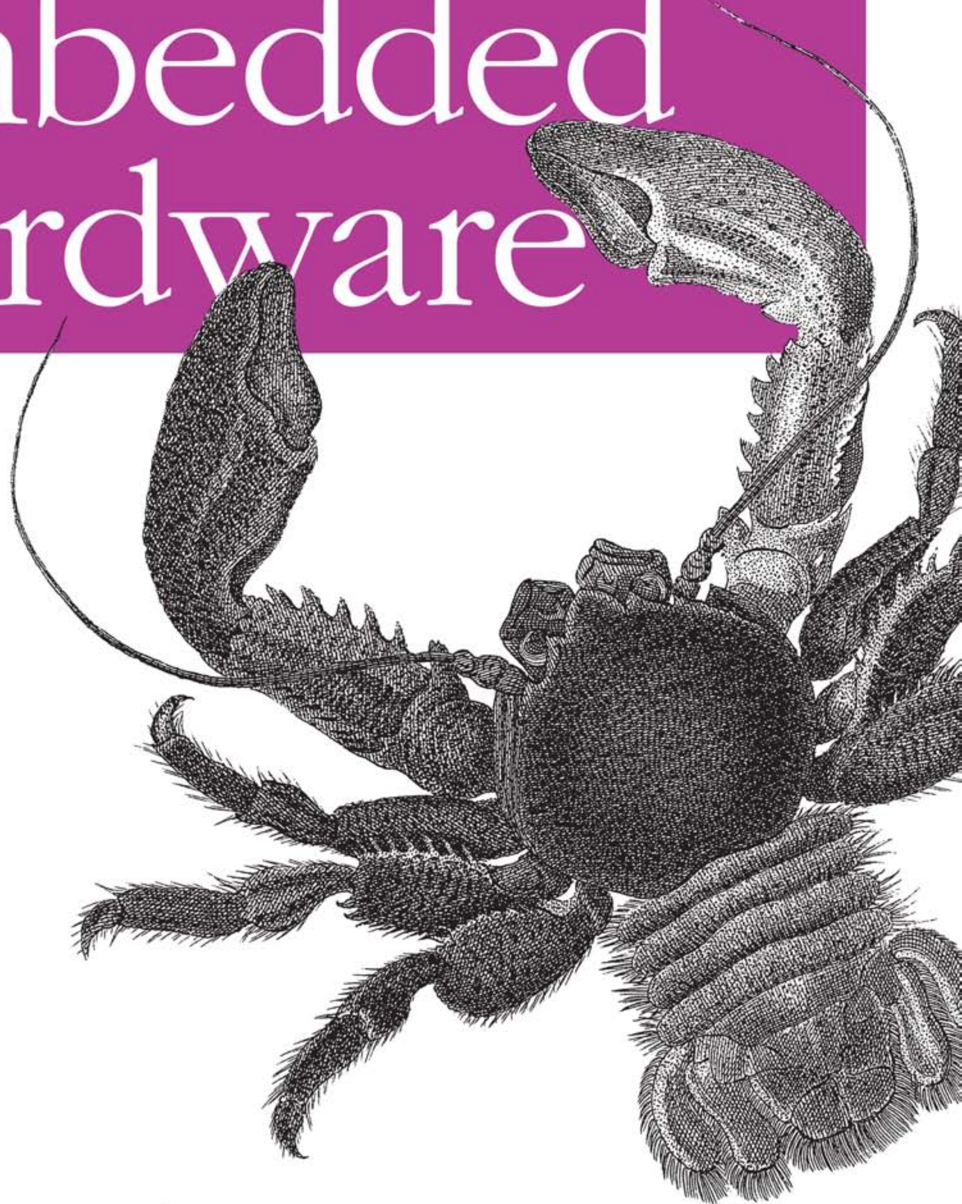
Create New Computers and Devices

2nd Edition

Designing

Embedded Hardware



O'REILLY®

John Catsoulis

Hardware

O'REILLY®

Designing Embedded Hardware

Embedded computer systems literally surround us: they're in our cell phones, PDAs, cars, TVs, refrigerators, heating systems, and more. In fact, embedded systems are one of the most rapidly growing segments of the computer industry today.

Along with the growing list of devices for which embedded computer systems are appropriate, interest is growing among programmers, hobbyists, and engineers of all types in how to design and build devices of their own. Furthermore, the knowledge offered by this book of the fundamentals of these computer systems can benefit anyone who has to evaluate and apply the systems.

The second edition of *Designing Embedded Hardware* has been updated to include information on the latest generation of processors and microcontrollers, including the new MAXQ processor. The book spells out the basics of embedded design for beginners while providing material useful for advanced systems designers.

Designing Embedded Hardware steers a course between books dedicated to writing code for particular microprocessors and those that stress the philosophy of embedded-system design without providing any practical information. Author John Catsoulis brings a wealth of real-world experience to show readers how to design and create entirely new embedded devices and computerized gadgets, as well as how to customize and extend off-the-shelf systems.

Loaded with real-world examples, this book also provides a roadmap to the pitfalls and traps to avoid. *Designing Embedded Hardware* includes:

- Theory and practice of embedded systems
- Understanding schematics and datasheets
- Powering an embedded system
- Producing and debugging an embedded system
- Processors such as the PIC, Atmel AVR, and Motorola 68000 series
- Digital Signal Processing (DSP) architectures
- Protocols (SPI and I2C) used to add peripherals
- RS-232C, RS-422, infrared communication, and USB
- CAN and Ethernet networking
- Pulse Width Monitoring and motor control

If you want to build your own embedded system, or tweak an existing one, this invaluable book gives you the knowledge and practical skills you need.

www.oreilly.com

US $44.95 **CAN $62.95**

ISBN: 978-0-596-00755-3

Safari BOOKS ONLINE ENABLED

Includes FREE 45-Day Online Edition

Designing Embedded Hardware

Other resources from O'Reilly

Related titles

Building Embedded Linux Systems

Home Hacking Projects for Geeks

Hardware Hacking Projects for Geeks

PC Hardware Buyer's Guide

Programming Embedded Systems in C and C++

Car PC Hacks

Smart Home Hacks

oreilly.com

oreilly.com is more than a complete catalog of O'Reilly books. You'll also find links to news, events, articles, weblogs, sample chapters, and code examples.

oreillynet.com is the essential portal for developers interested in open and emerging technologies, including new platforms, programming languages, and operating systems.

Conferences

O'Reilly brings diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.

Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Search across thousands of electronic books simultaneously and zero in on the information you need in seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today for free.

SECOND EDITION

Designing Embedded Hardware

John Catsoulis

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Designing Embedded Hardware, Second Edition
by John Catsoulis

Copyright © 2005, 2002 O'Reilly Media, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram

Production Editor: Sanders Kleinfeld

Cover Designer: Emma Colby

Interior Designer: David Futato

Printing History:

November 2002: First Edition.

May 2005: Second Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Designing Embedded Hardware*, the image of a porcelain crab, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-00755-3
[M] [9/09]

Dedication

This book is dedicated to my uncle, Vince Catsoulis.

Much of the information contained in this document is based on personal knowledge and experience. While it is believed that the information contained herein is correct, no responsibility is accepted for its validity. The hardware designs, software and descriptive text contained herein are provided for educational purposes only. It

is the responsibility of the reader to independently verify all information. Original manufacturers' data should be used at all times when implementing a design.

The author or O'Reilly & Associates, Inc make no warranty, representation or guarantee regarding the suitability of any hardware or software described herein for any particular purpose, nor do they assume any liability arising out of the application or use of any product, system, circuit or software and specifically disclaim any and all liability, including without limitation, consequential or incidental damages. The hardware and software described herein are not designed, intended, or authorized for use in any application intended to support or sustain life, or any other application where the failure of a system could create a situation where personal injury, death, loss of data or information, or damages to property may occur. Should the reader implement any design described herein for any application, the reader shall indemnify and hold the author, O'Reilly & Associates, Inc and their respective shareholders, officers, employees and distributors harmless against all claims, costs, damages and expenses and reasonable solicitor fees arising out of, directly or indirectly, any claim of personal injury, death, loss of data or information, or damages to property associated with such unintended or unauthorized use.

Table of Contents

Preface . . . . **xi**

1. An Introduction to Computer Architecture **1**

Concepts 2
Memory 16
Input/Output 20
DMA 20
Embedded Computer Architecture 26

2. Assembly Language **30**

Registers 32
Machine Code 32
Signed Numbers 34
Addressing Modes 35
Coding in Assembly 37
Disassembly 40
Position-Independent Code 41
Loops 41
Masking 42
Indexed Addressing 43
Stacks 44
Timing of Instructions 45

3. Forth/Open Firmware **48**

Introducing Forth 48
String Words 51
Stack Manipulation 52
Creating New Words 54

Comments 56
if ... else 57
Loops 58
Data Structures 61
Interacting with Hardware and Memory 62
Forth Programming Guidelines 64

4. Electronics 101 65
Voltage and Current 65
Analog Signals 67
Power 68
Reading Schematics 68
Resistors 73
Capacitors 80
RC Circuits 83
Inductors 86
Transformers 89
Diodes 90
Crystals 93
Digital Signals 98
Electrical Characteristics 99
Logic Gates 108
The Importance of Reading the Datasheet 109

5. Power Sources 110
The Stuff Out of the Wall 110
Batteries 111
Low Power Design 111
Regulators 112
LM78xx Regulators 114
MAX603/MAX604 Regulators 116
MAX1615 Regulator 117
MAX724 Regulator 118
Electrical Noise and Interference 119

6. Building Hardware 124
Tools 124
Soldering 129
Quick Construction 136

Printed-Circuit Boards 140
Building It 153
JTAG 157

7. Adding Peripherals Using SPI . . . 160
Serial Peripheral Interface 160

8. Adding Peripherals Using I²C . . . 174
Overview of I²C 174
Adding a Real-Time Clock with I²C 178
Adding a Small Display with I²C 179

9. Serial Ports . . . 180
UARTs 180
Error Detection 182
Old Faithful: RS-232C 183
RS-422 190
RS-485 192

10. IrDA . . . 196
Introduction to IrDA 196

11. USB . . . 203
Introduction to USB 204
USB Packets 206
Physical Interface 208
Implementing a USB Interface 211

12. Networks . . . 215
Controller Area Network (CAN) 215
Ethernet 219

13. Analog . . . 226
Amplifiers 226
Analog to Digital Conversion 229
Interfacing an External ADC 233
Temperature Sensor 235
Light Sensor 237
Accelerometer 240
Pressure Sensors 242

Magnetic-Field Sensor 244
Digital to Analog Conversion 245
PWM 248
Motor Control 249
Switching Big Loads 256

14. The PIC Microcontrollers . . . 258
A Tale of Two Processors 258
Starting Simple 260
A Bigger PIC 263
PIC-Based Environmental Datalogger 265
Motor Control with a PIC 271

15. The AVR Microcontrollers . . . 277
The AVR Architecture 278
The ATtiny15 Processor 280
Downloading Code 287
A Bigger AVR 289
AVR-Based Datalogger 290
Bus Interfacing 291

16. 68HC11 . . . 316
Architecture of the 68HC11 316
A Simple 68HC11-Based Computer 317

17. MAXQ . . . 327
Architectural Overview 327
Schematics 329

18. 68000-Series Computers . . . 334
The 68000 Architecture 335
A Simple 68000-Based Computer 339

19. DSP-Based Controllers . . . 348
The DSP56800 351
A DSP56805-Based Computer 353
JTAG 361

Index . . . 363

Preface

[Enlightenment] resides as comfortably in the circuits of a digital computer … as at the top of a mountain or in the petals of a flower.

—Robert M. Pirsig
Zen and the Art of Motorcycle Maintenance

Welcome to the second edition of *Designing Embedded Hardware*. In these pages, I hope to give you an understanding of the design process for creating computer hardware. Just as there is beauty in well-written software, there is beauty in well-designed hardware. With embedded computers, you get to understand the machine at all levels, at once aware of currents flowing through circuit traces and software executing complex algorithms. In fact, it is not possible to write embedded software without understanding the hardware, nor is it possible to design hardware without understanding software. You become involved with the machine to a degree beyond that which is possible with desktop computers. Best of all, it's a lot of fun.

In selecting chips and designs for this book, I have tried to choose, where possible, parts that are both trivial to use yet exceptionally useful. I have no connection, financial or otherwise, with any of the companies or businesses mentioned in this book, and I receive no benefits from them. Every component or product included in this book is there based on its own merits. You may notice a prevalence of components from certain manufacturers. This simply reflects my personal preference for using their chips, based on my experience. Such companies produce chips that are easy to use, are reliable and robust, have great technical support, and provide thorough and comprehensive technical data. In other words, they meet all the necessary requirements for inclusion in a book for beginners.

When the first edition of *Designing Embedded Hardware* was published, I deliberately left out software. There were two reasons for this. First, there are many good books already written on C programming, embedded firmware development in C, porting Linux to embedded systems, coding in Python, writing Java software, and so on. (And of course, the best of these are naturally published by O'Reilly.) The second

reason is that assembly language, that most basic of programming tools, is so different from processor to processor that it would not have been possible to cover all the instruction sets of the processors in the book, let alone do them justice. However, I have decided to include some software in this edition. I won't even attempt to cover the instructions of each processor in this book. What I will do is show some simple assembly language techniques. While the instructions may be wildly different between architectures, the basic concepts are the same.

Also new to this book is a chapter on the Forth programming language. Forth is a relatively old language that has faded from the forefront of software development, and as such, it's rare to find a book giving the language good coverage. Forth is a very useful tool for embedded system development to which many engineers have yet to be exposed. The language is the basis of the Open Firmware standard and is used by design engineers at Apple, Sun, and many other manufacturers. It's a useful language to know, and it is worth taking the time to learn.

Many of the designs in this book look easy, and they are. They are intended as simple building blocks, allowing you to mix and match to achieve the embedded systems you need. I hope you will find this book useful. Once you've finished reading it, go and build something!

—John Catsoulis
Brisbane, Australia
January 2005
jtc@embedded.com.au
http://www.embedded.com.au

Organization of This Book

This book is informally divided into four sections. The first covers fundamental concepts and introductory material. The second section gives an overview of assembly language and Forth. From there, we'll look at peripherals and how to add functionality to your embedded systems. Finally, we'll look at a variety of processors widely used in embedded systems, and look at the design process for integrating them into computers.

Chapter 1 gives an overview of computer architectures and explains what constitutes an embedded system. Chapters 2 and 3 explore software with assembly language and Forth.

Chapter 4 provides some background electronics theory and introduces some important concepts. If you're already electronics-savvy, then you can skip on to Chapter 5, where we'll look at providing power for your embedded system. We'll also look at how to protect your embedded computer against electrical interference and other gremlins that can cause it grief. In Chapter 6, you'll see how to physically produce and debug an embedded computer system.

Chapters 7 and 8 cover SPI and I^2C, two protocols that allow a wide range of small peripherals to be added to microcontrollers. Chapters 9, 10, and 11 cover serial interfaces. These give our embedded system access to host computers and to external peripherals such as modems. We'll look at RS-232C, RS-422, Infrared communication, and USB.

Networks are covered in Chapter 12, where we'll see how to add a low-cost industrial network (CAN) to our embedded computer. Also in Chapter 12, we learn how to add an Ethernet port to our embedded system, by which we can connect to other computers, servers, and gateways and, through them, to the wider Internet.

In Chapter 13, we'll look at real-world interfacing. We'll see how to convert analog signals into digital values for processing and, conversely, how to convert digital values back into analog voltages. We'll also see how to interface sensors to our embedded system, whereby we can measure temperature, light, pressure, acceleration and magnetic fields. Also in Chapter 13, we'll look at Pulse Width Modulation and motor control. We'll see how to use an embedded computer to control small electric motors.

Chapter 14 begins the microprocessor section of the book, where we'll look at the first of our embedded processor architectures, the Microchip PIC. In subsequent chapters, we'll meet a variety of processors, from tiny standalone, 8-bit microcontrollers to 32-bit, bus-based chips with some computing grunt. While it is not possible to cover every embedded processor (as there are literally many hundreds), the chips chosen are representative of various classes of processor. The skills you learn will be adaptable to whatever processor you choose for your application.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact O'Reilly for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Designing Embedded Hardware*, by John Catsoulis. Copyright 2005 O'Reilly Media, Inc., 0-596-00755-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact the publisher at *permissions@oreilly.com*.

Conventions

The conventions used in this book are as follows:

Main text

```
Source Code
```

Signal (active high)

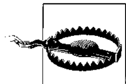
$\overline{\textbf{Signal}}$ (active low)

Hexadecimal numbers are denoted with the prefix *0x*, and sometimes with the prefix *$*, where appropriate for certain processors.

Binary numbers are denoted by the prefix *%*.

K is 1,024, while k is 1,000.

This icon indicates a tip, suggestion, or general note.



This icon indicates a warning or caution.

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, it means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/dbhardware2/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

To contact the author directly with comments or questions, send email to:

jtc@embedded.com.au

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

I'd like to give a special thank you to my editors, Andy Oram and Mike Hendrickson. I'd also like to thank Jon Orwant, editor of the first edition. In the past, I have often read in prefaces how authors thank their editors for the help they gave. It is only now that I understand the depth and significance of this help.

As you have no doubt already noticed, O'Reilly publishes beautifully presented books. I would like to thank the production team, especially Sanders Kleinfeld, for their hard work. This book is as much the result of their efforts as it is mine. I'd also like to thank David Chu for all his help.

Thank you to Dallas Semiconductor, Kathy Vehorn, Don Loomis, Mike Quarles, and Moe Rubenzahl for their assistance and for allowing me access to pre-release versions of the MAXQ processor. Thank you to Peter Paine, Donna Mack and Cooper Tools, Karen Rolan and Fluke Corporation, and Tektronix for allowing me to use their images in the book. Thank you also to Rupert Baines of Picochip for his assistance.

Geoff McDonald has been a great friend and has made many helpful suggestions regarding the content of this book. He also proofread the book, and I thank him for all his help.

Thanks to Michael, Mary, Renee, and Mitchell Lees. Michael did significant proofreading and offered many helpful comments.

I'd like to thank Dr. Jeff O'Keefe for his long friendship and support over the years. He's been a good friend ever since we were undergrads together, blowing up integrated circuits and irradiating lecturers in second-year lab!

Thank you to Prof. Neil Bergmann, Dr. John Williams, Keith Ball, Denis Bill, Barry Bettridge, and the staff of the School of ITEE at the University of Queensland for their help and support.

I'd like to thank my friends and colleagues David Nicholls, Peter Stewart, Mark Gentile, Professor John Devlin, Richard Wiltshire, Michelle and Robert Salier, Addy and Derek Clark, Kam Tam, Phil McDonald, and Vamsi Madasu.

Finally and most importantly, I'd like to thank my extended family for their love and support. Most especially, I'd like to thank my sister Kris, and my two nephews, Andrew and James, whose love and good humor have made life worth living. I'd also like to thank Chris and Jeff Goopy for always being there, and my cousins Theo and Maree; David and Jenevieve; Michael, Andrew and Karen; Antony; and Fiona, Drew, Ashley, and Max for their care and support. A special thank you to my uncles, Vince and Dave Catsoulis, who have shown me the meaning of love, honor, and strength of character. I owe them much.

CHAPTER 1

An Introduction to Computer Architecture

Each machine has its own, unique personality which probably could be defined as the intuitive sum total of everything you know and feel about it. This personality constantly changes, usually for the worse, but sometimes surprisingly for the better...

—Robert M. Pirsig
Zen and the Art of Motorcycle Maintenance

This book is about designing and building specialized computers. We all know what a computer is. It's that box that sits on your desk, quietly purring away (or rattling if the fan is shot), running your programs and regularly crashing (if you're not running some variety of Unix). Inside that box is the electronics that runs your software, stores your information, and connects you to the world. It's all about processing information. Designing a computer, therefore, is about designing a machine that holds and manipulates data.

Computer systems fall into essentially two separate categories. The first, and most obvious, is that of the desktop computer. When you say "computer" to someone, this is the machine that usually comes to her mind. The second type of computer is the embedded computer, a computer that is integrated into another system for the purposes of control and/or monitoring. Embedded computers are far more numerous than desktop systems, but far less obvious. Ask the average person how many computers he has in his home, and he might reply that he has one or two. In fact, he may have 30 or more, hidden inside his TVs, VCRs, DVD players, remote controls, washing machines, cell phones, air conditioners, game consoles, ovens, toys, and a host of other devices.

In this chapter, we'll look at computer architecture in general. This is applicable to both embedded and desktop computers, because the primary difference between an embedded machine and a general-purpose computer is its application. The basic principles of operation and the underlying architectures are fundamentally the same.

Both have a processor, memory, and often several forms of input and output. The primary difference lies in their intended use, and this is reflected in the system design and their software. Desktop computers can run a variety of application programs, with system resources orchestrated by an operating system. By running different application programs, the functionality of the desktop computer is changed. One moment, it may be used as a word processor; the next it is an MP3 player or a database client. Which software is loaded and run is under user control.

In contrast, the embedded computer is normally dedicated to a specific task. In many cases, an embedded system is used to replace application-specific electronics. The advantage of using an embedded microprocessor over dedicated electronics is that the functionality of the system is determined by the software, not the hardware. This makes the embedded system easier to produce, and much easier to evolve, than a complicated circuit.

The embedded system typically has one application and one application only, which is permanently running. The embedded computer may or may not have an operating system, and rarely does it provide the user with the ability to arbitrarily install new software. The software is normally contained in the system's nonvolatile memory, unlike a desktop computer where the nonvolatile memory contains boot software and (maybe) low-level drivers only.

Embedded hardware is often much simpler than a desktop system, but it can also be far more complex too. An embedded computer may be implemented in a single chip with just a few support components, and its purpose may be as crude as a controller for a garden-watering system. Alternatively, the embedded computer may be a 150-processor, distributed parallel machine responsible for all the flight and control systems of a commercial jet. As diverse as embedded hardware may be, the underlying principles of design are the same.

This chapter introduces some important concepts relating to computer architecture, with specific emphasis on those topics relevant to embedded systems. Its purpose is to give you grounding before moving on to the more hands-on information that begins in Chapter 2. In this chapter, you'll learn about the basics of processors, interrupts, the difference between RISC and CISC, parallel systems, memory, and I/O.

Concepts

Let's start at the beginning.

In essence, a computer is a machine designed to process, store, and retrieve data. Data may be numbers in a spreadsheet, characters of text in a document, dots of color in an image, waveforms of sound, or the state of some system, such as an air conditioner or a CD player. *All data is stored in the computer as numbers*. It's easy to forget this when we're deep in C code, contemplating complex algorithms and data structures.

The computer manipulates the data by performing operations on the numbers. Displaying an image on a screen is accomplished by moving an array of numbers to the video memory, each number representing a pixel of color. To play an MP3 audio file, the computer reads an array of numbers from disk and into memory, manipulates those numbers to convert the compressed audio data into raw audio data, and then outputs the new set of numbers (the raw audio data) to the audio chip.

Everything that a computer does, from web browsing to printing, involves moving and processing numbers. The electronics of a computer is nothing more than a system designed to hold, move, and change numbers.

A computer system is composed of many parts, both hardware and software. At the heart of the computer is the processor, the hardware that executes the computer programs. The computer also has memory, often several different types in one system. The memory is used to store programs while the processor is running them, as well as store the data that the programs are manipulating. The computer also has devices for storing data, or exchanging data with the outside world. These may allow the input of text via a keyboard, the display of information on a screen, or the movement of programs and data to or from a disk drive.

The software controls the operation and functionality of the computer. There are many "layers" of software in the computer (Figure 1-1). Typically, a given layer will only interact with the layers immediately above or below it.

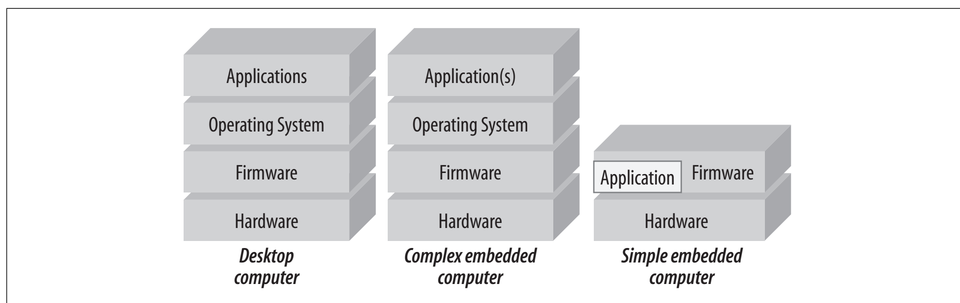


Figure 1-1. Software layers

At the lowest level, there are programs that are run by the processor when the computer first powers up. These programs initialize the other hardware subsystems to a known state and configure the computer for correct operation. This software, because it is permanently stored in the computer's memory, is known as *firmware*.

The *bootloader* is located in the firmware. The bootloader is a special program run by the processor that reads the operating system from disk (or nonvolatile memory or network interface) and places it in memory so that the processor may then run it. The bootloader is present in desktop computers and workstations, and may be present in some embedded computers.

Above the firmware, the operating system controls the operation of the computer. It organizes the use of memory and controls devices such as the keyboard, mouse, screen, disk drives, and so on. It is also the software that often provides an interface to the user, enabling her to run application programs and access her files on disk. The operating system typically provides a set of software tools for application programs, providing a mechanism by which they too can access the screen, disk drives, and so on. Not all embedded systems use or even need an operating system. Often, an embedded system will simply run code dedicated to its task, and the presence of an operating system is overkill. In other instances, such as network routers, an operating system provides necessary software integration and greatly simplifies the development process. Whether an operating system is needed and useful really depends on the intended purpose of the embedded computer and, to a lesser degree, on the preference of the designer.

At the highest level, the *application software* constitutes the programs that provide the functionality of the computer. Everything below the application is considered *system software*. For embedded computers, the boundary between application and system software is often blurred. This reflects the underlying principle in embedded design that a system should be designed to achieve its objective in as simple and straightforward a manner as possible.

Processors

The processor is the most important part of a computer, the component around which everything else is centered. In essence, the processor is the computing part of the computer. A processor is an electronic device capable of manipulating data (information) in a way specified by a sequence of instructions. The instructions are also known as *opcodes* or *machine code*. This sequence of instructions may be altered to suit the application, and, hence, computers are programmable. A sequence of instructions is what constitutes a program.

Instructions in a computer are numbers, just like data. Different numbers, when read and executed by a processor, cause different things to happen. A good analogy is the mechanism of a music box. A music box has a rotating drum with little bumps, and a row of prongs. As the drum rotates, different prongs in turn are activated by the bumps, and music is produced. In a similar way, the bit patterns of instructions feed into the execution unit of the processor. Different bit patterns activate or deactivate different parts of the processing core. Thus, the bit pattern of a given instruction may activate an addition operation, while another bit pattern may cause a byte to be stored to memory.

A sequence of instructions is a machine-code program. Each type of processor has a different *instruction set*, meaning that the functionality of the instructions (and the bit patterns that activate them) varies. Processor instructions are often quite simple, such as “add two numbers” or “call this function.” In some processors, however,

they can be as complex and sophisticated as "if the result of the last operation was zero, then use this particular number to reference another number in memory, and then increment the first number once you've finished." This will be covered in more detail in the section on CISC and RISC processors, later in this chapter.

Basic System Architecture

The processor alone is incapable of successfully performing any tasks. It requires memory (for program and data storage), support logic, and at least one I/O device ("input/output device") used to transfer data between the computer and the outside world. The basic computer system is shown in Figure 1-2.

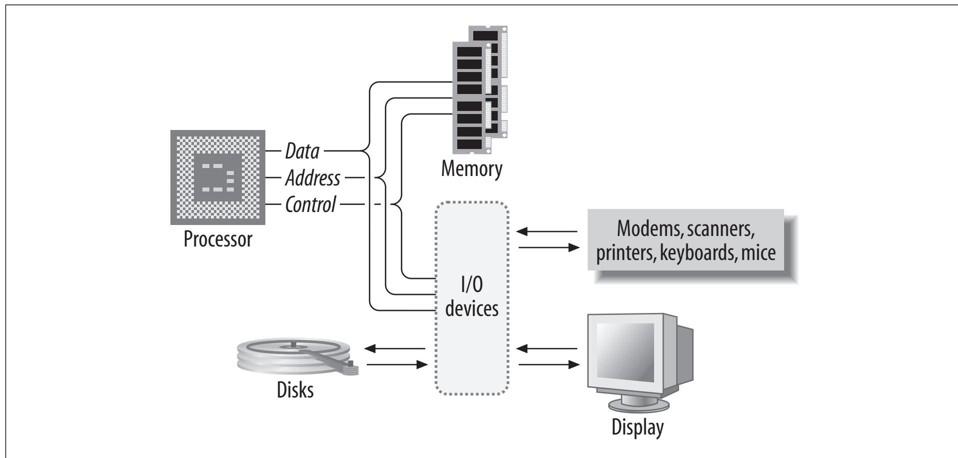


Figure 1-2. Basic computer system

A *microprocessor* is a processor implemented (usually) on a single, integrated circuit. With the exception of those found in some large supercomputers, nearly all modern processors are microprocessors, and the two terms are often used interchangeably. Common microprocessors in use today are the Intel Pentium series, Freescale/IBM PowerPC, MIPS, ARM, and the Sun SPARC, among others. A microprocessor is sometimes also known as a *CPU* (*Central Processing Unit*).

A *microcontroller* is a processor, memory, and some I/O devices contained within a single, integrated circuit, and intended for use in embedded systems. The buses that interconnect the processor with its I/O exist within the same integrated circuit. The range of available microcontrollers is very broad. They range from the tiny PICs and AVRs (to be covered in this book) to PowerPC processors with built-in I/O, intended for embedded applications. In this book, we will look at both microprocessors and microcontrollers.

Microcontrollers are very similar to *System-on-Chip* (*SoC*) processors, intended for use in conventional computers such as PCs and workstations. SoC processors have a

different suite of I/O, reflecting their intended application, and are designed to be interfaced to large banks of external memory. Microcontrollers usually have all their memory on-chip and may provide only limited support for external memory devices.

The memory of the computer system contains both the instructions that the processor will execute and the data it will manipulate. The memory of a computer system is never empty. It always contains something, whether it be instructions, meaningful data, or just the random garbage that appeared in the memory when the system powered up.

Instructions are read (fetched) from memory, while data is both read from and written to memory, as shown in Figure 1-3.

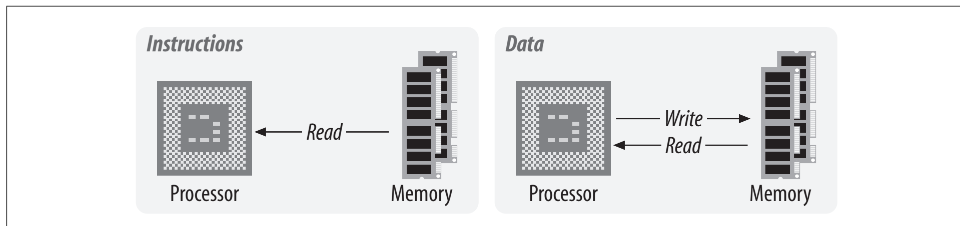


Figure 1-3. Data flow

This form of computer architecture is known as a *Von Neumann machine*, named after John Von Neumann, one of the originators of the concept. With very few exceptions, nearly all modern computers follow this form. Von Neumann computers are what can be termed control-flow computers. The steps taken by the computer are governed by the sequential control of a program. In other words, the computer follows a step-by-step program that governs its operation.

There are some interesting non–Von Neumann architectures, such as the massively parallel Connection Machine and the nascent efforts at building biological and quantum computers, or neural networks.

A classical Von Neumann machine has several distinguishing characteristics:

There is no real difference between data and instructions. A processor can be directed to begin execution at a given point in memory, and it has no way of knowing whether the sequence of numbers beginning at that point is data or instructions. The instruction 0x4143 may also be data (the number 0x4143, or the ASCII characters “A” and “C”). The processor has no way of telling what is data or what is an instruction. If a number is to be executed by the processor, it is an instruction; if it is to be manipulated, it is data.

Because of this lack of distinction, the processor is capable of changing its instructions (treating them as data) under program control. And because the processor has no way of distinguishing between data and instruction, it will

blindly execute anything that it is given, whether it is a meaningful sequence of instructions or not.

Data has no inherent meaning. There is nothing to distinguish between a number that represents a dot of color in an image and a number that represents a character in a text document. Meaning comes from how these numbers are treated under the execution of a program.

Data and instructions share the same memory. This means that sequences of instructions in a program may be treated as data by another program. A compiler creates a program binary by generating a sequence of numbers (instructions) in memory. To the compiler, the compiled program is just data, and it is treated as such. It is a program only when the processor begins execution. Similarly, an operating system loading an application program from disk does so by treating the sequence of instructions of that program as data. The program is loaded to memory just as an image or text file would be, and this is possible due to the shared memory space.

Memory is a linear (one-dimensional) array of storage locations. The processor's memory space may contain the operating system, various programs, and their associated data, all within the same linear space.

Each location in the memory space has a unique, sequential address. The address of a memory location is used to specify (and select) that location. The memory space is also known as the *address space*, and how that address space is partitioned between different memory and I/O devices is known as the *memory map*. The address space is the array of all addressable memory locations. In an 8-bit processor (such as the 68HC11) with a 16-bit address bus, this works out to be 2^{16} = 65,536 = 64K of memory. Hence, the processor is said to have a 64K address space. Processors with 32-bit address buses can access 2^{32} = 4,294,967,296 = 4G of memory.

Some processors, notably the Intel x86 family, have a separate address space for I/O devices with separate instructions for accessing this space. This is known as *ported I/O*. However, most processors make no distinction between memory devices and I/O devices within the address space. I/O devices exist within the same linear space as memory devices, and the same instructions are used to access each. This is known as *memory-mapped I/O* (Figure 1-4). Memory-mapped I/O is certainly the most common form. Ported I/O address spaces are becoming rare, and the use of the term even rarer.

Most microprocessors available are standard Von Neumann machines. The main deviation from this is the *Harvard architecture*, in which instructions and data have different memory spaces (Figure 1-5) with separate address, data, and control buses for each memory space. This has a number of advantages in that instruction and data fetches can occur concurrently, and the size of an instruction is not set by the size of the standard data unit (word).

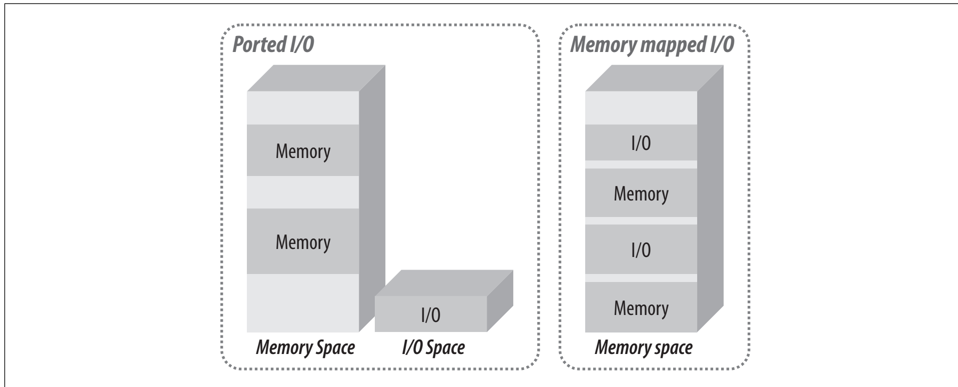


Figure 1-4. Ported versus memory-mapped I/O spaces

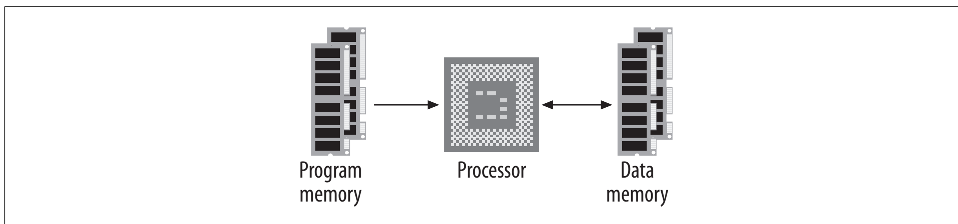


Figure 1-5. Harvard architecture

Buses

A bus is a physical group of signal lines that have a related function. Buses allow for the transfer of electrical signals between different parts of the computer system and thereby transfer information from one device to another. For example, the data bus is the group of signal lines that carry data between the processor and the various subsystems that comprise the computer. The "width" of a bus is the number of signal lines dedicated to transferring information. For example, an 8-bit-wide bus transfers 8 bits of data in parallel.

The majority of microprocessors available today (with some exceptions) use the three-bus system architecture (Figure 1-6). The three buses are the *address bus*, the *data bus*, and the *control bus*.

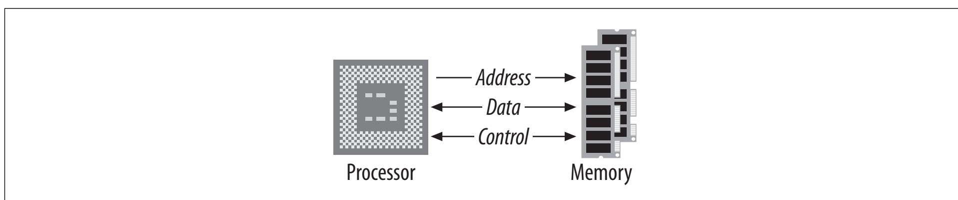


Figure 1-6. Three-bus system

The data bus is bidirectional, the direction of transfer being determined by the processor. The address bus carries the address, which points to the location in memory that the processor is attempting to access. It is the job of external circuitry to determine in which external device a given memory location exists and to activate that device. This is known as *address decoding*. The control bus carries information from the processor about the state of the current access, such as whether it is a write or a read operation. The control bus can also carry information back to the processor regarding the current access, such as an address error. Different processors have different control lines, but there are some control lines that are common among many processors. The control bus may consist of output signals such as read, write, valid address, etc. A processor usually has several input control lines too, such as reset, one or more interrupt lines, and a clock input.

A few years ago, I had the opportunity to wander through, in, and around CSIRAC (pronounced "sigh-rack"). This was one of the world's first digital computers, designed and built in Sydney, Australia, in the late 1940s. It was a massive machine, filling a very big room with the type of solid hardware that you can really kick. It was quite an experience looking over the old machine. I remember at one stage walking *through* the disk controller (it was the size of small room) and looking up at a mass of wires strung overhead. I asked what they were for. "That's the data bus!" came the reply.

CSIRAC is now housed in the museum of the University of Melbourne. You can take an online tour of the machine, and even download a simulator, at *http://www.cs.mu.oz.au/csirac*.

Processor operation

There are six basic types of access that a processor can perform with external chips. The processor can write data to memory or write data to an I/O device, read data from memory or read data from an I/O device, read instructions from memory, and perform internal manipulation of data within the processor.

In many systems, writing data to memory is functionally identical to writing data to an I/O device. Similarly, reading data from memory constitutes the same external operation as reading data from an I/O device, or reading an instruction from memory. In other words, the processor makes no distinction between memory and I/O.

The internal data storage of the processor is known as its *registers*. The processor has a limited number of registers, and these are used to hold the current data/operands that the processor is manipulating.

ALU

The Arithmetic Logic Unit (ALU) performs the internal arithmetic manipulation of data in the processor. The instructions that are read and executed by the processor control the data flow between the registers and the ALU. The instructions also

control the arithmetic operations performed by the ALU via the ALU's control inputs. A symbolic representation of an ALU is shown in Figure 1-7.

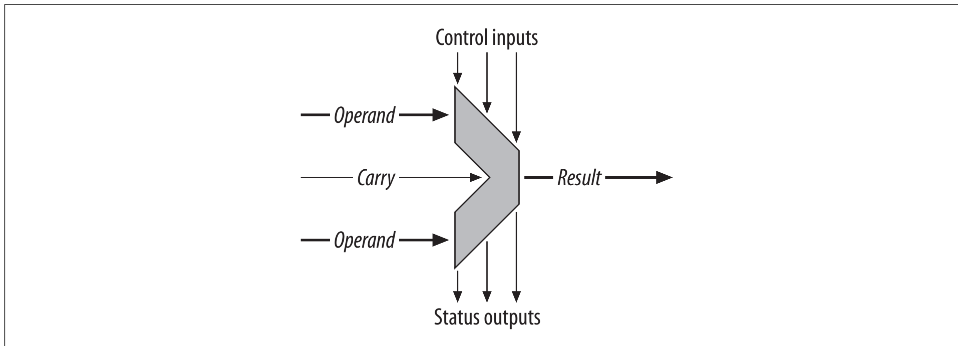


Figure 1-7. ALU block diagram

Whenever instructed by the processor, the ALU performs an operation (typically one of addition, subtraction, NOT, AND, OR, XOR, shift left/right, or rotate left/right) on one or more values. These values, called *operands*, are typically obtained from two registers, or from one register and a memory location. The result of the operation is then placed back into a given destination register or memory location. The status outputs indicate any special attributes about the operation, such as whether the result was zero, negative, or if an overflow or carry occurred. Some processors have separate units for multiplication and division, and for bit shifting, providing faster operation and increased throughput.

Each architecture has its own unique ALU features, and this can vary greatly from one processor to another. However, all are just variations on a theme, and all share the common characteristics just described.

Interrupts

Interrupts (also known as *traps* or *exceptions* in some processors) are a technique of diverting the processor from the execution of the current program so that it may deal with some event that has occurred. Such an event may be an error from a peripheral, or simply that an I/O device has finished the last task it was given and is now ready for another. An interrupt is generated in your computer every time you type a key or move the mouse. You can think of it as a hardware-generated function call.

Interrupts free the processor from having to continuously check the I/O devices to determine whether they require service. Instead, the processor may continue with other tasks. The I/O devices will notify it when they require attention by asserting one of the processor's interrupt inputs. Interrupts can be of varying priorities in some processors, thereby assigning differing importance to the events that can interrupt the processor. If the processor is servicing a low-priority interrupt, it will pause it in order to service a higher-priority interrupt. However, if the processor is servicing an

interrupt and a second, lower-priority interrupt occurs, the processor will ignore that interrupt until it has finished the higher-priority service.

When an interrupt occurs, the usual procedure is for the processor to save its state by pushing its registers and program counter onto the stack. The processor then loads an *interrupt vector* into the program counter. The interrupt vector is the address at which an *interrupt service routine* (*ISR*) lies. Thus, loading the vector into the program counter causes the processor to begin execution of the ISR, performing whatever service the interrupting device required. The last instruction of an ISR is always a *Return from Interrupt* instruction. This causes the processor to reload its saved state (registers and program counter) from the stack and resume its original program. Interrupts are largely transparent to the original program. This means that the original program is completely "unaware" that the processor was interrupted, save for a lost interval of time.

Processors with *shadow registers* use these to save their current state, rather than pushing their register bank onto the stack. This saves considerable memory accesses (and therefore time) when processing an interrupt. However, since only one set of shadow registers exists, a processor servicing multiple interrupts must "manually" preserve the state of the registers before servicing the higher interrupt. If it does not, important state information will be lost. Upon returning from an ISR, the contents of the shadow registers are swapped back into the main register array.

Hardware interrupts

There are two ways of telling when an I/O device (such as a serial controller or a disk controller) is ready for the next sequence of data to be transferred. The first is *busy waiting* or *polling*, where the processor continuously checks the device's status register until the device is ready. This wastes the processor's time but is the simplest to implement. For some time-critical applications, polling can reduce the time it takes for the processor to respond to a change of state in a peripheral.

A better way is for the device to generate an interrupt to the processor when it is ready for a transfer to take place. Small, simple processors may only have one (or two) interrupt inputs, so several external devices may have to share the interrupt lines of the processor. When an interrupt occurs, the processor must check each device to determine which one generated the interrupt. (This can also be considered a form of polling.) The advantage of interrupt polling over ordinary polling is that the polling occurs only when there is a need to service a device. Polling interrupts is suitable only in systems that have a small number of devices; otherwise, the processor will spend too long trying to determine the source of the interrupt.

The other technique of servicing an interrupt is by using *vectored interrupts*,* by which the interrupting device provides the interrupt vector that the processor is to

* Note that this is different from an interrupt vector stored in memory.

take. Vectored interrupts reduce considerably the time it takes the processor to determine the source of the interrupt. If an interrupt request can be generated from more than one source, it is therefore necessary to assign priorities (levels) to the different interrupts. This can be done in either hardware or software, depending on the particular application. In this scheme, the processor has numerous interrupt lines, with each interrupt corresponding to a given interrupt vector. So, for example, when an interrupt of priority 7 occurs (interrupt lines corresponding to "7" are asserted), the processor loads vector 7 into its program counter and starts executing the service routine specific to interrupt 7.

Vectored interrupts can be taken one step further. Some processors and devices support the device by actually placing the appropriate vector onto the data bus when they generate an interrupt. This means the system can be even more versatile, so that instead of being limited to one interrupt per peripheral, each device can supply an interrupt vector specific to the event that is causing the interrupt. However, the processor must support this function, and most do not.

Some processors have a feature known as a *fast hardware interrupt*. With this interrupt, only the program counter is saved. It assumes that the ISR will protect the contents of the registers by manually saving their state as required. Fast interrupts are useful when an I/O device requires a very fast response from a processor and cannot wait for the processor to save all its registers to the stack. A special (and separate) interrupt line is used to generate fast interrupts.

Software interrupts

A software interrupt is generated by an instruction. It is the lowest-priority interrupt and is generally used by programs to request a service to be performed by the system software (operating system or firmware).

So why are software interrupts used? Why isn't the appropriate section of code called directly? For that matter, why use an operating system to perform tasks for us at all? It gets back to compatibility. Jumping to a subroutine (calling a function) is jumping to a specific address in memory. A future version of the system software may not locate the subroutines at the same addresses as earlier versions. By using a software interrupt, our program does not need to know where the routines lie. It relies on the entry in the vector table to direct it to the correct location.

CISC and RISC

There are two major approaches to processor architecture: *Complex Instruction Set Computer* (*CISC*, pronounced "Sisk") processors and *Reduced Instruction Set Computer* (*RISC*) processors. Classic CISC processors are the Intel x86, Motorola 68xxx, and National Semiconductor 32xxx processors, and, to a lesser degree, the

Intel Pentium. Common RISC architectures are the Freescale/IBM PowerPC, the MIPS architecture, Sun's SPARC, the ARM, the Atmel AVR, and the Microchip PIC.

CISC processors have a single processing unit, external memory, and a relatively small register set and many hundreds of different instructions. In many ways, they are just smaller versions of the processing units of mainframe computers from the 1960s.

The tendency in processor design throughout the late 70s and early 80s was toward bigger and more complicated instruction sets. Need to input a string of characters from an I/O port? Well, with CISC (80x86 family), there's a *single instruction* to do it! The diversity of instructions in a CISC processor can run to well over 1,000 opcodes in some processors, such as the Motorola 68000. This had the advantage of making the job of the assembly-language programmer easier, since you had to write fewer lines of code to get the job done. As memory was slow and expensive, it also made sense to make each instruction do more. This reduced the number of instructions needed to perform a given function, and thereby reduced memory space and the number of memory accesses required to fetch instructions. As memory got cheaper and faster, and compilers became more efficient, the relative advantages of the CISC approach began to diminish. One main disadvantage of CISC is that the processors themselves get increasingly complicated as a consequence of supporting such a large and diverse instruction set. The control and instruction decode units are complex and slow, the silicon is large and hard to produce, and they consume a lot of power and therefore generate a lot of heat. As processors became more advanced, the overheads that CISC imposed on the silicon became oppressive.

A given processor feature when considered alone may increase processor performance but may actually decrease the performance of the total system, if it increases the total complexity of the device. It was found that by streamlining the instruction set to the most commonly used instructions, the processors become simpler and faster. Fewer cycles are required to decode and execute each instruction, and the cycles are shorter. The drawback is that more (simpler) instructions are required to perform a task, but this is more than made up for in the performance boost to the processor. For example, if both cycle time and the number of cycles per instruction are each reduced by a factor of four, while the number of instructions required to perform a task grows by 50%, the execution of the processor is sped up by a factor of eight.

The realization of this led to a rethink of processor design. The result was the RISC architecture, which has led to the development of very high-performance processors. The basic philosophy behind RISC is to move the complexity from the silicon to the language compiler. The hardware is kept as simple and fast as possible.

A given complex instruction can be performed by a sequence of much simpler instructions. For example, many processors have an `xor` (exclusive OR) instruction

for bit manipulation, and they also have a `clear` instruction to set a given register to zero. However, a register can also be set to zero by `xor`-ing it with itself. Thus, the separate `clear` instruction is no longer required. It can be replaced with the already present `xor`. Further, many processors are able to clear a memory location directly by writing a zero to it. That same function can be implemented by clearing a register and then storing that register to the memory location. The instruction to load a register with a literal number can be replaced with the instruction for clearing a register, followed by an `add` instruction with the literal number as its operand. Thus, six instructions (`xor`, `clear` *`reg`*, `clear` *`memory`*, `load` *`literal`*, `store`, and `add`) can be replaced with just three (`xor`, `store`, and `add`).

So the following CISC assembly pseudocode:

```
clear 0x1000    ; clear memory location 0x1000
load  r1,#5     ; load register 1 with the value 5
```

becomes the following RISC pseudocode:

```
xor   r1,r1      ; clear register 1
store r1,0x1000  ; clear memory location 0x1000
add   r1,#5      ; load register 1 with the value 5
```

The resulting code size is bigger, but the reduced complexity of the instruction decode unit can result in faster overall operation. Dozens of such code optimizations exist to give RISC its simplicity.

RISC processors have a number of distinguishing characteristics. They have large register sets (in some architectures numbering over 1,000), thereby reducing the number of times the processor must access main memory. Often-used variables can be left inside the processor, reducing the number of accesses to (slow) external memory. Compilers of high-level languages (such as C) take advantage of this to optimize processor performance.

By having smaller and simpler instruction decode units, RISC processors have fast instruction execution, and this also reduces the size and power consumption of the processing unit. Generally, RISC instructions will take only one or two cycles to execute (this depends greatly on the particular processor). This is in contrast to instructions for a CISC processor, whose instructions may take many tens of cycles to execute. For example, one instruction (integer multiplication) on an 80486 CISC processor takes 42 cycles to complete. The same instruction on a RISC processor may take just one cycle. Instructions on a RISC processor have a simple format. All instructions are generally the same length (which makes instruction decode units simpler).

RISC processors implement what is known as a "load/store" architecture. This means that the only instructions that actually reference memory are `load` and `store`. In contrast, many (most) instructions on a CISC processor may access or manipulate memory. On a RISC processor, all other instructions (aside from `load` and `store`) work on the registers only. This facilitates the ability of RISC processors to complete

(most of) their instructions in a single cycle. Consequently, RISC processors do not have the range of addressing modes that are found on CISC processors.

RISC processors also often have pipelined instruction execution. This means that while one instruction is being executed, the next instruction in the sequence is being decoded, while the third one is being fetched. At any given moment, several instructions will be in the pipeline and in the process of being executed. Again, this provides improved processor performance. Thus, even though not all instructions may be completed in a single cycle, the processor may issue and retire instructions on each cycle, thereby achieving effective single-cycle execution. Some RISC processors have overlapped instruction execution, in which `load` operations may allow the execution of subsequent, unrelated instructions to continue before the data requested by the `load` has been returned from memory. This allows these instructions to overlap the `load`, thereby improving processor performance.

Due to their low power consumption and computing power, RISC processors are becoming widely used, particularly in embedded computer systems, and many RISC attributes are appearing in what are traditionally CISC architectures (such as with the Intel Pentium). Ironically, many RISC architectures are adding some CISC-like features, and so the distinction between RISC and CISC is blurring.

An excellent discussion of RISC architectures and processor performance topics can be found in Kevin Dowd and Charles Severance's *High Performance Computing* (O'Reilly).

So, which is better for embedded and industrial applications, RISC or CISC? If power consumption needs to be low, then RISC is probably the better architecture to use. However, if the available space for program storage is small, then a CISC processor may be a better alternative, since CISC instructions get more "bang" for the byte.

Digital Signal Processors

A special type of processor architecture is that of the *Digital Signal Processor* (*DSP*). These processors have instruction sets and architectures optimized for numerical processing of array data. They often extend the Harvard architecture concept further by not only having separate data and code spaces, but also by splitting the data spaces into two or more banks. This allows concurrent instruction fetch and data accesses for multiple operands. As such, DSPs can have very high throughput and can outperform both CISC and RISC processors in certain applications.

DSPs have special hardware well suited to numerical processing of arrays. They often have *hardware looping*, whereby special registers allow for and control the repeated execution of an instruction sequence. This is also often known as *zero-overhead looping*, since no conditions need to be explicitly tested by the software as part of the looping process. DSPs often have dedicated hardware for increasing the speed of arithmetic operations. High-speed multipliers, Multiply-And-Accumulate (MAC) units, and barrel shifters are common features.

DSP processors are commonly used in embedded applications, and many conventional embedded microcontrollers include some DSP functionality.

Memory

Memory is used to hold data and software for the processor. There is a variety of memory types, and often a mix is used within a single system. Some memory will retain its contents while there is no power, yet will be slow to access. Other memory devices will be high-capacity, yet will require additional support circuitry and will be slower to access. Still other memory devices will trade capacity for speed, yielding relatively small devices, yet will be capable of keeping up with the fastest of processors.

Memory chips can be organized in two ways, either in *word-organized* or *bit-organized* schemes. In the word-organized scheme, complete nybbles, bytes, or words are stored within a single component, whereas with bit-organized memory, each bit of a byte or word is allocated to a separate component (Figure 1-8).

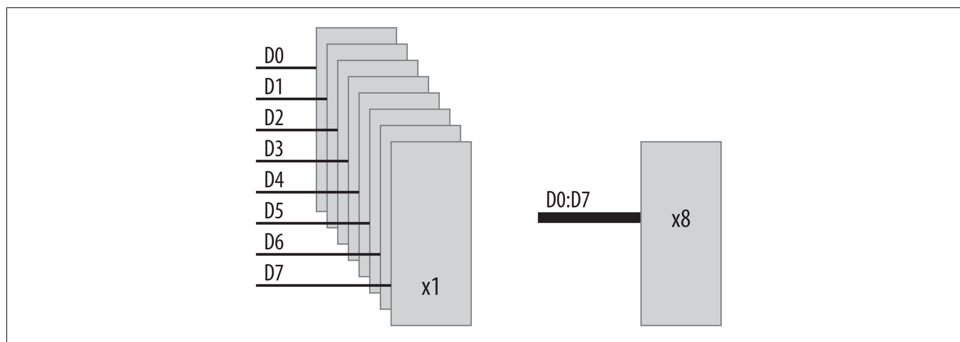


Figure 1-8. Eight bit-organized 8×1 devices and one word-organized 8×8 device

Memory chips come in different sizes, with the width specified as part of the size description. For instance, a DRAM (dynamic RAM) chip might be described as being 4M×1 (bit-organized), whereas a SRAM (static RAM) may be 512K×8 (word-organized). In both cases, each chip has exactly the same storage capacity, but organized in different ways. In the DRAM case, it would take eight chips to complete a memory block for an 8-bit data bus, whereas the SRAM would only require one chip. However, because the DRAMs are organized in parallel, they are accessed simultaneously. The final size of the DRAM block is (4M×1)×8 devices, which is 32M. It is common practice for multiple DRAMs to be placed on a *memory module*. This is the common way that DRAMs are installed in standard computers.

The common widths for memory chips are x1, x4, and x8, although x16 devices are available. A 32-bit-wide bus can be implemented with thirty-two x1 devices, eight x4 devices, or four x8 devices.

RAM

RAM stands for *Random Access Memory*. This is a bit of a misnomer, since most (all) computer memory may be considered "random access." RAM is the "working memory" in the computer system. It is where the processor may easily write data for temporary storage. RAM is generally *volatile*, losing its contents when the system loses power. Any information stored in RAM that must be retained must be written to some form of permanent storage before the system powers down. There are special nonvolatile RAMs that integrate a battery-backup system, such that the RAM remains powered even when the rest of the computer system has shut down.

RAMs generally fall into two categories: *static RAM* (also known as *SRAM*) and *dynamic RAM* (also known as *DRAM*).

SRAMs use pairs of logic gates to hold each bit of data. SRAMs are the fastest form of RAM available, require little external support circuitry, and have relatively low power consumption. Their drawbacks are that their capacity is considerably less than DRAM, while being much more expensive. Their relatively low capacity requires more chips to be used to implement the same amount of memory. A modern PC built using nothing but SRAM would be a *considerably* bigger machine and would cost a small fortune to produce. (It would be *very* fast, however.)

DRAM uses arrays of what are essentially capacitors to hold individual bits of data. The capacitor arrays will hold their charge only for a short period before it begins to diminish. Therefore, DRAMs need continuous refreshing, every few milliseconds or so. This perpetual need for refreshing requires additional support and can delay processor access to the memory. If a processor access conflicts with the need to refresh the array, the refresh cycle must take precedence.

DRAMs are the highest-capacity memory devices available and come in a wide and diverse variety of subspecies. Interfacing DRAMs to small microcontrollers is generally not possible, and certainly not practical. Most processors with large address spaces include support for DRAMs. Connecting DRAMs to such processors is simply a case of "connecting the dots" (or pins, as the case may be). For those processors that do not include DRAM support, special DRAM controller chips are available that make interfacing the DRAMs very simple indeed.

Many processors have instruction and/or data *caches*, which store recent memory accesses. These caches are (often, but not always) internal to the processors and are implemented with fast memory cells and high-speed data paths. Instruction execution normally runs out of the instruction cache, providing for fast execution. The processor is capable of rapidly reloading the caches from main memory should a cache miss occur. Some processors have logic that is able to anticipate a cache miss and begin the cache reload prior to the cache miss occurring. Caches are implemented using very fast SRAM and are most often used in large systems to compensate for the slowness of DRAM.

ROM

ROM stands for *Read-Only Memory*. This is also a bit of a misnomer, since many (modern) ROMs can also be written to. ROMs are *nonvolatile memory*, requiring no power to retain their contents. They are generally slower than RAM, and considerably slower than fast static RAM.

The primary purpose of ROM within a system is to hold the code (and sometimes data) that needs to be present at power-up. Such software is generally known as *firmware* and contains software to initialize the computer by placing I/O devices into a known state. It may contain either a bootloader program to load an operating system off disk or network or, in the case of an embedded system, it may contain the application itself.

Many microcontrollers contain on-chip ROM, thereby reducing component count and simplifying system design.

Standard ROM is fabricated (in a simplistic sense) from a large array of diodes. The unwritten bit state for a ROM is all 1s, each byte location reading as 0xFF. The process of loading software into a ROM is known as *burning the ROM*. This term comes from the fact that the programming process is performed by passing a sufficiently large current through the appropriate diodes to "blow them," or *burn* them, thereby creating a zero at that bit location. A device known as a *ROM burner* can accomplish this, or, if the system supports it, the ROM may be programmed in-circuit. This is known as *In-System Programming* (*ISP*) or, sometimes, *In-Circuit Programming* (*ICP*).

One-Time Programmable (*OTP*) ROMs, as the name implies, can be burned once only. Computer manufacturers typically use them in systems where the firmware is stable and the product is shipping in bulk to customers. *Mask-programmable* ROMs are also one-time programmable, but unlike OTPs, they are burned by the chip manufacturer prior to shipping. Like OTPs, they are used once the software is known to be stable and have the advantage of lowering production costs for large shipments.

EPROM

OTP ROMs are great for shipping in final products, but they are wasteful for debugging, since with each iteration of code change, a new chip must be burned and the old one thrown away. As such, OTPs make for a very expensive development option. No sane person uses OTPs for development work.

A (slightly) better choice for system development and debugging is the *Erasable Programmable Read-Only Memory*, or EPROM. Shining ultraviolet light through a small window on the top of the chip can erase the EPROM, allowing it to be reprogrammed and reused. They are pin- and signal-compatible with comparable OTP

and mask devices. Thus, an EPROM can be used during development, while OTPs can be used in production with no change to the rest of the system.

EPROMs and their equivalent OTP cousins range in capacity from a few kilobytes (exceedingly rare these days) to a megabyte or more.

The drawback with EPROM technology is that the chip must be removed from the circuit to be erased, and the erasure can take many minutes to complete. The chip is then inserted into the burner, loaded with software, and then placed back in-circuit. This can lead to very slow debug cycles. Further, it makes the device useless for storing changeable system parameters. EPROMs are relatively rare these days. You can still buy them, but flash-based memory (to be discussed shortly) is far more common and is the medium of choice.

EEROM

EEROM is *Electrically Erasable Read-Only Memory*, also known as *EEPROM* (*Electrically Erasable Programmable Read-Only Memory*). Very rarely, it is also called *Electrically Alterable Read-Only Memory* (*EAROM*). EEROM can be pronounced as either "e-e ROM" or "e-squared ROM," or sometimes just "e-squared" for short.

EEROMs can be erased and reprogrammed in-circuit. Their capacity is significantly smaller than standard ROM (typically only a few kilobytes), and so they are not suited to holding firmware. Instead, they are typically used for holding system parameters and mode information to be retained during power-off.

It is common for many microcontrollers to incorporate a small EEROM on-chip for holding system parameters. This is especially useful in embedded systems and may be used for storing network addresses, configuration settings, serial numbers, servicing records, and so on.

Flash

Flash is the newest ROM technology and is now dominant. Flash memory has the reprogrammability of EEROM and the large capacity of standard ROMs. Flash chips are sometimes referred to as "flash ROMs" or "flash RAMs." Since they are not like standard ROMs or standard RAMs, I prefer just to call them "flash" and save on the confusion.

Flash is normally organized as sectors and has the advantage that individual sectors may be erased and rewritten without affecting the contents of the rest of the device. Typically, before a sector can be written, it must be erased. It can't just be written over as with a RAM.

There are several different flash technologies, and the erasing and programming requirements of flash devices vary from manufacturer to manufacturer.

Input/Output

The address space of the processor can contain devices other than memory. These are input/output devices (I/O devices, also known as *peripherals*) and are used by the processor to communicate with the external world. Some examples are serial controllers that communicate with keyboards, mice, modems, etc.; parallel I/O devices that control some external subsystem; or disk-drive controllers, video and audio controllers, or network interfaces.

There are three main ways in which data may be exchanged with the external world:

Programmed I/O
: The processor accepts or delivers data at times convenient to it (the processor).

Interrupt-driven I/O
: External events control the processor by requesting the current program be suspended and the external event be serviced. An external device will interrupt the processor (assert an interrupt control line into the processor), at which time the processor will suspend the current task (program) and begin executing an interrupt service routine. The service of an interrupt may involve transferring data from input to memory, or from memory to output.

Direct Memory Access (DMA)
: DMA allows data to be transferred from I/O devices to memory directly without the continuous involvement of the processor. DMA is used in high-speed systems, where the rate of data transfer is important. Not all processors support DMA.

DMA

DMA is a way of streamlining transfers of large blocks of data between two sections of memory, or between memory and an I/O device. Let's say you want to read in 100M from disk and store it in memory. You have two options.

One option is for the processor to read one byte at a time from the disk controller into a register and then store the contents of the register to the appropriate memory location. For each byte transferred, the processor must read an instruction, decode the instruction, read the data, read the next instruction, decode the instruction, and then store the data. Then the process starts over again for the next byte.

The second option in moving large amounts of data around the system is DMA. A special device, called a *DMA Controller* (*DMAC*), performs high-speed transfers between memory and I/O devices. Using DMA bypasses the processor by setting up a *channel* between the I/O device and the memory. Thus, data is read from the I/O device and written into memory without the need to execute code to perform the transfer on a byte-by-byte (or word-by-word) basis.

In order for a DMA transfer to occur, the DMAC must have use of the address and data buses. There are several ways in which this could be implemented by the system designer. The most common approach (and probably the simplest) is to suspend the operation of the processor and for the processor to "release" its buses (the buses are tristate). This allows the DMAC to "take over" the buses for the short period required to perform the transfer. Processors that support DMA usually have a special control input that enables a DMAC (or some other processor) to request the buses.

There are four basic types of DMA:

Standard block transfer
: Accomplished by the DMA controller performing a sequence of memory transfers. The transfers involve a load operation from a source address followed by a store operation to a destination address. Standard block transfers are initiated under software control and are used for moving data structures from one region of memory to another.

Demand-mode transfers
: Similar to standard mode except that the transfer is controlled by an external device. Demand-mode transfers are used to move data between memory and I/O or vice versa. The I/O device requests and synchronizes the movement of data.

Fly-by transfer
: Provides high-speed data movement in the system. Instead of using multiple bus accesses as with conventional DMA transfers, fly-by transfers move data from source to destination in a single access. The data is not read into the DMAC before going to its destination. During a fly-by transfer, memory and I/O are given different bus control signals. For example, an I/O device is given a read request at the same time that memory is given a write request. Data moves from the I/O device straight into the memory device.

Data-chaining transfers
: Allow DMA transfers to be performed as specified by a linked-list in memory. Data chaining is started by specifying a pointer to a *descriptor* in memory. The descriptor is a table specifying byte count, source address, destination address, and a pointer to the next descriptor. The DMAC loads the relevant information about the transfer from this table and begins moving data. The transfer continues until the number of bytes transferred is equal to the entry in the byte-count field. On completion, the pointer to the next descriptor is loaded. This continues until a null pointer is found.

To illustrate the use of DMA, let's consider the example of a fly-by transfer of data from a hard-disk controller to memory. A DMA transfer begins by the processor configuring the DMAC for the transfer. This setup involves specifying the source, destination, and size of the data, as well as other parameters. The disk controller generates a request for service to the DMAC (not the processor). The DMAC then

generates a HOLD or BR (bus request) to the processor. The processor completes the current instruction; places the address, control, and data buses in a high-impedance state (*floats*, tristates, or releases them); and responds to the DMAC with a HOLD-acknowledge or BG (bus granted) and enters a dormant state. Upon receiving a HOLD-acknowledge, the DMAC places the address of the memory location where the transfer to memory will begin onto the address bus and generates a WRITE to the memory while the disk controller places the data on the data bus. Hence, a direct memory access is accomplished from the disk controller to the memory.

In a similar fashion, transfers from memory to I/O devices are also possible. DMACs are capable of handling block transfers of data. The DMAC automatically increments the address on the address bus to point to each successive memory location as the I/O device generates (or receives) data. Once the transfer is complete, the buses are returned to the processor and it resumes normal operation.

Not all DMA controllers support all forms of DMA. Some DMA controllers simply read data from a source, hold it internally, and then store it to a destination. They perform the transfer in exactly the same way that a processor would. The advantage in using a DMA controller instead of a processor is that if the transfer were to be performed by the processor, each transfer would still have program fetches associated with it. Thus, even though the transfer takes place by sequential reads and writes, the DMA controller does not also have to fetch and execute code, thereby providing a faster transfer than a processor.

Support for DMA is normally not found in small microcontrollers. Some mid-range processors (16-bit, low-end 32-bit) may have DMA support. All high-end processors (32-bit and above) will have DMA support, and many include a DMA controller on-chip. Similarly, peripherals intended for small-scale computers will not provide DMA support, whereas peripherals intended for high-speed and powerful computers definitely *will* have DMA support.

Parallel and Distributed Computers

Some embedded applications require greater performance than is achievable from a single processor. For cost reasons, it may not be practical to implement a design with the latest superscalar RISC processor, or perhaps the application lends itself to distributed processing where the tasks are run across several communicating machines. It may make more sense to use a fleet of lower-cost processors, distributed throughout the installation. It is becoming increasingly common to see embedded systems implemented using parallel processors.

Introduction to parallel architectures

The traditional architecture for computers follows the conventional, Von Neumann serial architecture. Computers based on this form usually have a single, sequential