

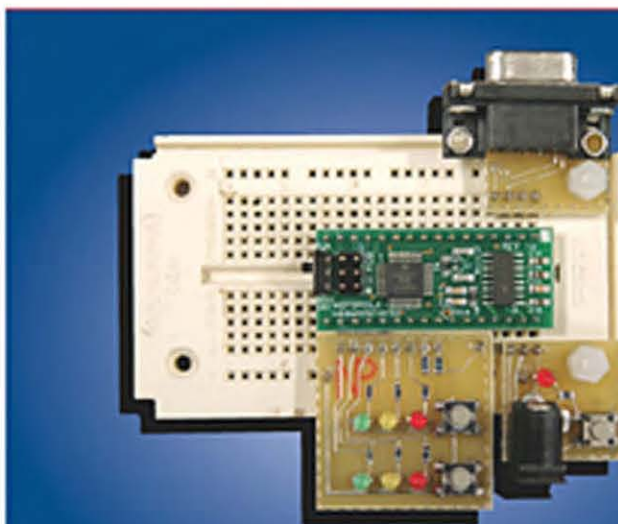


ACADEMIC PRESS SERIES IN ENGINEERING

Introduction to Microcontrollers

Second Edition

Architecture, Programming, and
Interfacing for the Freescale 68HC12



CD-ROM
Included

G. Jack Lipovski



Documents

Document Sharing Hub

Introduction To Microcontrollers

Second Edition

This Page Intentionally Left Blank

Second Edition

Introduction to Microcontrollers Architecture, Programming, and Interfacing for the Motorola 6812

G. J. Lipovski

**Department of Electrical and Computer Engineering
University of Texas
Austin, Texas**



Amsterdam • Boston • Heidelberg • London • New York • Oxford • Paris • San Diego •
San Francisco • Singapore • Sydney • Tokyo

Elsevier Academic Press
200 Wheeler Road, 6th Floor, Burlington, MA 01803, USA
525 B Street, Suite 1900, San Diego, California 92101-4495, USA
84 Theobald's Road, London WC1X 8RR, UK

This book is printed on acid-free paper.

Copyright © 2004, Elsevier Inc. All rights reserved. 

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopy, recording, or any information storage and retrieval system, without permission in writing from the publisher.

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone: (+44) 1865 843830, fax: (+44) 1865 853333, e-mail: permissions@elsevier.com.uk. You may also complete your request on-line via the Elsevier homepage (<http://elsevier.com>), by selecting "Customer Support" and then "Obtaining Permissions."

Library of Congress Cataloging-in-Publication Data

Lipovski, G. Jack.

Introduction to microcontrollers : architecture, programming and interfacing of the Motorola 6812 / G.J. Lipovski.

p. cm.

Includes index.

ISBN 0-12-451838-9 (alk. paper)

1. Programmable controllers. 2. Microprocessors. I. Title.

TJ223.P76L58 2004

629.8'95—dc22

2004052840

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN: 0-12-451838-9

For all information on all Academic Press publications
visit our Web site at www.academicpress.com

Printed in the United States of America

04 05 06 07 08 09 9 8 7 6 5 4 3 2 1

Dedicated to my father
Joseph Lipovski

This Page Intentionally Left Blank

Contents

Preface	xi
List of Figures	xvii
List of Tables	xxvii
Acknowledgments	xxix
About the Author	xxxix
Chapter 1 Basic Computer Structure and the 6812	1
1.1 Basic Computer Structure	1
1.2 The Instruction	5
1.3 A Few Instructions and Some Simple Programs	9
1.4 6812 Microcontroller Organizations	17
1.5 Variable Word Width	21
1.6 Summary and Further Reading	23
Chapter 2 The Instruction Set	29
2.1 Move Instructions	30
2.2 Arithmetic Instructions	37
2.3 Logic Instructions	41
2.4 Edit Instructions	44
2.5 Control Instructions	47
2.6 Input/Output Instructions	54
2.7 Special Instructions	54
2.8 Remarks	57
Chapter 3 Addressing Modes	63
3.1 Opcode Byte Addressing Modes	64
3.2 Post Byte Index Addressing Modes	67
3.3 Relative Addressing and Position Independence	75
3.4 Stack Index Addressing, Reentrancy, and Recursion	78
3.5 Architectural Notions of Addressing	85
3.6 Summary	88
Chapter 4 Assembly Language Programming	95
4.1 Introductory Example and Assembler Printout	97

4.2	Assembler Directives	101
4.3	Mechanics of a Two-Pass Assembler.	105
4.4	Character String Operations	109
4.5	A Simplified Two-Pass Assembler.	112
4.6	Debugging Source Code Programs	116
4.7	Summary.	117
Chapter 5	Advanced Assemblers, Linkers, and Loaders	129
5.1	Cross-Assemblers and Downloaders	129
5.2	Relocatable Assemblers and Loaders.	133
5.3	Conditional Assemblers	136
5.4	Macro Assemblers	138
5.5	Documentation	141
5.6	Summary.	143
Chapter 6	Assembly-Language Subroutines	149
6.1	Local Variables	151
6.2	Passing Parameters	162
6.3	Passing Arguments by Value, Reference, and Name	175
6.4	Calling and Returning Mechanisms.	176
6.5	Summary.	185
Chapter 7	Arithmetic Operations	193
7.1	Multiplication and Division	194
7.2	Integer Conversion.	198
7.3	From Formulas to Macro Programs	203
7.4	Simple Macro Expansions	208
7.5	Long Integer Arithmetic	211
7.6	Optimization	214
7.7	Floating-Point Arithmetic and Conversion.	216
7.8	Fuzzy Logic	226
7.9	Summary.	232
Chapter 8	Programming in C and C++.	239
8.1	Compilers and Interpreters	239
8.2	Operators and Assignment Statements.	242
8.3	Conditional and Loop Statements	244
8.4	Constants and Variables	247
8.5	Procedures and Their Arguments	250
8.6	An Example	253
8.7	Object-Oriented Programming in C++.	256
8.8	Summary.	266

Chapter 9	Implementation of C Procedures	275
9.1	Global and Local Variables.	276
9.2	Expressions and Assignment Statements	279
9.3	Conditional Statements.	287
9.4	Loop Statements, Arrays, and Structs	295
9.5	Procedure Calls and Arguments	300
9.6	Examples from Character String Procedures	304
9.7	Summary.	305
Chapter 10	Elementary Data Structures	313
10.1	What a Data Structure Is	313
10.2	Indexable Data Structures.	315
10.3	Sequential Data Structures	321
10.4	Linked List Structures	327
10.5	Summary.	333
Chapter 11	Input/Output	341
11.1	Input and Output Devices.	341
11.2	Parallel Ports	344
11.3	Input and Output Software	348
11.4	Synchronization Hardware	352
11.5	Gadfly Synchronization	353
11.6	Interrupt Synchronization.	356
11.7	Analog-to-Digital and Digital-to-Analog Conversion	362
11.8	UART Protocol	364
11.9	Summary and Further Reading	369
Chapter 12	Other Microcontrollers	379
12.1	The 6811.	379
12.2	The 6808.	382
12.3	The 6805.	386
12.4	The 68300 Series and ColdFire.	388
12.5	The 500 Series.	399
12.6	The M-CORE Series	403
12.7	Selecting a Microcontroller for an Application	407
Appendix 1	Number Representations and Binary Arithmetic	415
A1.1	Number Representations	415
A1.2	Binary Arithmetic	416
A1.3	Remarks	419
Appendix 2	Sequential Machine Interpreters	421
A2.1	Logic Control	421

A2.2	Timer Control	421
A2.2.1	A Two-Dimensional Array Cycle Interpreter	422
A2.2.2	A Table Cycle Interpreter	423
A2.3	Logic-Timer Control	424
A2.3.1	Mealy Model Sequential Machines	424
A2.3.2	Three-D Array Sequential Machine Interpreter	425
A2.3.3	Linked List Sequential Machine Interpreter	426
A2.4	A Moore Model Interpreter	428
A2.5	Synchronized Sequential Machines	429
A2.5.1	Graphical and Tabular Models	429
A2.5.2	A Linked List Interpreter	430
A2.6	An Interrupt-Based Synchronized Sequential Machine Interpreter	433
A2.7	Choice of an Interpreter	434
Appendix 3	Using Metrowerks Code Warrior	441
A3.1	Loading Metrowerks Code Warrior Software	441
A3.2	Opening Code Warrior for the Assembler Project.	441
A3.3	Running Examples for Chapters 1 through 3	442
A3.4	Running Examples for Chapters 4 through 11	443
A3.5	Downloading to a Target Board	444
A3.6	POD Mode or P&E Microcomputer Interface	445
A3.7	Running Multiple Metrowerks Applications	446

Preface

Programming is an essential engineering skill. To almost any engineer, it is as important as circuit design to an electrical engineer, as statistics to a civil engineer, and as heat transfer to a chemical engineer. The engineer has to program in high-level languages to solve problems. He or she also should be able to read assembly-language programs to understand what a high-level language does. Finally, he or she should understand the capabilities of a microcontroller because they are components in many systems designed, marketed, and maintained by engineers. The first goal of this book then is to teach engineers how a computer executes instructions. The second goal is to teach the engineer how a high-level language statement converts to assembler language. A final goal is to teach the engineer what can be done on a small computer and how the microcomputer is interfaced to the outside world. Even the nonprogramming engineer should understand these issues. Although this book is written for engineers, it will serve equally well for anyone, even hobbyists, interested in these goals.

The reader is taught the principles of assembly-language programming by being shown how to program a particular microcomputer, the Motorola 6812. The important thing about the 6812 is that it has a straightforward and clean, yet powerful, instruction set, midway between smaller and more powerful microcontrollers. From it the reader can adjust to these smaller or more powerful microcontrollers. The best way to learn these principles is to write a lot of programs, debug them, and see them work on a real microcontroller. This hands-on experience can be inexpensively obtained on the 6812. A recently introduced target board, the M68DKIT812C32, costs only about \$25 (see the front cover). Of course, this price doesn't include the personal computer that hosts the 6812 target system.

This book is designed for a sophomore course that teaches how a microcontroller works from the "bottom up." The reader learns how the binary adder is used in the add instruction and in index addressing, and then how instructions and addressing modes can be efficiently used to implement assembler language programs. This leads to a functional and practical understanding of how to compile a C or C++ program by hand to generate efficient assembler language programs. It also leads to a thorough understanding of how microcontrollers can be programmed to interact with the outside

world. Students who have taken the course using this textbook have testified to my department chairman, several years after graduation, that they have learned more about microcontrollers, and more about engineering design, from this course than from any of the other courses they took at our highly rated University! “Bottom-up” really works.

The following discussion outlines the book and explains several decisions that were made when we wrote the book. Optional chapters are available for readers having various interests. The main skills taught in each chapter are summarized.

Chapters 1 to 3 discuss programming using hand-translated machine code and the implementation of machine instructions in a simplified microcontroller. The assembler is not introduced until Chapter 4. This gives the engineering student a fine feeling for the machine and how it works and helps him or her resolve problems encountered later with timing in input/output programming or with the use of addressing modes in managing data structures. It removes the “magic” from computer programming. Chapter 1 explains how a microprocessor interacts with the memory and how it executes the instruction cycle. The explanation focuses on a microcomputer and is simplified to provide just enough background for the remainder of the text. Simple instructions and elementary programs are introduced next. Pointing out that there is no best program to solve a problem, Chapter 1 observes what makes a good program and encourages the reader to appreciate good programming style. A discussion of the available organizations of 6812 microcontrollers concludes this chapter.

In Chapter 2, the main concept is the alternative forms of the same kind of instruction on the 6812. Rather than listing the instructions alphabetically, as is desirable in a reference book, we group together instructions that perform the same type of function. Our groups are the classic ones, namely, the move, arithmetic, logical, edit, control, and input/output groups. Although other groupings are also useful, this one seems to encourage the student to try alternative instructions as a way of looking for the best instruction for his or her purpose. The 6812 has an extensive set of addressing modes that can be used with most instructions; these are covered in Chapter 3. The different addressing modes are introduced with a goal of explaining why these modes are useful as well as how they work. Examples at the end of the chapter illustrate the use of these modes with the instructions introduced in Chapter 2.

The end of Chapter 3 shows the use of program-relative addressing for position independence and the use of stack addressing for recursion and reentrancy.

Chapters 4 to 6 show how a program can be more easily written in assembler and the high-level C language and translated into machine code by an assembler. Chapter 4 introduces the assembler, explains assembler directives and symbolic addresses, and introduces limitations of forward referencing in a two-pass assembler. It actually explains a primitive assembler program, which helps the student understand how a symbol table works, as well as

how character strings are manipulated in any microcontroller program. Chapter 5, which may be omitted if the reader is not going to write assembler language programs, provides insights needed by programmers who write large assembler language programs. A general discussion of related programs, including macro, conditional, and relocatable assemblers and linkers, is given.

Chapter 6 develops assembler language subroutines. It illustrates techniques used in assembler language at an implementation level (such as storing local variables on the stack, or passing arguments in registers or on the stack). The trap instruction, a kind of subroutine, is discussed, which serves to introduce hardware interrupts in Chapter 11.

Chapter 7 covers arithmetic routines. An implementation is shown for unsigned and signed multiplication and division. This is important background for our later course on Computer Architecture. It also gives the students some taste for what is done when programming digital signal processors. Conversion between different bases is discussed and examples are given illustrating conversion from ASCII decimal to binary and vice versa. Stack operation and Polish notation are shown to be useful in realizing arithmetic routines of any complexity. Multiple-precision integer arithmetic is discussed by means of examples of 32-bit operations including multiplication and division. Floating-point representations are introduced, and the mechanics of common floating-point operations are discussed. Finally, a 6812-oriented introduction of fuzzy logic is presented.

Chapter 8, which may be omitted if the reader is already familiar with C, discusses compilers and interpreters and briefly introduces C programming to provide a background for later chapters.

Chapter 9 introduces the implementation of C procedures. Several constructs in C, such as switch statements, are shown implemented in assembler language. The techniques C uses to hold local variables and to pass arguments to a subroutine on the stack are shown implemented in assembler language. This chapter really brings home the meaning of C expressions and statements and helps the student understand what he or she is doing in any machine, not just a microcontroller, including programming of servers, workstations, and parallel computers. It also makes less magical programming in other languages than C or C++. In my opinion, it is for this reason that this course has become required of all Electrical and Computer Engineering students in our department, not just those who intend to design embedded systems.

Chapter 10 covers elementary data structures. The simplest, including the character string used in earlier chapters, and the more involved deque and linked list structures are related to the addressing modes available in the 6812. The main theme of this chapter is that the correct storage of data can significantly improve the efficiency of a program. This chapter also provides essential background for the discussion of interpreters in Appendix 2.

Chapter 11 introduces input/output programming. Input and output devices are characterized. Then the 6812's parallel ports are described. Input

and output software is illustrated with some examples. We then show 6812 synchronization hardware, to introduce gadfly and interrupt synchronization. Finally we show how D-to-A and A-to-D conversion is done, and how the 6812 uses the SCI to send and receive serial data.

Chapter 12 shows how the assembly language of a different microcontroller might be learned once that of the 6812 has been learned. Although we would like to discuss other popular microcontrollers, we believe that we could fill another book doing that. To illustrate the idea, we look at the near relatives less costly than the 6812, in particular, the 6805, 6808, and 6811. We also discuss briefly more powerful microcontrollers such as the 683xx, coldfire, 500, and MCORE series. The main theme is that once you understand the instruction set of one microcontroller, you can quickly learn to program efficiently on other microcontrollers.

Appendix 1 gives background in number systems that should be covered by prerequisite courses. Appendix 2 covers sequential machine and other interpreters. It is needed in our curriculum, and there is no better place to cover it than in the course that uses this textbook, but it might not be suitable for other programs. Thus it is included as an appendix that can be skipped if it is not needed. Appendix 3 covers techniques that can be used with the Metrowerks software on the enclosed CD-ROM.

This book systematically develops the concepts of programming of a microcontroller in high-level language and assembly language. It also covers the principles of good programming practice through top-down design and the use of data structures. It is suitable as an introductory text for a core course in an engineering curriculum on assembly language programming or as the first course on microcomputers that demonstrates what a small computer can do. It may also be used by those who want to delve more deeply into assembly language principles and practices. You should find, as we have, that programming is a skill that magnifies the knowledge and control of the programmer, and you should find that programming, though very much an important engineering skill, is fun and challenging. This book is dedicated to showing you that.

Problems are a major part of a good textbook. We have developed more than twenty problems for each chapter, and for each topic we generally have at least two problems, one that can be assigned for homework, while the other can be used in a quiz or exam. Some of these problems are “brain teasers” that are designed to teach the student that even simple programs should be tested, generally at their extreme values, to be sure they work. Often the obvious and simple solutions succumb to truncation or overflow errors. Also, problems in Chapter 11, including the keyless entry design and the experiment that plays “The Eyes of Texas” on a pair of earphones, are absolutely great motivators for sophomores who get them to work on a real microcontroller board. They help students see how exciting computer engineering is. This has a good effect on retention. An instructor’s manual, available from the publisher, includes solutions to all the problems given at the end of each chapter. We also provide worksheets that really drill the students

on the topic of hand-compiling C or C++ into efficient assembler language programs. These can be passed out in class and their problems solved as the examples in Chapter 9 are discussed. The instructor presents an example from the textbook, the students work the associated problem in the worksheet, and the solutions are discussed. After we introduced this teaching technique, the student's exam grades shot up 15%! Homework problems, lab exercises, and worksheet problems force students to teach themselves. Once the student learns how to teach himself or herself, he or she is prepared to tackle any new technology that might appear in the course of his or her professional life with assurance that he or she can handle it. This book does a lot to develop those essential skills that will make the student a successful Computer Engineer.

The introduction of the very inexpensive M68DKIT812C32 development board raises some challenges to the development of this second edition of the textbook. On one hand, we clearly see the advantages of this board. Students can buy it and work at home, where they get a much better appreciation of embedded systems than they do in the laboratory. We plan to use this board in a series of four courses to make it attractive to students. On the other hand, we have tons of Adapt812 development boards, and over a hundred M68HC12B32EVB development boards on our inventory that we have a hard time discarding because of state laws. We suspect that other Universities and Colleges have exactly the same problem. So this book in general, and Chapter 11 and Appendix 2 in particular, are written to be used primarily with the M68DKIT812C32 development board, but have sufficient details to adapt the examples to Adapt812 and M68HC12B32EVB development boards. These details can be easily skipped over by students who use the new M68DKIT812C32 development board.

This book was developed largely from a book by the author and T. J. Wagner on the 6809. The author expresses his gratitude for the contributions made by Dr. Wagner through his writing of much of the earlier book.

This Page Intentionally Left Blank

List of Figures

Figure 1.1.	Simplified Computer Structure	2
Figure 1.2.	Registers in the 6812	6
Figure 1.3.	Registers and Memory	10
Figure 1.4.	Data Operator Arithmetic	12
Figure 1.5.	Program for 8-Bit Addition	12
Figure 1.6.	Bits in the Condition Code Register	13
Figure 1.7.	Addition of 2-Byte Numbers	13
Figure 1.8.	Program for 16-Bit Addition.	13
Figure 1.9.	Alternative Program for 16-Bit Addition	14
Figure 1.10.	Most Efficient Program for 16-Bit Addition.	16
Figure 1.11.	Single-Chip Mode of the MC68HC812A4.	17
Figure 1.12.	Expanded Wide Multiplexed Bus Mode of the MC68HC812A4	18
Figure 1.13.	Single-Chip Mode of the MC68HC912B32.	19
Figure 1.14.	Single-Chip Mode of the MC9S12C32	19
Figure 1.15.	Memory Maps for the 6812	20
Figure 1.16.	Variable Word Width Implementation	21
Figure 2.1.	A Program Segment to Clear a Byte	31
Figure 2.2.	A Stack	32
Figure 2.3.	Transfers Between Registers.	35
Figure 2.4.	Program Segment to Initialize the Condition Code Register.	37
Figure 2.5.	Program Segment for 32-Bit Addition.	39
Figure 2.6.	Multiplication	40
Figure 2.7.	Program Segment for 16-Bit Unsigned Multiplication . . .	41
Figure 2.8.	Program Segment for BCD Addition.	41
Figure 2.9.	Program Segment for Conversion from Celsius to Fahrenheit	41
Figure 2.10.	Common Logic Operations.	43
Figure 2.11.	Bit-by-Bit AND	43

Figure 2.12.	Shifts and Rotates	45
Figure 2.13.	Shift Hardware	46
Figure 2.14.	Program Segment to Swap Nibbles.	46
Figure 2.15.	Program Segment for Insertion of Some Bits.	46
Figure 2.16.	Decision Tree	49
Figure 2.17.	Program Segment for a Decision Tree.	49
Figure 2.18.	Program Segment for Setting, Clearing, and Testing of a Bit	50
Figure 2.19.	Program Segment for a Wait Loop	51
Figure 2.20.	Subroutine and Handler Addresses	52
Figure 2.21.	Program Segment for Swap Subroutine.	54
Figure 2.22.	Program Segment for Ensuring a Value is Between Limits	56
Figure 2.23.	Program Segment for a Multiply and Add Operation	56
Figure 3.1.	Opcode Byte Coding	67
Figure 3.2.	Post Byte Coding.	69
Figure 3.3.	Program Segment to Add 2 Bytes Using Vector Indexing	69
Figure 3.4.	Offset Calculation	70
Figure 3.5.	Autoincrement Address Calculation	72
Figure 3.6.	Program Segment to Add 2 Bytes Using Autoincrementing	72
Figure 3.7.	Accumulator Index Address Calculation	73
Figure 3.8.	Simplified Control Hardware for Relative Addressing	75
Figure 3.9.	Program Segment Using BRA, Illustrating Position Independence.	76
Figure 3.10.	Program Segment to Put a Square Wave on an Output Port.	76
Figure 3.11.	A Stack Buffer for Two Stacks	79
Figure 3.12.	Subroutine to Compute $n!$ Recursively	80
Figure 3.13.	Subroutine to Compute $n!$ in a Loop.	80
Figure 3.14.	Nonreentrant Subroutine to Clear Memory	81
Figure 3.15.	Reentrant Subroutine to Clear Memory.	81
Figure 3.16.	A Stack Buffer for Nested Segments.	83
Figure 3.17.	Program Segment to Clear a Block of Memory	83
Figure 3.18.	Program Segment to Move a Block of Memory.	84
Figure 3.19.	Program Segment to Add Vector Elements	84
Figure 3.20.	Program Segment to Compute an Inner Product	85
Figure 3.21.	A Table	93
Figure 4.1.	Program MAX.	97
Figure 4.2.	Assembler Source Code for the Program MAX.	98

Figure 4.3.	Assembler Listing for the Program MAX	98
Figure 4.4.	Assembler Listing for the Program Square	105
Figure 4.5.	Program to Select Negative Odd Numbers	106
Figure 4.6.	Program with Illegal Forward Reference	108
Figure 4.7.	Program without Forward Reference	108
Figure 4.8.	Print Program	109
Figure 4.9.	Character Move Program Segment and Data	110
Figure 4.10.	Character String Compare Program Segment and Data	111
Figure 4.11.	Program Segment to Insert an Entry into the Symbol Table	111
Figure 4.12.	Subroutine to Insert a Label as an Operand from the Symbol Table	112
Figure 4.13.	Subroutine to get the Opcode	112
Figure 4.14.	Converting ASCII Hex String to a Binary Number	113
Figure 4.15.	Simple Computer and Assembler SA1 Specifications	114
Figure 4.16.	Machine and Source Code	114
Figure 4.17.	Assembler Main Program	114
Figure 4.18.	Assembler Directives	114
Figure 4.19.	Assembler Pass 1	115
Figure 4.20.	Assembler Pass 2	115
Figure 4.21.	A Program with Errors	117
Figure 4.22.	Program with Relative Address Reference	120
Figure 4.23.	Program with Direct or Page Zero Reference	120
Figure 4.24.	Program with Errors	123
Figure 4.25.	Another Program with Errors	123
Figure 4.26.	Assumptions for SA2 Assembler	124
Figure 4.27.	Machine and Source Code for SA2	125
Figure 4.28.	FINDLBL Subroutine with Errors	125
Figure 4.29.	Line Print Subroutine with Errors	126
Figure 5.1.	Loader Record	131
Figure 5.2.	An S-Record	131
Figure 5.3.	A Program	133
Figure 5.4.	Sections in Different Source Code Files	134
Figure 5.5.	A Parameter File	135
Figure 5.6.	Loop Macro to Add Consecutive Values	140
Figure 5.7.	Recursive Macro to Add up to Eight Values	141
Figure 5.8.	Example Control Sequence Program	143
Figure 5.9.	Flowchart for Example Control Sequence	144
Figure 6.1.	Subroutine Calling and Returning	150

Figure 6.2.	Program Segment with a Plurality of Entry and Exit Points	152
Figure 6.3.	Inner Product Utilizing a Global Variable such as TEMP (a Bad Example)	153
Figure 6.4.	Changing a Global Variable before It Has Been Completely Used	153
Figure 6.5.	Inner Product Utilizing Different Global Variables (a Bad Example)	154
Figure 6.6.	Inner Product Utilizing Registers	154
Figure 6.7.	Inner Product Program Segment Utilizing Local Variables on the Stack	156
Figure 6.8.	Nested Subroutines Using Local Variables Stored on the Stack	156
Figure 6.9.	Local Variables Stored on the Stack, for Successive Subroutines	157
Figure 6.10.	Using Symbolic Names for Stacked Local Variables . . .	158
Figure 6.11.	Defining Symbolic Names for Stacked Local Variables by Sizes	158
Figure 6.12.	Declaring Symbolic Names for Local Variables Using DS Directives	159
Figure 6.13.	Declaring Symbolic Names for Extended Local Access.	160
Figure 6.14.	Accessing Stacked Local Variables Using a Stack Marker	161
Figure 6.15.	A Subroutine with Parameters in Registers	163
Figure 6.16.	Change a Global Parameter before Its Subroutine Has Used It	164
Figure 6.17.	A Subroutine with Parameters in Global Variables	165
Figure 6.18.	Calling a Subroutine for Figure 6.17	165
Figure 6.19.	Calling a Subroutine with Parameters on the Stack for Figure 6.21	166
Figure 6.20.	Location of Parameters Passed on the Hardware Stack.	166
Figure 6.21.	A Subroutine with Parameters on the Stack	167
Figure 6.22.	Revised Subroutine with Local Variables and Parameters on the Stack	167
Figure 6.23.	Calling a Subroutine with Parameters on the Stack for Figure 6.22	168
Figure 6.24.	Parameters Passed after the Call	159
Figure 6.25.	A Subroutine with Parameters after the Call, Which Pulls the Return	170

Figure 6.26.	A Subroutine Calling Sequence for Figure 6.25	170
Figure 6.27.	A Subroutine with Parameters after the Call, Which Uses RTS	171
Figure 6.28.	A Subroutine Call with Parameters after the Call for Figure 6.27	171
Figure 6.29.	A Subroutine with In-Line Parameters That Are Addresses	172
Figure 6.30.	An In-Line Argument List of Addresses for Figure 6.29	172
Figure 6.31.	Implementation of a C or C++ Switch—Case Statement	173
Figure 6.32.	Parameters in a Table	174
Figure 6.33.	Calling Sequence for Passing Arguments in a Table.	174
Figure 6.34.	Simple Subroutine	177
Figure 6.35.	A Subroutine Saving and Restoring Registers	177
Figure 6.36.	Saving and Restoring All the Registers	178
Figure 6.37.	A Subroutine with Multiple Entry Points.	179
Figure 6.38.	Calling the <i>i</i> th Subroutine for Figure 6.37	179
Figure 6.39.	A Jump Vector.	180
Figure 6.40.	Calling the <i>i</i> th Subroutine of a Jump Vector	180
Figure 6.41.	Emulation of SWI	180
Figure 6.42.	Emulation of RTI	181
Figure 6.43.	An SWI handler.	181
Figure 6.44.	Calling an SWI handler	182
Figure 6.45.	A Trap Handler	183
Figure 6.46.	Calling a Trap Handler.	184
Figure 6.47.	Program for Problem 13.	188
Figure 7.1.	An 8-Bit Unsigned Multiply Subroutine	195
Figure 7.2.	An 8-Bit Unsigned Divide Subroutine.	196
Figure 7.3.	An 8-Bit Signed Multiply Subroutine	197
Figure 7.4.	Conversion from Decimal to Binary by Multiplication by Powers of 10.	200
Figure 7.5.	Conversion from Decimal to Binary by Multiplication by 10.	201
Figure 7.6.	Conversion from Binary to Decimal by Recursive Division by 10.	202
Figure 7.7.	Passing Parameters on the Stack.	204
Figure 7.8.	Stack Movement When Evaluating (7)	205
Figure 7.9.	Finding a Parsing Tree for (7).	206
Figure 7.10.	Method to Write a Sequence of Subroutine Calls for (7)	207

Figure 7.11.	PUSH and PULL Macro Expansions for 8-Bit Data	209
Figure 7.12.	Generating Code for Arithmetic on the 6812	210
Figure 7.13.	Generating Code for 32-Bit Arithmetic on the 6812	211
Figure 7.14.	Add Subroutine for 32-Bit Numbers	212
Figure 7.15.	A 32-Bit Negation Subroutine.	212
Figure 7.16.	Multiplication of 32-Bit Numbers	213
Figure 7.17.	A 32-Bit by 32-Bit Unsigned Multiply Subroutine.	214
Figure 7.18.	A 32-Bit by 32-Bit Unsigned Divide Subroutine	215
Figure 7.19.	Optimized Implementation for 8-Bit Data	216
Figure 7.20.	Procedure for Floating-Point Addition.	225
Figure 7.21.	Fuzzy Logic Control System	227
Figure 7.22.	Membership Functions	228
Figure 7.23.	A Fuzzy Inference Kernel.	231
Figure 8.1.	Conditional Statements.	245
Figure 8.2.	Case Statements.	245
Figure 8.3.	Loop Statements	246
Figure 8.4.	A Huffman Coding Tree.	253
Figure 8.5.	Other Huffman Codes	269
Figure 9.1.	A Simple Program	276
Figure 9.2.	C Program with Local and Global Variables	280
Figure 9.3.	C Program with Some Operators.	282
Figure 9.4.	C Program with Incrementing and Decrementing.	284
Figure 9.5.	C Program with ORing, ANDing, and Shifting	285
Figure 9.6.	A Program with Boolean Operators.	288
Figure 9.7.	A Program with If-Then Expressions	291
Figure 9.8.	Assembly Language for a Decision Tree	293
Figure 9.9.	Alternative Assembly Language for a Case Statement	295
Figure 9.10.	For, While, and Do While Loops	296
Figure 9.11.	Array Manipulation Program	298
Figure 9.12.	Clearing a Vector with a Do While Loop.	299
Figure 9.13.	A Program Setting and Testing a Bit	299
Figure 9.14.	A Subroutine to Raise a Number to a Power	301
Figure 9.15.	Stack for <i>power</i> Procedure	301
Figure 9.16.	An Object and Its Pointers	302
Figure 9.17.	The Strlen Procedure	304
Figure 9.18.	The Strchr Procedure	305
Figure 9.19.	The Strncpy Procedure.	306
Figure 9.20.	The Strncmp Procedure	307
Figure 10.1.	A Vector	314
Figure 10.2.	A Histogram	316

Figure 10.3.	An Array	318
Figure 10.4.	Subroutine ZTRANS	319
Figure 10.5.	Deque Data Structure	324
Figure 10.6.	Buffer for a Deque Wrapped on a Drum	324
Figure 10.7.	Subroutines for Pushing and Pulling B from the Top of the Deque	326
Figure 10.8.	Tree Representing the String (12)	328
Figure 10.9.	Linked List Representation of the Tree Shown in Figure 10.8	328
Figure 10.10.	Path for Scanning the Tree	329
Figure 10.11.	Flowchart for Scanning Tree.	330
Figure 10.12.	Subroutine SCAN Using Indexes	331
Figure 10.13.	Subroutine SCAN Using Address Pointers.	332
Figure 10.14.	Linked List Data Structure for SCAN	332
Figure 10.15.	Graph of Linked List for Problem 10.21	339
Figure 11.1.	Diagram of a Microcomputer (Identical to Figure 1.1).	342
Figure 11.2.	A Memory and Its Connection to the MPU (Compare to Figure 1.1)	342
Figure 11.3.	Some 6812 Parallel I/O Ports	344
Figure 11.4.	The Timer Subsystem.	346
Figure 11.5.	Simple Devices	347
Figure 11.6.	Bouncing and Racing	348
Figure 11.7.	Program to Output a Buffer	349
Figure 11.8.	Program to Input to a Buffer.	350
Figure 11.9.	A Traffic Light Application.	350
Figure 11.10.	Flowcharts for Programmed I/O	351
Figure 11.11.	Gadfly Synchronization Hardware.	353
Figure 11.12.	Gadfly Synchronization Software	354
Figure 11.13.	Gadfly Geiger Counter Application	354
Figure 11.14.	Gadfly Traffic Light Application	355
Figure 11.15.	Interrupt Hardware.	357
Figure 11.16.	Steps in Processing an Interrupt	357
Figure 11.17.	Interrupt Geiger Counter Application	359
Figure 11.18.	Interrupt Traffic Light Application	360
Figure 11.19.	D-to-A Converter.	362
Figure 11.20.	6812 A–D Subsystem.	363
Figure 11.21.	A-to-D Conversion.	364
Figure 11.22.	Data Logging.	365
Figure 11.23.	Frame Format for UART Signals	365
Figure 11.24.	6812 Serial Communication Interface	366

Figure 11.25.	C Program	368
Figure 11.26.	Interrupt SCI Program	369
Figure 11.27.	High-Current Parallel Output Port.	371
Figure 11.28.	Parallel Input Port of the 'A4 or 'B32.	371
Figure 11.29.	Parallel D-to-A Output Device	372
Figure 11.30.	Johnson Counter Sequence	372
Figure 11.31.	Program with Disassembly	375
Figure 11.32.	Another Program with Disassembly	376
Figure 12.1.	A 6811 Dot Product Subroutine	380
Figure 12.2.	Registers of the 6808	383
Figure 12.3.	Bits in the 6808 Condition Code Register	383
Figure 12.4.	A 6808 Dot Product Subroutine	386
Figure 12.5.	The Register Set of the 6805	387
Figure 12.6.	Bits in the 6805 Condition Code Register	387
Figure 12.7.	A 6805 Dot Product Subroutine	389
Figure 12.8.	Register Set of 68300 Series Microcomputers	390
Figure 12.9.	Memory Spaces in the 68300 Series	391
Figure 12.10.	A 68300 Series Dot Product Subroutine	398
Figure 12.11.	User Registers for the 500 Series	399
Figure 12.12.	A 500 Series Dot Product Subroutine	403
Figure 12.13.	M-CORE Register Set	403
Figure 12.14.	An M-CORE Dot Product Subroutine	407
Figure A1.1.	Four-Bit Binary Representations	417
Figure A2.1.	Traffic Light Sequence	422
Figure A2.2.	A Two-D Array Cycle Interpreter	422
Figure A2.3.	A Table Interpreter.	423
Figure A2.4.	A Mealy Sequential Machine	424
Figure A2.5.	A Three-D Array Interpreter	425
Figure A2.6.	A Linked List Interpreter Using Indexes	426
Figure A2.7.	A Linked List Interpreter Using Addresses	426
Figure A2.8.	A Linked List Structure	427
Figure A2.9.	A Moore Model Sequential Machine.	428
Figure A2.10.	A Linked List Moore Model Interpreter Using Indexes	428
Figure A2.11.	A Synchronized Sequential Machine	429
Figure A2.12.	A Linked List Synchronized Sequential Machine Interpreter	431
Figure A2.13.	A Mealy Sequential Machine	435
Figure A2.14.	Another Mealy Sequential Machine	435
Figure A2.15.	A Moore Sequential Machine	436
Figure A2.16.	Another Moore Sequential Machine	436

Figure A2.17. A Synchronized Sequential Machine	436
Figure A2.18. Another Synchronized Sequential Machine	437
Figure A2.19. A Moore Model Synchronized Sequential Machine Interpreter	438
Figure A2.20. Parallel Input and Parallel D-to-A Output Device.	439

This Page Intentionally Left Blank

List of Tables

Table 2.1	Move Instructions Using an Effective Address	32
Table 2.2	Stack Move Instructions	33
Table 2.3	Special Move Instructions	34
Table 2.4	Add Instructions Using an Effective Address	38
Table 2.5	Arithmetic Instructions That Do Not Use an Effective Address	38
Table 2.6	Logic Instructions	42
Table 2.7	Edit Instructions	45
Table 2.8	Conditional Branch Instructions	48
Table 2.9	Other Control Instructions	50
Table 2.10	Special Instructions	55
Table 4.1	ASCII Codes	96
Table 4.2	Addressing Modes	100
Table 4.3	Assembler Directives	101
Table 5.1	S-Record Fields	132
Table 5.2	Relocation Directives	134
Table 5.3	Conditional Directives	136
Table 5.4	Macro Directive	139
Table 5.5	Macro Arguments	139
Table 5.6	Listing Directives	142
Table 8.1	Conventional C Operators Used in Expressions	243
Table 8.2	Special C Operators	243
Table 8.3	Conditional Expression Operators	246
Table 11.1	Interrupt Vectors in the 6812'A4	358
Table 12.1	Instruction Set and Addressing Modes of the 6811	381
Table 12.2	Instruction Set and Addressing Modes of the 6808	384
Table 12.3	Instruction Set and Addressing Modes of the 6805	388
Table 12.4	Addressing Modes for the 68300 Series	392
Table 12.5	Move and Arithmetic Instructions for the 68300 Series	393
Table 12.6	Other Instructions for the 68300 Series	394
Table 12.7	Instructions for the 500 Series	400
Table 12.8	Instructions for the M-CORE Series	404

This Page Intentionally Left Blank

Acknowledgments

The author would like to express his deepest gratitude to everyone who contributed to the development of this book. The students of EE 319K at the University of Texas at Austin during Fall 1998 significantly helped correct this book; special thanks are due to Chao Tan, Brett Wisdom, Alex Winbow, Eric Wood, John Prochnow, and Nathan Madins for finding the most errors. After the first edition went to print, additional errors were found, principally, by Omar Azzam, Alex Kuo, James Tanghongs, and Allison Holloway. This text was prepared and run off using Windows 2000 and LaserWriter, running Microsoft Word. The book cover was photographed by Kirk Tuck (used with permission). I am pleased to write this description of the Motorola 6812, which is an incredibly powerful component and a vehicle for teaching a considerable range of concepts.

G. J. L.

This Page Intentionally Left Blank

About the Author

G. Jack Lipovski has taught electrical engineering and computer science at the University of Texas since 1976. He is a computer architect internationally recognized for his design of the pioneering database computer, CASSM, and the parallel computer, TRAC. His expertise in microcomputers is also internationally recognized by his being a past director of Euromicro and an editor of *IEEE Micro*. Dr. Lipovski has published more than 70 papers, largely in the proceedings of the annual symposium on computer architecture, the IEEE transactions on computers, and the national computer conference. He holds eight patents, generally in the design of logic-in-memory integrated circuits for database and graphics geometry processing. He has authored seven books and edited three. He has served as chairman of the IEEE Computer Society Technical Committee on Computer Architecture, member of the Computer Society Governing Board, and chairman of the Special Interest Group on Computer Architecture of the Association for Computer Machinery. He has been elected Fellow of the IEEE and a Golden Core Member of the IEEE Computer Society. He received his Ph.D. from the University of Illinois, 1969, and has taught at the University of Florida and at the Naval Postgraduate School, where he held the Grace Hopper chair in Computer Science. He has consulted for Harris Semiconductor, designing a microcomputer, and for the Microelectronics and Computer Corporation, studying parallel computers. He founded the company Linden Technology Ltd. and is the chairman of its board. His current interests include parallel computing, database computer architectures, artificial intelligence computer architectures, and microcomputers.

This Page Intentionally Left Blank

Basic Computer Structure and the 6812

Computers, and microcomputers in particular, are among the most useful tools that humans have developed. They are not the news media's mysterious half-human forces implied by "The computer will decide . . ." or "It was a computer error!" No, computers are actually like levers; as a lever amplifies what the human arm can do, so the computer amplifies what the human brain can do. Good commands are amplified, and the computer is a great tool, but bad commands are likewise amplified, and good commands incorrectly programmed are also amplified. "To err is human, but to really foul things up, you need a computer." You have to study and exercise this tool to make it useful; that is the purpose of this book. The computer also has to be used with insight and consideration for its effects on society, but that will not be studied in this book.

We shall study the computer as an engineer studies any tool—we begin by finding out just how it ticks. We make our discussion concrete using the superior Motorola 6812 microcomputer designed by Motorola, as a means of teaching the operations of computers in general. In this chapter we introduce basic computer structure. We discuss memory, how memory words are read to tell the microcomputer what to do, and how these words are written and read to save the microcomputer's data. Finally, we describe a small but useful subset of 6812 instructions to show how a computer reads and carries out an instruction and a program, to introduce the idea of programming.

After reading this chapter, you should be able to approach a typical instruction, to be introduced in the next two chapters, with an understanding about what the mnemonic, the machine code, and a sequence of memory reads and writes may mean for that instruction. This chapter then provides background for the discussion of instructions that we will present in the next two chapters.

► 1.1

Basic Computer Structure

What is a microcomputer, and how does it execute the instructions that a programmer writes for it? This question is explored now at a level of abstraction that will be adequate for this text. We do know that many readers will object to one aspect of the following discussion, and we want to answer that

objection *a priori*, so that those readers will not miss the point. We will introduce a seemingly large number of terms. Don't miss the objective: We are really introducing concepts. The reader should think about the concepts rather than memorize definitions. Like your physics text, this text has to use terms in a fairly precise way to avoid ambiguity. Your physics text, you may recall, used the word "work" in a very precise way, as the product of force times distance, which is a bit different from the conversational use of the word "work" as used in the expression, "He's doing a lot of work." We will use terms such as "read" and "fetch" in a similar way. When defined, they will be written in *italics* and will be listed in the index. We ask you to learn the term and its meaning even though you do not have to memorize the wording of the definition. But take heart, because although we do have a few concepts that have to be learned, and we have to learn the terms for those concepts, we do not have many formulas or equations to deal with. Accept our challenge to understand these terms; then you will enjoy the latter discussions even more than if you muddle through this section without thinking about the terminology.

You probably know what microcomputers and computers are, to some degree, but let us discuss the term "computer" so that if you get into an argument about whether a hand calculator is a computer, you can respond knowledgeably.

A microcomputer is a kind of computer or, more accurately, a kind of von Neumann computer, named after the scientific giant of our century who invented it. All *von Neumann computers* have four components: *memory*, *controller*, *data operator* (sometimes called an arithmetic-logic unit), and *input-output (I/O)*, which are connected by an address and data bus. A simplified diagram of a computer is shown in Figure 1.1. Briefly, the memory stores both the data and the program, and the input-output provides the communication with the outside world. In a conventional personal computer system input-output is done through peripherals such as CRTs, keyboards, scanners, printers, modems, and so on. In typical microcontroller applications the input-output system provides the necessary connections, or interfacing, to the device, of which the microcontroller is a component, such as an automobile, kitchen appliance, toy, or laboratory instrument. The data operator performs arithmetic and logical operations on data, such as addition, ANDing, and so on. The controller controls the flow of information between the components by means of control lines (which are not shown in Figure 1.1), directing what computation is to be done. The input/output, con-

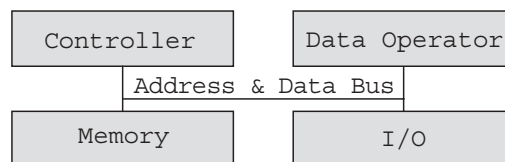


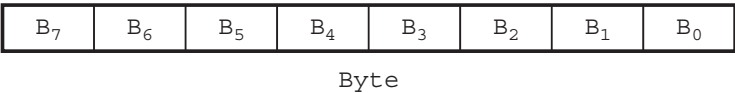
FIGURE 1.1. Simplified Computer Structure

troller, and data operator may themselves contain a small amount of memory in *registers*.

A microcomputer is a computer that is implemented using low-cost integrated circuits (ICs) and is therefore cheap enough to be used in an incredible range of applications for which a large computer would be infeasible. For the purposes of this book, if the data operator and the controller are together on a single IC, but other ICs are needed to complete the computer, that IC is called a *microprocessor*; the computer that uses a microprocessor is called a *microcomputer*; and, if a complete computer is on a single integrated circuit, that integrated circuit is called a *single-chip microcontroller*.

Some aspects of microcomputers apply to all computers. We will often discuss an aspect of the computer and, of course, that aspect applies to microcontrollers, on which we are concentrating. The microcomputer's, or microcontroller's, controller and data operator is abbreviated *MPU* (microprocessor unit). The abbreviation CPU (central processor unit) is often used to denote the controller and data operator, but that term leads subtly to the idea that the CPU is central and most important, but this is misleading, especially when a computer system has many MPUs, none of which is "central."

We now look more closely at the memory and the MPU. We can think of the memory as a large number of cells, each able to store a 0 or a 1—that is, a binary digit or 1 bit of memory. The bits are grouped together in units called *bytes*, which consist of 8 bits. Within a particular byte the bits are labeled B₇, . . . , B₀ as shown.



The right-hand bits are called lower-order or least significant, and the left-hand bits are called higher-order or most significant. There is nothing sacred about this labeling, and, in fact, many computer manufacturers reverse it. A *word* in memory is the number of bits that are typically read or written as a whole. In small microcomputers, a word is 1 byte, so that the terms "word" and "byte" are often used interchangeably. In this text, the 6812 can read an 8-bit or a 16-bit word, which is 2 bytes. In a 16-bit word, bits are numbered from 15 (on the left) to 0 (on the right). In the memory, each byte has an address between 0 and $2^N - 1$, where N is the total number of address bits. In the 6812, N is essentially 16, so each address between 0 and 65,535 is described by its 16-bit binary representation (see Appendix 1), although some 6812 versions can extend this range.

The MPU controls the memory by means of a clock and a read/write line and communicates to it by an address bus and a data bus, shown in Figure 1.1. A *line* or *wire* carries 1 bit of information at each instance of time by the level of voltage on it. Each line can carry a true (1) or a false (0) value. A *bus* is a collection of lines that can move a word or an address in

parallel, so that 1 bit of a word or address is on one line of the bus. The data bus moves an 8-bit or 16-bit word to or from memory and from or to the MPU, and the address bus moves a 16-bit address from the MPU to the memory. A *clock* is a signal that is alternately a 0 and 1 (a square wave). A *clock cycle* is the time interval from when the clock just becomes 0 until the next time it just becomes 0, and the *clock rate*, or *clock frequency*, is the reciprocal of the clock cycle time. Contemporary 6812 microcontrollers essentially use a 24 MHz clock.

In one clock cycle, the MPU can *read* a word from the memory by putting an address on the address bus and making the *read/write line* 1 for the cycle and then picking up the word on the data bus at the end of the cycle. It can also *write* a word into the memory in one clock cycle by putting the address on the address bus and putting the word on the data bus and making the read/write line 0 for the cycle. A read or a write to memory is also called an *access* to memory.

We can enlarge our description of how the memory works. Assume that we want to get the contents of a particular word or byte from memory, that is, read a word from memory. The MPU first puts a 1 for read on the read/write line and then puts the address of the desired word or byte on the address bus throughout the duration of a clock cycle. The memory is designed so that, at the end of the clock cycle, the desired word is put on the data bus. The MPU then places a copy of the contents of the word or byte on the data bus into some register inside the MPU as required by the instruction that it is executing. This is done without changing the contents in memory of the word or byte addressed.

To write a word into memory, the address of the word is put on the address bus, the word is put on the data bus, and the read/write line has 0 (to indicate a write) for a full clock cycle. The memory is designed to store the word at that address at the end of the clock cycle. After the word is stored at the given address, the MPU may still retain in one of its registers a copy of the word that has just been written.

The MPU can read or write an 8-bit or a 16-bit word in memory in one clock cycle. Such a memory is usually called *random access memory (RAM)* because each byte is equally accessible or can be selected at random without changing the time of the access. With microcomputer applications, it is not unusual to have part of the memory bytes in *read only memory (ROM)*. A ROM is capable of a read operation but not a write operation; its words are written when it is made at the factory and are retained even when the power is turned off. If the data in a RAM are lost when power is turned off, the RAM is termed *volatile*; otherwise, it is termed *nonvolatile*. RAM memories are essentially volatile. The term RAM is also used almost universally to imply memories that you can read and write in, even though ROM memories can be randomly accessed with a read operation. The part of memory that is in ROM is typically used to store a *program* for a microcomputer that only executes one program. For example, the microcontroller used in an automobile would be running the same program every time it is

used, so that the part of the memory that is used for storing the program is in ROM.

► 1.2 The Instruction

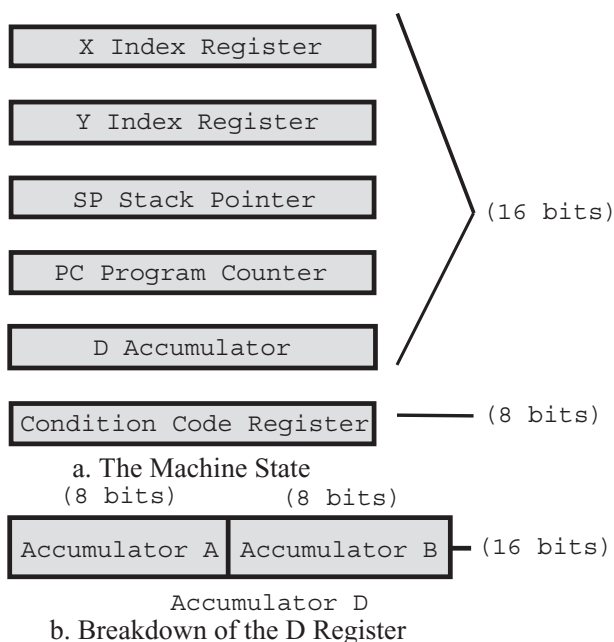
We now examine the notion of an *instruction*, which is an indivisible operation performed by the MPU. It can be described statically as a collection of bits stored in memory or as a line of a program, or dynamically as a sequence of actions by the controller. In this discussion we begin with a simplified dynamic view of the instruction and then develop a static view. Examples are offered to combine these views to explain the static aspects of the operation code, addressing mode, machine code, and mnemonics. We conclude with an expanded view of the dynamic aspects of the instruction cycle.

The controller will send commands to memory to read or write and will send commands to all other parts of the computer to effectively carry out the intentions of the programmer. The specification of what the control unit is to do is contained in a program, a sequence of instructions stored, for the most part, in consecutive bytes of memory. To execute the program, the MPU controller repeatedly executes the *instruction cycle* (or *fetch/execute cycle*):

1. Fetch (read) the next instruction from memory.
2. Execute the instruction.

As we shall see with the 6812 MPU, reading an instruction from memory will require that 1 or more bytes be read from memory. To execute the instruction, some bytes may also be read or written. These two activities, read and execute, seem to be read or write operations with the memory but are quite different to the MPU, and we use different terms for them. To *fetch* means to read a word from memory to be used as an instruction in the controller. The first step in the cycle shown previously is the *fetch phase*. To *recall* means to read a word into the MPU that is not part of the instruction. The recall and write operations are done in the second step of the instruction, which is called the *execute phase*. Thus, when we talk about fetching a word, you can be sure that we are talking about reading the instruction or part of the instruction.

The 6812's registers are shown in Figure 1.2, where the top five registers hold 16 bits and the condition code register holds 8 bits. The 16-bit D register is composed of two 8-bit registers A and B; D, A, and B are called *accumulators* because arithmetic operations can be done with their contents with the results placed back in the registers to accumulate the result. This accumulating aspect of registers D, A, and B will be assumed to be understood so that we often refer to (register) "D," "A," or "B" rather than "accumulator D," "accumulator A," or "accumulator B." The registers A and B are always the left and right halves of register D; if you put \$12 in register A and \$34 in register B then read register D, it has \$1234. Similarly, if you put \$5678 in register D, then reading register A gives \$56 and reading register

**FIGURE 1.2.** Registers in the 6812

B gives \$78. Registers X and Y are index registers, and SP is a stack pointer; they are used in address calculations. The program counter, PC, is used to fetch the instruction. It is called a counter because it normally increments each time it is used. The condition code register (CC) has bits that are used to indicate results of tests that can be used in conditional branch instructions.

At the beginning of the instruction cycle it is assumed that the program counter contains the address of the first byte of the instruction. As each byte of the instruction is fetched, the PC is incremented by 1, so that the PC always has the address of the next byte of the instruction to be read from memory. When the instruction cycle is completed, the PC then automatically contains the address of the first byte of the next instruction.

We now look at the instruction statically as 1 or more memory bytes or as a line of a program. This discussion will introduce new concepts, but we have tried to keep the number down so that the examples can be discussed without your having too many terms to deal with. The examples will help to clarify the concepts that we introduce below.

Each instruction in a microcomputer carries out an operation. The types of operations provided by a von Neumann computer can be categorized as follows:

1. Move.
2. Arithmetic.
3. Logical.

4. Edit.
5. Control.
6. Input/output.

At this time, we are interested in how instructions are stored in memory as part of a program and how they are executed by the 6812.

After the instruction is fetched, the execute phase of the fetch/execute cycle will often use an address in memory for the input data of the operation or for the location of the result of the operation. This location is called the *effective address*. The 6812, like most microcomputers, is a *one-address computer* because each instruction can specify at most one effective address in memory. For instance, if an instruction were to move a word from location 100 in memory into register A, then 100 is the effective address. This effective address is generally determined by some bits in the instruction. The *addressing mode* specifies how the effective address is to be determined, and there are binary numbers in the instruction that are used to determine the address. The effective address is calculated at the beginning of the execute phase, just after the instruction is fetched and before any of the operations to execute the instruction actually take place.

An instruction in the 6812 is stored in memory as 1 or more bytes. The first, and possibly only, byte of the instruction is generally the operation code byte. The *operation code byte* contains the *operation code* (*opcode*, for short), which specifies the operation to be carried out and the specification of the addressing mode. The remaining bytes of the instruction, if any, specify the effective address according to the given addressing mode. The bytes representing the instruction can be represented as a sequence of ones and zeros, that is, a binary number. The trouble with this is that it is hard to remember and to check an 8-bit or longer string of ones and zeros. To make it easier, we can represent the bit pattern as a *hexadecimal number*. A hexadecimal number will be prefixed by a dollar sign (\$) to distinguish it from a decimal number. (For example, 193 = \$C1. If you are unfamiliar with hexadecimal numbers, see Appendix 1.) When the opcode, addressing modes, and constants used to determine the address are represented either as binary or hexadecimal numbers, we call this representation the *machine code* because it is the actual format used to store the instruction in the *machine* (microcomputer), and this format is used by the machine to determine what is to be done.

Machine code is quite useful for making small changes in a program that is being run and corrected or *debugged*. However, writing even a moderately long program in machine code is a punishment that should be reserved for the fifth level of Dante's inferno. In Chapter 4, we discuss how text produced by an editor is converted by an *assembler* to the machine code stored in the computer's memory. The text input to the assembler is called *source code*. In a line of source code, to make remembering the instructions easier, a three- to five-character *mnemonic* is used to describe the operation, and its addressing information may be given as a hexadecimal or a decimal number. A line

of source code, consisting of mnemonics and addressing information, can be converted by hand into the hexadecimal machine code equivalents using Motorola's *CPU12 Reference Guide* (you can order it from Motorola by using reference number CPU12RG/D). In the first three chapters, we want to avoid using the assembler, so we can see clearly just how the computer ticks. We will hand-convert mnemonics and addressing information to hexadecimal machine code and work with hexadecimal machine code.

We now look at a load immediate instruction in detail, to introduce concepts about instructions in general. The load instruction will move a byte into an accumulator, either A or B. Its simplest addressing mode is called *immediate*. For instance, to put a specific number, say \$2F, in accumulator A, execute the instruction whose source code line is

LDAA #\$2F

where the symbol “#” denotes immediate addressing and the symbol “\$” is used to indicate that the number that follows is in hexadecimal. This instruction is stored in memory as the 2 consecutive bytes:

\$86
\$2F

(Look in *CPU12 Reference Guide*, Instruction Set Summary, for the mnemonic LDAA, and, under it, find \$86 under the Machine Coding column, in the row beginning LDAA #opr8i, which also has the addressing mode IMM for 8-bit immediate addressing.)

Looking dynamically at an instruction, an operation (e.g., add, subtract, load, clear, etc.) may be carried out with inputs (or *operands*) and may produce a result. The instruction is executed in the instruction cycle as follows.

1. Fetch the first byte of the instruction from memory.
2. Increment the PC by one.
3. Decode the opcode that was fetched in step 1.
4. Repeat steps 1 and 2 to fetch all bytes of the instruction.
5. Calculate the effective address to access memory, if needed.
6. Recall the operand from memory, if needed.
7. Execute the instruction, which may include writing the result into memory.

The controller fetches the first byte, \$86. The program counter is incremented. The controller decodes \$86. The controller fetches the second byte, \$2F, putting it into accumulator A. The program counter is incremented. After this instruction is executed, another instruction is fetched and executed.

► 1.3

A Few Instructions and Some Simple Programs

Now that we have examined the instruction from static and dynamic points of view, we will look at some simple programs. The machine code for these programs will be described explicitly so that you can try out these programs on a real 6812 and see how they work or at least so that you can clearly visualize this experience. The art of determining which instructions have to be put where is introduced together with a discussion of the bits in the condition code register. We will discuss what we consider to be a good program versus a bad program, and we will discuss what information is going to be covered in Chapters 2 and 3. We will also introduce an alternative to the immediate addressing mode using the load instruction. Then we bring in the store, add, software interrupt, and add with carry instructions to make the programs more interesting as we explain the notions of programming in general.

We first consider some variations of the load instruction to illustrate different addressing modes and representations of instructions. We may want to put another number into register A. Had we wanted to put \$3E into A rather than \$2F, only the second byte of the instruction would be changed, with \$3E replacing \$2F. The same instruction as

```
LDAA #$3E
```

could also be written using a decimal number as the immediate operand: for example,

```
LDAA #62
```

Either line of source code would be converted to machine code as follows:

\$86 \$3E

We can load register B using a different opcode byte. Had we wanted to put \$2F into accumulator B, the first byte would be changed from \$86 to \$C6 and the instruction mnemonic would be written as

```
LDAB #$2F
```

We now introduce the direct addressing mode. Although the immediate mode is useful for initializing registers when a program is started, the immediate mode would not be able to do much work by itself. We would like to load words that are at a known location but whose actual value is not known at the time the program is written. One could load accumulator B with the contents of memory location \$3840. This is called *direct addressing*, as opposed to immediate addressing. The addressing mode, direct, uses no pound sign “#” and a 2-byte address value as the effective address; it loads the word at this address into the accumulator. The instruction mnemonic for this is

```
LDAA $3840
```

and the instruction appears in memory as the 3 consecutive bytes.

\$B6
\$38
\$40

Notice that the “#” is missing in this mnemonic because we are using direct addressing instead of immediate addressing. Also, the second 2 bytes of the instruction give the address of the operand, with the high byte of the address first.

The store instruction is like the load instruction described earlier except that it works in the reverse manner (and a STAA or STAB with the immediate addressing mode is neither sensible nor available). It moves a word from a register in the MPU to a memory location specified by the effective address. The mnemonic, for store from A, is STAA; the instruction

```
STAA 14378
```

will store the byte in A into location 14378 (decimal). Its machine code is

\$7A
\$38
\$2A

where the number 14378 is stored in hexadecimal as \$382A. With direct addressing, 2 bytes are always used to specify the address even though the first byte may be zero.

Figure 1.3 illustrates the implementation of the load and store instruction in a simplified computer, which has two accumulators and four words of

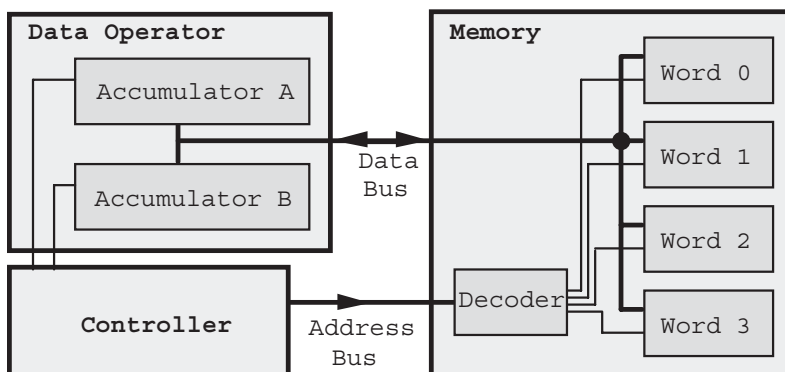


FIGURE 1.3. Registers and Memory

memory. The data operator has accumulators A and B, and a memory has four words and a decoder. For the instruction `LDAA 2`, which loads accumulator A with the contents of memory word 2, the controller sends the address 2 on the address bus to the memory decoder; this decoder enables word 2 to be read onto the data bus, and the controller asserts a control signal to accumulator A to store the data on the data bus. The data in word 2 are not lost or destroyed. For the instruction `STAB 1`, which stores accumulator B into the memory word 1, the controller asserts a control signal to accumulator B to output its data on the data bus, and the controller sends the address 1 on the address bus to the memory decoder; this decoder enables word 1 to store the data on the data bus. The data in accumulator B are not lost or destroyed.

The `ADD` instruction is used to add a number from memory to the number in an accumulator, putting the result into the same accumulator. To add the contents of memory location `$0BAC` to accumulator A, the instruction

`ADDA $3BAC`

appears in memory as

<code>\$BB</code>
<code>\$3B</code>
<code>\$AC</code>

The addition of a number on the bus to an accumulator such as accumulator A is illustrated by a simplified computer having a data bus and an accumulator (Figure 1.4). The data operator performs arithmetic operations using an adder (see Figure 1.4). Each 1-bit adder, shown as a square in Figure 1.4b, implements the truth table shown in Figure 1.4a. Registers A, B, and S may be any of the registers shown in Figure 1.2 or instead may be data from a bus. The two words to be added are put in registers A and B, Cin is 0, and the adder computes the sum, which is stored in register S. Figure 1.4c shows the symbol for an adder. Figure 1.4d illustrates the addition of a memory word to accumulator A. The word from accumulator A is input to the adder while the word on the data bus is fed into the other input. The adder's output is written into accumulator A.

When executing a program, we need an instruction to end it. When using the true-time debugger with state-of-the-art hardware, background (mnemonic: `BGND`) halts the microcontroller, when using the debugger `DEBUG_12` or `MonitorHCS12` with less-expensive hardware, software interrupt (mnemonic: `SWI`) serves as a halt instruction. In either case, the `BRA` instruction discussed in the next chapter can be used to stop. When you see the instruction `BRA *`, which means “branch to yourself,” in a program, think “halt the program.” This last instruction will be explained in Chapter 2.

Figure 1.5 shows four instructions in locations `$380D` through `$3817`, to be executed in the simulator only (see Appendix 3); it adds two numbers in

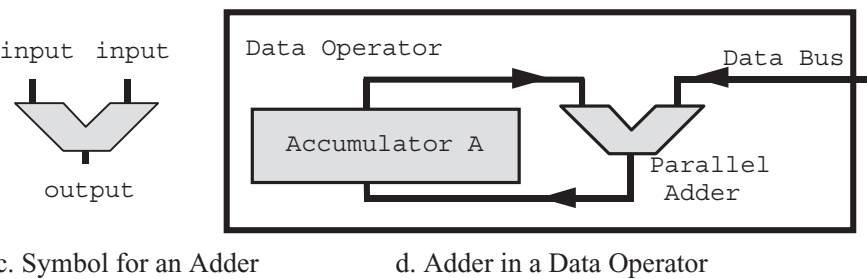
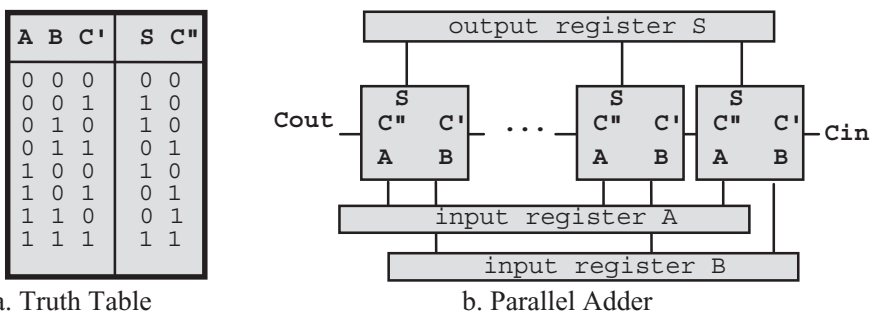


FIGURE 1.4. Data Operator Arithmetic

locations \$3840 and \$3841, putting the result into location \$3842, and stops with a “branch-to-yourself.”

We will now look at condition code bits in general and the carry bit in particular. The carry bit is really rather obvious. When you add by hand, you may write the carry bits above the bits that you are adding so that you will remember to add it to the next bit. When the microcomputer has to add more than 8 bits, it has to save the carry output from one byte to the next, which is Cout in Figure 1.4b, just as you remembered the carry bits in adding by hand. This carry bit is stored in bit C of the condition code register shown in Figure 1.6. The microcomputer can input this bit into the next addition as Cin in Figure 1.4b. For example, when adding the 2-byte numbers \$349E

380D	B6	LDAA \$3840	; get 1st operand
380E	38		
380F	40		
3810	BB	ADDA \$3841	; add 2nd operand
3811	38		
3812	41		
3813	7A	STAA \$3842	; store sum
3814	38		
3815	42		
3816	20	BRA *	; stop
3817	FE		

FIGURE 1.5. Program for 8-Bit Addition

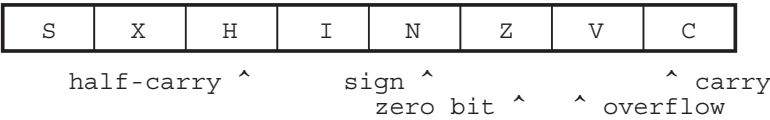


FIGURE 1.6. Bits in the Condition Code Register

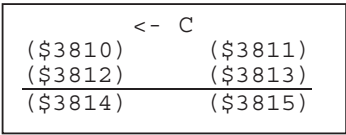


FIGURE 1.7. Addition of 2-Byte Numbers

and \$2570, we can add \$9E and \$70 to get \$0E, the low byte of the result, and then add \$34, \$25, and the carry bit to get \$5A, the high byte of the result (Figure 1.7). In this figure, C is the carry bit obtained from adding the contents of locations \$3811 and \$3813; (m) is used to denote the contents of memory location m, where m may be \$3811, etc. The *carry bit* (or *carry* for short) in the *condition code register* (Figure 1.6) is used in exactly this way.

The carry bit is also an error indicator after the addition of the most significant bytes is completed. As such, it may be tested by conditional branch instructions, to be introduced later. Other characteristics of the result are similarly saved in the controller’s condition code register. These are, in addition to the carry bit C, N (negative or sign bit), Z (zero bit), V (two’s-complement overflow bit or signed overflow bit), and H (*half-carry* bit) (see Figure 1.6). How 6812 instructions affect each of these bits is shown in the *CPU12 Reference Guide*, Instruction Set Summary, in the rightmost columns.

We now look at a simple example that uses the carry bit C. Figures 1.8 and 1.9 list two equally good programs to show that there is no way to have exactly one correct answer to a programming problem. After the example, we consider some ways to know if one program is better than another. Suppose that we want to add the two 16-bit numbers in locations \$3810, \$3811 and \$3812, \$3813, putting the sum in locations \$3814, \$3815. For all numbers, the higher-order byte will be in the smaller-numbered location. One possibility for doing this is the instruction sequence in Figure 1.8 that can

3820	F6 08 11	LDAB \$3811	; get low byte of 1st
3823	B6 38 10	LDAA \$3810	; get high byte of 1st
3826	FB 38 13	ADDB \$3813	; add low byte of 2nd
3829	B9 38 12	ADCA \$3812	; add high byte of 2nd
382C	7B 38 15	STAB \$3815	; store low sum byte
382F	7A 38 14	STAA \$3814	; store high sum byte
3832	20 FE	BRA *	; halt

FIGURE 1.8. Program for 16-Bit Addition

3820	B6 38 11	LDAA	\$3811 ; get low byte of 1st
3823	BB 38 13	ADDA	\$3813 ; add low byte of 2nd
3826	7A 38 15	STAA	\$3815 ; store low sum byte
3829	B6 38 10	LDAA	\$3810 ; get high byte of 1st
382C	B9 38 12	ADCA	\$3812 ; add high byte of 2nd
382F	7A 38 14	STAA	\$3814 ; store high sum byte
3832	20 FE	BRA *	; halt

FIGURE 1.9. Alternative Program for 16-Bit Addition

be executed in the simulator, where, for compactness, we give only the memory location of the first byte of the instruction.

In the program segment in Figure 1.8, which runs on the simulator, the instruction `ADCA $3812` adds the contents of A with the contents of location \$3812 and the C condition code bit, putting the result in A. At that point in the sequence, this instruction adds the two higher-order bytes of the two numbers together with the carry generated from adding the two lower-order bytes previously. This is, of course, exactly what we would do, as seen in Figure 1.7. Note that we can put this sequence in any 19 consecutive bytes of memory as long as the 19 bytes do not overlap with data locations \$3810 through \$3815.

We could also have used just one accumulator with the instruction sequence shown in Figure 1.9, which executes in the simulator. In this new sequence, the load and store instructions do not affect the carry bit C. (See the *CPU12 Reference Guide*, Instruction Set Summary. We will understand why instructions do not affect C as we look at more examples.) Thus, when the instruction `ADCA $3812` is performed, C has been determined by the `ADDB $3813` instruction.

The two programs in Figures 1.8 and 1.9 were equally acceptable. However, we want to discuss guidelines to writing good programs early in the book, so that you can be aware of them to know what we are expecting for answers to problems and so that you can develop a good programming style. A good program is shorter and faster and is generally clearer than a bad program that solves the same problem. Unfortunately, the fastest program is almost never the shortest or the clearest. The measure of a program has to be made on one of the qualities or on one of the qualities based on reasonable limits on the other qualities, according to the application. Also, the quality of clarity is difficult to measure but is often the most important quality of a good program. Nevertheless, we discuss the shortness, speed, and clarity of programs to help you develop good programming style.

The number of bytes in a program (its length) and its execution time are something we can measure. A short program is desired in applications for which program size affects the cost of the end product. Consider two manufacturers of computer games. These products feature high sales volume and low cost, of which the microcomputer and its memory are a significant part.

If one company uses the shorter program, its product may need fewer ROMs to store the program, may be substantially cheaper, and so may sell in larger volume. A good program in this environment is a short program. Among all programs doing a specific computation will be one that is the shortest. The quality of one of these programs is the ratio of the number of bytes of the shortest program to the number of bytes in the particular program. Although we never compute this *static efficiency* of a program, we will say that one program is more statically efficient than another to emphasize that it takes fewer bytes than the other program.

The *CPU12 Reference Guide*, Instruction Set Summary, gives the length of each instruction by showing its format. For instance, the LDAA # $\$2F$ instruction is shown alphabetically under LDAA in the line IMM. The pattern 86 ii, means that the opcode is $\$86$ and there is a 1-byte immediate operand ii, so the instruction is 2 bytes long.

The speed or execution time of a program is prized in applications for which the microcomputer has to keep up with a fast environment, such as in some communication switching systems, or for which income is related to how much computing can be done. A faster computer can do more computing and thus make more money. However, speed is often overemphasized: “My computer is faster than your computer.” To show you that this may be irrelevant, we like to tell this story about a computer manufacturer. This is a true story, but we will not use the manufacturer’s real name for obvious reasons. How do you make a faster version of a computer that executes the same instruction? The proper answer is that you run a lot of programs and find instructions that are used most often. Then you find ways to speed up the execution of those often-used instructions. Our company did just that. It found one instruction that was used very, very often and then found a way to really speed up that instruction. The machine should have been quite a bit faster, but it was not! The most common instruction was used in a routine that waited for input-output devices to finish their work. The new machine just waited faster than the old machine that it was to replace. The moral of the story is that many computers spend a lot of time waiting for input-output work to be done. A faster computer will just wait more. Sometimes speed is not as much a measure of quality as it is thought to be, but then in other environments, it is the most realistic measure of a program. As we shall see in later chapters, the speed of a particular program can depend on the input data to the program. Among all the programs doing the same computation with specific input data, there will be a program that takes the fewest number of clock cycles. The ratio of this number of clock cycles to the number of clock cycles in any other program doing the same computation with the same input data is called the *dynamic efficiency* of that program. Notice that dynamic efficiency does depend on the input data but not on the clock rate of the microprocessor. Although we never calculate dynamic efficiency explicitly, we do say that one program is more dynamically efficient than another to indicate that the first program performs the same computation more quickly than the other one over some range of input data.

3820	FC 38 10	LDD \$3810	; get 1st 16-bit word
3823	F3 38 12	ADDD \$3812	; add 2nd 16-bit word
3826	7C 38 14	STD \$3814	; store 16-bit word
3832	20 FE	BRA *	; halt

FIGURE 1.10. Most Efficient Program for 16-Bit Addition

The *CPU12 Reference Guide*, Instruction Set Summary, gives the instruction timing. For instance, the LDAA # $\$2F$ instruction is shown alphabetically under LDAA for the mode IMM. The Access Detail column indicates that this instruction takes one memory cycle of type p, which is a program word fetch. Generally, a memory cycle is $\frac{1}{24}$ μ s.

The *clarity* of a program is hard to evaluate but has the greatest significance in large programs that have to be written by many programmers and that have to be corrected and maintained for a long period. Clarity is improved if you use good documentation techniques, such as comments on each instruction that explain what you want it to do, and flowcharts and precise definitions of the inputs, outputs, and the state of each program, as explained in texts on software engineering. Some of these issues are discussed in Chapter 5. Clarity is also improved if you know the instruction set thoroughly and use the correct instruction, as developed in the next two chapters.

Although there are often two or more equally good programs, the instruction set may provide significantly better ways to execute the same operation, as illustrated by Figure 1.10. The 6812 has an instruction LDD to load accumulator D, an instruction AD DD to add to accumulator D, and an instruction STD to store accumulator D, which for accumulator D are analogous to the instructions LDAA, ADDA, and STAA for accumulator A. The program in Figure 1.10 performs the same operations as the programs given earlier but is much more dynamically and statically efficient and is clearer.

If you wish to write programs in assembly language, full knowledge of the computer's instruction set is needed to write the most efficient, or the clearest, program. The normal way to introduce an instruction set is to discuss operations first and then addressing modes. We will devote Chapter 2 to the discussion of instructions and Chapter 3 to the survey of addressing modes.

In summary, you should aim to write good programs. As we saw with the examples, there are equally good programs, and generally there are no best programs. Short, fast, clear programs are better than the opposite kind. However, the shortest program is rarely the fastest or the clearest. The decision as to which quality to optimize depends on the application. Whichever quality you choose, you should have as a goal the writing of clear, efficient programs. You should avoid the tendency to write sloppy programs that just barely work or that work for one combination of inputs but fail for others. Therefore, we will arbitrarily pick one of these qualities to optimize in the

problems at the end of the chapters. We want you to optimize static efficiency in your solutions, unless we state otherwise in the problem. Learning to work toward a goal should help you write better programs for any application when you train yourself to try to understand what goal you are working toward.

► 1.4

6812 Microcontroller Organizations

The 6812 is currently available in about two dozen implementations. Three of these are designated the MC68HC812A4 (abbreviated the 'A4), the MC68HC912B32 (abbreviated the 'B32), and the MC9S12C32 (abbreviated the 'C32). These are discussed herein.

The 'A4 can operate in the single-chip mode or the expanded bus mode. In the *single-chip mode*, the 'A4 can be the only chip in a system, for it is self-sufficient. The processor, memory, controller, and I/O are all in the chip (Figure 1.11). The memory consists of 1K words of RAM and 4K words of *electrically erasable programmable memory (EEPROM)*. The I/O devices include a dozen parallel I/O ports, a *serial peripheral interface (SPI)*, a *serial communication interface (SCI)*, a *timer*, and an *A/D converter*.

The 'A4's expanded bus mode removes three or four of the parallel ports, using their pins to send the address and data buses to other chips. RAM, ROM, *erasable programmable read-only memory (EPROM)*, and *programmable read-only memory (PROM)* can be added to an expanded bus. In a narrow expanded mode, ports A and B are removed for address lines, and port C is an 8-bit data bus. Port D is available for parallel I/O. In a wide expanded mode (Figure 1.12), port D is also unavailable for parallel I/O. In both modes, ports E, F, and G are available for bus control, chip selects, and memory control, or else for parallel I/O. Each parallel port, A, B, etc., is

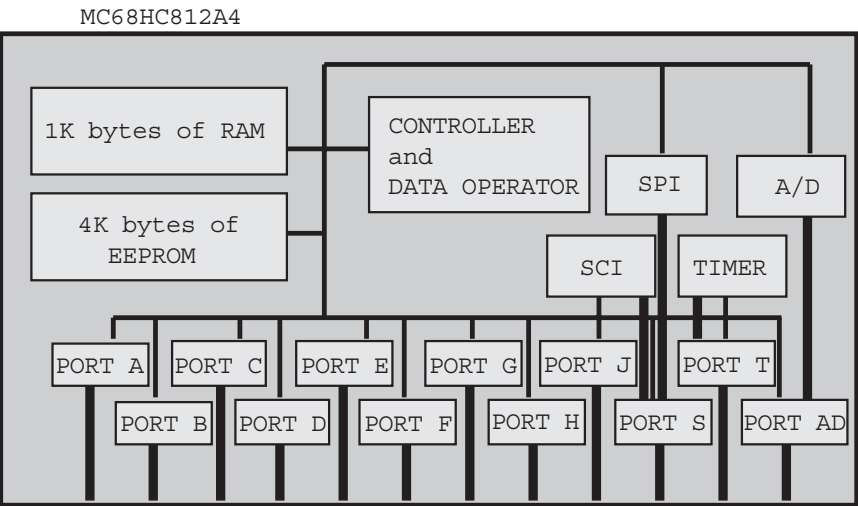


FIGURE 1.11. Single-Chip Mode of the MC68HC812A4