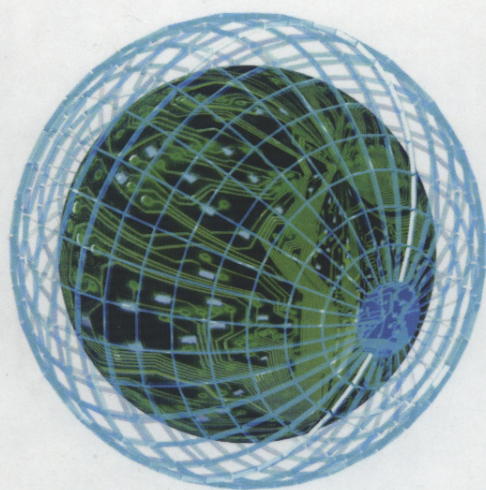


the avr microcontroller and embedded system

using assembly and c



MUHAMMAD ALI MAZIDI
SARMAD NAIMI
SEPEHR NAIMI



the avr microcontroller and embedded systems using assembly and c

MUHAMMAD ALI MAZIDI, SARMAD NAIMI, AND SEPEHR NAIMI



The AVR microcontroller from Atmel is one of the most widely used 8-bit microcontrollers in the world. In this book the authors use a step-by-step and systematic approach to show the programming of the AVR chip. Examples in both Assembly language and C show how to program many of the AVR features, such as timers, serial

communication, ADC, SPI, I2C, and PWM. The text is organized into two parts:

- The first seven chapters use Assembly language programming to examine the internal architecture of the AVR.
- Chapters 7–18 use both Assembly and C to show the AVR peripherals and I/O interfacing to real-world devices such as LCDs, motors, and sensors.

The AVR Microcontroller and Embedded Systems is the latest volume in the series of textbooks by Mazidi et al. This series of texts is widely used around the world by both industry and academics and has been translated into many languages. The other titles in the series are:

The x86 PC (5th ed.)

The 8051 Microcontroller and Embedded Systems (2nd ed.)

The PIC Microcontroller and Embedded Systems

The HCS12 Microcontroller and Embedded Systems

Titles to come include:

The ARM Microcontroller and Embedded Systems

CHAPTERS

- 0: Introduction to Computing
- 1: The AVR Microcontroller: History and Features
- 2: AVR Architecture and Assembly Language Programming
- 3: Branch, Call, and Time Delay Loop
- 4: AVR I/O Port Programming
- 5: Arithmetic, Logic Instructions, and Programs
- 6: AVR Advanced Assembly Language Programming
- 7: AVR Programming in C
- 8: AVR Hardware Connection, Hex File, and Flash Loaders
- 9: AVR Timer Programming in Assembly and C
- 10: AVR Interrupt Programming in Assembly and C
- 11: AVR Serial Port Programming in Assembly and C
- 12: LCD and Keyboard Interfacing
- 13: ADC, DAC, and Sensor Interfacing
- 14: Relay, Optoisolator, and Stepper Motor Interfacing with AVR
- 15: Input Capture and Wave Generation in AVR
- 16: PWM Programming and DC Motor Control in AVR
- 17: SPI Protocol and MAX7221 Display Interfacing
- 18: I2C Protocol and DS1307 RTC Interfacing

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

ISBN-13: 978-0-13-800331-9
ISBN-10: 0-13-800331-9



9 780138 003319

THE AVR MICROCONTROLLER AND EMBEDDED SYSTEMS

Using Assembly and C

Muhammad Ali Mazidi

Sarmad Naimi

Sepehr Naimi

Prentice Hall

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editor in Chief: Vernon Anthony
Acquisitions Editor: Wyatt Morris
Editorial Assistant: Chris Reed
Director of Marketing: David Gesell
Marketing Manager: Kara Clark
Senior Managing Coordinator: Alicia Wozniak
Marketing Assistant: Les Roberts
Senior Managing Editor: JoEllen Gohr

Project Manager: Rex Davidson
Senior Operations Supervisor: Pat Tonneman
Operations Specialist: Laura Weaver
Art Director: Dianne Ernsberger
Cover Designer: Jeff Vanik
Cover Art: Antonis Papantoniou, Fotolia.com
Printer/Binder: Courier/Kendallville
Cover Printer: Demand Production Center
Text Font: Times Roman

Copyright © 2011 Pearson Education, Inc., publishing as Prentice Hall, 1 Lake Street, Upper Saddle River, New Jersey, 07458. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 1 Lake Street, Upper Saddle River, New Jersey, 07458.

Library of Congress Cataloging in Publication Data

Mazidi, Muhammad Ali.

The AVR microcontroller and embedded systems: using Assembly and C /
Muhammad Ali Mazidi, Sarmad Naimi, Sepehr Naimi.

p. cm.

ISBN-13: 978-0-13-800331-9 (alk. paper)

ISBN-10: 0-13-800331-9 (alk. paper)

1. Atmel AVR microcontroller. 2. Embedded computer systems. 3. Assembler language (Computer program language) 4. C (Computer program language) I. Naimi, Sarmad. II. Naimi, Sepehr. III. Title.

TJ223.P76M378136 2009

004.16--dc22

2009039790

10 9 8 7 6 5 4 3 2 1

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

ISBN 10: 0-13-800331-9

ISBN 13: 978-0-13-800331-9

*This book is dedicated
to the memory of Dr. Kamal Bakhtavar
for all his sacrifices.
– Muhammad Ali Mazidi*

*This book is dedicated
to the memory of Dr. P. Javid
for his inspiring example of dedication to the education of young people.
– Sarmad Naimi*

*This book is dedicated to
Yaran.
– Sepehr Naimi*

Regard man as a mine rich in gems of
inestimable value. Education can, alone, cause it
to reveal its treasures, and enable mankind to
benefit therefrom.

Baha'u'llah

BRIEF CONTENTS

CHAPTERS

0:	Introduction to Computing	1
1:	The AVR Microcontroller: History and Features	39
2:	AVR Architecture and Assembly Language Programming	55
3:	Branch, Call, and Time Delay Loop	107
4:	AVR I/O Port Programming	139
5:	Arithmetic, Logic Instructions, and Programs	161
6:	AVR Advanced Assembly Language Programming	197
7:	AVR Programming in C	255
8:	AVR Hardware Connection, Hex File, and Flash Loaders	289
9:	AVR Timer Programming in Assembly and C	311
10:	AVR Interrupt Programming in Assembly and C	363
11:	AVR Serial Port Programming in Assembly and C	395
12:	LCD and Keyboard Interfacing	429
13:	ADC, DAC, and Sensor Interfacing	463
14:	Relay, Optoisolator, and Stepper Motor Interfacing with AVR	491
15:	Input Capture and Wave Generation in AVR	509
16:	PWM Programming and DC Motor Control in AVR	549
17:	SPI Protocol and MAX7221 Display Interfacing	603
18:	I2C Protocol and DS1307 RTC Interfacing	629

APPENDICES

A:	AVR Instructions Explained	695
B:	Basics of Wire Wrapping	733
C:	IC Interfacing and System Design Issues	737
D:	Flowcharts and Pseudocode	755
E:	AVR Primer for 8051 Programmers	761
F:	ASCII Codes	762
G:	Assemblers, Development Resources, and Suppliers	764
H:	Data Sheets	766

CONTENTS

CHAPTER 0: INTRODUCTION TO COMPUTING	1
SECTION 0.1: NUMBERING AND CODING SYSTEMS	2
SECTION 0.2: DIGITAL PRIMER	9
SECTION 0.3: SEMICONDUCTOR MEMORY	13
SECTION 0.4: CPU ARCHITECTURE	29
CHAPTER 1: THE AVR MICROCONTROLLER: HISTORY AND FEATURES	39
SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS	40
SECTION 1.2: OVERVIEW OF THE AVR FAMILY	44
CHAPTER 2: AVR ARCHITECTURE AND ASSEMBLY LANGUAGE PROGRAMMING	55
SECTION 2.1: THE GENERAL PURPOSE REGISTERS IN THE AVR	56
SECTION 2.2: THE AVR DATA MEMORY	59
SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY	61
SECTION 2.4: AVR STATUS REGISTER	71
SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES	75
SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING	80
SECTION 2.7: ASSEMBLING AN AVR PROGRAM	82
SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN THE AVR	85
SECTION 2.9: RISC ARCHITECTURE IN THE AVR	93
SECTION 2.10: VIEWING REGISTERS AND MEMORY WITH AVR STUDIO IDE	97
CHAPTER 3: BRANCH, CALL, AND TIME DELAY LOOP	107
SECTION 3.1: BRANCH INSTRUCTIONS AND LOOPING	108
SECTION 3.2: CALL INSTRUCTIONS AND STACK	118
SECTION 3.3: AVR TIME DELAY AND INSTRUCTION PIPELINE	128
CHAPTER 4: AVR I/O PORT PROGRAMMING	139
SECTION 4.1: I/O PORT PROGRAMMING IN AVR	140
SECTION 4.2: I/O BIT MANIPULATION PROGRAMMING	149
CHAPTER 5: ARITHMETIC, LOGIC INSTRUCTIONS, AND PROGRAMS	161
SECTION 5.1: ARITHMETIC INSTRUCTIONS	162
SECTION 5.2: SIGNED NUMBER CONCEPTS AND ARITHMETIC OPERATIONS	170
SECTION 5.3: LOGIC AND COMPARE INSTRUCTIONS	176
SECTION 5.4: ROTATE AND SHIFT INSTRUCTIONS AND DATA SERIALIZATION	183
SECTION 5.5: BCD AND ASCII CONVERSION	190
CHAPTER 6: AVR ADVANCED ASSEMBLY LANGUAGE PROGRAMMING	197
SECTION 6.1: INTRODUCING SOME MORE ASSEMBLER DIRECTIVES	198
SECTION 6.2: REGISTER AND DIRECT ADDRESSING MODES	202
SECTION 6.3: REGISTER INDIRECT ADDRESSING MODE	208

SECTION 6.4: LOOK-UP TABLE AND TABLE PROCESSING	216
SECTION 6.5: BIT-ADDRESSABILITY	226
SECTION 6.6: ACCESSING EEPROM IN AVR	233
SECTION 6.7: CHECKSUM AND ASCII SUBROUTINES	238
SECTION 6.8: MACROS	244
CHAPTER 7: AVR PROGRAMMING IN C	255
SECTION 7.1: DATA TYPES AND TIME DELAYS IN C	256
SECTION 7.2: I/O PROGRAMMING IN C	263
SECTION 7.3: LOGIC OPERATIONS IN C	265
SECTION 7.4: DATA CONVERSION PROGRAMS IN C	275
SECTION 7.5: DATA SERIALIZATION IN C	280
SECTION 7.6: MEMORY ALLOCATION IN C	282
CHAPTER 8: AVR HARDWARE CONNECTION, HEX FILE, AND FLASH LOADERS	289
SECTION 8.1: ATMEGA32 PIN CONNECTION	290
SECTION 8.2: AVR FUSE BITS	294
SECTION 8.3: EXPLAINING THE HEX FILE FOR AVR	300
SECTION 8.4: AVR PROGRAMMING AND TRAINER BOARD	305
CHAPTER 9: AVR TIMER PROGRAMMING IN ASSEMBLY AND C	311
SECTION 9.1: PROGRAMMING TIMERS 0, 1, AND 2	313
SECTION 9.2: COUNTER PROGRAMMING	348
SECTION 9.3: PROGRAMMING TIMERS IN C	353
CHAPTER 10: AVR INTERRUPT PROGRAMMING IN ASSEMBLY AND C	363
SECTION 10.1: AVR INTERRUPTS	364
SECTION 10.2: PROGRAMMING TIMER INTERRUPTS	369
SECTION 10.3: PROGRAMMING EXTERNAL HARDWARE INTERRUPTS	376
SECTION 10.4: INTERRUPT PRIORITY IN THE AVR	381
SECTION 10.5: INTERRUPT PROGRAMMING IN C	385
CHAPTER 11: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY AND C	395
SECTION 11.1: BASICS OF SERIAL COMMUNICATION	396
SECTION 11.2: ATMEGA32 CONNECTION TO RS232	403
SECTION 11.3: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY	405
SECTION 11.4: AVR SERIAL PORT PROGRAMMING IN C	419
SECTION 11.5: AVR SERIAL PORT PROGRAMMING IN ASSEMBLY AND C USING INTERRUPTS	422
CHAPTER 12: LCD AND KEYBOARD INTERFACING	429
SECTION 12.1: LCD INTERFACING	430
SECTION 12.2: KEYBOARD INTERFACING	452
CHAPTER 13: ADC, DAC, AND SENSOR INTERFACING	463
SECTION 13.1: ADC CHARACTERISTICS	464
SECTION 13.2: ADC PROGRAMMING IN THE AVR	469

SECTION 13.3: SENSOR INTERFACING AND SIGNAL CONDITIONING	480
SECTION 13.4: DAC INTERFACING	484
CHAPTER 14: RELAY, OPTOISOLATOR, AND STEPPER MOTOR INTERFACING WITH AVR	491
SECTION 14.1: RELAYS AND OPTOISOLATORS	492
SECTION 14.2: STEPPER MOTOR INTERFACING	498
CHAPTER 15: INPUT CAPTURE AND WAVE GENERATION IN AVR	509
SECTION 15.1: WAVE GENERATION USING 8-BIT TIMERS	510
SECTION 15.2: WAVE GENERATION USING TIMER1	520
SECTION 15.3: INPUT CAPTURE PROGRAMMING	531
SECTION 15.4: C PROGRAMMING	539
CHAPTER 16: PWM PROGRAMMING AND DC MOTOR CONTROL IN AVR	549
SECTION 16.1: DC MOTOR INTERFACING AND PWM	550
SECTION 16.2: PWM MODES IN 8-BIT TIMERS	560
SECTION 16.3: PWM MODES IN TIMER1	574
SECTION 16.4: DC MOTOR CONTROL USING PWM	597
CHAPTER 17: SPI PROTOCOL AND MAX7221 DISPLAY INTERFACING	603
SECTION 17.1: SPI BUS PROTOCOL	604
SECTION 17.2: SPI PROGRAMMING IN AVR	609
SECTION 17.3: MAX7221 INTERFACING AND PROGRAMMING	615
CHAPTER 18: I2C PROTOCOL AND DS1307 RTC INTERFACING	629
SECTION 18.1: I2C BUS PROTOCOL	630
SECTION 18.2: TWI (I2C) IN THE AVR	638
SECTION 18.3: AVR TWI PROGRAMMING IN ASSEMBLY AND C	642
SECTION 18.4: DS1307 RTC INTERFACING AND PROGRAMMING	654
SECTION 18.5: TWI PROGRAMMING WITH CHECKING STATUS REGISTER	668
APPENDIX A: AVR INSTRUCTIONS EXPLAINED	695
SECTION A.1: INSTRUCTION SUMMARY	696
SECTION A.2: AVR INSTRUCTIONS FORMAT	700
SECTION A.3: AVR REGISTER SUMMARY	732
APPENDIX B: BASICS OF WIRE WRAPPING	733
APPENDIX C: IC INTERFACING AND SYSTEM DESIGN ISSUES	737
SECTION C.1: OVERVIEW OF IC TECHNOLOGY	738
SECTION C.2: AVR I/O PORT STRUCTURE AND INTERFACING	744
SECTION C.3: SYSTEM DESIGN ISSUES	750
APPENDIX D: FLOWCHARTS AND PSEUDOCODE	755
APPENDIX E: AVR PRIMER FOR 8051 PROGRAMMERS	761

APPENDIX F: ASCII CODES	762
APPENDIX G: ASSEMBLERS, DEVELOPMENT RESOURCES, AND SUPPLIERS	764
APPENDIX H: DATA SHEETS	766
INDEX	771

PREFACE

Products using microprocessors generally fall into two categories. The first category uses high-performance microprocessors such as the Pentium in applications where system performance is critical. We have an entire book dedicated to this topic, *The x86 PC: Assembly Language, Design, and Interfacing*, published by Prentice Hall. In the second category of applications, performance is secondary; issues of cost, space, power, and rapid development are more critical than raw processing power. The microprocessor for this category is often called a *microcontroller*.

This book is for the second category of applications. The AVR is a widely used microcontroller. This book is intended for use in college-level courses teaching microcontrollers and embedded systems. It not only establishes a foundation of Assembly language programming, but also provides a comprehensive treatment of AVR interfacing for engineering students. From this background, the design and interfacing of microcontroller-based embedded systems can be explored. This book can also be used by practicing technicians, hardware engineers, computer scientists, and hobbyists.

Prerequisites

Readers should have had an introductory digital course. Knowledge of Assembly language would be helpful, but is not necessary. Although this book is written for those with no background in Assembly language programming, students with prior Assembly language experience will be able to gain a mastery of AVR architecture very rapidly and start on their projects right away. For the AVR C programming sections of the book, a basic knowledge of C programming is required. We use the AVR Studio compiler IDE from Atmel throughout the book. The AVR Studio compiler is available for free from the Atmel website (www.atmel.com). We encourage you to use the AVR Studio or some other IDE to simulate and run the programs in this book.

Overview

A systematic, step-by-step approach is used to cover various aspects of AVR C and Assembly language programming and interfacing. Many examples and sample programs are given to clarify the concepts and provide students with an opportunity to learn by doing. Review questions are provided at the end of each section to reinforce the main points of the section.

Chapter 0 covers number systems (binary, decimal, and hex), and provides an introduction to basic logic gates and computer memory. This chapter is designed especially for students, such as mechanical engineering students, who have not taken a digital logic course or those who need to refresh their memory on these topics.

Chapter 1 discusses the history of the AVR and features of the members such as ATmega32. It also provides a list of various members of the AVR family.

Chapter 2 discusses the internal architecture of the AVR and explains the use of a AVR assembler to create ready-to-run programs. It also explores the program counter and the flag register.

In Chapter 3 the topics of loop, jump, and call instructions are discussed,

with many programming examples.

Chapter 4 is dedicated to the discussion of I/O ports. This allows students who are working on a project to start experimenting with AVR I/O interfacing and start the hardware project as soon as possible.

Chapter 5 is dedicated to arithmetic, logic instructions, and programs.

Chapter 6 covers the AVR advanced addressing modes and explains how to access the data stored in the look-up table, as well as how to use EEPROM to store data and how to do macros.

The C programming of the AVR is covered in Chapter 7. We use the WinAVR compiler for this and throughout the book. The WinAVR is available for free from the winavr.sourceforge.net website.

In Chapter 8 we discuss the hardware connection of the AVR chip.

Chapter 9 describes the AVR timers and how to use them as event counters.

Chapter 10 provides a detailed discussion of AVR interrupts with many examples on how to write interrupt handler programs.

Chapter 11 is dedicated to serial data communication of the AVR and its interfacing to the RS232. It also shows AVR communication with COM ports of the x86 IBM PC and compatible computers.

Chapter 12 shows AVR interfacing with real-world devices such as LCDs and keyboards.

Chapter 13 shows AVR interfacing with real-world devices such as DAC chips, ADC chips, and sensors.

Chapter 14 covers the basic interfacing of the AVR chip to relays, optoisolators, and stepper motors.

In Chapter 15 we cover how to use AVR timers to generate waves and explain how to capture waves to measure period and duty cycle.

Chapter 16 shows PWM and basic interfacing to DC motors.

Chapter 17 covers the SPI bus protocol and describes how to interface 7-segment displays using MAX7221.

Finally, Chapter 18 shows how to connect and program the DS1307 real-time clock chip using the TWI (I2C) bus protocol.

The appendices have been designed to provide all reference material required for the topics covered in the book. Appendix A describes each AVR instruction in detail, with examples. Appendix A also provides the clock count for instructions and AVR I/O registers. Appendix B describes the basics of wire wrapping. Appendix C examines IC interfacing and logic families, as well as AVR I/O port interfacing and fan-out. Make sure you study this section before connecting the AVR to an external device. In Appendix D, the use of flowcharts and pseudocode is explored. Appendix E is for students familiar with 8051 architectures who need to make a rapid transition to AVR architecture. Appendix F provides the table of ASCII characters. Appendix G lists resources for assembler shareware, AVR trainers, and electronics parts. Appendix H contains data sheets for the AVR chip.

Lab Manual

The lab manual covers some very basic labs and can be found at the www.MicroDigitalEd.com website. The more advanced and rigorous lab assign-

ments are left up to the instructors depending on the course objectives, class level, and whether the course is graduate or undergraduate. The support materials for this text and other books by the authors can be found on this website, too.

Solutions Manual/PowerPoint® Slides

The end-of-chapter problems cover some very basic concepts. The more challenging and rigorous homework assignments are left up to the instructors depending on the course objectives, class level, and whether the course is graduate or undergraduate. The solutions manual and PowerPoint® slides for the drawings are available online for instructors only.

Online Instructor Resources

To access supplementary materials online, instructors need to request an instructor access code. Go to **www.prenhall.com**, click the **Instructor Resource Center** link, and then click **Register Today** for an instructor access code. Within 48 hours after registering you will receive a confirming e-mail including an instructor access code. Once you have received your code, go to the site and log on for full instructions on downloading the materials you wish to use.

Acknowledgments

This book is the result of the dedication and encouragement of many individuals. Our sincere and heartfelt appreciation goes to all of them.

Thanks to the reviewers of this edition:

Orod Haghighi Ara, BIHE University;

Arona Kosari, BIHE University;

Anahita Omidvar, BIHE University;

Vahid Mokhtari, BIHE University;

Farshid Hoori, BIHE University;

Navid HajatDoost, BIHE University;

Hootan Rahmanian, BIHE University;

Farzad Sabeti, BIHE University;

Moshtagh Samandari, BIHE University.

Numerous students found errors or made suggestions in improving this book. We would like to thank all of them for their enthusiasm and support. Those students are: Arash Noori, Soroush Taefi, Golriz Nourani, Mozhdeh Amiri, Negar Ziaee Nasrabadi, and Maryam NouhNezhad, all from the computer engineering department of BIHE.

Finally, we would like to thank the people at Prentice Hall, in particular our editor, Wyatt Morris, who continues to support and encourage our writing, and our project manager, Rex Davidson, who made the book a reality. We were lucky to get the best copy editors in the world, Janice Mazidi and Bret Workman. Thank you both for your fantastic job, as usual.

We enjoyed writing this book, and hope you enjoy reading it and using it for your courses and projects. Please let us know if you have any suggestions or find any errors.

Assemblers/Compilers

The AVR Studio can be downloaded from the following website:

<http://www.Atmel.com>

The WinAVR C compiler for AVR can be downloaded from the following website:

<http://winavr.sourceforge.net>

The tutorials for all the above assemblers/compilers and AVR Trainer boards can be found on the following website:

<http://www.MicroDigitalEd.com>

Trademark Information and Acknowledgments

All the figures, tables, and instructions related to the AVR family of microcontrollers used in this textbook belong to Atmel Corporation. Copyright of Atmel Corporation, Inc. 2009, used by permission.

Instruction mnemonics listed in Appendix A are from Atmel Corporation. Copyright of Atmel Corporation, Inc. 2009, used by permission.

The AVR data sheets listed in Appendix H are from Atmel Semiconductor. Copyright of Atmel Semiconductor, Inc. 2009, used by permission.

ABOUT THE AUTHORS

Muhammad Ali Mazidi went to Tabriz University and holds Master's degrees from both Southern Methodist University and the University of Texas at Dallas. He is currently a.b.d. on his Ph.D. in the Electrical Engineering Department of Southern Methodist University. He is co-author of some widely used textbooks, including *The x86 PC*, *The 8051 Microcontroller and Embedded Systems*, *The PIC Microcontroller and Embedded Systems*, and *The HCS12 Microcontroller and Embedded Systems*, also available from Prentice Hall. He teaches microprocessor-based system design at DeVry University in Dallas, Texas. He is the founder of MicroDigitalEd.com.

Sarmad Naimi graduated from the Computer Engineering department of BIHE university and is currently working on his Master's degree. His areas of interest include FPGA, RTOS, and real-time embedded systems.

Sepehr Naimi graduated from the Computer Engineering department of BIHE university and is currently working on his Master's degree. His areas of interest include high-performance microcontrollers, RTOS, and real-time embedded systems.

The authors can be contacted at the following e-mail addresses if you have any comments or suggestions, or if you find any errors.

mdebooks@yahoo.com

mmazidi@microdigitaled.com

SarmadNaimi@gmail.com

Sepehr.Naimi@gmail.com

CHAPTER 0

INTRODUCTION TO COMPUTING

OBJECTIVES

Upon completion of this chapter, you will be able to:

- >> Convert any number from base 2, base 10, or base 16 to any of the other two bases
- >> Describe the logical operations AND, OR, NOT, XOR, NAND, and NOR
- >> Use logic gates to diagram simple circuits
- >> Explain the difference between a bit, a nibble, a byte, and a word
- >> Give precise mathematical definitions of the terms *kilobyte*, *megabyte*, *gigabyte*, and *terabyte*
- >> Describe the purpose of the major components of a computer system
- >> Contrast and compare various types of semiconductor memories in terms of their capacity, organization, and access time
- >> Describe the relationship between the number of memory locations on a chip, the number of data pins, and the chip's memory capacity
- >> Contrast and compare PROM, EPROM, UV-EPROM, EEPROM, Flash memory EPROM, and mask ROM memories
- >> Contrast and compare SRAM, NV-RAM, and DRAM memories
- >> List the steps a CPU follows in memory address decoding
- >> List the three types of buses found in computers and describe the purpose of each type of bus
- >> Describe the role of the CPU in computer systems
- >> List the major components of the CPU and describe the purpose of each
- >> Understand the RISC and Harvard architectures

To understand the software and hardware of a microcontroller-based system, one must first master some very basic concepts underlying computer architecture. In this chapter (which in the tradition of digital computers is called Chapter 0), the fundamentals of numbering and coding systems are presented in Section 0.1. In Section 0.2, an overview of logic gates is given. The semiconductor memory and memory interfacing are discussed in Section 0.3. In Section 0.4, CPUs and Harvard and von Neumann architectures are discussed. Finally, in the last section we give a brief history of RISC architecture. Although some readers may have an adequate background in many of the topics of this chapter, it is recommended that the material be reviewed, however briefly.

SECTION 0.1: NUMBERING AND CODING SYSTEMS

Whereas human beings use base 10 (*decimal*) arithmetic, computers use the base 2 (*binary*) system. In this section we explain how to convert from the decimal system to the binary system, and vice versa. The convenient representation of binary numbers, called *hexadecimal*, also is covered. Finally, the binary format of the alphanumeric code, called *ASCII*, is explored.

Decimal and binary number systems

Although there has been speculation that the origin of the base 10 system is the fact that human beings have 10 fingers, there is absolutely no speculation about the reason behind the use of the binary system in computers. The binary system is used in computers because 1 and 0 represent the two voltage levels of on and off. Whereas in base 10 there are 10 distinct symbols, 0, 1, 2, ..., 9, in base 2 there are only two, 0 and 1, with which to generate numbers. Base 10 contains digits 0 through 9; binary contains digits 0 and 1 only. These two binary digits, 0 and 1, are commonly referred to as *bits*.

Converting from decimal to binary

One method of converting from decimal to binary is to divide the decimal number by 2 repeatedly, keeping track of the remainders. This process continues until the quotient becomes zero. The remainders are then written in reverse order to obtain the binary number. This is demonstrated in Example 0-1.

Example 0-1

Convert 25_{10} to binary.

Solution:

	Quotient	Remainder	
$25/2 =$	12	1	LSB (least significant bit)
$12/2 =$	6	0	
$6/2 =$	3	0	
$3/2 =$	1	1	
$1/2 =$	0	1	MSB (most significant bit)

Therefore, $25_{10} = 11001_2$.

Converting from binary to decimal

To convert from binary to decimal, it is important to understand the concept of weight associated with each digit position. First, as an analogy, recall the weight of numbers in the base 10 system, as shown in the diagram. By the same token, each digit position of a number in base 2 has a weight associated with it:

740683_{10}	=	
3×10^0	=	3
8×10^1	=	80
6×10^2	=	600
0×10^3	=	0000
4×10^4	=	40000
7×10^5	=	<u>700000</u>
		740683

$$110101_2 =$$

				<i>Decimal</i>	<i>Binary</i>
1×2^0	=	1×1	=	1	1
0×2^1	=	0×2	=	0	00
1×2^2	=	1×4	=	4	100
0×2^3	=	0×8	=	0	0000
1×2^4	=	1×16	=	16	10000
1×2^5	=	1×32	=	<u>32</u>	<u>100000</u>
				53	110101

Knowing the weight of each bit in a binary number makes it simple to add them together to get its decimal equivalent, as shown in Example 0-2.

Example 0-2

Convert 11001_2 to decimal.

Solution:

Weight:	16	8	4	2	1
Digits:	1	1	0	0	1
Sum:	$16 +$	$8 +$	$0 +$	$0 +$	$1 = 25_{10}$

Knowing the weight associated with each binary bit position allows one to convert a decimal number to binary directly instead of going through the process of repeated division. This is shown in Example 0-3.

Example 0-3

Use the concept of weight to convert 39_{10} to binary.

Solution:

Weight:	32	16	8	4	2	1
	1	0	0	1	1	1
	$32 +$	$0 +$	$0 +$	$4 +$	$2 +$	$1 = 39$

Therefore, $39_{10} = 100111_2$.

Hexadecimal system

Base 16, or the *hexadecimal* system as it is called in computer literature, is used as a convenient representation of binary numbers. For example, it is much easier for a human being to represent a string of 0s and 1s such as 100010010110 as its hexadecimal equivalent of 896H. The binary system has 2 digits, 0 and 1. The base 10 system has 10 digits, 0 through 9. The hexadecimal (base 16) system has 16 digits. In base 16, the first 10 digits, 0 to 9, are the same as in decimal, and for the remaining six digits, the letters A, B, C, D, E, and F are used. Table 0-1 shows the equivalent binary, decimal, and hexadecimal representations for 0 to 15.

Converting between binary and hex

To represent a binary number as its equivalent hexadecimal number, start from the right and group 4 bits at a time, replacing each 4-bit binary number with its hex equivalent shown in Table 0-1. To convert from hex to binary, each hex digit is replaced with its 4-bit binary equivalent. See Examples 0-4 and 0-5.

**Table 0-1: Base 16
Number System**

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Example 0-4

Represent binary 100111110101 in hex.

Solution:

First the number is grouped into sets of 4 bits: 1001 1111 0101.

Then each group of 4 bits is replaced with its hex equivalent:

1001	1111	0101
9	F	5

Therefore, $100111110101_2 = 9F5_{16}$ hexadecimal.

Example 0-5

Convert hex 29B to binary.

Solution:

	2	9	B
29B	=	0010	1001 1011

Dropping the leading zeros gives 1010011011.

Converting from decimal to hex

Converting from decimal to hex could be approached in two ways:

1. Convert to binary first and then convert to hex. Example 0-6 shows this method of converting decimal to hex.
2. Convert directly from decimal to hex by repeated division, keeping track of the remainders. Experimenting with this method is left to the reader.

Example 0-6(a) Convert 45_{10} to hex.

<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>	First, convert to binary.
1	0	1	1	0	1	$32 + 8 + 4 + 1 = 45$

$$45_{10} = 0010\ 1101_2 = 2D\ \text{hex}$$

(b) Convert 629_{10} to hex.

<u>512</u>	<u>256</u>	<u>128</u>	<u>64</u>	<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>
1	0	0	1	1	1	0	1	0	1

$$629_{10} = (512 + 64 + 32 + 16 + 4 + 1) = 0010\ 0111\ 0101_2 = 275\ \text{hex}$$

(c) Convert 1714_{10} to hex.

<u>1024</u>	<u>512</u>	<u>256</u>	<u>128</u>	<u>64</u>	<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>
1	1	0	1	0	1	1	0	0	1	0

$$1714_{10} = (1024 + 512 + 128 + 32 + 16 + 2) = 0110\ 1011\ 0010_2 = 6B2\ \text{hex}$$

Converting from hex to decimal

Conversion from hex to decimal can also be approached in two ways:

1. Convert from hex to binary and then to decimal. Example 0-7 demonstrates this method of converting from hex to decimal.
2. Convert directly from hex to decimal by summing the weight of all digits.

Example 0-7

Convert the following hexadecimal numbers to decimal.

(a) $6B2_{16} = 0110\ 1011\ 0010_2$

<u>1024</u>	<u>512</u>	<u>256</u>	<u>128</u>	<u>64</u>	<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>
1	1	0	1	0	1	1	0	0	1	0

$$1024 + 512 + 128 + 32 + 16 + 2 = 1714_{10}$$

(b) $9F2D_{16} = 1001\ 1111\ 0010\ 1101_2$

<u>32768</u>	<u>16384</u>	<u>8192</u>	<u>4096</u>	<u>2048</u>	<u>1024</u>	<u>512</u>	<u>256</u>	<u>128</u>	<u>64</u>	<u>32</u>	<u>16</u>	<u>8</u>	<u>4</u>	<u>2</u>	<u>1</u>
1	0	0	1	1	1	1	1	0	0	1	0	1	1	0	1

$$32768 + 4096 + 2048 + 1024 + 512 + 256 + 32 + 8 + 4 + 1 = 40,749_{10}$$

Table 0-2: Counting in Bases

Decimal	Binary	Hex
0	00000	0
1	00001	1
2	00010	2
3	00011	3
4	00100	4
5	00101	5
6	00110	6
7	00111	7
8	01000	8
9	01001	9
10	01010	A
11	01011	B
12	01100	C
13	01101	D
14	01110	E
15	01111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1A
27	11011	1B
28	11100	1C
29	11101	1D
30	11110	1E
31	11111	1F

Counting in bases 10, 2, and 16

To show the relationship between all three bases, in Table 0-2 we show the sequence of numbers from 0 to 31 in decimal, along with the equivalent binary and hex numbers. Notice in each base that when one more is added to the highest digit, that digit becomes zero and a 1 is carried to the next-highest digit position. For example, in decimal, $9 + 1 = 0$ with a carry to the next-highest position. In binary, $1 + 1 = 0$ with a carry; similarly, in hex, $F + 1 = 0$ with a carry.

Addition of binary and hex numbers

The addition of binary numbers is a very straightforward process. Table 0-3 shows the addition of two bits. The discussion of subtraction of binary numbers is bypassed since all computers use the addition process to implement subtraction. Although

Table 0-3: Binary Addition

A + B	Carry	Sum
0 + 0	0	0
0 + 1	0	1
1 + 0	0	1
1 + 1	1	0

computers have adder circuitry, there is no separate circuitry for subtractors. Instead, adders are used in conjunction with *2's complement* circuitry to perform subtraction. In other words, to implement " $x - y$ ", the computer takes the 2's complement of y and adds it to x . The concept of 2's complement is reviewed next. Example 0-8 shows the addition of binary numbers.

Example 0-8

Add the following binary numbers. Check against their decimal equivalents.

Solution:

	<i>Binary</i>	<i>Decimal</i>
	1101	13
+	1001	9
	10110	22

2's complement

To get the 2's complement of a binary number, invert all the bits and then

add 1 to the result. Inverting the bits is simply a matter of changing all 0s to 1s and 1s to 0s. This is called the *1's complement*. See Example 0-9.

Example 0-9

Take the 2's complement of 10011101.

Solution:

$$\begin{array}{rcl}
 & 10011101 & \text{binary number} \\
 & 01100010 & \text{1's complement} \\
 + & \underline{1} & \\
 & 01100011 & \text{2's complement}
 \end{array}$$

Addition and subtraction of hex numbers

In studying issues related to software and hardware of computers, it is often necessary to add or subtract hex numbers. Mastery of these techniques is essential. Hex addition and subtraction are discussed separately below.

Addition of hex numbers

This section describes the process of adding hex numbers. Starting with the least significant digits, the digits are added together. If the result is less than 16, write that digit as the sum for that position. If it is greater than 16, subtract 16 from it to get the digit and carry 1 to the next digit. The best way to explain this is by example, as shown in Example 0-10.

Example 0-10

Perform hex addition: 23D9 + 94BE.

Solution:

$$\begin{array}{rcl}
 & 23D9 & \text{LSD: } 9 + 14 = 23 \qquad 23 - 16 = 7 \text{ with a carry} \\
 + & \underline{94BE} & \qquad 1 + 13 + 11 = 25 \qquad 25 - 16 = 9 \text{ with a carry} \\
 & B897 & \qquad 1 + 3 + 4 = 8 \\
 & & \text{MSD: } 2 + 9 = B
 \end{array}$$

Subtraction of hex numbers

In subtracting two hex numbers, if the second digit is greater than the first, borrow 16 from the preceding digit. See Example 0-11.

Example 0-11

Perform hex subtraction: 59F – 2B8.

Solution:

$$\begin{array}{rcl}
 & 59F & \text{LSD: } 8 \text{ from } 15 = 7 \\
 - & \underline{2B8} & \qquad 11 \text{ from } 25 (9 + 16) = 14 (E) \\
 & 2E7 & \qquad 2 \text{ from } 4 (5 - 1) = 2
 \end{array}$$

ASCII code

The discussion so far has revolved around the representation of number systems. Because all information in the computer must be represented by 0s and 1s, binary patterns must be assigned to letters and other characters. In the 1960s a standard representation called *ASCII* (American Standard Code for Information Interchange) was established.

The ASCII (pronounced “ask-E”) code assigns binary patterns for numbers 0 to 9, all the letters of the English alphabet, both uppercase (capital) and lowercase, and many control codes and punctuation marks. The great advantage of this system is that it is used by most computers, so that information can be shared among computers. The ASCII system uses a total of 7 bits to represent each code. For example, 100 0001 is assigned to the uppercase letter “A” and 110 0001 is for the lowercase “a”. Often, a zero is placed in the most-significant bit position to make it an 8-bit code. Figure 0-1 shows selected ASCII codes. A complete list of ASCII codes is given in Appendix F. The use of ASCII is not only standard for keyboards used in the United States and many other countries but also provides a standard for printing and displaying characters by output devices such as printers and monitors.

Notice that the pattern of ASCII codes was designed to allow for easy manipulation of ASCII data. For example, digits 0 through 9 are represented by ASCII codes 30 through 39. This enables a program to easily convert ASCII to decimal by masking off the “3” in the upper nibble. Also notice that there is a relationship between the uppercase and lowercase letters. The uppercase letters are represented by ASCII codes 41 through 5A while lowercase letters are represented by codes 61 through 7A. Looking at the binary code, the only bit that is different between the uppercase “A” and lowercase “a” is bit 5. Therefore, conversion between uppercase and lowercase is as simple as changing bit 5 of the ASCII code.

Hex	Symbol	Hex	Symbol
41	A	61	a
42	B	62	b
43	C	63	c
44	D	64	d
...
59	Y	79	y
5A	Z	7A	z

Figure 0-1. Selected ASCII Codes

Review Questions

1. Why do computers use the binary number system instead of the decimal system?
2. Convert 34_{10} to binary and hex.
3. Convert 110101_2 to hex and decimal.
4. Perform binary addition: $101100 + 101$.
5. Convert 101100_2 to its 2's complement representation.
6. Add $36BH + F6H$.
7. Subtract $36BH - F6H$.
8. Write “80x86 CPUs” in its ASCII code (in hex form).

SECTION 0.2: DIGITAL PRIMER

This section gives an overview of digital logic and design. First, we cover binary logic operations, then we show gates that perform these functions. Next, logic gates are put together to form simple digital circuits. Finally, we cover some logic devices commonly found in microcontroller interfacing.

Binary logic

As mentioned earlier, computers use the binary number system because the two voltage levels can be represented as the two digits 0 and 1. Signals in digital electronics have two distinct voltage levels. For example, a system may define 0 V as logic 0 and +5 V as logic 1. Figure 0-2 shows this system with the built-in tolerances for variations in the voltage. A valid digital signal in this example should be within either of the two shaded areas.

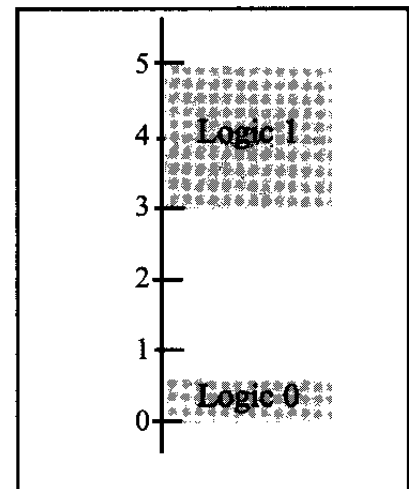


Figure 0-2. Binary Signals

Logic gates

Binary logic gates are simple circuits that take one or more input signals and send out one output signal. Several of these gates are defined below.

AND gate

The AND gate takes two or more inputs and performs a logic AND on them. See the truth table and diagram of the AND gate. Notice that if both inputs to the AND gate are 1, the output will be 1. Any other combination of inputs will give a 0 output. The example shows two inputs, x and y . Multiple outputs are also possible for logic gates. In the case of AND, if all inputs are 1, the output is 1. If any input is 0, the output is 0.

OR gate

The OR logic function will output a 1 if one or more inputs is 1. If all inputs are 0, then and only then will the output be 0.

Tri-state buffer

A buffer gate does not change the logic level of the input. It is used to isolate or amplify the signal.

Logical AND Function

Inputs		Output
X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

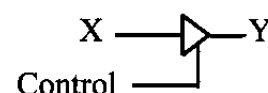


Logical OR Function

Inputs		Output
X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1



Buffer



Inverter

The inverter, also called NOT, outputs the value opposite to that input to the gate. That is, a 1 input will give a 0 output, while a 0 input will give a 1 output.

XOR gate

The XOR gate performs an exclusive-OR operation on the inputs. Exclusive-OR produces a 1 output if one (but only one) input is 1. If both operands are 0, the output is 0. Likewise, if both operands are 1, the output is also 0. Notice from the XOR truth table, that whenever the two inputs are the same, the output is 0. This function can be used to compare two bits to see if they are the same.

NAND and NOR gates

The NAND gate functions like an AND gate with an inverter on the output. It produces a 0 output when all inputs are 1; otherwise, it produces a 1 output. The NOR gate functions like an OR gate with an inverter on the output. It produces a 1 if all inputs are 0; otherwise, it produces a 0. NAND and NOR gates are used extensively in digital design because they are easy and inexpensive to fabricate. Any circuit that can be designed with AND, OR, XOR, and INVERTER gates can be implemented using only NAND and NOR gates. A simple example of this is given below. Notice in NAND, that if any input is 0, the output is 1. Notice in NOR, that if any input is 1, the output is 0.

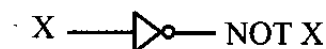
Logic design using gates

Next we will show a simple logic design to add two binary digits. If we add two binary digits there are four possible outcomes:

	Carry	Sum
0 + 0 =	0	0
0 + 1 =	0	1
1 + 0 =	0	1
1 + 1 =	1	0

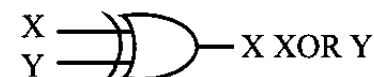
Logical Inverter

Input	Output
X	NOT X
0	1
1	0



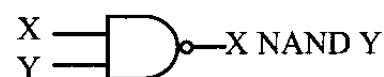
Logical XOR Function

Inputs	Output
X Y	X XOR Y
0 0	0
0 1	1
1 0	1
1 1	0



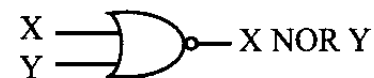
Logical NAND Function

Inputs	Output
X Y	X NAND Y
0 0	1
0 1	1
1 0	1
1 1	0



Logical NOR Function

Inputs	Output
X Y	X NOR Y
0 0	1
0 1	0
1 0	0
1 1	0



Notice that when we add $1 + 1$ we get 0 with a carry to the next higher place. We will need to determine the sum and the carry for this design. Notice that the sum column above matches the output for the XOR function, and that the carry column matches the output for the AND function. Figure 0-3(a) shows a simple adder implemented with XOR and AND gates. Figure 0-3(b) shows the same logic circuit implemented with AND and OR gates and inverters.

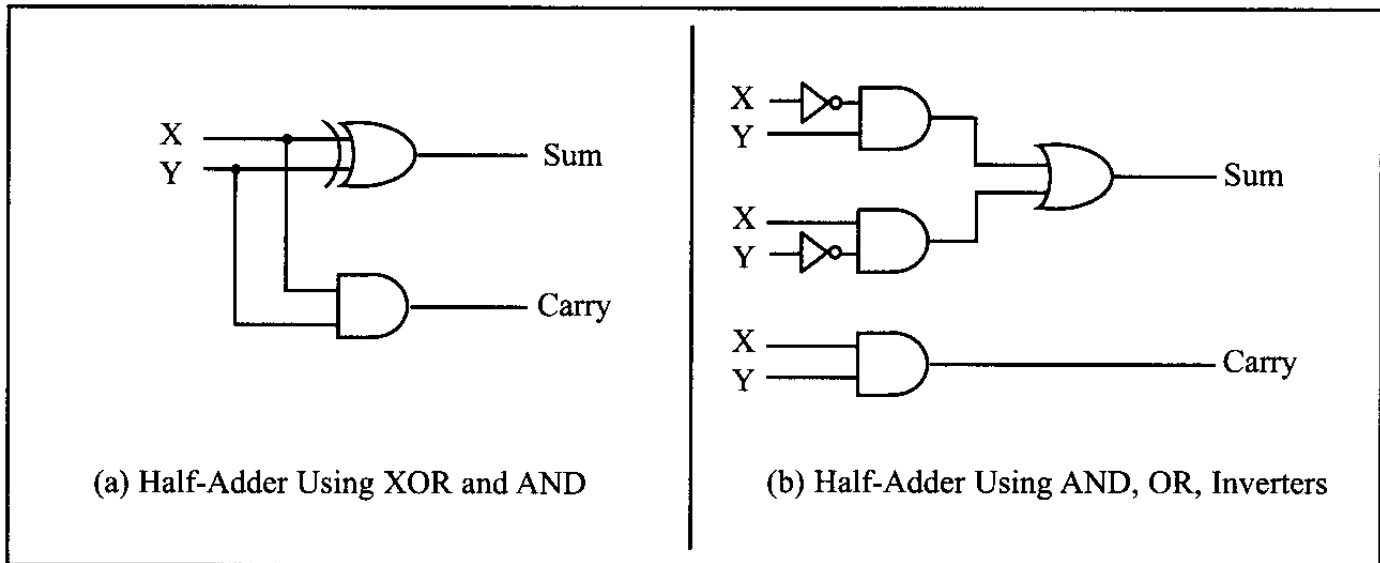


Figure 0-3. Two Implementations of a Half-Adder

Figure 0-4 shows a block diagram of a half-adder. Two half-adders can be combined to form an adder that can add three input digits. This is called a full-adder. Figure 0-5 shows the logic diagram of a full-adder, along with a block diagram that masks the details of the circuit. Figure 0-6 shows a 3-bit adder using three full-adders.

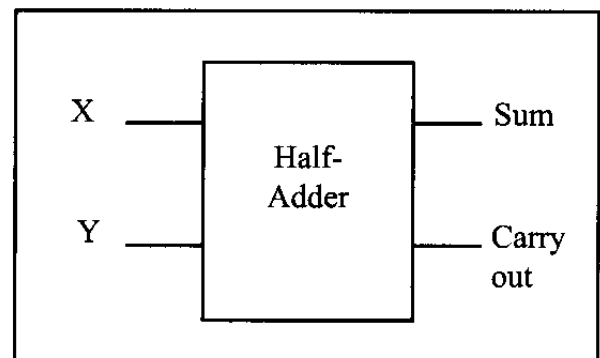


Figure 0-4. Block Diagram of a Half-Adder

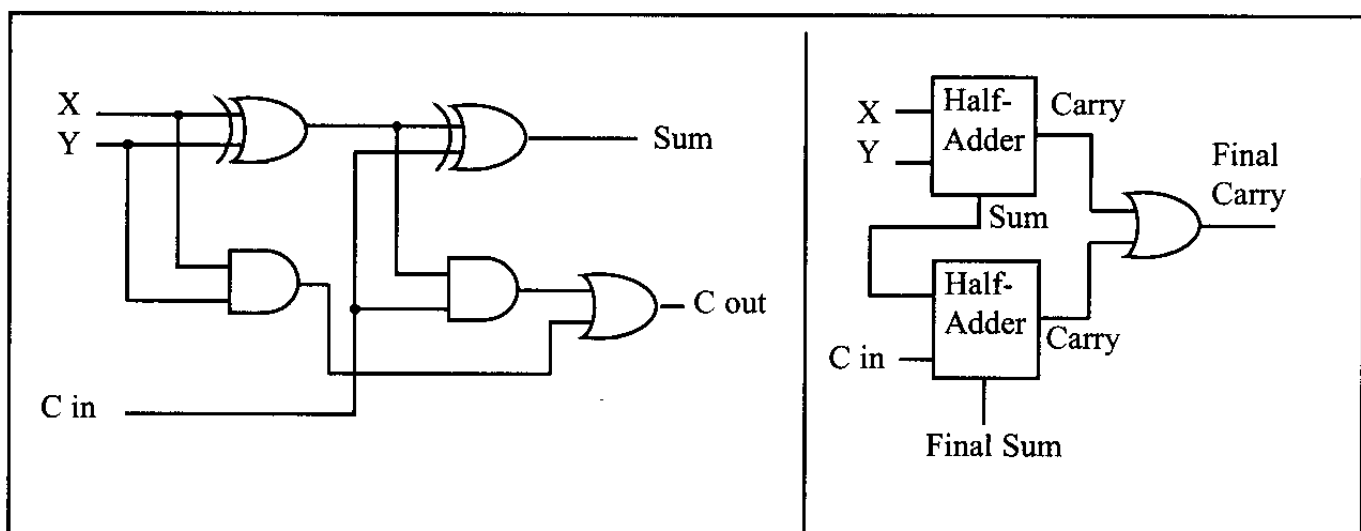


Figure 0-5. Full-Adder Built from a Half-Adder

Decoders

Another example of the application of logic gates is the decoder. Decoders are widely used for address decoding in computer design. Figure 0-7 shows decoders for 9 (1001 binary) and 5 (0101) using inverters and AND gates.

Flip-flops

A widely used component in digital systems is the flip-flop. Frequently, flip-flops are used to store data. Figure 0-8 shows the logic diagram, block diagram, and truth table for a flip-flop.

The D flip-flop is widely used to latch data. Notice from the truth table that a D-FF grabs the data at the input as the clock is activated. A D-FF holds the data as long as the power is on.

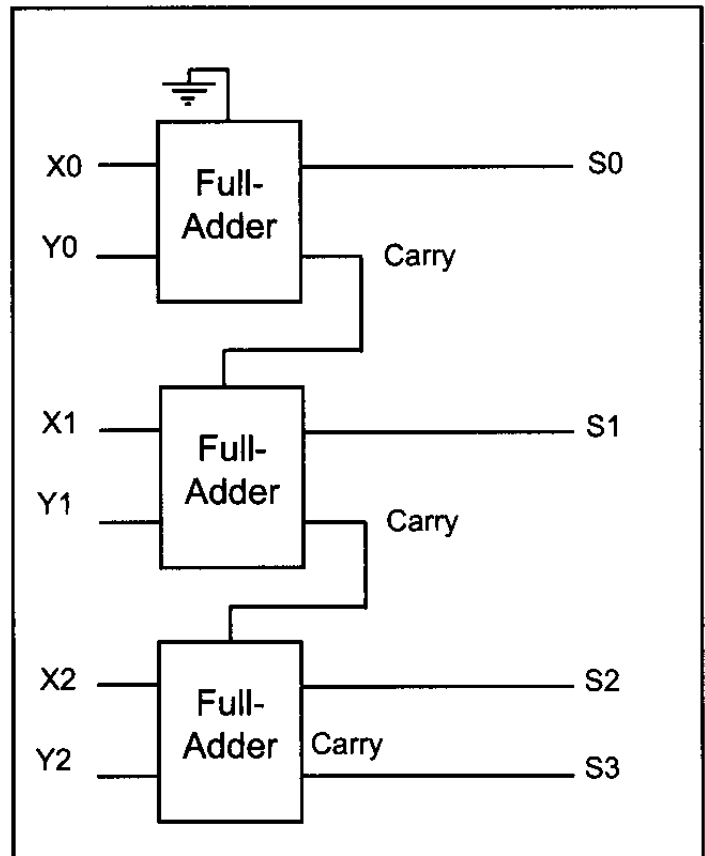


Figure 0-6. 3-Bit Adder Using Three Full-Adders

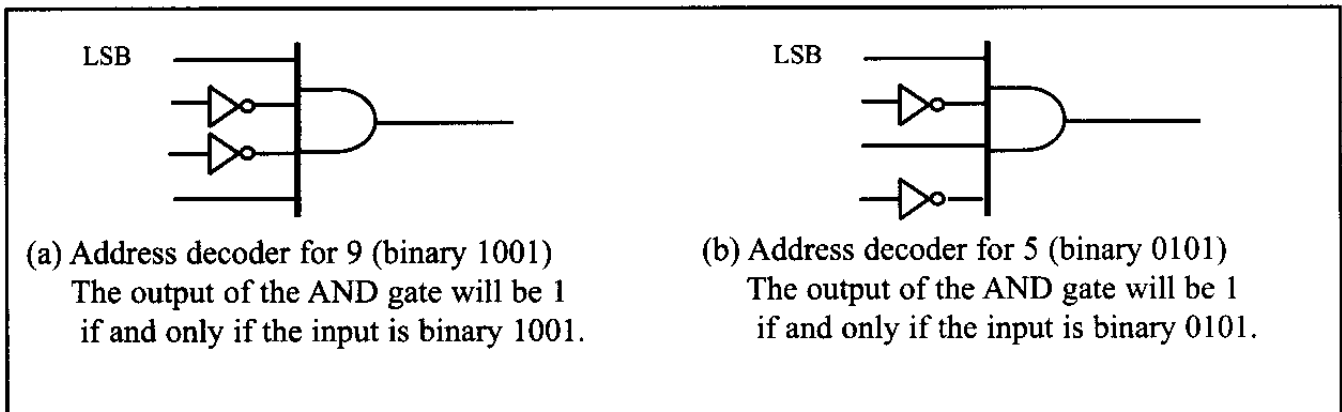


Figure 0-7. Address Decoders

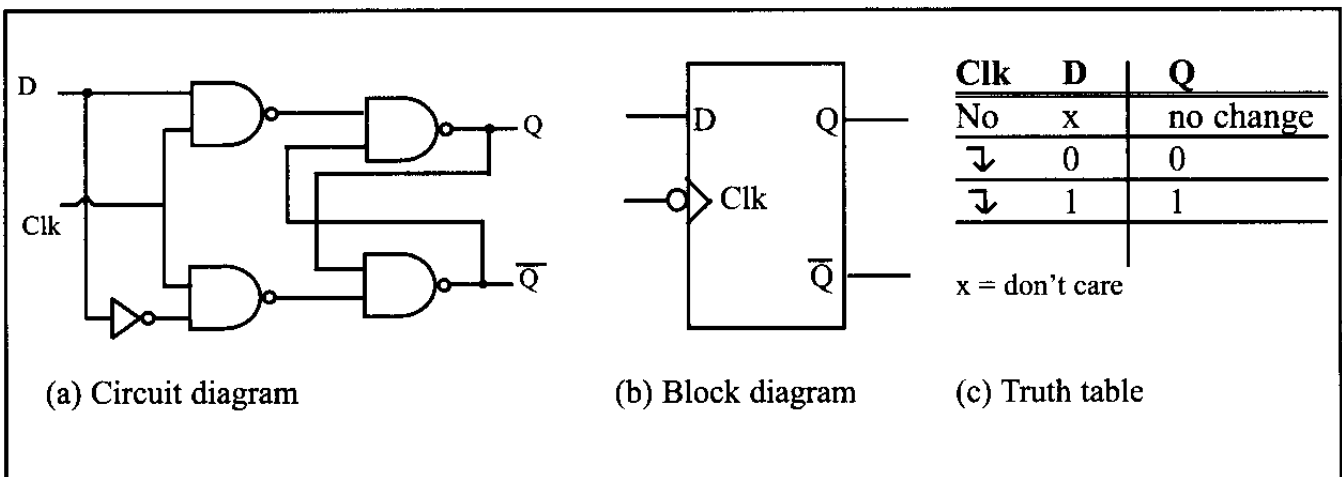


Figure 0-8. D Flip-Flops

Review Questions

1. The logical operation _____ gives a 1 output when all inputs are 1.
2. The logical operation _____ gives a 1 output when one or more of its inputs is 1.
3. The logical operation _____ is often used to compare two inputs to determine whether they have the same value.
4. A _____ gate does not change the logic level of the input.
5. Name a common use for flip-flops.
6. An address _____ is used to identify a predetermined binary address.

SECTION 0.3: SEMICONDUCTOR MEMORY

In this section we discuss various types of semiconductor memories and their characteristics such as capacity, organization, and access time. We will also show how the memory is connected to CPU. Before we embark on the subject of memory, it will be helpful to give an overview of computer organization and review some widely used terminology in computer literature.

Some important terminology

Recall from the discussion above that a *bit* is a binary digit that can have the value 0 or 1. A *byte* is defined as 8 bits. A *nibble* is half a byte, or 4 bits. A *word* is

two bytes, or 16 bits. The display is intended to show the relative size of these units. Of course, they could all be composed of any combination of zeros and ones.

Bit	0
Nibble	0000
Byte	0000 0000
Word	0000 0000 0000 0000

A *kilobyte* is 2^{10} bytes, which is 1024 bytes. The abbreviation K is often used to represent kilobytes. A *megabyte*, or *meg* as some call it, is 2^{20} bytes. That is a little over 1 million bytes; it is exactly 1,048,576 bytes. Moving rapidly up the scale in size, a *gigabyte* is 2^{30} bytes (over 1 billion), and a *terabyte* is 2^{40} bytes (over 1 trillion). As an example of how some of these terms are used, suppose that a given computer has 16 megabytes of memory. That would be 16×2^{20} , or $2^4 \times 2^{20}$, which is 2^{24} . Therefore 16 megabytes is 2^{24} bytes.

Two types of memory commonly used in microcomputers are *RAM*, which stands for “random access memory” (sometimes called *read/write memory*), and *ROM*, which stands for “read-only memory.” RAM is used by the computer for temporary storage of programs that it is running. That data is lost when the computer is turned off. For this reason, RAM is sometimes called *volatile memory*. ROM contains programs and information essential to operation of the computer. The information in ROM is permanent, cannot be changed by the user, and is not lost when the power is turned off. Therefore, it is called *nonvolatile memory*.

Internal organization of computers

The internal working of every computer can be broken down into three parts: CPU (central processing unit), memory, and I/O (input/output) devices. Figure 0-9 shows a block diagram of the internal organization of a computer.

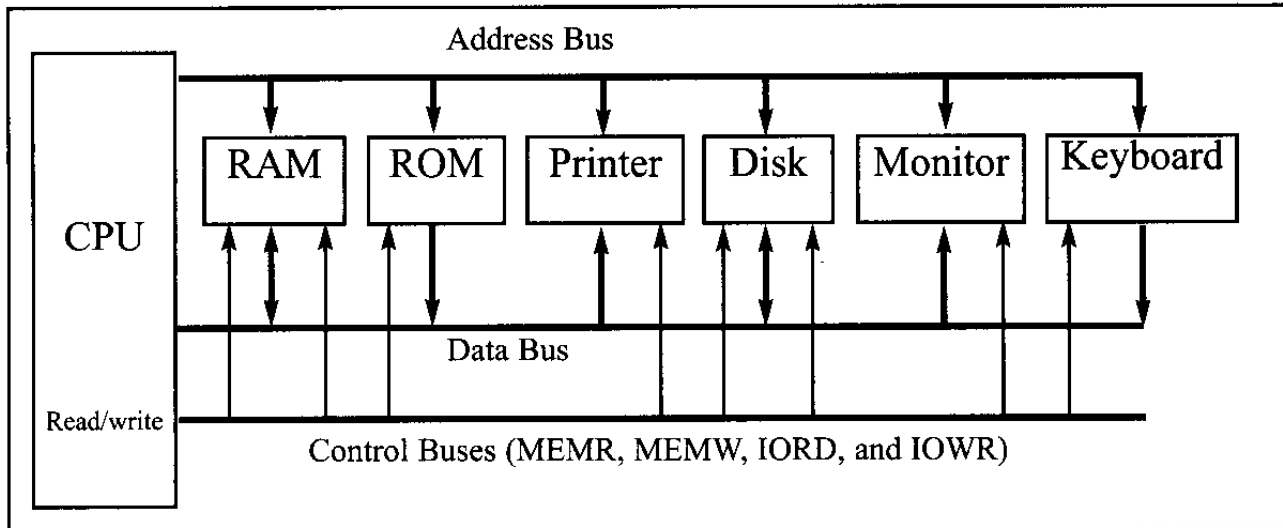


Figure 0-9. Internal Organization of a Computer

The function of the CPU is to execute (process) information stored in memory. The function of I/O devices such as the keyboard and video monitor is to provide a means of communicating with the CPU. The CPU is connected to memory and I/O through strips of wire called a *bus*. The bus inside a computer allows carrying information from place to place just as a street allows cars to carry people from place to place. In every computer there are three types of buses: address bus, data bus, and control bus.

For a device (memory or I/O) to be recognized by the CPU, it must be assigned an address. The address assigned to a given device must be unique; no two devices are allowed to have the same address. The CPU puts the address (in binary, of course) on the address bus, and the decoding circuitry finds the device. Then the CPU uses the data bus either to get data from that device or to send data to it. The control buses are used to provide read or write signals to the device to indicate if the CPU is asking for information or sending information. Of the three buses, the address bus and data bus determine the capability of a given CPU.

More about the data bus

Because data buses are used to carry information in and out of a CPU, the more data buses available, the better the CPU. If one thinks of data buses as highway lanes, it is clear that more lanes provide a better pathway between the CPU and its external devices (such as printers, RAM, ROM, etc.; see Figure 0-9). By the same token, that increase in the number of lanes increases the cost of construction. More data buses mean a more expensive CPU and computer. The average size of data buses in CPUs varies between 8 and 64 bits. Early personal computers such as Apple 2 used an 8-bit data bus, while supercomputers such as Cray used a 64-bit data bus. Data buses are bidirectional, because the CPU must use them either to receive or to send data. The processing power of a computer is related to the size of its buses, because an 8-bit bus can send out 1 byte at a time, but a 16-bit bus can send out 2 bytes at a time, which is twice as fast.

More about the address bus

Because the address bus is used to identify the devices and memory connected to the CPU, the more address buses available, the larger the number of

devices that can be addressed. In other words, the number of address buses for a CPU determines the number of locations with which it can communicate. The number of locations is always equal to 2^x , where x is the number of address lines, regardless of the size of the data bus. For example, a CPU with 16 address lines can provide a total of 65,536 (2^{16}) or 64K of addressable memory. Each location can have a maximum of 1 byte of data. This is because all general-purpose microprocessor CPUs are what is called *byte addressable*. As another example, the IBM PC AT uses a CPU with 24 address lines and 16 data lines. Thus, the total accessible memory is 16 megabytes ($2^{24} = 16$ megabytes). In this example there would be 2^{24} locations, and because each location is one byte, there would be 16 megabytes of memory. The address bus is a *unidirectional* bus, which means that the CPU uses the address bus only to send out addresses. To summarize: The total number of memory locations addressable by a given CPU is always equal to 2^x where x is the number of address bits, regardless of the size of the data bus.

CPU and its relation to RAM and ROM

For the CPU to process information, the data must be stored in RAM or ROM. The function of ROM in computers is to provide information that is fixed and permanent. This is information such as tables for character patterns to be displayed on the video monitor, or programs that are essential to the working of the computer, such as programs for testing and finding the total amount of RAM installed on the system, or for displaying information on the video monitor. In contrast, RAM stores temporary information that can change with time, such as various versions of the operating system and application packages such as word processing or tax calculation packages. These programs are loaded from the hard drive into RAM to be processed by the CPU. The CPU cannot get the information from the disk directly because the disk is too slow. In other words, the CPU first seeks the information to be processed from RAM (or ROM). Only if the data is not there does the CPU seek it from a mass storage device such as a disk, and then it transfers the information to RAM. For this reason, RAM and ROM are sometimes referred to as *primary memory* and disks are called *secondary memory*. Next, we discuss various types of semiconductor memories and their characteristics such as capacity, organization, and access time.

Memory capacity

The number of bits that a semiconductor memory chip can store is called *chip capacity*. It can be in units of Kbits (kilobits), Mbits (megabits), and so on. This must be distinguished from the storage capacity of computer systems. While the memory capacity of a memory IC chip is always given in bits, the memory capacity of a computer system is given in bytes. For example, an article in a technical journal may state that the 128M chip has become popular. In that case, it is understood, although it is not mentioned, that 128M means 128 megabits since the article is referring to an IC memory chip. However, if an advertisement states that a computer comes with 128M memory, it is understood that 128M means 128 megabytes since it is referring to a computer system.

Memory organization

Memory chips are organized into a number of locations within the IC. Each location can hold 1 bit, 4 bits, 8 bits, or even 16 bits, depending on how it is designed internally. The number of bits that each location within the memory chip can hold is always equal to the number of data pins on the chip. How many locations exist inside a memory chip? That depends on the number of address pins. The number of locations within a memory IC always equals 2 to the power of the number of address pins. Therefore, the total number of bits that a memory chip can store is equal to the number of locations times the number of data bits per location. To summarize:

1. A memory chip contains 2^x locations, where x is the number of address pins.
2. Each location contains y bits, where y is the number of data pins on the chip.
3. The entire chip will contain $2^x \times y$ bits, where x is the number of address pins and y is the number of data pins on the chip.

Table 0-4: Powers of 2

x	2^x
10	1K
11	2K
12	4K
13	8K
14	16K
15	32K
16	64K
17	128K
18	256K
19	512K
20	1M
21	2M
22	4M
23	8M
24	16M
25	32M
26	64M
27	128M

Speed

One of the most important characteristics of a memory chip is the speed at which its data can be accessed. To access the data, the address is presented to the address pins, the READ pin is activated, and after a certain amount of time has elapsed, the data shows up at the data pins. The shorter this elapsed time, the better, and consequently, the more expensive the memory chip. The speed of the memory chip is commonly referred to as its *access time*. The access time of memory chips varies from a few nanoseconds to hundreds of nanoseconds, depending on the IC technology used in the design and fabrication process.

The three important memory characteristics of capacity, organization, and access time will be explored extensively in this chapter. Table 0-4 serves as a reference for the calculation of memory organization. Examples 0-12 and 0-13 demonstrate these concepts.

ROM (read-only memory)

ROM is a type of memory that does not lose its contents when the power is turned off. For this reason, ROM is also called *non-volatile* memory. There are different types of read-only memory, such as PROM, EPROM, EEPROM, Flash EPROM, and mask ROM. Each is explained next.

PROM (programmable ROM) and OTP

PROM refers to the kind of ROM that the user can burn information into. In other words, PROM is a user-programmable memory. For every bit of the PROM, there exists a fuse. PROM is programmed by blowing the fuses. If the information burned into PROM is wrong, that PROM must be discarded since its internal fuses are blown permanently. For this reason, PROM is also referred to as

Example 0-12

A given memory chip has 12 address pins and 4 data pins. Find:

(a) the organization, and (b) the capacity.

Solution:

- (a) This memory chip has 4,096 locations ($2^{12} = 4,096$), and each location can hold 4 bits of data. This gives an organization of $4,096 \times 4$, often represented as $4K \times 4$.
- (b) The capacity is equal to 16K bits since there is a total of 4K locations and each location can hold 4 bits of data.

Example 0-13

A 512K memory chip has 8 pins for data. Find:

(a) the organization, and (b) the number of address pins for this memory chip.

Solution:

- (a) A memory chip with 8 data pins means that each location within the chip can hold 8 bits of data. To find the number of locations within this memory chip, divide the capacity by the number of data pins. $512K/8 = 64K$; therefore, the organization for this memory chip is $64K \times 8$.
- (b) The chip has 16 address lines since $2^{16} = 64K$.

OTP (one-time programmable). Programming ROM, also called *burning* ROM, requires special equipment called a ROM burner or ROM programmer.

EPROM (erasable programmable ROM) and UV-EPROM

EPROM was invented to allow making changes in the contents of PROM after it is burned. In EPROM, one can program the memory chip and erase it thousands of times. This is especially necessary during development of the prototype of a microprocessor-based project. A widely used EPROM is called UV-EPROM, where UV stands for ultraviolet. The only problem with UV-EPROM is that erasing its contents can take up to 20 minutes. All UV-EPROM chips have a window through which the programmer can shine ultraviolet (UV) radiation to erase the chip's contents. For this reason, EPROM is also referred to as UV-erasable EPROM or simply UV-EPROM. Figure 0-10 shows the pins for UV-EPROM chips.

To program a UV-EPROM chip, the following steps must be taken:

1. Its contents must be erased. To erase a chip, remove it from its socket on the system board and place it in EPROM erasure equipment to expose it to UV radiation for 15–20 minutes.
2. Program the chip. To program a UV-EPROM chip, place it in the ROM burner (programmer). To burn code or data into EPROM, the ROM burner uses 12.5 volts or higher, depending on the EPROM type. This voltage is referred

to as V_{PP} in the UV-EPROM data sheet.

- Place the chip back into its socket on the system board.

As can be seen from the above steps, not only is there an EPROM programmer (burner), but there is also separate EPROM erasure equipment. The main problem, and indeed the major disadvantage of UV-EPROM, is that it cannot be erased and programmed while it is in the system board. To provide a solution to this problem, EEPROM was invented.

Notice the patterns of the IC numbers in Table 0-5. For example, part number 27128-25 refers to UV-EPROM that has a capacity of 128K bits and access time of 250 nanoseconds. The capacity of the memory chip is indicated in the part number and the access time is given with a zero dropped. See Example 0-14. In part numbers, C refers to CMOS technology. Notice that 27XX always refers to UV-EPROM chips. For a comprehensive list of available memory chips see the JAMECO (jameco.com) or JDR (jdr.com) catalogs.

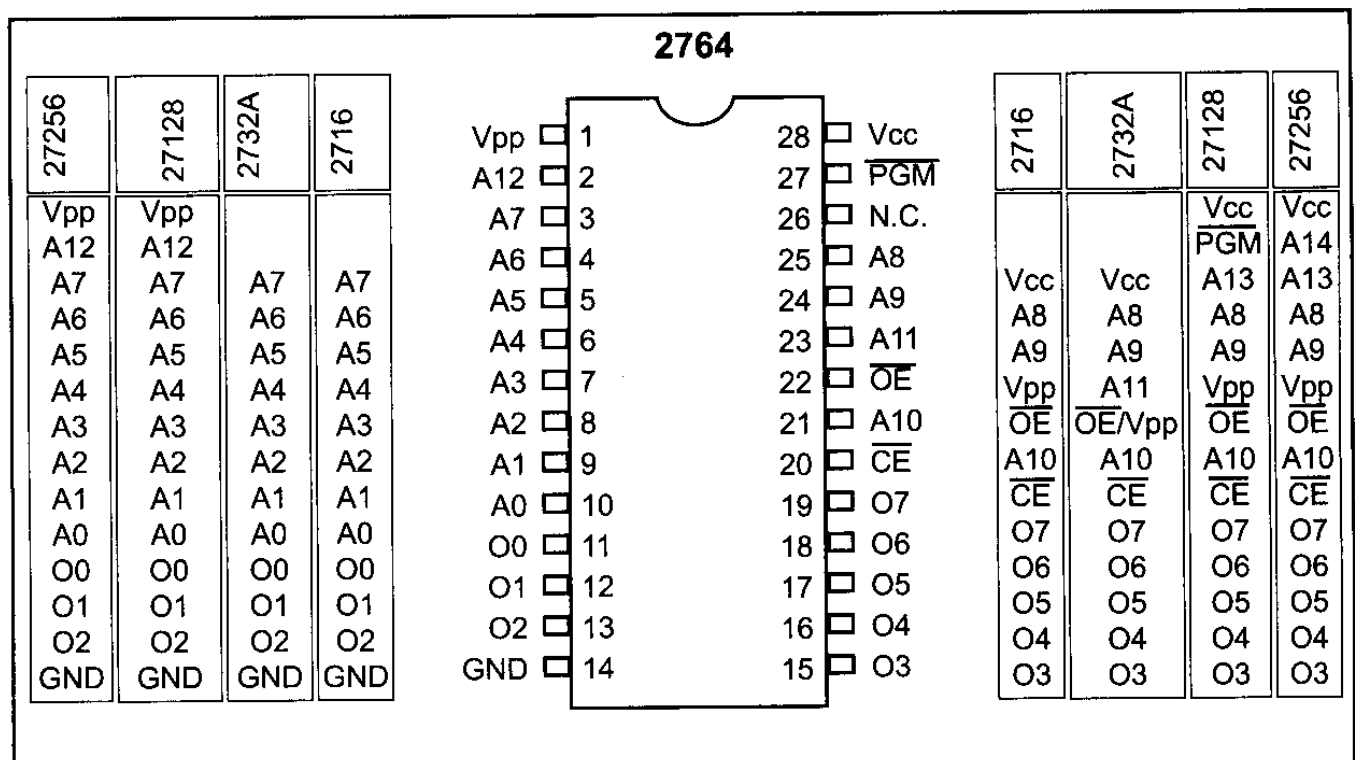


Figure 0-10. Pin Configurations for 27xx ROM Family

Example 0-14

For ROM chip 27128, find the number of data and address pins.

Solution:

The 27128 has a capacity of 128K bits. It has $16K \times 8$ organization (all ROMs have 8 data pins), which indicates that there are 8 pins for data and 14 pins for address ($2^{14} = 16K$).

Table 0-5: Some UV-EPROM Chips

Part #	Capacity	Org.	Access	Pins	V _{pp}
2716	16K	2K × 8	450 ns	24	25 V
2732	32K	4K × 8	450 ns	24	25 V
2732A-20	32K	4K × 8	200 ns	24	21 V
27C32-1	32K	4K × 8	450 ns	24	12.5 V CMOS
2764-20	64K	8K × 8	200 ns	28	21 V
2764A-20	64K	8K × 8	200 ns	28	12.5 V
27C64-12	64K	8K × 8	120 ns	28	12.5 V CMOS
27128-25	128K	16K × 8	250 ns	28	21 V
27C128-12	128K	16K × 8	120 ns	28	12.5 V CMOS
27256-25	256K	32K × 8	250 ns	28	12.5 V
27C256-15	256K	32K × 8	150 ns	28	12.5 V CMOS
27512-25	512K	64K × 8	250 ns	28	12.5 V
27C512-15	512K	64K × 8	150 ns	28	12.5 V CMOS
27C010-15	1024K	128K × 8	150 ns	32	12.5 V CMOS
27C020-15	2048K	256K × 8	150 ns	32	12.5 V CMOS
27C040-15	4096K	512K × 8	150 ns	32	12.5 V CMOS

EEPROM (electrically erasable programmable ROM)

EEPROM has several advantages over EPROM, such as the fact that its method of erasure is electrical and therefore instant, as opposed to the 20-minute erasure time required for UV-EPROM. In addition, in EEPROM one can select which byte to be erased, in contrast to UV-EPROM, in which the entire contents of ROM are erased. However, the main advantage of EEPROM is that one can program and erase its contents while it is still in the system board. It does not require physical removal of the memory chip from its socket. In other words, unlike UV-EPROM, EEPROM does not require an external erasure and programming device. To utilize EEPROM fully, the designer must incorporate the circuitry to program the EEPROM into the system board. In general, the cost per bit for EEPROM is much higher than for UV-EPROM.

Flash memory EPROM

Since the early 1990s, Flash EPROM has become a popular user-programmable memory chip, and for good reasons. First, the erasure of the entire contents takes less than a second, or one might say in a flash, hence its name, Flash memory. In addition, the erasure method is electrical, and for this reason it is sometimes referred to as Flash EEPROM. To avoid confusion, it is commonly called Flash memory. The major difference between EEPROM and Flash memory is that when Flash memory's contents are erased, the entire device is erased, in contrast to EEPROM, where one can erase a desired byte. Although in many Flash memories recently made available the contents are divided into blocks and the erasure can be done block by block, unlike EEPROM, Flash memory has no byte erasure option. Because Flash memory can be programmed while it is in its socket on the system board, it is widely used to upgrade the BIOS ROM of the PC. Some designers believe that Flash memory will replace the hard disk as a mass storage medium.

Table 0-6: Some EEPROM and Flash Chips**EEPROMs**

Part No.	Capacity	Org.	Speed	Pins	V _{PP}
2816A-25	16K	2K × 8	250 ns	24	5 V
2864A	64K	8K × 8	250 ns	28	5 V
28C64A-25	64K	8K × 8	250 ns	28	5 V CMOS
28C256-15	256K	32K × 8	150 ns	28	5 V
28C256-25	256K	32K × 8	250 ns	28	5 V CMOS

Flash

Part No.	Capacity	Org.	Speed	Pins	V _{PP}
28F256-20	256K	32K × 8	200 ns	32	12 V CMOS
28F010-15	1024K	128K × 8	150 ns	32	12 V CMOS
28F020-15	2048K	256K × 8	150 ns	32	12 V CMOS

This would increase the performance of the computer tremendously, since Flash memory is semiconductor memory with access time in the range of 100 ns compared with disk access time in the range of tens of milliseconds. For this to happen, Flash memory's program/erase cycles must become infinite, just like hard disks. Program/erase cycle refers to the number of times that a chip can be erased and reprogrammed before it becomes unusable. At this time, the program/erase cycle is 100,000 for Flash and EEPROM, 1000 for UV-EPROM, and infinite for RAM and disks. See Table 0-6 for some sample chips.

Mask ROM

Mask ROM refers to a kind of ROM in which the contents are programmed by the IC manufacturer. In other words, it is not a user-programmable ROM. The term *mask* is used in IC fabrication. Since the process is costly, mask ROM is used when the needed volume is high (hundreds of thousands) and it is absolutely certain that the contents will not change. It is common practice to use UV-EPROM or Flash for the development phase of a project, and only after the code/data have been finalized is the mask version of the product ordered. The main advantage of mask ROM is its cost, since it is significantly cheaper than other kinds of ROM, but if an error is found in the data/code, the entire batch must be thrown away. It must be noted that all ROM memories have 8 bits for data pins; therefore, the organization is ×8.

RAM (random access memory)

RAM memory is called *volatile* memory since cutting off the power to the IC results in the loss of data. Sometimes RAM is also referred to as RAWM (read and write memory), in contrast to ROM, which cannot be written to. There are three types of RAM: static RAM (SRAM), NV-RAM (nonvolatile RAM), and dynamic RAM (DRAM). Each is explained separately.

SRAM (static RAM)

Storage cells in static RAM memory are made of flip-flops and therefore do not require refreshing in order to keep their data. This is in contrast to DRAM, discussed below. The problem with the use of flip-flops for storage cells is that each cell requires at least 6 transistors to build, and the cell holds only 1 bit of data. In recent years, the cells have been made of 4 transistors, which still is too many. The use of 4-transistor cells plus the use of CMOS technology has given birth to a high-capacity SRAM, but its capacity is far below DRAM. Figure 0-11 shows the pin diagram for an SRAM chip.

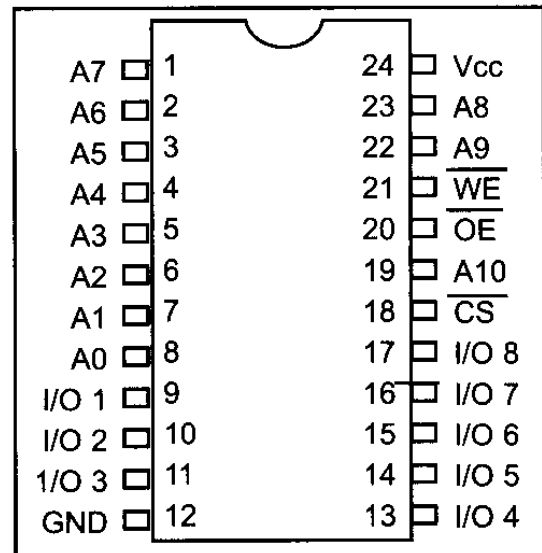


Figure 0-11. 2K × 8 SRAM Pins

The following is a description of the 6116 SRAM pins.

A0–A10 are for address inputs, where 11 address lines gives $2^{11} = 2K$.

WE (write enable) is for writing data into SRAM (active low).

OE (output enable) is for reading data out of SRAM (active low)

CS (chip select) is used to select the memory chip.

I/O0–I/O7 are for data I/O, where 8-bit data lines give an organization of $2K \times 8$.

The functional diagram for the 6116 SRAM is given in Figure 0-12.

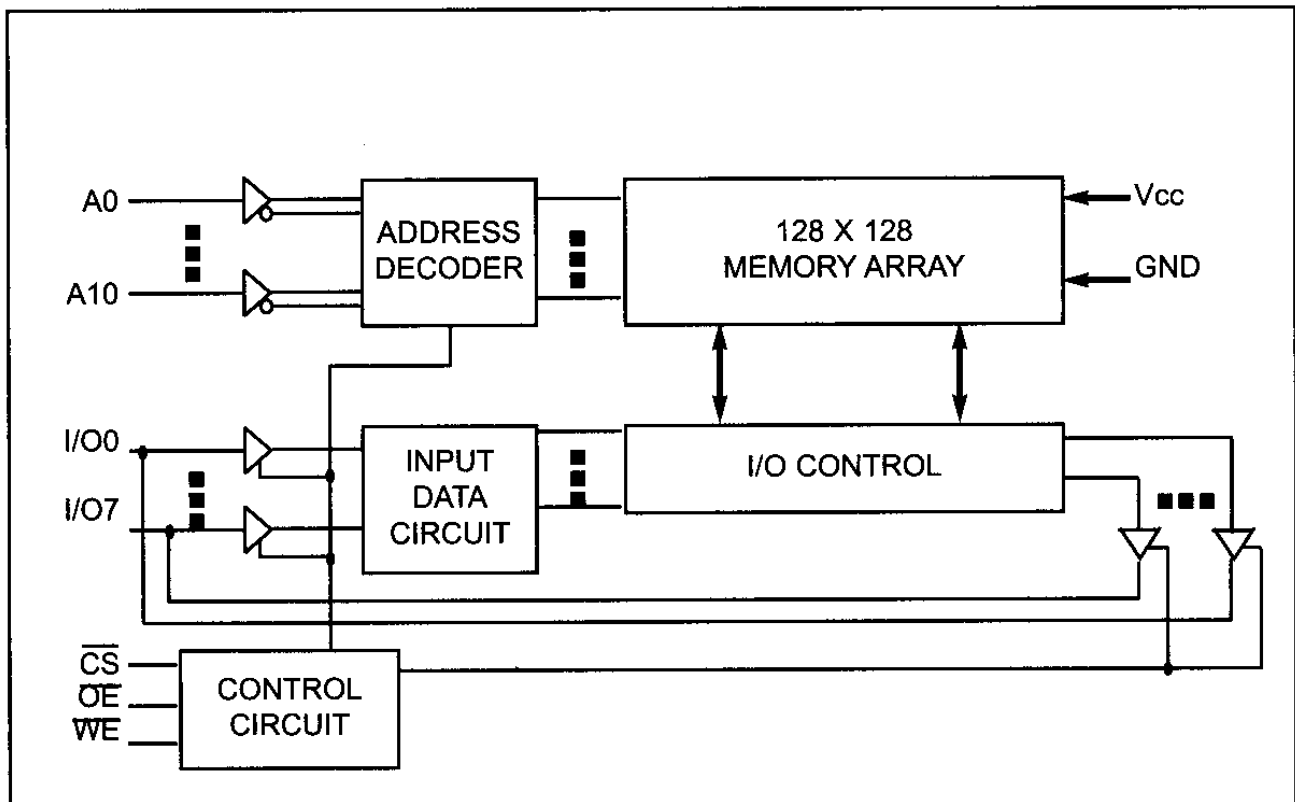


Figure 0-12. Functional Block Diagram for 6116 SRAM

Figure 0-13 shows the following steps to write data into SRAM.

1. Provide the addresses to pins A0–A10.
2. Activate the CS pin.
3. Make WE = 0 while RD = 1.
4. Provide the data to pins I/O0–I/O7.
5. Make WE = 1 and data will be written into SRAM on the positive edge of the WE signal.

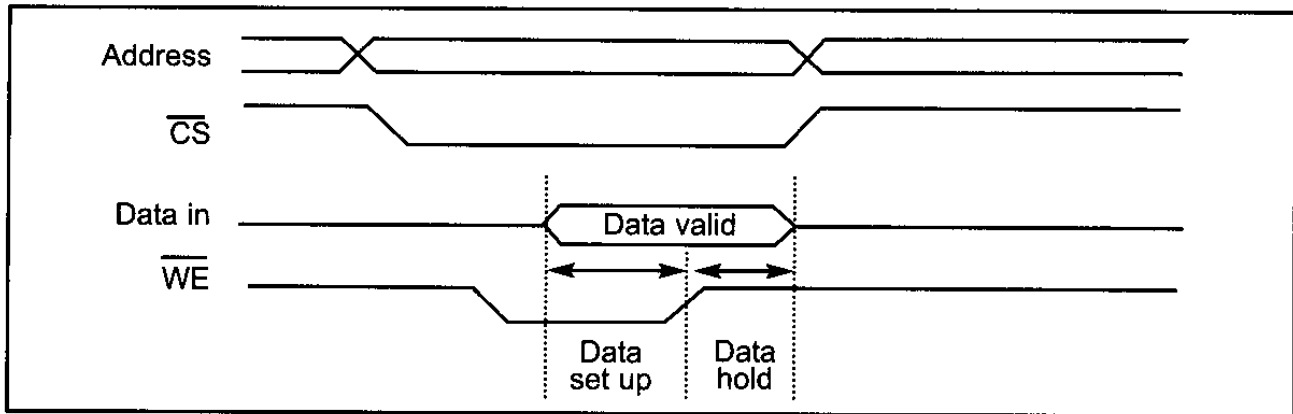


Figure 0-13. Memory Write Timing for SRAM

The following are steps to read data from SRAM. See Figure 0-14.

1. Provide the addresses to pins A0–A10. This is the start of the access time (t_{AA}).
2. Activate the CS pin.
3. While WE = 1, a high-to-low pulse on the OE pin will read the data out of the chip.

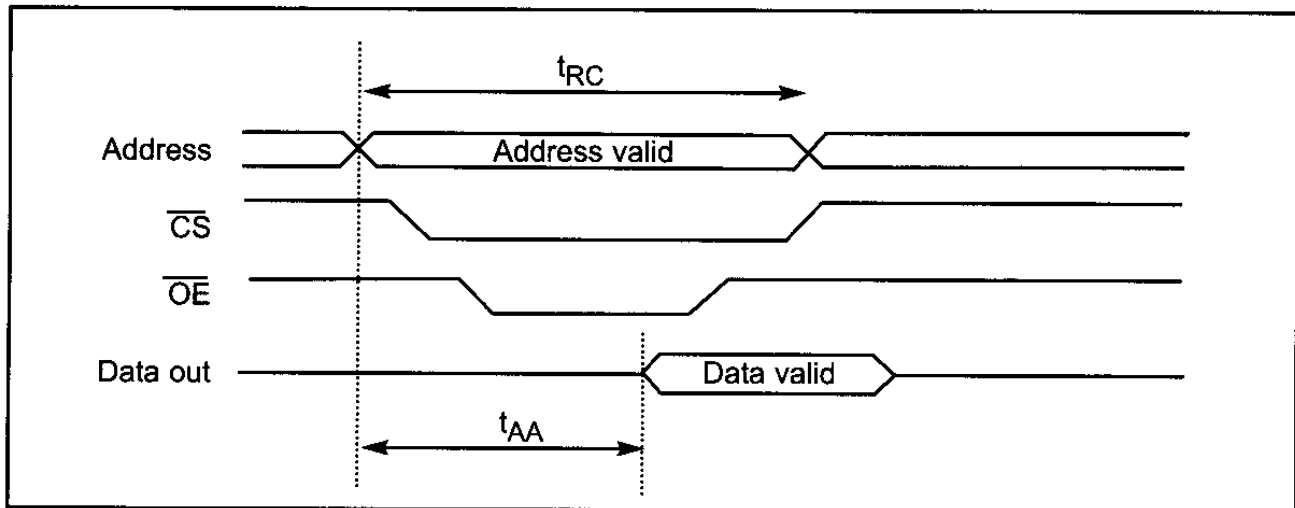


Figure 0-14. Memory Read Timing for SRAM

NV-RAM (nonvolatile RAM)

Whereas SRAM is volatile, there is a new type of nonvolatile RAM called NV-RAM. Like other RAMs, it allows the CPU to read and write to it, but when the power is turned off the contents are not lost. NV-RAM combines the best of RAM and ROM: the read and write ability of RAM, plus the nonvolatility of ROM. To retain its contents, every NV-RAM chip internally is made of the following components:

1. It uses extremely power-efficient (very low-power consumption) SRAM cells built out of CMOS.

Table 0-7: Some SRAM and NV-RAM Chips**SRAM**

Part No.	Capacity	Org.	Speed	Pins	V _{PP}
6116P-1	16K	2K × 8	100 ns	24	CMOS
6116P-2	16K	2K × 8	120 ns	24	CMOS
6116P-3	16K	2K × 8	150 ns	24	CMOS
6116LP-1	16K	2K × 8	100 ns	24	Low-power CMOS
6116LP-2	16K	2K × 8	120 ns	24	Low-power CMOS
6116LP-3	16K	2K × 8	150 ns	24	Low-power CMOS
6264P-10	64K	8K × 8	100 ns	28	CMOS
6264LP-70	64K	8K × 8	70 ns	28	Low-power CMOS
6264LP-12	64K	8K × 8	120 ns	28	Low-power CMOS
62256LP-10	256K	32K × 8	100 ns	28	Low-power CMOS
62256LP-12	256K	32K × 8	120 ns	28	Low-power CMOS

NV-RAM from Dallas Semiconductor

Part No.	Capacity	Org.	Speed	Pins	V _{PP}
DS1220Y-150	16K	2K × 8	150 ns	24	
DS1225AB-150	64K	8K × 8	150 ns	28	
DS1230Y-85	256K	32K × 8	85 ns	28	

2. It uses an internal lithium battery as a backup energy source.
3. It uses an intelligent control circuitry. The main job of this control circuitry is to monitor the V_{CC} pin constantly to detect loss of the external power supply. If the power to the V_{CC} pin falls below out-of-tolerance conditions, the control circuitry switches automatically to its internal power source, the lithium battery. The internal lithium power source is used to retain the NV-RAM contents only when the external power source is off.

It must be emphasized that all three of the components above are incorporated into a single IC chip, and for this reason nonvolatile RAM is a very expensive type of RAM as far as cost per bit is concerned. Offsetting the cost, however, is the fact that it can retain its contents up to ten years after the power has been turned off and allows one to read and write in exactly the same way as SRAM. Table 0-7 shows some examples of SRAM and NV-RAM parts.

DRAM (dynamic RAM)

Since the early days of the computer, the need for huge, inexpensive read/write memory has been a major preoccupation of computer designers. In 1970, Intel Corporation introduced the first dynamic RAM (random access memory). Its density (capacity) was 1024 bits and it used a capacitor to store each bit. Using a capacitor to store data cuts down the number of transistors needed to build the cell; however, it requires constant refreshing due to leakage. This is in contrast to SRAM (static RAM), whose individual cells are made of flip-flops. Since each bit in SRAM uses a single flip-flop, and each flip-flop requires six transistors,

SRAM has much larger memory cells and consequently lower density. The use of capacitors as storage cells in DRAM results in much smaller net memory cell size.

The advantages and disadvantages of DRAM memory can be summarized as follows. The major advantages are high density (capacity), cheaper cost per bit, and lower power consumption per bit. The disadvantage is that it must be refreshed periodically because the capacitor cell loses its charge; furthermore, while DRAM is being refreshed, the data cannot be accessed. This is in contrast to SRAM's flip-flops, which retain data as long as the power is on, do not need to be refreshed, and whose contents can be accessed at any time. Since 1970, the capacity of DRAM has exploded. After the 1K-bit (1024) chip came the 4K-bit in 1973, and then the 16K chip in 1976. The 1980s saw the introduction of 64K, 256K, and finally 1M and 4M memory chips. The 1990s saw 16M, 64M, 256M, and the beginning of 1G-bit DRAM chips. In the 2000s, 2G-bit chips are standard, and as the fabrication process gets smaller, larger memory chips will be rolling off the manufacturing line. Keep in mind that when talking about IC memory chips, the capacity is always assumed to be in bits. Therefore, a 1M chip means a 1-megabit chip and a 256K chip means a 256K-bit memory chip. However, when talking about the memory of a computer system, it is always assumed to be in bytes.

Packaging issue in DRAM

In DRAM there is a problem of packing a large number of cells into a single chip with the normal number of pins assigned to addresses. For example, a 64K-bit chip ($64K \times 1$) must have 16 address lines and 1 data line, requiring 16 pins to send in the address if the conventional method is used. This is in addition to V_{CC} power, ground, and read/write control pins. Using the conventional method of data access, the large number of pins defeats the purpose of high density and small packaging, so dearly cherished by IC designers. Therefore, to reduce the number of pins needed for addresses, multiplexing/demultiplexing is used. The method used is to split the address in half and send in each half of the address through the same pins, thereby requiring fewer address pins. Internally, the DRAM structure is divided into a square of rows and columns. The first half of the address is called the row and the second half is called the column. For example, in the case of DRAM of $64K \times 1$ organization, the first half of the address is sent in through the 8 pins A0–A7, and by activating RAS (row address strobe), the internal latches inside DRAM grab the first half of the address. After that, the second half of the address is sent in through the same pins, and by activating CAS (column address strobe), the internal latches inside DRAM latch the second half of the address. This results in using 8 pins for addresses plus RAS and CAS, for a total of 10 pins, instead of the 16 pins that would be required without multiplexing. To access a bit of data from DRAM, both row and column addresses must be provided. For this concept to work, there must be a 2-by-1 multiplexer outside the DRAM circuitry and a demultiplexer inside every DRAM chip. Due to the complexities associated with DRAM interfacing (RAS, CAS, the need for multiplexer and refreshing circuitry), some DRAM controllers are designed to make DRAM interfacing much easier. However, many small microcontroller-based projects that do not require much RAM (usually less than 64K bytes) use SRAM of types EEPROM and NV-RAM, instead of DRAM.

DRAM organization

In the discussion of ROM, we noted that all of these chips have 8 pins for data. This is not the case for DRAM memory chips, which can have $\times 1$, $\times 4$, $\times 8$, or $\times 16$ organizations. See Example 0-15 and Table 0-8.

In memory chips, the data pins are also called I/O. In some DRAMs there are separate D_{in} and D_{out} pins. Figure 0-15 shows a $256K \times 1$ DRAM chip with pins A0–A8 for address, RAS and CAS, WE (write enable), and data in and data out, as well as power and ground.

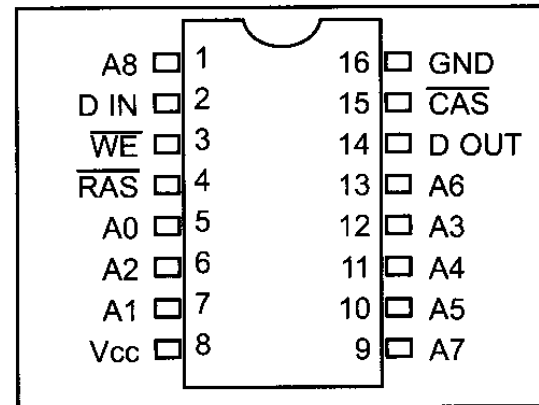


Figure 0-15. $256K \times 1$ DRAM

Example 0-15

Discuss the number of pins set aside for addresses in each of the following memory chips.
 (a) $16K \times 4$ DRAM (b) $16K \times 4$ SRAM

Solution:

Since $2^{14} = 16K$:

- (a) For DRAM we have 7 pins (A0–A6) for the address pins and 2 pins for RAS and CAS.
 (b) For SRAM we have 14 pins for address and no pins for RAS and CAS since they are associated only with DRAM. In both cases we have 4 pins for the data bus.

Table 0-8: Some DRAMs

Part No.	Speed	Capacity	Org.	Pins
4164-15	150 ns	64K	$64K \times 1$	16
41464-8	80 ns	256K	$64K \times 4$	18
41256-15	150 ns	256K	$256K \times 1$	16
41256-6	60 ns	256K	$256K \times 1$	16
414256-10	100 ns	1M	$256K \times 4$	20
511000P-8	80 ns	1M	$1M \times 1$	18
514100-7	70 ns	4M	$4M \times 1$	20

Memory address decoding

Next we discuss address decoding. The CPU provides the address of the data desired, but it is the job of the decoding circuitry to locate the selected memory block. To explore the concept of decoding circuitry, we look at various methods used in decoding the addresses. In this discussion we use SRAM or ROM for the sake of simplicity.

Memory chips have one or more pins called CS (chip select), which must be activated for the memory's contents to be accessed. Sometimes the chip select is also referred to as chip enable (CE). In connecting a memory chip to the CPU,

note the following points.

1. The data bus of the CPU is connected directly to the data pins of the memory chip.
2. Control signals RD (read) and WR (memory write) from the CPU are connected to the OE (output enable) and WE (write enable) pins of the memory chip, respectively.
3. In the case of the address buses, while the lower bits of the addresses from the CPU go directly to the memory chip address pins, the upper ones are used to activate the CS pin of the memory chip. It is the CS pin that along with RD/WR allows the flow of data in or out of the memory chip. No data can be written into or read from the memory chip unless CS is activated.

As can be seen from the data sheets of SRAM and ROM, the CS input of a memory chip is normally active low and is activated by the output of the memory decoder. Normally memories are divided into blocks, and the output of the decoder selects a given memory block. There are three ways to generate a memory block selector: (a) using simple logic gates, (b) using the 74LS138, or (c) using programmable logics such as CPLD and FPGA. Each method is described below.

Simple logic gate address decoder

The simplest method of constructing decoding circuitry is the use of a NAND gate. The output of a NAND gate is active low, and the CS pin is also active low, which makes them a perfect match. In cases where the CS input is active high, an AND gate must be used. Using a combination of NAND gates and inverters, one can decode any address range. An example of this is shown in Figure 0-16, which shows that A15–A12 must be 0011 in order to select the chip. This results in the assignment of addresses 3000H to 3FFFH to this memory chip.

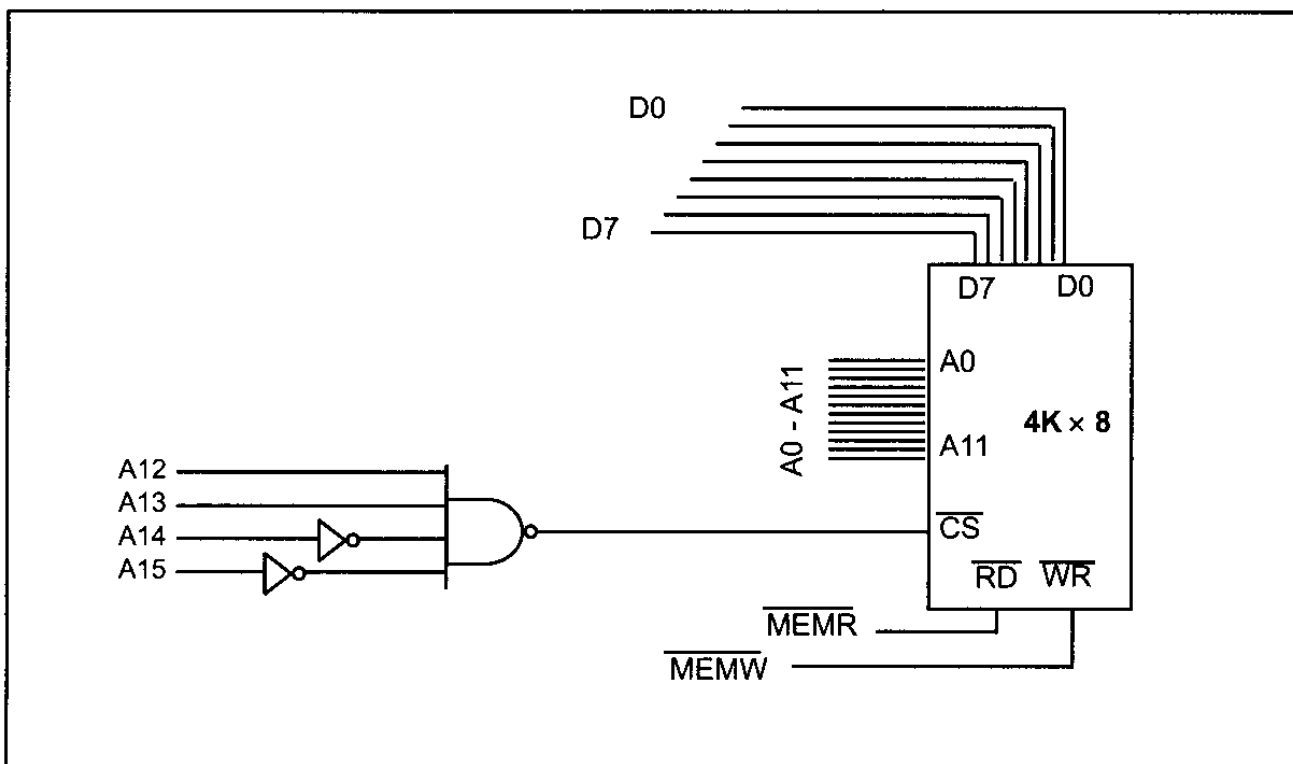


Figure 0-16. Logic Gate as Decoder

Using the 74LS138 3-8 decoder

This used to be one of the most widely used address decoders. The 3 inputs A, B, and C generate 8 active-low outputs Y0–Y7. See Figure 0-17. Each Y output is connected to CS of a memory chip, allowing control of 8 memory blocks by a single 74LS138. In the 74LS138, where A, B, and C select which output is activated, there are three additional inputs, G2A, G2B, and G1. G2A and G2B are both active low, and G1 is active high. If any one of the inputs G1, G2A, or G2B is not connected to an address signal (sometimes they are connected to a control signal), they must be activated permanently by either V_{CC} or ground, depending on the activation level. Example 0-16 shows the design and the address range calculation for the 74LS138 decoder.

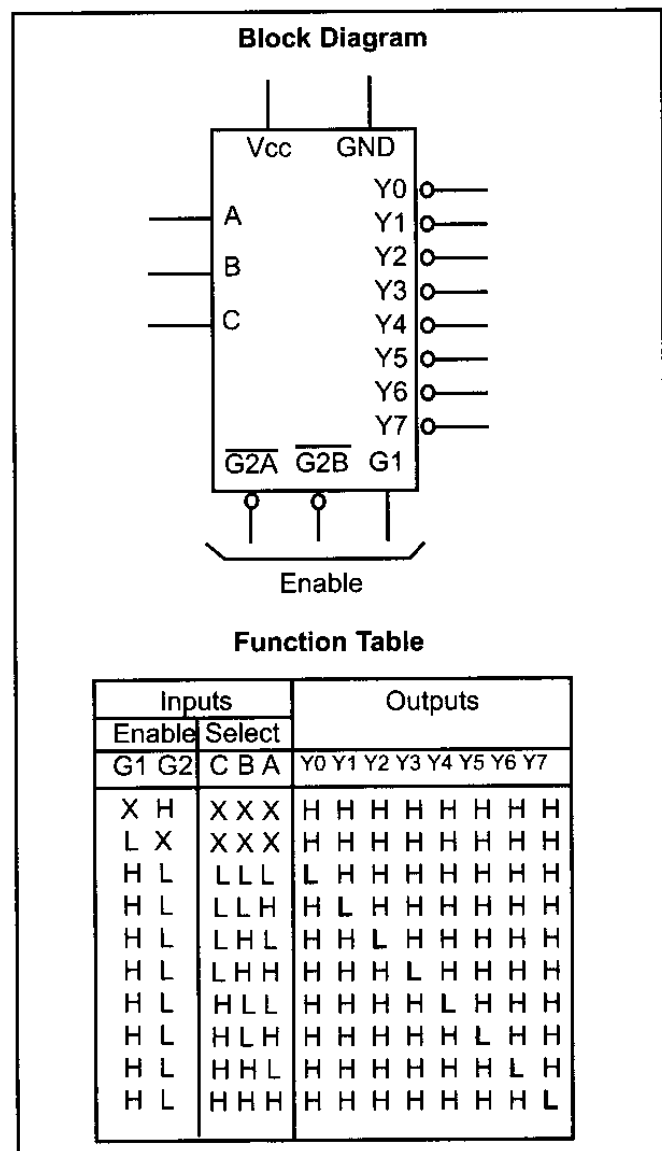


Figure 0-17. 74LS138 Decoder

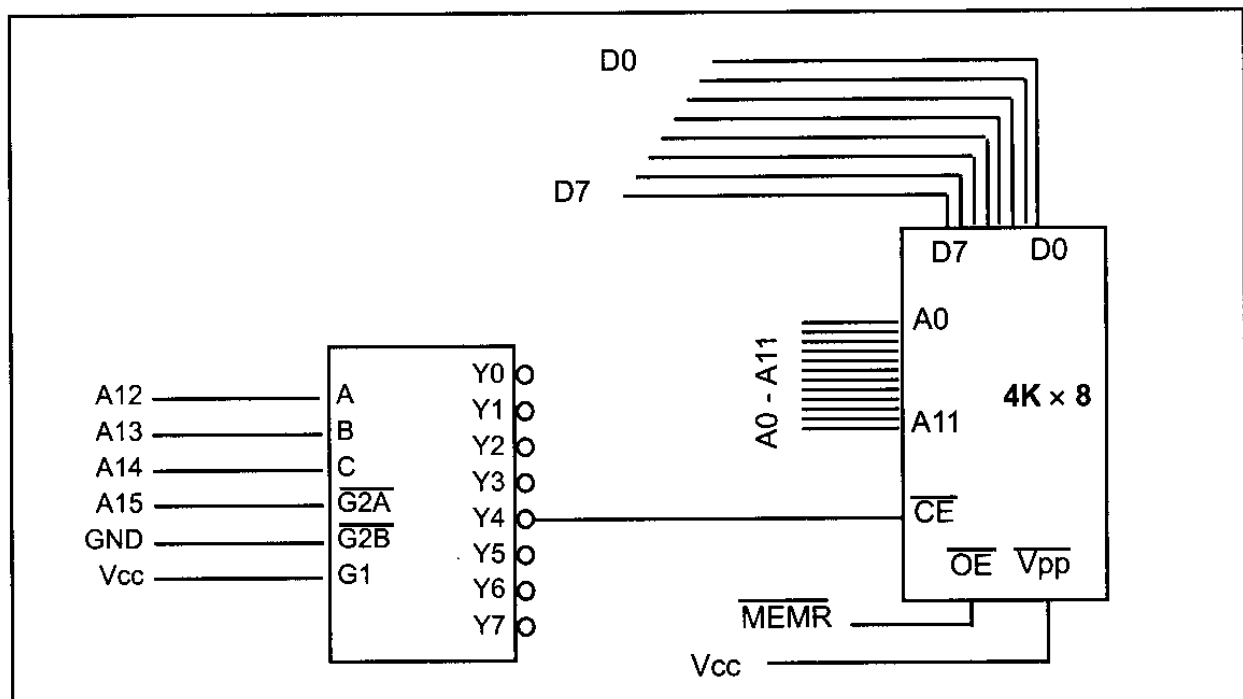


Figure 0-18. Using 74LS138 as Decoder

Example 0-16

Looking at the design in Figure 0-18, find the address range for the following:
(a) Y4, (b) Y2, and (c) Y7.

Solution:

(a) The address range for Y4 is calculated as follows.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1

The above shows that the range for Y4 is 4000H to 4FFFH. In Figure 0-18, notice that A15 must be 0 for the decoder to be activated. Y4 will be selected when A14 A13 A12 = 100 (4 in binary). The remaining A11–A0 will be 0 for the lowest address and 1 for the highest address.

(b) The address range for Y2 is 2000H to 2FFFH.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1

(c) The address range for Y7 is 7000H to 7FFFH.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Using programmable logic as an address decoder

Other widely used decoders are programmable logic chips such as PAL, GAL, and FPGA chips. One disadvantage of these chips is that they require PAL/GAL/FPGA software and a burner (programmer), whereas the 74LS138 needs neither of these. The advantage of these chips is that they can be programmed for any combination of address ranges, and so are much more versatile. This plus the fact that PAL/GAL/FPGA chips have 10 or more inputs (in contrast to 6 in the 74138) means that they can accommodate more address inputs.

Review Questions

1. How many bytes is 24 kilobytes?
2. What does “RAM” stand for? How is it used in computer systems?
3. What does “ROM” stand for? How is it used in computer systems?
4. Why is RAM called volatile memory?
5. List the three major components of a computer system.
6. What does “CPU” stand for? Explain its function in a computer.
7. List the three types of buses found in computer systems and state briefly the purpose of each type of bus.
8. State which of the following is unidirectional and which is bidirectional:
(a) data bus (b) address bus

9. If an address bus for a given computer has 16 lines, what is the maximum amount of memory it can access?
10. The speed of semiconductor memory is in the range of
 - (a) microseconds (b) milliseconds
 - (c) nanoseconds (d) picoseconds
11. Find the organization and chip capacity for each ROM with the indicated number of address and data pins.
 - (a) 14 address, 8 data (b) 16 address, 8 data (c) 12 address, 8 data
12. Find the organization and chip capacity for each RAM with the indicated number of address and data pins.
 - (a) 11 address, 1 data SRAM (b) 13 address, 4 data SRAM
 - (c) 17 address, 8 data SRAM (d) 8 address, 4 data DRAM
 - (e) 9 address, 1 data DRAM (f) 9 address, 4 data DRAM
13. Find the capacity and number of pins set aside for address and data for memory chips with the following organizations.
 - (a) $16K \times 4$ SRAM (b) $32K \times 8$ EPROM (c) $1M \times 1$ DRAM
 - (d) $256K \times 4$ SRAM (e) $64K \times 8$ EEPROM (f) $1M \times 4$ DRAM
14. Which of the following is (are) volatile memory?
 - (a) EEPROM (b) SRAM (c) DRAM (d) NV-RAM
15. A given memory block uses addresses 4000H–7FFFH. How many kilobytes is this memory block?
16. The 74138 is a(n) _____ by _____ decoder.
17. In the 74138 give the status of G2A and G2B for the chip to be enabled.
18. In the 74138 give the status of G1 for the chip to be enabled.
19. In Example 0-16, what is the range of addresses assigned to Y5?

SECTION 0.4: CPU ARCHITECTURE

In this section we will examine the inside of a CPU. Then, we will compare the Harvard and von Neumann architectures.

Inside CPU

A program stored in memory provides instructions to the CPU to perform an action. See Figure 0-19. The action can simply be adding data such as payroll data or controlling a machine such as a robot. The function of the CPU is to fetch these instructions from memory and execute them. To perform the actions of fetch and execute, all CPUs are equipped with resources such as the following:

1. Foremost among the resources at the disposal of the CPU are a number of *registers*. The CPU uses registers to store information temporarily. The information could be two values to be processed, or the address of the value needed to be fetched from memory. Registers inside the CPU can be 8-bit, 16-bit, 32-bit, or even 64-bit registers, depending on the CPU. In general, the more and bigger the registers, the better the CPU. The disadvantage of more and bigger registers is the increased cost of such a CPU.
2. The CPU also has what is called the *ALU* (arithmetic/logic unit). The ALU section of the CPU is responsible for performing arithmetic functions such as add,

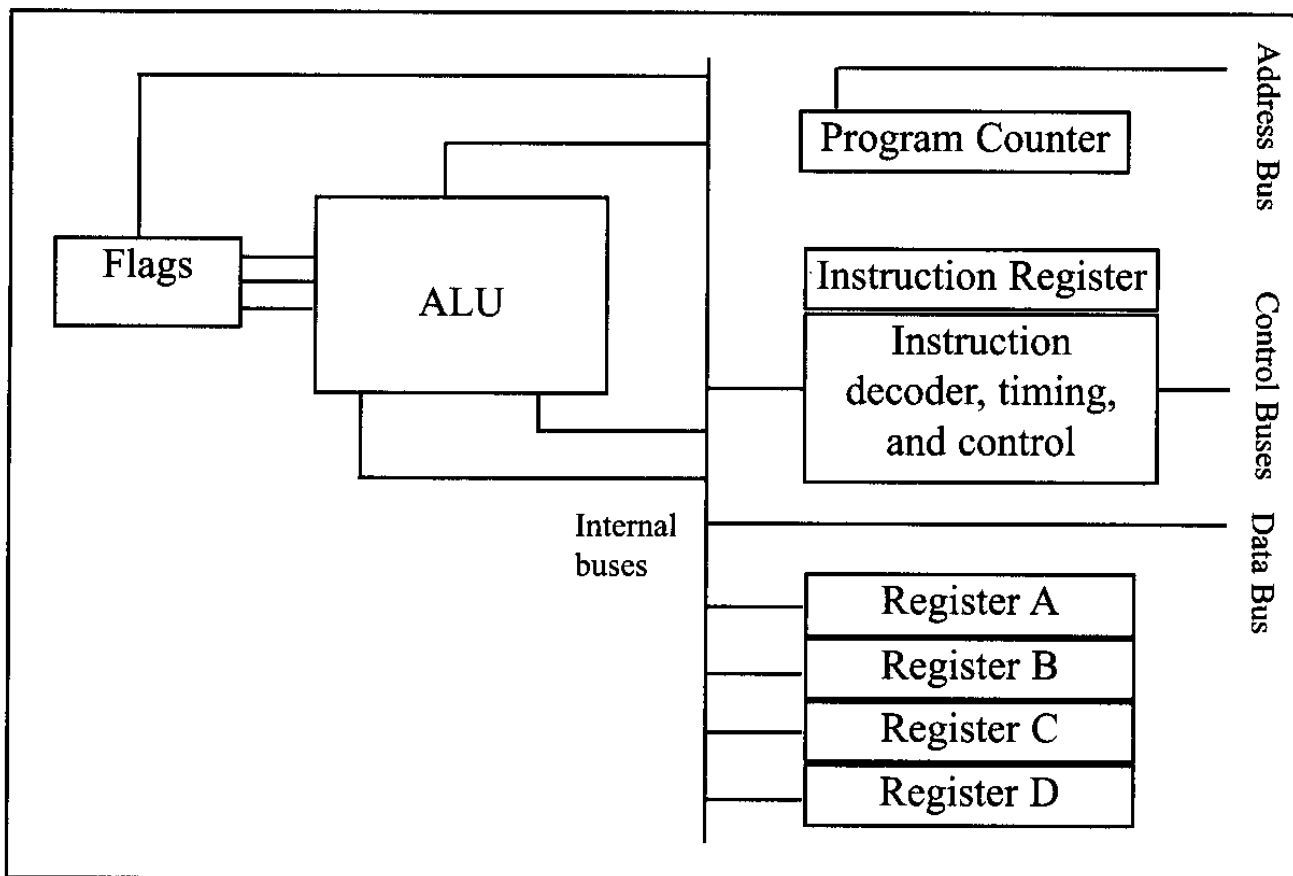


Figure 0-19. Internal Block Diagram of a CPU

- subtract, multiply, and divide, and logic functions such as AND, OR, and NOT.
- Every CPU has what is called a *program counter*. The function of the program counter is to point to the address of the next instruction to be executed. As each instruction is executed, the program counter is incremented to point to the address of the next instruction to be executed. The contents of the program counter are placed on the address bus to find and fetch the desired instruction. In the IBM PC, the program counter is a register called IP, or the instruction pointer.
 - The function of the *instruction decoder* is to interpret the instruction fetched into the CPU. One can think of the instruction decoder as a kind of dictionary, storing the meaning of each instruction and what steps the CPU should take upon receiving a given instruction. Just as a dictionary requires more pages the more words it defines, a CPU capable of understanding more instructions requires more transistors to design.

Internal working of CPUs

To demonstrate some of the concepts discussed above, a step-by-step analysis of the process a CPU would go through to add three numbers is given next. Assume that an imaginary CPU has registers called A, B, C, and D. It has an 8-bit data bus and a 16-bit address bus. Therefore, the CPU can access memory from addresses 0000 to FFFFH (for a total of 10000H locations). The action to be performed by the CPU is to put hexadecimal value 21 into register A, and then add to register A the values 42H and 12H. Assume that the code for the CPU to move a value to register A is 1011 0000 (B0H) and the code for adding a value to register A is 0000 0100 (04H). The necessary steps and code to perform these opera-

tions are as follows.

Action	Code	Data
Move value 21H into register A	B0H	21H
Add value 42H to register A	04H	42H
Add value 12H to register A	04H	12H

If the program to perform the actions listed above is stored in memory locations starting at 1400H, the following would represent the contents for each memory address location:

Memory address	Contents of memory address
1400	(B0)code for moving a value to register A
1401	(21)value to be moved
1402	(04)code for adding a value to register A
1403	(42)value to be added
1404	(04)code for adding a value to register A
1405	(12)value to be added
1406	(F4)code for halt

The actions performed by the CPU to run the program above would be as follows:

1. The CPU's program counter can have a value between 0000 and FFFFH. The program counter must be set to the value 1400H, indicating the address of the first instruction code to be executed. After the program counter has been loaded with the address of the first instruction, the CPU is ready to execute.
2. The CPU puts 1400H on the address bus and sends it out. The memory circuitry finds the location while the CPU activates the READ signal, indicating to memory that it wants the byte at location 1400H. This causes the contents of memory location 1400H, which is B0, to be put on the data bus and brought into the CPU.
3. The CPU decodes the instruction B0 with the help of its instruction decoder dictionary. When it finds the definition for that instruction it knows it must bring the byte in the next memory location into register A of the CPU. Therefore, it commands its controller circuitry to do exactly that. When it brings in value 21H from memory location 1401, it makes sure that the doors of all registers are closed except register A. Therefore, when value 21H comes into the CPU it will go directly into register A. After completing one instruction, the program counter points to the address of the next instruction to be executed, which in this case is 1402H. Address 1402 is sent out on the address bus to fetch the next instruction.
4. From memory location 1402H the CPU fetches code 04H. After decoding, the CPU knows that it must add the byte sitting at the next address (1403) to the contents of register A. After the CPU brings the value (in this case, 42H) into register A, it provides the contents of register A along with this value to the ALU to perform the addition. It then takes the result of the addition from the ALU's output and puts it into register A. Meanwhile the program counter becomes 1404, the address of the next instruction.

5. Address 1404H is put on the address bus and the code is fetched into the CPU, decoded, and executed. This code again is adding a value to register A. The program counter is updated to 1406H.
6. Finally, the contents of address 1406 are fetched in and executed. This HALT instruction tells the CPU to stop incrementing the program counter and asking for the next instruction. Without the HALT, the CPU would continue updating the program counter and fetching instructions.

Now suppose that address 1403H contained value 04 instead of 42H. How would the CPU distinguish between data 04 to be added and code 04? Remember that code 04 for this CPU means “move the next value into register A.” Therefore, the CPU will not try to decode the next value. It simply moves the contents of the following memory location into register A, regardless of its value.

Harvard and von Neumann architectures

Every microprocessor must have memory space to store program (code) and data. While code provides instructions to the CPU, the data provides the information to be processed. The CPU uses buses (wire traces) to access the code ROM and data RAM memory spaces. The early computers used the same bus for accessing both the code and data. Such an architecture is commonly referred to as *von Neumann (Princeton) architecture*. That means for von Neumann computers, the process of accessing the code or data could cause them to get in each other's way and slow down the processing speed of the CPU, because each had to wait for the other to finish fetching. To speed up the process of program execution, some CPUs use what is called *Harvard architecture*. In Harvard architecture, we have separate buses for the code and data memory. See Figure 0-20. That means that we need four sets of buses: (1) a set of data buses for carrying data into and out of the CPU, (2) a set of address buses for accessing the data, (3) a set of data buses for carrying code into the CPU, and (4) an address bus for accessing the code. See Figure 0-20. This is easy to implement inside an IC chip such as a microcontroller where both ROM code and data RAM are internal (on-chip) and distances are on the micron and millimeter scale. But implementing Harvard architecture for systems such as x86 IBM PC-type computers is very expensive because the RAM and ROM that hold code and data are external to the CPU. Separate wire traces for data and code on the motherboard will make the board large and expensive. For example, for a Pentium microprocessor with a 64-bit data bus and a 32-bit address bus we will need about 100 wire traces on the motherboard if it is von Neumann architecture (96 for address and data, plus a few others for control signals of read and write and so on). But the number of wire traces will double to 200 if we use Harvard architecture. Harvard architecture will also necessitate a large number of pins coming out of the microprocessor itself. For this reason you do not see Harvard architecture implemented in the world of PCs and workstations. This is also the reason that microcontrollers such as AVR use Harvard architecture internally, but they still use von Neumann architecture if they need external memory for code and data space. The von Neumann architecture was developed at Princeton University, while the Harvard architecture was the work of Harvard University.

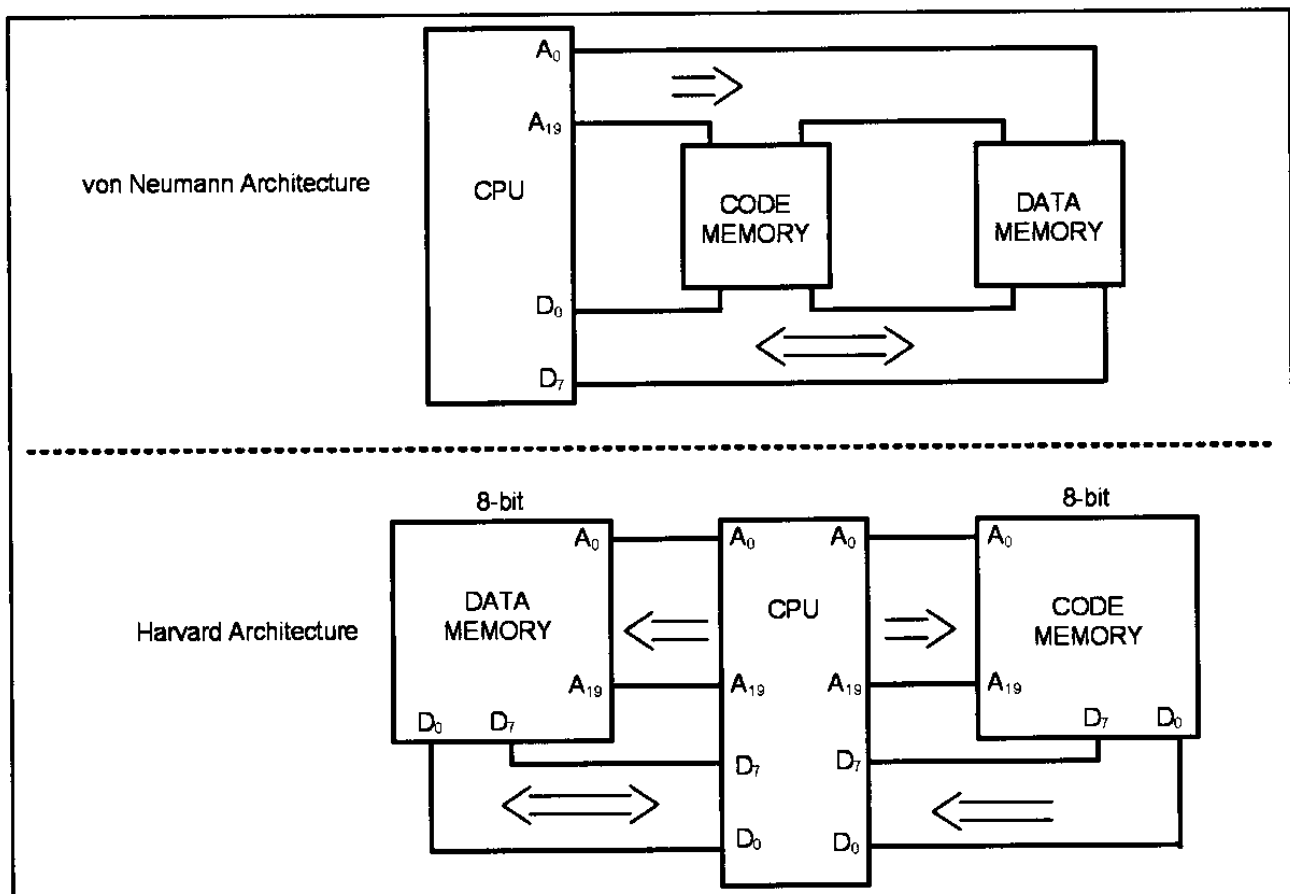


Figure 0-20. von Neumann vs. Harvard Architecture

Review Questions

1. What does "ALU" stand for? What is its purpose?
2. How are registers used in computer systems?
3. What is the purpose of the program counter?
4. What is the purpose of the instruction decoder?
5. True or false. Harvard architecture uses the same address and data buses to fetch both code and data.

SUMMARY

The binary number system represents all numbers with a combination of the two binary digits, 0 and 1. The use of binary systems is necessary in digital computers because only two states can be represented: on or off. Any binary number can be coded directly into its hexadecimal equivalent for the convenience of humans. Converting from binary/hex to decimal, and vice versa, is a straightforward process that becomes easy with practice. ASCII code is a binary code used to represent alphanumeric data internally in the computer. It is frequently used in peripheral devices for input and/or output.

The AND, OR, and inverter logic gates are the basic building blocks of simple circuits. NAND, NOR, and XOR gates are also used to implement circuit design. Diagrams of half-adders and full-adders were given as examples of the use of logic gates for circuit design. Decoders are used to detect certain addresses. Flip-flops are used to latch in data until other circuits are ready for it.

The major components of any computer system are the CPU, memory, and

I/O devices. “Memory” refers to temporary or permanent storage of data. In most systems, memory can be accessed as bytes or words. The terms *kilobyte*, *megabyte*, *gigabyte*, and *terabyte* are used to refer to large numbers of bytes. There are two main types of memory in computer systems: RAM and ROM. RAM (random access memory) is used for temporary storage of programs and data. ROM (read-only memory) is used for permanent storage of programs and data that the computer system must have in order to function. All components of the computer system are under the control of the CPU. Peripheral devices such as I/O (input/output) devices allow the CPU to communicate with humans or other computer systems. There are three types of buses in computers: address, control, and data. Control buses are used by the CPU to direct other devices. The address bus is used by the CPU to locate a device or a memory location. Data buses are used to send information back and forth between the CPU and other devices.

This chapter provided an overview of semiconductor memories. Types of memories were compared in terms of their capacity, organization, and access time. ROM (read-only memory) is nonvolatile memory typically used to store programs in embedded systems. The relative advantages of various types of ROM were described, including PROM, EPROM, UV-EPROM, EEPROM, Flash memory EPROM, and mask ROM.

Address decoding techniques using simple logic gates, decoders, and programmable logic were covered.

The computer organization and the internals of the CPU were also covered.

PROBLEMS

SECTION 0.1: NUMBERING AND CODING SYSTEMS

- Convert the following decimal numbers to binary:
(a) 12 (b) 123 (c) 63 (d) 128 (e) 1000
- Convert the following binary numbers to decimal:
(a) 100100 (b) 1000001 (c) 11101 (d) 1010 (e) 00100010
- Convert the values in Problem 2 to hexadecimal.
- Convert the following hex numbers to binary and decimal:
(a) 2B9H (b) F44H (c) 912H (d) 2BH (e) FFFFH
- Convert the values in Problem 1 to hex.
- Find the 2’s complement of the following binary numbers:
(a) 1001010 (b) 111001 (c) 10000010 (d) 111110001
- Add the following hex values:
(a) 2CH + 3FH (b) F34H + 5D6H (c) 20000H + 12FFH
(d) FFFFH + 2222H
- Perform hex subtraction for the following:
(a) 24FH – 129H (b) FE9H – 5CCH (c) 2FFFFH – FFFFFH
(d) 9FF25H – 4DD99H
- Show the ASCII codes for numbers 0, 1, 2, 3, ..., 9 in both hex and binary.
- Show the ASCII code (in hex) for the following strings:
“U.S.A. is a country” CR,LF
“in North America” CR,LF
(CR is carriage return, LF is line feed)

SECTION 0.2: DIGITAL PRIMER

11. Draw a 3-input OR gate using a 2-input OR gate.
12. Show the truth table for a 3-input OR gate.
13. Draw a 3-input AND gate using a 2-input AND gate.
14. Show the truth table for a 3-input AND gate.
15. Design a 3-input XOR gate with a 2-input XOR gate. Show the truth table for a 3-input XOR.
16. List the truth table for a 3-input NAND.
17. List the truth table for a 3-input NOR.
18. Show the decoder for binary 1100.
19. Show the decoder for binary 11011.
20. List the truth table for a D-FF.

SECTION 0.3: SEMICONDUCTOR MEMORY

21. Answer the following:
 - (a) How many nibbles are 16 bits?
 - (b) How many bytes are 32 bits?
 - (c) If a word is defined as 16 bits, how many words is a 64-bit data item?
 - (d) What is the exact value (in decimal) of 1 meg?
 - (e) How many kilobytes is 1 meg?
 - (f) What is the exact value (in decimal) of 1 gigabyte?
 - (g) How many kilobytes is 1 gigabyte?
 - (h) How many megs is 1 gigabyte?
 - (i) If a given computer has a total of 8 megabytes of memory, how many bytes (in decimal) is this? How many kilobytes is this?
22. A given mass storage device such as a hard disk can store 2 gigabytes of information. Assuming that each page of text has 25 rows and each row has 80 columns of ASCII characters (each character = 1 byte), approximately how many pages of information can this disk store?
23. In a given byte-addressable computer, memory locations 10000H to 9FFFFH are available for user programs. The first location is 10000H and the last location is 9FFFFH. Calculate the following:
 - (a) The total number of bytes available (in decimal)
 - (b) The total number of kilobytes (in decimal)
24. A given computer has a 32-bit data bus. What is the largest number that can be carried into the CPU at a time?
25. Below are listed several computers with their data bus widths. For each computer, list the maximum value that can be brought into the CPU at a time (in both hex and decimal).
 - (a) Apple 2 with an 8-bit data bus
 - (b) x86 PC with a 16-bit data bus
 - (c) x86 PC with a 32-bit data bus
 - (d) Cray supercomputer with a 64-bit data bus
26. Find the total amount of memory, in the units requested, for each of the following CPUs, given the size of the address buses:

- (a) 16-bit address bus (in K)
 - (b) 24-bit address bus (in megs)
 - (c) 32-bit address bus (in megabytes and gigabytes)
 - (d) 48-bit address bus (in megabytes, gigabytes, and terabytes)
27. Of the data bus and address bus, which is unidirectional and which is bidirectional?
 28. What is the difference in capacity between a 4M memory chip and 4M of computer memory?
 29. True or false. The more address pins, the more memory locations are inside the chip. (Assume that the number of data pins is fixed.)
 30. True or false. The more data pins, the more each location inside the chip will hold.
 31. True or false. The more data pins, the higher the capacity of the memory chip.
 32. True or false. The more data pins and address pins, the greater the capacity of the memory chip.
 33. The speed of a memory chip is referred to as its _____.
 34. True or false. The price of memory chips varies according to capacity and speed.
 35. The main advantage of EEPROM over UV-EPROM is _____.
 36. True or false. SRAM has a larger cell size than DRAM.
 37. Which of the following, EPROM, DRAM, or SRAM, must be refreshed periodically?
 38. Which memory is used for PC cache?
 39. Which of the following, SRAM, UV-EPROM, NV-RAM, or DRAM, is volatile memory?
 40. RAS and CAS are associated with which type of memory?
 - (a) EPROM (b) SRAM (c) DRAM (d) all of the above
 41. Which type of memory needs an external multiplexer?
 - (a) EPROM (b) SRAM (c) DRAM (d) all of the above
 42. Find the organization and capacity of memory chips with the following pins.

(a) EEPROM A0–A14, D0–D7	(b) UV-EPROM A0–A12, D0–D7
(c) SRAM A0–A11, D0–D7	(d) SRAM A0–A12, D0–D7
(e) DRAM A0–A10, D0	(f) SRAM A0–A12, D0
(g) EEPROM A0–A11, D0–D7	(h) UV-EPROM A0–A10, D0–D7
(i) DRAM A0–A8, D0–D3	(j) DRAM A0–A7, D0–D7
 43. Find the capacity, address, and data pins for the following memory organizations.

(a) 16K × 8 ROM	(b) 32K × 8 ROM
(c) 64K × 8 SRAM	(d) 256K × 8 EEPROM
(e) 64K × 8 ROM	(f) 64K × 4 DRAM
(g) 1M × 8 SRAM	(h) 4M × 4 DRAM
(i) 64K × 8 NV-RAM	
 44. Find the address range of the memory design in the diagram.
 45. Using NAND gates and inverters, design decoding circuitry for the address range 2000H–2FFFH.
 46. Find the address range for Y0, Y3, and Y6 of the 74LS138 for the diagrammed

- design.
47. Using the 74138, design the memory decoding circuitry in which the memory block controlled by Y0 is in the range 0000H to 1FFFH. Indicate the size of the memory block controlled by each Y.
 48. Find the address range for Y3, Y6, and Y7 in Problem 47.
 49. Using the 74138, design memory decoding circuitry in which the memory block controlled by Y0 is in the 0000H to 3FFFH space. Indicate the size of the memory block controlled by each Y.
 50. Find the address range for Y1, Y2, and Y3 in Problem 49.

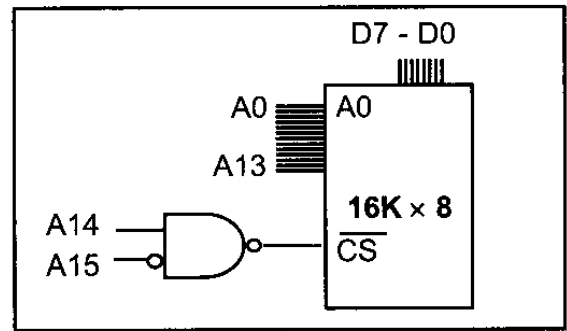


Diagram for Problem 44

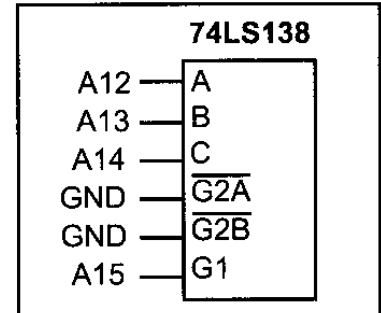


Diagram for Problem 46

SECTION 0.4: CPU AND HARVARD ARCHITECTURE

51. Which register of the CPU holds the address of the instruction to be fetched?
52. Which section of the CPU is responsible for performing addition?
53. List the three bus types present in every CPU.

ANSWERS TO REVIEW QUESTIONS

SECTION 0.1: NUMBERING AND CODING SYSTEMS

1. Computers use the binary system because each bit can have one of two voltage levels: on and off.
2. $34_{10} = 100010_2 = 22_{16}$
3. $110101_2 = 35_{16} = 53_{10}$
4. 1110001
5. 010100
6. 461
7. 275
8. 38 30 78 38 36 20 43 50 55 73

SECTION 0.2: DIGITAL PRIMER

1. AND
2. OR
3. XOR
4. Buffer
5. Storing data
6. Decoder

SECTION 0.3: SEMICONDUCTOR MEMORY

1. 24,576
2. Random access memory; it is used for temporary storage of programs that the CPU is run-

ning, such as the operating system, word processing programs, etc.

3. Read-only memory; it is used for permanent programs such as those that control the keyboard, etc.
4. The contents of RAM are lost when the computer is powered off.
5. The CPU, memory, and I/O devices
6. Central processing unit; it can be considered the “brain” of the computer; it executes the programs and controls all other devices in the computer.
7. The address bus carries the location (address) needed by the CPU; the data bus carries information in and out of the CPU; the control bus is used by the CPU to send signals controlling I/O devices.
8. (a) bidirectional (b) unidirectional
9. 64K, or 65,536 bytes
10. c
11. (a) $16K \times 8$, 128K bits (b) $64K \times 8$, 512K (c) $4K \times 8$, 32K
12. (a) $2K \times 1$, 2K bits (b) $8K \times 4$, 32K (c) $128K \times 8$, 1M
(d) $64K \times 4$, 256K (e) $256K \times 1$, 256K (f) $256K \times 4$, 1M
13. (a) 64K bits, 14 address, and 4 data (b) 256K, 15 address, and 8 data
(c) 1M, 10 address, and 1 data (d) 1M, 18 address, and 4 data
(e) 512K, 16 address, and 8 data (f) 4M, 10 address, and 4 data
14. b, c
15. 16K bytes
16. 3, 8
17. Both must be low.
18. G1 must be high.
19. 5000H–5FFFH

SECTION 0.4: CPU ARCHITECTURE

1. Arithmetic/logic unit; it performs all arithmetic and logic operations.
2. They are used for temporary storage of information.
3. It holds the address of the next instruction to be executed.
4. It tells the CPU what actions to perform for each instruction.
5. False

CHAPTER 1

THE AVR MICROCONTROLLER: HISTORY AND FEATURES

OBJECTIVES

Upon completion of this chapter, you will be able to:

- >> Compare and contrast microprocessors and microcontrollers
- >> Describe the advantages of microcontrollers for some applications
- >> Explain the concept of embedded systems
- >> Discuss criteria for considering a microcontroller
- >> Explain the variations of speed, packaging, memory, and cost per unit and how these affect choosing a microcontroller
- >> Compare and contrast the various members of the AVR family
- >> Compare the AVR with microcontrollers offered by other manufacturers

This chapter begins with a discussion of the role and importance of microcontrollers in everyday life. In Section 1.1 we also discuss criteria to consider in choosing a microcontroller, as well as the use of microcontrollers in the embedded market. Section 1.2 covers various members of the AVR family and their features. In addition, we provide a brief discussion of alternatives to the AVR chip such as the 8051, PIC, and 68HC11 microcontrollers.

SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS

In this section we discuss the need for microcontrollers and contrast them with general-purpose microprocessors such as the Pentium and other x86 microprocessors. We also look at the role of microcontrollers in the embedded market. In addition, we provide some criteria on how to choose a microcontroller.

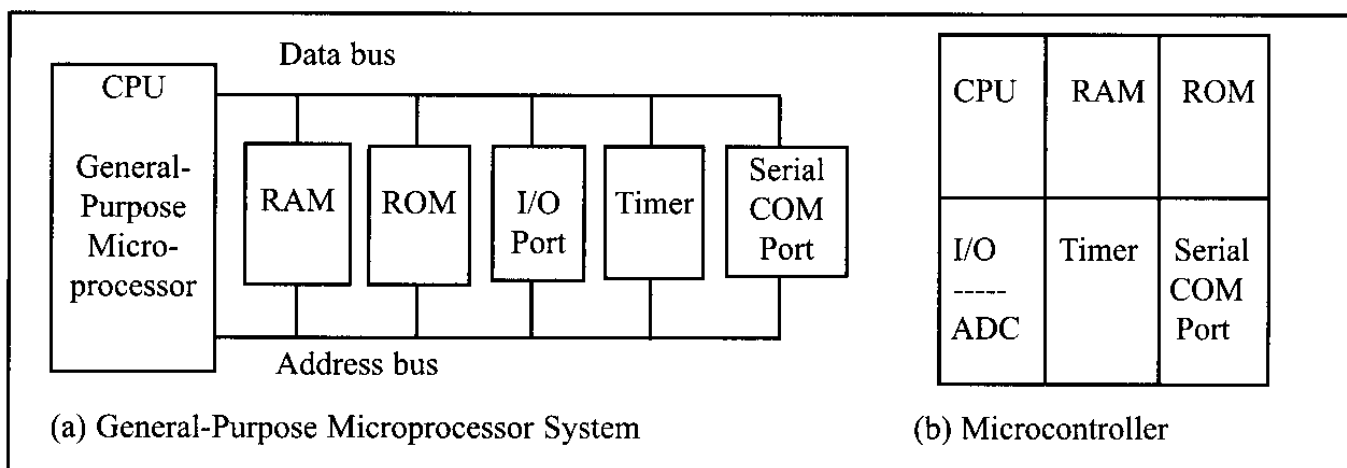


Figure 1-1. Microprocessor System Contrasted with Microcontroller System

Microcontroller versus general-purpose microprocessor

What is the difference between a microprocessor and a microcontroller? By microprocessor is meant the general-purpose microprocessors such as Intel's x86 family (8086, 80286, 80386, 80486, and the Pentium) or Motorola's PowerPC family. These microprocessors contain no RAM, no ROM, and no I/O ports on the chip itself. For this reason, they are commonly referred to as *general-purpose microprocessors*. See Figure 1-1.

A system designer using a general-purpose microprocessor such as the Pentium or the PowerPC must add RAM, ROM, I/O ports, and timers externally to make them functional. Although the addition of external RAM, ROM, and I/O ports makes these systems bulkier and much more expensive, they have the advantage of versatility, enabling the designer to decide on the amount of RAM, ROM, and I/O ports needed to fit the task at hand. This is not the case with microcontrollers. A microcontroller has a CPU (a microprocessor) in addition to a fixed amount of RAM, ROM, I/O ports, and a timer all on a single chip. In other words, the processor, RAM, ROM, I/O ports, and timer are all embedded together on one chip; therefore, the designer cannot add any external memory, I/O, or timer to it. The fixed amount of on-chip ROM, RAM, and number of I/O ports in microcontrollers makes them ideal for many applications in which cost and space are criti-

Home
Appliances
Intercom
Telephones
Security systems
Garage door openers
Answering machines
Fax machines
Home computers
TVs
Cable TV tuner
VCR
Camcorder
Remote controls
Video games
Cellular phones
Musical instruments
Sewing machines
Lighting control
Paging
Camera
Pinball machines
Toys
Exercise equipment
Office
Telephones
Computers
Security systems
Fax machine
Microwave
Copier
Laser printer
Color printer
Paging
Auto
Trip computer
Engine control
Air bag
ABS
Instrumentation
Security system
Transmission control
Entertainment
Climate control
Cellular phone
Keyless entry

Table 1-1: Some Embedded Products Using Microcontrollers

cal. In many applications, for example, a TV remote control, there is no need for the computing power of a 486 or even an 8086 microprocessor. In many applications, the space used, the power consumed, and the price per unit are much more critical considerations than the computing power. These applications most often require some I/O operations to read signals and turn on and off certain bits. For this reason some call these processors IBP, “itty-bitty processors.” (See “Good Things in Small Packages Are Generating Big Product Opportunities” by Rick Grehan, BYTE magazine, September 1994 (<http://www.byte.com>) for an excellent discussion of microcontrollers.)

It is interesting to note that many microcontroller manufacturers have gone as far as integrating an ADC (analog-to-digital converter) and other peripherals into the microcontroller.

Microcontrollers for embedded systems

In the literature discussing microprocessors, we often see the term *embedded system*. Microprocessors and microcontrollers are widely used in embedded system products. An embedded system is controlled by its own internal microprocessor (or microcontroller) as opposed to an external controller. Typically, in an embedded system, the microcontroller’s ROM is burned with a purpose for specific functions needed for the system. A printer is an example of an embedded system because the processor inside it performs one task only; namely, getting the data and printing it. Contrast this with a Pentium-based PC (or any x86 PC), which can be used for any number of applications such as word processor, print server, bank teller terminal, video game player, network server, or Internet terminal. A PC can also load and run software for a variety of applications. Of course, the reason a PC can perform myriad tasks is that it has RAM memory and an operating system that loads the application software into RAM and lets the CPU run it. In an embedded system, typically only one application software is burned into ROM. An x86 PC contains or is connected to various embedded products such as the keyboard, printer, modem, disk controller, sound card, CD-ROM driver, mouse, and so on. Each one of these peripherals has a microcontroller inside it that performs only one task. For example, inside every mouse a microcontroller performs the task of finding the mouse’s position and sending it to the PC. Table 1-1 lists some embedded products.

x86 PC embedded applications

Although microcontrollers are the preferred choice for many embedded systems, sometimes a microcontroller is inadequate for the task. For this reason, in recent years many manufacturers of general-purpose microprocessors such as Intel, Freescale

Semiconductor (formerly Motorola), and AMD (Advanced Micro Devices, Inc.) have targeted their microprocessors for the high end of the embedded market. Intel and AMD push their x86 processors for both the embedded and desktop PC markets. In the early 1990s, Apple computer began using the PowerPC microprocessors (604, 603, 620, etc.) in place of the 680x0 for the Macintosh. In 2007 Apple switched to the x86 CPU for use in the Mac computers. The PowerPC microprocessor is a joint venture between IBM and Freescale, and is targeted for the high end of the embedded market. It must be noted that when a company targets a general-purpose microprocessor for the embedded market it optimizes the processor used for embedded systems. For this reason these processors are often called *high-end embedded processors*. Another chip widely used in the high end of the embedded system design is the ARM (Advanced RISC Machine) microprocessor. Very often the terms *embedded processor* and *microcontroller* are used interchangeably.

One of the most critical needs of an embedded system is to decrease power consumption and space. This can be achieved by integrating more functions into the CPU chip. All the embedded processors based on the x86 and PowerPC 6xx have low power consumption in addition to some forms of I/O, COM port, and ROM, all on a single chip. In high-performance embedded processors, the trend is to integrate more and more functions on the CPU chip and let the designer decide which features to use. This trend is invading PC system design as well. Normally, in designing the PC motherboard we need a CPU plus a chipset containing I/O, a cache controller, a Flash ROM containing BIOS, and finally a secondary cache memory. New designs are emerging in industry. For example, many companies have a chip that contains the entire CPU and all the supporting logic and memory, except for DRAM. In other words, we have the entire computer on a single chip.

Currently, because of Linux and Windows standardization, many embedded systems use x86 PCs. In many cases, using x86 PCs for the high-end embedded applications not only saves money but also shortens development time because a vast library of software already exists for the Linux and Windows platforms. The fact that Windows and Linux are widely used and well-understood platforms means that developing a Windows-based or Linux-based embedded product reduces the cost and shortens the development time considerably.

Choosing a microcontroller

There are five major 8-bit microcontrollers. They are: Freescale Semiconductor's (formerly Motorola) 68HC08/68HC11, Intel's 8051, Atmel's AVR, Zilog's Z8, and PIC from Microchip Technology. Each of the above microcontrollers has a unique instruction set and register set; therefore, they are not compatible with each other. Programs written for one will not run on the others. There are also 16-bit and 32-bit microcontrollers made by various chip makers. With all these different microcontrollers, what criteria do designers consider in choosing one? Three criteria in choosing microcontrollers are as follows: (1) meeting the computing needs of the task at hand efficiently and cost effectively; (2) availability of software and hardware development tools such as compilers, assemblers, debuggers, and emulators; and (3) wide availability and reliable sources of the microcontroller. Next, we elaborate on each of the above criteria.

Criteria for choosing a microcontroller

1. The first and foremost criterion in choosing a microcontroller is that it must meet the task at hand efficiently and cost effectively. In analyzing the needs of a microcontroller-based project, we must first see whether an 8-bit, 16-bit, or 32-bit microcontroller can best handle the computing needs of the task most effectively. Among other considerations in this category are:
 - (a) Speed. What is the highest speed that the microcontroller supports?
 - (b) Packaging. Does it come in a DIP (dual inline package) or a QFP (quad flat package), or some other packaging format? This is important in terms of space, assembling, and prototyping the end product.
 - (c) Power consumption. This is especially critical for battery-powered products.
 - (d) The amount of RAM and ROM on the chip.
 - (e) The number of I/O pins and the timer on the chip.
 - (f) Ease of upgrade to higher-performance or lower-power-consumption versions.
 - (g) Cost per unit. This is important in terms of the final cost of the product in which a microcontroller is used. For example, some microcontrollers cost 50 cents per unit when purchased 100,000 units at a time.
2. The second criterion in choosing a microcontroller is how easy it is to develop products around it. Key considerations include the availability of an assembler, a debugger, a code-efficient C language compiler, an emulator, technical support, and both in-house and outside expertise. In many cases, third-party vendor (i.e., a supplier other than the chip manufacturer) support for the chip is as good as, if not better than, support from the chip manufacturer.
3. The third criterion in choosing a microcontroller is its ready availability in needed quantities both now and in the future. For some designers this is even more important than the first two criteria. Currently, of the leading 8-bit microcontrollers, the 8051 family has the largest number of diversified (multiple source) suppliers. (Supplier means a producer besides the originator of the microcontroller.) In the case of the 8051, which was originated by Intel, many companies also currently produce the 8051.

Notice that Freescale Semiconductor (Motorola), Atmel, Zilog, and Microchip Technology have all dedicated massive resources to ensure wide and timely availability of their products because their products are stable, mature, and single sourced. In recent years, companies have begun to sell *Field-Programmable Gate Array* (FPGA) and *Application-Specific Integrated Circuit* (ASIC) libraries for the different microcontrollers.

Mechatronics and microcontrollers

The microcontroller is playing a major role in an emerging field called *mechatronics*. Here is an excellent summary of what the field of mechatronics is all about, taken from the website of Newcastle University (<http://mechatronics2004.newcastle.edu.au/mech2004>), which holds a major conference every year on this subject:

“Many technical processes and products in the area of mechanical and

electrical engineering show an increasing integration of mechanics with electronics and information processing. This integration is between the components (hardware) and the information-driven functions (software), resulting in integrated systems called mechatronic systems.

The development of mechatronic systems involves finding an optimal balance between the basic mechanical structure, sensor and actuator implementation, automatic digital information processing and overall control, and this synergy results in innovative solutions. The practice of mechatronics requires multidisciplinary expertise across a range of disciplines, such as: mechanical engineering, electronics, information technology, and decision making theories.”

Review Questions

1. True or false. Microcontrollers are normally less expensive than microprocessors.
2. When comparing a system board based on a microcontroller and a general-purpose microprocessor, which one is cheaper?
3. A microcontroller normally has which of the following devices on-chip?
(a) RAM (b) ROM (c) I/O (d) all of the above
4. A general-purpose microprocessor normally needs which of the following devices to be attached to it?
(a) RAM (b) ROM (c) I/O (d) all of the above
5. An embedded system is also called a dedicated system. Why?
6. What does the term *embedded system* mean?
7. Why does having multiple sources of a given product matter?

SECTION 1.2: OVERVIEW OF THE AVR FAMILY

In this section, we first look at the AVR microcontrollers and their features and then examine the different families of AVR in more detail.

A brief history of the AVR microcontroller

The basic architecture of AVR was designed by two students of Norwegian Institute of Technology (NTH), Alf-Egil Bogen and Vegard Wollan, and then was bought and developed by Atmel in 1996.

You may ask what AVR stands for; AVR can have different meanings for different people! Atmel says that it is nothing more than a product name, but it might stand for Advanced Virtual RISC, or Alf and Vegard RISC (the names of the AVR designers).

There are many kinds of AVR microcontroller with different properties. Except for AVR32, which is a 32-bit microcontroller, AVRs are all 8-bit microprocessors, meaning that the CPU can work on only 8 bits of data at a time. Data larger than 8 bits has to be broken into 8-bit pieces to be processed by the CPU. One of the problems with the AVR microcontrollers is that they are not all 100% compatible in terms of software when going from one family to another family. To run programs written for the ATtiny25 on a ATmega64, we must recompile the program and possibly change some register locations before loading it into the ATmega64. AVRs are generally classified into four broad groups: Mega, Tiny,

Special purpose, and Classic. In this book we cover the Mega family because these microcontrollers are widely used. Also, we will focus on ATmega32 since it is powerful, widely available, and comes in DIP packages, which makes it ideal for educational purposes. For those who have mastered the Mega family, understanding the other families is very easy and straightforward. The following is a brief description of the AVR microcontroller.

AVR features

The AVR is an 8-bit RISC single-chip microcontroller with Harvard architecture that comes with some standard features such as on-chip program (code) ROM, data RAM, data EEPROM, timers and I/O ports. See Figure 1-2. Most AVR microcontrollers have some additional features like ADC, PWM, and different kinds of serial interface such as USART, SPI, I2C (TWI), CAN, USB, and so on. See Figures 1-3 and 1-4. Due to the importance of these peripherals, we have dedicated an entire chapter to many of them. The details of the RAM/ROM memory and I/O features of the Mega are given in the next few chapters.

AVR microcontroller program ROM

In microcontrollers, the ROM is used to store programs and for that reason it is called *program* or *code ROM*. Although the AVR has 8M (megabytes) of program (code) ROM space, not all family members come with that much ROM installed. The program ROM size can vary from 1K to 256K at the time of this writing, depending on the family member. The AVR was one of the first microcontrollers to use on-chip Flash memory for program storage. The Flash memory is

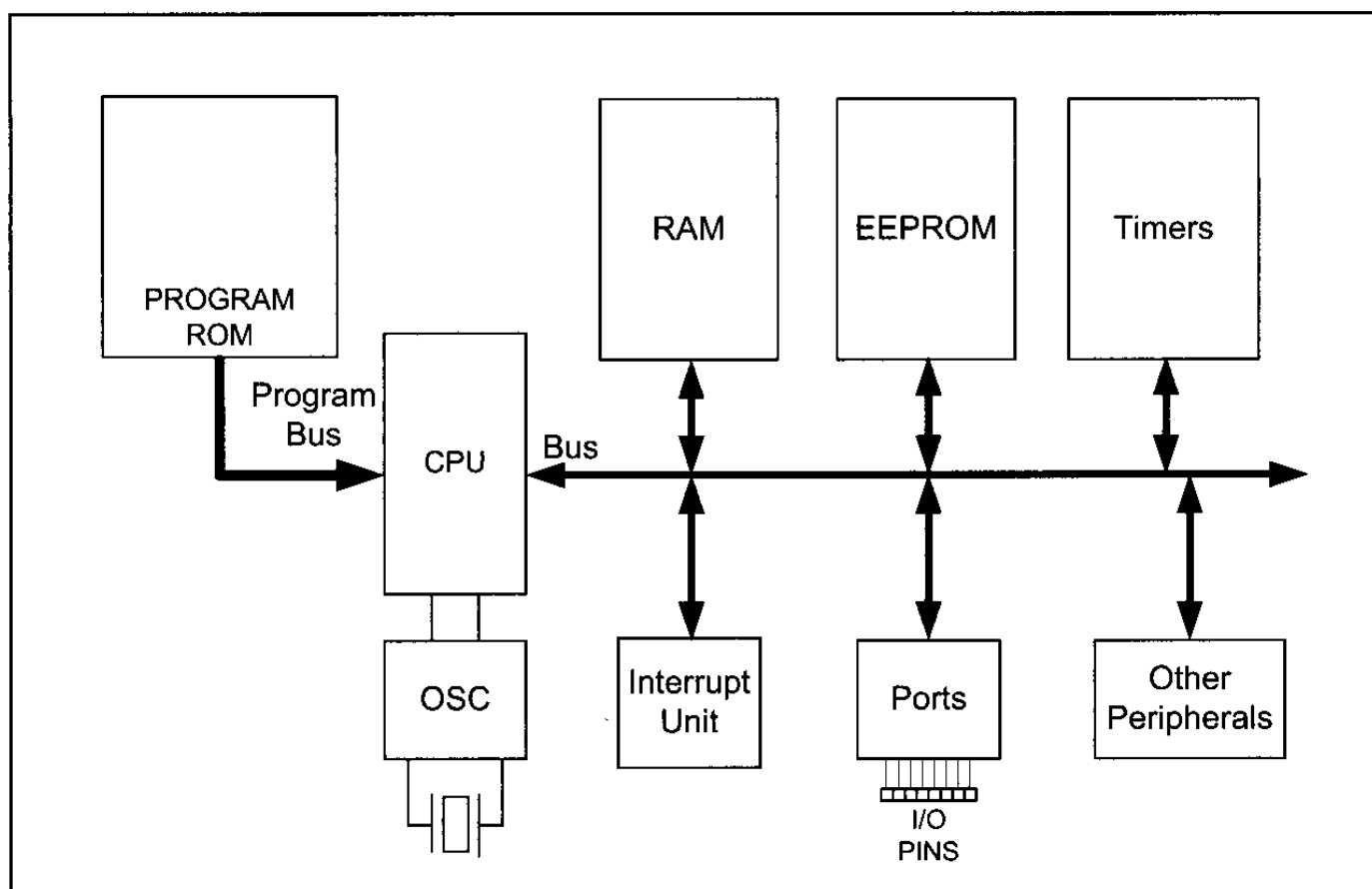


Figure 1-2. Simplified View of an AVR Microcontroller

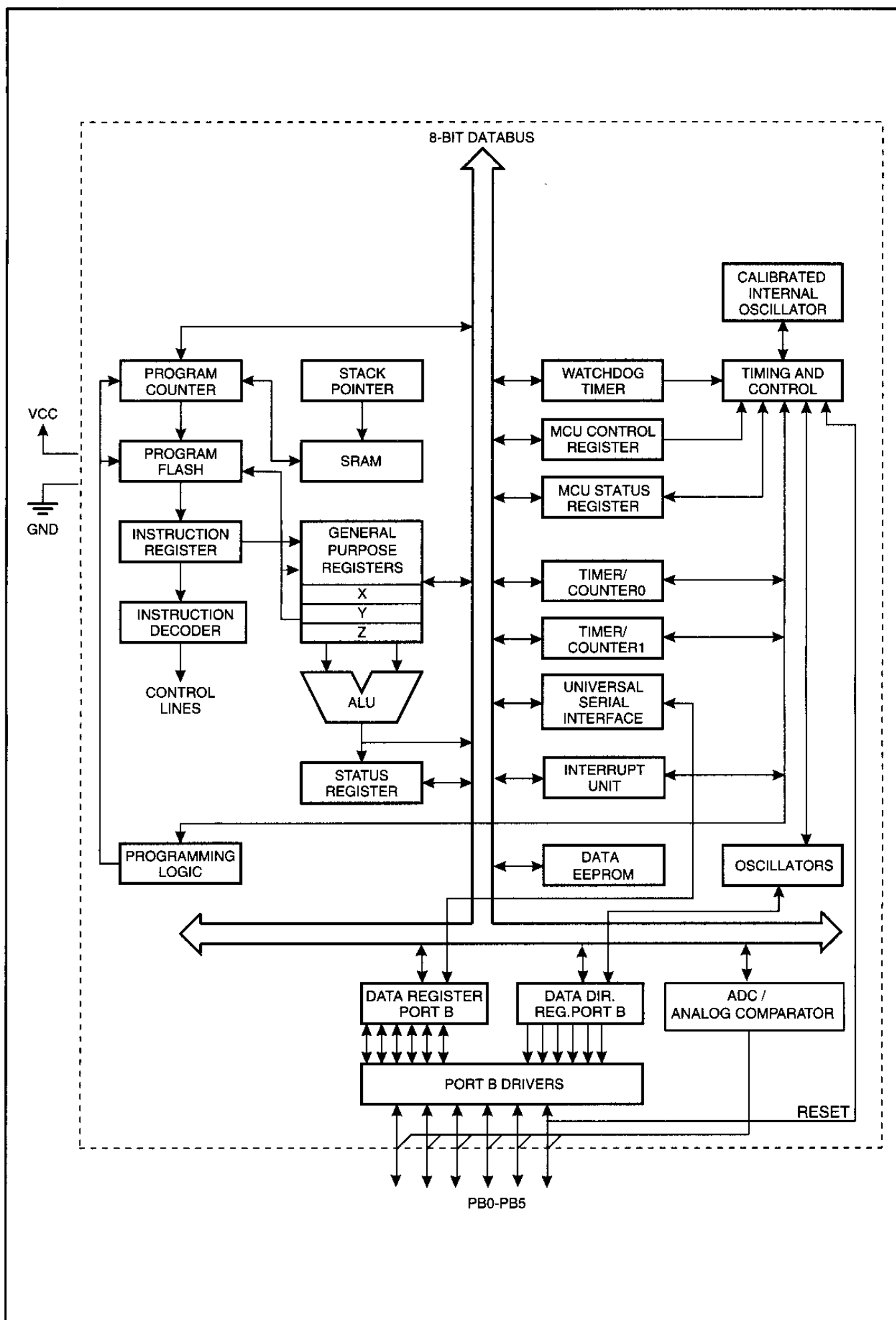


Figure 1-3. ATtiny25 Block Diagram

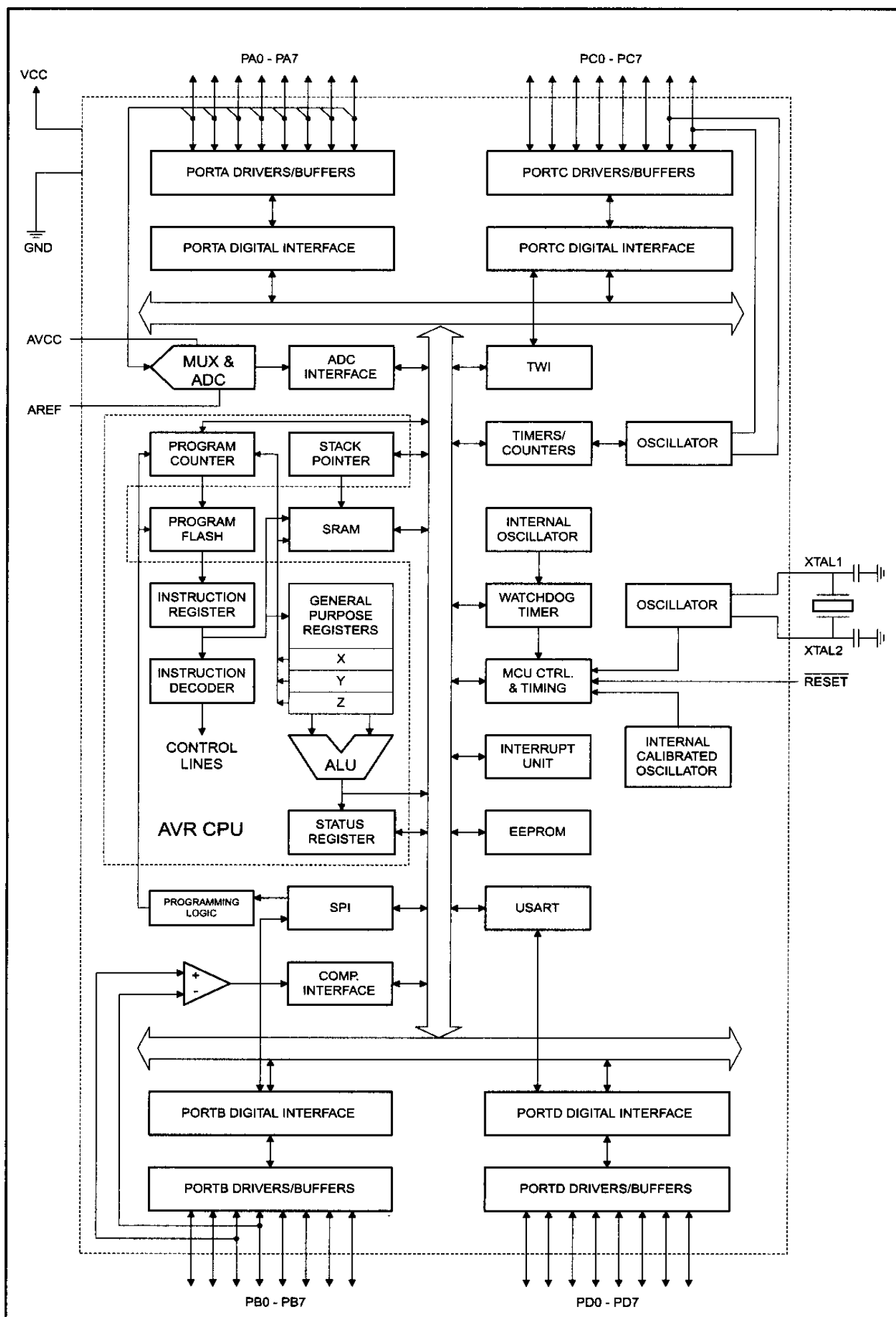


Figure 1-4. ATmega32 Block Diagram

ideal for fast development because Flash memory can be erased in seconds compared to the 20 minutes or more needed for the UV-EPROM. A discussion of the various types of ROM is given in Chapter 0, if you need to refresh your memory on these important memory technologies.

AVR microcontroller data RAM and EEPROM

While ROM is used to store program (code), the RAM space is for data storage. The AVR has a maximum of 64K bytes of data RAM space. Not all of the family members come with that much RAM. As we will see in the next chapter, the data RAM space has three components: general-purpose registers, I/O memory, and internal SRAM. There are 32 general-purpose registers in all of the AVR's, but the SRAM's size and the I/O memory's size varies from chip to chip. On the Atmel website, whenever the size of RAM is mentioned the internal SRAM size is meant. The internal SRAM space is used for a read/write scratch pad, as we will see in Chapter 2. In AVR, we also have a small amount of EEPROM to store critical data that does not need to be changed very often. You will see more about EEPROM in Chapter 6.

AVR microcontroller I/O pins

The AVR can have from 3 to 86 pins for I/O. The number of I/O pins depends on the number of pins in the package itself. The number of pins for the AVR package goes from 8 to 100 at this time. In the case of the 8-pin AT90S2323, we have 3 pins for I/O, while in the case of the 100-pin ATmega1280, we can use up to 86 pins for I/O. We will study I/O pins and programming in Chapter 4.

AVR microcontroller peripherals

Most of the AVR's come with ADC (analog-to-digital converter), timers, and USART (Universal Synchronous Asynchronous Receiver Transmitter) as standard peripherals. As we will see in Chapter 13, the ADC is 10-bit and the number of ADC channels in AVR chips varies and can be up to 16, depending on the number of pins in the package. The AVR can have up to 6 timers besides the watchdog timer. We will examine timers in Chapter 9. The USART peripheral allows us to connect the AVR-based system to serial ports such as the COM port of the x86 IBM PC, as we will see in Chapter 11. Most of the AVR family members come with the I²C and SPI buses and some of them have USB or CAN bus as well.

Table 1-2: Some Members of the Classic Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
AT90S2313	2K	128	128	15	0	2	SOIC20, PDIP20
AT90S2323	2K	128	128	3	0	1	SOIC8, PDIP8
AT90S4433	4K	128	256	20	6	2	TQFP32, PDIP28

Notes:

1. All ROM, RAM, and EEPROM memories are in bytes.
2. Data RAM (general-purpose RAM) is the amount of RAM available for data manipulation (scratch pad) in addition to the register space.

AVR family overview

AVR can be classified into four groups: Classic, Mega, Tiny, and special purpose.

Classic AVR (AT90Sxxxx)

This is the original AVR chip, which has been replaced by newer AVR chips. Table 1-2 shows some members of the Classic AVR that are not recommended for new designs.

Mega AVR (ATmegaxxxx)

These are powerful microcontrollers with more than 120 instructions and lots of different peripheral capabilities, which can be used in different designs. See Table 1-3. Some of their characteristics are as follows:

- Program memory: 4K to 256K bytes
- Package: 28 to 100 pins
- Extensive peripheral set
- Extended instruction set: They have rich instruction sets.

Table 1-3: Some Members of the ATmega Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATmega8	8K	1K	0.5K	23	8	3	TQFP32, PDIP28
ATmega16	16K	1K	0.5K	32	8	3	TQFP44, PDIP40
ATmega32	32K	2K	1K	32	8	3	TQFP44, PDIP40
ATmega64	64K	4K	2K	54	8	4	TQFP64, MLF64
ATmega1280	128K	8K	4K	86	16	6	TQFP100, CBGA

Notes:

1. All ROM, RAM, and EEPROM memories are in bytes.
2. Data RAM (general-purpose RAM) is the amount of RAM available for data manipulation (scratch pad) in addition to the register space.
3. All the above chips have USART for serial data transfer.

Tiny AVR (ATtinyxxxx)

As its name indicates, the microcontrollers in this group have less instructions and smaller packages in comparison to mega family. You can design systems with low costs and power consumptions using the Tiny AVR. See Table 1-4. Some of their characteristics are as follows:

- Program memory: 1K to 8K bytes
- Package: 8 to 28 pins
- Limited peripheral set
- Limited instruction set: The instruction sets are limited. For example, some of them do not have the multiply instruction.

Table 1-4: Some Members of the Tiny Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATtiny13	1K	64	64	6	4	1	SOIC8, PDIP8
ATtiny25	2K	128	128	6	4	2	SOIC8, PDIP8
ATtiny44	4K	256	256	12	8	2	SOIC14, PDIP14
ATtiny84	8K	512	512	12	8	2	SOIC14, PDIP14

Special purpose AVR

The ICs of this group can be considered as a subset of other groups, but their special capabilities are made for designing specific applications. Some of the special capabilities are: USB controller, CAN controller, LCD controller, Zigbee, Ethernet controller, FPGA, and advanced PWM. See Table 1-5.

Table 1-5: Some Members of the Special Purpose Family

Part Num.	Code ROM	Data RAM	Data EEPROM	Max I/O pins	Special Capabilities	Timers	Pin numbers & Package
AT90CAN128	128K	4K	4K	53	CAN	4	LQFP64
AT90USB1287	128K	8K	4K	48	USB Host	4	TQFP64
AT90PWM216	16K	1K	0.5K	19	Advanced PWM	2	SOIC24
ATmega169	16K	1K	0.5K	54	LCD	3	TQFP64, MLF64

AVR product number scheme

All of the product numbers start with AT, which stands for Atmel. Now, look at the number located at the end of the product number, from left to right, and find the biggest number that is a power of 2. This number most probably shows the amount of the microcontroller's ROM. For example, in ATmega1280 the biggest power of 2 that we can find is 128; so it has 128K bytes of ROM. In ATtiny44, the amount of memory is 4K, and so on. Although this rule has a few exceptions such as AT90PWM216, which has 16K of ROM instead of 2K, it works in most of the cases.

Other microcontrollers

There are many other popular 8-bit microcontrollers besides the AVR chip. Among them are the 8051, HCS08, PIC, and Z8. The AVR is made by Atmel Corp, as seen in Table 1-6. Microchip produces the PIC family. Freescale (formerly Motorola) makes the HCS08 and many of its variations. Zilog produces the Z8 microcontroller. The 8051 family is made by Intel and a number of other companies. To contrast the ATmega32 with the 8052 chip and PIC, examine Table 1-7.

For a comprehensive treatment of the 8051, HCS12, and PIC microcontrollers, see "The 8051 Microcontroller and Embedded Systems," "HCS12 Microcontroller and Embedded Systems," and "PIC Microcontroller and Embedded Systems" by Mazidi, et al.

Table 1-6: Some of the Companies that Produce Widely Used 8-bit Microcontrollers

Company	Web Site	Architecture
Atmel	http://www.atmel.com	AVR and 8051
Microchip	http://www.microchip.com	PIC16xxx/18xxx
Intel	http://www.intel.com/design/mcs51	8051
Philips/Signetics	http://www.semiconductors.philips.com	8051
Zilog	http://www.zilog.com	Z8 and Z80
Dallas Semi/Maxim	http://www.maxim-ic.com	8051
Freescale Semi	http://www.freescale.com	68HC11/HCS08

See <http://www.microcontroller.com> for a complete list.

Table 1-7: Comparison of 8051, PIC18 Family, and AVR (40-pin package)

Feature	8052	PIC18F452	ATmega32
Program ROM	8K	32K	32K
Data RAM (maximum space)	256 bytes	2K	2K
EEPROM	0 bytes	256 bytes	1K
Timers	3	4	3
I/O pins	32	35	32

Review Questions

1. Name three features of the AVR.
2. The AVR is a(n) _____-bit microprocessor.
3. Name the different groups of the AVR chips.
4. Which group of AVR has smaller packages?
5. Give the size of RAM in each of the following:
(a) ATmega32 (b) ATtiny25
6. Give the size of the on-chip program ROM in each of the following:
(a) ATtiny84 (b) ATmega32 (c) ATtiny25

See the following websites for AVR microcontrollers and AVR trainers:

<http://www.Atmel.com>

<http://www.MicroDigitalEd.com>

<http://www.digilentinc.com>

SUMMARY

This chapter discussed the role and importance of microcontrollers in everyday life. Microprocessors and microcontrollers were contrasted and compared. We discussed the use of microcontrollers in the embedded market. We also discussed criteria to consider in choosing a microcontroller such as speed, memory, I/O, packaging, and cost per unit. The second section of this chapter described various families of the AVR, such as Mega and Tiny, and their features. In addition, we discussed some of the most common AVR microcontrollers such as the ATmega32 and ATtiny25.

PROBLEMS

SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS

1. True or False. A general-purpose microprocessor has on-chip ROM.
2. True or False. Generally, a microcontroller has on-chip ROM.
3. True or False. A microcontroller has on-chip I/O ports.
4. True or False. A microcontroller has a fixed amount of RAM on the chip.
5. What components are usually put together with the microcontroller onto a single chip?
6. Intel's Pentium chips used in Windows PCs need external _____ and _____ chips to store data and code.
7. List three embedded products attached to a PC.
8. Why would someone want to use an x86 as an embedded processor?
9. Give the name and the manufacturer of some of the most widely used 8-bit microcontrollers.
10. In Question 9, which one has the most manufacture sources?
11. In a battery-based embedded product, what is the most important factor in choosing a microcontroller?
12. In an embedded controller with on-chip ROM, why does the size of the ROM matter?
13. In choosing a microcontroller, how important is it to have multiple sources for that chip?
14. What does the term "third-party support" mean?
15. Suppose that a microcontroller architecture has both 8-bit and 16-bit versions. Which of the following statements is true?
 - (a) The 8-bit software will run on the 16-bit system.
 - (b) The 16-bit software will run on the 8-bit system.

SECTION 1.2: OVERVIEW OF THE AVR FAMILY

16. What is the advantage of Flash memory over the other kinds of ROM?
17. The ATmega32 has _____ pins for I/O.
18. The ATmega32 has _____ bytes of on-chip program ROM.
19. The ATtiny44 has _____ bytes of on-chip data RAM.
20. The ATtiny44 has _____ ADCs.

21. The ATmega64 has _____ bytes of on-chip data RAM.
22. The ATmega1280 has _____ on-chip timer(s).
23. The ATmega32 has _____ bytes of on-chip data RAM.
24. Check the Atmel website to see if there is a RAMless version of the AVR. Give the part number if there is one.
25. Check the Atmel website to see if there is a ROMless version of the AVR. Give the part number if there is one.
26. Check the Atmel website to find three members of the AVR family that have USB controllers.
27. Check the Atmel website to find two members of the AVR family that have CAN controllers.
28. Give the amount of program ROM and data RAM for the following chips:
(a) ATmega32 (b) ATtiny44 (c) ATtiny84 (d) 90CAN128
29. What are the main differences between the ATmega16 and the ATmega32?
30. The ATmega16 has _____ bytes of data EEPROM.

ANSWERS TO REVIEW QUESTIONS

SECTION 1.1: MICROCONTROLLERS AND EMBEDDED PROCESSORS

1. True
2. A microcontroller-based system
3. (d)
4. (d)
5. It is dedicated because it does only one type of job.
6. Embedded system means that the application and the processor are combined into a single system.
7. Having multiple sources for a given part means you are not hostage to one supplier. More importantly, competition among suppliers brings about lower cost for that product.

SECTION 1.2: OVERVIEW OF THE AVR FAMILY

1. 64K of RAM space, 8M of on-chip ROM space, a large number of I/O pins, ADC, and different serial protocols such as SPI, USART, I2C, etc.
2. 8
3. Tiny, Mega, Classic, and special purpose
4. Tiny
5. (a) 2K bytes
(b) 128 bytes
6. (a) 8K bytes (b) 32K bytes (c) 2K bytes

CHAPTER 2

AVR ARCHITECTURE AND ASSEMBLY LANGUAGE PROGRAMMING

OBJECTIVES

Upon completion of this chapter, you will be able to:

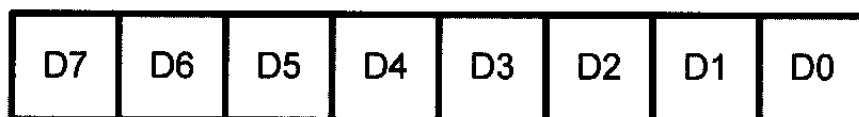
- >> List the registers of the AVR microcontroller
- >> Examine the data memory of the AVR microcontroller
- >> Perform simple operations, such as ADD and load, and access internal RAM memory in the AVR microcontroller
- >> Explain the purpose of the status register
- >> Discuss data RAM memory space allocation in the AVR microcontroller
- >> Code simple AVR Assembly language instructions
- >> Describe AVR data types and directives
- >> Assemble and run a AVR program using AVR Studio
- >> Describe the sequence of events that occur upon AVR power-up
- >> Examine programs in AVR ROM code
- >> Detail the execution of AVR Assembly language instructions
- >> Understand the RISC and Harvard architectures of the AVR microcontroller
- >> Examine the AVR's registers and data RAM using the AVR Studio simulator

CPUs use registers to store data temporarily. To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data. In Section 2.1 we look at the general purpose registers (GPRs) of the AVR. We demonstrate the use of GPRs with simple instructions such as LDI and ADD. Allocation of RAM memory inside the AVR and the addressing mode of the AVR are discussed in Sections 2.2 and 2.3. In Section 2.4 we discuss the status register's flag bits and how they are affected by arithmetic instructions. In Section 2.5 we look at some widely used Assembly language directives, pseudocode, and data types related to the AVR. In Section 2.6 we examine Assembly language and machine language programming and define terms such as mnemonics, opcode, operand, and so on. The process of assembling and creating a ready-to-run program for the AVR is discussed in Section 2.7. Step-by-step execution of an AVR program and the role of the program counter are examined in Section 2.8. The merits of RISC architecture are examined in Section 2.9. Section 2.10 discusses the AVR Studio.

SECTION 2.1: THE GENERAL PURPOSE REGISTERS IN THE AVR

CPUs use many registers to store data temporarily. To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data. In this section we look at the general purpose registers (GPRs) of the AVR and we demonstrate the use of GPRs with simple instructions such as LDI and ADD.

AVR microcontrollers have many registers for arithmetic and logic operations. In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched. The vast majority of AVR registers are 8-bit registers. In the AVR there is only one data type: 8-bit. The 8 bits of a register are shown in the diagram below. These range from the MSB (most-significant bit) D7



to the LSB (least-significant bit) D0. With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed.

In AVR there are 32 general purpose registers. They are R0–R31 and are located in the lowest location of memory address. See Figure 2-1. All of these registers are 8 bits.

The general purpose registers in AVR are the same as the accumulator in other microprocessors. They can be used by all arithmetic and logic instructions. To understand the use of the general purpose registers, we will show it in the context of two simple instructions: LDI and ADD.

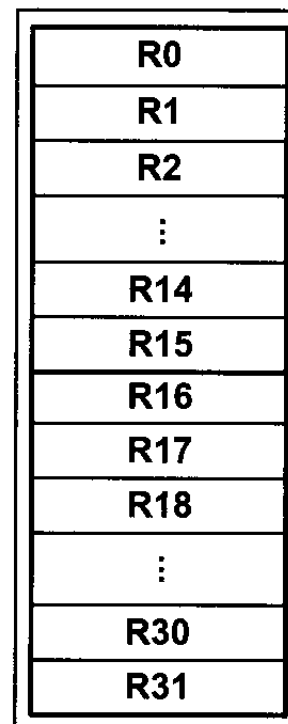


Figure 2-1. GPRs

LDI instruction

Simply stated, the LDI instruction copies 8-bit data into the general purpose registers. It has the following format:

```
LDI Rd,K      ;load Rd (destination) with Immediate value K
               ;d must be between 16 and 31
```

K is an 8-bit value that can be 0–255 in decimal, or 00–FF in hex, and Rd is R16 to R31 (any of the upper 16 general purpose registers). The I in LDI stands for “immediate.” If we see the word “immediate” in any instruction, we are dealing with a value that must be provided right there with the instruction. The following instruction loads the R20 register with a value of 0x25 (25 in hex).

```
LDI R20,0x25      ;load R20 with 0x25 (R20 = 0x25)
```

The following instruction loads the R31 register with the value 0x87 (87 in hex).

```
LDI R31,0x87      ;load 0x87 into R31 (R31 = 0x87)
```

The following instruction loads R25 with the value 0x15 (15 in hex and 21 in decimal).

```
LDI R25,0x79      ;load 0x79 into R25 (R25 = 0x79)
```

Note: We cannot load values into registers R0 to R15 using the LDI instruction. For example, the following instruction is not valid:

```
LDI R5,0x99      ;invalid instruction
```

Notice the position of the source and destination operands. As you can see, the LDI loads the right operand into the left operand. In other words, the destination comes first.

To write a comment in Assembly language we use ‘;’. It is the same as ‘//’ in C language, which causes the remainder of the line of code to be ignored. For instance, in the above examples the expressions mentioned after ‘;’ just explain the functionality of the instructions to you, and do not have any effects on the execution of the instructions.

When programming the GPRs of the AVR microcontroller with an immediate value, the following points should be noted:

1. If we want to present a number in hex, we put a dollar sign (\$) or a 0x in front of it. If we put nothing in front of a number, it is in decimal. For example, in “LDI R16,50”, R16 is loaded with 50 in decimal, whereas in “LDI R16,0x50”, R16 is loaded with 50 in hex.
2. If values 0 to F are moved into an 8-bit register such as GPRs, the rest of the bits are assumed to be all zeros. For example, in “LDI R16,0x5” the result will be R16 = 0x05; that is, R16 = 00000101 in binary.
3. Moving a value larger than 255 (FF in hex) into the GPRs will cause an error.

```
LDI R17, 0x7F2 ;ILLEGAL $7F2 > 8 bits ($FF)
```

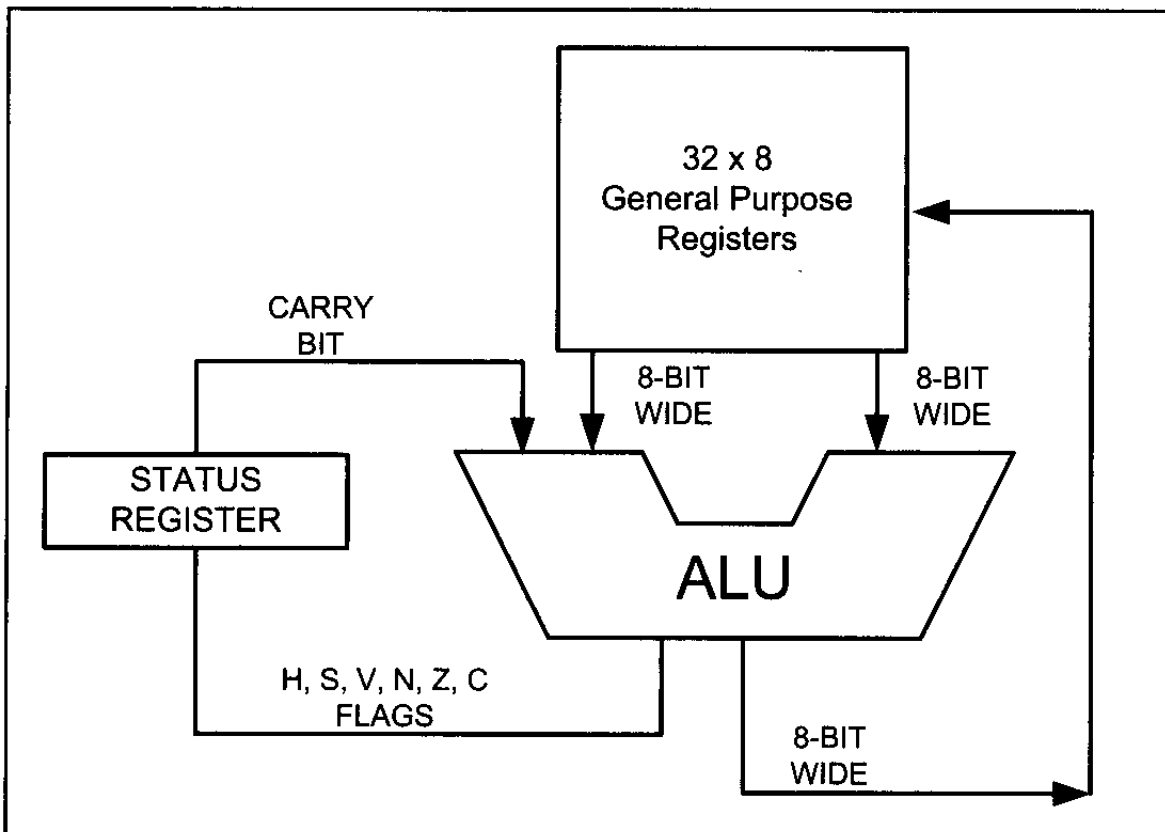


Figure 2-2. AVR General Purpose Registers and ALU

ADD instruction

The ADD instruction has the following format:

```
ADD Rd,Rr ;ADD Rr to Rd and store the result in Rd
```

The ADD instruction tells the CPU to add the value of Rr to Rd and put the result back into the Rd register. To add two numbers such as 0x25 and 0x34, one can do the following:

```
LDI R16,0x25      ;load 0x25 into R16
LDI R17,0x34      ;load 0x34 into R17
ADD R16,R17       ;add value R17 to R16 (R16 = R16 + R17)
```

Executing the above lines results in R16 = 0x59 (0x25 + 0x34 = 0x59)

Figure 2-2 shows the general purpose registers (GPRs) and the ALU in AVR. The affect of arithmetic and logic operations on the status register will be discussed in Section 2.4.

Review Questions

1. Write instructions to move the value 0x34 into the R29 register.
2. Write instructions to add the values 0x16 and 0xCD. Place the result in the R19 register.
3. True or false. No value can be moved directly into the GPRs.
4. What is the largest hex value that can be moved into an 8-bit register? What is the decimal equivalent of that hex value?
5. The vast majority of registers in the AVR are ____-bit.

SECTION 2.2: THE AVR DATA MEMORY

In AVR microcontrollers there are two kinds of memory space: code memory space and data memory space. Our program is stored in code memory space, whereas the data memory stores data. We will examine the code memory space in Section 2.8. In this section, we will discuss the data memory space. The data memory is composed of three parts: GPRs (general purpose registers), I/O memory, and internal data SRAM. See Figure 2-3.

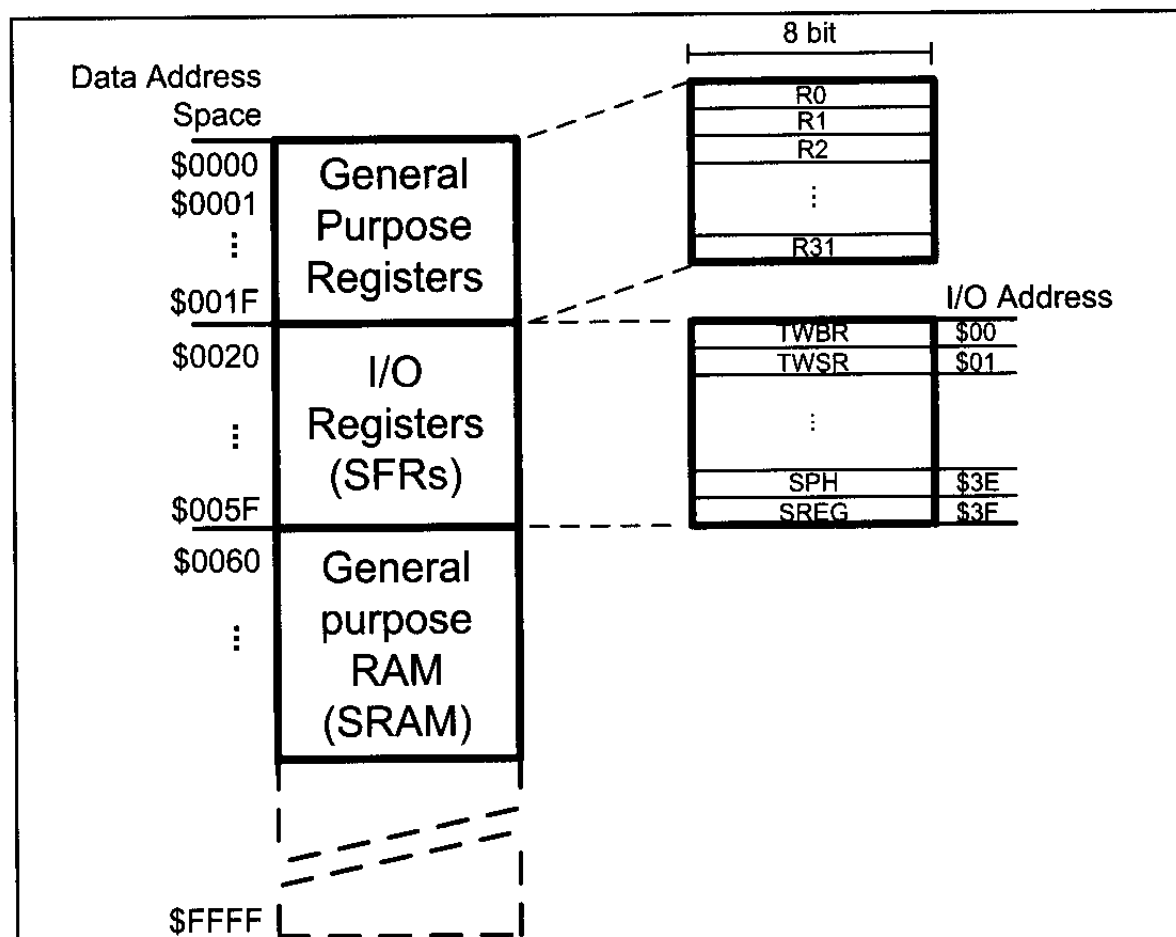


Figure 2-3. The Data Memory for AVR with No Extended I/O Memory

GPRs (general purpose registers)

As we discussed in the last section, the GPRs use 32 bytes of data memory space. They always take the address location \$00–\$1F in the data memory space, regardless of the AVR chip number. See Figure 2-3.

I/O memory (SFRs)

The I/O memory is dedicated to specific functions such as status register, timers, serial communication, I/O ports, ADC, and so on. The function of each I/O memory location is fixed by the CPU designer at the time of design because it is used for control of the microcontroller or peripherals. The AVR I/O memory is made of 8-bit registers. The number of locations in the data memory set aside for I/O memory depends on the pin numbers and peripheral functions supported by

that chip, although the number can vary from chip to chip even among members of the same family. However, all of the AVR chips have at least 64 bytes of I/O memory locations. This 64-byte section is called *standard I/O memory*. In AVR chips with more than 32 I/O pins (e.g., ATmega64, ATmega128, and ATmega256) there is also an extended I/O memory, which contains the registers for controlling the extra ports and the extra peripherals. See Figures 2-3 and 2-4. In other microcontrollers the I/O registers are called *SFRs* (*special function registers*) since each one is dedicated to a specific function. In contrast to SFRs, the GPRs do not have any specific function and are used for storing general data.

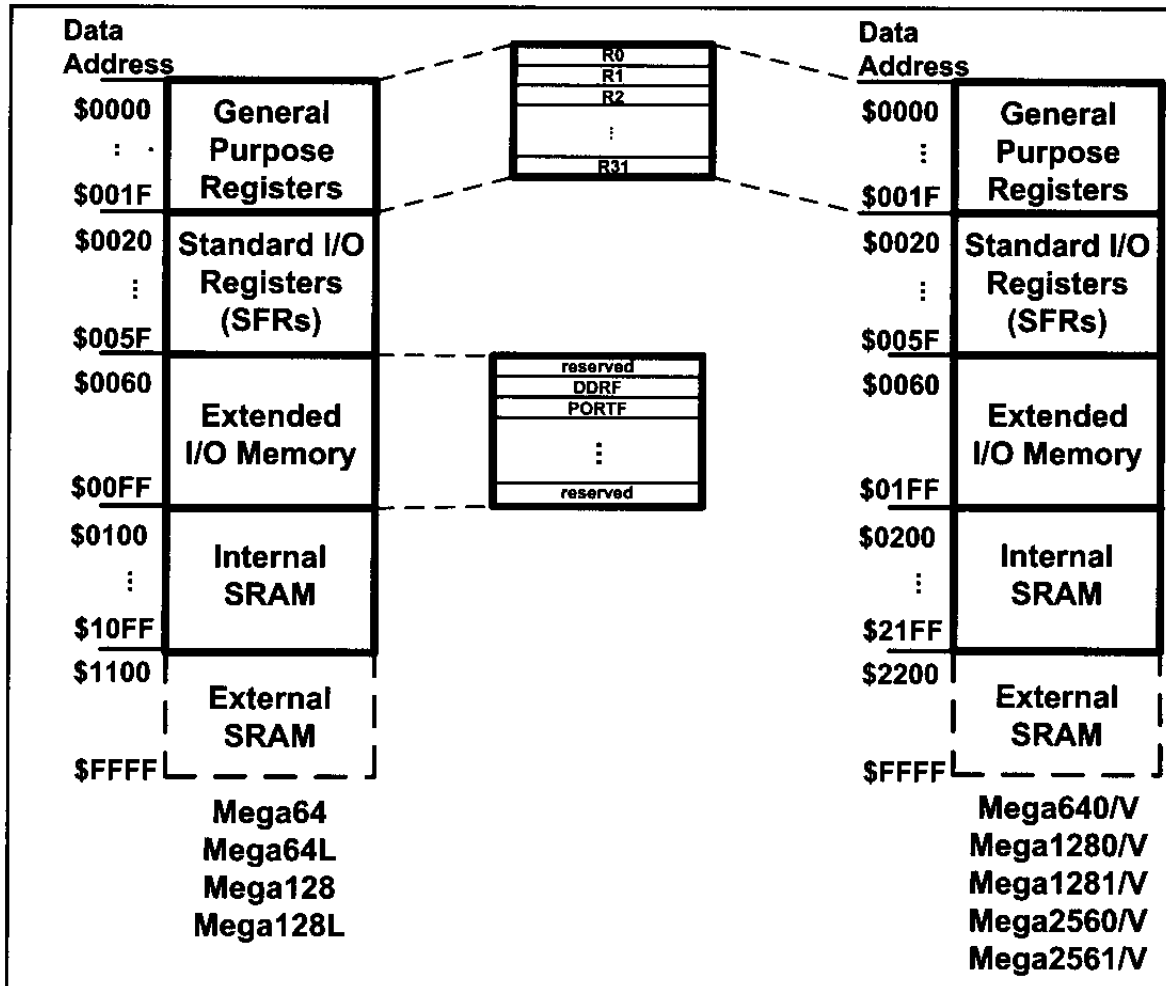


Figure 2-4. The Data Memory for the AVR chips with Extended I/O Memory

Internal data SRAM

Internal data SRAM is widely used for storing data and parameters by AVR programmers and C compilers. Generally, this is called *scratch pad*. Each location of the SRAM can be accessed directly by its address. We will use these locations in future chapters to store data brought into the CPU via I/O and serial ports. Each location is 8 bits wide and can be used to store any data we want as long as it is 8-bit. Again, the size of SRAM can vary from chip to chip, even among members of the same family. See Table 2-1 for a comparison of the data memories of various AVR chips. Also, see Figure 2-4.

SRAM vs. EEPROM in AVR chips

The AVR has an EEPROM memory that is used for storing data. As you saw in Chapter 0, EEPROM does not lose its data when power is off, whereas SRAM does. So, the EEPROM is used for storing data that should rarely be changed and should not be lost when the power is off (e.g., options and settings); whereas the SRAM is used for storing data and parameters that are changed frequently. The three parts of the data memory (GPRs, SFRs, and the internal SRAM) are made of SRAM. The EEPROM memory of AVR chips is covered in Chapter 6.

In AVR datasheets, EEPROM refers to the EEPROM's size, and SRAM is the internal SRAM size. By adding the sizes of GPR, SFRs (I/O registers), and SRAMs we get the data memory size. See Table 2-1.

Table 2-1: Data Memory Size for AVR Chips

	Data Memory (Bytes)	=	I/O Registers (Bytes)	+	SRAM (Bytes)	+	General Purpose Register
ATtiny25	224		64		128		32
ATtiny85	608		64		512		32
ATmega8	1120		64		1024		32
ATmega16	1120		64		1024		32
ATmega32	2144		64		2048		32
ATmega128	4352		64+160		4096		32
ATmega2560	8704		64+416		8192		32

Extracted from <http://www.atmel.com>

Review Questions

1. True or false. The I/O registers are used for storing data.
2. The GPRs together with I/O registers and SRAM are called _____.
3. The I/O registers in AVR are _____-bit.
4. The data memory space in AVR is divided into _____ parts.
5. The data memory space in AVR can be a maximum of _____ bytes.
6. The standard I/O memory space in AVR is _____ bytes.

SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY

The instructions we have used so far worked with the immediate (constant) value of K and the GPRs. They also used the GPRs as their destination. We saw simple examples of using LDI and ADD earlier in Section 2.1. The AVR allows direct access to other locations in the data memory. In this section we show the instructions accessing various locations of the data memory. This is one of the most important sections in the book for mastering the topic of AVR Assembly language programming.

LDS instruction (Load direct from data Space)

```
LDS    Rd, K    ;load Rd with the contents of location K (0 ≤ d ≤ 31)
                ;K is an address between $0000 to $FFFF
```

The LDS instruction tells the CPU to load (copy) one byte from an address in the data memory to the GPR. After this instruction is executed, the GPR will have the same value as the location in the data memory. The location in the data memory can be in any part of the data space; it can be one of the I/O registers, a location in the internal SRAM, or a GPR. For example, the “LDS R20,0x1” instruction will copy the contents of location 1 (in hex) into R20. As you can see in Figure 2-3, location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R1 to R20.

The following instruction loads R5 with the contents of location 0x200. As you can see in Figure 2-3, 0x200 is located in the internal SRAM:

```
LDS R5,0x200 ;load R5 with the contents of location $200
```

The following program adds the contents of location 0x300 to location 0x302. To do so, first it loads R0 with the contents of location 0x300 and R1 with the contents of location 0x302, then adds R0 to R1:

```
LDS    R0, 0x300    ;R0 = the contents of location 0x300
LDS    R1, 0x302    ;R1 = the contents of location 0x302
ADD    R1, R0        ;add R0 to R1
```

You can see the execution of “LDS R0,0x300” and “LDS R1,0x302” instructions in Figure 2-5. Figure 2-6 shows the contents of R0, R1 and locations 300 and 302 of data memory before and after the execution of each of the instructions, assuming that locations \$300 and \$302 contain a and β , respectively.

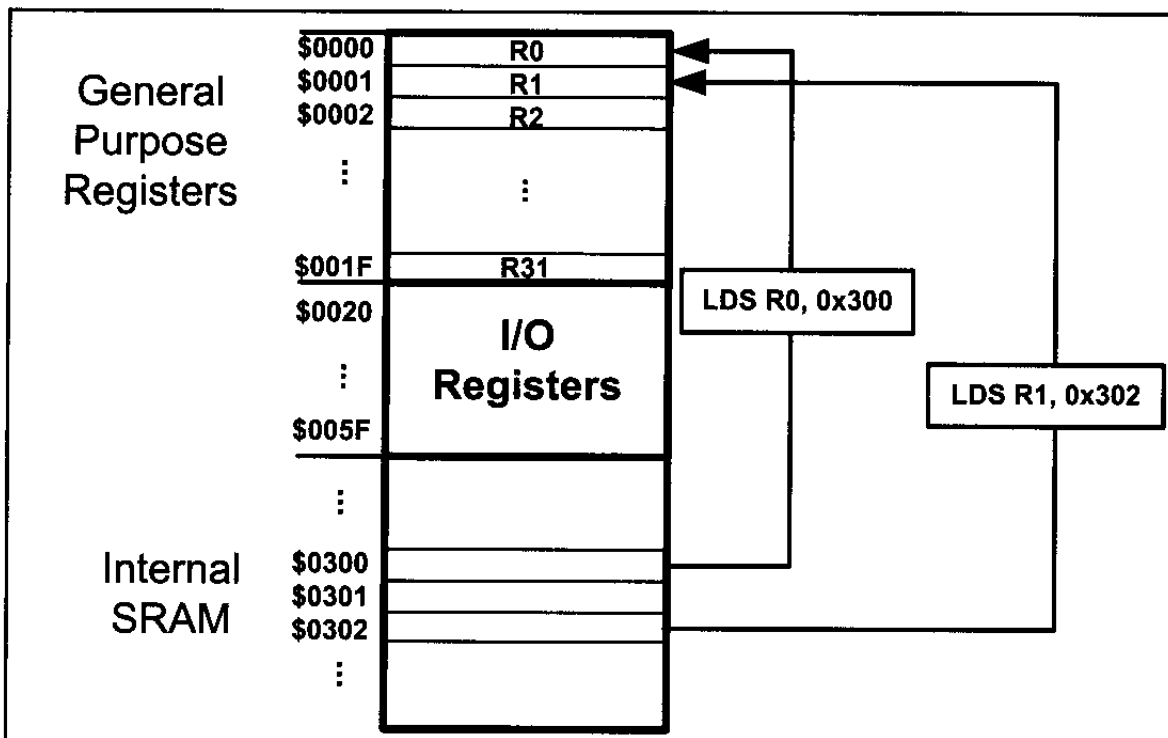


Figure 2-5. Execution of “LDS R0,0x300” and “LDS R1,0x302” Instructions

	R0	R1	Loc \$300	Loc \$302
Before LDS R0,0x300	?	?	α	β
After LDS R0,0x300	α	?	α	β
After LDS R1,0x302	α	β	α	β
After ADD R0, R1	$\alpha + \beta$	β	α	β

Figure 2-6. The Contents of R0, R1, and Locations \$300 and \$302

STS instruction (STore direct to data Space)

```
STS    K, Rr ;store register into location K
        ;K is an address between $0000 to $FFFF
```

The STS instruction tells the CPU to store (copy) the contents of the GPR to an address location in the data memory space. After this instruction is executed, the location in the data space will have the same value as the GPR. The location can be in any part of the data memory space; it can be one of the I/O registers, a location in the SRAM, or a GPR. For example, the “STS 0x1, R10” instruction will copy the contents of R10 into location 1. As you can see in Figure 2-3, location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R10 to R1.

The following instruction stores the contents of R25 to location 0x230. As you can see in Figure 2-3, 0x230 is located in the internal SRAM:

```
STS 0x230, R25 ;store R25 to data space location 0x230
```

The following program first loads the R16 register with value 0x55, then moves this value around to I/O registers of ports B, C, and D. As shown in Figure 2-7, the addresses of PORTB, PORTC, and PORTD are 0x38, 0x35, and 0x32, respectively:

```
LDI R16, 0x55 ;R16 = 55 (in hex)
STS 0x38, R16 ;copy R16 to Port B (PORTB = 0x55)
STS 0x35, R16 ;copy R16 to Port C (PORTC = 0x55)
STS 0x32, R16 ;copy R16 to Port D (PORTD = 0x55)
```

As we saw in Figure 2-3, PORTB, PORTC, and PORTD are part of the special function registers in the I/O memory. They can be connected to the I/O pins of the AVR microcontroller as we will see in Chapter 4. We can also store the contents of a GPR into any location in the SRAM region of the data space. The following program will put 0x99 into locations 0x200–0x203 of the SRAM region in the data memory:

	Address	Data
LDI R20, 0x99 ;R20 = 0x99	\$200	0x99
STS 0x200, R20 ;store R20 in loc 0x200	\$201	0x99
STS 0x201, R20 ;store R20 in loc 0x201	\$202	0x99
STS 0x202, R20	\$203	0x99
STS 0x203, R20 ;see the Mem. contents->		

Notice that you cannot copy (store) an immediate value directly into the SRAM location in the AVR. This must be done via the GPRs.

The following program adds the contents of location 0x220 to location 0x221, and stores the result in location 0x221:

```
LDS    R30, 0x220 ;load R30 with the contents of location 0x220
LDS    R31, 0x221 ;load R31 with the contents of location 0x221
ADD    R31, R30    ;add R30 to R31
STS    0x221, R31  ;store R31 to data space location 0x221
```

See Examples 2-1 and 2-2.

IN instruction (IN from I/O location)

IN Rd, A ;load an I/O location to the GPR ($0 \leq d \leq 31$), ($0 \leq A \leq 63$)

The IN instruction tells the CPU to load one byte from an I/O register to the GPR. After this instruction is executed, the GPR will have the same value as the I/O register. For example, the "IN R20, 0x16" instruction will copy the contents of location 16 (in hex) of the I/O memory into R20. As you can see in Figure 2-7, each location in I/O memory has two addresses: I/O address and data memory address. Each location in the data memory has a unique address called the *data memory address*. Each I/O register has a relative address in comparison to the beginning of the I/O memory; this address is called the *I/O address*. See Figure 2-3. You see the list of I/O registers in Figure 2-7.

Address		Name
Mem.	I/O	
\$20	\$00	TWBR
\$21	\$01	TWSR
\$22	\$02	TWAR
\$23	\$03	TWDR
\$24	\$04	ADCL
\$25	\$05	ADCH
\$26	\$06	ADCSRA
\$27	\$07	ADMUX
\$28	\$08	ACSR
\$29	\$09	UBRRL
\$2A	\$0A	UCSRB
\$2B	\$0B	UCSRA
\$2C	\$0C	UDR
\$2D	\$0D	SPCR
\$2E	\$0E	SPSR
\$2F	\$0F	SPDR
\$30	\$10	PIND
\$31	\$11	DDRD
\$32	\$12	PORTD
\$33	\$13	PINC
\$34	\$14	DDRC
\$35	\$15	PORTC

Address		Name
Mem.	I/O	
\$36	\$16	PINB
\$37	\$17	DDRB
\$38	\$18	PORTB
\$39	\$19	PINA
\$3A	\$1A	DDRA
\$3B	\$1B	PORTA
\$3C	\$1C	EECR
\$3D	\$1D	EEDR
\$3E	\$1E	EEARL
\$3F	\$1F	EEARH
\$40	\$20	UBRRC
		UBRRH
\$41	\$21	WDTCSR
\$42	\$22	ASSR
\$43	\$23	OCR2
\$44	\$24	TCNT2
\$45	\$25	TCCR2
\$46	\$26	ICR1L
\$47	\$27	ICR1H
\$48	\$28	OCR1BL
\$49	\$29	OCR1BH
\$4A	\$2A	OCR1AL

Address		Name
Mem.	I/O	
\$4B	\$2B	OCR1AH
\$4C	\$2C	TCNT1L
\$4D	\$2D	TCNT1H
\$4E	\$2E	TCCR1B
\$4F	\$2F	TCCR1A
\$50	\$30	SFIOR
\$51	\$31	OCDR
		OSCCAL
\$52	\$32	TCNT0
\$53	\$33	TCCR0
\$54	\$34	MCUCSR
\$55	\$35	MCUCR
\$56	\$36	TWCR
\$57	\$37	SPMCR
\$58	\$38	TIFR
\$59	\$39	TIMSK
\$5A	\$3A	GIFR
\$5B	\$3B	GICR
\$5C	\$3C	OCR0
\$5D	\$3D	SPL
\$5E	\$3E	SPH
\$5F	\$3F	SREG

Note: Although memory address \$20-\$5F is set aside for I/O registers (SFR) we can access them as I/O locations with addresses starting at \$00.

Figure 2-7. I/O Registers of the ATmega32 and Their Data Memory Address Locations

Example 2-1

State the contents of RAM locations \$212 to \$216 after the following program is executed:

```
LDI    R16, 0x99    ;load R16 with value 0x99
STS     0x212, R16
LDI     R16, 0x85    ;load R16 with value 0x85
STS     0x213, R16
LDI     R16, 0x3F    ;load R16 with value 0x3F
STS     0x214, R16
LDI     R16, 0x63    ;load R16 with value 0x63
STS     0x215, R16
LDI     R16, 0x12    ;load R16 with value 0x12
STS     0x216, R16
```

Solution:

After the execution of STS 0x212, R16 data memory location \$212 has value 0x99; after the execution of STS 0x213, R16 data memory location \$213 has value 0x85; after the execution of STS 0x214, R16 data memory location \$214 has value 0x3F; after the execution of STS 0x215, R16 data memory location \$215 has value 0x63; and so on, as shown in the chart.

Address	Data
\$212	0x99
\$213	0x85
\$214	0x3F
\$215	0x63
\$216	0x12

Example 2-2

State the contents of R20, R21, and data memory location 0x120 after the following program:

```
LDI     R20, 5      ;load R20 with 5
LDI     R21, 2      ;load R21 with 2
ADD     R20, R21     ;add R21 to R20
ADD     R20, R21     ;add R21 to R20
STS     0x120, R20   ;store in location 0x120 the contents of R20
```

Solution:

The program loads R20 with value 5. Then it loads R21 with value 2. Then it adds the R21 register to R20 twice. At the end, it stores the result in location 0x120 of data memory.

Location	Data	Location	Data	Location	Data	Location	Data	Location	Data
R20	5	R20	5	R20	7	R20	9	R20	9
R21		R21	2	R21	2	R21	2	R21	2
0x120		0x120		0x120		0x120		0x120	9
After LDI R20, 5		After LDI R21, 2		After ADD R20, R21		After ADD R20, R21		After STS 0x120, R20	

In the IN instruction, the I/O registers are referred to by their I/O addresses. For example, the “IN R20, 0x16” instruction will copy the contents of location \$16 of the I/O memory (whose data memory address is 0x36) into R20. As shown in Figure 2-7, I/O address 0x16 belongs to PINB, so the instruction copies the contents of PINB to R20.

The following instruction loads R19 with the contents of location 0x10 of the I/O memory:

```
IN R19, 0x10      ;load R19 with location $10 (R19 = PIND)
```

To work with the I/O registers more easily, we can use their names instead of their I/O addresses. For example, the following instruction loads R19 with the contents of PIND:

```
IN R19, PIND      ;load R19 with PIND
```

Notice that to be able to use the names of the I/O addresses instead of the I/O addresses we should include the proper header files, as discussed in Section 2.5. The details of I/O ports are discussed in Chapter 4.

The following program adds the contents of PIND to PINB, and stores the result in location 0x300 of the data memory:

```
IN    R1, PIND      ;load R1 with PIND
IN    R2, PINB      ;load R2 with PINB
ADD   R1, R2        ;R1 = R1 + R2
STS   0x300, R1     ;store R1 to data space location $300
```

IN vs. LDS

As we mentioned earlier, we can use the LDS instruction to copy the contents of a memory location to a GPR. This means that we can load an I/O register into a GPR, using the LDS instruction. So, what is the advantage of using the IN instruction for reading the contents of I/O registers over using the LDS instruction? The IN instruction has the following advantages:

1. The CPU executes the IN instruction faster than LDS. As you will see in Chapter 3, the IN instruction lasts 1 machine cycle, whereas LDS lasts 2 machine cycles.
2. The IN is a 2-byte instruction, whereas LDS is a 4-byte instruction. This means that the IN instruction occupies less code memory.
3. When we use the IN instruction, we can use the names of the I/O registers instead of their addresses.
4. The IN instruction is available in all of the AVRs, whereas LDS is not implemented in some of the AVRs.

Notice that in using the IN instruction we can access only the standard I/O memory, while we can access all parts of the data memory using the LDS instruction.

OUT instruction (OUT to I/O location)

OUT A, Rr ;store register to I/O location ($0 \leq r \leq 31$), ($0 \leq A \leq 63$)

The OUT instruction tells the CPU to store the GPR to the I/O register. After the instruction is executed, the I/O register will have the same value as the GPR. For example, the “OUT PORTD, R10” instruction will copy the contents of R10 into PORTD (location 12 of the I/O memory).

Notice that in the OUT instruction, the I/O registers are referred to by their I/O addresses (like the IN instruction).

The following program copies 0xE6 to the SPL register:

```
LDI    R20, 0xE6    ;load R20 with 0xE6
OUT     SPL, R20     ;out R20 to SPL
```

We must remember that we cannot copy an immediate value to an I/O register nor to an SRAM location.

The following program copies PIND to PORTA:

```
IN      R0, PIND     ;load R20 with the contents of I/O reg PIND
OUT     PORTA, R0    ;out R20 to PORTA
```

In Example 2-3 we use JMP to repeat an action indefinitely. JMP is similar to “goto” in the C language. We will study looping in Chapter 3.

Example 2-3

Write a program to get data from the PINB and send it to the I/O register of PORT C continuously.

Solution:

```
AGAIN: IN    R16, PINB    ;bring data from PortB into R16
      OUT    PORTC, R16   ;send it to Port C
      JMP    AGAIN       ;keep doing it forever
```

MOV instruction

The MOV instruction is used to copy data among the GPR registers of R0–R31. It has the following format:

```
MOV    Rd, Rr          ;Rd = Rr (copy Rr to Rd)
                        ;Rd and Rr can be any of the GPRs
```

For example, the following instruction copies the contents of R20 to R10:

```
MOV    R10, R20        ;R10 = R20
```

For instance, if R20 contains 60, after execution of the above instruction both R20 and R10 will contain 60.

More ALU instructions involving the GPRs

The following program adds 0x19 to the contents of location 0x220 and stores the result in location 0x221:

```
LDI    R20, 0x19    ;load R20 with 0x19
LDS    R21, 0x220    ;load R21 with the contents of location 0x220
ADD    R21, R20      ;R21 = R21 + R20
STS    0x221, R21    ;store R21 to location 0x221
```

INC instruction

```
INC    Rd            ;increment the contents of Rd by one (0 ≤ d ≤ 31)
```

The INC instruction increments the contents of Rd by 1. For example, the following instruction adds 1 to the contents of R2:

```
INC    R2            ;R2 = R2 + 1
```

The following program increments the contents of data memory location 0x430 by 1:

```
LDS    R20, 0x430    ;R20 = contents of location 0x430
INC    R20            ;R20 = R20 + 1
STS    0x430, R20    ;store R20 to location 0x430
```

SUB instruction

The SUB instruction has the following format:

```
SUB    Rd,Rr          ;Rd = Rd - Rr
```

The SUB instruction tells the CPU to subtract the value of Rr from Rd and put the result back into the Rd register. To subtract 0x25 from 0x34, one can do the following:

```
LDI    R20, 0x34      ;R20 = 0x34
LDI    R21, 0x25      ;R21 = 0x25
SUB    R20, R21        ;R20 = R20 - R21
```

The following program subtracts 5 from the contents of location 0x300 and stores the result in location 0x320:

```
LDS    R0, 0x300      ;R0 = contents of location 0x300
LDI    R16, 0x5        ;R16 = 0x5
SUB    R0, R16         ;R0 = R0 - R16
STS    0x320, R0       ;store the contents of R0 to location 0x320
```

The following program decrements the contents of R10, by 1:

```
LDI    R16, 0x1       ;load 1 to R16
SUB    R10, R16        ;R10 = R10 - R16
```

Table 2-2: ALU Instructions Using Two GPRs

Instruction		
ADD	Rd, Rr	ADD Rd and Rr
ADC	Rd, Rr	ADD Rd and Rr with Carry
AND	Rd, Rr	AND Rd with Rr
EOR	Rd, Rr	Exclusive OR Rd with Rr
OR	Rd, Rr	OR Rd with Rr
SBC	Rd, Rr	Subtract Rr from Rd with carry
SUB	Rd, Rr	Subtract Rr from Rd without carry

Rd and Rr can be any of the GPRs. See Chapter 5 for examples of the instructions in Table 2-2.

DEC instruction

The DEC instruction has the following format:

```
DEC    Rd                ;Rd = Rd - 1
```

The DEC instruction decrements (subtracts 1 from) the contents of Rd and puts the result back into the Rd register. For example, the following instruction subtracts 1 from the contents of R10:

```
DEC    R10                ;R10 = R10 - 1
```

In the following program, we put the value 3 into R30. Then the value in R30 is decremented.

```
LDI    R30, 3            ;R30 = 3
DEC     R30               ;R30 has 2
DEC     R30               ;R30 has 1
DEC     R30               ;R30 has 0
```

In the next chapter we will use the DEC instruction for looping.

Table 2-3: Some Instructions Using a GPR as Operand

Instruction		
CLR	Rd	Clear Register Rd
INC	Rd	Increment Rd
DEC	Rd	Decrement Rd
COM	Rd	One's Complement Rd
NEG	Rd	Negative (two's complement) Rd
ROL	Rd	Rotate left Rd through carry
ROR	Rd	Rotate right Rd through carry
LSL	Rd	Logical Shift Left Rd
LSR	Rd	Logical Shift Right Rd
ASR	Rd	Arithmetic Shift Right Rd
SWAP	Rd	Swap nibbles in Rd

Chapters 3 through 6 will show how to use the instructions in Table 2-3.

COM instruction

The “COM Rd” instruction complements (inverts) the contents of Rd and places the result back into the Rd register. In the following program, we put 0x55 into R16 and then send it to the SFR location of PORTB. Then the content of R16 is complemented, which becomes AA in hex. The 01010101 (0x55) is inverted and becomes 10101010 (0xAA).

```
LDI    R16,0x55    ;R16 = 0x55
OUT     PORTB, R16  ;copy R16 to Port B SFR (PB = 0x55)
COM     R16         ;complement R16          (R16 = 0xAA)
OUT     PORTB, R16  ;copy R16 to Port B SFR (PB = 0xAA)
```

Examine Example 2-4.

Example 2-4

Write a simple program to toggle the I/O register of PORT B continuously forever.

Solution:

```
LDI     R20, 0x55    ;R20 = 0x55
OUT      PORTB, R20   ;move R20 to Port B SFR (PB = 0x55)
L1:     COM     R20    ;complement R20
OUT      PORTB, R20   ;move R20 to Port B SFR
JMP      L1          ;repeat forever (see Chapter 3 for JMP)
```

The above concepts are important and must be understood since there are a large number of instructions with these formats.

Regarding Tables 2-2 and 2-3 the following points must be noted:

1. The instructions in Table 2-2 operate on two GPR registers of source (Rr) and destination (rd) and then place the result in the destination register (Rd)
2. The instructions in Table 2-3 operate on a single GPR register and place the result in the same register.

Review Questions

1. True or false. No value can be loaded directly into internal SRAM.
2. Write instructions to load value 0x95 into the *SPL* I/O register.
3. Write instructions to add 2 to the contents of R18.
4. Write instructions to add the values 0x16 and 0xCD. Place the result in location 0x400 of the data memory.
5. What is the largest hex value that can be moved into a location in the data memory? What is the decimal equivalent of the hex value?
6. “ADD R16, R3” puts the result in _____.
7. What does “OUT OCR0, R23” do?
8. What is wrong with “STS OCR0, R23”? What does it do?

SECTION 2.4: AVR STATUS REGISTER

Like all other microprocessors, the AVR has a flag register to indicate arithmetic conditions such as the carry bit. The flag register in the AVR is called the *status register (SReg)*. In this section, we discuss various bits of this register and provide some examples of how it is altered. Chapters 3 and 5 show how the flag bits of the status register are used.

AVR status register

The status register is an 8-bit register. It is also referred to as the *flag register*. See Figure 2-8 for the bits of the status register. The bits C, Z, N, V, S, and H are called *conditional flags*, meaning that they indicate some conditions that result after an instruction is executed. Each of the conditional flags can be used to perform a conditional branch (jump), as we will see in Chapters 3 and 5.

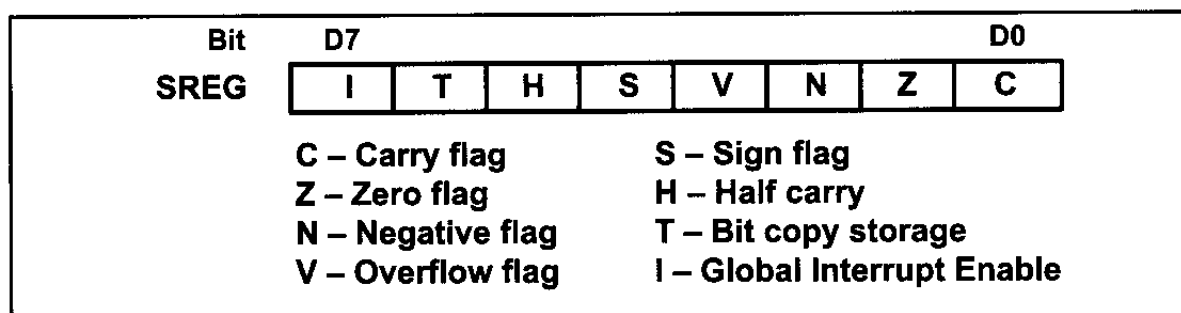


Figure 2-8. Bits of Status Register (SREG)

The following is a brief explanation of the flag bits of the status register. The impact of instructions on this register is then discussed.

C, the carry flag

This flag is set whenever there is a carry out from the D7 bit. This flag bit is affected after an 8-bit addition or subtraction. Chapter 5 shows how the carry flag is used.

Z, the zero flag

The zero flag reflects the result of an arithmetic or logic operation. If the result is zero, then $Z = 1$. Therefore, $Z = 0$ if the result is not zero. See Chapter 3 to see how we use the Z flag for looping.

N, the negative flag

Binary representation of signed numbers uses D7 as the sign bit. The negative flag reflects the result of an arithmetic operation. If the D7 bit of the result is zero, then $N = 0$ and the result is positive. If the D7 bit is one, then $N = 1$ and the result is negative. The negative and V flag bits are used for the signed number arithmetic operations and are discussed in Chapter 5.

V, the overflow flag

This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations while the overflow

flag is used to detect errors in signed arithmetic operations. The V and N flag bits are used for signed number arithmetic operations and are discussed in Chapter 5.

S, the Sign bit

This flag is the result of Exclusive-ORing of N and V flags. See Chapter 5 for more information.

H, Half carry flag

If there is a carry from D3 to D4 during an ADD or SUB operation, this bit is set; otherwise, it is cleared. This flag bit is used by instructions that perform BCD (binary coded decimal) arithmetic. In some microprocessors this is called the AC flag (Auxiliary Carry flag). See Chapter 5 for more information.

The T flag bit is discussed in Chapter 6 while Chapter 10 covers the I flag.

ADD instruction and the status register

Next we examine the impact of the ADD instruction on the flag bits C, H, and Z of the status register. Some examples should clarify their meanings. Although all the flag bits C, Z, H, V, S, and N are affected by the ADD instruction, we will focus on flags C, H, and Z for now. The other flag bits are discussed in Chapter 5, because they relate only to signed number operations. Examine Example 2-5 to see the impact of the DEC instruction on selected flag bits. See also Examples 2-6 through 2-8 to see the impact of the ADD instruction on selected flag bits.

Example 2-5

Show the status of the Z flag during the execution of the following program:

```
LDI    R20, 4        ;R20 = 4
DEC     R20           ;R20 = R20 - 1
DEC     R20           ;R20 = R20 - 1
DEC     R20           ;R20 = R20 - 1
DEC     R20           ;R20 = R20 - 1
```

Solution:

The Z flag is one when the result is zero. Otherwise, it is cleared (zero). Thus:

After	Value of R20	The Z flag
LDI R20, 4	4	0
DEC R20	3	0
DEC R20	2	0
DEC R20	1	0
DEC R20	0	1

Example 2-6

Show the status of the C, H, and Z flags after the addition of 0x38 and 0x2F in the following instructions:

```
LDI    R16, 0x38
LDI    R17, 0x2F
ADD    R16, R17    ;add R17 to R16
```

Solution:

\$38	0011 1000	
+ \$2F	<u>0010 1111</u>	
\$67	0110 0111	R16 = 0x67

C = 0 because there is no carry beyond the D7 bit.

H = 1 because there is a carry from the D3 to the D4 bit.

Z = 0 because the R16 (the result) has a value other than 0 after the addition.

Example 2-7

Show the status of the C, H, and Z flags after the addition of 0x9C and 0x64 in the following instructions:

```
LDI    R20, 0x9C
LDI    R21, 0x64
ADD    R20, R21    ;add R21 to R20
```

Solution:

\$9C	1001 1100	
+ \$64	<u>0110 0100</u>	
\$100	0000 0000	R20 = 00

C = 1 because there is a carry beyond the D7 bit.

H = 1 because there is a carry from the D3 to the D4 bit.

Z = 1 because the R20 (the result) has value 0 in it after the addition.

Example 2-8

Show the status of the C, H, and Z flags after the addition of 0x88 and 0x93 in the following instructions:

```
LDI    R20, 0x88
LDI    R21, 0x93
ADD    R20, R21    ;add R21 to R20
```

Solution:

\$ 88	1000 1000	
+ \$ 93	<u>1001 0011</u>	
\$11B	0001 1011	R20 = 0x1B

C = 1 because there is a carry beyond the D7 bit.

H = 0 because there is no carry from the D3 to the D4 bit.

Z = 0 because the R20 has a value other than 0 after the addition.

Not all instructions affect the flags

Some instructions affect all the six flag bits C, H, Z, S, V, and N (e.g., ADD). But some instructions affect no flag bits at all. The load instructions are in this category. And some instructions affect only some of the flag bits. The logic instructions (e.g., AND) are in this category.

Table 2-4 shows the instructions and the flag bits affected by them. Appendix A provides a complete list of all the instructions and their associated flag bits.

Table 2-4: Instructions That Affect Flag Bits

Instruction	C	Z	N	V	S	H
ADD	X	X	X	X	X	X
ADC	X	X	X	X	X	X
ADIW	X	X	X	X	X	
AND		X	X	X	X	
ANDI		X	X	X	X	
CBR		X	X	X	X	
CLR		X	X	X	X	
COM	X	X	X	X	X	
DEC		X	X	X	X	
EOR		X	X	X	X	
FMUL	X	X				
INC		X	X	X	X	
LSL	X	X	X	X		X
LSR	X	X	X	X		
OR		X	X	X	X	
ORI		X	X	X	X	
ROL	X	X	X	X		X
ROR	X	X	X	X		
SEN			1			
SEZ		1				
SUB	X	X	X	X	X	X
SUBI	X	X	X	X	X	X
TST		X	X	X	X	

Note: X can be 0 or 1. (See Chapter 5 for how to use these instructions.)

Flag bits and decision making

There are instructions that will make a conditional jump (branch) based on the status of the flag bits. Table 2-5 provides some of these instructions. Chapter 3 discusses the conditional branch instructions and how they are used.

Table 2-5: AVR Branch (Jump) Instructions Using Flag Bits

Instruction	Action
BRLO	Branch if C = 1
BRSH	Branch if C = 0
BREQ	Branch if Z = 1
BRNE	Branch if Z = 0
BRMI	Branch if N = 1
BRPL	Branch if N = 0
BRVS	Branch if V = 1
BRVC	Branch if V = 0

Review Questions

1. The flag register in the AVR is called the _____.
2. What is the size of the flag register in the AVR?
3. Find the C, Z, and H flag bits for the following code:

```
LDI    R20, 0x9F
LDI    R21, 0x61
ADD    R20, R21
```

4. Find the C, Z, and H flag bits for the following code:

```
LDI    R17, 0x82
LDI    R23, 0x22
ADD    R17, R23
```

5. Find the C, Z, and H flag bits for the following code:

```
LDI    R20, 0x67
LDI    R21, 0x99
ADD    R20, R21
```

SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES

In this section we look at some widely used data formats and directives supported by the AVR assembler.

AVR data type

The AVR microcontroller has only one data type. It is 8 bits, and the size of each register is also 8 bits. It is the job of the programmer to break down data larger than 8 bits (00 to 0xFF, or 0 to 255 in decimal) to be processed by the CPU. For examples of how to process data larger than 8 bits, see Chapter 5. The data types used by the AVR can be positive or negative. A discussion of signed numbers is given in Chapter 5 also. The bit-addressable data is discussed in Chapters 4 and 6.

Data format representation

There are four ways to represent a byte of data in the AVR assembler. The numbers can be in hex, binary, decimal, or ASCII formats. The following are examples of how each works.

Hex numbers

There are two ways to show hex numbers:

1. Put 0x (or 0X) in front of the number like this: `LDI R16, 0x99`
2. Put \$ in front of the number, like this: `LDI R22, $99`

We use both of these methods in this book, because many application notes out there use one of them and we need to get used to them.

Here are a few lines of code that use the hex format:

```
LDI    R28,$75      ;R28 = 0x75
SUBI   R28,0x11     ;R28 = 0x75 - 0x11 = 0x64
SUBI   R28,0x20     ;R28 = 0x64 - 0x20 = 0x44
ANDI   R28,0xF      ;R28 = 0x44 - 0x0F = 0x35
```

Binary numbers

There is only one way to represent binary numbers in an AVR assembler. It is as follows:

```
LDI R16,0b10011001      ;R16 = 10011001 or 99 in hex
```

The uppercase B will also work. Here are some examples of how to use it:

```
LDI    R23,0b00100101   ;R23 = $25
SUBI   R23,0B00010001   ;R23 = $25 - $11 = $14
```

Decimal numbers

To indicate decimal numbers in an AVR assembler we simply use the decimal (e.g., 12) and nothing before or after it. Here are some examples of how to use it:

```
LDI    R17, 12      ;R17 = 00001100 or 0C in hex
SUBI   R17, 2       ;R17 = 12 - 2 = 10 where 10 is equal to 0x0A
```

ASCII characters

To represent ASCII data in an AVR assembler we use single quotes as follows:

```
LDI R23,'2'      ;R23 = 00110010 or 32 in hex (See Appendix F)
```

This is the same as other assemblers such as the 8051 and x86. Here are some more examples:

```
LDI    R20,'9';R20 = 0x39, which is hex number for ASCII '9'
SUBI   R20,'1';R20 = 0x39 - 0x31 = 0x8
        ;(31 hex is for ASCII '1')
```

To represent a string, double quotes are used; and for defining ASCII strings (more than one character), we use the .DB (define byte) directive. We will see .DB usage in Chapter 6.

Assembler directives

While instructions tell the CPU what to do, directives (also called *pseudo-instructions*) give directions to the assembler. For example, the LDI and ADD instructions are commands to the CPU, but .EQU, .DEVICE, and .ORG are directives to the assembler. The following sections present some more widely used directives of the AVR and how they are used. The directives help us develop our program easier and make our program legible (more readable).

.EQU (equate)

This is used to define a constant value or a fixed address. The .EQU directive does not set aside storage for a data item, but associates a constant number with a data or an address label so that when the label appears in the program, its constant will be substituted for the label. The following uses .EQU for the counter constant, and then the constant is used to load the R21 register:

```
.EQU COUNT = 0x25
...
LDI R21, COUNT ;R21 = 0x25
```

When executing the above instruction “LDI R21, COUNT”, the register R21 will be loaded with the value 25H. What is the advantage of using .EQU? Assume that a constant (a fixed value) is used throughout the program, and the programmer wants to change its value everywhere. By the use of .EQU, the programmer can change it once and the assembler will change all of its occurrences throughout the program. This allows the programmer to avoid searching the entire program trying to find every occurrence.

We mentioned earlier that we can use the names of the I/O registers instead of their addresses (e.g., we can write “OUT PORTA,R20” instead of “OUT 0x1B,R20”). This is done with the help of the .EQU directive. In include files such as M32DEF.INC the I/O register names are associated with their addresses using the .EQU directive. For example, in M32DEF.INC the following pseudo-instruction exists, which associates 0x1B (the address of PORTB) with the PORTB.

```
.EQU PORTB = 0x1B
```

.SET

This directive is used to define a constant value or a fixed address. In this regard, the .SET and .EQU directives are identical. The only difference is that the value assigned by the .SET directive may be reassigned later.

Using .EQU for fixed data assignment

To get more practice using .EQU to assign fixed data, examine the following:

```
                ;in hexadecimal
.EQU DATA1 = 0x39      ;one way to define hex value
.EQU DATA2 = $39       ;another way to define hex value

                ;in binary
```

```
.EQU DATA3 = 0b00110101    ;binary (35 in hex)

                                ;in decimal
.EQU DATA4 = 39              ;decimal numbers (27 in hex)

                                ;in ASCII
.EQU DATA5 = '2'             ;ASCII characters
```

We use `.DB` to allocate code ROM memory locations for fixed data such as ASCII strings. See Chapter 6 for more examples.

Using `.EQU` for SFR address assignment

`.EQU` is also widely used to assign SFR addresses. Examine the following code:

```
.EQU COUNTER = 0x00           ;counter value 00
.EQU PORTB = 0x18             ;SFR Port B address
LDI R16, COUNTER              ;R16 = 0x00
OUT PORTB, R16                ;Port B (loc 0x18) now has 00 too
```

Using `.EQU` for RAM address assignment

Another common usage of `.EQU` is for the address assignment of the internal SRAM. Examine the following rewrite of an earlier example using `.EQU`:

```
.EQU SUM = 0x120              ;assign RAM loc to SUM
LDI R20, 5                    ;load R20 with 5
LDI R21, 2                    ;load R21 with 2
ADD R20, R21                  ;R20 = R20 + R21
ADD R20, R21                  ;R20 = R20 + R21
STS SUM, R20                  ;store the result in loc 0x120
```

This is especially helpful when the address needs to be changed in order to use a different AVR chip for a given project. It is much easier to refer to a name than a number when accessing RAM address locations.

`.ORG` (origin)

The `.ORG` directive is used to indicate the beginning of the address. It can be used for both code and data.

`.INCLUDE` directive

The `.include` directive tells the AVR assembler to add the contents of a file to our program (like the `#include` directive in C language). In Table 2-6, you see the files that you must include whenever you want to use any of the AVRs.

For example, when you want to use ATmega32, you must write the following instruction at the beginning of your program:

```
.INCLUDE "M32DEF.INC"
```

Table 2-6: Some of the Common AVR's and Their Include Files

ATMEGA		ATTINY		Special Purpose	
ATmega8	m8def.inc	ATtiny11	tn11def.inc	AT90CAN32	can32def.inc
ATmega16	m16def.inc	ATtiny12	tn12def.inc	AT90CAN64	can64def.inc
ATmega32	m32def.inc	ATtiny22	tn22def.inc	AT90PWM2	pwm2def.inc
ATmega64	m64def.inc	ATtiny44	tn44def.inc	AT90PWM3	pwm3def.inc
ATmega128	m128def.inc	ATtiny85	tn85def.inc	AT90USB646	usb646def.inc
ATmega256	m256def.inc				
ATmega2560	m2560def.inc				

Rules for labels in Assembly language

By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain. There are several rules that names must follow. First, each label name must be unique. The names used for labels in Assembly language programming consist of alphabetic letters in both uppercase and lowercase, the digits 0 through 9, and the special characters question mark (?), period (.), at (@), underline (_), and dollar sign (\$). The first character of the label must be an alphabetic character. In other words, it cannot be a number. Every assembler has some reserved words that must not be used as labels in the program. Foremost among the reserved words are the mnemonics for the instructions. For example, “LDI” and “ADD” are reserved because they are instruction mnemonics. In addition to the mnemonics there are some other reserved words. Check your assembler for the list of reserved words.

Review Questions

1. Give two ways for hex data representation in the AVR assembler.
2. Show how to represent decimal 99 in formats of (a) hex, (b) decimal, and (c) binary in the AVR assembler.
3. What is the advantage in using the .EQU directive to define a constant value?
4. Show the hex number value used by the following directives:
(a) .EQU ASC_DATA = '4' (b) .EQU MY_DATA=0B00011111
5. Give the value in R22 for the following:

```
.EQU MYCOUNT = 15
LDI R22, MYCOUNT
```
6. Give the value in data memory location 0x200 for the following:

```
.EQU MYCOUNT = 0x95
.EQU MYREG = 0x200
LDI R22, MYCOUNT
STS MYREG, R22
```
7. Give the value in data memory 0x63 for the following:

```
.EQU MYDATA = 12
.EQU MYREG = 0x63
.EQU FACTOR = 0x10
LDI R19, MYDATA
ADD R19, FACTOR
STS MYREG, R19
```

SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING

In this section we discuss Assembly language format and define some widely used terminology associated with Assembly language programming.

While the CPU can work only in binary, it can do so at a very high speed. It is quite tedious and slow for humans, however, to deal with 0s and 1s in order to program the computer. A program that consists of 0s and 1s is called *machine language*. In the early days of the computer, programmers coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, Assembly languages were developed, which provided mnemonics for the machine code instructions, plus other features that made programming faster and less prone to error. The term *mnemonic* is frequently used in computer science and engineering literature to refer to codes and abbreviations that are relatively easy to remember. Assembly language programs must be translated into machine code by a program called an *assembler*. Assembly language is referred to as a *low-level language* because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know all the registers of the CPU and the size of each, as well as other details.

Today, one can use many different programming languages, such as BASIC, Pascal, C, C++, Java, and numerous others. These languages are called *high-level languages* because the programmer does not have to be concerned with the internal details of the CPU. Whereas an *assembler* is used to translate an Assembly language program into machine code (sometimes also called *object code* or opcode for operation code), high-level languages are translated into machine code by a program called a *compiler*. For instance, to write a program in C, one must use a C compiler to translate the program into machine language. Next we look at AVR Assembly language format.

Structure of Assembly language

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions. An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items.

An Assembly language program (see Program 2-1) is a series of statements, or lines, which are either Assembly language instructions such as ADD and LDI, or statements called *directives*. While instructions tell the CPU what to do, directives (also called *pseudo-instructions*) give directions to the assembler. For example, in Program 2-1, while the LDI and ADD instructions are commands to the CPU, .ORG and .EQU are directives to the assembler. The directive .ORG tells the assembler to place the opcode at memory location 0, while .EQU introduces a new expression equal to a known expression.

An Assembly language instruction consists of four fields:

```
[ label:]      mnemonic  [ operands]  [ ;comment]
```

```

;AVR Assembly Language Program To Add Some Data.
;store SUM in SRAM location 0x300.

.EQU SUM    = 0x300      ;SRAM loc $300 for SUM

.ORG 00              ;start at address 0
LDI R16, 0x25        ;R16 = 0x25
LDI R17, $34         ;R17 = 0x34
LDI R18, 0b00110001  ;R18 = 0x31
ADD R16, R17         ;add R17 to R16
ADD R16, R18         ;add R18 to R16
LDI R17, 11          ;R17 = 0x0B
ADD R16, R17         ;add R17 to R16
STS SUM, R16         ;save the SUM in loc $300
HERE: JMP HERE       ;stay here forever

```

Program 2-1: Sample of an Assembly Language Program

Brackets indicate that a field is optional and not all lines have them. Brackets should not be typed in. Regarding the above format, the following points should be noted:

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.
2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

```

LDI    R23, $55
ADD    R23, R19
SUBI   R23, $67

```

ADD and LDI are the mnemonics that produce opcodes; the “\$55” and “\$67” are the operands. Instead of a mnemonic and an operand, these two fields could contain assembler pseudo-instructions, or directives. Remember that directives do not generate any machine code (opcode) and are used only by the assembler, as opposed to instructions that are translated into machine code (opcode) for the CPU to execute. In Program 2-1 the commands .ORG (origin) and .EQU are examples of directives. More of these pseudo-instructions are discussed in future chapters.

3. The comment field begins with a semicolon comment indicator “;”. Comments may be at the end of a line or on a line by themselves. The assembler ignores comments, but they are indispensable to programmers. Although comments are optional, it is recommended that they be used to describe the program in a way that makes it easier for someone else to read and understand.
4. Notice the label “HERE” in the label field in Program 2-1. In the JMP the AVR is told to stay in this loop indefinitely. If your system has a monitor program you do not need this line and should delete it from your program. In Section 2.7 we will see how to create a ready-to-run program.

Review Questions

1. What is the purpose of pseudo-instructions?
2. _____ are translated by the assembler into machine code, whereas _____ are not.
3. True or false. Assembly language is a high-level language.
4. Which of the following instructions produces opcode? List all that do.
(a) LDI R16, 0x25 (b) ADD R23, R19 (c) .ORG 0x500 (d) JMP HERE
5. Pseudo-instructions are also called _____.
6. True or false. Assembler directives are not used by the CPU itself. They are simply a guide to the assembler.
7. In Question 4, which one is an assembler directive?

SECTION 2.7: ASSEMBLING AN AVR PROGRAM

Now that the basic form of an Assembly language program has been given, the next question is: How it is created, assembled, and made ready to run? The steps to create an executable Assembly language program (Figure 2-9) are outlined as follows:

1. First we use a text editor to type in a program similar to Program 2-1. In the case of the AVR microcontrollers, we use the AVRStudio IDE, which has a text editor, assembler, simulator, and much more all in one software package. It is an excellent development software that supports all the AVR chips and is free. Many editors or word processors are also available that can be used to create or edit the program. A widely used editor is the Notepad in Windows, which comes with all Microsoft operating systems. Notice that the editor must be able

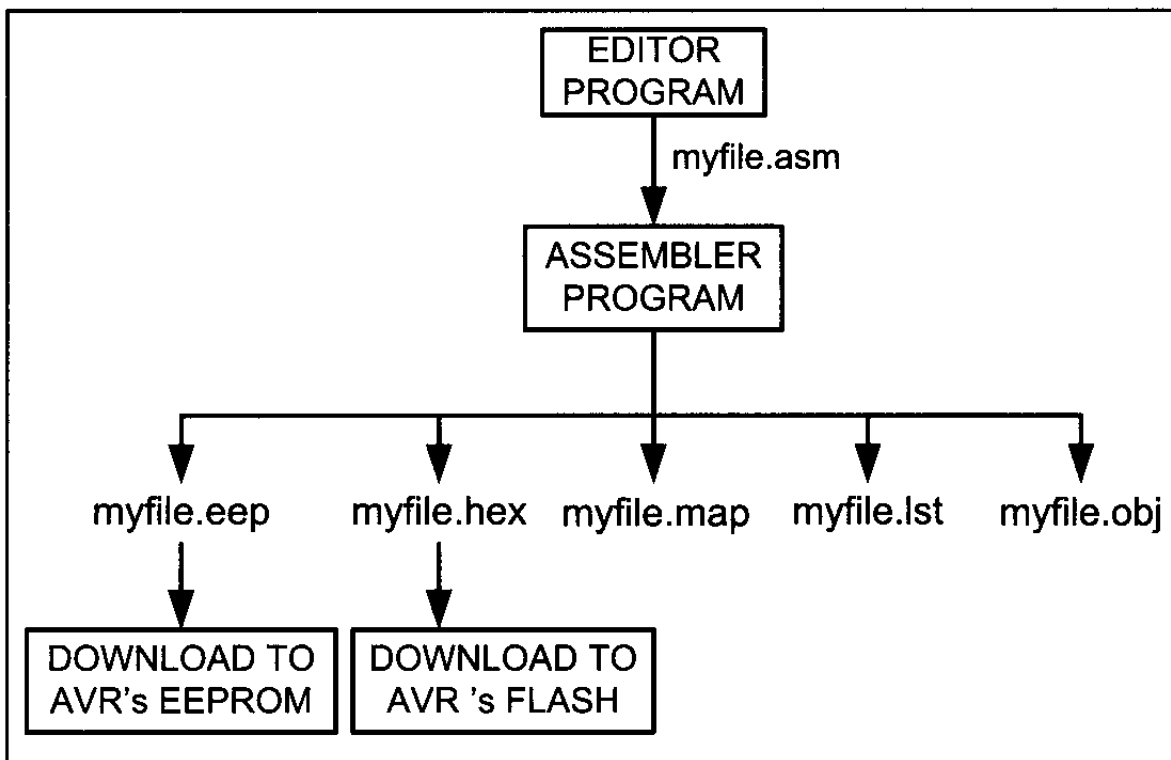


Figure 2-9. Steps to Create a Program

```

AVRASM: AVR macro assembler 2.1.2 (build 99 Nov  4 2005 09:35:05)
Copyright (C) 1995-2005 ATMEL Corporation

F:\AVR\Sample\Sample.asm(7): error: Invalid register
F:\AVR\Sample\Sample.asm(8): error: Operand(s) out of range in 'ldi r17,0x3432'
F:\AVR\Sample\Sample.asm(9): error: Undefined symbol: R38
F:\AVR\Sample\Sample.asm(9): error: Invalid register
F:\AVR\Sample\Sample.asm(16): No EEPROM data, deleting F:\AVR\Sample\Sample.eep

Assembly failed, 4 errors, 0 warnings

```

Figure 2-10. Sample of an AVR Error

to produce an ASCII file. For assemblers, the file names follow the usual DOS conventions, but the source file has the extension “asm”. The “asm” extension for the source file is used by an assembler in the next step.

2. The “asm” source file containing the program code created in step 1 is fed to the AVR assembler. The assembler produces an object file, a hex file, an eeprom file, a list file, and a map file. The object file has the extension “obj”, the hex file extension is “hex”, the list file extension is “lst”, the map file extension is “map”, and the eeprom file has the extension “eep”. After a successful link, the hex file is ready to be burned into the AVR’s program ROM and is downloaded into the AVR Trainer. We can write the eeprom file into the AVR’s EEPROM to initialize the EEPROM. See Chapter 8 for more details.

More about asm and object files

The asm file is also called the *source* file and must have the “asm” extension. As mentioned earlier, this file is created with a text editor such as Windows Notepad. Many assemblers come with a text editor. The assembler converts the asm file’s Assembly language instructions into machine language and provides the obj (object) file. The object file, as mentioned earlier, has an “obj” as its extension. The object file is used as input to a simulator or an emulator.

Before we can assemble a program to create a ready-to-run program, we must make sure that it is error free. The AVR Studio IDE provides us error messages and we examine them to see the nature of syntax errors. The assembler will not assemble the program until all the syntax errors are fixed. A sample of an error message is shown in Figure 2-10.

“lst” and “map” files

The map file shows the labels defined in the program together with their values. Examine Figure 2-11. It shows the Map file of Program 2-1.

The lst (list) file, which is optional, is very useful to the programmer. The

```

AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Sun Apr 06 23:39:32 2008

EQU  SUM          00000300
CSEG HERE        00000009

```

Figure 2-11. Map File of Program 2-1


```

;store SUM in SRAM location 0x300.
.DEVICE ATmega32
.EQU SUM = 0x300 ;SRAM loc $300 for SUM

.ORG 00 ;start at address 0
000000 e205 LDI R16, 0x25 ;R16 = 0x25
000001 e314 LDI R17, $34 ;R17 = 0x34
000002 e321 LDI R18, 0b00110001 ;R18 = 0x31
000003 0f01 ADD R16, R17 ;add R17 to R16
000004 0f02 ADD R16, R18 ;add R18 to R16
000005 e01b LDI R17, 11 ;R17 = 0x0B
000006 0f01 ADD R16, R17 ;add R17 to R16
000007 9300 0300 STS SUM, R16 ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE ;stay here forever

```

RESOURCE USE INFORMATION

 ...

Memory use summary [bytes] :

Segment	Begin	End	Code	Data	Used	Size	Use%
[.cseg]	0x000000	0x000016	22	0	22	unknown	-
[.dseg]	0x000060	0x000060	0	0	0	unknown	-
[.eseg]	0x000000	0x000000	0	0	0	unknown	-

Assembly complete, 0 errors, 0 warnings

Figure 2-12. List File of Program 2-1

list shows the binary and source code; it also shows which instructions are used in the source code, and the amount of memory the program uses. See Figure 2-12.

Many assemblers assume that the list file is not wanted unless you indicate that you want to produce it. These files can be accessed by a text editor such as Notepad and displayed on the monitor, or sent to the printer to get a hard copy. The programmer uses the list and map files to ensure correct system design.

There are many different AVR assemblers available for free nowadays. If you use the Windows operating system, AVR Studio can be a good choice. It has a nice environment and provides great help.

Review Questions

1. True or false. The AVR Studio IDE and Windows Notepad text editor both produce an ASCII file.
2. True or false. The extension for the source file is “asm”.
3. Which of the following files can be produced by a text editor?
 (a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst
4. Which of the following files is produced by an assembler?
 (a) myprog.asm (b) myprog.obj (c) myprog.hex (d) myprog.lst

SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN THE AVR

In this section we discuss the role of the program counter (PC) in executing a program and show how the code is fetched from ROM and executed. We will also discuss the program (code) ROM space for various AVR family members. Finally, we examine the Harvard architecture of the AVR.

Program counter in the AVR

The most important register in the AVR microcontroller is the PC (program counter). The program counter is used by the CPU to point to the address of the next instruction to be executed. As the CPU fetches the opcode from the program ROM, the program counter is incremented automatically to point to the next instruction. The wider the program counter, the more memory locations a CPU can access. That means that a 14-bit program counter can access a maximum of 16K ($2^{14} = 16K$) program memory locations.

In AVR microcontrollers each Flash memory location is 2 bytes wide. For example, in ATmega32, whose Flash is 32K bytes, the Flash is organized as $16K \times 16$, and its program counter is 14 bits wide ($2^{14} = 16K$ memory locations). The ATmega64 has a 15-bit program counter, so its Flash has 32K locations ($2^{15} = 32K$), with each location containing 2 bytes ($32K \times 2 \text{ bytes} = 64K \text{ bytes}$).

In the case of a 16-bit program counter, the code space is 64K ($2^{16} = 64K$), which occupies the 0000–\$FFFF address range. The program counter in the AVR family can be up to 22 bits wide. This means that the AVR family can access program addresses 000000 to \$3FFFFFF, a total of 4M locations. Because each Flash location is 2 bytes wide, the AVR can have a maximum of 8M bytes of code. However, at the time of this writing, none of the members of the AVR family have the entire 8M bytes of on-chip ROM installed. See Table 2-7.

Table 2-7: AVR On-chip ROM Size and Address Space

	On-chip Code ROM (Bytes)	Code Address Range (Hex)	ROM Organization
ATtiny25	2K	00000–003FF	$1K \times 2 \text{ bytes}$
ATmega8	8K	00000–00FFF	$4K \times 2 \text{ bytes}$
ATmega32	32K	00000–03FFF	$16K \times 2 \text{ bytes}$
ATmega64	64K	00000–07FFF	$32K \times 2 \text{ bytes}$
ATmega128	128K	00000–0FFFF	$64K \times 2 \text{ bytes}$
ATmega256	256K	00000–1FFFF	$128K \times 2 \text{ bytes}$

ROM memory map in the AVR family

As we just discussed, some family members have only a few kilobytes of on-chip ROM and some, such as the ATmega128, have 128K of ROM. The point to remember is that no member of the AVR family can access more than 4M words of opcode because the program counter in the AVR can be a maximum of 22 bits wide (000000 to \$3FFFFFF address range). It must be noted that while the first

Example 2-9

Find the ROM memory address of each of the following AVR chips:

- (a) ATtiny25 with 2 KB
- (b) ATmega16 with 16 KB
- (c) ATmega64 with 64 KB

Solution:

- (a) With 2K bytes of on-chip ROM memory, we have 2048 bytes ($2 \times 1024 = 2048$). As each address location in AVR is 2 bytes, its Flash has 1024 locations ($2048 / 2 = 1024$). This maps to address locations of 0000 to 003FF. Notice that 0 is always the first location.
- (b) With 16K bytes of on-chip ROM memory, we have 16,384 bytes ($16 \times 1024 = 16,384$), and 8192 locations ($16384 / 2 = 8192$), which gives 0000–01FFF.
- (c) With 64K we have 65,535 bytes ($64 \times 1024 = 65,535$), and 32,768 locations. Converting 32,768 to hex, we get 8000; therefore, the memory space is 0000 to 07FFF.

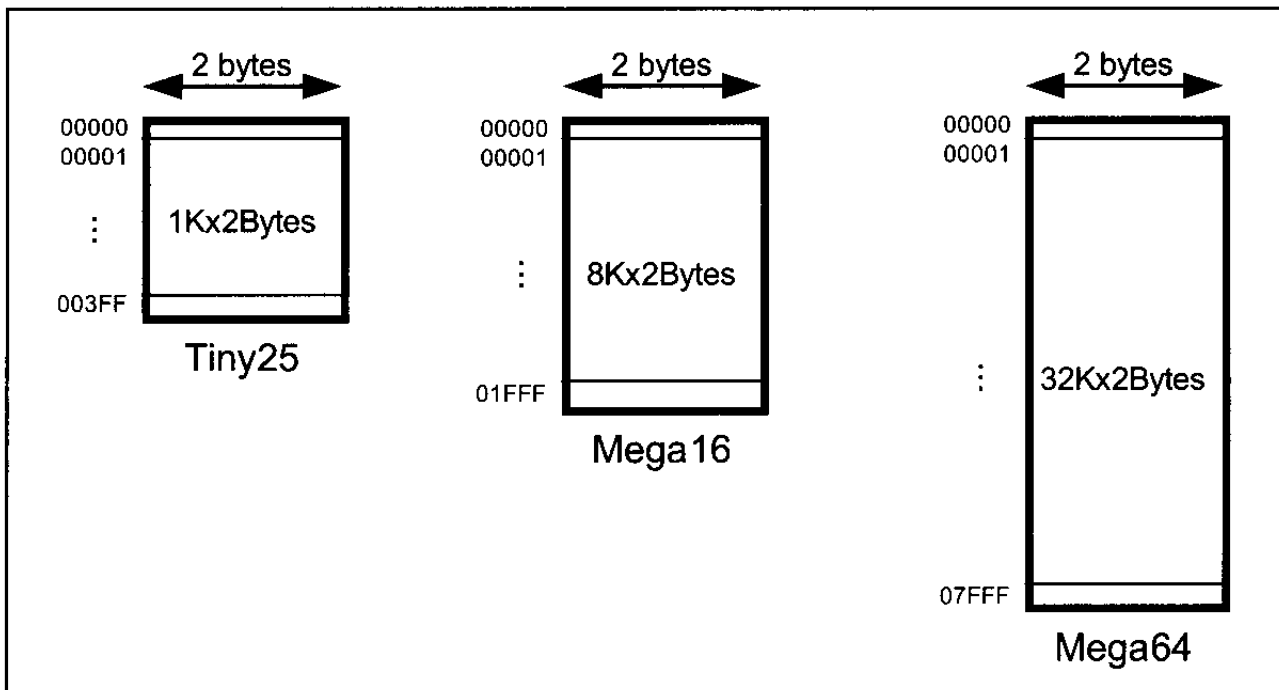


Figure 2-13. AVR On-Chip Program (code) ROM Address Range

location of program ROM inside the AVR has the address of 000000, the last location can be different depending on the size of the ROM on the chip. (See Figure 2-13.) Among the AVR family members, the ATmega8 has 8K of on-chip ROM. This 8K ROM memory is organized as 4K \times 2 bytes and has memory addresses of 00000 to 000FFF. Therefore, the first location of on-chip ROM of this AVR has an address of 00000 and the last location has the address of 000FFF. Look at Example 2-9 to see how this is computed.

Where the AVR wakes up when it is powered up

One question that we must ask about any microcontroller (or microprocessor) is: At what address does the CPU wake up when power is applied? Each microprocessor is different. In the case of the AVR microcontrollers (that is, all members regardless of the family and variation), the microcontroller wakes up at memory address 0000 when it is powered up. By powering up we mean applying V_{CC} to the RESET pin as discussed in Chapter 8. In other words, when the AVR is powered up, the PC (program counter) has the value of 00000 in it. This means that it expects the first opcode to be stored at ROM address \$00000. For this reason, in the AVR system, the first opcode must be burned into memory location \$00000 of program ROM because this is where it looks for the first instruction when it is booted. We achieve this by using the .ORG statement in the source program as shown earlier. Next we discuss the step-by-step action of the program counter in fetching and executing a sample program.

Placing code in program ROM

To get a better understanding of the role of the program counter in fetching and executing a program, we examine the action of the program counter as each instruction is fetched and executed. First, we examine once more the list file of the sample program and show how the code is placed into the Flash ROM of the AVR chip. As we can see, the opcode and operand for each instruction are listed on the left side of the list file.

After the program is burned into ROM of an AVR family member such as ATmega32 or ATtiny11, the opcode and operand are placed in ROM memory locations starting at 0000 as shown in the Program 2-1 list file.

The list shows that address 0000 contains E205, which is the opcode for moving a value into R16, and the operand (in this case 0x25) to be moved to R16. Therefore, the instruction “LDI R16, 0x25” has a machine code of “E205”, where E is the opcode and 205 is the operand. See Figures 2-14 and 2-15. Similarly, the machine code “E314” is located in ROM memory location 0001 and represents the opcode and the operands for the instruction “LDI R17, \$34”. In

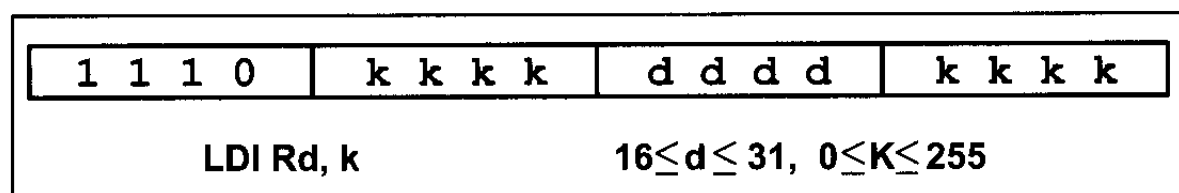


Figure 2-14. The Machine Code for Instruction “LDI Rd, k” in Binary

<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">E k₁ d k₀</div> LDI Rd, k ₁ k ₀	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">E 2 0 5</div> LDI R16, 0x25
<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">E 3 1 4</div> LDI R17, 0x34	<div style="border: 1px solid black; display: inline-block; padding: 2px 10px;">E 3 2 1</div> LDI R18, 0x31

Figure 2-15. The Machine Code for Instruction “LDI Rd, k” in Hex

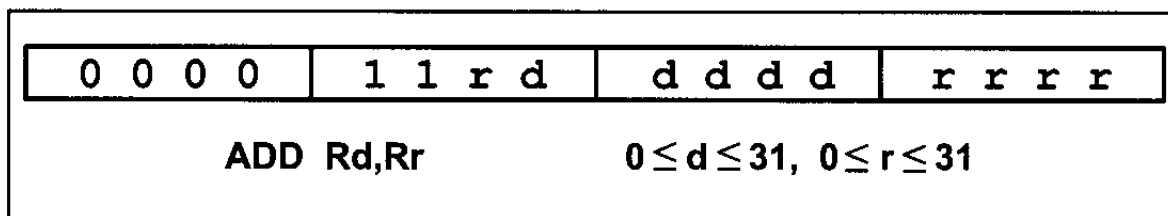


Figure 2-16. The Machine Code for Instruction “ADD Rd,Rr” in Binary



Figure 2-17. The Machine Code for Instruction “ADD Rd,Rr” in Hex

the same way, machine code “E321” is located in memory location 0002 and represents the opcode and the operand for the instruction “LDI R18, 0b00110001”. The memory location 0003 has the machine code of 0F01, which is the opcode and the operands for the instruction “ADD R16,R17”. Similarly, the machine code “0F02” is located in memory location 0004 and represents the opcode and the operands for the instruction “ADD R16, R18”. See Figures 2-16 and 2-17. The memory location 0005 has the opcode and operand for the “LDI R17, 11” instruction. The memory location 0006 has the opcode and operand for the “ADD R16,R17” instruction. The opcode for instruction “STS SUM, R16” is located at address 00007 and its address of 0x300 at address 00008. The opcode for “JMP HERE” and its target address are located in locations 00009 and 0000A. While all the instructions in this program are 2-byte instructions, the JMP and STS instructions are 4-byte instructions. The reasons are explained at the end of this section.

Executing a program instruction by instruction

Assuming that the above program is burned into the ROM of an AVR chip, the following is a step-by-step description of the action of the AVR upon applying power to it:

1. When the AVR is powered up, the PC (program counter) has 00000 and starts to fetch the first instruction from location 00000 of the program ROM. In the case of the above program the first code is E205, which is the code for moving operand 0x25 to R16. Upon executing the code, the CPU places the value of 25 in R16. Now one instruction is finished. Then the program counter is incremented to point to 00001 (PC = 00001), which contains code E314, the machine code for the instruction “LDI R17, 0x34”.
2. Upon executing the machine code E314, the value 0x34 is loaded to R17. Then the program counter is incremented to 0002.
3. ROM location 0002 has the machine code for instruction “LDI R18, 0x31”. This instruction is executed and now PC = 0003.
4. This process goes on until all the instructions up to “ADD R16,R17” are fetched and executed. Notice that all the above instructions are 2-byte instructions; that is, each one takes two bytes of ROM (one word).
5. Now PC = 0007 points to the next instruction, which is “STS SUM, R16”. This is a 2-word (4-byte) instruction. It takes addresses of 07 and 08. When the

instruction is executed, the content of R16 is stored into memory location 0x300. After the execution of this instruction, PC = 0009.

6. Now PC = 0009 points to the next instruction, which is "JMP HERE". This is a 2-word (4-byte) instruction. It takes addresses of 09 and 0A. After the execution of this instruction, PC = 0009. This keeps the program in an infinite loop. The fact that the program counter points at the next instruction to be executed explains why some microprocessors (notably the x86) call the program counter the *instruction pointer*.

ROM width in the AVR

As we have seen so far in this section, each location of the address space holds two bytes (a word). If we have 16 address lines, this will give us 2^{16} locations, which is 64K of memory location with an address map of 0000–FFFFH. To bring in more information (code or data) into the CPU, AVR increased the width of the data bus to 16 bits. In other words, the AVR is word-addressable. In contrast, the 8051 CPU is byte addressable. In a sense, the data bus is like traffic lanes on the highway where each lane is 8 bits wide. The more lanes, the more information we can bring into the CPU for processing. For the AVR, the internal data bus between the code ROM and the CPU is 16 bits wide, as shown in Figure 2-18. Therefore, the 64K ROM space is shown as $32K \times 16$ using a 16-bit word data size.

The same rule applies to the entire program address space of AVR, which is 8M, organized as $4M \times 16$. The widening of the data path between the program ROM and the CPU is another way in which the AVR designers increased the processing power of the AVR family. Another reason to make the code ROM 16 bits wide is to match it with the instruction width of the AVR because the vast majority of the instructions are 2-byte instructions. This way, the CPU brings in an instruction from ROM every time it makes a trip to the program ROM. That will make instruction fetch a single cycle, as we will see in the next chapter when instruction timing is discussed.

The AVR designers have made all instructions either 2-byte or 4-byte; there are no 1-byte or 3-byte instructions, as is the case with the x86 and 8051 chips. This is part of the RISC architectural philosophy, which we will study in the next section. It must also be noted that the data memory SRAM in the AVR microcontroller is still 8-bit, and it is byte-addressable.

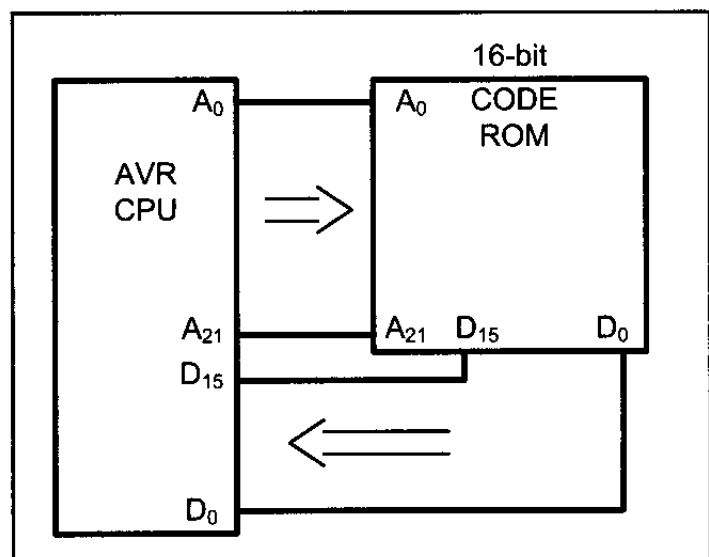


Figure 2-18. Program ROM Width for the AVR

Harvard architecture in the AVR

As we mentioned in Chapter 0, AVR uses Harvard architecture, which means that there are separate buses for the code and the data memory. See Figure 2-19. The Program Bus provides access to the Program Flash ROM whereas the Data Bus is used for bringing data to the CPU.

As we can see in Figure 2-19, in the Program Bus, the data bus is 16 bits wide and the address bus is as wide as the PC register to enable the CPU to address the entire Program Flash ROM.

In the Data Bus, the data bus is 8 bits wide. As a result, the CPU can access one byte of data at a time. The address bus is 16 bits wide. Thus the data memory space can be up to 64K bytes.

In Sections 2-2 and 2-3, you learned about data memory space and how to use the STS and LDS instructions. When the CPU wants to execute the “LDS Rn,k” instruction, it puts k on the address bus of the Data Bus, and receives data through the data bus. For example, to execute “LDS R20, 0x90”, the CPU puts 0x90 on the address bus. The location \$90 is in the SRAM (see Figure 2-4). Thus, the SRAM puts the contents of location \$90 on the data bus. The CPU gets the contents of location \$90 through the data bus and puts it in R20.

The “STS k,Rn” instruction is executed similarly. The CPU puts k on the address bus and the contents of Rn on the data bus. The unit whose address is on the address bus receives the contents of data bus. For example, to execute the “STS \$100,R30” instruction the CPU puts the contents of R30 on the data bus and \$100 on the address bus. Because \$100 is bigger than \$60, the address belongs to SRAM; thus SRAM gets the contents of the data bus and puts it in location \$100 of the SRAM.

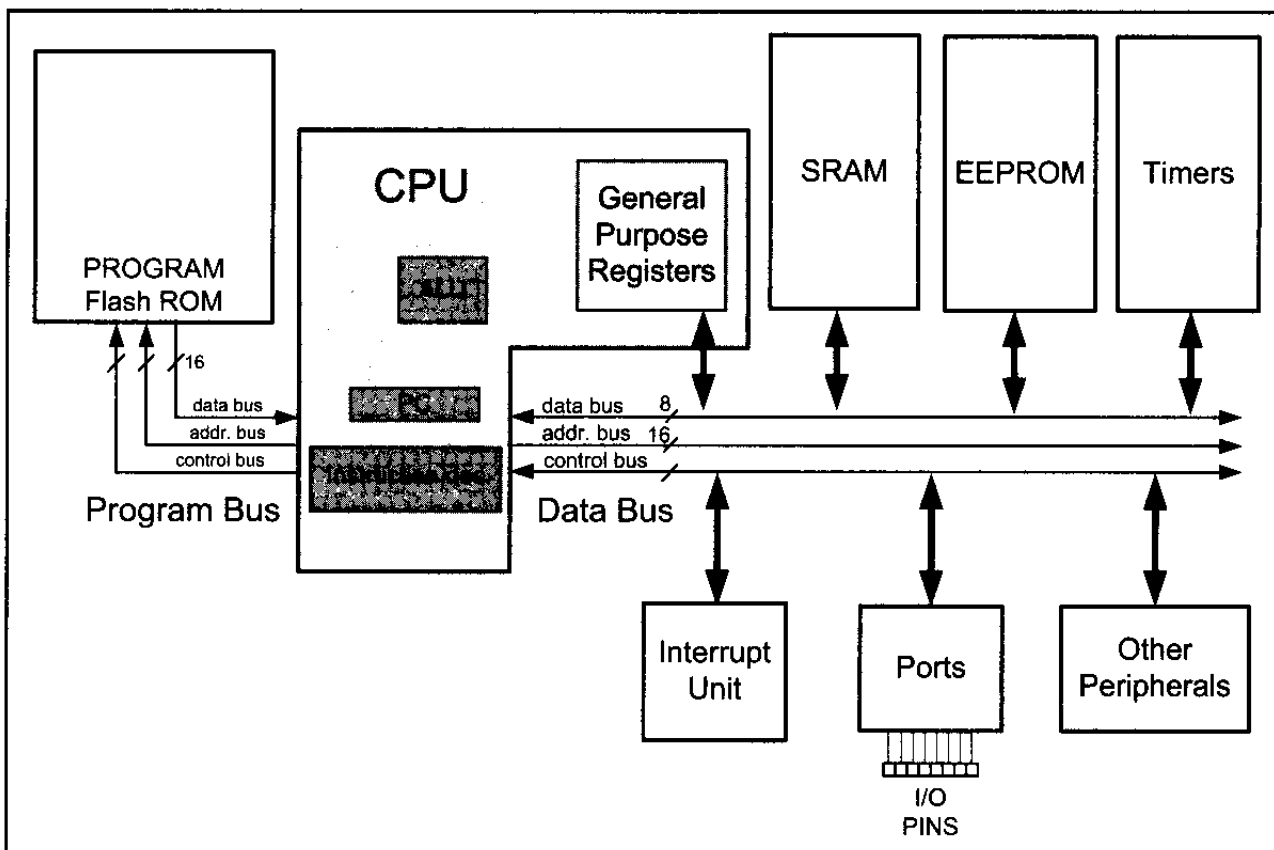


Figure 2-19. Harvard Architecture in the AVR

Little endian vs. big endian war

Examine the placing of the code in the AVR ROM, shown in Figure 2-20. The low byte goes to the low memory location, and the high byte goes to the high memory address. This convention is called little endian to contrast it with big endian. The origin of the terms *big endian* and *little endian* is from an argument in a Gulliver's Travels story over how an egg should be opened: from the big end or the little end. In the big endian method, the high byte goes to the low address, whereas in the little endian method, the high byte goes to the high address and the low byte to the low address. All Intel microprocessors and many microcontrollers use the little endian convention. Freescale (formerly Motorola) microprocessors, along with some mainframes, use big endian. The difference might seem as trivial as whether to break an egg from the big end or the little end, but it is a nuisance in converting software from one camp to be run on a computer of the other camp. Some microprocessors, such as the PowerPC from IBM/Freescale, let the software designer choose little endian or big endian convention.

Address	High byte	Low byte
00000	E2	05
00001	E3	14
00002	E3	21
00003	0F	01
00004	0F	02
00005	E0	1B
00006	0F	01
00007	93	00
00008	03	00
00009	94	0C
0000A	00	09

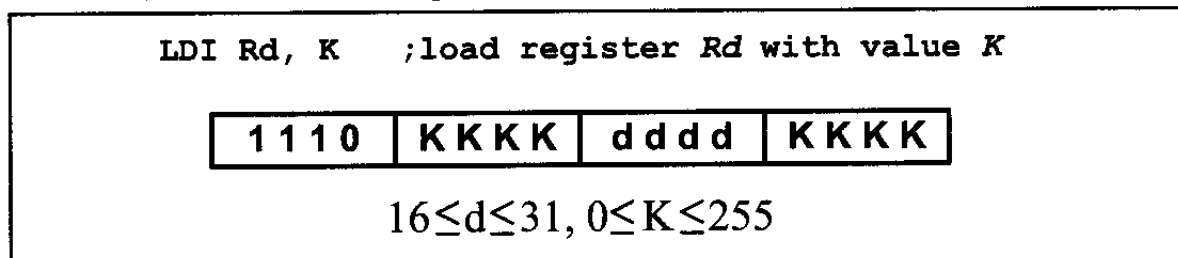
Figure 2-20. AVR Program ROM Contents for Program 2-1 List File

Instruction size of the AVR

Recall that the AVR instructions are either 2-byte or 4-byte. Almost all the instructions in the AVR are 2-byte instructions. The exceptions are STS, JMP, and a few others. Next we explore the instruction size and formation for a few of the instructions we have used in this chapter. This should give you some insight into the instructions of the AVR.

LDI instruction formation

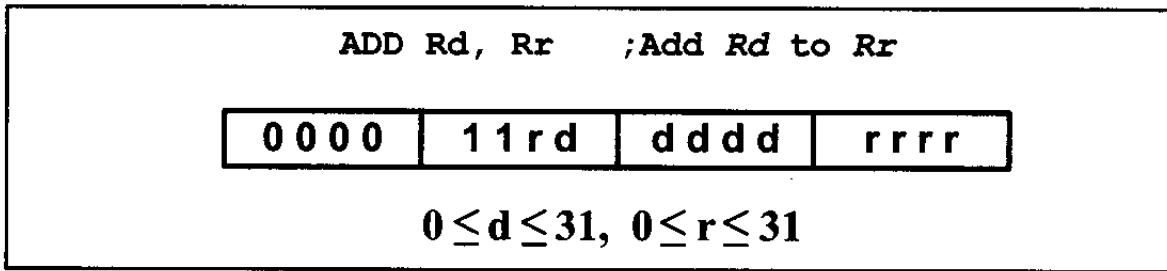
The LDI is a 2-byte (16-bit) instruction. Of the 16 bits, the first 4 bits are set aside for the opcode, the second and the fourth 4 bits are used for the value of 00 to \$FF, and the third 4 bits present the destination register. This is shown below.



ADD instruction formation

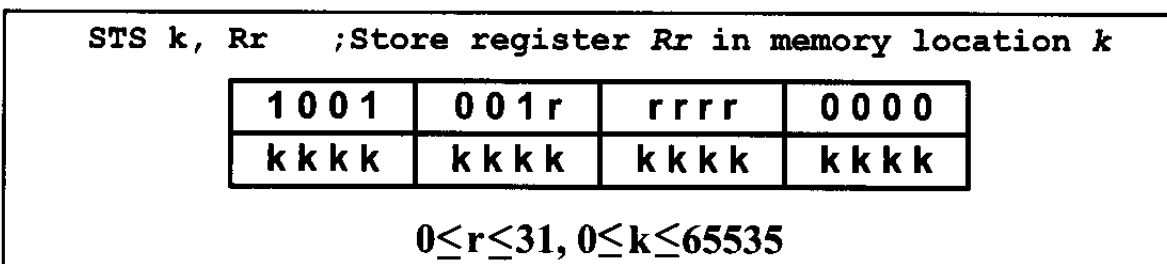
The ADD is a 2-byte (16-bit) instruction. Of the 16 bits, the first 6 bits are

set aside for the opcode, and the other 10 bits represent the source and the destination registers. This is shown below.



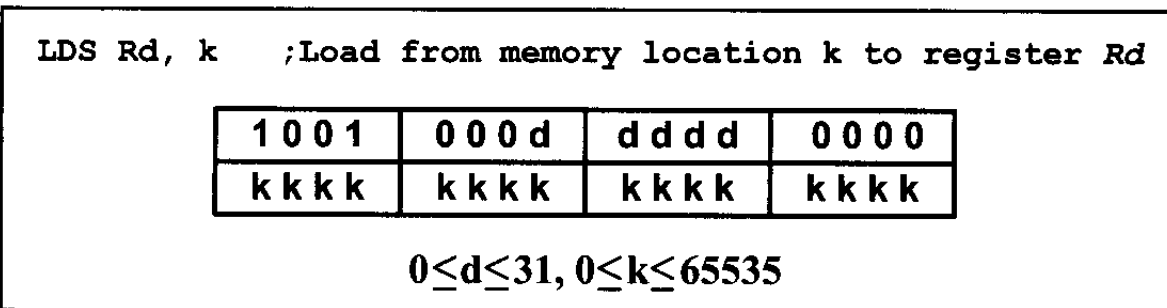
STS instruction formation

The STS is a 4-byte (32-bit) instruction. Of the 32 bits, the first 16 bits are set aside for the opcode and the address of the source, and the other 16 bits are used for the address of the destination. This is shown below.



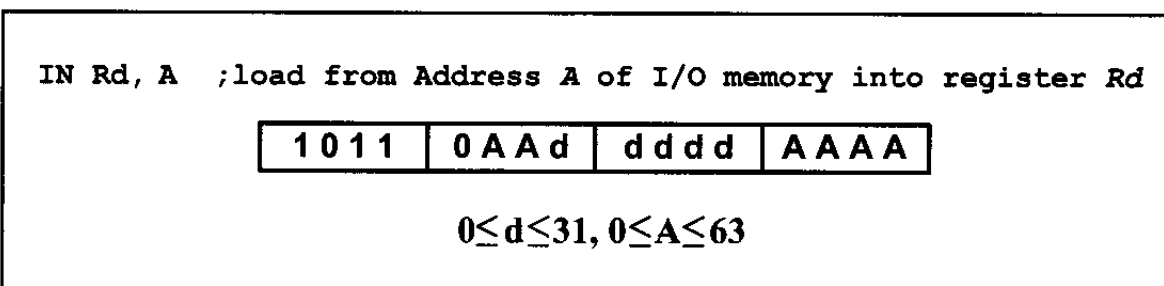
LDS instruction formation

The LDS is a 4-byte (32-bit) instruction. Of the 32 bits, the first 16 bits are set aside for the opcode and the destination register, and the other 16 bits are used for the address of the source memory location. This is shown below.



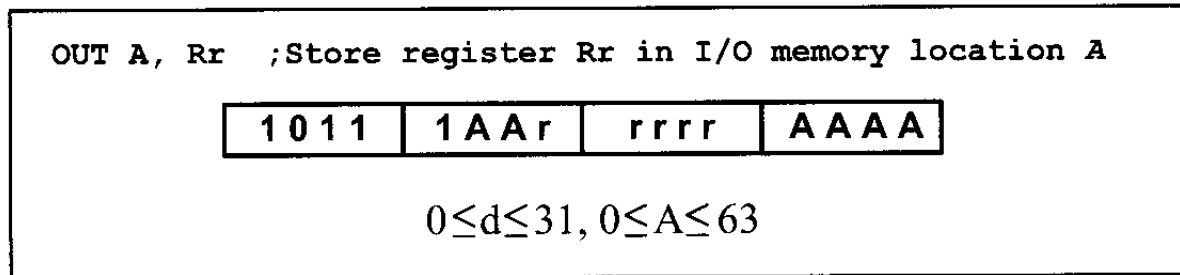
IN instruction formation

The IN is a 2-byte (16-bit) instruction. Of the 16 bits, the first 5 bits are set aside for the opcode, and the other 11 bits are used for the address of the source memory location, and destination register. This is shown below.



OUT instruction formation

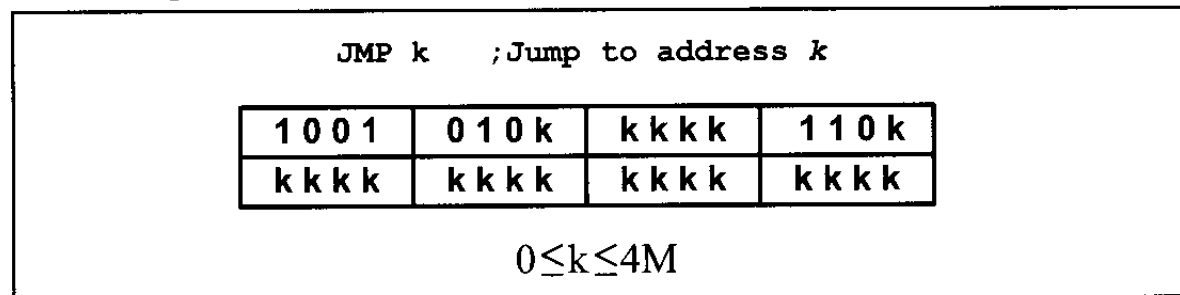
The OUT is a 2-byte (16-bit) instruction. Of the 16 bits, the first 5 bits are set aside for the opcode, and the other 11 bits are used for the address of the source memory location and destination register. This is shown below.



JMP instruction formation

The JMP is a 4-byte (32-bit) instruction. Of the 32 bits, only 10 bits are set aside for the opcode, and the rest (22 bits) are used for the target address of the JMP. This is shown below.

The 22-bit address gives us 4M of address space; so, it can address all of the ROM space.



Review Questions

1. In the AVR, the program counter is, at most, _____ bits wide.
2. True or false. Every member of the AVR family, regardless of the program ROM size, wakes up at memory \$0000 when it is powered up.
3. At what ROM location do we store the first opcode of an AVR program?
4. The instruction “LDI R20, 0x44” is a _____-byte instruction.
5. The instruction “JMP label” is a _____-byte instruction.
6. True or false. All the instructions in the AVR are 2- or 4-byte instructions.

SECTION 2.9: RISC ARCHITECTURE IN THE AVR

There are three ways available to microprocessor designers to increase the processing power of the CPU:

1. Increase the clock frequency of the chip. One drawback of this method is that the higher the frequency, the more power and heat dissipation. Power and heat dissipation is especially a problem for hand-held devices.
2. Use Harvard architecture by increasing the number of buses to bring more information (code and data) into the CPU to be processed. While in the case of

x86 and other general purpose microprocessors this architecture is very expensive and unrealistic, in today's microcontrollers this is not a problem. As we saw in the last section, the AVR has Harvard architecture.

3. Change the internal architecture of the CPU and use what is called RISC architecture.

Atmel used all three methods to increase the processing power of the AVR microcontrollers. In this section we discuss the merits of RISC architecture and examine how the AVR microcontrollers have adapted it.

RISC architecture

In the early 1980s, a controversy broke out in the computer design community, but unlike most controversies, it did not go away. Since the 1960s, in all mainframes and minicomputers, designers put as many instructions as they could think of into the CPU. Some of these instructions performed complex tasks. An example is adding data memory locations and storing the sum into memory. Naturally, microprocessor designers followed the lead of minicomputer and mainframe designers. Because these microprocessors used such a large number of instructions, many of which performed highly complex activities, they came to be known as CISC (complex instruction set computer) processors. According to several studies in the 1970s, many of these complex instructions etched into CPUs were never used by programmers and compilers. The huge cost of implementing a large number of instructions (some of them complex) into the microprocessor, plus the fact that a good portion of the transistors on the chip are used by the instruction decoder, made some designers think of simplifying and reducing the number of instructions. As this concept developed, the resulting processors came to be known as RISC (reduced instruction set computer).

Features of RISC

The following are some of the features of RISC as implemented by the AVR microcontroller.

Feature 1

RISC processors have a fixed instruction size. In a CISC microcontroller such as the 8051, instructions can be 1, 2, or even 3 bytes. For example, look at the following instructions in the 8051:

CLR	C	;clear Carry flag, a 1-byte instruction
ADD	Accumulator, #mybyte	;a 2-byte instruction
LJMP	target_address	;a 3-byte instruction

This variable instruction size makes the task of the instruction decoder very difficult because the size of the incoming instruction is never known. In a RISC architecture, the size of all instructions is fixed. Therefore, the CPU can decode the instructions quickly. This is like a bricklayer working with bricks of the same size as opposed to using bricks of variable sizes. Of course, it is much more efficient to use bricks of the same size. In the last section we saw how the AVR uses 2-byte instructions with very few 4-byte instructions.

Feature 2

One of the major characteristics of RISC architecture is a large number of registers. All RISC architectures have at least 32 registers. Of these 32 registers, only a few are assigned to a dedicated function. One advantage of a large number of registers is that it avoids the need for a large stack to store parameters. Although a stack can be implemented on a RISC processor, it is not as essential as in CISC because so many registers are available. In the AVR microcontrollers the use of 32 general purpose registers satisfies this RISC feature. The stack for the AVR is covered in the next chapter.

Feature 3

RISC processors have a small instruction set. RISC processors have only basic instructions such as ADD, SUB, MUL, LOAD, STORE, AND, OR, EOR, CALL, JUMP, and so on. The limited number of instructions is one of the criticisms leveled at the RISC processor because it makes the job of Assembly language programmers much more tedious and difficult compared to CISC Assembly language programming. This is one reason that RISC is used more commonly in high-level language environments such as the C programming language rather than Assembly language environments. It is interesting to note that some defenders of CISC have called it “complete instruction set computer” instead of “complex instruction set computer” because it has a complete set of every kind of instruction. How many of these instructions are used and how often is another matter. The limited number of instructions in RISC leads to programs that are large. Although these programs can use more memory, this is not a problem because memory is cheap. Before the advent of semiconductor memory in the 1960s, however, CISC designers had to pack as much action as possible into a single instruction to get the maximum bang for their buck. In the ATmega we have around 130 instructions. We will examine more of the instruction set for the AVR in future chapters.

Feature 4

At this point, one might ask, with all the difficulties associated with RISC programming, what is the gain? The most important characteristic of the RISC processor is that more than 95% of instructions are executed with only one clock cycle, in contrast to CISC instructions. Even some of the 5% of the RISC instructions that are executed with two clock cycles can be executed with one clock cycle by juggling instructions around (code scheduling). Code scheduling is most often the job of the compiler. We will examine the instruction cycle time and pipelining of the AVR in Chapter 3.

Feature 5

RISC processors have separate buses for data and code. In all the x86 processors, like all other CISC computers, there is one set of buses for the address (e.g., A0–A24 in the 80286) and another set of buses for data (e.g., D0–D15 in the 80286) carrying opcodes and operands in and out of the CPU. To access any section of memory, regardless of whether it contains code or data operands, the same

address bus and data bus are used. In RISC processors, there are four sets of buses: (1) a set of data buses for carrying data (operands) in and out of the CPU, (2) a set of address buses for accessing the data, (3) a set of buses to carry the opcodes, and (4) a set of address buses to access the opcodes. The use of separate buses for code and data operands is commonly referred to as Harvard architecture. We examined the Harvard architecture of the AVR in the previous section.

Feature 6

Because CISC has such a large number of instructions, each with so many different addressing modes, microinstructions (microcode) are used to implement them. The implementation of microinstructions inside the CPU employs more than 40–60% of transistors in many CISC processors. RISC instructions, however, due to the small set of instructions, are implemented using the hardwire method. Hardwiring of RISC instructions takes no more than 10% of the transistors.

Feature 7

RISC uses load/store architecture. In CISC microprocessors, data can be manipulated while it is still in memory. For example, in instructions such as “ADD Reg, Memory”, the microprocessor must bring the contents of the external memory location into the CPU, add it to the contents of the register, then move the result back to the external memory location. The problem is there might be a delay in accessing the data from external memory. Then the whole process would be stalled, preventing other instructions from proceeding in the pipeline. In RISC, designers did away with these kinds of instructions. In RISC, instructions can only load from external memory into registers or store registers into external memory locations. There is no direct way of doing arithmetic and logic operations between a register and the contents of external memory locations. All these instructions must be performed by first bringing both operands into the registers inside the CPU, then performing the arithmetic or logic operation, and then sending the result back to memory. This idea was first implemented by the Cray 1 supercomputer in 1976 and is commonly referred to as load/store architecture. In the last section, we saw that the arithmetic and logic operations are between the data memory (internal) locations, but none involves a ROM location. For example, there is no “ADD ROM-Loc” instruction in AVR.

In concluding this discussion of RISC processors, it is interesting to note that RISC technology was explored by the scientists at IBM in the mid-1970s, but it was David Patterson of the University of California at Berkeley who in 1980 brought the merits of RISC concepts to the attention of computer scientists. It must also be noted that in recent years CISC processors such as the Pentium have used some RISC features in their design. This was the only way they could enhance the processing power of the x86 processors and stay competitive. Of course, they had to use lots of transistors to do the job, because they had to deal with all the CISC instructions of the x86 processors and the legacy software of DOS/Windows.

Review Questions

1. What do RISC and CISC stand for?
2. True or false. The CISC architecture executes the vast majority of its instructions in 2, 3, or more clock cycles, while RISC executes them in one clock.
3. RISC processors normally have a _____ (large, small) number of general-purpose registers.
4. True or false. Instructions such as “ADD R16, ROMmemory” do not exist in RISC microcontrollers such as the AVR.
5. How many instructions does the ATmega have?
6. True or false. While CISC instructions are of variable sizes, RISC instructions are all the same size.
7. Which of the following operations do not exist for the ADD instruction in RISC?
(a) register to register (b) immediate to register (c) memory to memory
8. True or false. Harvard architecture uses the same address and data buses to fetch both code and data.

SECTION 2.10: VIEWING REGISTERS AND MEMORY WITH AVR STUDIO IDE

The AVR microcontroller has great tools and support systems, many of them free or inexpensive. AVR Studio is an assembler and simulator provided for free by Atmel Corporation and can be downloaded from the www.atmel.com website. See <http://www.MicroDigitalEd.com> for tutorials on how to use the AVR Studio assembler and simulator.

Many assemblers and C compilers come with a simulator. Simulators allow us to view the contents of registers and memory after executing each instruction (single-stepping). It is strongly recommended to use a simulator to single-step some of the programs in this chapter and future chapters. Single-stepping a program with a simulator gives us a deeper understanding of microcontroller architecture, in addition to the fact that we can use it to find the errors in our programs.

Figures 2-21 through 2-23 show screenshots for AVR simulators from AVR Studio.

See the following website for a tutorial on using AVR Studio:

<http://www.MicroDigitalEd.com>

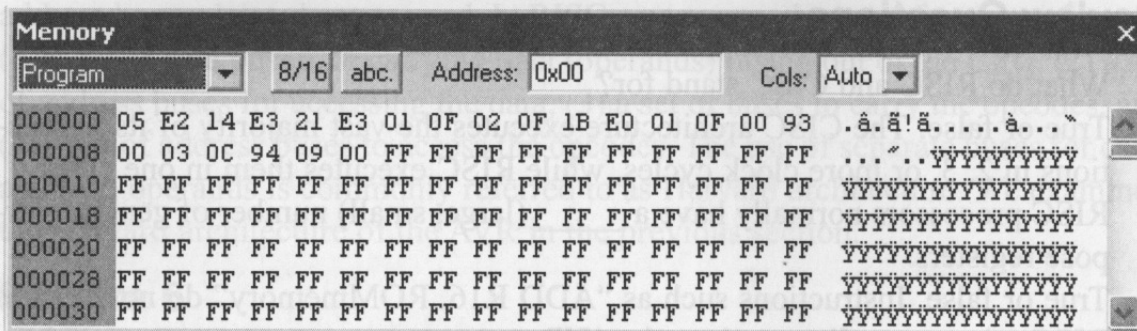


Figure 2-21. Data Memory Window in AVR Studio IDE

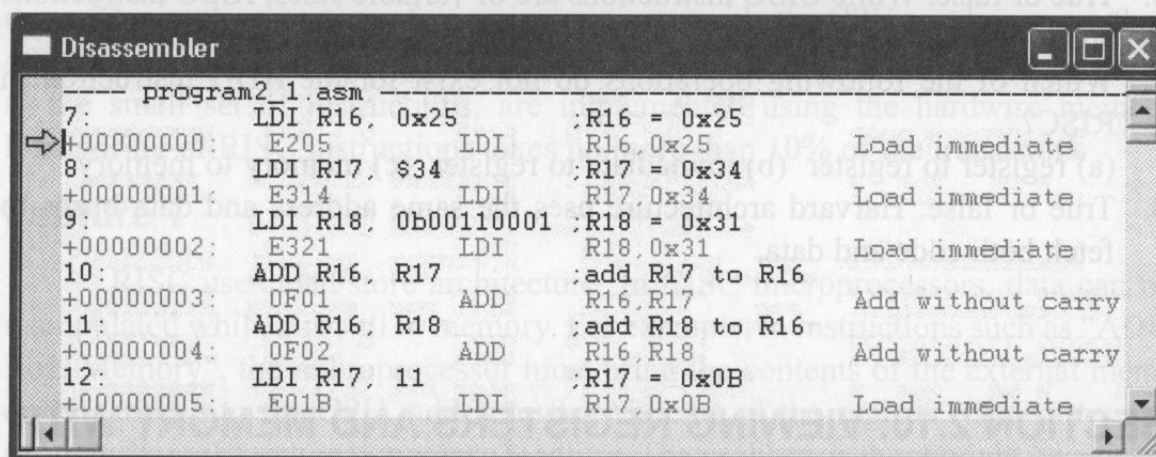


Figure 2-22. Program ROM (Disassembler) Window in AVR Studio IDE

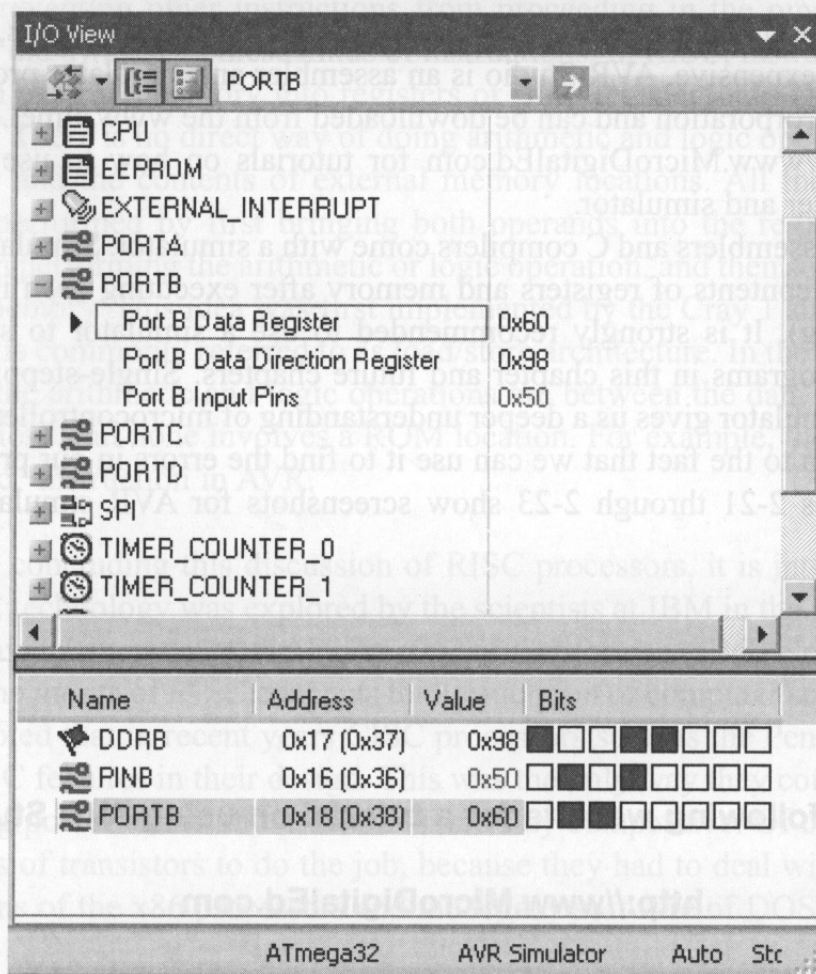


Figure 2-23. I/O View Window in AVR Studio IDE

SUMMARY

This chapter began with an exploration of the major registers of the AVR, including general purpose registers, I/O registers, and internal SRAM, and the program counter. The use of these registers was demonstrated in the context of programming examples. The process of creating an Assembly language program was described from writing the source file, to assembling it, linking, and executing the program. The PC (program counter) register always points to the next instruction to be executed. The way the AVR uses program Flash ROM space was explored because AVR Assembly language programmers must be aware of where programs are placed in ROM, and how much memory is available.

An Assembly language program is composed of a series of statements that are either instructions or pseudo-instructions, also called *directives*. Instructions are translated by the assembler into machine code. Pseudo-instructions are not translated into machine code: They direct the assembler in how to translate instructions into machine code. Some pseudo-instructions, called *data directives*, are used to define data. Data is allocated in byte-size increments. The data can be in binary, hex, decimal, or ASCII formats.

Flags are useful to programmers because they indicate certain conditions, such as carry or zero, that result from execution of instructions. The concepts of the RISC and Harvard architectures were also explored.

The RISC architecture allows the design of much more powerful microcontrollers. It has a simple instruction set and uses a large number of registers. Harvard architecture allows us to bring more code and data to the CPU faster. The use of a wider data bus in the AVR allows us to fetch an instruction every cycle because the AVR instructions are typically 2 bytes.

PROBLEMS

SECTION 2.1: THE GENERAL PURPOSE REGISTERS IN THE AVR

1. AVR is a(n) _____-bit microcontroller.
2. The general purpose registers are _____ bits wide.
3. The value in LDI is _____ bits wide.
4. The largest number that can be loaded into the GPRs is _____ in hex.
5. What is the result of the following code and where is it kept?

```
LDI    R20, $15
LDI    R21, $13
ADD    R20, R21
```
6. Which of the following is (are) illegal?
(a) LDI R20, 500 (b) LDI R23, 50 (c) LDI R1, 00
(d) LDI R16, \$255 (e) LDI R42, \$25 (f) LDI R23, 0xF5
(g) LDI 123, 0x50
7. Which of the following is (are) illegal?
(a) ADD R20, R11 (b) ADD R16, R1 (c) ADD R52, R16

8. What is the result of the following code and where is it kept?
- ```
LDI R19, $25
ADD R19, $1F
```
9. What is the result of the following code and where is it kept?
- ```
LDI    R21, 0x15
ADD    R21, 0xEA
```
10. True or false. We have 32 general purpose registers in the AVR.

SECTION 2.2: THE AVR DATA MEMORY

11. AVR data memory consists of _____ (Flash ROM, internal SRAM).
12. True or false. The special function register in AVR is called the I/O register.
13. True or false. The I/O registers are part of the data memory space.
14. True or false. The general-purpose registers are not part of the data memory space.
15. True or false. The data memory is the same size in all members of AVR.
16. If we add the I/O registers, internal RAM, and general purpose register sizes together we should get the total space for the _____.
17. Find the data memory size for the following AVR chips:
(a) ATmega32 (b) ATmega16 (c) ATtiny44
18. What is the difference between the EEPROM and data RAM space in the AVR?
19. Can we have an AVR chip with no EEPROM?
20. Can we have an AVR chip with no data memory?
21. What is the address range for the internal RAM?
22. What is the maximum number of bytes that the AVR can have for the data memory?

SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY

23. Show a simple code to load values \$30 and \$97 into locations \$105 and \$106, respectively.
24. Show a simple code to load the value \$55 into locations \$300–\$308.
25. Show a simple code to load the value \$5F into the PORTB I/O register.
26. True or false. We cannot load immediate values into the internal RAM directly.
27. Show a simple code to (a) load the value \$11 into locations \$100–\$105, and (b) add the values together and place the result in R20 as they are added.
28. Repeat Problem 27, except place the result in location \$105 after the addition is done.
29. Show a simple code to (a) load the value \$15 into location \$67, and (b) add it to R19 five times and place the result in R19 as the values are added. R19 should be zero before the addition starts.
30. Repeat Problem 29, except place the result in location \$67.
31. Write a simple code to complement the contents of location \$68 and place the result in R27.
32. Write a simple code to copy data from location \$68 to PORTC using R19.

SECTION 2.4: AVR STATUS REGISTER

33. The status register is a(n) _____ -bit register.
34. Which bits of the status register are used for the C and H flag bits, respectively?
35. Which bits of the status register are used for the V and N flag bits, respectively?
36. In the ADD instruction, when is C raised?
37. In the ADD instruction, when is H raised?
38. What is the status of the C and Z flags after the following code?
- ```
LDI R20, 0xFF
LDI R21, 1
ADD R20, R21
```
39. Find the C flag value after each of the following codes:
- |                   |                |                   |
|-------------------|----------------|-------------------|
| (a) LDI R20, 0x54 | (b) LDI R23, 0 | (c) LDI R30, 0xFF |
| LDI R25, 0xC4     | LDI R16, 0xFF  | LDI R18, 0x05     |
| ADD R20, R25      | ADD R23, R16   | ADD R30, R18      |
40. Write a simple program in which the value 0x55 is added 5 times.

## SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES

41. State the value (in hex) used for each of the following data:
- ```
.EQU MYDAT_1 = 55
.EQU MYDAT_2 = 98
.EQU MYDAT_3 = 'G'
.EQU MYDAT_4 = 0x50
.EQU MYDAT_5 = 200
.EQU MYDAT_6 = 'A'
.EQU MYDAT_7 = 0xAA
.EQU MYDAT_8 = 255
.EQU MYDAT_9 = 0B10010000
.EQU MYDAT_10 = 0b01111110
.EQU MYDAT_11 = 10
.EQU MYDAT_12 = 15
```
42. State the value (in hex) for each of the following data:
- ```
.EQU DAT_1 = 22
.EQU DAT_2 = $56
.EQU DAT_3 = 0b10011001
.EQU DAT_4 = 32
.EQU DAT_5 = 0xF6
.EQU DAT_6 = 0B11111011
```
43. Show a simple code to (a) load the value \$11 into locations \$60–\$65, and (b) add them together and place the result in R29 as the values are added. Use .EQU to assign the names TEMP0–TEMP5 to locations \$60–\$65.

SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING and  
SECTION 2.7: ASSEMBLING AN AVR PROGRAM

44. Assembly language is a \_\_\_\_\_ (low, high)-level language while C is a \_\_\_\_\_ (low, high)-level language.
45. Of C and Assembly language, which is more efficient in terms of code generation (i.e., the amount of ROM space it uses)?
46. Which program produces the obj file?
47. True or false. The source file has the extension "asm".
48. True or false. The source code file can be a non-ASCII file.
49. True or false. Every source file must have .ORG and .EQU directives.
50. Do the .ORG and .SET directives produce opcodes?
51. Why are the directives also called pseudocode?
52. True or false. The .ORG directive appears in the ".lst" file.
53. The file with the \_\_\_\_\_ extension is downloaded into AVR Flash ROM.
54. Give three file extensions produced by AVR Studio.

SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN  
THE AVR

55. Every AVR family member wakes up at address \_\_\_\_\_ when it is powered up.
56. A programmer puts the first opcode at address \$100. What happens when the microcontroller is powered up?
57. Find the number of bytes each of the following instructions takes:  
(a) LDI R19, 0x5      (b) LDI R30, \$9F      (c) ADD R20, R21  
(d) ADD R22, R20      (e) LDI R18, 0x41      (f) LDI R28, 20  
(g) ADD R1, R3      (h) JMP
58. Write a program to (a) place each of your 5-digit ID numbers into a RAM locations starting at address 0x100, (b) add each digit to R19 and store the sum in RAM location 0x306, and (c) use the program listing to show the ROM memory addresses and their contents.
59. Find the address of the last location of on-chip program ROM for each of the following:  
(a) AVR with 32 KB      (b) AVR with 8 KB  
(c) AVR with 64 KB      (d) AVR with 16 KB  
(f) AVR with 128 KB
60. Show the lowest and highest values (in hex) that the ATmega32 program counter can take.
61. A given AVR has \$7FFF as the address of the last location of its on-chip ROM. What is the size of on-chip ROM for this AVR?
62. Repeat Question 61 for \$3FF.
63. Find the on-chip program ROM size in K for the AVR chip with the following address ranges:  
(a) \$0000–\$1FFF      (b) \$0000–\$3FFF  
(c) \$0000–\$7FFF      (d) \$0000–\$FFFF  
(e) \$0000–\$1FFFF      (f) \$00000–\$3FFFF  
(g) \$00000–\$FFF      (h) \$00000–\$1FF

64. Find the on-chip program ROM size in K for the AVR chips with the following address ranges:
- |                       |                       |
|-----------------------|-----------------------|
| (a) \$00000–\$3FF     | (b) \$00000–\$7FF     |
| (c) \$00000–\$7FFFF   | (d) \$00000–\$FFFFFF  |
| (e) \$00000–\$1FFFFFF | (f) \$00000–\$3FFFFFF |
| (g) \$00000–\$5FFF    | (h) \$00000–\$BFFFF   |

(Some of the above might not be in production yet.)

65. How wide is the program ROM in the AVR chip?
66. How wide is the data bus between the CPU and the program ROM in the AVR chip?
67. In instruction “LDI R21, K” explain why the K value cannot be larger than 255 decimal.
68. \$0C01 is the machine code for the \_\_\_\_\_ (LDI, STS, JMP, ADD) instruction.
69. In “STS memLocation, R22”, explain what the size of the instruction is and how it allows one to cover the entire range of the data memory in the AVR chip.
70. In “LDS Rd, memLocation”, explain what the size of the instruction is and how it allows one to cover the entire range of the data memory in the AVR chip.
71. Explain how the instruction “JMP target-addr” is able to cover the entire 4M address space of the AVR chip.

## SECTION 2.9: RISC ARCHITECTURE IN THE AVR

72. What do RISC and CISC stand for?
73. In \_\_\_\_\_ (RISC, CISC) architecture we can have 1-, 2-, 3-, or 4-byte instructions.
74. In \_\_\_\_\_ (RISC, CISC) architecture instructions are fixed in size.
75. In \_\_\_\_\_ (RISC, CISC) architecture instructions are mostly executed in one or two cycles.
76. In \_\_\_\_\_ (RISC, CISC) architecture we can have an instruction to ADD a register to external memory.
77. True or false. Most instructions in CISC are executed in one or two cycles.

## ANSWERS TO REVIEW QUESTIONS

### SECTION 2.1: THE GENERAL PURPOSE REGISTERS IN THE AVR

- LDI R29, 0x34
- LDI R18, 0x16  
LDI R19, 0xCD  
ADD R19, R18
- False
- FF hex and 255 in decimal
- 8

## SECTION 2.2: THE AVR DATA MEMORY

1. False
2. Data memory
3. 8
4. 3
5. 64K
6. 64

## SECTION 2.3: USING INSTRUCTIONS WITH THE DATA MEMORY

1. True
2. LDI R20,0x95  
OUT SPL,R20
3. LDI R19,2  
ADD R18,R19
4. LDI R20,0x16  
LDI R21,0xCD  
ADD R20,R21
5. FF in hex or 255 in decimal
6. R16
7. It copies the contents of R23 into the OCR0 I/O register.
8. The OCR0 presents the I/O address of the OCR0 register (\$3C); but we should use the data memory address while using the STS instruction. The instruction copies the contents of R23 into the location with the data memory address of \$3C (the EECR I/O register).

## SECTION 2.4: AVR STATUS REGISTER

1. SREG
2. 8 bits
- 3.

| Hex  | binary      |                                        |
|------|-------------|----------------------------------------|
| 9F   | 1001 1111   |                                        |
| + 61 | + 0110 0001 |                                        |
| 100  | 10000 0000  | This leads to C = 1, H = 1, and Z = 1. |

4.

| Hex  | binary      |                                        |
|------|-------------|----------------------------------------|
| 82   | 1000 0010   |                                        |
| + 22 | + 0010 0010 |                                        |
| A4   | 1010 0100   | This leads to C = 0, H = 0, and Z = 0. |

5.

| Hex  | binary      |                                        |
|------|-------------|----------------------------------------|
| 67   | 0110 0111   |                                        |
| + 99 | + 1001 1001 |                                        |
| 100  | 10000 0000  | This leads to C = 1, H = 1, and Z = 1. |

## SECTION 2.5: AVR DATA FORMAT AND DIRECTIVES

1. .EQU DATA1 = 0x9F  
.EQU DATA2 = \$9F
2. .EQU DATA1 = 0x99  
.EQU DATA2 = 99  
.EQU DATA3 = 0b10011001
3. If the value is to be changed later, it can be done once in one place instead of at every occurrence.

4. (a) \$34 (b) \$1F
5. 15 in decimal (0x0F in hex)
6. Value of location 0x200 = (0x95)
7. \$0C + \$10 = \$1C will be in data memory location \$63.

#### SECTION 2.6: INTRODUCTION TO AVR ASSEMBLY PROGRAMMING

1. The real work is performed by instructions such as LDI and ADD. Pseudo-instructions, also called assembly directives, instruct the assembler in doing its job.
2. The instruction mnemonics, pseudo-instructions
3. False
4. All except (c)
5. Assembler directives
6. True
7. (c)

#### SECTION 2.7: ASSEMBLING AN AVR PROGRAM

1. True
2. True
3. (a)
4. (b), (c), and (d)

#### SECTION 2.8: THE PROGRAM COUNTER AND PROGRAM ROM SPACE IN THE AVR

1. 22
2. True
3. 0000H
4. 2
5. 4
6. True

#### SECTION 2.9: RISC ARCHITECTURE IN THE AVR

1. RISC is reduced instruction set computer; CISC stands for complex instruction set computer.
2. True
3. Large
4. True
5. Around 130
6. True
7. (c)
8. False