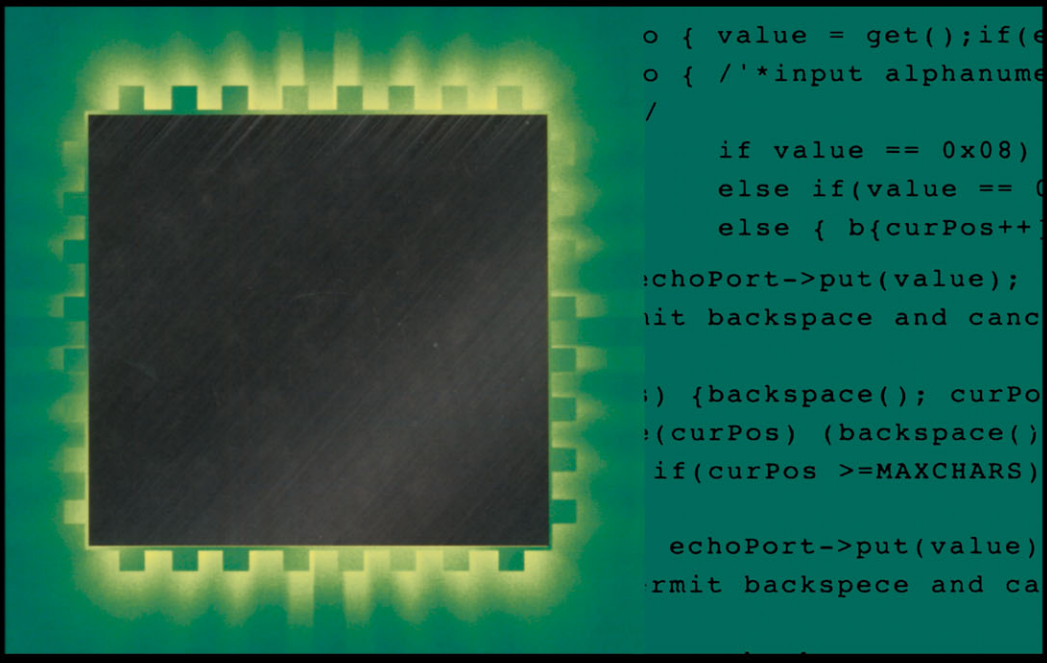
ACADEMIC PRESS SERIES IN ENGINEERING

Embedded Microcontroller Interfacing

for M·CORE Systems



G. Jack Lipovski

FREE SOFTWARE INCLUDED ON CD-ROM

Embedded Microcontroller Interfacing for M·CORE Systems

Academic Press Series in Engineering

Series Editor
J. David Irwin
Auburn University

Designed to bring together interdependent topics in electrical engineering, mechanical engineering, computer engineering, and manufacturing, the Academic Press Series in Engineering provides state-of-the-art handbooks, textbooks, and professional reference books for researchers, students, and engineers. This series provides readers with a comprehensive group of books essential for success in modern industry. A particular emphasis is given to the applications of cutting-edge research. Engineers, researchers, and students alike will find the Academic Press Series in Engineering to be an indispensable part of their design toolkit.

Published books in the series:
Industrial Controls and Manufacturing, 1999, E. Kamen
DSP Integrated Circuits, 1999, L. Wanhammar
Time Domain Electromagnetics, 1999, S. M. Rao
Single- and Multi-Chip Microcontroller Interfacing for the Motorola 68HC12, 1999, G. J. Lipovski
Control in Robotics and Automation, 1999, B. K. Ghosh, N. Xi, T. J. Tarn
Soft Computing and Intelligent Systems, 1999, N. K. Sinha, M. M. Gupta
Introduction to Microcontrollers, 1999, G. J. Lipovski
Control of Induction Motors, 2000, A. M. Trzynadlowski
Embedded Microcontroller Interfacing for M·CORE Systems, 2000, G. J. Lipovski

Embedded Microcontroller Interfacing for M·CORE Systems

G. Jack Lipovski
Department of Electrical and Computer Engineering
University of Texas
Austin, Texas

ACADEMIC PRESS
A Harcourt Science and Technology Company

San Diego San Francisco New York Boston
London Sydney Tokyo

ACADEMIC PRESS
A Harcourt Science and Technology Company
525 B Street, Suite 1900, San Diego, CA 92101-4495, USA
http://www.academicpress.com

Academic Press
Harcourt Place, 32 Jamestown Road, London NW1 7BY, UK
http://www.academicpress.com

Library of Congress Catalog Card Number: 00-102018
ISBN: 0-12-451832-X

Printed in the United States of America

00 01 02 03 04 05 IP 9 8 7 6 5 4 3 2 1

Dedicated to my wife
Isabelle Lipovski

This Page Intentionally Left Blank

Contents

Preface

The embedded microcontroller industry is moving towards inexpensive microcontrollers with significant amounts of ROM and RAM, and some user-designed hardware that is put on a single microcontroller chip. In these microcontrollers, the majority of the design cost is incurred in the writing of software that will be used in them. The memory available in such microcontrollers permits the use of real-time operating systems. Further, C++ compilers permit the use of classes to encapsulate the function members, their data members, and their hardware, in an object. Both of these techniques reduce software design cost. This book aims to give the principles of and concrete examples of design, especially software design, of the Motorola MMC2001, a particular M·CORE embedded microcontroller.

The first four chapters of the book provide background. The first chapter is aimed at the high-level programmer who will need to acquire a reading knowledge of assembler language to be able to debug his or her high-level language programs. The second chapter is aimed at the hardware designer, who will need to know enough C and C++ programming to be able to write the programs in an embedded microcontroller. The third chapter introduces the real-time operating system, including the use of device drivers. The fourth chapter provides information for programmers who need to understand the issues involved in hardware design, including the design of ASIC modules that are implemented in an M·CORE chip. While many readers will be familiar with one or more of these topics, the designer of embedded microcontrollers needs to be familiar with all of them. These chapters bring the reader to an adequate level of background needed for embedded microcontroller design.

The next three chapters are the core of this book. The fifth chapter discusses the alternatives to the parallel port, and ways to program interfaces to control them. The sixth chapter describes alternatives to interrupts, and ways to program interrupt and other synchronization interfaces. The seventh chapter highlights the techniques for and problems with time slice operation of embedded microcontrollers. A simple multithreaded time sharing system is introduced, followed by an object-oriented time sharing system. The use of real-time operating systems multitasking is then discussed.

Chapter 8 shows how to design additional hardware to be added into the MMC2001 chip. It gives an ASIC design example, and describes a processor architecture that is suitable for special-purpose designs. The last two chapters provide some examples of system design. Chapter 9 discusses communication techniques and shows several programming approaches to the MMC2001 UART device. The tenth chapter shows the programming of display and storage systems.

This book provides a concrete understanding of hardware-software tradeoffs, high-level languages, and embedded microcontroller operating systems. Because these very practical areas should be understood by many if not all computer engineering graduate students, this book is written as a textbook for a graduate level course. However, it will also be very useful to practitioners, especially those who will work with the Motorola M·CORE embedded microcontroller. It is therefore also written for engineers who need to understand and use these microcontrollers.

List of Figures

List of Tables

Acknowledgments

The author would like to express his deepest gratitude to everyone who contributed to the development of this book. Special thanks are due to Jim Thomas, who initiated the development of this book, and Greg Watkins, who coordinated its development with M·CORE personnel. I also acknowledge extensive and helpful proofreading from several of these personnel, especially Steve Sobel, Kirby Kyle, and Howard Owens at Motorola, and Phil Walsh and Alan Anderson at Microware.

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 data-base computer, CASSM, and the parallel computer, TRAC. His expertise in microcomputers has brought international recognition—he has served as a director of Euromicro and an editor of IEEE Micro. Dr. Lipovski has published more than 70 papers, largely in the proceedings of the International Symposium on Computer Architecture (ISCA), the IEEE Transactions on Computers and the National Computer Conference. At the 25th ISCA, Dr. Lipovski was noted as having written more papers at this prestigious symposium than any other author. He holds 12 patents, generally in the design of logic-in-memory integrated circuits for database and graphics geometry processing. He has authored nine 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. degree 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 Linden Technology Ltd., and is the chairman of its board. His current interests include parallel computing, data-base computer architectures, artificial intelligence computer architectures, and microcomputers.

This Page Intentionally Left Blank

1

Microcomputer Architecture

Microcomputers, microprocessors, and microprocessing are at once quite familiar and a bit fuzzy to most engineers and computer scientists. When we ask the question: "What is a microcomputer?" we get a wide range of answers. This chapter aims to clear up these terms. Also, the designer needs to be sufficiently familiar with the microcomputer instruction set to be able to read the object code generated by a C compiler. Clearly, we have to understand these concepts to be able to discuss and design I/O interfaces. This chapter contains essential material on microcomputers and microprocessors needed as a basis for understanding the discussion of interfacing in the rest of the book.

We recognize that the designer must have a comprehensive knowledge about basic computer architecture and organization. But the goal of this book is to impart enough knowledge so the reader, on completing it, should be ready to design good hardware and software for microcomputer interfaces. We have to trade material devoted to basics for material needed to design interface systems. There is so much to cover and so little space, that we will simply offer a summary of the main ideas. If you have had this material in other courses or absorbed it from your work or from reading those fine trade journals and hobby magazines devoted to microcomputers, this chapter should bring it all together. Some of you can pick up the material just by reading this condensed version. Others should get an idea of the amount of background needed to read the rest of the book.

For this chapter, we assume the reader is fairly familiar with some kind of Assembly Language on a large or small computer or is able to pick it up quickly. In this chapter, he or she should learn about the software view of microcomputers and embedded systems in general, and the M·CORE embedded processor in particular.

1.1 An Introduction to the Microcomputer

Just what is a microcomputer and a microprocessor, and what is the meaning of microprogramming — which is often confused with microcomputers? This section will survey these concepts and other commonly misunderstood terms in digital systems design. It describes the architecture of digital computers and gives a definition of architecture. Note that all *italicized* words are in the index and are listed at the end of each chapter; these serve as a glossary to help you find terms that you may need later.

Because the microcomputer is much like other computers except that it is smaller and less expensive, these concepts apply to large computers as well as microcomputers. The concept of the computer is presented first, and the idea of an instruction is scrutinized next. The special characteristics of microcomputers will be delineated last.

1.1.1 Computer Architecture

Actually, the first and perhaps the best paper on computer architecture, "Preliminary discussion of the logical design of an electronic computing instrument," by A. W. Burks, H. H. Goldstein, and J. von Neumann, was written 15 years before the term was coined. We find it fascinating to compare the design therein with all computers produced to date. It is a tribute to von Neumann's genius that this design, originally intended to solve nonlinear differential equations, has been successfully used in business data processing, information handling, and industrial control, as well as in numeric problems. His design is so well defined that most computers — from large computers to microcomputers — are based on it, and they are called *von Neumann computers.*

In the early 1960s a group of computer designers at IBM — including Fred Brooks — coined the term "architecture" to describe the "blueprint" of the IBM 360 family of computers, from which several computers with different costs and speeds (for example, the IBM 360/50) would be designed. The *architecture* of a computer is, strictly speaking, its instruction set and the input/output (I/O) connection capabilities. More generally, the architecture is the view of the hardware as seen by the programmer. Computers with the same architecture can execute the same programs and have the same I/O devices connected to them. Designing a collection of computers with the same "blueprint" or architecture has been done by several manufacturers. This definition of the term "computer architecture" applies to this fundamental level of design, as used in this book. However, outside of this book the term "computer architecture" has become very popular and is also rather loosely used to describe the computer system in general, including the implementation techniques and organization discussed next.

The *organization* of a digital system like a computer is usually shown by a block diagram which shows the registers, busses, and data operators in the computer. Two computers have the same organization if they have the same block diagram. For instance, Motorola manufactures several computers having the same architecture but different organizations to suit different applications. Incidentally, the organization of a computer is also called its *implementation.* Finally, the *realization* of the computer is its actual hardware interconnection and construction. It is entirely reasonable for a company to change the realization of one of its computers by replacing the hardware in a block of its block diagram with a newer type of hardware, which might be faster or cheaper. In this case the implementation or organization remains the same while the realization is different. In this book we will name the component by its full part number, like PMC2001HDCPU34 when we want to discuss an actual realization. However, we are usually interested only in the orga-

nization or the architecture only. In these cases, we will refer to an organization as a partial name without the suffix, such as MMC2001 without HDCPU34, and refer to the architecture as an M·CORE architecture or a number 6812. This should clear up any ambiguity, while also being a natural, easy-to-read shorthand.

The architecture of von Neumann computers is disarmingly simple, and the following analogy shows just how simple. (For an illustration of the following terms, see Figure 1.1) Imagine a person in front of a mailbox, with an adding machine and window to the outside world. The mailbox, with numbered boxes or slots, is analogous to the *primary memory*; the adding machine, to the *data operator* (arithmetic-logic unit); the person, to the *controller*; and the window, to *input/output* (I/O). The person's hands *access* the memory. Each slot in the mailbox has a paper that has a string of, say, 8 1s and 0s (*bits*) on it. A string of 8 bits is a *byte*, and four bits is a *nibble*. A string of 16 bits is called a *halfword*, and 32 bits is called a *word*.

The primary memory may be in part a *random access memory* (RAM) (so-called because the person is free to access its data in any order at random, without having to wait any longer for data because it is in a different location). RAM may be *static ram — SRAM —* if bits are stored in flip-flops, or *dynamic ram — DRAM —* if bits are stored as charges in capacitors. Memory that is normally written at the factory, never to be rewritten by the user, is called *read-only memory — ROM*. A *programmable read-only memory — PROM —* can be written once by a user, by blowing fuses to store bits in it. An *erasable programmable read-only memory — EPROM —* can be erased by ultraviolet light, and then written electrically by a user. An *electrically erasable programmable read-only memory — EEPROM —* can be erased and then written by a user, but erasing and writing words in EEPROM takes several milliseconds. A variation of this memory, called *flash*, is less expensive but can not be erased one word at a time.

With the left hand the person takes out a word from slot or box n, reads it as an instruction, and replaces it. Bringing a word from the mailbox (primary memory) to the person (controller) is called *fetching*. The hand that fetches a word from box n is

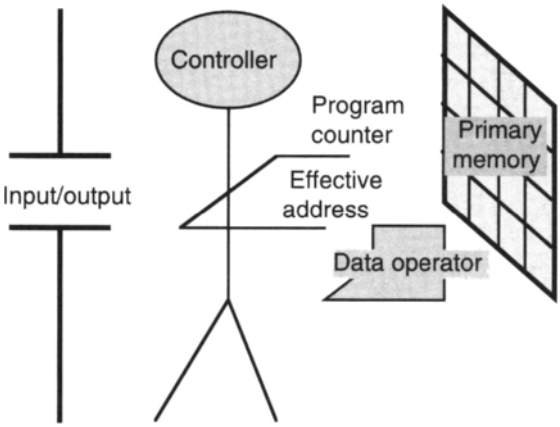


Figure 1.1. Analogy to the von Neumann Computer

analogous to the *program counter*. It is ready to take the word from the next box, box $n + 1$, when the next instruction is to be fetched.

An instruction in the M·CORE processor is a *binary code* such as 01001100. Consistent with the notation used by Motorola, binary codes are denoted in this book by a 0b (zero bee), followed by 1s or 0s. (Decimal numbers, by comparison, will not use any special symbols.) Since all those 1s and 0s are hard to remember, a convenient format is often used, called *hexadecimal notation*. In this notation, a 0x (zero ex) is written (to designate that the number is in hexadecimal notation), and the bits, in groups of 4, are represented as if they were "binary coded" digits 0 to 9 or letters A, B, C, D, E, and F to represent values 10, 11, 12, 13, 14, and 15, respectively. For example, %0100 is the binary code for 4, and %1100 is the binary code for 12, which, in hexadecimal notation, is represented as 0xC. The binary code 01001100, mentioned previously, is represented as 0x4C in hexadecimal notation. Whether the binary code or the simplified hexadecimal code is used, instructions written this way are called *machine-coded* instructions because that is the actual code fetched from the primary memory of the machine, or computer.

However, this is too cumbersome. So a *mnemonic* (which means a memory aid) is used to represent the instruction. All instructions in the M·CORE are entirely described by one 16-bit halfword. The M·CORE instruction 0x6001 actually puts a one into register `r1`, so it is written as

```
movi r1, #1
```

(The M·CORE registers such as `r1` are described in §1.2[1]. The mnemonic `movi` is described in §1.2.1. Strictly speaking, M·CORE mnemonics should be written in lower case to conform with Motorola's Applications Binary Interface Standards Manual MCOREABISM/AD.)

As better technology becomes available, and as experience with an architecture reveals its weaknesses, a new architecture may be crafted that includes most of the old instruction set and some new instructions. Programs written for the old computer should also run, with little or no change, on the new one, and more efficient programs can perhaps be written using new features of the new architectures. Such a new architecture is *upward compatible* from the old one if this property is preserved. If an architecture executes the same machine code the same way, it is fully upward compatible, but more generally, if it executes the same mnemonic instructions, even though they may be coded as different machine codes, then the architecture is *source code upward compatible*. The 6812 architecture is source code upward compatible from the 6811.

An *assembler* is a program that converts mnemonics into machine code so the programmer can write in convenient mnemonics and the output machine code is ready to be put in primary memory to be fetched as an instruction. The mnemonics are therefore called *assembly-language* instructions. A *compiler* is a program that converts statements in a *high-level language* either to assembly language, to be input

[1] § means "Section."

to an assembler, or to machine code, to be stored in memory and fetched by the controller.

While a lot of interface software is written in assembly language and many examples in this book are discussed using this language, most will be written in the high-level language C. However, quick fixes to programs are occasionally even written in machine code. Moreover, an engineer should want to know exactly how an instruction is stored and how the controller understands it. Therefore, in this chapter we will show the assembly language and machine code for some assembly-language instructions.

Now that we have some ideas about instructions, we resume the analogy to illustrate some things an instruction might do. For example, an instruction may direct the controller to clear register `r1` and write this word from `r1` to a box, where the address is the sum of a register `r2` plus 20. In the M·CORE architecture an instruction to store a word from `r1` into the word at the location indicated by `r2` plus twenty, is fetched as:

```
0x9152
```

where each byte essentially represents one of the instruction's parameters, and is represented by mnemonics as

```
st.w r1, (r2, 20)
```

in assembly language. The main operation — writing a word into the mailbox (primary memory) from the adding machine (data operator) — is called *memorizing* data. The right hand is used to get the word; it is analogous to the *effective address*.

As with instructions, assembly language uses a shorthand to represent locations in memory. A *symbolic address*, which is actually some address in memory, is a name that means something to the programmer. For example, `ALPHA` might be the twenty. Then the assembly-language instruction above can be written as follows:

```
st.w r1, (r2, ALPHA)
```

Other symbolic addresses and other locations can be substituted, of course. A symbolic address is just a representation of a number, which usually happens to be the numerical address in primary memory, or an offset of the word in primary memory relative to a register pointer. As a number, it can be added to other numbers, doubled, and so on. In particular, the instruction

```
st.w r1, (r2, ALPHA+4)
```

will store the word from register `r1` into the 24th location below that pointed to by `r2`.

Before going on, we point out a feature of the von Neumann computer that is easy to overlook, but is at once von Neumann's greatest contribution to computer architecture and yet a major problem in computing. Because instructions *and* data are stored in the primary memory, there is no way to distinguish one from the other

except by which hand (program counter or effective address) is used to get the data. We can conveniently use memory not needed to store instructions — if few are to be stored — to store more data, and vice versa. It is possible to modify an instruction as if it were data, just before it is fetched, although a good computer scientist would shudder at the thought. However, through an error (*bug*) in the program, it is possible to start fetching data words as if they were instructions, which produces strange results fast.

Generally, after such an instruction has been executed, the left hand (program counter) is in position to fetch the next instruction in box $n + 1$. For instance, if the pair of words shown below are in consecutive locations, they are executed sequentially:

```
0x6001
0x9152
```

These instructions are indicated in assembly-language source code in successive lines:

```
movi    r1, 0
st.w    r1, (r2, 20)
```

A *program sequence* is a sequence of instructions fetched from consecutive locations one after another. The program sequence given here cleared the word that is five words below the word whose address is in `r2`. Unless something is done to change the left hand (program counter), a sequence of words in contiguously numbered boxes will be fetched and executed as a program sequence. For example, a sequence of load and store instructions can be fetched and executed to copy a collection of words from one place in the mailbox into another place. However, when the controller reads the instruction, it may direct the left hand to move to a new location (load a new number in the program counter). Such an instruction is called a *jump*, which is an example of a *control instruction*. Such instructions will be discussed further in §1.2.3, where concrete examples using the M·CORE instruction set are described. To facilitate the memory access functions, the effective address can be computed in a number of ways, called *addressing modes*. M·CORE addressing modes will be explained in §1.2.1.

1.1.2 The Instruction

In this section the concept of an instruction is described from different points of view. The instruction is discussed first with respect to fetching, decoding, and executing them. Then the instruction is discussed in relation to hardware-software trade-offs. Some concepts used in choosing the best instruction set are also discussed.

The controller fetches a word or a couple of words from primary memory and sends commands to all the modules to execute the instruction. An instruction, then, is essentially a complex command carried out under the direction of a single word or a couple of words fetched as an inseparable group from memory.