Designing with Microprocessors

Lawrence E. Getgen

Designing with Microprocessors

Lawrence E. Getgen
California State University, Chico

SCIENCE RESEARCH ASSOCIATES, INC.
Chicago, Henley-on-Thames, Sidney, Toronto
A Subsidiary of IBM

Acquisition Editor	Alan W. Lowe
Project Editors	Richard Myers
	Dan Kirklin
Compositor	The Clarinda Company
Text Designer	Allen Carr
Illustrator	Allen Carr and House of Graphics
Cover Designer	Allen Carr

Library of Congress Cataloging in Publication Data

Getgen, Lawrence E., 1924-
Designing with microprocessors.

Includes index.
1. Microprocessors. 2. Microprocessors—Programming.
I. Title.
TK7895.M5G48 1985 001.64′2 85-2033
ISBN 0-574-21600-6

ISBN 0-574-21600-6
Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

This text on microprocessor programming, hardware, and applications is aimed at engineering and computer-science students who will ultimately apply this material in the industrial environment. Both students who have had an introduction to computer programming and the more naive industrial development groups are likely to find the techniques included here different from what they expect.

There is a strong—almost overwhelming—belief among engineers that much program documentation and related documentation standards such as those given in this text are unnecessary. The individual may bypass these steps from laziness or indifference. The industrial concern does the same, because software costs money, and it believes that eliminating this unnecessary paperwork will save a considerable sum.

The usual sequence of events for both individual programmers and industrial concerns is to try the "easy way" first. This is "scene one." They provide documentation consisting, in total, of an Assembly listing. There is no specification for the program and no program description; the software design is nil; GO TOs are rampant; modules are nonexistent; and a program library is unheard of.

"Scene two" begins when bugs are found in the program, the user wants the operating features to be modified, and the original programmers have moved to higher paying positions. At this stage, the person stuck with the program maintenance becomes a convert to proper documentation and programming techniques or chooses a career change. As the programming expenses mount (unexpectedly, since program maintenance was not thought of as part of the project budget), management is deciding whether or not to continue the project.

"Scene three" is enacted only by the mature individual or company. At this stage, programming methods are established and documentation is standardized. A program library is established. Change procedures are mandated. The system to be developed as a specification, approved honestly and competently by all parties concerned; the program has a description both at the top and at the module level; the modules have prologues; often even standard hardware is used.

Experience shows that it is extremely difficult to begin scene three. It is almost always necessary to burn one's fingers before acknowledging that a fire exists. A number of factors influence this situation, and it is worthwhile to comment on some of them.

Until the advent of the microprocessor, engineers used the computer principally for problem solving. In this case, many of the following apply:

1. The program is often relatively small and is dedicated to a single task or to solving a single problem.
2. The program is used by only one person, the program writer.
3. At the time of writing, the program writer anticipates that the program will be used only once. Since no future use is intended, the programmer does not write support documentation.
4. No formal test program is written. The program is informally tested on the problem that it is to solve.
5. The program exhibits a lack of structure.
6. Program documentation, if it exists, consists of a text user's guide and a 1- to 3-page flowchart.

Contrast these characteristics with the usual microprocessor application:

1. A small program, such as that for a communication-system controller, might easily be 12,000 lines long. A telephone central office switch may use 15 to 20 16-bit microprocessors, with a total of 200,000 lines of high-level code. The applications will almost always be multitask and real-time, and certainly maintenance and options will be involved.
2. Many people may be involved in the writing of the program.
3. Access to the program data base may be required by the user in order to enter system-configuration information.
4. Maintenance and options are a certainty. These may be as extensive as the original programming effort.
5. The lifetime of commercial equipment should exceed five years, and the lifetime of telecommunication equipment is twenty years. Although the program for given equipment should stabilize in a few years, it may still require updating and is far from having a one-shot usage.
6. The program is tested module by module and then is string tested (i.e., strings of modules are tested), and the system is tested for conformance with the system specification.
7. The program is written in a structured format.
8. The program documentation is extensive, containing system specifications, program descriptions, structure charts, module descriptions, test programs, and program code.

The extent to which design, documentation, testing, and maintenance enter into a program can be seen by the following statistics.[1]

Industrial programming teams report that the programming rate for a fully documented and tested program product is 1000 statements per year

[1]F. P. Brooks, *The Mythical Man-Month* (Reading, Massachusetts: Addison-Wesley, 1975), p. 4–6.

per programmer. This is less than 20 statements per week, or 4 statements per day!

We all know that we can code faster than 4 statements per day! How is the rest of our time spent? First, the rate of 1000 statements per year applies to significantly large programs. Compared to one-programmer, stand-alone programs, a large program requires a great deal of communication. *Communication* is used here in a broad sense. It includes not only communication between people, e.g., programmer to programmer, programmers to management, programmers to customer, but communication between program modules as well. It also includes standardization that will enable the program to fit into a family of program products. The communication problems increase significantly with program size.

As the requirement for compatibility within the family of program products is dropped, the programming rate is increased threefold. Such a program can be tested, repaired, and maintained by anybody and is usable in many operating environments for many sets of data, but it is not a systems-wide product. As the environment is increasingly simplified to the point where the program is a stand-alone and as testing, documentation, and maintenance requirements are decreased, the rate again increases at least threefold.

Another unexpected figure concerns program maintenance. Maintenance is the work spent on a program after its original release, including corrections to the original program and the addition of features that make it truly operable. Surprisingly, the maintenance effort on a program is likely to be as great as the effort on the original, prerelease program. In view of this figure, it is startling that many novice programmers (and novice companies, for that matter) make no budget allowance whatsoever for program maintenance. Some rude awakenings await the programming world. Since the magnitude of maintenance effort depends on documentation quality much more than the original programming effort does, maintenance alone is a strong justification for suitable software design, proper documentation, and established administrative methods.

Against this background, consider the problem as it is addressed by this text. First, this text pertains mainly to the single-programmer environment. The sequel will be aimed at the multiprogrammer environment. Before addressing the multiprogrammer situation, we encourage individual programmers to design and document their programs properly. At every step in the programming process, it is necessary to repeat that the programmer is not writing the program just for himself or herself, but for others—in particular, for the maintenance programmer.

Use of This Text

The notes for this text have been used for the last two years for a one-semester microprocessor-applications course for electrical engineering students. The prerequisite was a course in digital logic design. Some students had taken a basic Assembly language programming course; others had not. The text, however, was developed for use in a computer engineering program, now being introduced. The computer engineering students study, almost equally, in the electrical engineering and computer science areas. These students have both Assembly language and digital logic design as prerequisites. The text has been readily usable in these various situations. However, those who have been exposed to Assembly language are usually able to cover the text material in one semester, whereas others are likely to cover only the first ten chapters.

The emphasis to be placed on the various microprocessors will be governed by the laboratory facilities. The microprocessors supported in the lab will require more emphasis than the others. Many instructors will feel that to discuss more than one microprocessor only confuses the student. If each is discussed in detail, this is likely to be true. However, a presentation at the architectural level allows the student to gain a lot from noting the similarities that exist in the various microprocessor units. Instructors should also point out the features provided by each device because of its difference from the other architectures.

The 16-bit microprocessors have not been included in this text. The primary reason for this is that the course using this text is heavily project-oriented. It seems unwarranted to require student expenditures for 16-bit hardware when an 8-bit machine is suitable for project applications.

Peripheral components are presented not with the objective of showing operation of all devices of a given type (PIOs, for example) or of presenting design-level detail. The object is to survey available functions. The presentation is application-oriented. Design details are left to the manufacturers' manuals, which are to be used in support of the text. These manuals are the manufacturers' product specifications and are therefore the ultimate design authority for any device.

Chapters 1 through 6 are hardware-oriented. The presentation covers the Zilog Z80,[2] the Intel 8085, and the Motorola 6800 family. These microprocessors are all "good" machines: they are all relatively conventional in their configuration and are easy to understand. Yet their architectural differences are of interest, too. These are among the devices that the reader is likely to encounter. As mentioned, it is intended that these chapters be supported by the pertinent manufacturers' user's manuals. The user's manuals provide thorough and excellent technical specifications, timing diagrams and timing requirements, and design detail. These are the documents the student should learn to use for design. It would

[2]Zilog and Z80 are trademarks of Zilog, Inc., with whom Science Research Associates, Inc. is not associated.

be both superfluous and presumptuous to attempt to repeat the material in these excellent documents here. The object is to support the user's manuals by providing application detail and practical usage information.

Chapter 5 introduces general-purpose peripheral devices. The first of these is the nonprogrammable parallel I/O port. This is followed by some practical considerations in the use of these devices. Bus buffers and bus terminations, as well as pull-up and pull-down resistors, are discussed. Also in the list of general-purpose peripherals are programmable PIOs, the keyboard-display controller, the counter-timer unit, and the DMA controller.

Chapter 6 presents communication interfacing. This includes a thorough discussion of the RS-232C asynchronous communication interface and its application to microprocessor systems. Synchronous and asynchronous data transmission are defined. The Centronics parallel interface is introduced, and some comments are made concerning printer interfacing. The 20 ma current loop interface and some of its implementations are described.

Chapter 7 considers hardware from the system viewpoint. It describes some typical system circuit boards and some often-used bus structures.

Chapter 8 presents the subset of Assembly instructions that are common to both the Zilog Z80 and the Intel 8085. These instructions are given in both the Z80 and the 8085 formats. An appendix provides more detailed instruction sets for each microprocessor: the Z80, 8085, and 6800; Chapter 8 serves merely as a starting place to introduce Assembly programming. Mention of Assembly instructions is not to imply that high-level, structured languages are to be completely bypassed. All programming in the following chapters is preceded by both text and pseudo-code descriptions. The pseudo-code structures are sufficiently like PLM or Pascal structures to allow the pseudo-code to be replaced by these languages in most instances.

Chapter 9 presents bottom-up programming—the building and use of tools. This is not only a necessary consideration but also a means by which the student can do useful programming early in the text, before more advanced concepts have been presented. Chapter 9 presents the program-module specification, the module description or prologue, the pseudo-code description, and the hierarchy chart. At this stage, the module test program is introduced. Its use is emphasized in the laboratory projects.

Subroutines are introduced in Chapter 10. The concept of subroutines is presented in their historical evolution, as far as microprocessor use is concerned, from the original concept of a module implemented only when needed repeatedly to the modern concept of the structure block called by the mainline program. The positional relationship of subroutine parameters is discussed, as are local and global variables. Passage of parameters by value, by reference, and by value-result is covered. Implementation outlines are presented for each of these cases. The function-type subroutine is also included.

Chapter 11 presents top-down structured programming. Program decomposition is introduced, along with an outline of the various types of coupling that can exist between program modules as the program is decomposed according to different premises. Attention is shifted from the hierarchy chart of Chapter 9 to the structure chart. Now the communication between modules becomes evident, and the concept of intermodule coupling is reinforced. Once the whole program or subprogram has been viewed, the concept of module string testing is introduced. Related to this is also the introduction of test drivers and test stubs.

Chapter 12 introduces data lists. Linear lists are described, as are forward, backward, and doubly linked lists. Practical applications are given, and the material presented in the earlier chapters is reinforced.

Chapter 13 adds some thoughts on subroutines that were not required at the first presentation, in Chapter 10. The use of list handlers is shown. The first list handlers are implemented as subroutines. Subsequently, macros are presented, and it is shown how the list handlers can be implemented, using macros.

Chapter 14 outlines the use of a microprocessor development system. As with earlier chapters, the intent of this chapter is not to duplicate the tremendous amount of detail given in the manufacturers' manuals but to provide the user with an overall picture of the development system and of the tasks which it can perform. As a rule, it is not the manufacturers' intent to provide a general, overall picture. It is presumed that the user has by some means obtained this information. This chapter is of value to those who must learn a system on their own. For those who are learning in a classroom environment, this chapter will be reinforced by the laboratory assignments. Although this is the last chapter in the text, it is not intended that it be the last topic in the course. It should be presented when it fits best with the laboratory activities. Since use of a development system can not be covered in one lab session, it may be advantageous to present the material of this chapter upon several occasions, as the topics are introduced in the laboratory.

Zilog, Z80, Zilog Z80 and material attributable to Zilog, Inc. are used with permission. Neither the author nor SRA is associated with Zilog, Inc. in any way, and no implication to the contrary is intended. Similarly, material relating to Intel Corporation and to Motorola, Inc. is included with permission. It should be mentioned that the devices described herein often do not represent the latest products of these companies. The choice has been made for educational reasons.

This text is dedicated with sincere appreciation to the companies listed above. It is also dedicated to Hewlett-Packard, Inc. and to Tektronix, Inc., which have generously contributed to the implementation of the California State University Chico computer engineering laboratory and have provided every encouragement in the development of the computer engineering program. Two of my peers at CSUC deserve special thanks as respected professionals, as friends, and as industrious co-workers on the computer engineering project. These are Dr. Larry Wear and Dr. Bill Lane. We have jointly pursued the same goal, from different vantage points, for several years. It speaks well that our respect for one another has continually increased as we have shared this effort.

The following persons read the manuscript and provided many helpful suggestions: Porter Sherman, University of Bridgeport; Michael Andrews, Space Tech Corporation; and Dr. Alan D. Wilcox, P. E., Bucknell University. Roy W. Goody of Mission College, Santa Clara, offered useful technical comments in the final editorial stages. The contributions of all these individuals are acknowledged with gratitude.

Contents

1

General Description of a Microprocessor System

1.1 Definitions

Before commencing with the general description of a microprocessor system, it is necessary to introduce some definitions. These serve to establish the usage and generally are true. However, the reader should be aware that some terms are perhaps machine or author dependent. These can usually be distinguished by the context or by the definition given in the machine documentation.

1.1.1 Binary

Binary means capable of having either one of two possible states. Examples are

1. A binary number system consists only of the symbols 0 and 1.
2. A binary switch operation might be depicted by OFF, ON; UP, DOWN; or RIGHT, LEFT.
3. Binary decisions or descriptions can be made concerning groups. For a set of numbers, the following are binary decisions:
   a. ODD, EVEN
   b. X .gt. N (i.e., X greater than N)
   c. X .1t. N (i.e., X less than N)

1.1.2 Bit

A *bit* (short for *bi*nary digi*t*) represents one binary choice. With respect to microprocessors, each bus lead in the system can have one of two possible voltage levels at any instant. These two voltage levels are designated *logic-0* and *logic-1*. A bus lead or other point in a system depicting these levels represents one bit of information. Note that the actual voltages are not 0 volts and 1 volt but are voltages in one of two possible voltage ranges representing a binary choice.

In logic-circuit design, it is common practice to represent a logic signal that is in the *true* state by a mnemonic such as *OPEN*. The complement of this signal is usually written as $\overline{OPEN}$. This practice is sometimes followed in microprocessor design, but it is probably more common to indicate the complement as *OPEN/*. This allows the complement to be more readily expressed in printed material.

A further variation is common in microprocessor usage. For a bus, which can have either of two states, *logic-1* or *logic-0*, assume that the mnemonic assigned to the logic-1 state is *OPEN*. Also assume that, rather than using the mnemonic *OPEN/* for the logic-0 state, we wish to use some variation of the name *CLOSED*. In such a case, *CLOSED/* is used. CLOSED/ is, in practice, called an *active-low* signal. A frequently encountered example is the use of *MR* (memory read) for the logic-1 state of the memory read-write control line and *MW/* for its logic-0 state.

1.1.3 Byte

Because microprocessors operate sequentially, performing one operation at a time, there is a speed advantage in handling a number of binary signals simultaneously rather than one bit at a time.

These sets of binary signals are given names relating to their *width*, i.e., the number of bits in the set. The most common and also the most standard such grouping, other than the bit, is the *byte*. A byte is an 8-bit-wide signal. A byte-wide register is shown in Figure 1-1.

1.1.4 Nibble

A *nibble*, represented in Figure 1-1, is a 4-bit-wide signal. A byte consists of two nibbles. The lower nibble represents the least significant four bits of the byte, and the upper nibble represents the most significant four.

1.1.5 Word

Word is less well defined than *bit*, *byte*, or *nibble*. The term may be used in a general sense to mean a set of binary signals. For instance, one might speak of adding WORD1 to WORD2. This usage allows one to describe a process generally, without restriction to a particular data width.

Word may also be used to represent some standard or frequently used data width associated with a particular computer or microprocessor. For

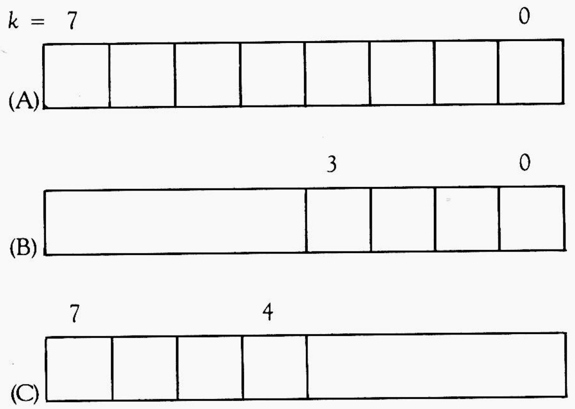

Figure 1-1
(A) Representation of a byte-wide register. $k = 0$ represents the *least significant* bit position, and $k = 7$ represents the *most significant* bit position.
(B) Location of least significant or low nibble in register.
(C) Location of most significant or high nibble in register.