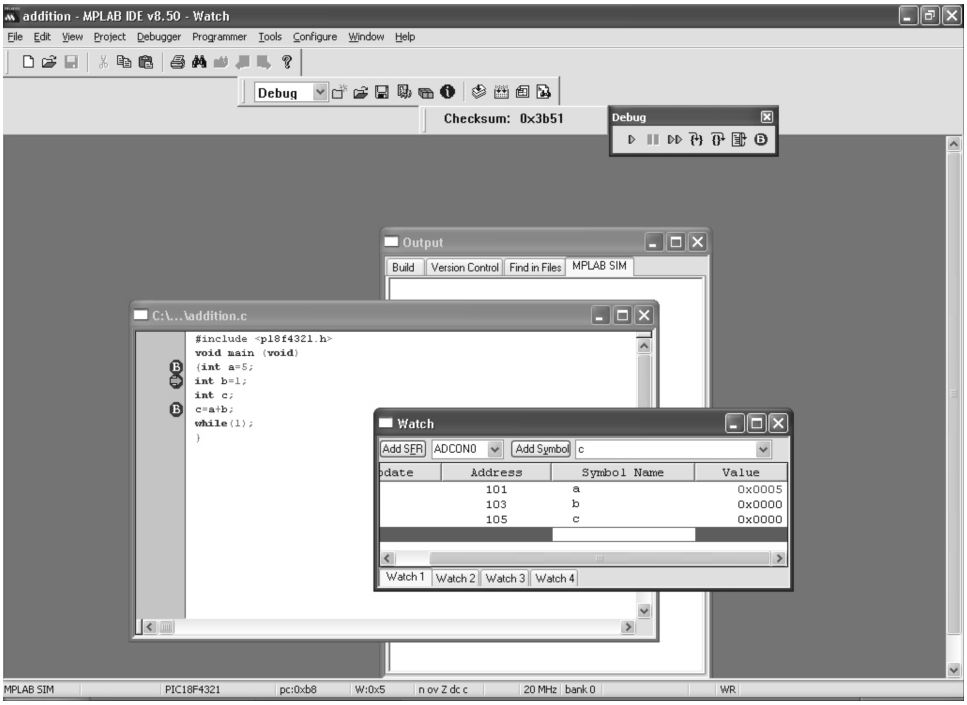


B in red on the left side of the line would indicate that the breakpoint is inserted. Similarly, insert the breakpoints for' 'b and 'c', and obtain the following display:

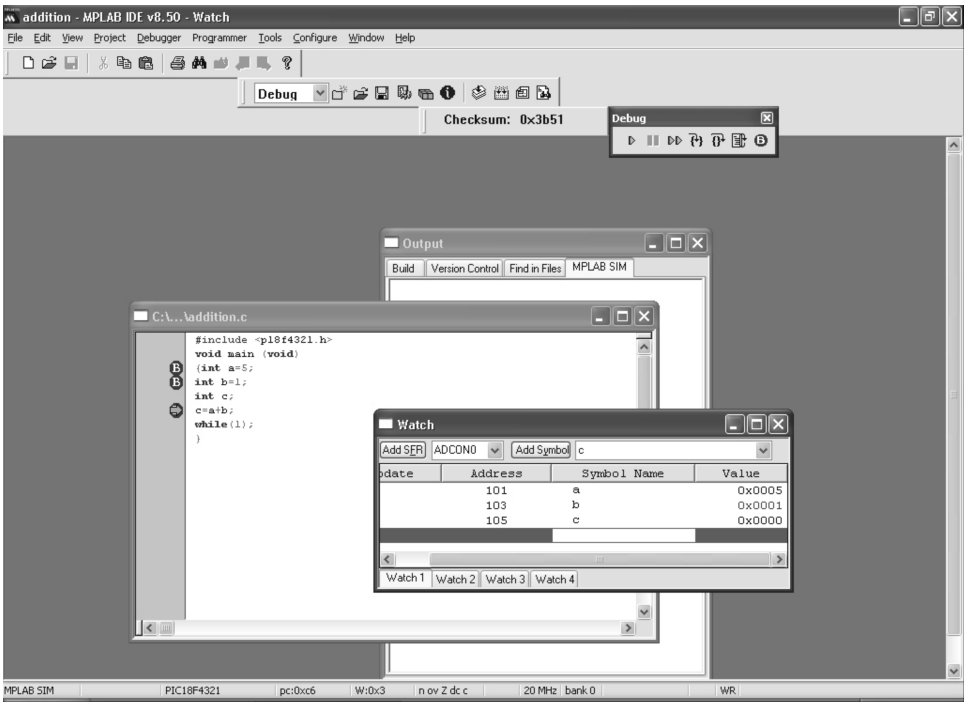


Next go to the Debug menu and Watch menu to see the contents of a, b, and c as each line is executed.

First go to Debug menu, and left click on reset (first symbol on right), and then click on the single arrow called the 'Run' arrow (left most arrow on the Debug menu), the code int a = 5; will be executed next. Click on single arrow again, the code is executed, and the following will be displayed:



Note that ‘a’ contains 5. Next, left click on the single arrow, the following will be displayed:

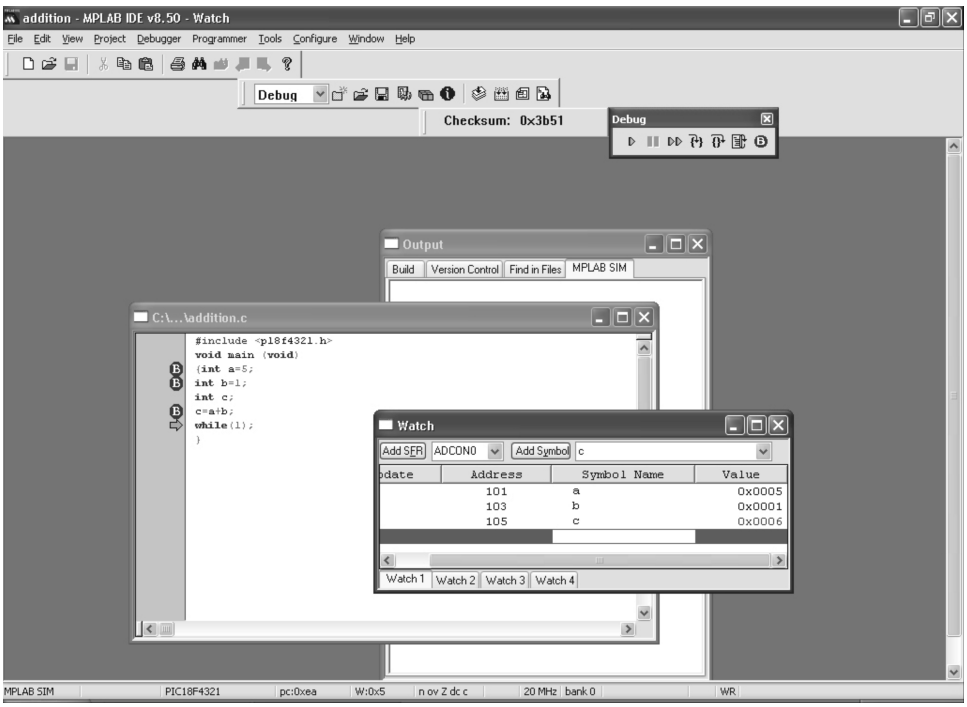


Note that 'b' contains 1 after execution of int b = 1;

Next, left click on the single arrow, and then left click on Halt (icon with two vertical lines, second from left on the Debug menu) to see the final result after execution of the line

```
c = a + b ;
```

as follows:



In the above, see that 'c' contains 6 (final answer).

The debugging is now complete.

APPENDIX H: INTERFACING THE PIC18F4321 TO A PERSONAL COMPUTER OR A LAPTOP USING PICkit™ 3

Appendix D contains the procedure for how to download a compiled C-program (compiled using the MPLAB C18 compiler of Appendix G) from a personal computer or a laptop into the PIC18F4321's memory. Microchip's PICkit3 is used to accomplish this. The PIC18F4321 is contained on the breadboard. Note that the PIC18F4321-based hardware on the breadboard is designed and built by the designer for a specific application.

After successful downloading of the desired program, the PIC18F-based breadboard can be disconnected from the Personal computer or Laptop. The PIC18F4321 chip on the breadboard can then execute the programs downloaded into its memory upon activating the RESET pushbutton. Thus, the desired task can be accomplished.

Appendix H contains the following:

H.1 INITIAL HARDWARE SETUP FOR THE PIC18F4321
H.2 CONNECTING THE PERSONAL COMPUTER (PC) OR THE LAPTOP TO THE PIC18F4321 VIA PICkit™ 3
H.3 PROGRAMMING THE PIC18F4321 FROM A PERSONAL COMPUTER OR THE LAPTOP USING PICkit™3

H.1 INITIAL HARDWARE SETUP FOR THE PIC18F4321

Figure H.1 shows the initial set up for the PIC18F4321 microcontroller. Pin #1 of the PIC18F4321 is the RESET input for the microcontroller. The $\overline{\text{MCLR}}$ (pin #1) must be connected to the reset circuit shown in the figure. There are two pairs of pins on the PIC18F4321 that must be connected to power and ground. For example, pins 11 and 32 must be connected directly to +5 V while pins 12 and 31 are connected directly to ground. Be sure not to connect any capacitors to these pins, connect them directly to either ground or +5 V. With any of the PIC18 family microcontrollers containing an "F" in the name, such as the PIC18F4321, the operating Vdd range is between 4.2 and 5.5 V. Figure H.1 also shows the proper connections for the header that will connect to the programmer. Note that the programmer has six pins but the sixth pin (Aux pin) makes no connections. After the PIC18F4321 is properly connected, the appropriate software must be installed.

There are two programs that must be installed in order to interface with the PIC18F. The first program is called *MPLAB* and the latest version can be downloaded at www.microchip.com/MPLAB. The second is called *MPLAB C18* and is the C compiler for the PIC18F which can be found at www.microchip.com/c18. Note that at this site there is

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

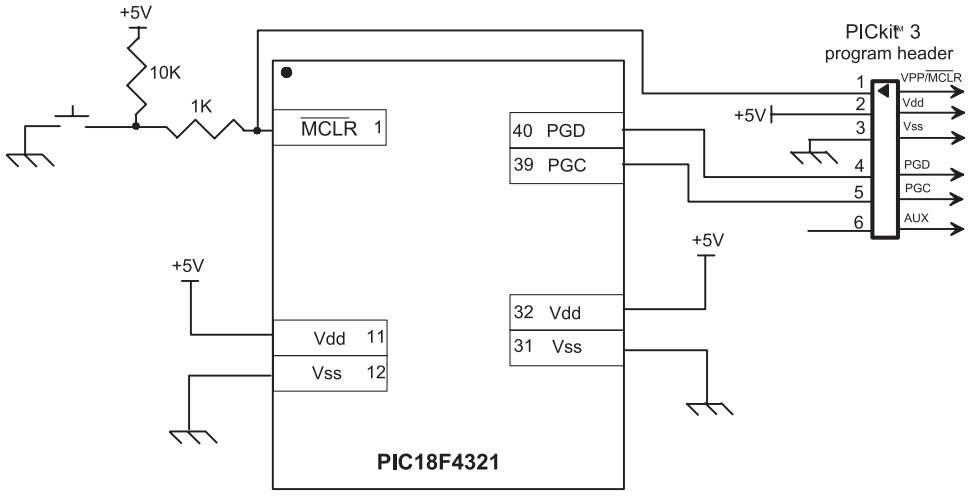


FIGURE H.1 Initial set up for the PIC18F4321

a link for academic use of the C18 compiler; be sure to click on the link and download the student C18 compiler. After the software has been installed, the PIC18F is now ready to be implemented.

H.2 CONNECTING THE PERSONAL COMPUTER (PC) OR THE LAPTOP TO THE PIC18F4321 VIA PICkit3

First, the PIC18F4321 initial setup circuit on the breadboard should be implemented. Next, a personal computer or a Laptop should be connected to the PIC18F4321 using the PICkit™ 3. Figure H.2 shows a simplified block diagram of the implementation.

Figure H.3 shows a pictorial view of the implementation. The following picture shows how the initial setup along with the reset circuit for the above block diagram would look like after building it on a breadboard:

Once the circuit is built, the PICkit™ 3 can be connected to the USB port of the computer as shown in Figure H.4. Next, the header part of the PICkit™ 3 can be connected to the header pins on the breadboard as shown in Figure H.5.

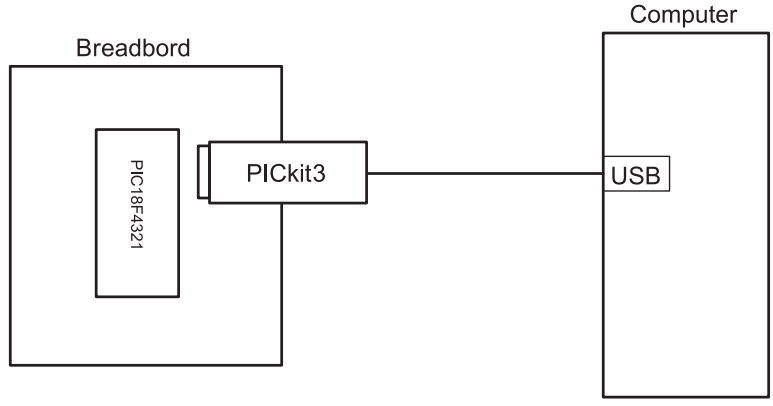


FIGURE H.2 PIC18F4321 computer interface using the PICkit™ 3

The other necessary I/O devices such as switches, LEDs, LCDs, and seven-segment displays can now be connected to the PIC18F4321 on the breadboard to perform some meaningful experiments. After implementing the desired hardware, the PIC18F4321 can then be programmed using the MPLAB software.

H.3 PROGRAMMING THE PIC18F4321 FROM A PERSONAL COMPUTER OR A LAPTOP USING THE PICkit3

In order to configure PICkit3 from a personal computer or a laptop, the user needs to click on the 'Programmer', and then select PICkit3 as follows:

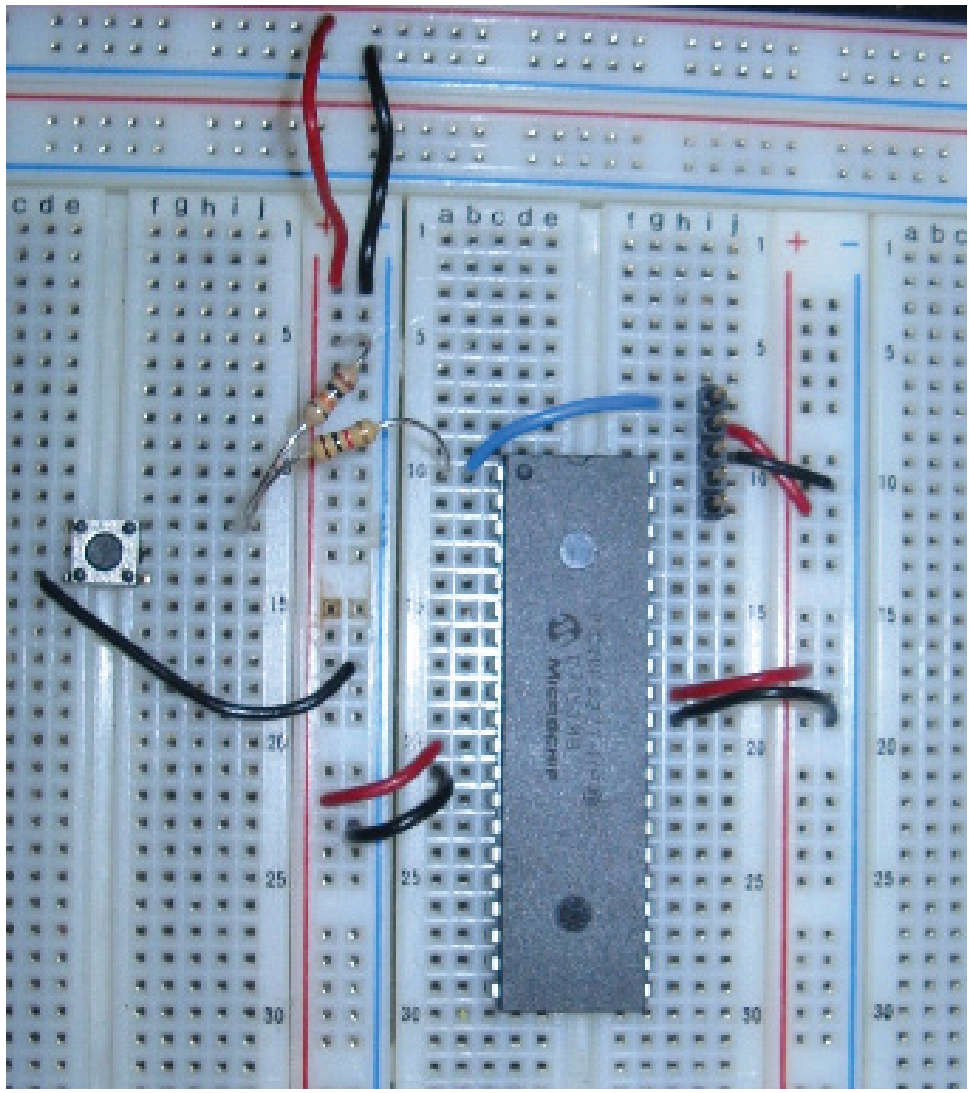


FIGURE H.3 Pictorial view of the breadboard implementation



FIGURE H.4 Pictorial view of connecting the PICkit™ 3 to the USB port

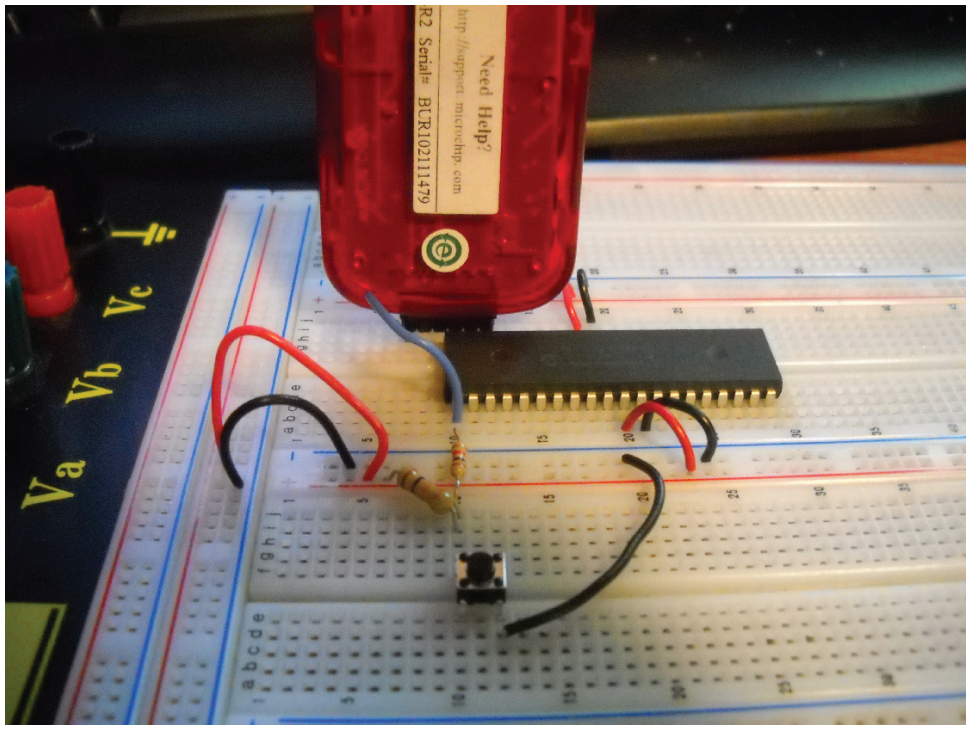
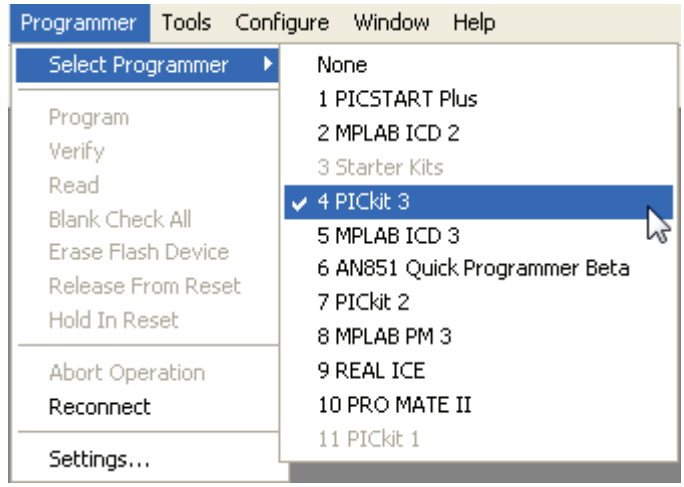
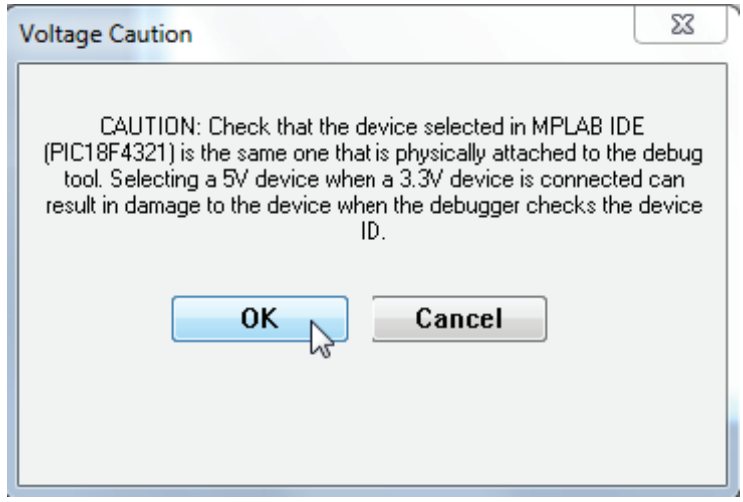


FIGURE H.5 Connecting the PICkit™ 3 to the breadboard

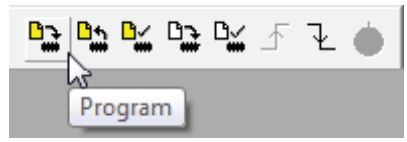


The following screen shot with the warning sign will appear. Just make sure that the PIC18F4321 microcontroller is connected to the proper voltage, and then click "OK" as follows:

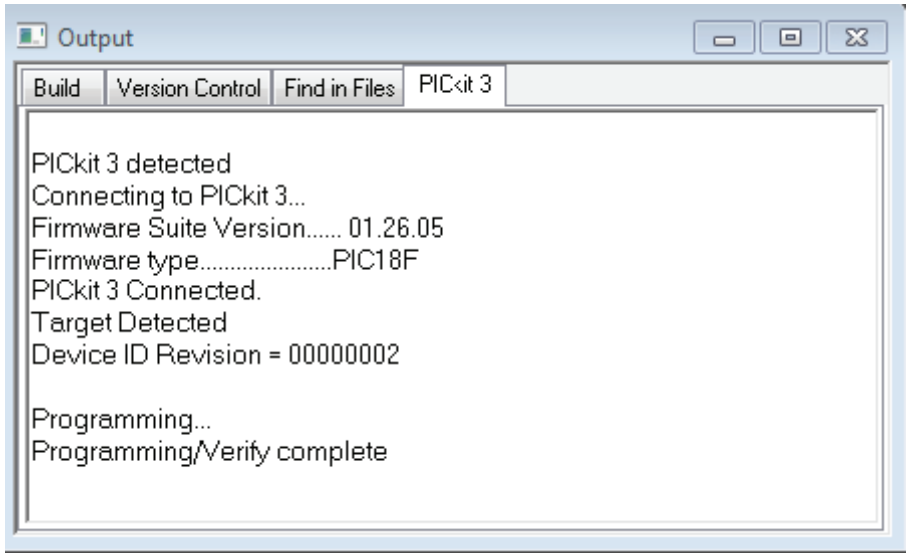


The PICkit3 is now connected.

Several options will appear at the top menu to program the PIC18F4321. The screen shot is provided below:



After successfully assembling or compiling a program, click the "program" option and MPLAB will download the program into the microcontroller.
The following message will apear indicating that the code was successfully programmed and verified onto the PIC18F:



This will complete downloading the programs from the computer into the PIC18F4321 microcontroller.

BIBLIOGRAPHY

Avinash, *Using the ADC (Analog to Digital Converter) of PIC Microcontroller, July 21, 2010. http://extremeelectronics.co.in/author/Avinash*

bestmicrocontrolerprojects.com *PIC Microcontroller hardware programming methods.*

byteparadigm.com *Introduction to I²C and SPI protocols.*

exploreembedded.com/wiki/PIC18F4520_Timer *PIC18F4520 Timer*

Gaonkar, Ramesh S., *Fundamentals of Microcontrollers and Applications in Embedded Systems (with the PIC18F Microcontroller Family*, *Thomson Delmar Learning, 2007.*

Geek Stuff and the geekstuff.com *C Arrays basics.*

Huang, Han-Way, *PIC Microcontroller: An Introduction to Software and Hardware Interfacing. Thomson Delmar Learning, 2005.*

Johnsonbaugh, R. and Kalin., M., *C for Scientists and Engineers*, *Prntice Hall, 1997.*

Majidi, M. A., Mckinlay, R. D., and Causey, D., *PIC Microcontroller and Embedded Systems using assembly and C for PIC18*, *Prentice Hall, 2008.*

Majidi, M. A., Mckinlay, R. D., and Causey, D., *PIC Microcontroller and Embedded Systems using assembly and C for PIC18, Muhammad Ali Mazidi, 2016.*

Microchip, *Using the PIC Devices' SSP and MCCP modules for slave I²C Communication, 2008.*

Microchip, *PIC18F46K20 Starter Kit C18 Lessons, 2007.*

Microchip Technology, Inc., *PIC18F4321 Family Data Sheet, 2009.*

microchipdeveloper.com, *External clock oscillator mode.*

microchipdeveloper.com, *Internal oscillator.*

microcontrollerboard.com. *PIC Analog Digital Converter Tutorial.*

Mikroelktronika, *Clock Oscillator, learn.mikroe.com.*

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

mikroe.com and the Hi-Tech C Manual, ver. 1.1, *PIC Microcontroller - The basics of C programming language.*

pic.9w2bsr.com *SPI tutorial, June 26, 2009.*

Programiz.com *Programiz C.*

Rafiquzzaman, M., *Microprocessor Theory and Applications with 68000/68020 and Pentium, 2008.*

Rafiquzzaman, M., *Microcontroller Theory and Applications with the PIC18F, Wiley, 2011.*

Rafiquzzaman, M., *Fundamentals of Digital Logic and Microcomputer Design, 6th Edition, Wiley, 2014.*

Rafiquzzaman, M., *Microprocessors and microcomputer Development Systems - Designing Microprocessor-Based Systems, Harper and Row, 1984.*

Rafiquzzaman, M., *Microcomputer Theory and Applicatins with the INTEL SDK-85, 2nd ed., John Wiley & Sons, 1987.*

Rafiquzzaman, M., *Microprocessors - Theory and Applications - Intel and Motorola, Prentice-Hall, 1992.*

Rafiquzzaman, M., and Chandra, R., *Mondern Computer Architecture, West / PWS, 1988.*

Rafiquzzaman, M., *Microprocessors and Microcomputer-Based System Design,* 1st ed., CRC Press, *1990.*

Rafiquzzaman, M., *Microprocessors and Microcomputer-Based System Design, 2nd ed., CRC Press, 1995.*

Robosumo cookbook, *Understanding timing and delays in the PIC18F4620, Wordpress.com, 2013.*

Wikipedia, *Serial Peripheral Interaface bus.*

INDEX

A

A/D converter, 2, 3, 301 - 315, 371
  Acquisition time, 302
  ADC channels, 302
  ADC clock, 302 - 303
  Interrupt-driven ADC, 306, 308
  Polled ADC, 306 - 307
  Programming PIC18F in assembly, 309, 310 -314
  Programming PIC18F in C , 309, 314 -315
  Reference voltage, 302
  Resolution, 302
Accumulator, 22, 371
Accumulator-based CPU, 22
ADC, See A/D converter
ADCON0 register, 303 -304
ADCON1 register, 304 - 305
ADCON2 register, 305 -306
Addition of signed and unsigned binary numbers, 67
ADDLW, 130 - 131
Address, 3, 371
Address bus, 4, 18, 19
Address decoding techniques, 42 -45
Addressing modes, 3, 104 -112
ADDWF, 122, 131, 383
ADDWFC, 122, 131, 132, 383
Alphanumeric Codes, 10
ALU, 4, 21. 30
Analog to Digital converter, See A/D Converter
ANDLW, 138 -139, 383
ANDWF, 139 -140, 383
Arduino, 14
Arithmetic and logic unit, See ALU
ARM, 13
ASCII, 10, 371
Assembler, 59 - 67, 115 -119, 371, 435 - 463
  Delimiters, 62 -63
  Directives, 63 - 65
  Fields, 65 - 67
  List file, 119
  MPLAB, 62, 115 - 119, 435 - 463
  Object code, 59, 377
  Pseudoinstructions, 63 - 65
  Specifying numbers, 63
  Types, 61 - 62
Assembler Directives, 63 - 65
Assembly Language,
  A/D Converter programming, 309, 310 -314
  Capture mode programming, 322 - 325
  CCP programming, 321 - 334
  Compare mode programming, 326 - 327
  Configuring ports, 211 - 213
  DC Motor control, 332 - 324
  Delay routines, 188 - 190
  Interrupt I/O programming, 236 - 239, 243, 245, 246, 248, 250
  LCD programming, 252 - 258
  MPLAB tutorial, 435 - 463
  Programmed I/O, 216 - 220
  PWM mode programming, 329 - 332
  Serial I/O programming, 344 - 346, 354 - 357
  Timers, 279 - 281, 284, 290 - 291
  Voltmeter design, 309 - 315
Assembly Language definition, 371
Assembly Language vs. C Language, 88 - 89
Asynchronous serial data transmission, 337
Atmel AVR, 13

B

Basic CPU Registers, 22
Bcc, 155 - 156
BC, 156, 383
BCD, 10 - 11, 372
BCF, 146, 147, 148, 383
BICMOS, 2
Big-endian, 4, 371
Binary, 1
Binary-coded-decimal, See BCD
Bit, 4, 372
BN, 156, 383
BNC, 110, 156, 383
BNN, 156, 383

© 2018 John Wiley & Sons, Inc. Published 2018 by John Wiley & Sons, Inc.

BNOV, 156, 383
BNZ, 156, 383
BOV, 156, 383
BRA, 110, 156, 157
BSF, 147, 148, 383
BTFSC, 157, 158, 384
BTFSS, 158, 159, 384
BTG, 147, 148, 384
Buffer, 372
Bus, 4, 18 - 19, 372
Byte, 4, 8
BZ, 110 - 112, 384

C

C language,
  Arrays, 88, 222 - 223
  Basics, 76 - 78
  Basics for PIC18F4321, 220 - 224
  Bit manipulation operations, 79 - 81
  Configuring ports, 213 - 215
  Control structures, 81 -85
  A/D converter programming, 309, 314 - 315
  Data types, 78 - 79, 222
  D/A programming, 315 - 316
  DC Motor, 332, 333, 334 - 336
  Functions, 86 - 87, 322
  I^2C mode, 348 - 358
  Interrupt I/O, 243 -252
  LCD, 252 - 254, 258 - 259
  Macros, 88
  MPLAB tutorial, 465 - 492
  Preprocessor directives, 220
  Programmed I/O, 208 - 211, 213 - 215, 220 - 224
  PWM mode, 329 - 331, 332
  Serial I/O, 346 - 348, 357 - 358
  SPI mode, 326 - 348
  Structures, 85 - 86
  Timers, 273 - 301
  Unions, 85 -86
  Voltmeter design, 309, 310, 314 - 315
CALL, 167 - 170, 173 - 177, 384
Carry flag, 98, 99, 131, 132
CCP, 321 - 336, 372
  Programming in assembly, 324 - 325, 327, 331, 333 - 334
  Programming in C, 325 - 326, 328 - 329, 331, 332, 334 - 336
Central processing unit, See CPU
CISC, 4, 33 - 34, 372
Clock, 4, 199 - 202, 372
CLRF, 124, 384
CLRWDT, 169, 384
CMOS, 2, 372
CMOS outputs, 48
Codes,
  Alphanumeric, 10
  ASCII, 10
  BCD, 10 - 11
  EBCDIC, 10
COMF, 139, 140, 384
Compiler, 1, 372 - 373
Complementary MOS, See CMOS
Computer, 1, 373
Conditional programmed I/O, 52 - 53, 231 - 233
Control bus, 29, 30
Control Unit, 21, 373
Control Unit Design,
  Hardwired Control, 30, 31
  Microprogrammed Control, 30, 31
CPFSEQ, 158, 159, 160, 384
CPFSGT, 158, 159, 384
CPFSLT, 158, 159, 384
CPU,
  ALU, 4, 30
  Control Unit, 4, 29 - 31
  Registers, 6 - 7
CPU definition, 372
CPU registers, 22
Cross Assembler, 62

D

D/A, 2, 3, 315 - 316, 373
DAC, See D/Aconverter
Data, 1, 373
Data bus, 4
Data-direction register, 50
DAW, 131, 134, 384
Debouncing, 259, 260
DECF, 131, 133, 384
DECFSZ, 158, 160, 385
DECFSNZ, 158, 160, 385
Delay routine, 188 - 190, 223 - 224
Delimiters, 62 -63
Digital to Analog converter, See D/A
DRAM, 6, 38, 39, 40, 373
Duty cycle, 322, 329, 373
Dynamic RAM, See DRAM

E

EBCDIC, 10,374
E^2PROM, 4, 373
EEPROM, 4, 373
Embedded Controller, 14 - 15
EPROM, 5, 373
Evolution of the Microcontroller, 11 - 14

F

Fetch, 23
Fetch timing diagram, 40
Flags, 26 - 27, 374
Flash memory, 5, 374
Flowcharts, 89

G

General-purpose Register-based CPU, 22 - 23

General-purpose Registers, 26
GOTO, 156, 385

H

Hardware, 1, 374
Hardware stack, 27 - 29
Hardwired control, 31, 374
Harvard architecture, 5, 374
HC11, 4, 12, 13
HCMOS, 2
High-level language, 59, 74 - 88, 374
High-speed CMOS, See HCMOS

I

I²C,337, 338, 348 - 358, 374
I/O,
  Interrupt I/O, 53 - 55
  Programmed I/O, 50 - 53
INCF, 131, 133, 385
INCFSZ, 158, 161, 162, 385
INCFSNZ, 158, 161, 385
Index Register, 26, 375
Input/Output, See I/O
Instruction, 1, 375
Instruction Fetch Timing Diagram, 40
Instruction format, 65 - 67
Instruction Register, 22
Instruction Set, 67 - 73, 375
INT0, 238 - 252
INT1, 238 - 252
INT2, 238 - 252
INTCON register, 237
INTCON2 register, 240, 241
INTCON3 register, 237
Intel 8051, 12
Internal interrupts, 55, 236, 238, 252, 375
Interrupt Address Vector, 55
Interrupt I/O, 53 - 55, 231, 236 - 252, 375
Interrupt Priorities, 241 - 252
Interrupt Types, 54 - 55
Interrupt service routine, 233, 240, 243, 244
IORLW, 139, 140, 385
IORWF, 139, 140, 141, 385
ISR, See Interupt service routine

K

Keyboard interface, 259 - 269
Keybounce, 259, 260, 375

L

LCD,
  Programming in assembly, 252 - 258
  Programming in C, 252 - 254, 258 - 259
LCD commands, 256
LCD definition, 252, 375
LED, 49 - 51, 215 - 216, 375
LFSR, 109, 124, 386
Light Emitting Diodes, See LED
Liquid Crystal Displays, See LCD
Little-endian, 5, 375

M

Machine language, 59, 60, 376
Main memory, 38 - 39, 378
Main Memory Array Design, 42 - 45
Maskable interrupts, 54, 55
Masking operation, 139
Memory, 37 - 47
Memory Organization, 42
Memory map, 376
Memory-mapped I/O, 51, 376
Memory Types,
  DRAM, 6, 38, 39, 40, 373
  E²PROM, 4, 373
  EAROM, 4, 373
  EPROM, 5, 373
  Flash, 5, 39, 374
  Primary memory, 37, 378
  RAM, 6, 39, 40
  ROM, 6, 39
  SRAM, 6, 37, 38, 39
Microcomputer, 2, 5, 376
Microcontroller, 2, 5, 376
Microcoontroller Bus, 18
Microcontroller input/output circuit,
  LED, 50, 51
  Seven-segment displays, 50, 51
  Switch, 49
Microcontroller-LED interface, 215, 216
Microprocessor, 1, 25, 376
Microprogrammed Control Unit, 30, 31
Microprogramming, 31, 376 - 377
MOVF, 124, 125, 386
MOVFF, 124, 125, 386
MOVLB, 102, 124, 386
MOVLW, 124, 125, 386
MOVWF, 124, 125, 386
MPLAB,
  Assembly language tutorial, 435 -463
  C-language tutorial, 465 - 492
  PICkit3 initial set up, 493 - 498
MSP430, 13, 14, 22
MULLW, 131, 133, 386
MULWF, 131, 133, 386

N

NEGF, 131, 133, 386
Nibble, 4, 377
NOP, 169, 386

O

Object codes, 59
Open collector outputs, 47, 48
Overflow, 8 - 9

OSCCON register, 200 - 202

P

Packed BCD, 10 - 11
Parity, 377
Peripheral interrupt, 252, 308
PIC18F,
  A/D Converter, 302 - 315
  Addressing modes, 104 - 112
  Bank Select Register, 97, 98, 101 -103
  CCP, 321 - 336
  D/A converter, 315 - 316
  Data memory, 101 - 103
  DC Motor control, 332 - 336
  Delay routines, 188 - 190
  Instruction format, 120 - 121
  Instruction set, 121 - 150, 155 - 188, 383 - 431
  Interrupts, 236 - 252
  Memory organization, 100 - 104
  MPLAB, 115 - 119, 435 - 492
  Peripheral interrupt, 252
  Pipelining, 95 - 96
  Program memory, 101
  Registers, 96 - 100
  Serial I/O, 338 - 358
  Special Function Registers, 103
  Stack, 171 - 173
  Status flags, 99 - 100
  Timers,
    INTCON register, 278
    PIE1 register, 287
    PIE2 register, 295
    PIR1 register, 286
    PIR2 register, 294
    Polling timer Flags, 274
    Prescaler, 275
    Postscaler, 275
    Timer0, 273, 275 - 283
    Timer1, 273, 282 - 289
    Timer2, 273, 289 - 293
    Timer3, 273, 293 - 301
    T0CON register, 276
    T1CON register, 285
    T2CON register, 291
    T3CON register, 293
    Timer interrupts, 274 - 275
PIC18F4321,
  Absolute mode, 104, 105
  Access bank, 101 - 102
  A/D converter, 302 - 315
  ADC, See A/D converter
  ADCON0 register, 303 - 304
  ADCON1 register, 304 - 305
  ADCON2 register, 305 - 306
  Addressing modes, 104 - 112
  Analog interface, 301 - 316
  Arithmetic instructions,130 - 138
  Assembler, 115 - 119
  Assembler directives, 116 - 118
  Bit addressing mode, 112
Bit manipulation instructions, 147 - 148
Block diagram, 34 - 35
Branch instructions, 155 - 157
BSR, 98
Capture mode, 321, 322 - 326
CCP module, 321
CCPxCON register, 323, 326 - 329
Clock, 199 - 202
Compare instructions, 158, 159, 160
Compare mode, 326 - 329
Comparison with other microcontrollers, 13, 94
Conditional branch instructions, 155 - 157
Configuring I/O ports, 211 - 215
D/A converter, 315 - 316
DAC, See D/A converter
Data memory, 101 - 103
Data Movement instructions, 124 - 126
DC motor control, 332 - 336
Default crystal frequency, 201
Delay routine, 188 - 190
Direct mode, 104, 105
EEPROM, 100, 101
External interrupts, 238 - 251
External interrupts in default, 238 - 248
File Select Registers, 97, 98
Flags, 99, 100
Flash memory, 13, 101
FSR, 97, 98
Hardware stack pointer, 97, 98
I²C, 337, 348 - 358
I/O, 208 - 252
I/O instructions, 211 - 215
I/O port configuration, 211 -215
Immediate mode, 104
Implied addressing mode, 104
Indexed mode, 106, 108, 109
Indirect mode, 105 - 109
Inherent mode, 104
Instruction format, 120 - 121
Instruction set, 121 - 177, 383 - 431
Instruction Set in alphabetical order, 383 - 387
Instructions affecting status flags, 123
INT0, 238 - 252
INT1, 238 - 252
INT2, 238 - 252
INTCON register, 237
INTCON2 register, 240, 241
INTCON3 register, 237
Internal interrupts, 236, 238, 252
Interrupt-on-change, 242, 243
Interrupt priorities, 240, 241, 244 - 251
Interrupt procedure, 236
Interrupt registers, 240
Interrupt Service Routine, 233, 240, 243, 244
Interrupt triggering levels, 243
Interrupt types, 236, 238
IPEN bit, 241, 242
Jump instructions, 155 - 157
Keyboard/display Interface, 259 - 269
LCD Interface, 252 - 259

Literal mode, 104
Logic instructions, 138 - 141
Manual reset, 203 - 204
Memory organization, 100 - 104
MPLAB, 115 - 119, 435 - 498
OSCCON register, 200
PICkit3, 206 - 207, 493 - 498
PIE1 register, 287
PIE2 register, 295
Pins and Signals, 195 - 199
Pipelining, 95 - 96
PIR1 register, 286
PIR2 register, 294
Ports, 210 - 211
Postdecrement mode, 107 - 108
Postincrement mode, 106 - 107
Power-on reset, 202 - 203
Preincrement mode, 108
PRODH/PRODL, 97, 98
Program counter, 97, 98
Program memory, 100 - 101
Programmed I/O, 208 - 211, 213 - 215, 216 - 220
PWM mode, 321, 329 - 336
RCON register, 204, 220, 242
Registers, 96 - 100
Relative mode, 110 - 112
Reset, 202 - 205
RETFIE, 243
Rotate instructions, 141 - 143
Serial I/O, 336 - 358
Seven-segment display, 215 - 216
Skip instructions, 158
Special Function Registers, 103, 433
SPI, 337 - 348
SRAM, 101 - 104
SSPADD, 350, 353
SSPBUF register, 339, 340
SSPCON1 register, 339, 341, 350, 352
SSPCON2, 350, 353
SSPSR register, 339, 340
SSPSTAT register, 339, 342, 350, 351
Stack, 167, 168, 169, 170, 171, 172, 173
Stack Pointer, 97, 98
Status register, 99 - 100
STKPTR, 97
Subroutine instructions, 168
Subroutines, 170, 173 - 177
System control instructions, 170
Table Read/Write instructions, 109, 110, 163 - 166
TBLPTR, 163 - 166
Test/Compare/Skip instructions, 157 - 162
Timer0, 273, 275 - 283
Timer1, 273, 282 - 289
Timer2, 273, 289 - 293
Timer3, 273, 293 - 301
T0CON register, 276
T1CON register, 285
T2CON register, 291
T3CON register, 293
Voltmeter design, 309 - 315
WREG, 97
PIC18F4520, 94
Pipelining, 378
Program, 1, 378
Program Counter, 22, 378

R

RAM, 6, 39, 378
Random Access Memory, See RAM
RCALL, 386
RCON register, 204, 240, 242
READ Timing Diagram, 40, 41
Read-Only Memory, See ROM
Register, 378, 379
RESET, 169, 386
RETFIE, 167
RETLW, 167, 185 - 186, 386
RETURN, 167, 170, 386
RISC, 6, 33 - 34, 70
RLCF, 141, 143. 386
RLNCF, 1422, 143, 387
ROM, 39, 378
RRCF, 142, 143, 387
RRNCF, 142, 143, 387

S

Serial I/O, 338 - 358
SETF, 124, 125, 387
Seven Segment Displays,
Common anode, 215
Common cathode, 215
Signed binary numbers, 7, 8, 379
Signed division, 68 - 70, 182 - 183
Sign extension, 7, 8
Signed multiplication, 67 - 68, 178 - 180
Single-chip microcomputer, 2
Single-Chip Microprocessor, 2
SLEEP, 169
Software, 1, 379
SPI, 337, 338 - 348, 380
SRAM, 6, 38, 39, 40
Stack,
Definition, 27, 380
Hardware stack, 171, 172
Software stack, 172, 173
Stack Pointer, 27 - 28
Standard I/O,
Static RAM, See SRAM
Status Register, 26 - 27
SUBFWB, 131, 132, 387
SUBLW, 131, 132, 387
Subroutine, 167 - 170, 173 - 177, 380
SUBWF, 131, 132, 387
SUBWFB, 131, 132, 387
SWAPF, 124 - 126, 387
Synchronous serial data transmission, 336 - 337
System bus, 18 - 19

T

TBLRD, TBLWT, 163 - 166, 387
TBLWT, 163 - 166, 387
Texas Instruments MSP430,
Timers, See PIC18F Timers
Totem pole outputs, 47 - 48
Tristate outputs, 48
TSTFSZ, 158, 162, 387
TTL outputs, 47 - 48
Two-Pass Assembler, 62
T0CON register, 276
T1CON register, 285
T2CON register, 291
T3CON register, 293

U

Unpacked BCD, 10, 11
Unsigned binary numbers, 7
Unsigned division, 68, 180 - 182
Unsigned multiplication, 67
Unsigned number, 7, 380

V

VLSI, 380
von Neumann architecture, 7

W

WREG, 97
WRITE Timing Diagram, 41 - 42

X

XORLW, 139, 141, 387
XORWF, 139, 141, 387

Z

Zero extension, 7
Zero flag, 27, 99