

# Computers as Components

## Principles of Embedded Computing System Design

*Fifth Edition*

**Marilyn Wolf**



Computers as Components  
Principles of Embedded Computing System Design  
Fifth Edition

Morgan Kaufmann is an imprint of Elsevier  
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Copyright © 2023 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: [www.elsevier.com/permissions](http://www.elsevier.com/permissions).

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

#### Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-323-85128-2

For information on all Morgan Kaufmann publications visit our website at  
<https://www.elsevier.com/books-and-journals>

*Publisher:* Katey Birtcher  
*Acquisitions Editor:* Steve Merken  
*Editorial Project Manager:* Chris Hockaday  
*Production Project Manager:* Janish Paul  
*Cover Designer:* Mark Rogers

Printed in India

9 8 7 6 5 4 3 2 1



Working together  
to grow libraries in  
developing countries

[www.elsevier.com](http://www.elsevier.com) • [www.bookaid.org](http://www.bookaid.org)

*For Alec*

# Foreword to the First Edition

Digital system design has entered a new era. At a time when the design of microprocessors has shifted into a classical optimization exercise, the development of embedded computing systems in which microprocessors are merely components has become a wide-open frontier, examples of which include wireless systems, wearable systems, networked systems, smart appliances, industrial process systems, advanced automotive systems, and biologically interfaced systems.

Driven by advances in sensors, transducers, microelectronics, processor performance, operating systems, communications technology, user interfaces, and packaging technology, as well as by a deeper understanding of human needs and market possibilities, a vast new range of systems and applications is emerging. Thus, the architects and designers of embedded systems are currently tasked with making these possibilities a reality.

However, at present, embedded system design is practiced as a craft. Although knowledge regarding the component hardware and software subsystems is abundant, no system design methodologies for orchestrating the overall design process are in common use, and embedded system design is still run in an ad-hoc manner in most projects.

Several challenges in embedded system design arise from changes in the underlying technology, and the subtleties of how it can all be correctly mingled and integrated. Other difficulties are a result of new and often unfamiliar types of system requirements. The significant improvements in infrastructure and technology for communication and collaboration have opened up unprecedented possibilities for a quick design response to market needs. However, effective design methodologies and associated design tools haven't been made available for rapidly following up on these opportunities.

At the beginning of the VLSI era, transistors and wires were the fundamental components, and the rapid design of computers on a chip was the dream. Today, the CPU as well as various specialized processors and subsystems are merely basic components, and the rapid, effective design of very complex embedded systems is the goal. System specifications have become much more complex and they must also meet real-time deadlines, consume little power, effectively support complex real-time user interfaces, be very cost-competitive, and be designed to be upgradable.

Wayne Wolf has created the first textbook for systematically dealing with this array of new system design requirements and challenges. He presents formalisms and a methodology for embedded system design that can be employed by the new type of tall-thin system architect who truly understands the foundations of system design across a very wide range of its component technologies.

Moving from the basics of each technology dimension, Wolf presents formalisms for specifying and modeling system structures and behaviors, and subsequently clarifies these ideas through a series of design examples. He explores the complexities that are involved and how to deal with them systematically. You will emerge with a sense

of clarity regarding the nature of the design challenges ahead, and with knowledge of the key methods and tools for tackling these challenges.

As the first textbook on embedded system design, this book will prove to be invaluable as a means for acquiring knowledge in this important and newly emerging field. It will also serve as a reference in actual design practice, and will be a trusted companion in future design adventures. I highly recommend it to you.

**Lynn Conway**  
*Professor Emerita, Electrical Engineering and  
Computer Science, University of Michigan*

# Preface to the First Edition

Although microprocessors have long been a part of our lives, they have only become sufficiently powerful to take on truly sophisticated functions in the past few years. The result of this explosion in microprocessor power, driven by Moore's Law, is the emergence of embedded computing as a discipline. In the early days of microprocessors, when all of the components were relatively small and simple, it was necessary and desirable to focus on individual instructions and logic gates. At present, with systems containing tens of millions of transistors as well as tens of thousands of lines of high-level language code, design techniques that help us to deal with complexity must be used.

This book attempts to capture some of the basic principles and techniques of this new discipline of embedded computing. Certain challenges of embedded computing are well known in the desktop computing world. For example, a careful analysis of program traces is often required to achieve the highest performance from pipelined, cached architectures. Similarly, with the increasing complexity of embedded systems, the techniques that have been developed in software engineering for specifying complex systems have become important. Another example is the design of systems with multiple processes. The requirements are very different on a desktop general-purpose operating system and a real-time operating system; the real-time techniques that have been developed for larger real-time systems over the past 30 years are now finding common use in microprocessor-based embedded systems.

Other challenges in embedded computing are new. One good example is power consumption: although it has not been a major consideration in traditional computer systems, it is an essential concern for battery-operated embedded computers and is important in many situations in which the power supply capacity is limited by weight, cost, or noise. Another issue is deadline-driven programming. Embedded computers often impose hard deadlines on completion times for programs; this type of constraint is rare in the desktop world. As embedded processors have become faster, caches and other CPU elements have also made the execution times less predictable. However, through careful analysis and clever programming, embedded programs with predictable execution times can be designed even in the face of unpredictable system components such as caches.

Fortunately, many tools are available for dealing with the challenges that are presented by complex embedded systems, including high-level languages, program performance analysis tools, processes, and real-time operating systems. However, understanding how all of these tools work together is in itself a complex task. This book takes a bottom-up approach to understanding embedded system design techniques. By first grasping the fundamentals of microprocessor hardware and software, powerful abstractions can be built to aid us in creating complex systems.

## A note to embedded system professionals

This book is not a manual for understanding a particular microprocessor. Why should the techniques presented here be of interest to you? There are two reasons. First, techniques such as high-level language programming and real-time operating systems are very important in the creation of large, complex embedded systems that actually work. The industry is littered with failed system designs that did not work because their designers attempted to hack their way out of problems, rather than stepping back and taking a wider view of the issue. Second, the components that are used to build embedded systems are constantly changing, but the principles remain constant. Once the basic principles that are involved in the creation of complex embedded systems are understood, a new microprocessor (or even programming language) can rapidly be learned and the same fundamental principles can be applied to new components.

## A note to teachers

The traditional microprocessor system design class originated in the 1970s, when microprocessors were exotic yet relatively limited. That traditional class emphasizes breadboarding hardware and software to build a complete system. As a result, it focuses on the characteristics of a particular microprocessor, including its instruction set and bus interface.

This book takes a more abstract approach to embedded systems. Although I have taken the opportunity to discuss real components and applications, this book is fundamentally not a microprocessor data book. As a result, its approach may initially appear unfamiliar. Rather than focusing on the particulars, the book attempts to study more generic examples to develop more generally applicable principles. However, I believe that this approach is both fundamentally easier to teach and more useful to students in the long run. It is easier because one can rely less on complex laboratory setups and spend more time on pencil-and-paper exercises, simulations, and programming exercises. It is more useful to students because their eventual work in this area will almost certainly use different components and facilities than those used at your school. Once students learn the fundamentals, it is much easier for them to learn the details of new components.

Hands-on experience is essential in gaining physical intuition regarding embedded systems. A certain amount of hardware building experience is very valuable; I believe that every student should recognize the smell of burning plastic integrated circuit packages. However, I urge you to avoid the tyranny of hardware building. If you spend too much time building a hardware platform, you will not have sufficient time to write interesting programs for it. Furthermore, as a practical matter, most classes do not have the time to allow students to build sophisticated hardware platforms with high-performance I/O devices, and possibly, multiple processors. A great deal can be learned about hardware by measuring and evaluating an existing hardware platform. The experience of programming complex embedded systems will teach students

a substantial amount about hardware, and debugging interrupt-driven code is an experience that few students are likely to forget.

The homepage for the book (<https://educate.elsevier.com/book/details/9780323851282>) includes the overheads, instructor's manual, laboratory materials, links to related websites, and a link to a password-protected ftp site that contains solutions to the exercises.

---

## Acknowledgments

I owe a word of thanks to many people who helped me in the preparation of this book. Several people offered advice in various aspects of the book: Steve Johnson (Indiana University) on specification, Louise Trevillyan and Mark Charney (both IBM Research) on program tracing, Margaret Martonosi (Princeton University) on cache miss equations, Randy Harr (Synopsys) on low power, Phil Koopman (Carnegie Mellon University) on distributed systems, Joerg Henkel (NEC C&C Labs) on low-power computing and accelerators, Lui Sha (University of Illinois) on real-time operating systems, John Rayfield (ARM) on the ARM architecture, David Levine (Analog Devices) on compilers and SHARC, and Con Korikis (Analog Devices) on SHARC. Many people served as reviewers at various stages: David Harris (Harvey Mudd College); Jan Rabaey (University of California at Berkeley); David Nagle (Carnegie Mellon University); Randy Harr (Synopsys); Rajesh Gupta, Nikil Dutt, Frederic Doucet, and Vivek Sinha (University of California at Irvine); Ronald D. Williams (University of Virginia); Steve Sapiro (SC Associates); Paul Chow (University of Toronto); Bernd G. Wenzel (Eurostep); Steve Johnson (Indiana University); H. Alan Mantooth (University of Arkansas); Margarida Jacome (University of Texas at Austin); John Rayfield (ARM); David Levine (Analog Devices); Ardsher Ahmed (University of Massachusetts/Dartmouth University); and Vijay Madisetti (Georgia Institute of Technology). I also owe a big word of thanks to my editor, Denise Penrose. Denise put a great deal of effort into finding and talking to potential users of this book to help us to understand what readers wanted to learn. This book owes a great deal to her insight and persistence. Cheri Palmer and her production team did an excellent job on an impossibly tight schedule. The mistakes and miscues are, of course, all mine.

# Preface to the Second Edition

Embedded computing is more important today than it was in 2000, when the first edition of this book appeared. Embedded processors have appeared in even more products, ranging from toys to airplanes. Systems-on-chips now use up to hundreds of CPUs and the cell phone is on its way to becoming the new standard computing platform. As indicated in my column in *IEEE Computer* in September 2006, there are at least half a million embedded systems programmers in the world today; probably closer to 800,000.

In this edition, I have attempted to both update and revamp. One major change is that the book now uses the TI C55x DSP. I have thoroughly rewritten the discussion on real-time scheduling. I have attempted to expand on performance analysis as a theme at as many levels of abstraction as possible. Given the importance of multiprocessors in even the most mundane embedded systems, this edition also presents a more general view of hardware/software co-design and multiprocessors.

One of the changes in the field is that this material is being taught at increasingly lower levels of the curriculum. What used to be graduate material is now upper-division undergraduate material, and some of this material will percolate down to the sophomore level in the foreseeable future. I believe that you can use subsets of this book to cover both more advanced and more basic courses. Certain advanced students may not need the background material of the earlier chapters and can spend more time on software performance analysis, scheduling, and multiprocessors. When teaching introductory courses, software performance analysis is an alternative path to exploring microprocessor architectures as well as software; such courses can focus on the first few chapters.

The new website for this book and my other books is <http://www.waynewolf.com>. You can find overheads for the material in this book, suggestions for labs, and links to more information on embedded systems on this site.

---

## Acknowledgments

I would like to thank a number of people who have helped me with this second edition. Cathy Wicks and Naser Salameh of Texas Instruments offered invaluable help in understanding the C55x. Richard Barry of [freeRTOS.org](http://freeRTOS.org) not only graciously allowed me to quote from the source code of his operating system, but also helped to clarify the explanation of the code. My editor at Morgan Kaufmann, Chuck Glaser, knew when to be patient, when to be encouraging, and when to be cajoling. (He also has great taste in sushi restaurants.) And of course, Nancy and Alec patiently let me type away. Any problems, small or large, with this book are, of course, solely my responsibility.

**Wayne Wolf**  
Atlanta, GA

# Preface to the Third Edition

This third edition reflects the continued evolution of my thoughts on embedded computing and suggestions of the users of this book. One important goal was expanding the coverage of embedded computing applications. Learning about topics such as digital still cameras and cars requires a lot of effort. Hopefully this material will provide some useful insight into the parts of these systems that most directly affect the design decisions faced by embedded computing designers. I have also expanded the range of processors that are used as examples. I have included sophisticated processors, including the TI C64x and advanced ARM extensions. Furthermore, I have included the PIC16F to illustrate the properties of small RISC embedded processors. Finally, I have reorganized the coverage of networks and multiprocessors to provide a more unified view of these closely related topics. You can find additional material on the course website at <http://www.marilynwolf.us>. The site includes a complete set of overheads, sample labs, and pointers to additional information.

I would like to thank Nate McFadden, Todd Green, and Andre Cuello for their editorial patience and care during this revision. I would also like to thank the anonymous reviewers and Prof. Andrew Pleszkun of the University of Colorado for their insightful comments on the drafts. I must extend a special thanks to David Anderson, Phil Koopman, and Bruce Jacob who helped me to figure certain things out. I would also like to thank the Atlanta Snowpocalypse of 2011 for providing me with a large block of uninterrupted writing time.

Most important, this is the right time to acknowledge the profound debt of gratitude I owe to my father. He taught me how to work: not just how to do certain things, but how to approach problems, develop ideas, and bring them to fruition. Along the way, he taught me how to be a considerate, caring human being. Thanks, Dad.

**Marilyn Wolf**  
Atlanta, GA  
December 2011

# Preface to the Fourth Edition

Preparing this fourth edition of *Computers as Components* has made me realize just how old I am. I completed the final draft of the first edition in late 1999. Since then, embedded computing has evolved considerably, but the core principles remain. I have made changes throughout the book: fixing problems, improving presentations, in some cases reordering material to improve the flow of ideas, and deleting a few small items. Hopefully, these changes have improved the book.

The two biggest changes are the addition of a chapter on the Internet-of-Things (IoT) and the coverage of safety and security throughout the book. IoT has emerged as an important topic since the third edition was published, but it builds on existing technologies and themes. The new IoT chapter reviews several wireless networks that are used in IoT applications. It also presents several models for the organization of IoT systems. Safety and security have long been important issues in embedded computing—the first edition of this book discussed medical device safety—but a series of incidents have highlighted the critical nature of this topic.

In previous editions, advanced topics were covered in [Chapter 8](#), which included both multiprocessor systems-on-chips and networked embedded systems. This material has been expanded and separated into three chapters: the IoT chapter ([Chapter 8](#)) covers the material on OSI and Internet protocols as well as IoT-specific topics; a chapter on automobiles and airplanes ([Chapter 9](#)) explores networked embedded systems within the context of vehicles, in addition to covering several examples of safety and security; and the embedded multiprocessor chapter ([Chapter 10](#)) deals with multiprocessor systems-on-chips and their applications.

As always, overheads are available on the book website at <http://www.marilynwolf.us>. Several pointers to outside web material also appear on this website, whereas my new blog, <http://embeddedcps.blogspot.com/>, provides a stream of posts on topics of interest to embedded computing people.

I would like to thank my editor, Nate McFadden, for his help and guidance. Any deficiencies in the book are, of course, the result of my own failings.

**Marilyn Wolf**  
Atlanta, GA  
November 2015

# Preface to the Fifth Edition

This fifth edition of *Computers as Components* becomes available directly after the book's twentieth anniversary. I completed the final draft of the first edition in late 1999. By that time, embedded computing had emerged as a new aspect of computing. The field has matured into an important area of computer systems, as embedded computers surround us and help us in almost every conceivable aspect of our lives.

Looking back, I think that several early decisions held up well: ARM, UML, and real-time computing. Low-power computing is even more important now than it was for the first edition. Furthermore, the Internet-of-Things has emerged as a vital application for embedded computing.

I have made changes throughout the book for this edition. I have added material on safety and security—physical safety and information security—throughout the book. I have also updated the discussions in many sections.

As always, overheads are available on the book website at <https://www.marilynwolf.com>. You can find related videos on my YouTube Embedded Systems Channel.

I would like to thank my editor, Stephen Merken, for his help and guidance. I would also like to thank Alice Grant, Michelle Fisher, Paul Janish, and Chris Hockaday as well as their production team for their care and expertise. I greatly appreciate the thoughtful and detailed comments of the reviewers. Any deficiencies in the book are, of course, the result of my own failings.

**Marilyn Wolf**  
Lincoln, NE  
March 2022

# Embedded Computing

# 1

## CHAPTER POINTS

---

- Why we embed microprocessors in systems.
- What is difficult and unique about embedding computing?
- Real-time and low-power computing.
- Embedded computing as the foundation for Internet-of-Things (IoT) systems and cyber-physical systems (CPS).
- Design methodologies.
- Unified Modeling Language (UML).
- A guided tour of this book.

---

## 1.1 Introduction

For us to understand how to design embedded computing systems, we first need to understand the use cases—how and why microprocessors are used for control, user interface, signal processing, and many other tasks. The **microprocessor** has become so common that it is easy to forget how hard some things are to do without it.

We first review the various uses of microprocessors. We then review the major reasons why microprocessors are used in system design—delivering complex behaviors, fast design turnaround, and so on. Next, in [Section 1.2](#), we walk through the design of a simple example to understand the major steps in designing an embedded computing system. [Section 1.3](#) includes an in-depth look at techniques for specifying embedded systems; we use these specification techniques throughout the book. In [Section 1.4](#), we use a model train controller as an example for applying these specification techniques. [Section 1.5](#) provides a chapter-by-chapter tour of the book.

---

## 1.2 Complex systems and microprocessors

We tend to think of our laptop as a computer, but it is really one of many types of computer systems. A computer is a stored program machine that fetches and executes

instructions from a memory device. We can attach different types of devices to the computer, load the memory with different types of software, and build many types of systems ranging from simple appliances to complex machines, like robots.

What is an **embedded computer system**? Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer. Thus a PC is not an embedded computing system. However, a thermometer built from a microprocessor is an embedded computing system.

This means that embedded computing system design is a useful skill for many types of products. Automobiles, medical devices, and even household appliances make extensive use of microprocessors. Designers in many fields must be able to identify where microprocessors can be used, design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing. Computer engineering, like mechanical design or thermodynamics, is a fundamental discipline that can be applied in many domains. However, embedded computing system design does not stand alone. Many of the challenges encountered in the design of an embedded computing system are not computer engineering. For example, they may be mechanical or analog electrical problems. In this book, we are primarily interested in the embedded computer itself; therefore we concentrate on the hardware and software that enable the desired functions in the final product.

### 1.2.1 Embedding computers

Microprocessors come in many levels of sophistication; they are usually classified by their word size. A **microcontroller** is a complete computer system on a chip: CPU, memory, I/O devices. An 8-bit microcontroller is designed for low-cost applications and includes on-board memory and I/O devices; a 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory; and a 32-bit or 64-bit reduced instruction set computer (**RISC**) microprocessor offers very high performance for computation-intensive applications.

Given the wide variety of microprocessor types available, it should be no surprise that microprocessors are used in many ways. There are many household uses of microprocessors and microcontrollers. The typical microwave oven has at least one microprocessor to control oven operation. Many houses have advanced thermostat systems that change the temperature level at various times during the day. The modern camera is a prime example of the powerful features that can be added under microprocessor control.

Digital televisions make extensive use of embedded processors. In some cases, specialized CPUs are designed to execute important algorithms, such as audio or video algorithms.

Many of today's cars operate with over 100 million lines of code [Zax12, Owe15]. The Ford F-150 operates with 150 million lines [Sar16]. A high-end automobile may have 100 microprocessors, but even inexpensive cars today use around 40. Some of these microprocessors do very simple things, such as detect whether seat belts are in use. Others control critical functions, such as ignition and braking systems. The big push toward microprocessor-based engine control came from two nearly simultaneous

developments: The oil shock of the 1970s caused consumers to place much higher value on fuel economy, and fears of pollution resulted in laws restricting automobile engine emissions. The combination of low fuel consumption and low emissions is very difficult to achieve. To meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.

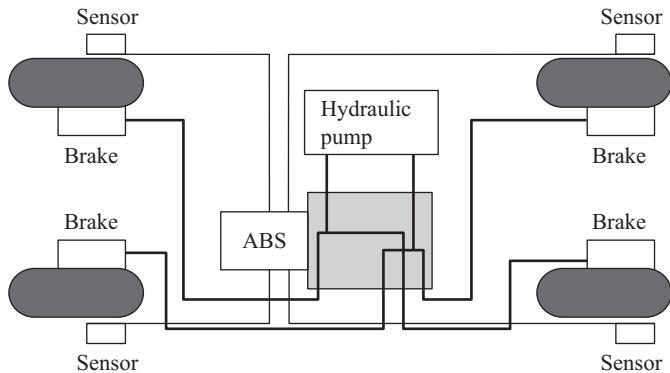
Braking is another important automotive subsystem that uses microprocessors to provide advanced capabilities. Design Example 1.1 describes some of the microprocessors used in BMW 850i.

---

### Design Example 1.1: BMW 850i Brake and Stability Control System

The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car. An antilock brake system (ABS) reduces skidding by pumping the brakes. An automatic stability control + traction (ASC+T) system intervenes with the engine during maneuvering to improve the car's stability. These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.

Consider the ABS. The purpose of an ABS is to temporarily release the brake on a wheel so that the wheel does not lose traction and skid. It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves, as shown in the accompanying diagram. This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking. The ABS system uses sensors on each wheel to measure its speed of rotation. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent skidding.



The ASC+T system's job is to control the engine power and the brake to improve the car's stability during maneuvers. The ASC+T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting. The ASC+T can be turned off by the driver, which can be important when operating with tire snow chains.

The ABS and ASC+T must clearly communicate because the ASC+T interacts with the brake system. Since the ABS was introduced several years earlier than the ASC+T, it was important to be able to interface ASC+T to the existing ABS module and to other existing electronic modules. The engine and control management units include the electronically controlled throttle, digital engine management, and electronic transmission control. The ASC+T control unit has two microprocessors on two printed circuit boards: one of which concentrates on logic-relevant components and the other concentrates on performance-specific components.

### 1.2.2 Characteristics of embedded computing applications

Embedded computing is, in many ways, much more demanding than the sort of programs that you may have written for PCs or workstations. Functionality is important in both general-purpose and embedded computing, but embedded applications must meet many other constraints as well.

On the one hand, embedded computing systems must provide sophisticated functionality:

- *Complex algorithms*: The operations performed by the microprocessor may be very sophisticated. For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel consumption.
- *User interface*: Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options. The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

- *Real time*: Many embedded computing systems must perform in real time. Hence, if the data aren't ready by a certain deadline, the system breaks. In some cases, failure to meet a deadline is unsafe and can even endanger lives. In other cases, missing a deadline doesn't create safety problems but does create unhappy customers. Missed deadlines in printers, for example, can result in scrambled pages.
- *Multirate*: Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time. They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of **multirate** behavior. The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized. Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

Constraints of various sorts are also very important:

- *Manufacturing cost*: The total cost of building a system is very important in many cases. Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required, and the types of I/O devices.
- *Power and energy*: Power consumption directly affects the cost of hardware, because a larger power supply may be necessary. Energy consumption affects battery life, which is important in many applications. Heat dissipation and thermal behavior are critical constraints in many computing systems.
- *Size*: Physical size is an important constraint in many systems.

Finally, most embedded computing systems are designed by small teams on tight deadlines. The use of small design teams for microprocessor-based systems is a self-fulfilling prophecy. The fact that systems can be built with microprocessors by only a few people invariably encourages management to assume that all microprocessor-based systems can be built by small teams. Tight deadlines are facts of life in today's internationally competitive environment. However, building a product using embedded software makes a lot of sense: Hardware and software can be debugged somewhat independently, and design revisions can be made much more quickly.

### 1.2.3 Why use microprocessors?

There are many ways to design a digital system: custom logic, field-programmable gate arrays (FPGAs), and so on. Why use microprocessors? There are two answers:

- Microprocessors are a very efficient way to implement digital systems.
- Microprocessors make it easier to design families of products that can be built to provide various feature sets at different price points and be extended to provide new features to keep up with rapidly changing markets.

#### CPU<sup>s</sup> are flexible

The paradox of digital design is that using a predesigned instruction set processor may in fact result in faster implementation of your application than designing your own custom logic. It is tempting to think that the overhead of fetching, decoding, and executing instructions is so high that it cannot be recouped.

#### CPU<sup>s</sup> are efficient

But there are two factors that work together to make microprocessor-based designs fast. First, microprocessors execute programs very efficiently. Modern processors can execute at least one instruction per clock cycle most of the time, and high-performance processors can execute several instructions per cycle. Although there is overhead that must be paid for interpreting instructions, it can often be hidden by clever utilization of parallelism within the CPU.

#### CPU<sup>s</sup> are highly optimized

Second, microprocessor manufacturers spend a great deal of money to make their CPUs run very fast. They hire large teams of designers to tweak every aspect of the microprocessor to make it run at the highest possible speed. Few products can justify the dozens or hundreds of computer architects and chip designers customarily employed in the design of a single microprocessor. Chips designed by small design teams are less likely to be as highly optimized for speed (or power) as are microprocessors. They also utilize the latest manufacturing technology.

It is also surprising but true that microprocessors are very efficient utilizers of logic. The generality of a microprocessor and the need for a separate memory may suggest that microprocessor-based designs are inherently much larger than custom logic designs. However, in many cases, the microprocessor is smaller when size is measured in units of logic gates. When special-purpose logic is designed for a particular function, it cannot be used for other functions. A microprocessor, on the other hand, can be used for many algorithms simply by changing the program it executes. Because so many modern systems make use of complex algorithms and user interfaces, we would generally have to design many custom logic blocks to implement

all required functionality. Many of those blocks will often sit idle. For example, the processing logic may sit idle when user interface functions are performed. Implementing several functions on a single processor often makes much better use of the available hardware budget.

#### Programmability

Microprocessors provide substantial advantages and relatively small drawbacks. As a result, embedded computing is the best choice in a wide variety of systems. The programmability of microprocessors can be a substantial benefit during the design process. It allows program design to be separated (at least to some extent) from the design of the hardware on which programs will be run. While one team is designing the board that contains the microprocessor, I/O devices, memory, and so on, others can be writing programs.

#### Software as differentiator

Software is the principal differentiator in many products. In many product categories, competing products use one of a handful of chips as hardware platforms. Added software serves to differentiate these products by features.

#### How many platforms?

Why not use PCs for all embedded computing? Put another way, how many hardware **platforms** do we need for embedded computing systems? PCs are widely used and provide a very flexible programming environment. Components of PCs are, in fact, used in many embedded computing systems. However, several factors keep us from using the stock PC as a universal embedded computing platform: cost, physical size, and power consumption. A variety of embedded platforms has been developed for applications, such as automotive or motor control. Platforms, such as Arduino and Raspberry Pi, provide capable microcontrollers, a wide range of peripherals, and strong software development environments.

#### Real time

First, real-time performance requirements often drive us to different architectures. As we will see later in the book, real-time performance is often best achieved with multiprocessors. Heterogeneous multiprocessors are designed to match the characteristics of the application software that runs on them, providing performance improvements.

#### Low power and low cost

Second, low power and low cost also drive us away from PC architectures and toward multiprocessors. Personal computers are designed to satisfy a broad mix of computing requirements and to be very flexible. These features increase the complexity and price of the components. They also cause the processor and other components to use more energy to perform a given function. Custom embedded systems that are designed for an application, such as a cell phone, consume several orders of magnitude less power than PCs with equivalent computational performance, and they are considerably less expensive as well.

#### Physics of software

A central subject of this book is what we might call the **physics of software**. Computing is a physical act. Executing a program takes time and consumes energy. Software performance and energy consumption are very important properties when we are connecting our embedded computers to the real world. We need to understand the sources of performance and power consumption if we are to be able to design programs that meet our application's goals. Luckily, we don't need to optimize our programs by directly considering the flow of electrons in microelectronics. In many cases, we can make very high-level decisions about the structure of our programs

to greatly improve their real-time performance and power consumption. As much as possible, we want to make computing abstractions work for us as we work on the physics of our software systems.

### 1.2.4 Embedded computing, IoT systems, and Cyber-Physical Systems

The discipline of embedded computing gives us a set of tools that we can use to design computer systems that interact with the physical world. Two styles of system design have emerged to support the wide range of applications in which computers interact with physical objects: Internet-of-Things (IoT) systems and cyber-physical systems (CPS).

#### IoT systems

An IoT system is a set of sensors, actuators, and computation units connected by a network. The network often has wireless links but may also include wired links. The IoT system monitors, analyzes, evaluates, and acts.

#### Cyber-Physical Systems

A cyber-physical system uses computers to build controllers, such as feedback control systems. A networked control system, in which a set of interfaces of a physical machine communicates over a bus with a CPU, is an important example of a cyber-physical system.

#### IoT and CPS

IoT and CPS are related but distinct, with a gray zone in between. One way to distinguish them is by sample rate: IoT systems tend to operate at lower sample rates, while cyber-physical systems run at higher sample rates. As a result, IoT systems are often more physically distributed, for example, a manufacturing plant; CPSs are more tightly coupled, such as an airplane or automobile.

#### Edge computing

IoT and CPSs are both examples of **edge computing**. When our computer must respond quickly to events in the physical world, we often don't have time to send queries to a remote data center and wait for a response. Computing at the edge must be responsive and energy efficient. Edge devices must also communicate with other devices at the edge as well as cloud computing resources.

### 1.2.5 Safety and security

Two trends make safety and security major concerns for embedded system designers. Embedded computers are increasingly found in safety-critical and other important systems that people use every day: cars, medical equipment, and so on. Many of these systems are connected to the Internet, either directly or indirectly, through maintenance devices or hosts. Connectivity makes embedded systems much more vulnerable to attack by malicious people. The danger of unsafe operation makes security problems even more important.

#### Security

**Security** relates to a system's ability to prevent malicious attacks. Security was originally applied to information processing systems, such as banking systems. In these cases, we are concerned with the data stored in the computer system. Stuxnet which we discuss in [Section 7.6.1](#), shows that computer security vulnerabilities can open the door to attacks that can compromise the physical plant of a CPS, not just its information. Like dependability, security has several related concepts. **Integrity**

refers to the data's proper values; an attacker should not be able to change values. **Privacy** refers to the unauthorized release of data.

### Safety

**Safety** relates to the way in which energy is released or controlled [Koo10]. Breaches in security can cause embedded computers to operate a physical machine improperly, causing a safety problem. Poor system design can also cause similar safety problems. Safety is a vital concern for any computer connected to physical devices, whether the threats come from malicious external activity, poor design, or improper use.

### Safe and secure systems

Safety and security are related but distinct concepts [Wol18]. An insecure system may not necessarily present a safety problem, but the combination of these two dangers is particularly potent and is a novel aspect of embedded computing and CPSs. Security is traditionally associated with IT systems. A typical security breach does not directly endanger anyone's safety. Unauthorized disclosure of private information may, for example, allow the victim of a breach to be threatened, but a typical credit card database breach doesn't directly cause safety problems. Similarly, the safety of a traditional mechanical system is not directly related to any information about that system. Today, the situation is profoundly different; we need safe and secure systems. A poorly designed car can allow an attacker to install software and take over its operation. It may even cause the car to drive to a location at which the driver can be assaulted.

### Safety and security technologies

We need to deploy a combination of technologies to ensure the safety and security of our embedded systems:

- **Cryptography** provides mathematical tools, such as encryption, which allow us to protect information. We discuss cryptography in [Section 4.9](#).
- **Security protocols** make use of cryptography to provide functions, such as authenticating the source of a piece of software or the integrity of our system configuration. We discuss security protocols in [Section 5.11](#).
- **Safe and secure hardware architectures** are required to ensure that our cryptographic operations and security protocols are not compromised by adversaries and that operational errors are caught early. We discuss safe and secure hardware architectures in [Sections 3.8 and 4.9](#).

## 1.2.6 Challenges in embedded computing system design

We have high expectations of modern engineered devices and systems. Old-fashioned mechanical devices don't provide the user experience that is expected in the 21<sup>st</sup> century.

First, we expect our devices to be capable. They must be feature-rich, providing many options on what they can do and how to do them.

They must be very efficient. Battery-powered devices must go a long time between charges. They must be physically small, even though they are capable. They must be inexpensive, and they must be reliable. They must work as expected without failure.

for long periods. If they do break, they must do so safely, and they should adjust themselves as they operate to maintain their accuracy and effectiveness.

What barriers prevent us from taking advantage of the opportunities provided by microprocessors? In order to leverage the power of embedded computers, we must master several technical challenges.

Embedded computers and software need to operate in real time. The world doesn't stop for the computer to catch up. Computer hardware and software must be designed so that they respond promptly. Real-time operation often requires concurrency—the ability to perform several tasks at once. Concurrent software is difficult to design but necessary to support the natural concurrency of many mechanical systems.

Many embedded computer systems must also operate as low-power devices; they must efficiently use electrical energy. Battery-powered devices need to draw as little power from the battery as possible to maximize battery life. Even machines connected to the electrical grid must be efficient. Electric power costs money, and the heat generated by electric power consumption causes a new set of problems.

Embedded computing systems must operate safely and securely. Although information security is an important characteristic of all computer systems, the nature of embedded computing means that new types of security problems must be solved.

Luckily, we have several tools at our disposal to help us solve these challenges and unlock the potential of embedded computing.

**Performance analysis** helps us to ensure that our embedded computer systems meet their real-time requirements. We can analyze our software and the computer hardware on which it runs to determine how long is required to execute the given function. If the software is too slow, we have tools at our disposal to help identify the problems and propose solutions.

Similarly, **power analysis** helps us ensure that our embedded computer systems operate at acceptable levels of power consumption.

**Design methodology** (the steps we follow to turn our ideas into finished products) are very important to the success of an embedded system design project. Methodologies help us refine a design, identify challenges, and identify good ways to solve those challenges.

### 1.2.7 Performance of embedded computing systems

When we talk about the performance of PC programs, what do we really mean? Most programmers have a vague notion of performance. They want their program to run fast enough, and they may be worried about the asymptotic complexity of their program. Most general-purpose programmers use no tools designed to help them improve the performance of their programs.

Embedded system designers, in contrast, have a very clear performance goal in mind; their program must meet its **deadline**. At the heart of embedded computing is **real-time computing**, which is the science and art of programming to deadlines. The program receives its input data, and the deadline is the time at which a

#### Real-time and concurrent execution

#### Low-power embedded computing

#### Safety and security

#### Fundamental techniques

#### Performance in general-purpose computing

#### Performance in embedded computing

### Understanding real-time performance

computation must be finished. If the program doesn't produce the required output by the deadline, then the program doesn't work, even if the output that it eventually produces is functionally correct.

This notion of deadline-driven programming is at once simple and demanding. It isn't easy to determine whether a large, complex program running on a sophisticated microprocessor will meet its deadline. We need tools to help us analyze the real-time performance of embedded systems; we also must adopt programming disciplines and styles that make it possible to analyze these programs.

To understand the real-time behavior of an embedded computing system, we must analyze the system at several levels of abstraction. As we move through this book, we will work our way up from the lowest layers that describe components of the system up through the highest layers that describe the complete system. Those layers include:

- *CPU*: The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.
- *Platform*: The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.
- *Program*: The CPU sees only a small window of a program at any given time. We must consider the structure of the entire program to determine its overall behavior.
- *Task*: We generally run several programs simultaneously on a CPU, creating a **multitasking system**. The tasks interact with each other in ways that have profound implications for performance.
- *Multiprocessor*: Many embedded systems have more than one processor; they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

---

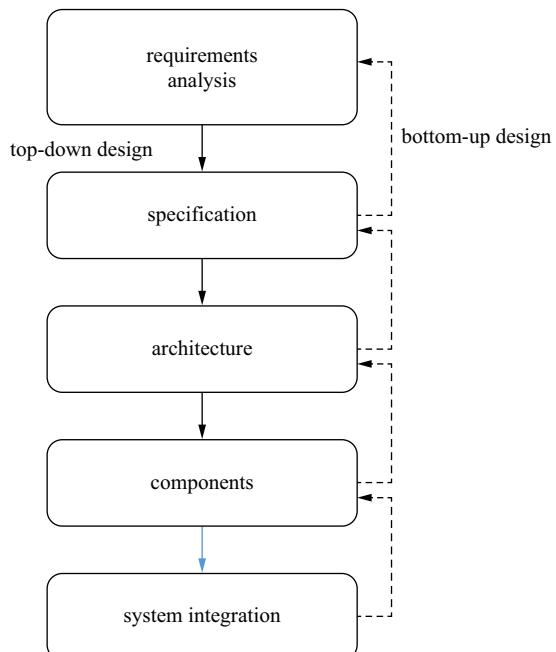
## 1.3 The embedded system design process

This section provides an overview of the embedded system design process aimed at two objectives. First, it introduces the various steps in embedded system design before we delve into them in more detail. Second, it allows us to consider the design **methodology** itself. A design methodology is important for three reasons. First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing **performance** or functional testing. Second, it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and produces a completed design would be a daunting task. However, by first breaking the process into manageable steps, we can work on automating (or at least semiautomating) the steps one at a time. Third, a design methodology makes it much easier for members of a design team to communicate. By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain

times, and what they are to hand off when they complete their assigned steps. Because most embedded systems are designed by teams, coordination is perhaps the most important role of a well-defined design methodology.

Fig. 1.1 summarizes the major steps in the embedded system design process. In this top-down view, we start with the system **requirements analysis** to capture some basic elements of the system. In the next step, **specification**, we create a more detailed, complete description of what we want. However, the specification states only how the system behaves, not how it is built. The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components. Once we know the components we need, we can design them, including both software modules and any specialized hardware. Based on those components, we can finally build a complete system.

In this section, we consider design from the **top down**. We begin with the most abstract description of the system and conclude with concrete details. The alternative is a **bottom-up** view in which we start with components to build a system. Bottom-up design steps are shown in Fig. 1.1 as dashed-line arrows. We need bottom-up design because we do not have perfect insight into how later stages of the design process will turn out. Decisions at one stage of design are based upon estimates of what will happen later: How fast can we make a particular function run? How much memory will we need? How much system bus capacity do we need? If our estimates are



**FIGURE 1.1**

Major levels of abstraction in the design process.

inadequate, we may have to backtrack and amend our original decisions to take the new facts into account. In general, the less experience we have with the design of similar systems, the more we will have to rely on bottom-up design information to help us refine the system.

The steps in the design process are only one axis along which we can view embedded system design. We also need to consider the major goals of the design:

- manufacturing cost
- performance (throughput, latency, and deadlines)
- power consumption

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:

- We must *analyze* the design at each step to determine how we can meet the specifications.
- We must then *refine* the design to add detail.
- We must verify the design to ensure that it still meets all system goals, such as cost and speed.

### 1.3.1 Requirements

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components.

A requirement is a need for the system; a specification is a complete set of requirements for the system. We proceed in two phases. First, we gather an informal description from the customers in a process known as requirements analysis. We then flesh out the requirements into a specification and a complete description of the requirements that contains enough information to begin designing the system architecture.

Separating requirements analysis and specification is often necessary owing to the large gap between what the customers can describe about the system they want and what the architects need to design the system. Consumers of embedded systems are usually not embedded systems or product designers. Their understanding of the system is based on how they envision user interactions with the system. They may have unrealistic expectations as to what can be done within their budgets, and they may also express their desires in a language very different from system architects' jargon. Capturing a consistent set of requirements from the customer and then massaging those requirements into a more formal specification is a structured way to manage the process of translating from the consumer's language to that of the designer's.

Requirements may be **functional** or **nonfunctional**. We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical nonfunctional requirements include:

- *Performance*: The speed of the system is often a major consideration both for the usability of the system and its ultimate cost. As we have noted, performance may

be a combination of soft performance metrics, such as approximate time to perform a user-level function, and hard deadlines by which a particular operation must be completed.

- *Cost*: The target cost or purchase price of a system is almost always a consideration. Cost typically has two major components: **manufacturing cost**, which includes the cost of components and assembly, and **nonrecurring engineering (NRE)** costs, which include the personnel, tooling, and other costs of designing the system.
- *Physical size and weight*: The physical aspects of the final system can vary greatly, depending upon the application. An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight. A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.
- *Power consumption*: Power, of course, is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life. However, the customer is unlikely to be able to describe the allowable wattage.

#### Validating requirements

Validating a set of requirements is ultimately a psychological task because it requires understanding both what people want and how they communicate those needs. One good way to refine at least the user interface portion of a system's requirements is to build a **mock-up**. The mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation. It should give the customer a good idea of how the system will be used and how the user can react to it. Physical, nonfunctional models of devices can also give customers a better idea of characteristics, such as size and weight.

#### Simple requirements form

Requirements analysis for big systems can be complex and time consuming. However, capturing a relatively small amount of information in a clear, simple format is a good start toward understanding system requirements. To introduce the discipline of requirements analysis as part of system design, we use a simple requirements methodology.

Fig. 1.2 shows a sample **requirements form** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of a system. Let's consider the entries in the form:

- *Name*: This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people, but can also crystallize the purpose of the machine.
- *Purpose*: This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.
- *Inputs and outputs*: These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:
  - *Types of data*: Analog electronic signals? Digital data? Mechanical inputs?
  - *Data characteristics*: Periodically arriving data, such as digital audio samples? Occasional user inputs? How many bits per data element?

Name	GPS moving map
Purpose	
Inputs	
Outputs	
Functions	
Performance	
Manufacturing cost	
Power	
Physical size and weight	

**FIGURE 1.2 Sample requirements form.**

- *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?
- *Functions:* This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?
- *Performance:* Many embedded computing systems spend at least some time controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early because they must be carefully measured during implementation to ensure that the system works properly.
- *Manufacturing cost:* This includes primarily the cost of the hardware components. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture: A machine that is meant to sell at \$25 most likely has a very different internal structure than a \$1000 system.
- *Power:* Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.
- *Physical size and weight:* You should give some indication of the physical size of the system to help guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel-mounted voice recorder.

A more thorough requirements analysis for a large system might use a form similar to Fig. 1.2 as a summary of the longer requirements document. After an introductory section containing this form, a longer requirements document could include details on

each of the items mentioned in the introduction. For example, each individual feature described in the introduction in a single sentence may be described in detail in a section of the specification.

#### Use cases

Another important means of understanding requirements is a set of **use cases**, which are descriptions of the system's use by actors. A use case describes how human users or other machines interact with the system. A basic use case diagram shows the set of possible actors—either people or other machines—and the operations they may perform. In some cases, cyber-physical systems may require a more thorough description of the use case, including a sequence diagram, which shows the series of operations that the user will perform. A set of use cases that describe typical usage scenarios often helps to clarify what the system needs to do. Analyzing the use cases for the system helps designers better understand its requirements.

#### Internal consistency of requirements

After writing the requirements, you should check them for internal consistency: Did you forget to assign a function to an input or output? Did you consider all the modes in which you want the system to operate? Did you place an unrealistic number of features into a battery-powered, low-cost machine?

To practice the capture of system requirements, Example 1.1 creates the requirements for a GPS moving map system.

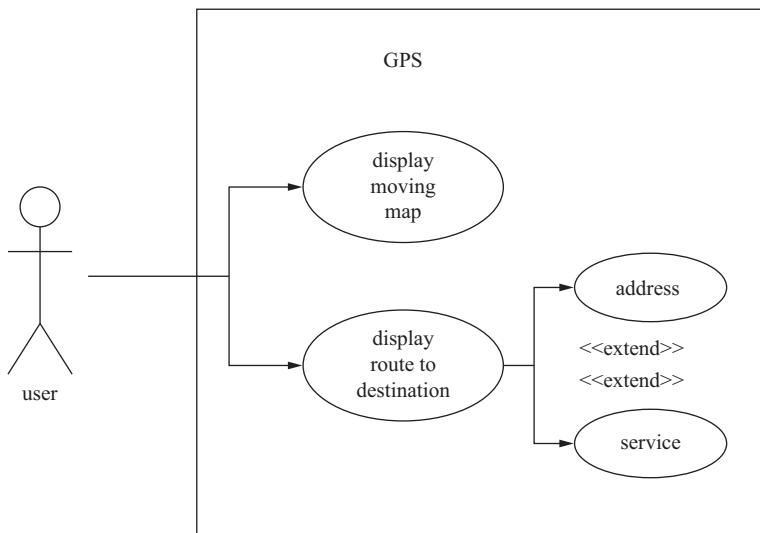
---

### Example 1.1: Requirements Analysis of a GPS Moving Map

The moving map is a small device, an alternative to a smartphone map, which displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change position. The device can also show directions to a destination on the map. The moving map obtains its position from the GPS, a satellite-based navigation system. GPS moving maps are often used for hiking or as accessories for cars.



A simple use case diagram shows that the user may interact with the driver.



The user has two main ways to use the GPS unit: simply display a map that moves with position, and asking the device to show a route to a desired destination. That basic functionality can be extended in two ways: giving an address for the destination or selecting from a list of possible services (gas, restaurant, and so on).

What requirements might we have for our GPS moving map? Here is an initial list:

- *Functionality*: This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.
- *User interface*: The screen should have at least  $400 \times 600$  pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed, to allow the user to make selections to control the system.
- *Performance*: The map should scroll smoothly with screen updates at least 10 frames per second. Upon power-up, a display should take no more than 1 s to appear, and the system should be able to verify its position and display the current map within 15 s.
- *Cost*: The selling cost (street price) of the unit should be no more than \$50. Selling price in turn helps to determine the allowable values for manufacturing cost. For example, if we use a rule-of-thumb in which the street cost is  $4\times$  that of the manufacturing cost, the cost to build this device should be no more than \$12.50. Selling price also influences the amount of money that should be spent on nonrecurring engineering costs. High NRE costs will not be recoverable if the profit on each device is low.
- *Physical size and weight*: The device should fit comfortably in the palm of the hand.
- *Power consumption*: The device should run for at least 8 h on four AA batteries with at least 30 min of those 8 h comprising operation with the screen on.

Notice that many of these requirements are not specified in engineering units. For example, physical size is measured relative to a hand, not in centimeters. Although these requirements must ultimately be translated into something that can be used by the designers, keeping a record of what the customer wants can help resolve questions about the specification that may crop up later during design.

Based on this discussion, let's write a requirements chart for our moving map system.

Name	GPS moving map
Purpose	Consumer-grade moving map for driving use
Inputs	Power button, two control buttons
Outputs	Back-lit LCD display 400×600
Functions	Uses a five-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude
Performance	Updates screen within 0.25 s upon movement
Manufacturing cost	\$12.50
Power	100 mW
Physical size and weight	No more than 2"×6", 12 oz

This chart adds some requirements in engineering terms that will be of use to the designers. For example, it provides actual dimensions of the device. The manufacturing cost was derived from the selling price by using a simple rule-of-thumb: The selling price is four to five times the **cost of goods sold** (the total of all the component costs).

### 1.3.2 Specification

The specification is more precise and complete; it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it fully reflects the system's requirements and does so in a way that can be clearly followed during design.

Specification is probably the least familiar phase of this methodology for neophyte designers, but it is essential to creating working systems with a minimum of designer effort. Designers who lack a clear idea of what they want to build when they begin typically make faulty assumptions early in the process that aren't obvious until they have a working system. At that point, the only solution is to take the machine apart, throw away some of it, and start again. Not only does this take a lot of extra time, but the result is also very likely to be inelegant, kludgy, and bug-ridden.

The specification should be understandable enough so that someone can verify that it meets the system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build. Designers can run into several types of problems caused by unclear specifications. If the behavior of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality. If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

A specification of the GPS system would include several components:

- data received from the GPS satellites;
- map data;
- user interface;
- operations that must be performed to satisfy customer requests;
- background actions required to keep the system running, such as operating the GPS receiver.

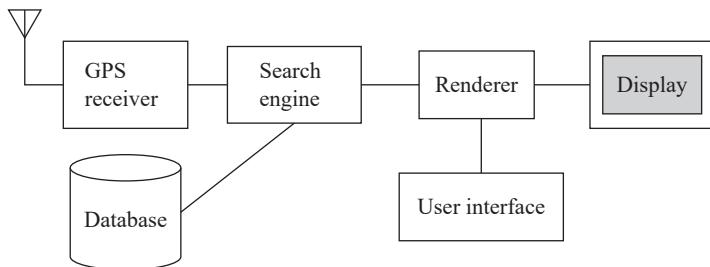
**UML**, a language for describing specifications, will be introduced in the next section. We will practice writing specifications in each chapter as we work through example system designs. We also study specification techniques in more detail in [Chapter 7](#).

### 1.3.3 Architecture design

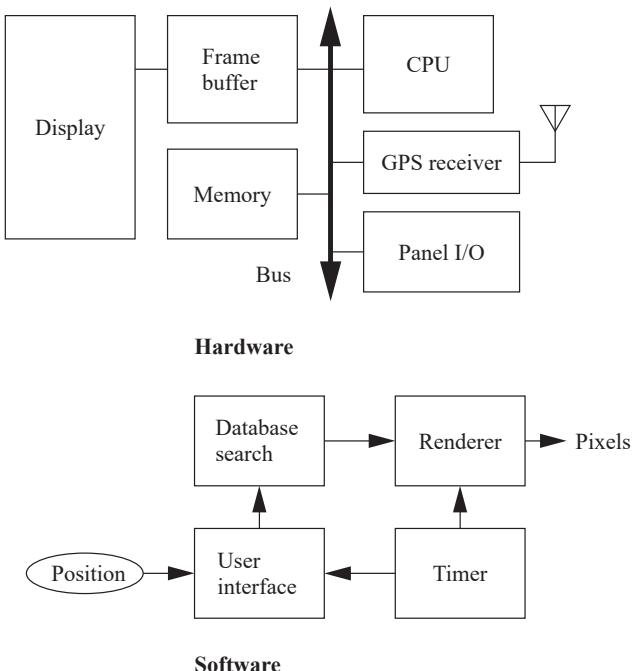
The specification does not say how the system does things, only what the system does. Describing how the system implements those functions is the purpose of the architecture. The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture. The creation of the architecture is the first phase of what many designers think of as design.

To understand what an architectural description is, let's look at a sample architecture for the moving map of Example 1.1. [Fig. 1.3](#) shows a sample system architecture in the form of a **block diagram** that shows major operations and data flows among them. This block diagram is still quite abstract; we haven't yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. We clearly see, for example, that we need to search the topographic database and to render (draw) the results for the display. We have chosen to separate those functions so that we can potentially do them in parallel, which includes performing rendering separately from searching the database to help us update the screen more fluidly.

Only after we have designed an initial architecture that is not biased toward too many implementation details should we refine the system block diagram into two



**FIGURE 1.3 Block diagram for the moving map.**



**FIGURE 1.4** Hardware and software architectures for the moving map.

block diagrams: one for hardware and another for software. These refined block diagrams are shown in Fig. 1.4. The hardware block diagram clearly shows that we have one CPU surrounded by memory and I/O devices. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU. The software block diagram follows the system block diagram, but we have added a timer to control when we read the buttons on the user interface and render data onto the screen. To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed.

Architectural descriptions must be designed to satisfy both functional and nonfunctional requirements. Not only must all the required functions be present, but we must meet cost, speed, power, and other nonfunctional constraints. Starting out with a system architecture and refining that to hardware and software architectures is one good way to ensure that we meet all specifications: We can concentrate on the functional elements in the system block diagram and then consider the nonfunctional constraints when creating the hardware and software architectures.

How do we know that our hardware and software architectures in fact meet constraints on speed, cost, and so on? We must somehow be able to estimate the properties of the components of the block diagrams, such as the search and rendering

functions in the moving map system. Accurate estimation derives in part from experience, both general design experience and particular experience with similar systems. However, we can sometimes create simplified models to help us make more accurate estimates. Sound estimates of all nonfunctional constraints during the architecture phase are crucial because decisions based on bad data will show up during the final phases of design, indicating that we did not, in fact, meet the specification.

### 1.3.4 Designing hardware and software components

The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification. The components will in general include both hardware—FPGAs, boards, and so on—and software modules.

Some of the components will be ready-made. The CPU, for example, will be a standard component in almost all cases, as will memory chips and many other components. In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component. We can also make use of standard software modules. One good example is the topographic database. Standard topographic databases exist, and you probably want to use standard routines to access the database. Not only are the data in a predefined format, but they are highly compressed to save storage. Using standard software for these access functions not only saves design time, but it may also give us a faster implementation for specialized functions, such as the data decompression phase.

You will have to design some components yourself. Even if you are using only standard integrated circuits, you may be required to design the printed circuit board that connects them. You will probably have to do a lot of custom programming as well. When creating these embedded software modules, you must of course make use of your expertise to ensure that the system runs properly in real time and that it doesn't take up more memory space than is allowed. The power consumption of the moving map software example is particularly important. You may need to be very careful about how you read and write memory to minimize power. For example, because memory accesses are a major source of power consumption, memory transactions must be carefully planned to avoid reading the same data several times.

### 1.3.5 System integration

Only after the components are built do we have the satisfaction of putting them together and seeing a working system. Of course, this phase usually consists of a lot more than just plugging everything together and standing back. Bugs are typically found during system integration, and good planning can help us find bugs quickly. By building up the system in phases and running properly chosen tests, we can often find bugs more easily. If we debug only a few modules at a time, we are more likely to uncover simple bugs and will be able to easily recognize them. Only by fixing the simple bugs early will we be able to uncover the more complex or obscure bugs that can

be identified only by giving the system a hard workout. We need to ensure during the architectural and component design phases that we make it as easy as possible to assemble the system in phases and test functions independently.

System integration is difficult because it usually uncovers problems. It is often hard to observe the system in sufficient detail to determine exactly what is wrong. The debugging facilities for embedded systems are usually much more limited than what you would find on desktop systems. As a result, determining why things don't work correctly and how they can be fixed is a challenge. Careful attention to inserting appropriate debugging facilities during design can help ease system integration problems, but the nature of embedded computing means that this phase will always be a challenge.

### 1.3.6 Formalisms for system design

As mentioned in the previous section, we perform a number of different design tasks at different levels of abstraction throughout this book: creating requirements and specifications, architecting the system, designing the code, and designing the tests. It is often helpful to conceptualize these tasks as diagrams. Luckily, there is a visual language that can be used to capture all these design tasks: the Unified Modeling Language [Boo99, Pil05]. UML was designed to be useful at many levels of abstraction in the design process. It is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction.

UML is an **object-oriented** modeling language. Object-oriented design emphasizes two concepts of importance:

- It encourages the design to be described as a number of interacting objects, rather than as a few large monolithic blocks of code.
- At least some of those objects will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines. It is sometimes important to implement something we think of at a high level as a single object using several distinct pieces of code or to otherwise break up the object correspondence in the implementation. However, thinking of the design in terms of actual objects helps us understand the natural structure of the system.

Object-oriented (often abbreviated as OO) specifications can be seen in two complementary ways:

- It allows a system to be described in a way that closely models real-world objects and their interactions.
- It provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems' components to real-world objects.

Both views are useful. At a minimum, object-oriented specification is a set of linguistic mechanisms. In many cases, it is useful to describe a system in terms of

real-world analogs. However, performance, cost, and so on may dictate that we change the specification to be different in some ways from the real-world elements we are trying to model and implement. In this case, the object-oriented specification mechanisms are still useful.

What is the relationship between an object-oriented specification and an object-oriented programming language, such as C++ [Str97]? A specification language may not be executable, but both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

UML is a large language, and covering all of it is beyond the scope of this book. In this section, we introduce only a few basic concepts. In later chapters, as we need a few more UML concepts, we introduce them to the basic modeling elements introduced here. Because UML is so rich, there are many graphical elements in a UML diagram. It is important to be careful to use the correct drawing to describe something. For instance, UML distinguishes between arrows with open and filled-in arrowheads, as well as solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you.

We also won't take a strict object-oriented approach. We may not always use objects for certain elements of a design; in some cases, such as when taking particular aspects of the implementation into account, it may make sense to use another design style. However, object-oriented design is widely applicable, and no designer can be considered design literate without understanding it.

### 1.3.7 Structural description

By **structural descriptions** of the basic components of the system, we will learn how to describe how these components act in the next section. The principal component of an object-oriented design is, naturally enough, the **object**. An object includes a set of **attributes** that define its internal state. When implemented in a programming language, these attributes usually become variables or constants held in a data structure. In some cases, we will add the type of the attribute after the attribute name for clarity, but we do not always have to specify a type for an attribute. An object describing a display, such as a CRT screen, is shown in the UML notation in Fig. 1.5. The text in the folded-cornerpage icon is a **note**; it does not correspond to an object in the system and only serves as a comment. The attribute is, in this case, an array of pixels that hold the contents of the display. The object is identified in two ways: It has a unique name, and it is a member of a **class**. The name is underlined to show that this is a description of an object and not of a class.

#### Classes as types

A class is a form of type definition. All objects derived from the same class have the same attributes, although their attributes may have different values. A class also defines the **operations** that determine how the object interacts with the rest of the world. In a programming language, the operations would become pieces of code used to manipulate the object. The UML description of the *Display* class is shown in Fig. 1.6. The class has the name that we saw used in the *d1* object because *d1* is an instance of class *Display*. The *Display* class defines the *pixels* attribute seen in

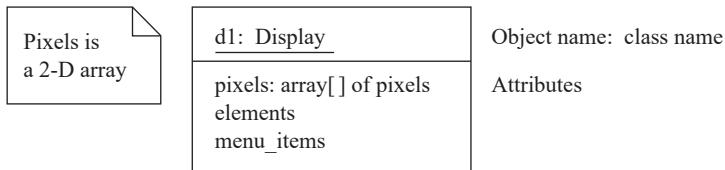


FIGURE 1.5 An object in UML notation.

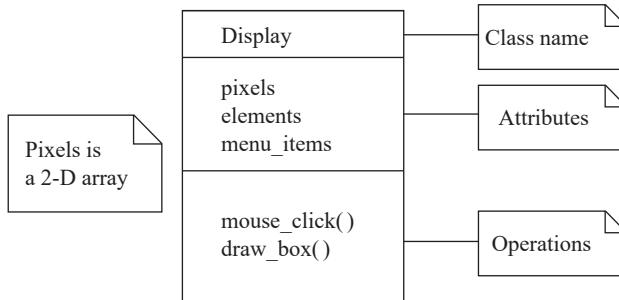


FIGURE 1.6 A class in UML notation.

the object. When we instantiate the class of an object, that object will have its own memory so that different objects of the same class have their own values for the attributes. Other classes can examine and modify class attributes; if we must do something more complex than use the attribute directly, we define a behavior to perform that function.

A class defines both the **interface** for a particular type of object and that object's **implementation**. When we use an object, we do not directly manipulate its attributes; we can only read or modify the object's state through the operations that define the interface to the object. The implementation includes both the attributes and whatever code is used to implement the operations. As long as we do not change the behavior of the object seen at the interface, we can change the implementation as much as we want. This lets us improve the system by, for example, speeding up an operation or reducing the amount of memory required without requiring changes to anything else that uses the object.

Clearly, the choice of an interface is a very important decision in object-oriented design. The proper interface must provide ways to access the object's state and update the state because we cannot directly see the attributes. We need to make the object's interface general enough so that we can make full use of its capabilities. However, excessive generality often makes the object large and slow. Big, complex interfaces also make the class definition difficult for designers to understand and use properly.

There are several types of **relationships** that can exist between objects and classes:

- **Association** occurs between objects that communicate with each other but have no ownership relationship between them.

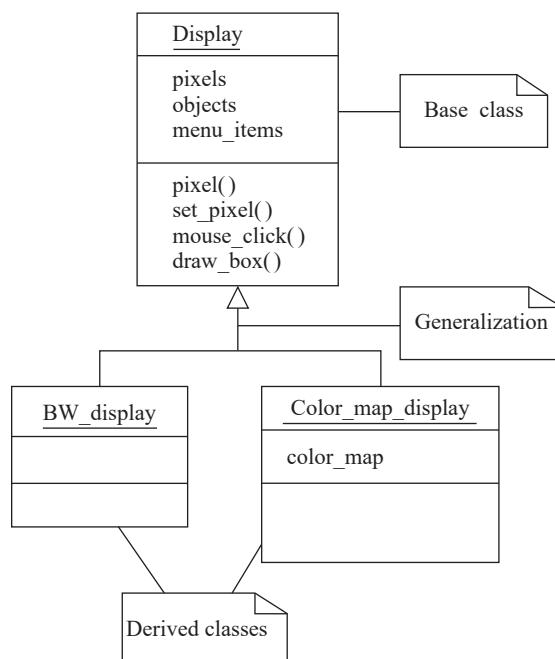
### Choose your interface properly

- **Aggregation** describes a complex object made of smaller objects.
- **Composition** is a type of aggregation in which the owner does not allow access to the component objects.
- **Generalization** allows us to define one class in terms of another.

The elements of a UML class or object do not necessarily directly correspond to statements in a programming language. If the UML is intended to describe something more abstract than a program, there may be a significant gap between the contents of the UML and a program implementing it. The attributes of an object do not necessarily reflect variables in the object. An attribute is some value that reflects the current state of the object. In the program implementation, that value could be computed from some other internal variables. The behaviors of the object would, at a higher-level specification, reflect the basic things that can be done with an object. Implementing all these features may require breaking up a behavior into several smaller behaviors. For example, you should initialize the object before you start to change its internal state.

#### Derived classes

UML, like most object-oriented languages, allows us to define one class in terms of another. An example is shown in Fig. 1.7, where we **derive** two types of displays. The first, *BW\_display*, describes a black-and-white display. This does not require us to add new attributes or operations, but we can specialize both to work on one-bit pixels. The second, *Color\_map\_display*, uses a graphic device known as a color map to allow



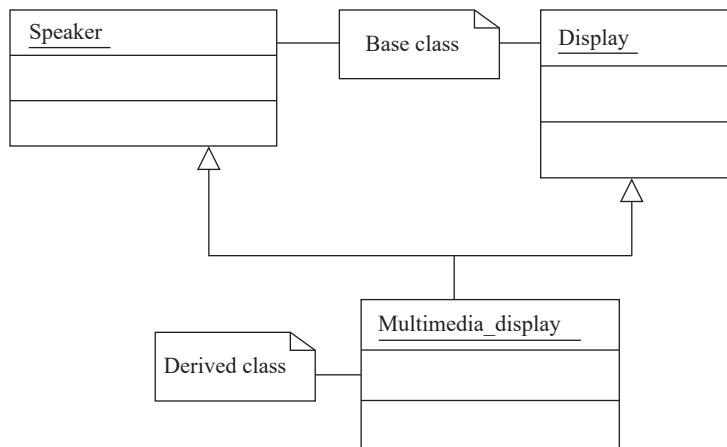
**FIGURE 1.7** Derived classes as a form of generalization in UML notation.

the user to select from many available colors, even with a few bits per pixel. This class defines a *color\_map* attribute that determines how pixel values are mapped onto display colors. A **derived class** inherits all the attributes and operations from its **base class**. In this class, *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class. This relation is transitive. If *Display* were derived from another class, both *BW\_display* and *Color\_map\_display* would inherit all the attributes and operations of *Display*'s base class as well. Inheritance has two purposes. It, of course, allows us to succinctly describe one class that shares some characteristics with another class. More importantly, it captures those relationships between classes and documents them. If we ever need to change any of the classes, knowledge of the class structure helps us determine the reach of changes. For example, should the change affect only *Color\_map\_display* objects, or should it change all *Display* objects?

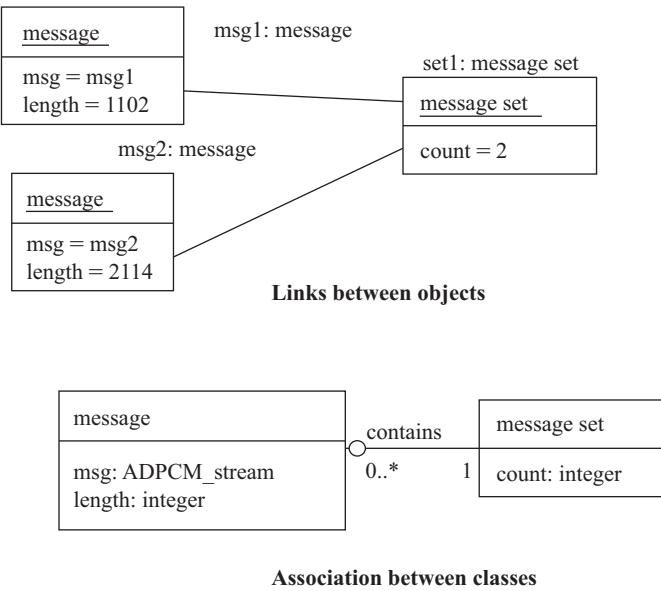
#### Generalization and inheritance

UML considers inheritance to be one form of generalization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrow-head. Both *BW\_display* and *Color\_map\_display* are specific versions of *Display*, so *Display* generalizes both. UML also allows us to define **multiple inheritance**, in which a class is derived from more than one base class. Most object-oriented programming languages support multiple inheritance as well. An example of multiple inheritance is shown in Fig. 1.8; we have omitted the details of the classes' attributes and operations for simplicity. In this case, we have created a *Multimedia\_display* class by combining the *Display* class with a *Speaker* class for sound. The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*. Because multiple inheritance causes the sizes of the attribute set and operations to expand so quickly, it should be used with care.

A **link** describes a relationship between objects; association is to link as class is to object. We need links because objects often do not stand-alone; associations let us



**FIGURE 1.8** Multiple inheritance in UML notation.

**FIGURE 1.9** Links and associations.

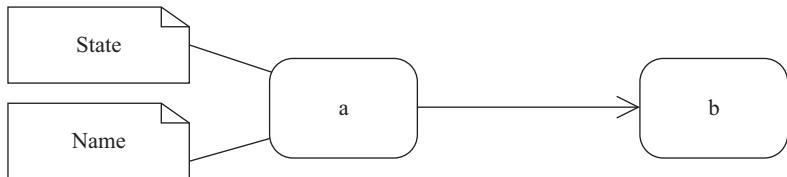
capture type information about these links. Fig. 1.9 shows examples of links and an association. When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in this example) and points to the active messages. In this case, the link defines the *contains* relation. When generalized into classes, we define an association between the message set class and the message class. The association is drawn as a line between the two labeled with the name of the association: *contains*. The ball and the number at the message class end indicate that the message set may include zero or more message objects. Sometimes we may want to attach data to the links themselves; we can specify this in the association by attaching a class-like box to the association's edge, which holds the association's data.

Typically, we find that we use a certain combination of elements in an object or class many times. We can give these patterns names, which are called **stereotypes** in UML. A stereotype name is written in the form `<<signal>>`.

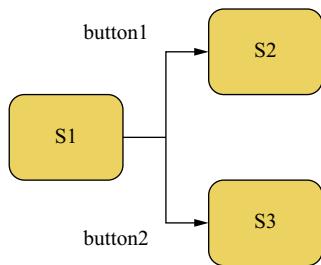
### 1.3.8 Behavioral description

We must specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a **state machine**. Fig. 1.10 shows the UML states; the transition between two states is shown by a skeleton arrow.

These state machines do not rely on the operation of a clock as in hardware; rather, changes from one state to another are triggered by the occurrence of **events**. An event

**FIGURE 1.10**

A state and transition in UML notation.

**FIGURE 1.11 Events in UML state machines.**

is some type of action. Fig. 1.11 illustrates this aspect of UML state machines: The machine transitions from S1 to S2 or S3 only when button1 or button2 is pressed. The event may originate outside the system, such as the button press. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine.

UML defines several special types of events, as illustrated in Fig. 1.12:

- A **signal** is an asynchronous occurrence. It is defined in UML by an object that is labeled as a «signal». The object in the diagram serves as a declaration of the event's existence. Because it is an object, a signal may have parameters that are passed to the signal's receiver.
- A **call event** follows the model of a procedure call in a programming language.
- A **time-out event** causes the machine to leave a state after a certain amount of time. The label tm(time-value) on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism.

We show the occurrence of all types of signals in a UML diagram in the same way as a label on a transition.

Let's consider a simple state machine specification to understand the semantics of UML state machines. A state machine for the operation of the display is shown in Fig. 1.13. The start and stop states are special states that help us organize the flow of the state machine. The states in the state machine represent different conceptual

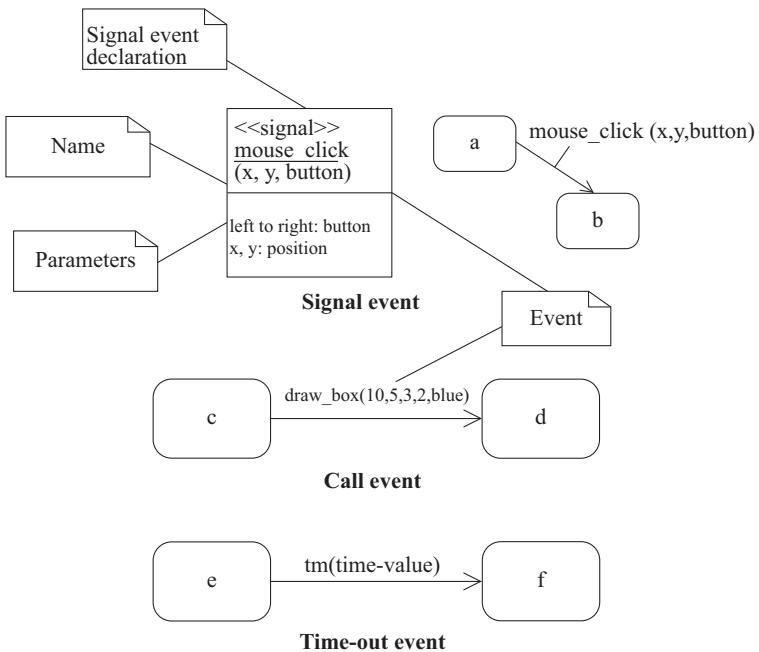


FIGURE 1.12 Signal, call, and time-out events in UML notation.

operations. In some cases, we take conditional transitions out of states based on inputs or the results of some computation done in the state. In other cases, we make an unconditional transition to the next state. Both the unconditional and conditional transitions make use of the call event. Splitting a complex operation into several

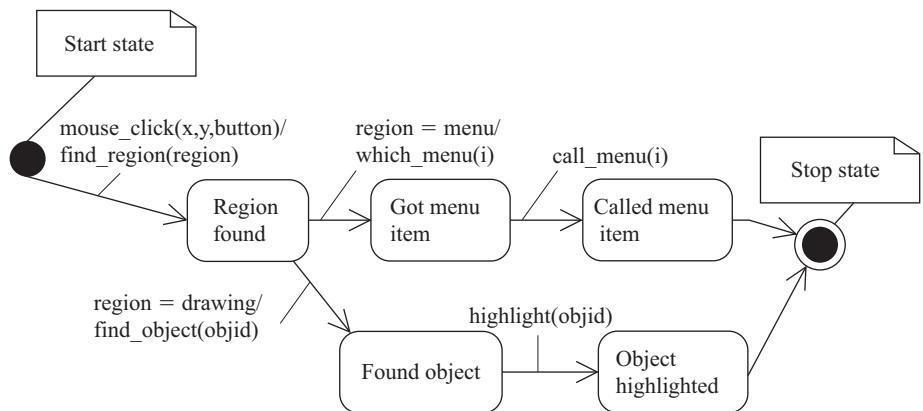


FIGURE 1.13 A state machine specification in UML notation.

states helps document the required steps, much as subroutines can be used to structure code.

It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved. In this case, we can create a **sequence diagram**, which is often used to describe use cases. A sequence diagram is like a hardware timing diagram, although the time flows vertically in a sequence diagram, whereas time typically flows horizontally in a timing diagram. The sequence diagram is designed to show a particular scenario or choice of events; it is not convenient for showing several mutually exclusive possibilities.

An example of a mouse click and its associated actions is shown in Fig. 1.14. The mouse click occurs on the menu region. Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all objects remain alive for the entire sequence, but in other cases, objects may be created or destroyed during processing. The boxes along the lifelines show the *focus of control* in the sequence when the object is actively processing. In this case, the mouse object is active only long enough to create the *mouse\_click* event. The display object remains in play longer; it then uses call events to invoke the menu object twice: once to determine which menu item was selected and again to execute the menu call. The *find\_region()* call is internal to the display object, so it does not appear as an event in the diagram.

## 1.4 Design example: Model train controller

In order to learn how to use UML to model systems, we specify a simple system, a **model train controller**. Model trains run on tracks and often represent scale models of full-sized trains. Simple trains either run at a constant speed or use a simple speed control. Modern model trains can make use of digital train controllers that send

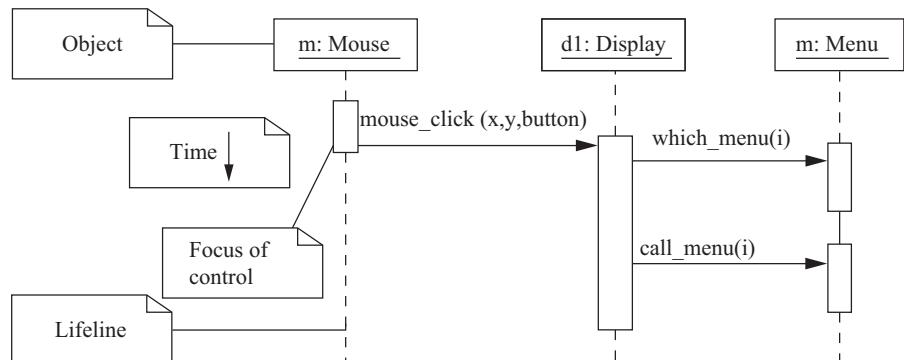
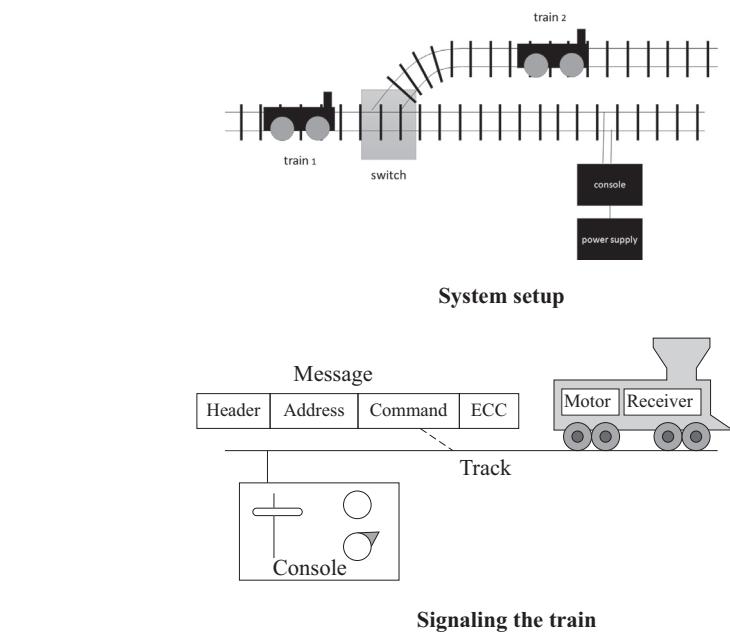


FIGURE 1.14 A sequence diagram in UML notation.

messages along the tracks to the train engine. Train controllers not only allow sophisticated control of the train—various speeds and reverse—but they also allow several trains to run on the track at different speeds. The train controller allows the hobbyist to recreate much more sophisticated train environments.

**Fig. 1.15** shows a train controller in the context of a model train layout. Several trains may be on the track at the same time, each with its own address. The controller may also control other parts of the layout, such as switches that select which track a train will follow at the switch point. The user sends messages to a train through a control box attached to the tracks. The control box may have familiar controls, such as a throttle, an emergency stop button, and so on. Because the train receives its electrical power from the two rails of the track, the control box can send signals to the train over the tracks by modulating the power-supply voltage. As shown in the figure, the control panel sends packets over the tracks to the receiver on the train. The train includes analog electronics to sense the bits being transmitted and a control system to set the train motor's speed and direction based on those commands. Each packet includes an address so that the console can control several trains on the same track. The packet also includes an error correction code (ECC) to guard against transmission errors. This is a one-way communication system; the model train cannot send status or acknowledgments back to the user.



**FIGURE 1.15** A model train control system.

We start by analyzing the requirements for the train control system. We base our system on a real standard developed for model trains. We then develop two specifications: a simple, high-level specification and a more detailed one.

### 1.4.1 Requirements

Before we can create a system specification, we should understand the requirements. Here is a basic set of requirements for the system:

- The console shall be able to control up to eight trains on a single track.
- The speed of each train shall be controllable by a throttle to at least 63 different levels in each direction (forward and reverse).
- There shall be an inertia control that shall allow the user to adjust the responsiveness of the train to commanded changes in speed. Higher inertia means that the train responds more slowly to a change in the throttle, simulating the inertia of a large train. The inertia control will provide at least eight different levels.
- There shall be an emergency stop button.
- An error detection scheme will be used to transmit messages.

We can put the requirements into our chart format:

Name	Model train controller
Purpose	Control speed of up to eight model trains
Inputs	Throttle, inertia setting, emergency stop, train number
Outputs	Train control signals
Functions	Set engine speed based upon inertia settings; respond to emergency stop
Performance	Can update train speed at least 10 times per second
Manufacturing cost	\$50
Power	10 W (plugs into wall)
Physical size and weight	Console should be comfortable for two hands, approximate size of standard keyboard; weight less than 2 lb.

We will develop our system using a widely used standard for model train control. We could develop our own train control system from scratch, but basing our system upon a standard has several advantages in this case. It reduces the amount of work we must do, and it allows us to use a wide variety of existing trains and other pieces of equipment.

### 1.4.2 Digital Command Control (DCC)

The DCC standard ([http://www.nmra.org/standards/DCC/standards\\_rps/DCCStds.html](http://www.nmra.org/standards/DCC/standards_rps/DCCStds.html)) was created by the National Model Railroad Association to support interoperable digitally controlled model trains. Hobbyists started building homebrew digital control systems in the 1970s, and Marklin developed its own digital control system in the 1980s. DCC was created to provide a standard that could be built by any manufacturer so that hobbyists could mix and match components from multiple vendors.

#### DCC documents

The DCC standard is given in two documents:

- Standard S-9.1, the DCC Electrical Standard, defines how bits are encoded on the rails for transmission.
- Standard S-9.2, the DCC Communication Standard, defines the packets that carry information.

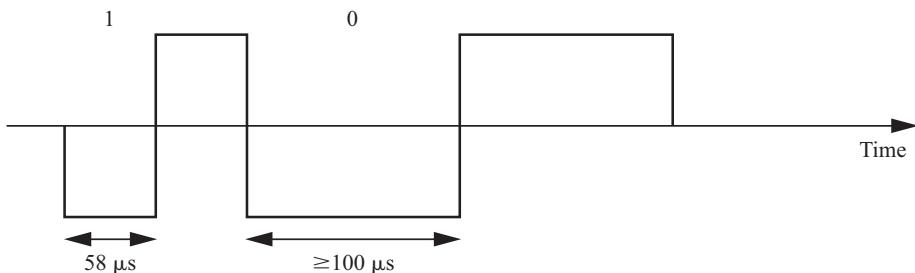
Any DCC-conforming device must meet these specifications. DCC also provides several recommended practices. These are not strictly required, but they provide some hints to manufacturers and users as to how to best use DCC.

The DCC standard does not specify many aspects of a DCC train system. It doesn't define the control panel, the type of microprocessor used, the programming language to be used, or many other aspects of a real model train system. The standard concentrates on those aspects of system design that are necessary for interoperability. Over-standardization, or specifying elements that don't really need to be standardized, only makes the standard less attractive and harder to implement.

#### DCC Electrical Standard

The Electrical Standard deals with voltages and currents on the track. Although the electrical engineering aspects of this part of the specification are beyond the scope of the book, we briefly discuss the data encoding here. The standard must be carefully designed because the main function of the track is to carry power to the locomotives. The signal encoding system should not interfere with power transmission either to DCC or non-DCC locomotives. A key requirement is that the data signal should not change the average power supply voltage on the train's rails.

The data signal swings between two voltages around the power-supply voltage. As shown in Fig. 1.16, bits are encoded in the time between transitions, not by voltage levels. A zero is at least 100 s, whereas a one is nominally 58 s. The specification



**FIGURE 1.16** Bit encoding in digital command and control.

**DCC Communication Standard**

also gives the allowable variations in bit times that a conforming DCC receiver must be able to tolerate.

The standard also describes other electrical properties of the system, such as allowable transition times for signals.

The DCC Communication Standard describes how bits are combined into packets and the meaning of some important packets. Some packet types are left undefined in the standard, but typical uses are given in the recommended practices documents.

We can write the basic packet format as a regular expression:

$$PSA(sD) + E \quad (\text{Eq. 1.1})$$

In this regular expression,

- P is the preamble, which is a sequence of at least ten 1 bits. The command station should send at least 14 of these 1 bits, some of which may be corrupted during transmission.
- S is the packet start bit. It is a 0 bit.
- A is an address data byte that gives the address of the unit, with the most significant bit of the address transmitted first. An address is 8 bits long. The addresses, 00000000, 11111110, and 11111111, are reserved.
- s is the data byte start bit, which, like the packet start bit, is 0.
- D is the data byte, which includes 8 bits. A data byte may contain an address, instruction data, or error correction information.
- E is a packet end bit, which is a 1 bit.

A packet includes one or more data byte start bit/data byte combinations. Note that the address data byte is a specific type of data byte.

A **baseline packet** is the minimum packet that must be accepted by all DCC implementations. More complex packets are given in a recommended practice document. A baseline packet has three data bytes: An address data byte gives the intended receiver of the packet; the instruction data byte provides a basic instruction; and an error correction data byte is used to detect and correct transmission errors.

The instruction data byte carries several pieces of information. Bits 0–3 provide a 4-bit speed value. Bit 4 has an additional speed bit, which is interpreted as the least significant speed bit. Bit 5 gives direction, with one for forward and zero for reverse. Bits 7–8 are set as 01 to indicate that this instruction provides speed and direction.

The error correction data byte is the bitwise exclusive OR of the address and instruction data bytes.

The standard says that the command unit should send packets frequently because a packet may be corrupted. Packets should be separated by at least 5 ms.

**Baseline packet****Packet transmission rates**

### 1.4.3 Conceptual specification

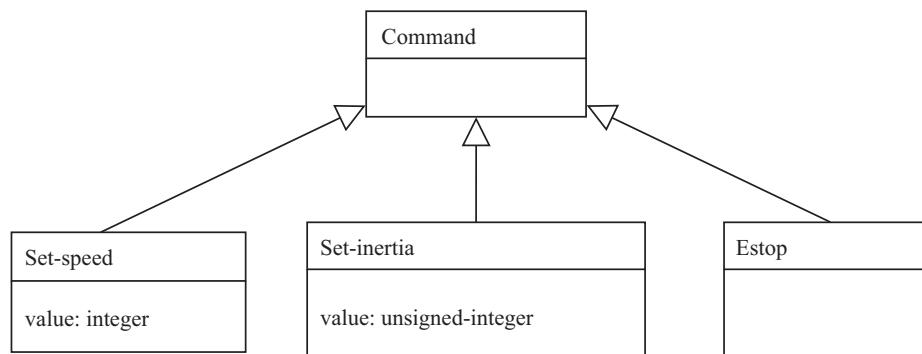
DCC specifies some important aspects of the system, particularly those that allow equipment to interoperate. However, DCC deliberately does not specify everything about a model train control system. We must round out our specification with details

**Commands**

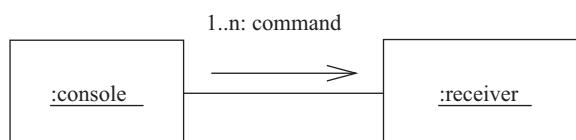
that complement the DCC spec. A **conceptual specification** allows us to understand the system a little better. We use the experience gained by writing the conceptual specification to help us write a detailed specification to be given to a system architect. This specification doesn't correspond to what any commercial DCC controllers do, but it is simple enough to allow us to cover some basic concepts in system design.

A train control system turns **commands** into **packets**. A command comes from the command unit while a packet is transmitted over the rails. Commands and packets may not be generated in a 1:1 ratio. In fact, the DCC standard says that command units should resend packets in case a packet is dropped during transmission. Fig. 1.17 shows a generic command class and several specific commands derived from that base class. *Estop* (emergency stop) requires no parameters, whereas *Set-speed* and *Set-inertia* do.

We now need to model the train control system itself. There are clearly two major subsystems: the command unit and the train-board component. Each of these subsystems has its own internal structure. The basic relationship between them is illustrated in Fig. 1.18. This figure shows a UML **collaboration diagram**. We could have used another type of figure, such as a class or object diagram, but we want to emphasize the transmit/receive relationship between these major subsystems. The command unit and receiver are each represented by objects; the command unit sends a sequence of packets to the train's receiver, as illustrated by the arrow. The notation on the arrow provides both the type of message sent and its sequence in a flow of messages;



**FIGURE 1.17 Class diagram for the train controller commands.**

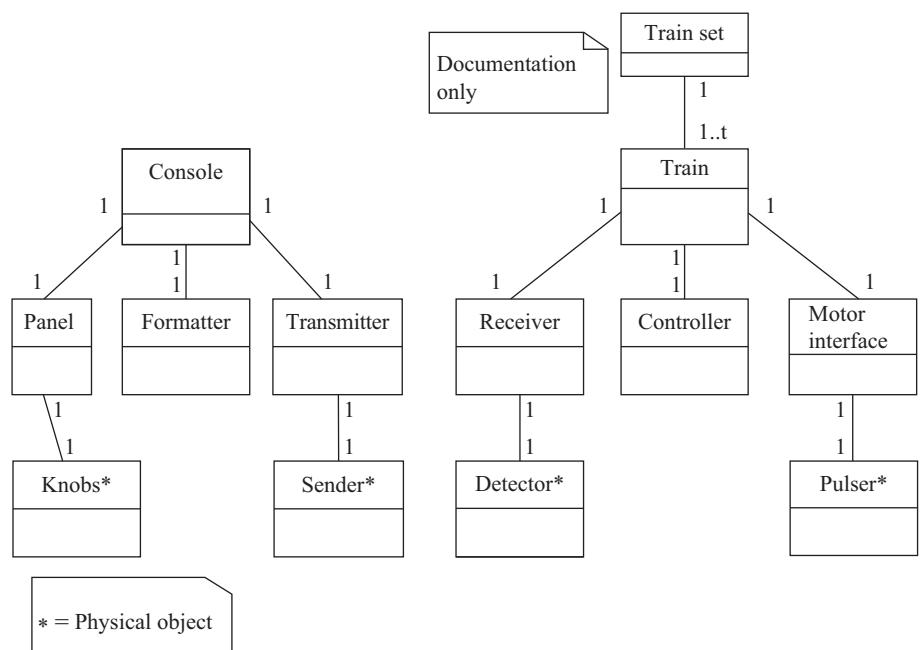


**FIGURE 1.18 UML collaboration diagram for major subsystems of the train controller system.**

because the console sends all the messages, we have numbered the arrow's messages as 1...n. Those messages are, of course, carried over the track. Because the track isn't a computer component and is purely passive, it does not appear in the diagram. However, it would be perfectly legitimate to model the track in the collaboration diagram, and in some situations, it may be wise to model such nontraditional components. For example, if we are worried about what happens when the track breaks, modeling the tracks would help us identify failure modes and recovery mechanisms.

Let's break down the command unit and receiver into their major components. The console needs to perform three functions: read the state of the front panel on the command unit, format messages, and transmit messages. The train receiver must also perform three major functions: receive the message, interpret the message (taking into account the current speed, inertia setting, and so on), as well as actually controlling the motor. In this case, let's use a class diagram to represent the design; we could also use an object diagram if we wished. The UML class diagram is shown in Fig. 1.19. It shows the console class using three classes, one for each of its major components. These classes must define some behaviors, but for the moment, we concentrate on the basic characteristics of these classes:

- The *Console* class describes the command unit's front panel, which contains the analog knobs and hardware to interface to the digital parts of the system.



**FIGURE 1.19** A UML class diagram for the train controller showing the composition of the subsystems.

- The *Formatter* class includes behaviors that know how to read the panel knobs and create a bit stream for the required message.
- The *Transmitter* class interfaces analog electronics to send the message along the track.

There will be one instance of the *Console* class and one instance of each of the component classes, as shown by the numeric values at each end of the relationship links. We also show some special classes that represent analog components, ending the name of each with an asterisk:

- *Knobs\** describes the actual analog knobs, buttons, and levers on the control panel.
- *Sender\** describes the analog electronics that send bits along the track.

Likewise, the *Train* makes use of three other classes that define its components:

- The *Receiver* class knows how to turn the analog signals on the track into digital form.
- The *Controller* class includes behaviors that interpret the commands and figure out how to control the motor.
- The *Motor interface* class defines how to generate the analog signals required to control the motor.

We define two classes to represent analog components:

- *Detector\** detects analog signals on the track and converts them into digital form.
- *Pulser\** turns digital commands into the analog signals required to control the motor speed.

We also define a special class, *Train set*, to help us remember that the system can handle multiple trains. The values on the relationship edge show that one train set can have  $t$  trains. We would not actually implement the train set class, but it does serve as useful documentation of the existence of multiple receivers.

[Fig. 1.20](#) shows a sequence diagram for the simple use of DCC with two trains. The console is first set to select train 1 and then to set the speed. The console is then set to select train 2 and set its speed.

#### 1.4.4 Detailed specification

Now that we have a conceptual specification that defines the basic classes, let's refine it to create a more detailed specification. We won't create a complete specification, but we will add detail to the classes and look at some of the major decisions in the specification process to get a better handle on how to write good specifications.

At this point, we need to define the analog components in a little more detail because their characteristics will strongly influence the *Formatter* and *Controller*.

[Fig. 1.21](#) shows a class diagram for these classes; this diagram shows a little more detail than [Fig. 1.19](#) because it includes attributes and behaviors of these classes. The *Panel* has three knobs: *train number* (which train is currently being controlled), *speed* (which can be positive or negative), and *inertia*. It also has one button for

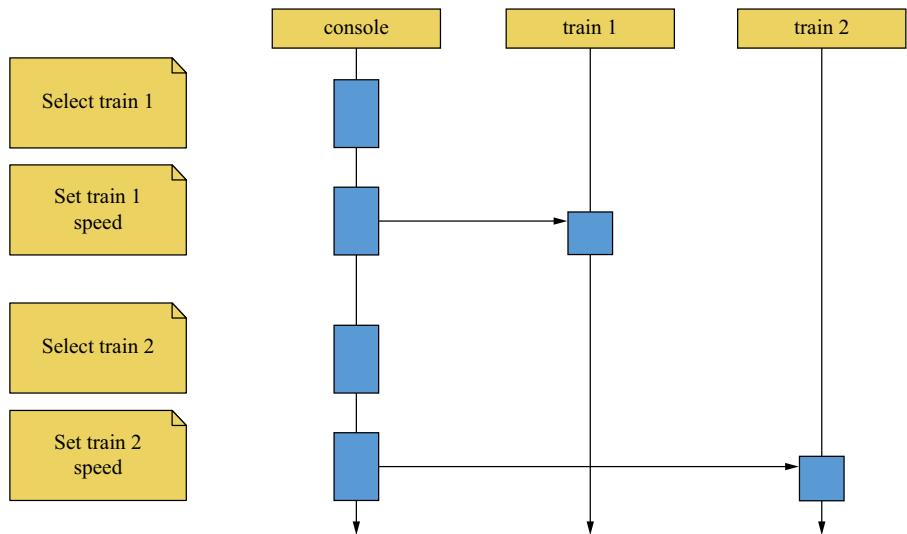


FIGURE 1.20 Use case for setting speed of two trains.

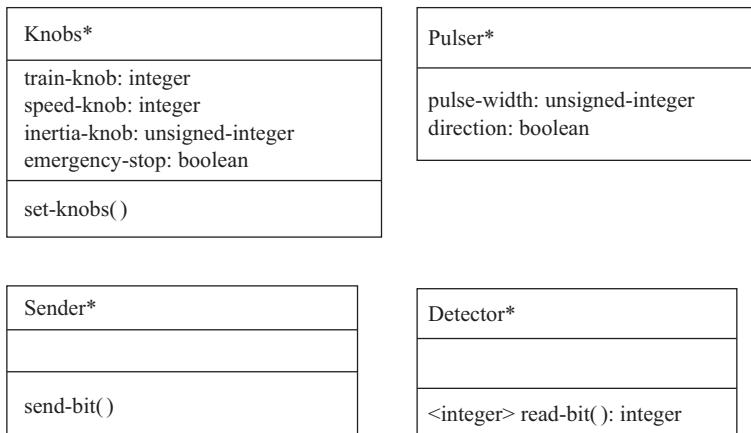


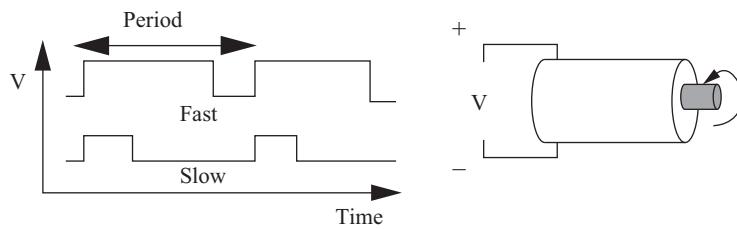
FIGURE 1.21 Classes describing analog physical objects in the train control system.

*emergency-stop*. When we change the train number setting, we also want to reset the other controls to the proper values for that train so that the previous train's control settings are not used to change the current train's settings. To do this, *Knobs* must provide a *set-knobs* behavior that allows the rest of the system to modify the knob settings. If we wanted or needed to model the user, we would expand on this class definition to provide methods that a user object would call to specify these parameters. The motor

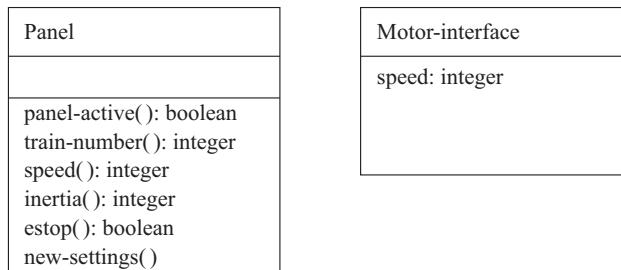
system takes its motor commands in two parts. The *Sender* and *Detector* classes are relatively simple: They simply put out and pick up a bit, respectively.

To understand the *Pulser* class, let's consider how we control the train motor's speed. As shown in Fig. 1.22, the speed of electric motors is commonly controlled using pulse-width modulation: Power is applied in a pulse for a fraction of some fixed interval, with the fraction of the time that power is applied determining the speed. The digital interface to the motor system specifies that pulse width is an integer, with the maximum value being maximum engine speed. A separate binary value controls direction. Note that the motor control takes an unsigned speed with a separate direction, while the panel specifies speed as a signed integer, with negative speeds corresponding to reverse.

Fig. 1.23 shows the classes for the panel and motor interfaces. These classes form the software interfaces to their respective physical devices. The *Panel* class defines a behavior for each of the controls on the panel. We have chosen not to define an internal variable for each control because their values can be read directly from the physical device, but a given implementation may choose to use internal variables. The *new-settings* behavior uses the *set-knobs* behavior of the *Knobs\** class to change the knobs settings whenever the train number setting is changed. *Motor-interface* defines an attribute for speed that can be set by other classes. As we will see in a moment, the controller's job is to incrementally adjust the motor's speed to provide smooth acceleration and deceleration.

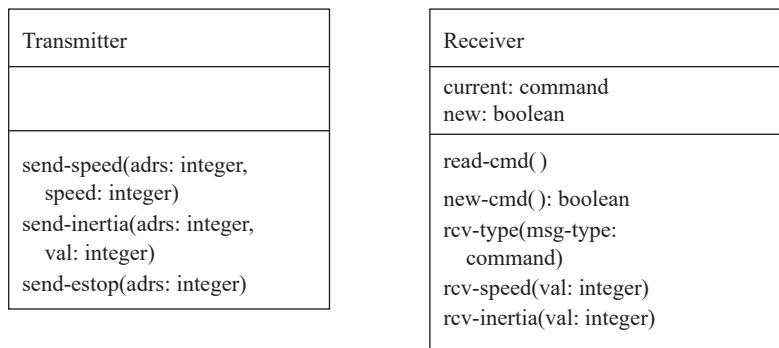


**FIGURE 1.22** Controlling motor speed by pulse-width modulation.



**FIGURE 1.23**

Class diagram for the panel and motor interface.



**FIGURE 1.24 Class diagram for the *Transmitter* and *Receiver*.**

---

The *Transmitter* and *Receiver* classes are shown in Fig. 1.24. They provide the software interface to the physical devices that send and receive bits along the track. The *Transmitter* provides a distinct behavior for each type of message that can be sent. It internally takes care of formatting the message. The *Receiver* class provides a *read-cmd* behavior to read a message off the tracks. We can assume for now that the receiver object allows this behavior to run continuously to monitor the tracks and intercept the next command. (We consider how to model such continuously running behavior as processes in Chapter 6.) We use an internal variable, *current*, to hold the current command. Another variable, *new*, holds a flag showing when the command has been processed. Separate behaviors let us read out the parameters for each type of command; these messages also reset the new flag to show that the command has been processed. We don't need a separate behavior for an *Estop* message because it has no parameters; knowing the type of message is sufficient.

Now that we have specified the subsystems around the formatter and controller, it is easier to see what sorts of interfaces these two subsystems may need.

The *Formatter* class is shown in Fig. 1.25. The formatter holds the current control settings for all the trains. The *send-command* method is a utility function that serves as the interface to the transmitter. The *operate* function performs the basic actions for the object. At this point, we only need a simple specification, which states that the formatter repeatedly reads the panel, determines whether any settings have changed, and sends out the appropriate messages. The *panel-active* behavior returns true whenever the panel's values do not correspond to the current values.

The role of the formatter during the panel's operation is illustrated by the sequence diagram of Fig. 1.26. The figure shows two changes to the knob settings: first to the throttle, inertia, or emergency stop and then to the train number. The panel is called periodically by the formatter to determine if any control settings have changed. If a setting has changed for the current train, the formatter decides to send a command, issuing a *send-command* behavior to cause the transmitter to send the bits. Because transmission is serial, it takes a noticeable amount of time for the transmitter to finish a command; in the meantime, the formatter continues to check the panel's control

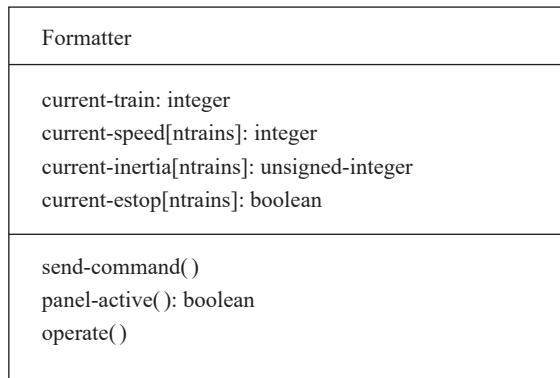
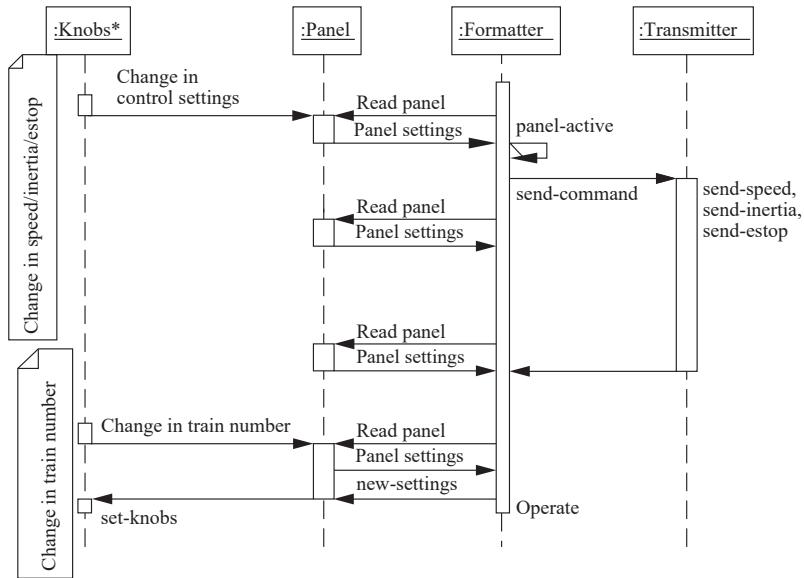
FIGURE 1.25 Class diagram for the *Formatter* class.

FIGURE 1.26 Sequence diagram for transmitting a control input.

settings. If the train number has changed, the formatter must cause the knob settings to be reset to the proper values for the new train.

We have not yet specified the operation of any of the behaviors. We define what a behavior does by writing a state diagram. The state diagram for a very simple version of the `operate` behavior of the `Formatter` class is shown in Fig. 1.27. This behavior watches the panel for activity: If the train number changes, it updates the panel display; otherwise, it causes the required message to be sent. Fig. 1.28 shows a state diagram for the `panel-active` behavior.

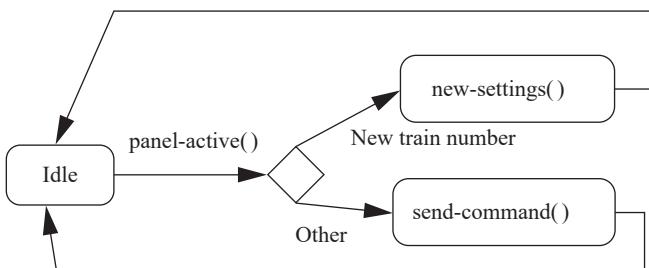
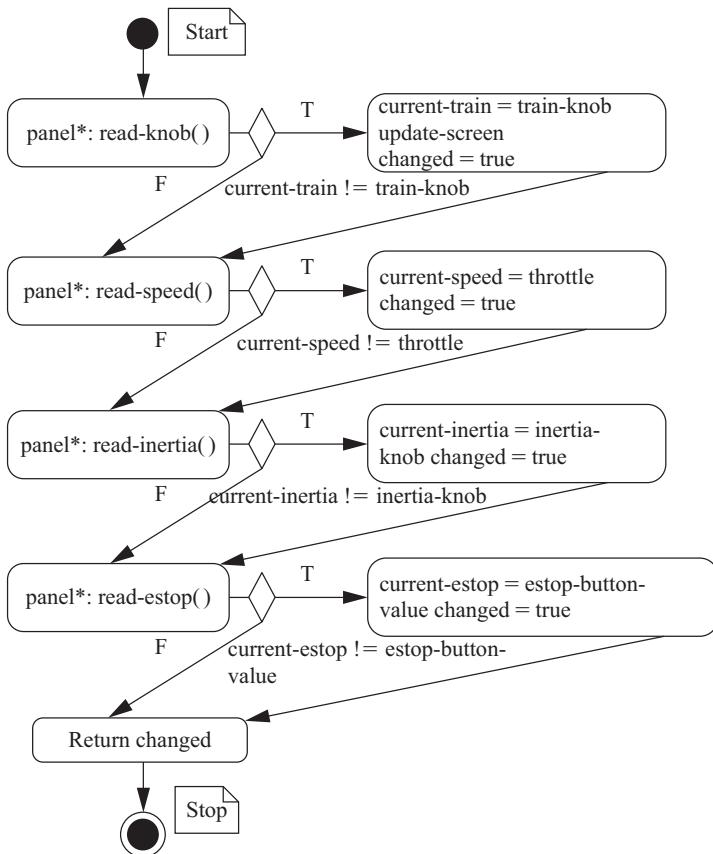
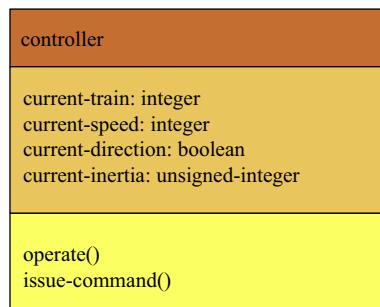
FIGURE 1.27 State diagram for the formatter *operate* behavior.

FIGURE 1.28 State diagram for the panel-activate behavior.

The definition of the train's *Controller* class is shown in Fig. 1.29. The *operate* behavior is called by the receiver when it gets a new command; *operate* looks at the contents of the message and uses the *issue-command* behavior to change the speed,



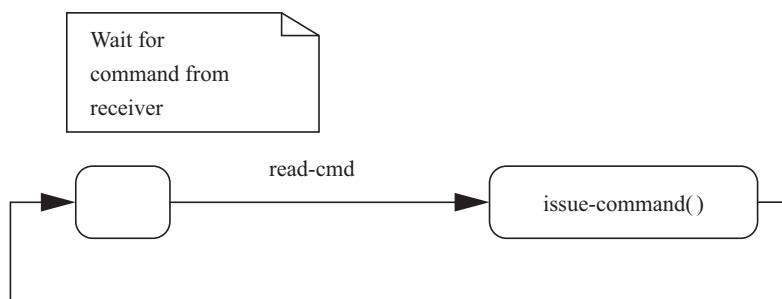
**FIGURE 1.29** Class diagram for the *Controller* class.

direction, and inertia settings, as necessary. A specification for *operate* is shown in Fig. 1.30.

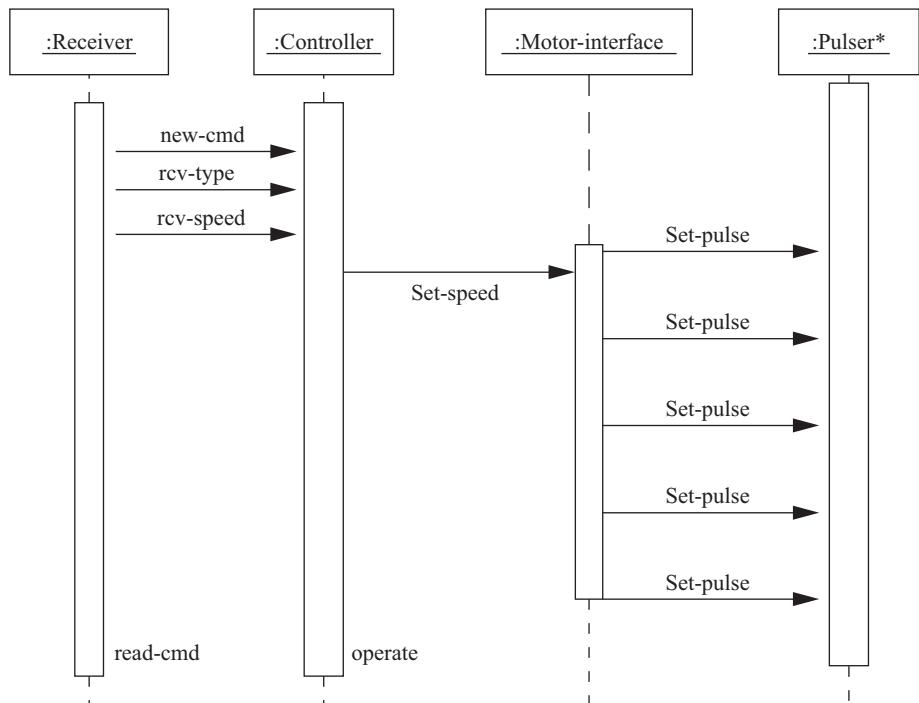
The operation of the *Controller* class during the reception of a *set-speed* command is illustrated in Fig. 1.31. The *Controller*'s *operate* behavior must execute several behaviors to determine the nature of the message. Once the speed command has been parsed, it must send a sequence of commands to the motor to smoothly change the train's speed.

#### Refining command classes

It is also a good idea to refine our notion of a command. These changes result from the need to build a potentially upward-compatible system. If the messages were entirely internal, we would have more freedom in specifying messages that we could use during architectural design. However, because these messages must work with a variety of trains and we may want to add more commands in a later version of the system, we need to specify the basic features of messages for compatibility. There are three important issues. First, we need to specify the number of bits used to determine the message type. We chose three bits because that gives us five unused message codes. Second, we need to include information about the length of the data fields, which is determined by the resolution for speeds and inertia set by the requirements. Third,



**FIGURE 1.30** State diagram for the *Controller*'s *operate* behavior.



**FIGURE 1.31 Sequence diagram for a *set-speed* command received by the train.**

we need to specify the error detection mechanism; we choose to use a single-parity bit. We can update the classes to provide this extra information, as shown in Fig. 1.32.

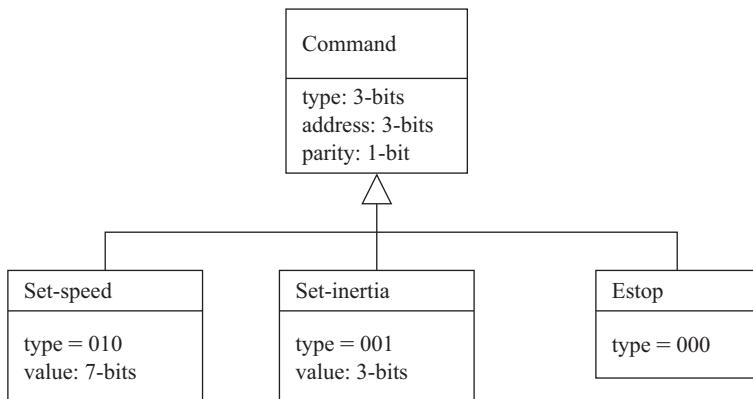
#### 1.4.5 Lessons learned

This example illustrates some general concepts. First, standards are important. We often cannot avoid working with standards, but standards often save us work and allow us to make use of components designed by others. Second, specifying a system isn't easy. You often learn a lot about the system you are trying to build by writing a specification. Third, specification invariably requires making some choices that may influence the implementation. Good system designers use their experience and intuition to guide them when these kinds of choices must be made.

---

## 1.5 A guided tour of this book

The most efficient way to learn all the necessary concepts is to move from the bottom up. This book is arranged so that you learn about the properties of components and build toward more complex systems and a more complete view of the system design

**FIGURE 1.32**

Refined class diagram for the train controller commands.

process. Veteran designers have learned enough bottom-up knowledge from experience to know how to use a top-down approach to designing a system, but when learning things for the first time, the bottom-up approach allows you to build more sophisticated concepts based on lower-level ideas.

We use several organizational devices throughout the book to help you. Application Examples focus on a particular end-use application and how it relates to embedded system design. We also make use of Programming Examples to describe software designs. In addition to these, most chapters use significant system design examples to demonstrate the major concepts of the chapter.

Each chapter includes questions that are intended to be answered on paper as homework assignments. The chapters also include lab exercises. These are more open ended and are intended to suggest activities that can be performed in the lab to help illuminate various concepts in the chapter.

Throughout the book, we use several CPUs as examples: the advanced RISC machine (ARM) processor, the Texas Instruments C55x digital signal processor (DSP), the PIC16F, and the Texas Instruments C64x. All are well-known microprocessors used in many embedded applications. Using real microprocessors helps make concepts more concrete. However, our aim is to learn concepts that can be applied to all sorts of microprocessors. Although microprocessors will evolve over time (Warhol's Law of Computer Architecture [Wol92] states that every microprocessor architecture will be the price/performance leader for 15 min), the concepts of embedded system design are fundamental and long term.

### 1.5.1 Chapter 2: Instruction sets

In [Chapter 2](#), we begin our study of microprocessors by concentrating on **instruction sets**. The chapter covers the instruction sets of the ARM, Microchip PIC16F, TI C55x,

and TI C64x microprocessors in separate sections. All these microprocessors are very different. Understanding all details of both is not strictly necessary to the design of embedded systems. However, comparing them does provide some interesting lessons in instruction set architectures.

Understanding some aspects of the instruction set is important both for concreteness and for seeing how architectural features can affect performance and other system attributes. However, many mechanisms, such as caches and memory management, can be understood in general before we go on to detail how they are implemented in these three microprocessors.

We do not introduce a design example in this chapter; it is difficult to build even a simple working system without understanding other aspects of the CPU that will be introduced in [Chapter 3](#). However, understanding instruction sets helps us to understand problems, such as execution speed and code size, that we study throughout the book.

### 1.5.2 Chapter 3: CPUs

[Chapter 3](#) rounds out our discussion of microprocessors by focusing on the following important mechanisms that are not part of the instruction set itself:

- We introduce the fundamental mechanisms of **input** and **output**, including interrupts.
- We also study the **cache** and **memory management unit**.

We also begin to consider how CPU hardware affects important characteristics of program execution. Program performance and power consumption are very important parameters in embedded system design. An understanding of how architectural aspects, such as pipelining and caching affect these system characteristics is a foundation for analyzing and optimizing programs in later chapters.

Our study of program performance begins with instruction-level performance. The basics of pipeline and cache timing serve as the foundation for our studies of larger program units.

We use as an example a simple data compression unit, concentrating on the programming of the core compression algorithm.

### 1.5.3 Chapter 4: Computing platforms

[Chapter 4](#) looks at the combined hardware and software platform for embedded computing. The microprocessor is very important, but only part of a system that includes memory, I/O devices, and low-level software. We need to understand the basic characteristics of the platform before we move on to build sophisticated systems.

The basic embedded computing platform includes a microprocessor, I/O hardware, I/O driver software, and memory. Application-specific software and hardware can be added to this platform to turn it into an embedded computing platform. The

microprocessor is at the center of both the hardware and software structure of the embedded computing system. The CPU controls the bus that connects to memory and I/O devices; the CPU also runs software that talks to the devices. In particular, I/O is central to embedded computing. Many aspects of I/O are not typically studied in modern computer architecture courses, so we need to master the basic concepts of input and output before we can design embedded systems.

Chapter 4 covers several important aspects of the platform:

- We study in detail how the CPU talks to memory and devices using the microprocessor **bus**.
- Based on our knowledge of bus operation, we study the structure of the **memory system** and types of **memory components**.
- We look at basic techniques for embedded system **design** and **debugging**.
- We study system-level performance analysis to see how bus and memory transactions affect the execution time of systems.

We illustrate these principles using two design examples. We use an alarm clock as a simple example of an appliance built on an embedded computing platform. We consider a jet engine controller as a more sophisticated example of a microprocessor plus bus used as an embedded computing platform.

#### 1.5.4 Chapter 5: Program design and analysis

Chapter 5 moves on to the software side of the computer system to understand how complex sequences of operations are executed as programs. Given the challenges of embedded programming—meeting strict performance goals, minimizing program size, reducing power consumption—this is an especially important topic. We build upon the fundamentals of computer architecture to understand how to design embedded programs.

- We develop some basic software components—data structures and their associated routines—that are useful in embedded software.
- As part of our study of the relationship between programs and instructions, we introduce a model for high-level language programs known as the **control/data flow graph (CDFG)**. We use the CDFG model extensively to help us analyze and optimize programs.
- Because embedded programs are increasingly written in higher-level languages, we look at the processes for compiling, assembling, and linking to understand how high-level language programs are translated into instructions and data. Some of the discussion surveys basic techniques for translating high-level language programs. We also spend time on compilation techniques designed specifically to meet embedded system challenges.
- We develop techniques for the **performance analysis** of programs. It is difficult to determine the speed of a program simply by examining its source code. We learn how to use a combination of the source code, its assembly language

implementation, and expected data inputs to analyze program execution time. We also study some basic techniques for optimizing program performance.

- An important topic related to performance analysis is **power analysis**. We build on performance analysis methods to learn how to estimate the power consumption of programs.
- It is critical that the programs that we design function correctly. The CDFG and techniques we have learned for performance analysis are related to techniques for **testing programs**. We develop techniques that can methodically develop a set of tests for a program in order to excise bugs.

At this point, we can consider the performance of a complete program. We introduce the concept of worst-case execution time as a basic measure of program execution time.

We use two design examples in [Chapter 5](#). The first is a simple software modem. A modem translates between the digital world of the microprocessor and the analog transmission scheme of the telephone network. Rather than using analog electronics to build a modem, we can use a microprocessor and special-purpose software. Because the modem has strict real-time deadlines, this example lets us exercise our knowledge of the microprocessor and of program analysis. The second is a digital still camera, which is considerably more complex than the modem, both in the algorithms it must execute and the variety of tasks it must perform.

### 1.5.5 [Chapter 6: Processes and operating systems](#)

[Chapter 6](#) builds on our knowledge of programs to study a special type of software component, the **process**, and operating systems that use processes to create systems. A process is an execution of a program; an embedded system may have several processes running concurrently. A separate **real-time operating system (RTOS)** controls when the processes run on the CPU. Processes are important to embedded system design because they help us juggle multiple events happening at the same time. A real-time embedded system that is designed without processes usually ends up as a mess of spaghetti code that doesn't operate properly.

We study the basic concepts of processes and process-based design in this chapter:

- We begin by introducing the **process abstraction**. A process is defined by a combination of the program being executed and the current state of the program. We learn how to switch contexts between processes.
- In order to make use of processes, we must be able to **schedule** them. We discuss process priorities and how they can be used to guide scheduling.
- We cover the fundamentals of **interprocess communication**, including the various styles of communication and how they can be implemented. We also look at the various uses of these interprocess communication mechanisms in systems.
- The real-time operating system (RTOS) is the software component that implements process abstraction and scheduling. We study how RTOSs implement schedules, how programs interface to the operating system, and how we can

evaluate the performance of systems built from RTOSs. We also survey some examples of RTOSs.

Tasks introduce a new level of complexity to performance analysis. Our study of real-time scheduling provides an important foundation for the study of multitasking systems.

[Chapter 6](#) analyzes an engine control unit for an automobile. This unit must control the fuel injectors and spark plugs of an engine based upon a set of inputs. Relatively complex formulas govern its behavior, all of which must be evaluated in real time. The deadlines for these tasks vary over several orders of magnitude.

### 1.5.6 [Chapter 7: System design techniques](#)

[Chapter 7](#) studies the design of large, complex embedded systems. We introduce important concepts that are essential for the successful completion of large embedded system projects, and we use those techniques to help us integrate the knowledge obtained throughout the book.

This chapter delves into several topics related to large-scale embedded system design:

- We revisit the topic of **design methodologies**. Based on our more detailed knowledge of embedded system design, we can better understand the role of methodology and the possible variations in methodologies.
- We study system **requirements analysis and specification methods**. Proper specifications become increasingly important as system complexity grows. More formal specification techniques help us capture intent clearly, consistently, and unambiguously. System analysis methodologies provide a framework for understanding specifications and assessing their completeness.
- We look at some **system modeling** methods that allow us to capture a design at several levels of abstraction.
- We consider **system analysis** and the **design of architectures** that meet our functional and nonfunctional requirements.
- We look at **quality assurance** techniques. The program testing techniques covered in [Chapter 5](#) are a good foundation but may not scale easily to complex systems. Additional methods are required to ensure that we exercise complex systems to shake out bugs.
- We consider **safety** and **security** as aspects of **dependability**.

### 1.5.7 [Chapter 8: Internet-of-Things](#)

The Internet-of-Things has emerged as a key application area for embedded computing. We look at the range of applications covered by IoT. We study wireless networks used to connect to IoT devices. We review the basics of databases that are used to tie together devices into IoT systems. As a design example, we consider a smart home with a variety of sensors.

### 1.5.9 Chapter 9: Automotive and aerospace systems

Automobiles and airplanes are important examples of networked control systems and of safety-critical embedded systems. We will see how cars and airplanes are operated using multiple networks and many communicating processors of various types. We look at the structure of some important networks used in distributed embedded computing, such as the controller area network widely used in cars and the inter-integrated circuit network for consumer electronics. We consider safety and security in vehicles.

### 1.5.10 Chapter 10: Embedded multiprocessors

The final chapter is an advanced topic that may be left out of initial studies and introductory classes without losing the basic principles underlying embedded computing. Many embedded systems are multiprocessors—computer systems with more than one processing element. The multiprocessor may use CPUs and DSPs; it may also include nonprogrammable elements known as **accelerators**. Multiprocessors are often more energy efficient and less expensive than platforms that try to do all the required computing on one big CPU. We look at the architectures of multiprocessors as well as the programming challenges of single-chip multiprocessors.

The chapter analyzes an accelerator for use in a video compression system. Digital video requires performing a huge number of operations in real time; video also requires large volumes of data transfers. As such, it provides a good way to study not only the design of the accelerator itself, but also how it fits into the overall system.

---

## 1.6 Summary

Embedded microprocessors are everywhere. Microprocessors allow sophisticated algorithms and user interfaces to be added inexpensively to an amazing variety of products. Microprocessors also help reduce design complexity and time by separating out hardware and software design. Embedded system design is much more complex than programming PCs because we must meet multiple design constraints, including performance, cost, and so on. In the remainder of this book, we build a set of techniques from the bottom up that will allow us to conceive, design, and implement sophisticated microprocessor-based systems.

---

## What we learned

- Embedded computing can be fun. It can also be difficult thanks to the combination of complex functionality and strict constraints that we must satisfy.
- Trying to hack together a complex embedded system probably won't work. You need to master a number of skills and understand the design process.

- Your system must meet certain functional requirements, such as features. It may also have to perform tasks to meet deadlines, limit its power consumption, be of a certain size, or meet other nonfunctional requirements.
  - A hierarchical design process takes the design through several levels of abstraction. You may need to do both top-down and bottom-up design.
  - We use UML to describe designs at several levels of abstraction.
  - This book takes a bottom-up view of embedded system design.
- 

## Further reading

Koopman [Koo10] describes in detail the phases of embedded computing system development. Spasov [Spa99] describes how 68HC11 microcontrollers are used in Canon EOS cameras. Douglass [Dou98] gives a good introduction to UML for embedded systems. Other foundational books on object-oriented design include Rumbaugh et al. [Rum91], Booch [Boo91], Shlaer and Mellor [Shl92], and Selic et al. [Sel94]. Bruce Schneier's book *Applied Cryptography* [Sch96] is an outstanding reference on the field.

---

## Questions

- Q1-1** Briefly describe the distinction between requirements and specifications.
- Q1-2** Give an example of a requirement on a smart voice command speaker.
- Q1-3** Give an example of a requirement on a smartphone camera.
- Q1-4** How could a security breach on a commercial airliner's Wi-Fi network result in a safety problem for the airplane?
- Q1-5** Given an example of a specification on a smart speaker, giving both type of specification and any required values. Take your example from an existing product and identify that product.
- Q1-6** Given an example of a specification on a smartphone camera, giving both type of specification and any required values. Take your example from an existing product and identify that product.
- Q1-7** Briefly describe the distinction between specification and architecture.
- Q1-8** At what stage of the design methodology would we determine what type of CPU to use?
- Q1-9** At what stage of the design methodology would we choose a programming language?

- Q1-10** Should an embedded computing system include software designed in more than one programming language? Justify your answer.
- Q1-11** At what stage of the design methodology would we test our design for functional correctness?
- Q1-12** Compare and contrast top-down and bottom-up design.
- Q1-13** Give an example of a design problem that is best solved using top-down techniques.
- Q1-14** Give an example of a design problem that is best solved using bottom-up techniques.
- Q1-15** Provide a concrete example of how bottom-up information from the software programming phase of design may be useful in refining the architectural design.
- Q1-16** Give a concrete example of how bottom-up information from I/O device hardware design may be useful in refining the architectural design.
- Q1-17** Create a UML state diagram for the *issue-command()* behavior of the *Controller* class of Fig. 1.27.
- Q1-18** Show how a *Set-speed* command flows through the refined class structure described in Fig. 1.18, moving from a change on the front panel to the required changes on the train:
  - Show it in the form of a collaboration diagram.
  - Show it in the form of a sequence diagram.
- Q1-19** Show how a *Set-inertia* command flows through the refined class structure described in Fig. 1.18, moving from a change on the front panel to the required changes on the train:
  - Show it in the form of a collaboration diagram.
  - Show it in the form of a sequence diagram.
- Q1-20** Show how an *Estop* command flows through the refined class structure described in Fig. 1.18, moving from a change on the front panel to the required changes on the train:
  - Show it in the form of a collaboration diagram.
  - Show it in the form of a sequence diagram.
- Q1-21** Draw a state diagram for a behavior that sends the command bits on the track. The machine should generate the address, generate the correct message type, include the parameters, and generate the ECC.
- Q1-22** Draw a state diagram for a behavior that parses the bits received by the train. The machine should check the address, determine the message type, read the parameters, and check the ECC.

- Q1-23** Draw a state diagram for the train receiver class.
- Q1-24** Draw a class diagram for the classes required in a basic microwave oven. The system should be able to set the microwave power level between one and nine and time a cooking run up to 59 min and 59 s in 1-s increments. Include classes for the physical interfaces to the front panel, door latch, and microwave unit.
- Q1-25** Draw a collaboration diagram for the microwave oven of Q1-23. The diagram should show the flow of messages when the user first sets the power level to seven, then sets the timer to 2:30, and then runs the oven.

---

## Lab exercises

- L1-1** How would you measure the execution speed of a program running on a microprocessor? You may not always have a system clock available to measure time. To experiment, write a piece of code that performs some function that takes a small but measurable amount of time, such as a matrix algebra function. Compile and load the code onto a microprocessor, and then try to observe the behavior of the code on the microprocessor's pins.
- L1-2** Complete the detailed specification of the train controller that was started in [Section 1.4](#). Show all the required classes. Specify the behaviors for those classes. Use object diagrams to show the instantiated objects in the complete system. Develop at least one sequence diagram to show system operation.
- L1-3** Develop a requirements description for an interesting device. The device may be a household appliance, a computer peripheral, or whatever you wish.
- L1-4** Write a specification for an interesting device in UML. Try to use a variety of UML diagrams, including class diagrams, object diagrams, sequence diagrams, and so on.

# Instruction Sets

# 2

## CHAPTER POINTS

---

- A brief review of computer architecture taxonomy and assembly language.
  - Four very different architectures: Arm, PIC16F, TI C55x, and TI C64x.
- 

## 2.1 Introduction

In this chapter, we begin our study of microprocessors by studying **instruction sets**: the programmer’s interface to the hardware. Although we hope to do as much programming in high-level languages as possible, the instruction set is the key to analyzing the performance of programs. By understanding the types of instructions that the **central processing unit (CPU)** provides, we gain insight into alternative ways to implement a particular function.

We use four CPUs as examples. The Arm processor [Fur96][Jag95][Slo04] is widely used in cell phones and many other systems. (The Arm architecture comes in several versions; we will concentrate on Arm version 7 (ARMv7) but will consider features of other versions as well.) The PIC16F is an 8-bit microprocessor that is designed for very efficient, low-cost implementations. The Texas Instruments (TI) C55x and C64x are two very different **digital signal processors (DSPs)** [Tex01] [Tex02][Tex10]. The C55x has a more traditional architecture, whereas the C64x uses very-long instruction word (VLIW) techniques to provide high-performance parallel processing.

We will start with a brief introduction to the terminology of computer architectures and instruction sets, followed by detailed descriptions of the Arm, PIC16F, C55x, and C64x instruction sets.

---

## 2.2 Preliminaries

In this section, we will look at some general concepts in computer architecture and programming, including different styles of computer architecture and the nature of assembly language.

**von Neumann architectures**

**Harvard architectures**

### 2.2.1 Computer architecture taxonomy

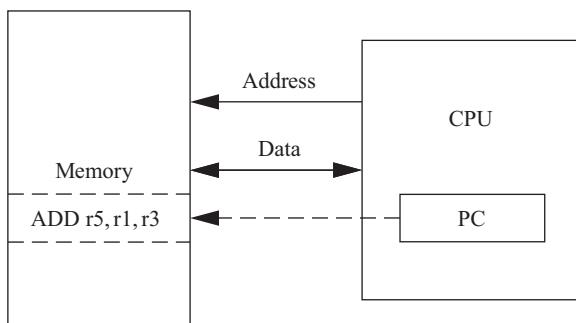
Before we delve into the details of microprocessor instruction sets, it is helpful to develop some basic terminology. We do so by reviewing a taxonomy of the basic ways that we can organize a computer.

A block diagram for one type of computer is shown in Fig. 2.1. The computing system consists of a CPU and a **memory**. The memory holds both data and instructions, and can be read or written when given an address. A computer whose memory holds both data and instructions is known as a **von Neumann** machine.

The CPU has several internal **registers** that store values that are used internally. One of those registers is the **program counter (PC)**, which holds the address of an instruction in memory. The CPU fetches the instruction from memory, decodes the instruction, and executes it. The PC does not directly determine what the machine does next, but only indirectly by pointing to an instruction in memory. By changing only the instructions in memory, we can change what the CPU does. It is this separation of the instruction memory from the CPU that distinguishes a stored-program computer from a general finite-state machine.

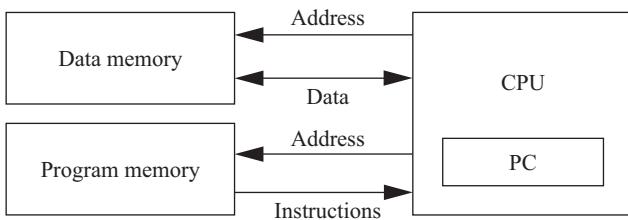
An alternative to the von Neumann style of organizing computers is the **Harvard architecture**, which is nearly as old as the von Neumann architecture. As shown in Fig. 2.2, a Harvard machine has separate memories for the data and program. The PC points to program memory, not data memory. As a result, it is harder to write self-modifying programs (programs that write data values and then use those values as instructions) on Harvard machines.

Harvard architectures are widely used today for one very simple reason: the separation of program and data memories provides higher performance for digital signal processing. Processing signals in real time places great strains on the data access system in two ways: first, large amounts of data flow through the CPU; second, those data must be processed at precise intervals, not just when the CPU gets around to it. Data sets that arrive continuously and periodically are called **streaming data**. Having two



**FIGURE 2.1**

A von Neumann architecture computer.

**FIGURE 2.2**

A Harvard architecture.

memories with separate ports provides higher memory bandwidth; not making data and memory compete for the same port also makes it easier to move the data at the proper times. DSPs constitute a large fraction of all microprocessors sold today as they are widely used in audio and image/video processing, and most of these DSPs are Harvard architectures.

#### RISC vs. CISC

Another axis along which we can organize computer architectures relates to their instructions and how they are executed. Many early computer architectures were what is known today as **complex instruction set computers (CISCs)**. These machines provided a variety of instructions that may perform very complex tasks, such as string searching. They also generally used a number of different instruction formats of varying lengths. One of the advances in the development of high-performance microprocessors was the concept of **reduced instruction set computers (RISCs)**. These computers tended to provide somewhat fewer and simpler instructions. RISC machines generally use **load/store** instruction sets: operations cannot be performed directly on memory locations, only on registers. The instructions were also chosen so that they could be efficiently executed in **pipelined** processors. Early RISC designs substantially outperformed the CISC designs of the period. As it turns out, we can use RISC techniques to efficiently execute at least a common subset of CISC instruction sets, so the performance gap between RISC-like and CISC-like instruction sets has narrowed somewhat.

#### Instruction set characteristics

Beyond the basic RISC/CISC characterization, we can classify computers by several characteristics of their instruction sets. The instruction set of the computer defines the interface between the software modules and underlying hardware; the instructions define what the hardware will do under certain circumstances. Instructions can have a variety of characteristics, including:

- fixed vs. variable length;
- addressing modes;
- numbers of operands;
- types of operations supported.

#### Word length

We often characterize architectures by their data word length: 4-bit, 8-bit, 16-bit, 32-bit, and so on. In some cases, the lengths of a data word, an instruction, and an

### Little-endian vs. big-endian

### Instruction execution

### Architectures and implementations

### CPUs and systems

address are the same. Particularly for computers that are designed to operate on smaller words, instructions and addresses may be longer than the basic data word.

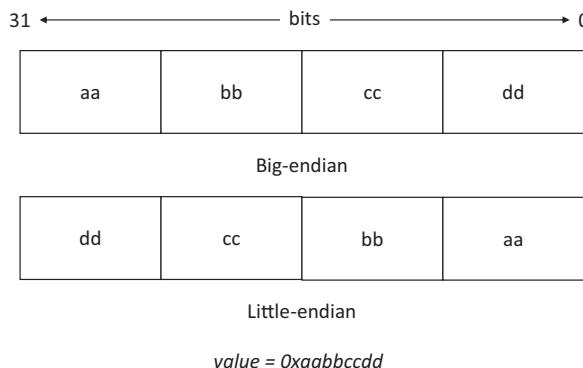
One subtle but important characterization of architectures is the way that they number bits, bytes, and words. Cohen [Coh81] introduced the terms **little-endian** mode (with the lowest-order byte residing in the low-order bits of the word) and **big-endian** mode (with the lowest-order byte stored in the highest bits of the word). Fig. 2.3 illustrates the two representations of a value 0xaabbccdd (where the prefix 0x is the C language notation for hexadecimal).

We can also characterize processors by their instruction execution, which is a separate concern from the instruction set. A **single-issue** processor executes one instruction at a time. Although it may have several instructions at different stages of execution, only one can be at any particular stage of execution. Several other types of processors allow **multiple-issue** instruction. A **superscalar** processor uses specialized logic to identify instructions at run time that can be executed simultaneously. A **VLIW** processor relies on the compiler to determine what combinations of instructions can be legally executed together. Superscalar processors often use too much energy and are too expensive for widespread use in embedded systems. VLIW processors are often used in high-performance embedded computing.

The set of registers that is available for use by programs is called the **programming model**, also known as the **programmer model**. (The CPU has many other registers that are used for internal operations and are unavailable to programmers.)

There may be several different implementations of an architecture. In fact, the architecture definition serves to define those characteristics that must be true of all implementations and what may vary among implementations. Different CPUs may offer different clock speeds, different cache configurations, changes to the bus or interrupt lines, and many other changes that can make one model of CPU more attractive than another for any given application.

The CPU is only part of a complete computer system. In addition to the memory, we also need I/O devices to build a useful system. We can build a computer from



**FIGURE 2.3**

Big-endian and little-endian number representations.

several different chips, but many useful computer systems come on a single chip. The term **microcontroller** is usually used to refer to a computer system chip that includes a CPU, flash, RAM, and some I/O devices. A system-on-chip generally refers to a larger processor; a multiprocessor system-on-chip includes multiple processing elements.

### 2.2.2 Assembly languages

[Fig. 2.4](#) shows a fragment of Arm assembly code to remind us of the basic features of assembly languages. Assembly languages usually share the same basic features:

- One instruction appears per line.
- **Labels**, which give names to memory locations, start in the first column.
- Instructions must start in the second column or after to distinguish them from labels.
- Comments run from some designated comment character (; in the case of Arm) to the end of the line.

Assembly language follows this relatively structured form to make it easy for the **assembler** to parse the program and to consider most aspects of the program line by line. (It should be remembered that early assemblers were written in assembly language to fit in a very small amount of memory. Those early restrictions have been carried into modern assembly languages by tradition.) [Fig. 2.5](#) shows the format of an Arm data processing instruction such as an **ADD**. For the instruction

```
ADDGT r0,r3,#5
```

the *cond* field would be set according to the GT condition (1100), the *opcode* field would be set to the binary code for the **ADD** instruction (0100), the first *operand* register *Rn* would be set to 3 to represent *r3*, the destination register *Rd* would be set to 0 for *r0*, and the *operand 2* field would be set to the immediate value of 5.

Assemblers must also provide some **pseudo-ops**, also known as **assembler directives**, to help programmers to create complete assembly language programs. An example of a pseudo-op is one that allows data values to be loaded into memory locations. These allow constants, for example, to be set into memory. An example of a memory allocation pseudo-op for Arm is:

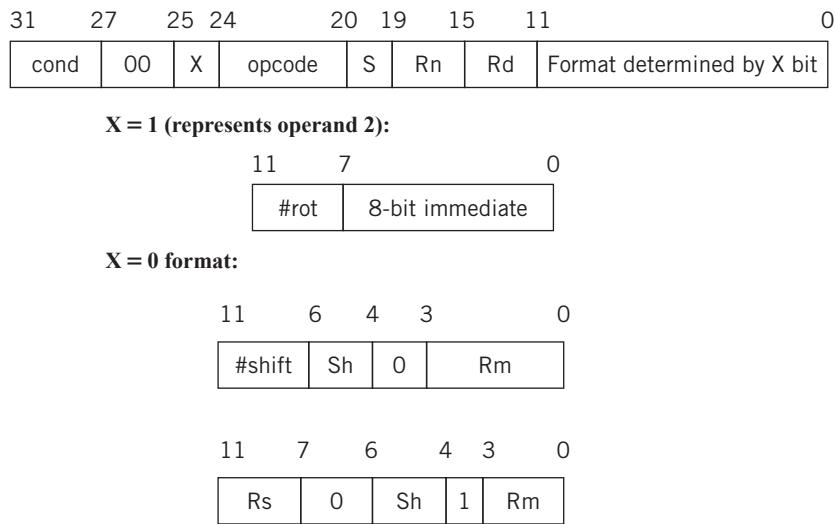
```
BIGBLOCK % 10
```

```
label1    ADR r4,c
          LDR r0,[r4]      ; a comment
          ADR r4,d
          LDR r1,[r4]
          SUB r0,r0,r1      ; another comment
```

**FIGURE 2.4**

---

An example of Arm assembly language.

**FIGURE 2.5**

Format of an Arm data processing instruction.

The `Arm %` pseudo-op allocates a block of memory of the size that is specified by the operand and initializes those locations to zero.

### 2.2.3 VLIW processors

CPUs can execute programs faster if they can execute more than one instruction at a time. If the operands of one instruction depend on the results of a previous instruction, the CPU cannot start the new instruction until the earlier instruction has finished. However, adjacent instructions may not depend directly on each other. In this case, the CPU can execute several simultaneously.

Several different techniques have been developed to parallelize execution. Desktop and laptop computers often use superscalar execution. A superscalar processor scans the program during execution to find sets of instructions that can be executed together. Digital signal processing systems are more likely to use **very-long instruction word (VLIW)** processors. These processors rely on the compiler to identify sets of instructions that can be executed in parallel. Superscalar processors can find parallelism that VLIW processors cannot, as some instructions may be independent in some situations and not others. However, superscalar processors are more expensive in terms of both cost and energy consumption. Because it is relatively easy to extract

**Packets**

parallelism from many DSP applications, the efficiency of VLIW processors can be leveraged more easily by digital signal processing software.

**Inter-instruction dependencies**

In modern terminology, a set of instructions is bundled together into a **VLIW packet**, which is a set of instructions that may be executed together. The execution of the next packet will not start until all of the instructions in the current packet have finished executing. The compiler identifies packets by analyzing the program to determine sets of instructions that can always execute together.

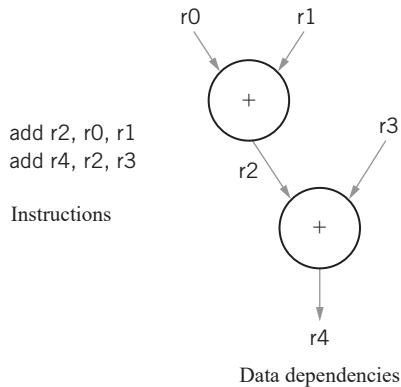
To understand parallel execution, let's first understand what constrains instructions from executing in parallel. A **data dependency** is a relationship among the data that are operated on by instructions. In the example of Fig. 2.6, the first instruction writes into  $r_2$  while the second instruction reads from it. As a result, the first instruction must finish before the second instruction can perform its addition. The data dependency graph shows the order in which these operations must be performed.

Branches can also introduce **control dependencies**. Consider this simple branch:

```
bnz r3,foo
      add r0,r1,r2
  foo: ...
```

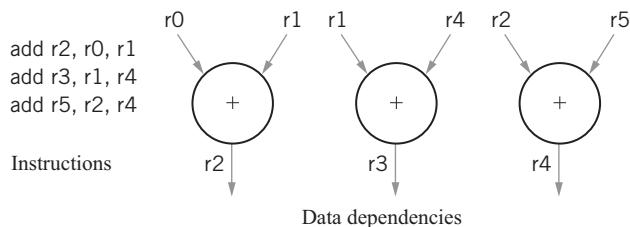
The add instruction is executed only if the branch instruction that precedes it does not take its branch.

Opportunities for parallelism arise because many combinations of instructions do not introduce data or control dependencies. The natural grouping of assignments in the source code suggests some opportunities for parallelism that can also be influenced by how the object code uses registers. Consider the example of Fig. 2.7.



**FIGURE 2.6**

Data dependencies and order of instruction execution.

**FIGURE 2.7**


---

Instructions without data dependencies.

Although these instructions use common input registers, the result of one instruction does not affect the result of the other instructions.

#### VLIW and embedded computing

A number of different processors have implemented VLIW execution modes and these processors have been used in many embedded computing systems. Because the processor does not have to analyze data dependencies at run time, VLIW processors are smaller and consume less power than superscalar processors. VLIW is very well suited to many signal processing and multimedia applications. For example, cellular telephone base stations must perform the same processing on many parallel data streams. Channel processing is easily mapped onto VLIW processors because there are no data dependencies between the different signal channels.

---

## 2.3 Arm processor

In this section, we concentrate on the Arm processor. Arm is a family of RISC architectures that have been developed over many years. Arm does not manufacture its own chips; rather, it licenses its architecture to companies who either manufacture the CPU itself or integrate the Arm processor into a larger system.

The textual description of instructions, as opposed to their binary representation, is called an assembly language. Arm instructions are written one per line, starting after the first column. Comments begin with a semicolon and continue to the end of the line. A label, which gives a name to a memory location, is placed at the beginning of the line, starting in the first column:

```
LDR r0,[r8] ; a comment
label ADD r4,r0,r1W
```

### 2.3.1 Processor and memory organization

This discussion will concentrate on the ARMv7 architecture [ARM96]. This version includes a 32-bit instruction set and a 16-bit Thumb instruction set. ARMv8 supports a 64-bit instruction set mode and a 32-bit mode that is compatible with earlier architectures such as ARMv7 [ARM13B]. ARMv9 provides support for

security and for vector operations useful for scientific computing and artificial intelligence [Arm21].

The ARMv7-A architecture supports four basic types of data:

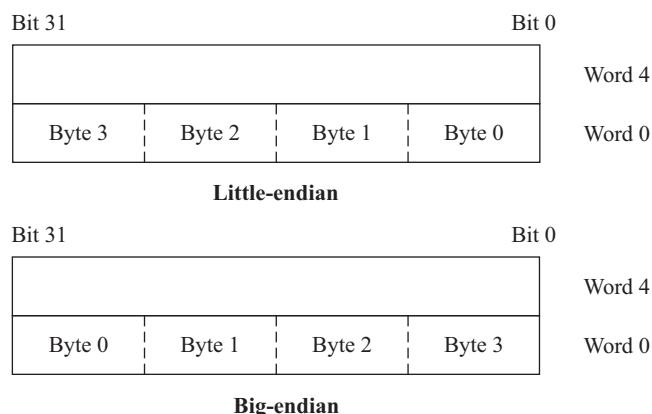
- an 8-bit byte;
- a 16-bit half-word;
- a 32-bit long word;
- a 64-bit doubleword.

ARMv7 allows addresses to be 32 bits long. An address refers to a byte, not a word. Therefore, the word 0 in the Arm address space is at location 0, the word 1 is at 4, the word 2 is at 8, and so on. (As a result, the PC is incremented by 4 in the absence of a branch.) The Arm processor can be configured at power-up to address the bytes in a word in either the little-endian or big-endian mode, as shown in Fig. 2.8.

General-purpose computers have sophisticated instruction sets. Some of this sophistication is required simply to provide the functionality of a general computer, whereas other aspects of instruction sets may be provided to increase the performance, reduce the code size, or otherwise improve the program characteristics. In this section, we concentrate on the functionality of the Arm instruction set.

### 2.3.2 Data operations

Arithmetic and logical operations in C are performed in variables. Variables are implemented as memory locations. Therefore, to be able to write instructions to perform C expressions and assignments, we must consider both the arithmetic and logical instructions, as well as instructions for reading and writing memory.



**FIGURE 2.8**

Byte organizations within an Arm word.

```

int a, b, c, x, y, z;
x = (a + b) - c;
y = a*(b + c);
z = (a << 2) | (b & 15);

```

**FIGURE 2.9**

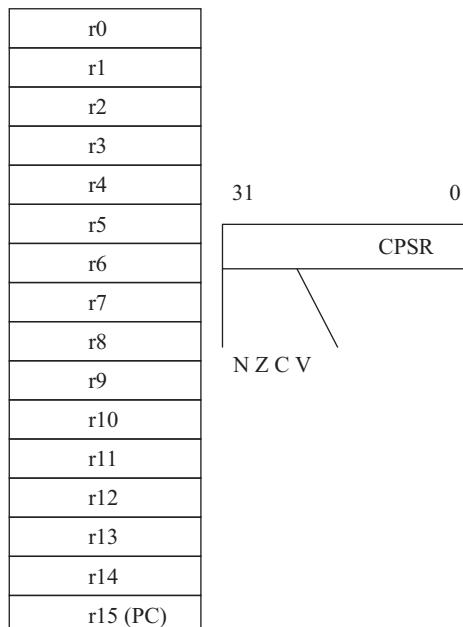
A C fragment with data operations.

**Fig. 2.9** shows a sample fragment of C code with data declarations and several assignment statements. The variables *a*, *b*, *c*, *x*, *y*, and *z* all become data locations in memory. In most cases, the data are kept relatively separate from instructions in the program's memory image.

#### Arm programming model

In the Arm processor, arithmetic and logical operations cannot be performed directly on memory locations. While some processors allow such operations to directly reference main memory, Arm is a **load–store architecture**, so data operands must first be loaded into the CPU and then stored back into main memory to save the results.

**Fig. 2.10** shows the registers in the ARMv7-R application-level programmers' model, including 16 registers, each of which is 32 bits long. The registers r0 through r12 are general purpose and identical. Except for r15, they are identical: any operation

**FIGURE 2.10**

The basic Arm programming model.

that can be done on one of them can be done on the others. Three other registers have special uses:

r13 is the **stack pointer (sp)** used to perform stack operations. While this register could be used as a general-purpose register, uses other than as a stack pointer are deprecated.

r14 is the link register to return information from subroutines.

r15 is the program counter. The PC should, of course, not be overwritten for use in data operations. However, giving the PC the properties of a general-purpose register allows the PC value to be used as an operand in computations, which can make certain programming tasks easier.

Another important register that is defined as part of the system-level programmers' model is the **current program status register (CPSR)**. This register is set automatically during every arithmetic, logical, or shifting operation. The top four bits of the CPSR hold the following useful information about the results of that arithmetic or logical operation:

- The negative (N) bit is set when the result is negative in two's-complement arithmetic.
- The zero (Z) bit is set when every bit of the result is zero.
- The carry (C) bit is set when there is a carry out of the operation.
- The overflow (V) bit is set when an arithmetic operation results in an overflow.

These bits can be used to check the results of an arithmetic operation easily. However, if a chain of arithmetic or logical operations is performed and the intermediate states of the CPSR bits are important, they must be checked at each step because the next operation changes the CPSR values.

Example 2.1 illustrates the computation of CPSR bits.

---

### Example 2.1 Status Bit Computation in Arm

An Arm word is 32 bits. In C notation, a hexadecimal number starts with 0x, such as 0xffffffff, which is a two's-complement representation of -1 in a 32-bit word.

Here are some sample calculations:

- $-1 + 1 = 0$ : Written in 32-bit format, this becomes 0xffffffff + 0x1 = 0x0, giving the CPSR value of NZCV = 0110.
- $0 - 1 = -1$ : 0x0 - 0x1 = 0xffffffff, with NZCV = 1000.
- $(2^{31} - 1) + 1 = -2^{31}$ : 0x7fffffff + 0x1 = 0x80000000, with NZCV = 0101.

---

The basic form of a data instruction is simple:

ADD r0,r1,r2

This instruction sets register r0 to the sum of the values that are stored in r1 and r2. In addition to specifying registers as sources for operands, instructions may also

provide **immediate operands**, which encode a constant value directly into the instruction. For example,

```
ADD r0,r1,#2
```

sets r0 to r1 + 2.

The major data operations are summarized in Fig. 2.11. The arithmetic operations perform addition and subtraction; the with-carry versions include the current value of the carry bit in the computation. RSB performs a subtraction with the order of the two operands reversed, so that RSB r0,r1,r2 sets r0 to  $r2 - r1$ . The bitwise logical operations perform logical AND, OR, and XOR operations (the exclusive OR is called

ADD	Add
ADC	Add with carry
SUB	Subtract
SBC	Subtract with carry
RSB	Reverse subtract
RSC	Reverse subtract with carry
MUL	Multiply
MLA	Multiply and accumulate

**Arithmetic**

AND	Bit-wise and
ORR	Bit-wise or
EOR	Bit-wise exclusive-or
BIC	Bit clear

**Logical**

LSL	Logical shift left (zero fill)
LSR	Logical shift right (zero fill)
ASL	Arithmetic shift left
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right extended with C

**Shift/rotate**

**FIGURE 2.11**

Arm data instructions.

EOR). The `BIC` instruction stands for bit clear: `BIC r0,r1,r2` sets `r0` to `r1` and the bitwise NOT of `r2`. This instruction uses the second source operand as a mask: where a bit in the mask is 1, the corresponding bit in the first source operand is cleared. The `MUL` instruction multiplies two values, but with some restrictions: no operand may be an immediate and the two source operands must be different registers. The `MLA` instruction performs a multiply-accumulate operation, which is particularly useful in matrix operations and signal processing. The instruction

`MLA r0,r1,r2,r3`

sets `r0` to the value  $r1 \times r2 + r3$ .

The shift operations are not separate instructions; rather, shifts can be applied to arithmetic and logical instructions. The shift modifier is always applied to the second source operand. A left shift moves bits up towards the most significant bits, whereas a right shift moves bits down to the least significant bit in the word. The `LSL` and `LSR` modifiers perform left and right logical shifts, filling the least significant or most significant bits of the operand with zeros as appropriate. The arithmetic shift left is equivalent to an `LSL`, but the `ASR` copies the sign bit: if the sign is 0, a 0 is copied, and if the sign is 1, a 1 is copied. The rotate modifiers always rotate right, moving the bits that fall off the least significant bit up to the most significant bit in the word. The `RRX` modifier performs a 33-bit rotate, with the CPSR's C bit being inserted above the sign bit of the word; this allows the carry bit to be included in the rotation.

The instructions in Fig. 2.12 are comparison operands. They do not modify general-purpose registers, but only set the values of the NZCV bits of the CPSR register. The compare instruction `CMP r0, r1` computes  $r0 - r1$ , sets the status bits, and throws away the result of the subtraction. `CMN` uses an addition to set the status bits. `TST` performs a bitwise AND on the operands, whereas `TEQ` performs an exclusive-or.

Fig. 2.13 summarizes the Arm move instructions. The instruction `MOV r0,r1` sets the value of `r0` to the current value of `r1`. The `MVN` instruction complements the operand bits (one's complement) during the move.

Values are transferred between registers and memory using the load—store instructions summarized in Fig. 2.14. `LDRB` and `STRB` load and store bytes rather than whole words, whereas `LDRH` and `SDRH` operate on half-words and `LDRSH` extends the sign bit on loading. An Arm address may be 32 bits long. The Arm load and store instructions do not directly refer to main memory addresses, because a 32-bit address would not fit

<code>CMP</code>	Compare
<code>CMN</code>	Negated compare
<code>TST</code>	Bit-wise test
<code>TEQ</code>	Bit-wise negated test

FIGURE 2.12

Arm compare instructions.

MOV	Move
MVN	Move negated

**FIGURE 2.13**


---

Arm move instructions.

LDR	Load
STR	Store
LDRH	Load half-word
STRH	Store half-word
LDRSH	Load half-word signed
LDRB	Load byte
STRB	Store byte
ADR	Set register to address

**FIGURE 2.14**


---

Arm load—store instructions and pseudo-ops.

into an instruction that includes an opcode and operands. Instead, the Arm uses **register-indirect addressing**. In register-indirect addressing, the value that is stored in the register is used as the address to be fetched from memory; the result of that fetch is the desired operand value. Thus, as illustrated in Fig. 2.15, if we set  $r1 = 0x100$ , the instruction

`LDR r0,[r1]`

sets  $r0$  to the value of memory location  $0x100$ . Similarly, `STR r0,[r1]` would store the contents of  $r0$  in the memory location whose address is given in  $r1$ . There are several possible variations:

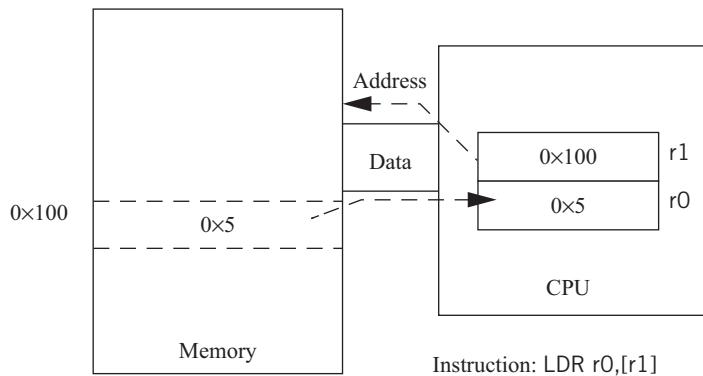
`LDR r0,[r1,-r2]`

loads  $r0$  from the address given by  $r1 - r2$ , and

`LDR r0,[r1,#4]`

loads  $r0$  from the address  $r1 + 4$ .

This begs the question of how we get an address into a register, as we need to be able to set a register to an arbitrary 32-bit value. In the Arm, the standard way to set a register to an address is by performing arithmetic on a register. One choice for the register to use for this operation is the PC. By adding or subtracting a constant equal to the distance between the current instruction (i.e., the instruction that is computing the

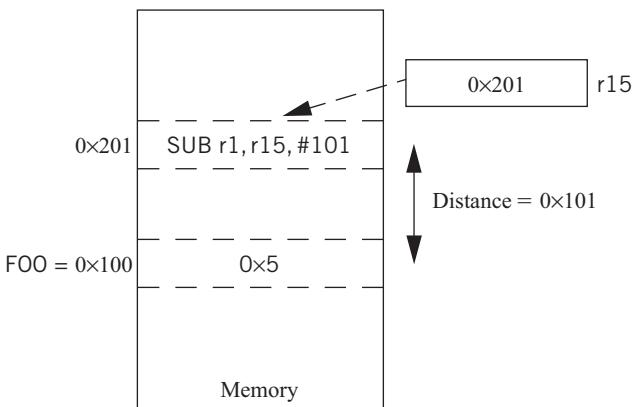
**FIGURE 2.15**

Register indirect addressing in the Arm.

address) and the desired location to or from the PC, we can generate the desired address without performing a load. The Arm programming system provides an `ADR` pseudo-op to simplify this step. Thus, as shown in Fig. 2.16, if we give location 0x100 the name `FOO`, we can use the pseudo-op

`ADR r1,FOO`

to perform the same function of loading `r1` with the address 0x100. Another technique is used in high-level languages like C. As we will see when we discuss procedure calls, languages use a mechanism called a **frame** to pass parameters between functions. For the moment, a simplified view of the process is sufficient. A register holds a frame pointer (`fp`) that points to the top of the frame; elements within the frame are

**FIGURE 2.16**

Computing an absolute address using the PC.

accessed using offsets from fp. The assembler syntax [fp, #-n] is used to take the nth location from fp.

Example 2.2 illustrates how to implement C assignments in Arm instructions.

---

### Example 2.2 C Assignments in Arm instructions

We will use the assignments of Fig. 2.9. The semicolon (;) begins a comment after an instruction, which continues to the end of that line. The C language statement

$x = (a + b) - c;$

can be implemented using r0 for a, r1 for b, r2 for c, and r3 for x. We also need registers for indirect addressing. In this case, we will reuse the same indirect addressing register, fp, for each variable load. The code must load the values of a, b, and c into these registers before performing the arithmetic, and it must store the value of x back into memory when it is done.

The following is code generated by the gcc compiler for this statement; comments have been added to illustrate their use. It uses fp to hold the variables: a is at -24, b is at -28, c is at -32, and x is at -36:

```
ldr r2, [fp, #-24] ; load r2
ldr r3, [fp, #-28] ; load r3
add r2, r2, r3 ; add, store result in r2
ldr r3, [fp, #-32] ;load r3
rsb r3, r3, r2 ; reverse subtract
str r3, [fp, #-36] ; store r3
```

The C language operation

$y = a * (b + c);$

can be coded similarly, but in this case, we will reuse more registers: r2 for both b and the term  $(b + c)$ ; and r3 for c, a, and the result y. Once again, we will use fp to store addresses for indirect addressing. The resulting code from gcc looks like this:

```
ldr r2, [fp, #-28]
ldr r3, [fp, #-32]
add r2, r2, r3
ldr r3, [fp, #-24]
mul r3, r2, r3
str r3, [fp, #-40]
```

The C language statement

$z = (a \ll 2) | (b \& 15);$

results in this gcc-generated code:

```
ldr r3, [fp, #-24]
mov r2, r3, asl #2
ldr r3, [fp, #-28]
and r3, r3, #15
orr r3, r2, r3
str r3, [fp, #-44]
```

---

### More addressing modes

We have already seen three addressing modes: register, immediate, and indirect. The Arm also supports several forms of **base-plus-offset addressing**, which is related to indirect addressing. However, rather than using a register value directly as an address, the register value is added to another value to form the address. For instance,

```
LDR r0,[r1,#16]
```

loads r0 with the value stored at location  $r1 + 16$ . Here, r1 is referred to as the **base** and the immediate value is the **offset**. When the offset is an immediate, it may have any value up to 4,096; another register may also be used as the offset. This addressing mode has two other variations: **auto-indexing** and **post-indexing**. Auto-indexing updates the base register, such that

```
LDR r0,[r1,#16]!
```

first adds 16 to the value of r1 and then uses that new value as the address. The ! operator causes the base register to be updated with the computed address so that it can be used again later. Our examples of base-plus-offset and auto-indexing instructions will fetch from the same memory location, but auto-indexing will also modify the value of the base register r1. Post-indexing does not perform the offset calculation until after the fetch has been performed. Consequently,

```
LDR r0,[r1],#16
```

will load r0 with the value stored at the memory location whose address is given by r1, and then add 16 to r1 and set r1 to the new value. In this case, the post-indexed mode fetches a different value than those of the other two examples, but ends up with the same final value for r1 as auto-indexing.

We have used the ADR pseudo-op to load addresses into registers to access variables because this leads to simple, easy-to-read code (at least by assembly language standards). Compilers tend to use other techniques to generate addresses, because they must deal with global and automatic variables.

### 2.3.3 Flow of control

The B (branch) instruction is the basic mechanism for changing the flow of control in Arm. The address that is the destination of the branch is often called the branch target. Branches are PC-relative: the branch specifies the offset from the current PC value to the branch target. The offset is in words, but because the Arm is byte-addressable, the offset is multiplied by four (shifted left two bits, actually) to form a byte address. Thus, the instruction

```
B #100
```

will add 400 to the current PC value.

EQ	Equals zero	Z = 1
NE	Not equal to zero	Z = 0
CS	Carry set	C = 1
CC	Carry clear	C = 0
MI	Minus	N = 1
PL	Nonnegative (plus)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 1 and Z = 0
LS	Unsigned lower or same	C = 0 or Z = 1
GE	Signed greater than or equal	N = V
LT	Signed less than	N ≠ V
GT	Signed greater than	Z = 0 and N = V
LE	Signed less than or equal	Z = 1 or N ≠ V

**FIGURE 2.17**


---

Condition codes in Arm.

We often wish to branch conditionally, based on the result of a given computation. The `if` statement is a common example. The Arm allows *any* instruction, including branches, to be executed conditionally. This allows branches as well as data operations to be conditional. Fig. 2.17 summarizes the condition codes.

We use Example 2.3 as a way to explore the uses of conditional execution.

---

### Example 2.3 Implementing `if` statement in Arm

We will use this `if` statement as an example:

```
if (a > b) {
    x = 5;
    y = c + d;
}
else x = c - d;
```

The implementation uses two blocks of code: one for the true case and another for the false case. Let us look at the gcc-generated code in sections. First, here is the compiler-generated code for the `a>b` test:

```
ldr r2, [fp, #-24]
ldr r3, [fp, #-28]
cmp r2, r3
bge .L2
```

Here is the code for the false block:

```
ldr    r3, [fp, #-32]
ldr    r2, [fp, #-36]
rsb    r3, r2, r3
str    r3, [fp, #-40]
.L3:
```

And here is the code for the true block:

```
.L2: mov r3, #5
      str r3, [fp, #-40]
      ldr r2, [fp, #-32]
      ldr r3, [fp, #-36]
      add r3, r2, r3
      str r3, [fp, #-44]
      b .L3
```

---

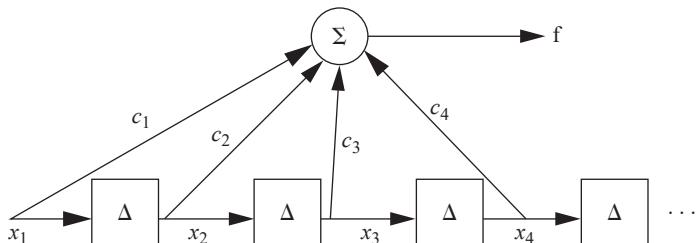
The loop is a very common C statement, particularly in signal processing code. Loops can be implemented naturally using conditional branches. Because loops often operate on values stored in arrays, loops are also a good illustration of another use of the base-plus-offset addressing mode. A simple but common use of a loop is in the **finite impulse response (FIR)** filter, which is explained in Application Example 2.1; the loop-based implementation of the FIR filter is described in Example 2.5.

### Application Example 2.1 FIR filters

A FIR is a commonly used method for processing signals; we make use of it in [Section 5.12](#). The FIR filter is a simple sum of products:

$$\sum_{1 \leq i \leq n} c_i x_i \quad (\text{Eq. 2.1})$$

In use as a filter, the  $x$ s are assumed to be samples of data taken periodically, while the  $c$ s are coefficients. This computation is usually drawn like this:



This representation assumes that the samples are arriving periodically and that the FIR filter output is computed once every time a new sample arrives. The  $\Delta$  boxes represent delay elements that store the recent samples to provide the  $x$ s. The delayed samples are individually multiplied by the  $c$ s and then summed to provide the filter output.

---

### Example 2.5 FIR filter for Arm

Here is the C code for an FIR filter:

```
for (i = 0, f = 0; i < N; i++)
    f = f + c[i] *x[i];
```

We can address the arrays *c* and *x* using base-plus-offset addressing: we will load one register with the address of the zeroth element of each array and use the register holding *i* as the offset.

Here is the gcc-generated code for the loop:

```
.LBB2:
    mov r3, #0
    str r3, [fp, #-24]
    mov r3, #0
    str r3, [fp, #-28]
.L2:
    ldr r3, [fp, #-24]
    cmp r3, #7
    ble .L5
    b .L3
.L5:
    ldr r3, [fp, #-24]
    mvn r2, #47
    mov r3, r3, asl #2
    sub r0, fp, #12
    add r3, r3, r0
    add r1, r3, r2
    ldr r3, [fp, #-24]
    mvn r2, #79
    mov r3, r3, asl #2
    sub r0, fp, #12
    add r3, r3, r0
    add r3, r3, r2
    ldr r2, [r1, #0]
    ldr r3, [r3, #0]
    mul r2, r3, r2
    ldr r3, [fp, #-28]
    add r3, r3, r2
    str r3, [fp, #-28]
    ldr r3, [fp, #-24]
    add r3, r3, #1
    str r3, [fp, #-24]
    b .L2
.L3:
```

---

The `mvn` instruction moves the bitwise complement of a value.

**C functions**

The other important class of a C statement to consider is the **function**. A C function returns a value (unless its return type is `void`); **subroutine** or **procedure** are the common names for such a construct when it does not return a value. Consider this simple use of a function in C:

```
x = a + b;  
foo(x);  
y = c - d;
```

A function returns to the code immediately after the function call; in this case, the assignment to `y`. A simple branch is insufficient because we would not know where to return. To return properly, we must save the PC value when the procedure/function is called and, when the procedure is finished, set the PC to the address of the instruction *just after* the call to the procedure. (You do not want to endlessly execute the procedure.)

The branch-and-link instruction is used in the Arm for procedure calls. For instance,

```
BL foo
```

will perform a branch and link to the code starting at location `foo` (using PC-relative addressing, of course). The branch and link is much like a branch, except that before branching, it stores the address of the instruction after the BL in `r14`. Thus, to return from a procedure, you simply move the value of `r14` to `r15`:

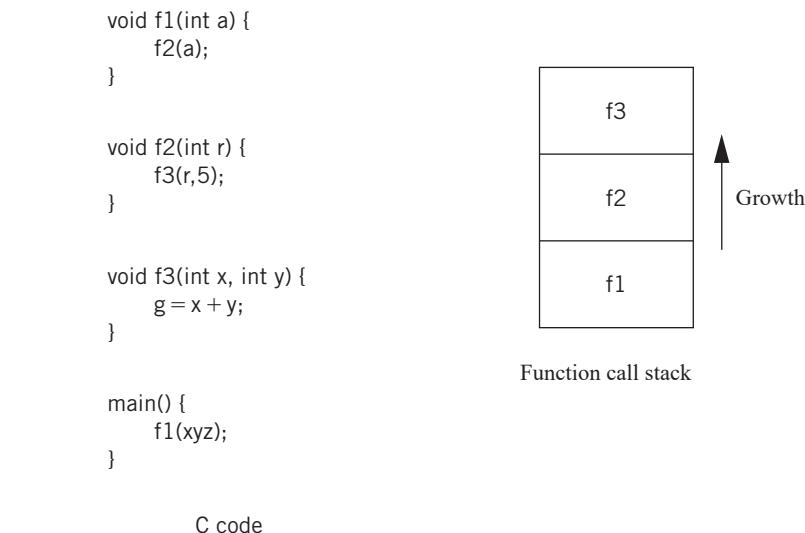
```
MOV r15,r14
```

Of course, you should not overwrite the PC value stored in `r14` during the procedure.

But this mechanism only lets us call procedures one level deep. If, for example, we call a C function within another C function, the second function call will overwrite `r14`, destroying the return address for the first function call. The standard procedure for allowing nested procedure calls (including recursive procedure calls) is to build a stack, as illustrated in Fig. 2.18. The C code shows a series of functions that call other functions: `f1()` calls `f2()`, which in turn calls `f3()`. The right side of the figure shows the state of the **procedure call stack** during the execution of `f3()`. The stack contains one **activation record** for each active procedure. When `f3()` finishes, it can pop the top of the stack to get its return address, leaving the return address for `f2()` waiting at the top of the stack for its return.

Most procedures need to pass parameters into the procedure and return values out of the procedure, as well as remember their return address.

We can also use the procedure call stack to pass parameters. The conventions that are used to pass values into and out of procedures are known as **procedure linkage**. To pass parameters into a procedure, the values can be pushed onto the stack just before the procedure call. Once the procedure returns, those values must be popped off the stack by the caller, because they may hide a return address or other useful information on the stack. A procedure may also need to save register values for the

**FIGURE 2.18**

Nested function calls and stacks.

registers that it modifies. The registers can be pushed onto the stack upon entry into the procedure and popped off the stack, restoring the previous values, before returning. Procedure stacks are typically built to grow down from high addresses.

Assembly language programmers can use any means they want to pass parameters. Compilers use standard mechanisms to ensure that any function may call any other. (If you write an assembly routine that calls a compiler-generated function, you must adhere to the compiler's procedure call standard.) The compiler passes parameters and return variables in a block of memory known as a **frame**. The frame is also used to allocate local variables. The stack elements are frames. A stack pointer (sp) defines the end of the current frame, while a frame pointer (fp) defines the end of the last frame. (The fp is technically necessary only if the stack frame can be grown by the procedure during execution.) The procedure can refer to an element in the frame by addressing relative to the sp. When a new procedure is called, the sp and fp are modified to push another frame onto the stack.

The Arm Procedure Call Standard (APCS) [Slo04] is a good illustration of a typical procedure linkage mechanism. Although the stack frames are in main memory, understanding how registers are used is key to understanding the mechanism, as explained below.

- r0–r3 are used to pass the first four parameters into the procedure. r0 is also used to hold the return value. If more than four parameters are required, they are put on the stack frame.
- r4–r7 hold register variables.

- r11 is the frame pointer and r13 is the stack pointer.
- r10 holds the limiting address on the stack size, which is used to check for stack overflows.

Other registers have additional uses in the protocol.

Example 2.6 illustrates the implementation of C functions and procedure calls.

---

### Example 2.6 Procedure calls in Arm

Here is a simple example of two procedures, one of which calls another:

```
void f2(int x) {
    int y;
    y = x+1;
}
void f1(int a) {
    f2(a);
}
```

This function has only one parameter, so x will be passed in r0. The variable y is local to the procedure, so it is put into the stack. The first part of the procedure sets up registers to manipulate the stack and then the procedure body is implemented. The ip register is the intra-procedure call scratch register, which is an alias for r12. Here is the code generated by the Arm gcc compiler for f2() with manually created comments to explain the code:

```
mov    ip, sp          ; set up f2()'s stack access
stmdfd sp!, {fp, ip, lr, pc}
sub    fp, ip, #4
sub    sp, sp, #8
str    r0, [fp, #-16]
ldr    r3, [fp, #-16]   ; get x
add    r3, r3, #1       ; add 1
str    r3, [fp, #-20]   ; assign to y
ldmea fp, {fp, sp, pc} ; return from f2()
```

And here is the code generated for f1():

```
mov    ip, sp          ; set up f1()'s stack access
stmdfd sp!, {fp, ip, lr, pc} ; stmdfd=store multiple full descending
sub    fp, ip, #4
sub    sp, sp, #4
str    r0, [fp, #-16]   ; save the value of a passed into f1()
ldr    r0, [fp, #-16]   ; load value of a for the f2() call
b1    f2                ; call f2()
ldmea fp, {fp, sp, pc} ; return from f1(), ldmea=load multiple
                        ; empty ascending
```

---

### 2.3.4 Advanced Arm features

Several models of Arm processors provide advanced features for a variety of applications.

DSP	Several extensions provide improved digital signal processing. Multiply–accumulate (MAC) instructions can perform a $16 \times 16$ or $32 \times 16$ MAC in one clock cycle. Saturation arithmetic can be performed with no overhead. A new instruction is used for arithmetic normalization.
SIMD	Multimedia operations are supported by single-instruction multiple-data (SIMD) operations. A single register is treated as having several smaller data elements, such as bytes. The same operation is applied to all elements in the register simultaneously.
NEON	NEON instructions go beyond the original SIMD instructions to provide a new set of registers and additional operations. The NEON unit has 32 registers, each 64 bits wide. Some operations also allow a pair of registers to be treated as a 128-bit vector. Data in a single register are treated as a vector of elements, each smaller than the original register, with the same operation performed in parallel on each vector element. For example, a 64-bit register can be used to perform SIMD operations on 8, 16, 32, 64, or single-precision floating-point numbers.
TrustZone	TrustZone extensions provide security features. A separate monitor mode allows the processor to enter a secure world to perform operations that are not permitted in the normal mode. A special instruction, namely the secure monitor call, can be used to enter the secure world, as can some exceptions.
Jazelle	The Jazelle instruction set allows the direct execution of 8-bit Java bytecodes. As a result, a bytecode interpreter does not need to be used to execute Java programs.
Cortex	The Cortex collection of processors is designed for compute-intensive applications:
	<ul style="list-style-type: none"> <li>• Cortex-A5 provides Jazelle execution of Java, floating-point processing, and NEON multimedia instructions.</li> <li>• Cortex-A8 is a dual-issue in-order superscalar processor.</li> <li>• Cortex-A9 can be used in a multiprocessor with up to four processing elements.</li> <li>• Cortex-A15 MPCore is a multicore processor with up to four CPUs.</li> <li>• The Cortex-R family is designed for real-time embedded computing. It provides SIMD operations for DSP, a hardware divider, and a memory protection unit for operating systems.</li> <li>• The Cortex-M family is designed for microcontroller-based systems that require low-cost and low-energy operation.</li> </ul>

---

## 2.4 PICmicro mid-range family

The PICmicro line includes several different families of microprocessors. We will concentrate on the mid-range family, the PIC16F family, which has an 8-bit word size and 14-bit instructions.

### 2.4.1 Processor and memory organization

The PIC16F family has a Harvard architecture with separate data and program memories. Models in this family hold up to 8,192 words of instruction memory in flash. An instruction word is 14 bits long. Data memory is byte addressable. They may have up to 368 bytes of data memory in static random-access memory and 256 bytes of electrically erasable programmable read-only memory (EEPROM) data memory.

Members of the family provide several low-power features, such as a sleep mode, the ability to select different clock oscillators to run at different speeds. They also provide security features such as code protection and component identification locations.

### 2.4.2 Data operations

**Address range**

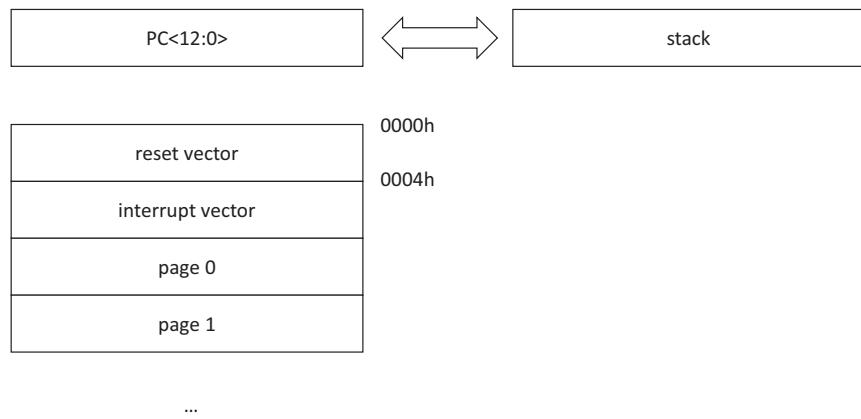
The PIC16F family uses a 13-bit program counter. Different members of the family provide different amounts of instruction or data memory: 2K instructions for the low-end models, 4K for medium, and 8K for large.

**Instruction space**

Fig. 2.19 shows the organization of the instruction space. The program counter can be loaded from a stack. The lowest addresses in memory hold the reset and interrupt vectors. The rest of memory is divided into four pages. The low-end devices have access only to page 0; the medium-range devices have access only to pages 1 and 2; the high-end devices have access to all four pages.

**Data space**

The PIC16F data memory is divided into four banks. Two bits of the STATUS register, RP<1:0>, select which bank is used. The PIC documentation uses the term **general-purpose register** to mean a data memory location. It also uses the term **file register** to refer to a location in the general-purpose register file. The lowest 32



**FIGURE 2.19**

Instruction space for the PIC16F.

locations of each bank contain **special function registers** that perform many different operations, primarily for the I/O devices. The remainder of each bank contains general-purpose registers.

Because different members of the family support different amounts of data memory, not all of the bank locations are available in all models. All models implement the special function registers in all banks. However, not all of the banks make general-purpose registers available. The low-end models provide general-purpose registers only in bank 0, the medium models provide them only banks 0 and 1, and the high-end models support general-purpose registers in all four banks.

#### Program counter

The 13-bit PC is shadowed by two other registers: PCL and PCLATH. Bits  $\text{PC}_{<7:0>}$  come from PCL and can be modified directly. Bits  $\text{PC}_{<12:8>}$  of the PC cannot be written but can be written using the PCLATH register. Writing to PCL will set the lower bits of the PC to the operand value and set the high bits of the PC from PCLATH.

#### PC stack

An 8-level stack is provided for the PC. This stack space is in a separate address space from either the program or data memory; the sp is not directly accessible. The stack is used by the subroutine CALL and RETURN/RETLW/RETFIE instructions. The stack actually operates as a circular buffer: when the stack overflows, the oldest value is overwritten.

#### STATUS register

STATUS is a special function register located in bank 0. It contains the status bits for the ALU, reset status, and bank select bits. A variety of instructions can affect bits in STATUS, including carry, digit carry, zero, register bank select, and indirect register bank select.

#### Addressing modes

PIC uses  $f$  to refer to one of the general-purpose registers in the register file. W refers to an accumulator that receives the ALU result; b refers to a bit address within a register; k refers to a literal, constant, or label.

Indirect addressing is controlled by the INDF and FSR registers. INDF is not a physical register. Any access to INDF causes an indirect load through the file select register FSR. FSR can be modified as a standard register. Reading from INDF uses the value of FSR as a pointer to the location to be accessed.

#### Data instructions

[Fig. 2.20](#) lists the data instructions in the PIC16F. Several different combinations of arguments are possible: ADDLW adds a literal  $k$  to the accumulator W; ADDWF adds W to the designated register  $f$ .

### 2.4.3 Flow of control

#### Jumps and branches

[Fig. 2.21](#) shows the PIC16F's flow of control instructions. GOTO is an unconditional branch.  $\text{PC}_{<10:0>}$  are set from the immediate value  $k$  in the instruction.  $\text{PC}_{<12:11>}$  come from PCLATH $<4:3>$ . Test-and-skip instructions such as INCFSZ take a register  $f$ . The  $f$  register is incremented and a skip occurs if the result is zero. This instruction also takes a one-bit  $d$  operand that determines where the incremented value of  $f$  is written: to W if  $d = 0$  and to  $f$  if  $d = 1$ . BTFSS is an example of a bit test-and-skip

ADDLW	Add literal and W
BCF	Bit clear f
ADDWF	Add W and f
BSF	Bit set f
ANDLW	AND literal with W
ANDWF	AND W with f
COMF	Complement f
CLRF	Clear f
DECFSNZ	Decrement f
CLRWF	Clear W
IORLW	Inclusive OR literal with W
INCF	Increment f
IORWF	Inclusive OR W with F
MOVF	Move f
MOVWF	Move W to f
MOVLW	Move literal to W
NOP	No operation
RLF	Rotate left F through carry
RRF	Rotate right F through carry
SUBWF	Subtract W from F
SWAPF	Swap nibbles in F
XORLW	Exclusive OR literal with W
CLRWDT	Clear watchdog timer
SUBLW	Subtract W from literal

**FIGURE 2.20**


---

Data instructions in the PIC16F.

instruction. It takes an f register argument and a three-bit b argument specifying the bit in f to be tested. This instruction also skips if the tested bit is 1.

#### Subroutines

Subroutines are called using the CALL instruction. The k operand provides the bottom 11 bits of the program counter, while the top two come from PCLATH<4:3>. The return address is pushed onto the stack. There are several variations of subroutine return, all of which use the top of the stack as the new PC value. RETURN performs a simple return; RETLW returns with an 8-bit literal k saved in the W register. RETFIE is used to return from interrupts, including enabling interrupts.

BTFS	Bit test f, skip if clear
BTFS	Bit test f, skip if set
CALL	Call subroutine
DECFSZ	Decrement f, skip if 0
INCFSZ	Increment f, skip if 0
GOTO	Unconditional branch
RETFIE	Return from interrupt
RETLW	Return with literal in W
RETURN	Return from subroutine
SLEEP	GO into standby mode

**FIGURE 2.21**

Flow of control instructions in the PIC16F.

## 2.5 TI C55x DSP

The TI C55x DSP is a family of DSPs that is designed for relatively high-performance signal processing. The family extends on previous generations of TI DSPs; the architecture is also defined to allow several different implementations that comply with the instruction set.

### Accumulator architecture

The C55x, like many DSPs, is an **accumulator architecture**, meaning that many arithmetic operations are of the form *accumulator* = *operand* + *accumulator*. Because one of the operands is the accumulator, it need not be specified in the instruction. Accumulator-oriented instructions are also well suited to the types of operations that are performed in digital signal processing, such as  $a_1x_1 + a_2x_2 + \dots$ . Of course, the C55x has more than one register and not all instructions adhere to the accumulator-oriented format. However, we will see that arithmetic and logical operations take a very different form in the C55x than they do in the Arm.

### Assembly language format

C55x assembly language programs follow the typical format

```
MPY *AR0, *CDP+, AC0
label: MOV #1, TO.
```

Assembler mnemonics are case insensitive. Instruction mnemonics are formed by combining a root with prefixes and/or suffixes. For example, the A prefix denotes an operation that is performed in addressing mode, while the 40 suffix denotes an arithmetic operation that is performed in 40-bit resolution. We will discuss the prefixes and suffixes in more detail when we describe the instructions.

The C55x also allows operations to be specified in an algebraic form:

```
AC1 = AR0 *coef(*CDP)
```

### 2.5.1 Processor and memory organization

We will use the term **register** to mean any type of register in the programmer model and the term **accumulator** to mean a register that is used primarily in the accumulator style.

#### Data types

The C55x supports several data types:

- A **word** is 16 bits long.
- A **longword** is 32 bits long.
- Instructions are byte addressable.
- Some instructions operate on addressed bits in registers.

#### Registers

The C55x has a number of registers. Few to none of these registers are general-purpose registers like those of the Arm. Registers are generally used for specialized purposes. Because the C55x registers are less regular, we will discuss them by how they may be used rather than simply listing them.

Most registers are **memory mapped**; that is, the register has an address in the memory space. A memory-mapped register can be referred to in assembly language in two different ways: either by referring to its mnemonic name or through its address.

The PC extension register XPC extends the range of the PC. The return address register RETA is used for subroutines.

#### Program counter and control flow

The C55x has four 40-bit accumulators: AC0, AC1, AC2, and AC3. The low-order bits 0–15 are referred to as AC0L, AC1L, AC2L, and AC3L; the high-order bits 16–31 are referred to as AC0H, AC1H, AC2H, and AC3H; and the guard bits 32–39 are referred to as AC0G, AC1G, AC2G, and AC3G. (Guard bits are used in numerical algorithms like signal processing to provide a larger dynamic range for intermediate calculations.)

#### Accumulators

The architecture provides six status registers. Three of the status registers, namely ST0 and ST1, and the processor mode status register PMST, are inherited from the C54x architecture. The C55x adds four registers: ST0\_55, ST1\_55, ST2\_55, and ST3\_55. These registers provide arithmetic and bit manipulation flags, a data page pointer and auxiliary register pointer, and processor mode bits, among other features.

#### Status registers

The stack pointer (SP) keeps track of the system stack. A separate system stack is maintained through the SSP register. The SPH register is an extended data page pointer for both the SP and SSP.

#### Stack pointers

Eight auxiliary registers AR0–AR7 are used by several types of instructions, notably for circular buffer operations. The coefficient data pointer (CDP) is used to read coefficients for polynomial evaluation instructions; CDPH is the main data page pointer for the CDP.

#### Auxiliary and coefficient data pointer registers

The circular buffer size register BK47 is used for circular buffer operations for the auxiliary registers AR4–7. Four registers define the start of circular buffers: BSA01 for auxiliary registers AR0 and AR1; BSA23 for AR2 and AR3; BSA45 for AR4 and AR5; and BSA67 for AR6 and AR7. The circular buffer size register BK03 is used to address circular buffers that are commonly used in signal processing. BKC is the

#### Circular buffers

**Single repeat registers**

circular buffer size register for CDP. BSAC is the circular buffer coefficient start address register.

**Block repeat registers**

Repeats of single instructions are controlled by the single repeat register CSR. This counter is the primary interface to the program. It is loaded with the required number of iterations. When the repeat starts, the value in CSR is copied into the repeat counter RPTC, which maintains the counts for the current repeat and is decremented during each iteration.

**Temporary registers**

Several registers are used for block repeats, which are instructions that are executed several times in a row. The block repeat counter BRC0 counts the block repeat iterations. The block repeat start and end registers RSA0L and REA0L keep track of the start and end points of the block.

The block repeat register 1 BRC1 and block repeat save register 1 BRS1 are used to repeat blocks of instructions. There are two repeat start address registers: RSA0 and RSA1. Each is divided into low and high parts: for example, RSA0L and RSA0H.

Four temporary registers, namely T0, T1, T2, and T3, are used for various calculations. These temporary registers are intended for miscellaneous use in code, such as holding a multiplicand for a multiply and holding shift counts.

Two transition register TRN0 and TRN1 are used for compare-and-extract-extremum instructions. These instructions are used to implement the Viterbi algorithm.

Several registers are used for addressing modes. The memory data page start address registers DP and DPH are used as the base address for data accesses. Similarly, the peripheral data page start address register PDP is used as a base for I/O addresses.

**Interrupts**

Several registers control interrupts. The interrupt mask registers 0 and 1, named IER0 and IER1, determine which interrupts will be recognized. The interrupt flag registers 0 and 1, named IFR0 and IFR1, keep track of currently pending interrupts. Two other registers, DBIER0 and DBIER1, are used for debugging. Two registers, the interrupt vector register DSP (and interrupt vector register host), are used as the base address for the interrupt vector table.

The C55x registers are summarized in [Fig. 2.22](#).

**Memory map**

The C55x supports a 24-bit address space, providing 16 MB of memory, as shown in [Fig. 2.23](#). Data, program, and I/O accesses are all mapped to the same physical memory, but these three spaces are addressed in different ways. The program space is byte addressable, so an instruction reference is 24 bits long. Data space is word addressable, so a data address is 23 bits long. (Its least significant bit is set to 0.) The data space is also divided into 128 pages of 64K words each. The I/O space is 64K words wide, so an I/O address is 16 bits. The situation is summarized in [Fig. 2.24](#).

Not all implementations of the C55x may provide all 16 MB of memory on chip. The C5510, for example, provides 352 KB of on-chip memory. The remainder of the memory space is provided by separate memory chips that are connected to the DSP.

The first 96 words of data page 0 are reserved for the memory-mapped registers. Because the program space is byte addressable, unlike the word-addressable data space, the first 192 words of the program space are reserved for those same registers.

<i>register mnemonic</i>	<i>description</i>
<i>AC0-AC3</i>	<i>accumulators</i>
<i>AR0-AR7, XAR0-XAR7</i>	<i>auxiliary registers and extensions of auxiliary registers</i>
<i>BK03, BK47, BKC</i>	<i>circular buffer size registers</i>
<i>BR<sub>C</sub>0-BR<sub>C</sub>1</i>	<i>block repeat counters</i>
<i>BRSI</i>	<i>BR<sub>C</sub>1 save register</i>
<i>CDP, CDPH, CDPX</i>	<i>coefficient data register: low (CDP), high (CDPH), full (CDPX)</i>
<i>CFCT</i>	<i>control flow context register</i>
<i>CSR</i>	<i>computed single repeat register</i>
<i>DBIER0-DBIER1</i>	<i>debug interrupt enable registers</i>
<i>DP, DPH, DPX</i>	<i>data page register: low (DP), high (DPH), full (DPX)</i>
<i>IER0-IER1</i>	<i>interrupt enable registers</i>
<i>IFR0-IFR1</i>	<i>interrupt flag registers</i>
<i>IVPD, IVPH</i>	<i>interrupt vector registers</i>
<i>PC, XPC</i>	<i>program counter and program counter extension</i>
<i>PDP</i>	<i>peripheral data page register</i>
<i>RETA</i>	<i>return address register</i>
<i>RPTC</i>	<i>single repeat counter</i>
<i>RSA0-RSA1</i>	<i>block repeat start address registers</i>

**FIGURE 2.22**

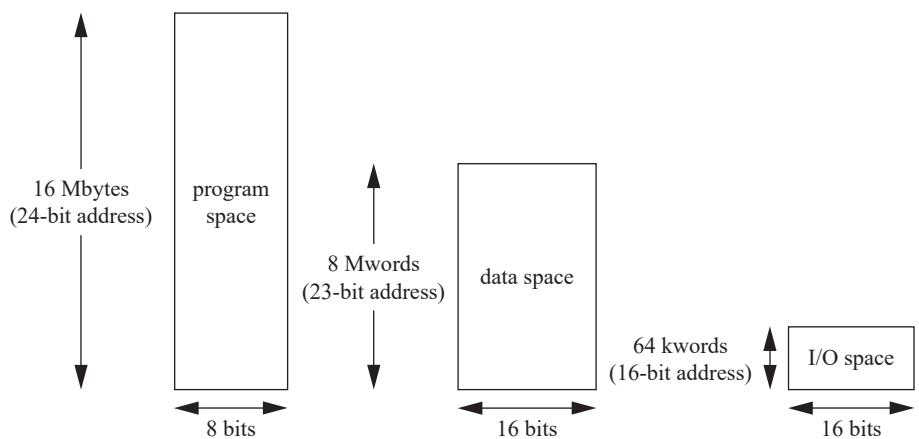
Registers in the TI C55x.

## 2.5.2 Addressing modes

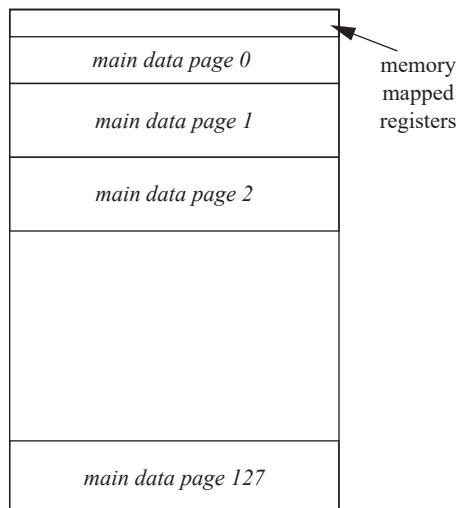
### Addressing modes

The C55x has three addressing modes:

- Absolute addressing supplies an address in the instruction.
- Direct addressing supplies an offset.
- Indirect addressing uses a register as a pointer.

**FIGURE 2.23**

Address spaces in the TMS320C55x.

**FIGURE 2.24**

The C55x memory map.

#### Absolute addressing

- Absolute addresses may be any of three different types:
- A k16 absolute address is a 16-bit value that is combined with the DPH register to form a 23-bit address.
- A k23 absolute address is a 23-bit unsigned number that provides a full data address.

- An I/O absolute address is of the form `port(#1234)`, where the argument to `port()` is a 16-bit unsigned value that provides the address in the I/O space.

#### Direct addressing

Direct addresses may be any of four different types:

- DP addressing is used to access data pages. The address is calculated as

$$A_{DP} = DPH[22 : 15] | (DP + Doffset).$$

`Doffset` is calculated by the assembler; its value depends on whether you are accessing a data page value or a memory-mapped register.

- SP addressing is used to access stack values in the data memory. The address is calculated as

$$A_{SP} = SPH[22 : 15] | (SP + Soffset).$$

`Soffset` is an offset that is supplied by the programmer.

- Register-bit direct addressing accesses bits in registers. The argument `@bitoffset` is an offset from the least significant bit of the register. Only a few instructions (register test, set, clear, and complement) support this mode.
- PDP addressing is used to access I/O pages. The 16-bit address is calculated as

$$A_{PDP} = PDP[15 : 6] | PDPoffset.$$

`PDPoffset` identifies the word within the I/O page. This addressing mode is specified with the `port()` qualifier.

#### Indirect addressing

Indirect addresses may be any of four different types:

- AR indirect addressing* uses an auxiliary register to point to data. This addressing mode is further subdivided into accesses to data, register bits, and I/O. To access a data page, the AR supplies the bottom 16 bits of the address, while the top 7 bits are supplied by the top bits of the XAR register. The AR supplies a bit number for register bits. (As with register-bit direct addressing, this only works on the register bit instructions.) When accessing the I/O space, the AR supplies a 16-bit I/O address. This mode may update the value of the AR register. Updates are specified by modifiers to the register identifier, such as adding `+` after the register name. Furthermore, the types of modifications that are allowed depend on the ARMS bit of the status register ST2\_55: 0 for DSP mode and 1 for control mode. A large number of such updates are possible: examples include `*ARn+`, which adds 1 to the register for a 16-bit operation and 2 to the register for a 32-bit operation; `*(ARn + ARO)` writes the value of `ARn + ARO` into `ARN`.
- Dual AR indirect addressing* allows two simultaneous data accesses, either for an instruction that requires two accesses or for executing two instructions in parallel. The register value may be updated depending on the modifiers to the register ID.
- CDP indirect addressing* uses the CDP register to access coefficients that may be in the data space, register bits, or I/O space. In the case of data space accesses, the

top 7 bits of the address come from CDPH and the bottom 16 come from the CDP. For register bits, the CDP provides a bit number. For I/O space accesses specified with `port()`, the CDP gives a 16 bit I/O address. The CDP register value may be updated depending on the modifiers to the register ID.

- *Coefficient indirect addressing* is similar to CDP indirect mode, but it is used primarily for instructions that require three memory operands per cycle.

Any of the indirect addressing modes may use circular addressing, which is useful for many DSP operations. Circular addressing is specified with the ARnLC bit in status register ST2\_55. For example, if bit AR0LC = 1, the main data page is supplied by AR0H, the buffer start register is BSA01, and the buffer size register is BK03.

#### Stack operations

The C55x supports two stacks: one for data and one for the system. Each stack is addressed by a 16-bit address. These two stacks can be relocated to different spots in the memory map by specifying a page using the high register: SP and SPH form XSP, the extended data stack; SSP and SPH form XSSP, the extended system stack. Note that both SP and SSP share the same page register SPH. XSP and XSSP hold 23-bit addresses that correspond to data locations.

The C55x supports three different stack configurations. These configurations depend on how the data and system stacks relate and how subroutine returns are implemented.

- In a dual 16-bit stack with fast return configuration, the data and system stacks are independent. A push or pop on the data stack does not affect the system stack. The RETA and CFCT registers are used to implement fast subroutine returns.
- In a dual 16-bit stack with slow return configuration, the data and system stacks are independent. However, RETA and CFCT are not used for slow subroutine returns; instead, the return address and loop context are stored on the stack.
- In a 32-bit stack with slow return configuration, SP and SSP are both modified by the same amount on any stack operation.

### 2.5.3 Data operations

#### Move instruction

The move (MOV) instruction moves data between registers and memory:

```
MOV src, dst
```

A number of variations of MOV are possible. The instruction can be used to move from memory into a register, from a register to memory, between registers, or from one memory location to another.

The ADD instruction adds a source and destination together and stores the result in the destination:

```
ADD src, dst
```

This instruction produces  $dst = dst + src$ . The destination may be an accumulator or another type. Variants allow constants to be added to the destination. Other variants

allow the source to be a memory location. The addition may also be performed on two accumulators, one of which has been shifted by a constant number of bits. Other variations are also defined.

A dual addition performs two adds in parallel:

```
ADD dual(Lmem), ACx, ACy
```

This instruction performs  $HI(ACy) = HI(Lmem) + HI(ACx)$  and  $LO(ACy) = LO(Lmem) + LO(ACx)$ . The operation is performed in 40-bit mode, but the lower 16 and upper 24 bits of the result are separated.

#### Multiply instructions

The multiply (MPY) instruction performs an integer multiplication:

```
MPY src,dst
```

Multiplications are performed on 16-bit values. Multiplication may be performed on accumulators, temporary registers, constants, or memory locations. The memory locations may be addressed either directly or using the coefficient addressing mode.

A multiply and accumulate is performed by the MAC instruction. It takes the same basic types of operands as does MPY. In the form

```
MAC ACx,Tx,ACy
```

the instruction performs  $ACy = ACy + (ACx * Tx)$ .

#### Compare instruction

The compare (CMP) instruction compares two values and sets a test control flag:

```
CMP Smem == val, TC1
```

The memory location is compared to a constant value. TC1 is set if the two are equal and cleared if they are not equal.

The CMP instruction can also be used to compare registers:

```
CMP src RELOP dst, TC1
```

The two registers can be compared using a variety of relational operators RELOP. If the U suffix is used on the instruction, the comparison is performed unsigned.

### 2.5.4 Flow of control

#### Branches

The B instruction is an unconditional branch. The branch target may be defined by the low 24 bits of an accumulator

```
B ACx
```

or by an address label

```
B label
```

The BCC instruction is a conditional branch:

```
BCC label, cond
```

The condition code determines the condition to be tested. Condition codes specify registers and the tests to be performed on them:

- Test the value of an accumulator:  $<0$ ,  $\leq 0$ ,  $>0$ ,  $\geq 0$ ,  $=0$ ,  $\neq 0$ .
- Test the value of the accumulator overflow status bit.
- Test the value of an auxiliary register:  $<0$ ,  $\leq 0$ ,  $>0$ ,  $\geq 0$ ,  $=0$ ,  $\neq 0$ .
- Test the carry status bit.
- Test the value of a temporary register:  $<0$ ,  $\leq 0$ ,  $>0$ ,  $\geq 0$ ,  $=0$ ,  $\neq 0$ .
- Test the control flags against 0 (condition prefixed by !) or against 1 (not prefixed by !) for combinations of AND, OR, and NOT.

### Loops

The C55x allows an instruction or a block of instructions to be repeated. Repeats provide efficient implementation of loops. Repeats may also be nested to provide two levels of repeats.

A single-instruction repeat is controlled by two registers. The single repeat counter, RPTC, counts the number of additional executions of the instruction to be executed; if RPTC = N, the instruction is executed a total of  $N + 1$  times. A repeat with a computed number of iterations may be performed using the computed single-repeat register CSR. The desired number of operations is computed and stored in CSR; the value of CSR is then copied into RPTC at the beginning of the repeat.

Block repeats perform a repeat on a block of contiguous instructions. A level 0 block repeat is controlled by three registers: the block repeat counter 0, BRC0, holds the number of times after the initial execution to repeat the instruction; the block repeat start address register 0, RSA0, holds the address of the first instruction in the repeat block; the repeat end address register 0, REA0, holds the address of the last instruction in the repeat block. (Note that, as with a single instruction repeat, if BRCn's value is N, the instruction or block is executed  $N + 1$  times.)

A level 1 block repeat uses BRC1, RSA1, and REA1. It also uses BRS1, the block repeat save register 1. Each time that the loop repeats, BRC1 is initialized with the value from BRS1. Before the block repeat starts, a load to BRC1 automatically copies the value to BRS1 to be sure that the right value is used for the inner loop executions.

A repeat cannot be applied to all instructions—some instructions cannot be repeated.

An unconditional subroutine call is performed by the `CALL` instruction:

`CALL target`

The target of the call may be a direct address or an address stored in an accumulator. Subroutines make use of the stack. A subroutine call stores two important registers: the return address and the loop context register. Both of these values are pushed onto the stack.

A conditional subroutine call is coded as

`CALLCC adrs,cond`

The address is a direct address; an accumulator value may not be used as the subroutine target. The conditional is as with other conditional instructions. As with the

### Nonrepeatable instructions

### Subroutines

unconditional CALL, CALLCC stores the return address and loop context register on the stack.

The C55x provides two types of subroutine returns: **fast return** and **slow return**. These vary on where they store the return address and loop context. In a slow return, the return address and loop context are stored on the stack. In a fast return, these two values are stored in registers: the return address register and control flow context register.

## Interrupts

Interrupts use the basic subroutine call mechanism. They are processed in four phases:

1. Receive the interrupt request.
2. Acknowledge the interrupt request.
3. Prepare for the interrupt service routine by finishing execution of the current instruction, storing registers, and retrieving the interrupt vector.
4. Process the interrupt service routine, which concludes with a return-from-interrupt instruction.

The C55x supports 32 interrupt vectors. Interrupts may be prioritized into 27 levels. The highest-priority interrupt is a hardware and software reset.

Most of the interrupts may be masked using the interrupt flag registers IFR1 and IFR2. Interrupt vectors 2–23, the bus error interrupt, the data log interrupt, and the real-time operating system interrupt can all be masked.

## 2.5.5 C coding guidelines

Some coding guidelines for the C55x [Tex01] not only provide more efficient code, but in some cases, should receive attention to ensure that the generated code is correct.

As with all digital signal processing code, the C55x benefits from careful attention to the required sizes of the variables. The C55x compiler uses some nonstandard lengths of data types: `char`, `short`, and `int` are all 16 bits, `long` is 32 bits, and `long long` is 40 bits. The C55x uses IEEE formats for `float` (32 bits) and `double` (64 bits). C code should not assume that `int` and `long` are the same types, that `char` is 8 bits long or that `long` is 64 bits. The `int` type should be used for fixed-point arithmetic, especially multiplications, and for loop counters.

The C55x compiler makes some important assumptions about operands of multiplications. This code generates a 32-bit result from the multiplication of two 16-bit operands:

```
long result = (long)(int)src1 * (long)(int)src2;
```

Although the operands were coerced to `long`, the compiler notes that each is 16 bits, so it uses a single-instruction multiplication.

The order of instructions in the compiled code depends in part on the C55x pipeline characteristics. The C compiler schedules code to minimize code conflicts and to

take advantage of parallelism wherever possible. However, if the compiler cannot determine that a set of instructions is independent, it must assume that the instructions are dependent and generate more restrictive, slower code. The `restrict` keyword can be used to tell the compiler that a given pointer is the only one in the scope that can point to a particular object. The `-pm` option allows the compiler to perform more global analysis and find more independent sets of instructions.

Example 2.7 shows a C implementation of an FIR filter on the C55x.

## Example 2.7 FIR Filter on the C55x

Here is assembly code generated by the TI C55x C compiler for the FIR filter with manually generated comments:

---

## 2.6 TI C64x

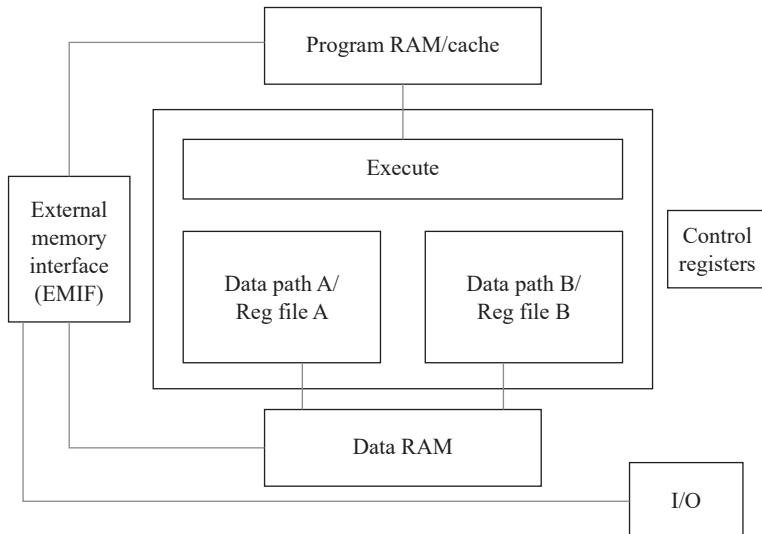
The TI TMS320C64x is a high-performance VLIW DSP. It provides both fixed-point and floating-point arithmetic. The CPU can execute up to eight instructions per cycle using eight general-purpose 32-bit registers and eight functional units.

[Fig. 2.25](#) shows a simplified block diagram of the C64x. Although instruction execution is controlled by a single execution unit, the instructions are performed by two data paths, each with its own register file. The CPU is a load/store architecture. The data paths are named A and B. Each data path provides four function units:

- The .L units (.L1 and .L2 in the two data paths) perform 32/40-bit arithmetic and comparison, 32-bit logical operations, and data packing/unpacking.
- The .S units (.S1 and .S2) perform 32-bit arithmetic, 32/40-bit shifts and bit-field operations, 32-bit logical operations, branches, and other operations.
- The .M units (.M1 and .M2) perform multiplications, bit interleaving, rotation, Galois field multiplication, and other operations.
- The .D units (.D1 and .D2) perform address calculations, loads and stores, and other operations.

Separate data paths perform data movement:

- Load from memory units .LD1 and .LD2: these units load registers from memory.
- Store from memory units .ST1 and .ST2 store register values to memory.
- Address paths .DA1 and .DA2 compute addresses. These units are associated with the .D1 and .D2 units in the data paths.



**FIGURE 2.25**

C64x block diagram.

- Register file **cross paths**.1X and .2X move data between register files A and B. Data must be explicitly moved from one register file to the other before it can be used in the other data path.

On-chip memory is organized as separate data and program memories. The external memory interface (EMIF) manages connections to external memory. The external memory is generally organized as a unified memory space.

The C64x provides a variety of 40-bit operations. A 40-bit value is stored in a register pair, with the least significant bits in an even-numbered register and the remaining bits in an odd-numbered register. A similar scheme is used for 64-bit values.

Instructions are fetched in groups known as **fetch packets**. A fetch packet includes eight words at a time and is aligned on 256-bit boundaries. Owing to the small size of some instructions, a fetch packet may include up to 14 instructions. The instructions in a fetch packet may be executed in varying combinations of sequential and parallel execution. An **execute packet** is a set of instructions that execute together. Up to eight instructions may execute together in a fetch packet, but all of them must use a different functional unit, either performing different operations on a data path or using corresponding function units in different data paths. The **p-bit** in each instruction encodes information about which instructions can be executed in parallel. Instructions may execute fully serially, fully parallelly, or partially serially.

Many instructions can be conditionally executed, as specified by an **s** that specifies the condition register and a **z field** that specifies a test for zero or nonzero.

Two instructions in the same execute packet cannot use the same resources or write to the same register in the same cycle. Here are some examples of these constraints:

- Instructions must use separate functional units. For example, two instructions cannot simultaneously use the .S1 unit.
- Most combinations of writing using the same .M unit are prohibited.
- Most cases of reading multiple values in the opposite register file using the .1X and .2X cross path units are prohibited.
- A delay cycle is executed when an instruction attempts to read a register that was updated in the previous cycle by a cross path operation.
- The .DA1 and .DA2 units cannot execute in one execute packet to load and store registers using a destination or source from the same register file. The address register must be in the same data path as the .D unit being used.
- At most four reads of the same register can occur in the same cycle.
- Two instructions in an execute packet cannot write to the same register in the same cycle.
- A variety of other constraints limit the combinations of instructions that are allowed in an execute packet.

The C64x provides **delay slots**, which were first introduced in RISC processors. Some effects of an instruction may take additional cycles to complete. The delay slot is a set of instructions following the given instruction; the delayed results of

the given instruction are not available in the delay slot. An instruction that does not need the result can be scheduled within the delay slot. Any instruction that requires the new value must be placed after the end of the delay slot. For example, a branch instruction has a delay slot of five cycles.

The C64x provides three addressing modes: linear, circular using BK0, and circular using BK1. The addressing mode is determined by the AMR addressing mode register. A linear address shifts the offset by 3, 2, 1, or 0 bits depending on the length of the operand and then adds the base register to determine the physical address. The circular addressing modes using BK0 and BK1 use the same shift and base calculation, but only modify bits 0 through N of the address.

The C64x provides atomic operations that can be used to implement semaphores and other mechanisms for concurrent communication. The LL (load linked) instruction reads a location and sets a link valid flag to true. The link valid flag is cleared when another process stores to that address. The SL (store linked) instruction prepares a word to be committed to memory by storing it in a buffer, but does not commit the change. The CMTL (commit linked stored) instruction checks the link valid flag and writes the SL-buffered data if the flag is true.

The processing of interrupts is mediated by registers. The interrupt flag register IFR is set when an interrupt occurs; the  $i^{\text{th}}$  bit of the IFR corresponds to the  $i^{\text{th}}$  level of interrupt. Interrupts are enabled and disabled using the interrupt enable register IER. Manual interrupts are controlled using the interrupt set register ISR and interrupt clear register ICR. The interrupt return pointer register IRP contains the return address for the interrupt. The interrupt service table pointer register ISTP points to a table of interrupt handlers. The processor supports nonmaskable interrupts.

The C64x+ is an enhanced version of the C64x. It supports a number of exceptions. The exception flag register (EFR) indicates which exceptions have been thrown. The exception clear register ECR can be used to clear bits in the EFR. The internal exception report register IERR indicates the cause of an internal exception. The C64x+ provides two modes of program execution: user and supervisor. Several registers, notably those related to interrupts and exceptions, are not available in user mode. A supervisor mode program can enter user mode using the B NRP instruction. A user mode program may enter supervisor mode by using an SWE or SWENR software interrupt.

---

## 2.7 Summary

When viewed from high above, all CPUs are similar: they read and write memory, perform data operations, and make decisions. However, there are many ways to design an instruction set, as illustrated by the differences among the Arm, C55x, C64x, and PIC16F. When designing complex systems, we generally view the programs in high-level language form, which hides many of the details of the instruction set. However, differences in instruction sets can be reflected in the nonfunctional characteristics, such as the program size and speed.

## What we learned

- Both the von Neumann and Harvard architectures are in common use today.
  - The programming model is a description of the architecture that is relevant to instruction operation.
  - Arm is a load—store architecture. It provides a few relatively complex instructions, such as saving and restoring multiple registers.
  - The PIC16F is a very small, efficient microcontroller.
  - The C55x provides a number of architectural features to support the arithmetic loops that are common in digital signal processing code.
  - The C64x organizes instructions into execution packets to enable parallel execution.
- 

## Further reading

The books by Jaggar [Jag95], Furber [Fur96], and Sloss et al. [Slo04] describe the Arm architecture. The Arm website, <http://www.arm.com>, contains a large number of documents describing various versions of Arm. Information on the PIC16F can be found at [www.microchip.com](http://www.microchip.com). Information on the C55x and C64x can be found at <http://www.ti.com>.

---

## Questions

- Q2-1** What is the difference between a big-endian and little-endian data representation?
- Q2-2** What is the difference between the Harvard and von Neumann architectures?
- Q2-3** Answer the following questions about the Arm programming model:
- How many general-purpose registers are there?
  - What is the purpose of the CPSR?
  - What is the purpose of the Z bit?
  - Where is the program counter kept?
- Q2-4** How would the Arm status word be set after these operations?
- $1 - 2$
  - $-232 + 1 - 1$
  - $-4 + 5$
  - $1 + 2$
- Q2-5** What is the meaning of these Arm condition codes?
- EQ
  - NE
  - MI
  - CS

- e. VS
- f. GE
- g. CC
- h. LT

**Q2-6** Explain the operation of the BL instruction, including the state of Arm registers before and after its operation.

**Q2-7** How do you return from an Arm procedure?

**Q2-8** In the following code, show the contents of the Arm function call stack just after each C function has been entered and just after the function exits. Assume that the function call stack is empty when main() begins.

```
int foo(int x1, int x2) {
    return x1 + x2;
}
int baz(int x1) {
    return x1 + 1;
}
int scum(int r) {
    for (i = 0; i = 2; i++)
        foo(r + i, 5);
}
main() {
    scum(3);
    baz(2);
}
```

**Q2-9** Why are specialized instruction sets such as Neon or Jazelle useful?

**Q2-10** Is the PIC16F a general-purpose register machine?

**Q2-11** How large is the PC stack in the PIC16F?

**Q2-12** Which two registers contribute to the PC value?

**Q2-13** What data types does the C55x support?

**Q2-14** How many accumulators does the C55x have?

**Q2-15** What C55x register holds arithmetic and bit manipulation flags?

**Q2-16** What is a block repeat in the C55x?

**Q2-17** How are the C55x data and program memory arranged in the physical memory?

**Q2-18** Where are C55x memory-mapped registers located in the address space?

**Q2-19** What is the AR register used for in the C55x?

- Q2-20** What is the difference between the DP and PDP addressing modes in the C55x?
- Q2-21** How many stacks are supported by the C55x architecture and how are their locations in memory determined?
- Q2-22** Which register controls single-instruction repeats in the C55x?
- Q2-23** What is the difference between slow and fast returns in the C55x?
- Q2-24** How many functional units does the C64x have?
- Q2-25** What is the difference between a fetch packet and an execute packet in the C64x?

---

### Lab exercises

- L2-1** Write a program that uses a circular buffer to perform FIR filtering.
- L2-2** Write a simple loop that lets you exercise the cache. By changing the number of statements in the loop body, you can vary the cache hit rate of the loop as it executes. You should be able to observe changes in the speed of execution by observing the microprocessor bus.
- L2-3** Compare the implementations of an FIR filter on two different processors. How do they compare in code size and performance?

## CPUs

## 3

**CHAPTER POINTS**

- Input and output mechanisms.
- Supervisor mode, exceptions, and traps.
- Memory management and address translation.
- Interrupts, supervisor mode, exceptions, and traps.
- Caches.
- Performance and power consumption of CPUs.
- Design example: Data compressor.

---

### 3.1 Introduction

This chapter describes aspects of CPUs that do not directly relate to their instruction sets. We consider mechanisms that are important to interfacing with other system elements, such as interrupts and memory management. We also take a first look at aspects of the CPU other than functionality; performance and power consumption are both vital attributes of programs that are only indirectly related to the instructions they use.

In Section 3.2, we study input and output mechanisms, including both busy/wait and interrupts. Section 3.3 introduces several specialized mechanisms for operations, such as detecting internal errors and protecting CPU resources. Section 3.4 introduces coprocessors that provide optional support for parts of the instruction set. Section 3.5 describes the CPU's view of memory, both memory management and caches. The next sections look at the nonfunctional attributes of execution: Section 3.6 looks at performance, and Section 3.7 considers power consumption. Section 3.8 looks at several issues related to safety and security. Finally, in Section 3.9, we use a data compressor as an example of a simple, yet interesting program.

## 3.2 Programming input and output

The basic techniques for I/O programming can be understood as relatively independent of the instruction set. In this section, we cover the basics of I/O programming and place them in the contexts of the Arm, C55x, and PIC16F. We begin by discussing the basic characteristics of I/O devices so that we can understand the requirements they place on programs that communicate with them.

### 3.2.1 I/O devices

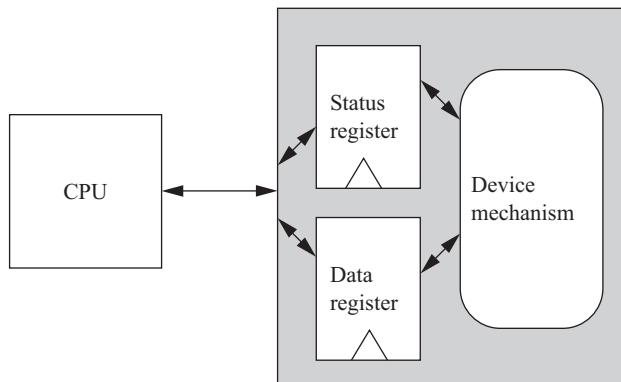
Input and output devices usually have some analog or nonelectronic components. However, the digital logic in the device that is most closely connected to the CPU very strongly resembles the logic you would expect in any computer system.

Fig. 3.1 shows the structure of a typical I/O device and its relationship to the CPU. The interface between the CPU and the device is a set of registers. The CPU talks to the device by reading and writing to/from the registers. A device typically has several registers:

- **Data registers** hold values that are treated as data by the device, such as data that are read or written by a disk.
- **Status registers** provide information about the device's operation, such as whether the current transaction has been completed.

Some registers may be read-only, such as a status register that indicates when the device is done, whereas others may be readable or writable.

Application Example 3.1 describes a classic I/O device.



**FIGURE 3.1**

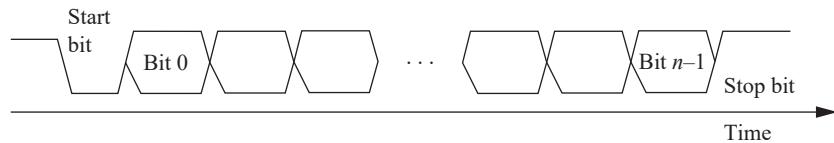
Structure of a typical I/O device.

---

### Application Example 3.1 The 8251 Universal Asynchronous Receiver/Transmitter

The 8251 Universal Asynchronous Receiver/Transmitter (**UART**) [Int82] is the original device used for serial communications, such as serial port connections on PCs. The 8251 was introduced as a stand-alone integrated circuit for early microprocessors. Today, its functions are typically subsumed by a larger chip, but these more advanced devices still use the basic programming interface defined by the 8251.

The UART is programmable for a variety of transmission and reception parameters. However, the basic format of transmission is simple. Data are transmitted as streams of characters in this form:



Every character starts with a start bit (a 0 value) and a stop bit (a 1 value). The start bit allows the receiver to recognize the start of a new character, and the stop bit ensures that there will be a transition at the start of the stop bit. The data bits are sent as high and low voltages at a uniform rate. That rate is known as the **baud rate**; the period of one bit is the inverse of the baud rate.

Before transmitting or receiving data, the CPU must set the UART's mode register to correspond to the data line's characteristics. The parameters for the serial port are familiar from the parameters for a serial communications program:

- mode[1:0]: mode and baud rate
  - 00: synchronous mode
  - 01: asynchronous mode, no clock prescaler
  - 10: asynchronous mode, 16× prescaler
  - 11: asynchronous mode, 64× prescaler
- mode[3:2]: number of bits per character
  - 00: 5 bits
  - 01: 6 bits
  - 10: 7 bits
  - 11: 8 bits
- mode[5:4]: parity
  - 00, 10: no parity
  - 01: odd parity
  - 11: even parity
- mode[7:6]: stop bit length
  - 00: invalid
  - 01: 1 stop bit
  - 10: 1.5 stop bits
  - 11: 2 stop bits

Setting bits in the command register tells the UART what to do:

- mode[0]: transmit enable
- mode[1]: set nDTR output
- mode[2]: enable receiver
- mode[3]: send break character
- mode[4]: reset error flags

- mode[5]: set nRTS output
- mode[6]: internal reset
- mode[7]: hunt mode

The status register shows the state of the UART and transmission:

- status [0]: transmitter ready
- status [1]: receive ready
- status [2]: transmission complete
- status [3]: parity
- status [4]: overrun
- status [5]: frame error
- status [6]: sync char detected
- status [7]: nDSR value

The UART includes transmit and receive buffer registers. It also includes registers for synchronous mode characters.

The *Transmitter Ready* output indicates that the transmitter is ready to accept a data character; the *Transmitter Empty* signal goes high when the UART has no characters to send. On the receiver side, the *Receiver Ready* pin goes high when the UART has a character ready to be read by the CPU.

---

### 3.2.2 Input and output primitives

Microprocessors can provide programming support for input and output in two ways: **I/O instructions** and **memory-mapped I/O**. Some architectures, such as the Intel x86, provide special instructions (*in* and *out* in the case of the Intel x86) for the input and output. These instructions provide a separate address space for I/O devices.

However, the most common way to implement I/O is through memory mapping—even CPUs that provide I/O instructions can also implement memory-mapped I/O. As the name implies, the memory-mapped I/O provides addresses for the registers in each I/O device. Programs use the CPU's normal read and write instructions to communicate with devices.

Example 3.1 illustrates the memory-mapped I/O on Arm.

---

#### Example 3.1 Memory-mapped I/O on Arm

We can use the *EQU* pseudo-op to define a symbolic name for the memory location of our I/O device:

```
DEV1    EQU 0x1000
```

Given that name, we can use the following standard code to read and write the device register:

```
LDR r1,#DEV1      ; set up device address
LDR r0,[r1]        ; read DEV1
LDR r0,#8          ; set up value to write
STR r0,[r1]        ; write 8 to device
```

---

How can we directly write I/O devices in a high-level language like C? When we define and use a variable in C, the compiler hides the variable's address from us. However, we can use pointers to manipulate the addresses of I/O devices. The traditional names for functions that read and write arbitrary memory locations are **peek** and **poke**. The peek function can be written in C as follows:

```
int peek(char *location) {
    return *location; /* de-reference location pointer */
}
```

The argument to peek is a pointer that is dereferenced by the C language `*` operator to read the location. Thus, to read a device register, we can write

```
#define DEV1 0x1000
...
dev_status = peek(DEV1); /* read device register */
```

The poke function can be implemented as

```
void poke(char *location, char newval) {
    (*location) = newval; /* write to location */
}
```

To write to the device's register, we can use the following code:

```
poke(DEV1,8); /* write 8 to device register */
```

These functions can, of course, be used to read and write arbitrary memory locations, not just devices.

### 3.2.3 Busy-wait I/O

The simplest way to communicate with devices in a program is **busy-wait I/O**. Devices are typically slower than the CPU and may require many cycles to complete an operation. If the CPU performs multiple operations on a single device, such as writing several characters to an output device, then it must wait for one operation to complete before starting the next one. If we try to start writing the second character before the device has finished with the first one, for example, the device will probably never print the first character. Asking an I/O device whether it is finished by reading its status register is often called **polling**.

Example 3.2 illustrates busy-wait I/O.

#### Example 3.2 Busy-wait I/O Programming

In this example, we want to write a sequence of characters to an output device. The device has two registers: one for the character to be written, and one for a status register. The status register's value is 1 when the device is busy writing and 0 when the write transaction has been completed.

We use the peek and poke functions to write the busy-wait routine in C. First, we define symbolic names for the register addresses:

```
#define OUT_CHAR 0x1000 /* output device character register */
#define OUT_STATUS 0x1001 /* output device status register */
```

The sequence of characters is stored in a standard C string, which is terminated by a null (0) character. We use peek and poke to send the characters and wait for each transaction to complete:

```
char *mystring = "Hello, world." /* string to write */
char * current_char; /* pointer to current position in string */
current_char = mystring; /* point to head of string */
while (*current_char != '\0') { /* until null character */
    poke(OUT_CHAR, *current_char); /* send character to device */
    while (peek(OUT_STATUS) != 0); /* keep checking status */
    current_char++; /* update character pointer */
}
```

The outer while loop sends the characters one at a time. The inner while loop checks the device status, which implements the busy-wait function by repeatedly checking the device status until the status changes to 0.

---

Example 3.3 illustrates a combination of input and output.

### Example 3.3 Copying Characters from Input to Output Using Busy-wait I/O

We want to read a character repeatedly from the input device and write it to the output device. First, we need to define the addresses for the device registers:

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
```

The input device sets its status register to 1 when a new character has been read; we must set the status register back to 0 after the character has been read so that the device is ready to read another character. When writing, we must set the output status register to 1 to start writing and wait for it to return to 0. We can use peek and poke to repeatedly perform the read/write operation:

```
while (TRUE) { /* perform operation forever */
    /*read a character into achar */
    while (peek(IN_STATUS) == 0); /* wait until ready */
    achar = (char)peek(IN_DATA); /* read the character */
    /* write achar */
    poke(OUT_DATA, achar);
    poke(OUT_STATUS,1); /* turn on device */
    while (peek(OUT_STATUS) != 0); /* wait until done */
}
```

---

### 3.2.4 Interrupts

#### Basics

Busy-wait I/O is extremely inefficient—the CPU does nothing but test the device status while the I/O transaction is in progress. In many cases, the CPU could do useful work in parallel with the I/O transaction:

- computation, as in determining the next output to send to the device or processing the last input received, and
- Control of other I/O devices.

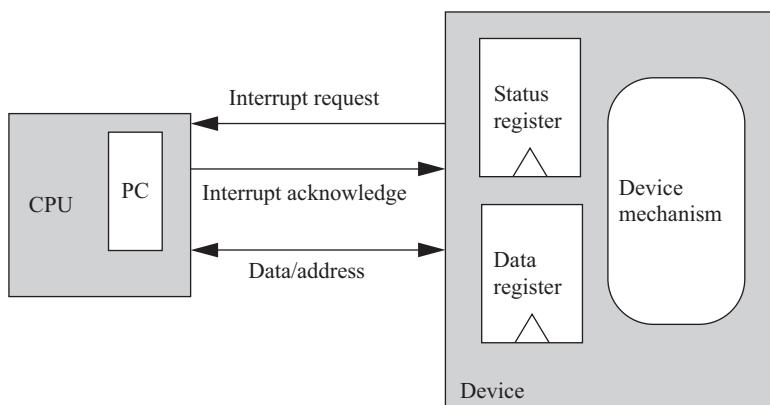
To allow parallelism, we must introduce new mechanisms into the CPU.

The **interrupt** mechanism allows devices to signal the CPU and to force the execution of a particular piece of code. When an interrupt occurs, the program counter's value is changed to point to an **interrupt handler** routine, also commonly known as a **device driver**, which takes care of the device: writing the next data, reading data that have just become ready, and so on. The interrupt mechanism, of course, saves the value of the PC at the interruption, so that the CPU can return to the program that was interrupted. Interrupts therefore allow the flow of control in the CPU to change easily between different **contexts**, such as a foreground computation and multiple I/O devices.

As shown in Fig. 3.2, the interface between the CPU and I/O device includes several signals that control the interrupt process:

- The I/O device asserts the **interrupt request** signal when it wants service from the CPU.
- The CPU asserts the **interrupt acknowledge** signal when it is ready to handle the I/O device's request.

The I/O device's logic decides when to interrupt. For example, it may generate an interrupt when its status register goes into the ready state. The CPU may not be able to



**FIGURE 3.2**

The interrupt mechanism.

immediately service an interrupt request, because it may be doing something else that must be finished first. For example, a program that talks to both a high-speed disk drive and low-speed keyboard should be designed to finish a disk transaction before handling a keyboard interrupt. Only when the CPU decides to acknowledge the interrupt does the CPU change the program counter to point to the device's handler. The interrupt handler operates much like a subroutine, except that it is not called by the executing program. The program that runs when no interrupt is being handled is often called the **foreground program**; when the interrupt handler finishes running in the background, it returns to the foreground program wherever processing was interrupted.

Before considering the details of how interrupts are implemented, let's look at the interrupt style of processing and compare it to busy-wait I/O. Example 3.4 uses interrupts as a basic replacement for busy-wait I/O.

---

### Example 3.4 Copying Characters from Input to Output with Basic Interrupts

As with Example 3.3, we repeatedly read a character from an input device and write it to an output device. We assume that we can write C functions that act as interrupt handlers. Those handlers will work with the devices in much the same way as in busy-wait I/O by reading and writing status and data registers. The main difference is in handling the output; the interrupt signals that the character is done, so the handler doesn't have to do anything.

We use a global variable, achar for the input handler to pass the character to the foreground program. Because the foreground program doesn't know when an interrupt occurs, we also use a global Boolean variable, gotchar to signal when a new character has been received. Here is the code for the input and output handlers:

```
void input_handler() { /* get a character and put in global */
    achar = peek(IN_DATA); /* get character */
    gotchar = TRUE; /* signal to main program */
    poke(IN_STATUS,0); /* reset status to initiate next transfer */
}
void output_handler() { /* react to character being sent */
    /* don't have to do anything */
}
```

The main program is reminiscent of the busy-wait program. It looks at gotchar to check when a new character has been read, and then, immediately sends it out to the output device.

```
main() {
    while (TRUE) { /* read then write forever */
        if (gotchar) { /* write a character */
            poke(OUT_DATA, achar); /* put character in device */
            poke(OUT_STATUS,1); /* set status to initiate write */
            gotchar = FALSE; /* reset flag */
        }
    }
}
```

---

The use of interrupts has made the main program somewhat simpler. However, this program design still does not allow the foreground program to do useful work. Example 3.5 uses a more sophisticated program design to let the foreground program work completely independently of I/O.

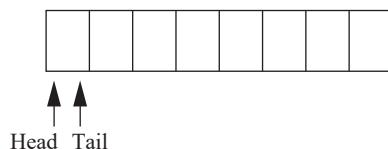
---

### Example 3.5 Copying Characters from Input to Output with Interrupts and Buffers

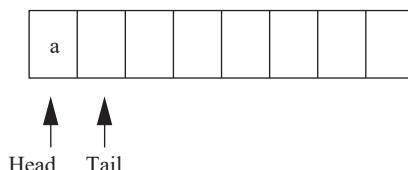
Because we don't need to wait for each character, we can make this I/O program more sophisticated than the one in Example 3.4. Rather than reading a single character and then writing it, the program performs reading and writing independently. We use an elastic buffer to hold the characters. The read and write routines communicate through the global variables used to implement the elastic buffer:

- A character string, `io_buf`, holds a queue of characters that have been read, but not yet written.
- A pair of integers, `buf_start` and `buf_end`, will point to the first and last characters read, respectively.
- An integer, `error`, will be set to zero whenever `io_buf` overflows.

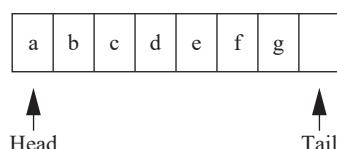
The elastic buffer allows the I/O devices to run at different rates. The queue, `io_buf`, acts as a wraparound buffer; we add characters to the tail when an input is received and take characters from the tail when we are ready for output. The head and tail wrap around the end of the buffer array to make the most efficient use of the array. Here is the situation at the start of the program's execution, where the tail points to the first available character and the head points to the ready character. As seen below, because the head and tail are equal, we know that the queue is empty.



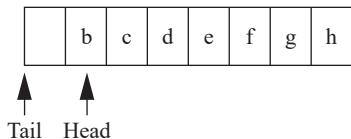
When the first character is read, the tail is incremented after the character is added to the queue, leaving the buffer and pointers to look like this:



When the buffer is full, we leave one character in the buffer unused. As the next figure shows, if we were to add another character and update the tail buffer (wrapping it around to the head of the buffer), we would be unable to distinguish a full buffer from an empty one.



Here is what happens when the output goes past the end of `io_buf`:



This code implements the elastic buffer, including declarations for the above global variables and some service routines for adding and removing characters from the queue. Because interrupt handlers are regular code, we can use subroutines to structure code, as with any program.

```
#define BUF_SIZE 8
char io_buf[BUF_SIZE]; /* character buffer */
int buf_head = 0, buf_tail = 0; /* current position in buffer */
int error = 0; /* set to 1 if buffer ever overflows */

void empty_buffer() { /* returns TRUE if buffer is empty */
    (buf_head == buf_tail) ? TRUE : FALSE;
}

void full_buffer() { /* returns TRUE if buffer is full */
    ((buf_tail+1) % BUF_SIZE == buf_head) ? TRUE : FALSE;
}

int nchars() { /* returns the number of characters in the buffer */
    if (buf_tail >= buf_head) return buf_tail - buf_head;
    else return BUF_SIZE - buf_head + buf_tail;
}

void add_char(char achar) { /* add a character to the buffer head */
    io_buf[buf_tail++] = achar;
    /* check pointer */
    if (buf_tail == BUF_SIZE)
        buf_tail = 0;
}

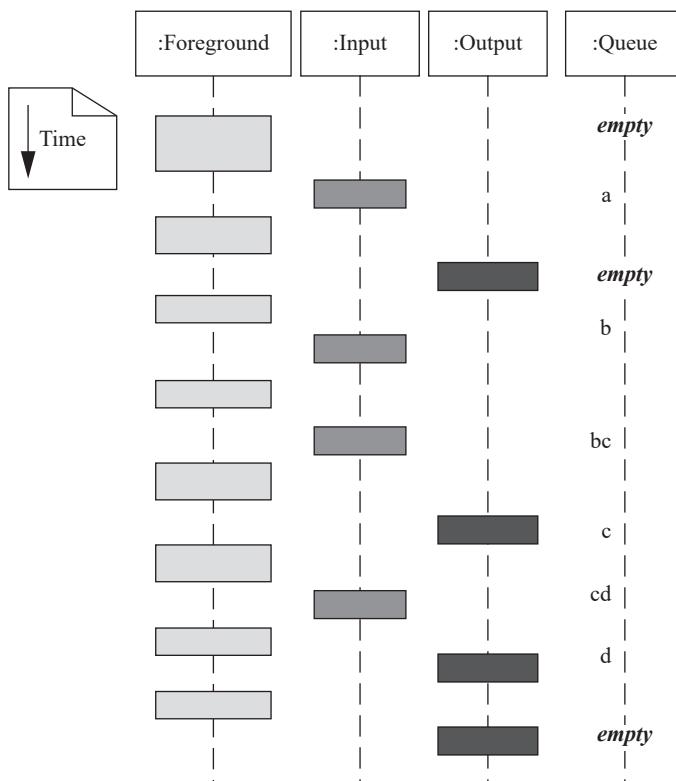
char remove_char() { /* take a character from the buffer head */
    char achar;
    achar = io_buf[buf_head++];
    /* check pointer */
    if (buf_head == BUF_SIZE)
        buf_head = 0;
    return achar;
}
```

Assume that we have two interrupt handling routines defined in C: `input_handler` for the input device and `output_handler` for the output device. These routines work with the device in much the same way as the busy-wait routines. The only complication is in starting the output device: If `io_buf` has characters waiting, the output driver can start a new output transaction by itself. However, if there are no characters waiting, an outside agent must start a new output action whenever the new character arrives. Rather than force the foreground program to look at the character buffer, we will have the input handler check to see whether there is only one character in the buffer and start a new transaction.

Here is the code for the input handler:

```
#define IN_DATA 0x1000
#define IN_STATUS 0x1001
void input_handler() {
    char achar;
    if(full_buffer()) /* error */
        error = 1;
    else { /* read the character and update pointer */
        achar = peek(IN_DATA); /* read character */
        add_char(achar); /* add to queue */
    }
    poke(IN_STATUS,0); /* set status register back to 0 */
    /* if buffer was empty, start a new output transaction */
    if(nchars()==1) { /*buffer had been empty until this interrupt */
        poke(OUT_DATA,remove_char()); /* send character */
        poke(OUT_STATUS,1); /* turn device on */
    }
}
#define OUT_DATA 0x1100
#define OUT_STATUS 0x1101
void output_handler() {
    if(!empty_buffer()) { /* start a new character */
        poke(OUT_DATA,remove_char()); /* send character */
        poke(OUT_STATUS,1); /* turn device on */
    }
}
```

The foreground program does not need to do anything; everything is taken care of by the interrupt handlers. The foreground program is free to do useful work, as it is occasionally interrupted by input and output operations. The following sample execution of the program in the form of a UML sequence diagram shows how input and output are interleaved with the foreground program. We have kept the last input character in the queue until the output is complete to make it clearer when input occurs. The simulation shows that the foreground program is not executing continuously, but it continues to run in its regular state, independent of the number of characters waiting in the queue.



Interrupts allow a great deal of concurrency, which can make proficient use of the CPU. However, when interrupt handlers are buggy, the errors can be complex to find. The fact that an interrupt can occur at any time means that the same bug can manifest itself in different ways when the interrupt handler interrupts different segments of the foreground program.

Example 3.6 illustrates the problems inherent in debugging interrupt handlers.

---

### Example 3.6 Debugging Interrupt Code

Assume that the foreground code performs a matrix multiplication operation,  $y = Ax + b$ :

```

for (i = 0; i < M; i++) {
    y[i] = b[i];
    for (j = 0; j < N; j++)
        y[i] = y[i] + A[i][j] * x[j];
}

```

We use the interrupt handlers of Example 3.6 to perform I/O while the matrix computation is performed, but with one small change: `read_handler` has a bug that causes it to change the value of `j`. Although this may seem far-fetched, remember that when the interrupt handler is written in assembly language, such bugs are easy to introduce. Any CPU register written by the interrupt handler must be saved before it is modified and restored before the handler exits. Any type of bug, such as forgetting to save the register or to properly restore it, can cause that register to mysteriously change value in the foreground program.

What happens to the foreground program when `j` changes the value during an interrupt depends on when the interrupt handler executes. Because the value of `j` is reset at each iteration of the outer loop, the bug will affect only one entry of the result, `y`. Clearly, the entry that changes will depend on when the interrupt occurs. Furthermore, the change observed in `y` depends not only on what new value is assigned to `j`, which may depend on the data handled by the interrupt code, but also when in the inner loop the interrupt occurs. An interrupt at the beginning of the inner loop will give a different result than one that occurs near the end. The number of possible new values for the result vector is much too large to consider manually, and the bug can't be found by enumerating the possible wrong values and correlating them with a given root cause. Even recognizing the error can be difficult. For example, an interrupt that occurs at the very end of the inner loop will not cause any change in the foreground program's results. Finding such bugs generally requires a great deal of tedious experimentation and frustration.

---

The CPU implements interrupts by checking the interrupt request line at the beginning of the execution of every instruction. If an interrupt request has been asserted, the CPU does not fetch the instructions pointed to by the PC. Instead, the CPU sets the PC to a predefined location, which is the beginning of the interrupt-handling routine. The starting address of the interrupt handler is usually given as a pointer. Rather than defining a fixed location for the handler, the CPU defines a location in the memory that holds the address of the handler, which can then reside anywhere in memory.

Because the CPU checks for interrupts at every instruction, it can respond quickly to service requests from devices. However, the interrupt handler must return to the foreground program without disturbing the foreground program's operation. Because subroutines perform a similar function, it is natural to build the CPU's interrupt mechanism to resemble its subroutine function. Most CPUs use the same basic mechanism for remembering the foreground program's PC as is used for subroutines. The subroutine call mechanism in modern microprocessors is typically a stack, so the interrupt mechanism puts the return address on the stack. Some CPUs use the same stack as for subroutines, whereas others define a special stack. The use of a procedure-like interface also makes it easier to provide a high-level language interface for interrupt handlers. The details of the C interface to interrupt handling routines vary with both the CPU and the underlying support software.

### **Priorities and vectors**

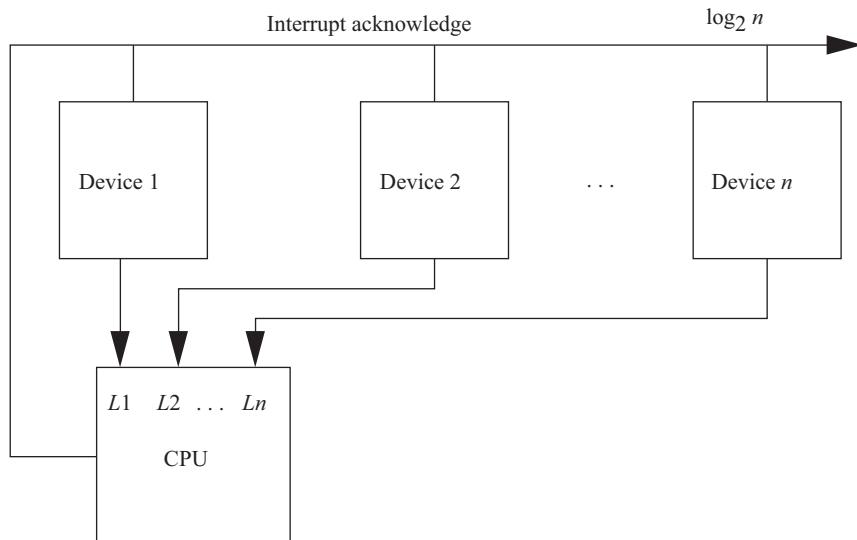
Providing a practical interrupt system requires more than a simple interrupt request line. Most systems have more than one I/O device, so there must be some mechanism for allowing multiple devices to interrupt. We also want to have flexibility in the locations of the interrupt-handling routines, the addresses for devices, and so on. There

are two ways in which interrupts can be generalized to handle multiple devices and to provide more flexible definitions for the associated hardware and software:

- **Interrupt priorities** allow the CPU to recognize some interrupts as more important than others.
- **Interrupt vectors** allow the interrupting device to specify its handler.

Prioritized interrupts not only allow multiple devices to be connected to the interrupt line, but they also allow the CPU to ignore less important interrupt requests while handling more important requests. As shown in Fig. 3.3, the CPU provides several different interrupt request signals, shown here as  $L_1, L_2, \dots$  up to  $L_n$ . Typically, the lower-numbered interrupt lines are given higher priority, so in this case, if devices 1, 2, and  $n$  all requested interrupts simultaneously, 1's request would be acknowledged because it is connected to the highest-priority interrupt line. Rather than provide a separate interrupt acknowledge line for each device, most CPUs use a set of signals that provide the priority number of the winning interrupt in binary form, so that interrupt level 7 requires three bits rather than seven. A device knows that its interrupt request was accepted by seeing its own priority number on the interrupt acknowledge lines.

How do we change the priority of a device? Simply connect it to a different interrupt request line. This requires hardware modification. Hence, if priorities need to be changeable, removable cards, programmable switches, or some other mechanism should be provided to make the change easy.



**FIGURE 3.3**

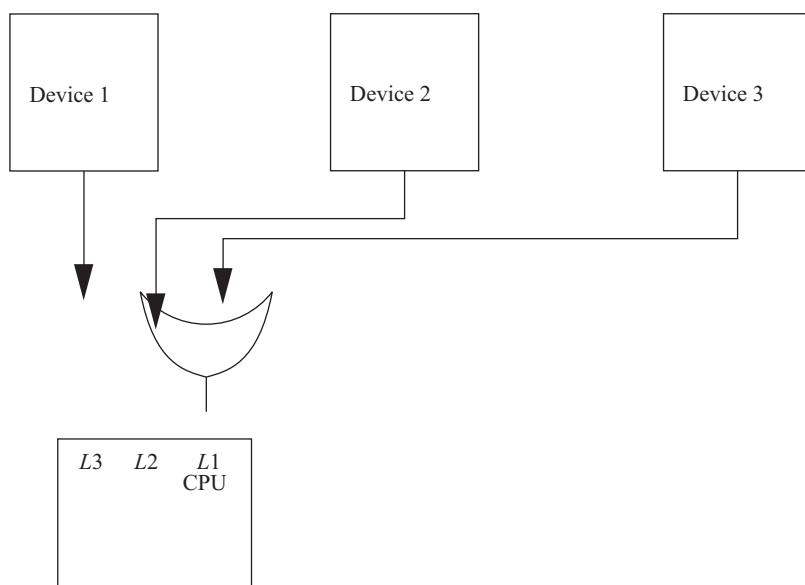
Prioritized device interrupts.

The priority mechanism must ensure that a lower-priority interrupt does not occur when a higher-priority interrupt is handled. The decision process is known as **masking**. When an interrupt is acknowledged, the CPU stores the priority level of that interrupt in an internal register. When a subsequent interrupt is received, its priority is checked against the priority register; the new request is acknowledged only if it has higher priority than the currently pending interrupt. When the interrupt handler exits, the priority register must be reset. The need to reset the priority register is one reason why most architectures introduce a specialized instruction to return from interrupts, rather than using the standard subroutine return instruction.

#### Power-down interrupts

The highest-priority interrupt is normally called the **nonmaskable interrupt (NMI)**. The NMI cannot be turned off and is usually reserved for interruptions caused by power failures. A simple circuit can be used to detect a dangerously low power supply, and the NMI interrupt handler can be used to save critical state in nonvolatile memory, turn off I/O devices to eliminate spurious device operation during power-down, and so on.

Most CPUs provide a relatively small number of interrupt priority levels, such as eight. Although more priority levels can be added with external logic, they may not be necessary in all cases. When several devices naturally assume the same priority, such as when you have several identical keypads attached to a single CPU, you can combine polling with prioritized interrupts to efficiently handle the devices. As shown in Fig. 3.4, you can use a small amount of logic external to the CPU to generate an interrupt whenever any of the devices you want to group together request service.



**FIGURE 3.4**

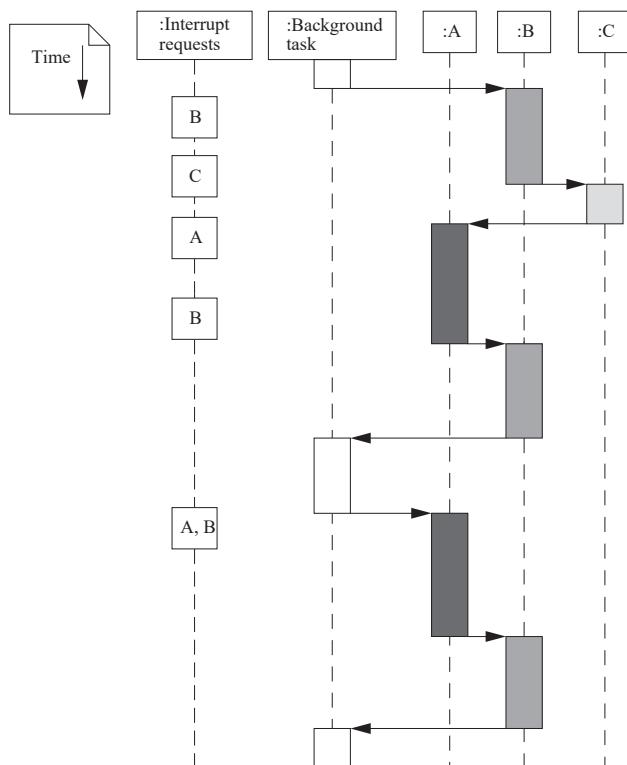
Using polling to share an interrupt over several devices.

The CPU will call the interrupt handler associated with this priority; the handler does not know which of the devices actually requested the interrupt. The handler uses software polling to check the status of each device. In this example, it will read the status registers of 1, 2, and 3 to see which of them is ready and requesting service.

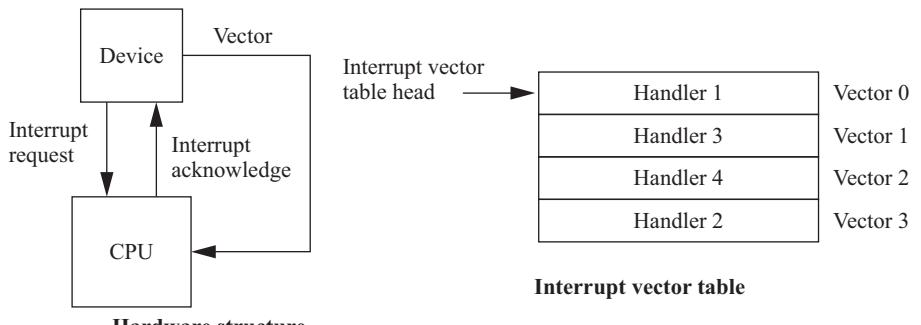
Example 3.7 illustrates how priorities affect the order in which I/O requests are handled.

### Example 3.7 I/O with Prioritized Interrupts

Assume that we have devices A, B, and C. A has priority 1 (highest priority), B has priority 2, and C has priority 3. This UML sequence diagram shows which interrupt handler executes as a function of time for a sequence of interrupt requests.



In each case, an interrupt handler keeps running until either it finishes, or a higher-priority interrupt arrives. The C interrupt handler is delayed by the handling required for the first B interrupt. The second B interrupt handler is similarly delayed by service for the A interrupt. When both A and B interrupt simultaneously, A's interrupt gets priority; when A's handler is finished, the priority mechanism automatically answers B's pending interrupt.

**FIGURE 3.5**

Interrupt vectors.

Vectors provide flexibility in a different dimension: the ability to define the interrupt handler that should service a request from a device. Fig. 3.5 shows the hardware structure required to support interrupt vectors. In addition to the interrupt request and acknowledge lines, additional interrupt vector lines run from the devices to the CPU. After a device's request is acknowledged, it sends its interrupt vector over those lines to the CPU. The CPU then uses the vector number as an index in a table stored in memory, as shown in Fig. 3.5. The location referenced in the interrupt vector table by the vector number gives the address of the handler.

There are two important things to notice about the interrupt vector mechanism. First, the device, not the CPU, stores its vector number. In this way, a device can be given a new handler simply by changing the vector number it sends without modifying the system software. For example, vector numbers can be changed by programmable switches. The second thing to notice is that there is no fixed relationship between vector numbers and interrupt handlers. The interrupt vector table allows arbitrary relationships between devices and handlers. The vector mechanism provides great flexibility in the coupling of hardware devices and the software routines that service them.

Most modern CPUs implement both prioritized and vectored interrupts. Priorities determine which device is serviced first, and vectors determine what routine is used to service the interrupt. The combination of the two provides a rich interface between hardware and software.

### ***Interrupt overhead***

Now that we have a basic understanding of the interrupt mechanism, we can consider the complete interrupt-handling process. When a device requests an interrupt, some steps are performed by the CPU, some by the device, and others by software:

1. **CPU:** The CPU checks for pending interrupts at the beginning of an instruction. It answers the highest-priority interrupt, which has a higher priority than that given in the interrupt priority register.
2. **Device:** The device receives the acknowledgment and sends the CPU its interrupt vector.

3. *CPU*: The CPU looks up the device handler address in the interrupt vector table using the vector as an index. A subroutine-like mechanism is used to save the current value of the PC and possibly other internal CPU states, such as general-purpose registers.
4. *Software*: The device driver may save an additional CPU state. It then performs the required operations on the device. It then restores any saved state and executes the interrupt return instruction.
5. *CPU*: The interrupt return instruction restores the PC and other automatically saved states to return execution to the interrupted code.

Interrupts do not come without a performance penalty. In addition to the execution time required for the code that talks directly to the devices, there is an execution time overhead associated with the interrupt mechanism:

- The interrupt itself has overhead comparable to a subroutine call. Because an interrupt causes a change in the program counter, it incurs a branch penalty. In addition, if the interrupt automatically stores CPU registers that action requires extra cycles, even if the state is not modified by the interrupt handler.
- In addition to the branch delay penalty, the interrupt requires extra cycles to acknowledge the interrupt and obtain the vector from the device.
- In general, the interrupt handler will save and restore CPU registers that were not automatically saved by the interrupt.
- The interrupt return instruction incurs a branch penalty, as well as the time required to restore the automatically saved state.

The time required for the hardware to respond to the interrupt, obtain the vector, and so on cannot be changed by the programmer. CPUs vary quite a bit amounting to internal state automatically saved by an interrupt. The programmer does have control over what state is modified by the interrupt handler, and it must be saved and restored. Careful programming can sometimes result in a small number of registers used by an interrupt handler, thereby saving time in maintaining the CPU state. However, such tricks usually require coding the interrupt handler in assembly language rather than a high-level language.

### ***Interrupts in Arm***

Arm7 supports two types of interrupts: fast interrupt requests (FIQs) and interrupt requests (IRQs). An FIQ takes priority over an IRQ. The interrupt table is always kept in the bottom memory addresses, starting at location zero. The entries in the table typically contain subroutine calls to the appropriate handler.

The Arm7 performs the following steps when responding to an interrupt [ARM99B]:

- saves the appropriate value of the PC to be used to return,
- copies the current program status register (CPSR) into a saved program status register (SPSR),

- forces bits in the CPSR to note the interrupt, and
- forces the PC to the appropriate interrupt vector.

When leaving the interrupt handler, the handler should:

- restore the proper PC value,
- restore the CPSR from the SPSR, and
- clear the interrupt-disable flags.

The worst-case latency to respond to an interrupt includes the following components:

- two cycles to synchronize the external request,
- up to 20 cycles to complete the current instruction,
- three cycles for data abort, and
- two cycles to enter the interrupt-handling state.

This adds up to 27 clock cycles. The best-case latency is four clock cycles.

A vectored interrupt controller (VIC) can be used to provide up to 32 vectored interrupts [ARM02]. The VIC is assigned a block of memory for its registers; the VIC base address should be in the upper 4K of memory to avoid increasing the access times of the controller's registers. An array, `VICVECTADDR`, in this block specifies the interrupt service routines' addresses; the array, `VICVECTPRIORITY`, gives the priorities of the interrupt sources.

### ***Interrupts in C55x***

Interrupts in the C55x [Tex04] take at least seven clock cycles. In many situations, they take 13 clock cycles.

- A maskable interrupt is processed in several steps when the interrupt request is sent to the CPU.
- The interrupt flag register (IFR) corresponding to the interrupt is set.
- The interrupt enable register (IER) is checked to ensure that the interrupt is enabled.
- The interrupt mask register (INTM) is checked to ensure that the interrupt is not masked.
- The IFR corresponding to the flag is cleared.
- Appropriate registers are saved as context.
- INTM is set to one to disable maskable interrupts.
- The disable bug mask bit (DGBM) is set to one to disable debug events.
- EALLOW is set to zero to disable access to nonCPU emulation registers.
- A branch is performed on the interrupt service routine.

The C55x provides two mechanisms—**fast-return** and **slow-return**—to save and restore registers for interrupts and other context switches. Both processes save the return address and loop context registers. The fast-return mode uses RETA to save the return address and CFCT for the loop context bits. In contrast, the slow-return mode saves the return address and loop context bits on the stack.

### ***Interrupts in PIC16F***

The PIC16F recognizes two types of interrupts. Synchronous interrupts generally occur from sources inside the CPU. Asynchronous interrupts are generally triggered from outside the CPU. The INTCON register contains the major control bits for the interrupt system. The Global Interrupt Enable (GIE) bit is used to allow all unmasked interrupts. The Peripheral Interrupt Enable bit (PEIE) enables/disables interrupts from peripherals. The TMR0 Overflow Interrupt Enable bit enables or disables the timer zero overflow interrupt. The INT External Interrupt Enable bit enables/disables the INT external interrupts. Peripheral Interrupt Flag registers PIR1 and PIR2 hold flags for peripheral interrupts.

The RETFIE instruction is used to return from an interrupt routine. This instruction clears the GIE bit, reenabling pending interrupts.

The latency of synchronous interrupts is  $3T_{CY}$ , where  $T_{CY}$  is the length of an instruction. The latency for asynchronous interrupts is 3 to  $3.75T_{CY}$ . One- and two-cycle instructions have the same interrupt latencies.

## **3.3 Supervisor mode, exceptions, and traps**

In this section, we consider exceptions and traps. These are mechanisms to handle internal conditions, and they are analogous to interrupts in form. We begin with a discussion of supervisor mode, which some processors use to handle exceptional events and protect executing programs from each other.

### **3.3.1 Supervisor mode**

As will become clearer in later chapters, complex systems are often implemented as several programs that communicate with each other. These programs may run under the command of an operating system. Thus, it may be desirable to provide hardware checks to ensure that the programs do not interfere with each other, for example, by erroneously writing into a segment of the memory used by another program. Software debugging is important, but it can leave some problems in a running system; hardware checks ensure an additional level of safety.

In such cases, it is often useful to have a **supervisor mode** provided by the CPU. Normal programs run in **user mode**. The supervisor mode has privileges that the user modes do not. For example, we will study memory management systems in Section 3.5 that allow the physical addresses of memory locations to be changed dynamically. Control of the memory management unit is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers, effectively moving code and data in the middle of program execution.

#### **Arm supervisor mode**

Not all CPUs have supervisor modes. Many DSPs, including the C55x, do not provide one. Arm, however, has a supervisor mode. The Arm instruction that puts the CPU in supervisor mode is called SWI:

SWI CODE\_1

It can, of course, be executed conditionally, as with any Arm instruction. SWI causes the CPU to go into supervisor mode and sets the PC to 0x08. The argument to SWI is a 24-bit immediate value that is passed on to the supervisor mode code; it allows the program to request various services from the supervisor mode.

In supervisor mode, the bottom five bits of the CPSR are all set to 1 to indicate that the CPU is in supervisor mode. The old value of the CPSR just before the SWI, stored in a register, is the SPSR. There are, in fact, several SPSRs for different modes; the supervisor mode SPSR is referred to as `SPSR_SVC`.

To return from supervisor mode, the supervisor restores the PC from register r14 and restores the CPSR from `SPSR_SVC`.

### 3.3.2 Exceptions

An **exception** is an internally detected error. A simple example is a division by zero. One way to handle this problem is to check every divisor before division to be sure it is not zero, but this would substantially increase the size of numerical programs and cost a great deal of CPU time evaluating the divisor's value. The CPU can more efficiently check the divisor's value during execution. Because the time at which a zero divisor will be found is not known in advance, this event is like an interrupt, except that it is generated inside the CPU. The exception mechanism provides a way for the program to react to such unexpected events. Resets, undefined instructions, and illegal memory accesses are other typical examples of exceptions.

Just as interrupts can be seen as an extension of the subroutine mechanism, exceptions are generally implemented as a variation of an interrupt. Because both deal with changes in the flow of control of a program, it makes sense to use similar mechanisms. However, exceptions are generated internally.

Exceptions generally require both prioritization and vectoring. Exceptions must be prioritized because a single operation may generate more than one exception, such as an illegal operand and an illegal memory access. The priority of exceptions is usually fixed by the CPU architecture. Vectoring provides a way for the user to specify the handler for the exception condition. The vector number for an exception is usually predefined by the architecture; it is used to index into a table of exception handlers.

### 3.3.3 Traps

A **trap**, also known as a **software interrupt**, is an instruction that explicitly generates an exception condition. The most common use of a trap is to enter supervisor mode. The entry into supervisor mode must be controlled to maintain security; if the interface between the user and supervisor mode is improperly designed, a user program may be able to sneak code into the supervisor mode that could be executed to perform harmful operations.

Arm provides the SWI interrupt for software interrupts. This instruction causes the CPU to enter supervisor mode. An opcode is embedded in the instruction that can be read by the handler.

---

### 3.4 Coprocessors

CPU architects often want to provide flexibility in terms of what features are implemented in the CPU. One way to provide such flexibility at the instruction set level is to allow **coprocessors**, which are attached to the CPU, and implement some of the instructions. For example, floating-point arithmetic was introduced into the Intel architecture by providing separate chips that implemented the floating-point instructions.

To support coprocessors, certain opcodes must be reserved in the instruction set for coprocessor operations. Because it executes instructions, a coprocessor must be tightly coupled to the CPU. When the CPU receives a coprocessor instruction, the CPU must activate the coprocessor and pass it the relevant instruction. Coprocessor instructions can load and store coprocessor registers, or perform internal operations. The CPU can suspend execution to wait for the coprocessor instruction to finish; it can also take a more superscalar approach and continue executing instructions while waiting for the coprocessor to finish.

A CPU may, of course, receive coprocessor instructions, even when there is no coprocessor attached. Most architectures use illegal instruction traps to handle these situations. The trap handler can detect the coprocessor instruction and, for example, execute it in software on the main CPU. Emulating coprocessor instructions in software is slower but provides compatibility.

#### Coprocessors in Arm

The Arm architecture provides support for up to 16 coprocessors attached to a CPU. Coprocessors can perform load and store operations on their own registers. They can also move data between the coprocessor registers and the main ARM registers.

An example of an Arm coprocessor is the floating-point unit. The unit occupies two coprocessor units in the Arm architecture, numbered 1 and 2, but it appears as a single unit to the programmer. It provides eight 80-bit floating-point data registers, floating-point status registers, and an optional floating-point status register.

---

### 3.5 Memory system mechanisms

Modern microprocessors do more than just read and write monolithic memories. Architectural features improve both the speed and capacity of memory systems. Microprocessor clock rates are increasing at a faster rate than memory speeds, such that memories fall farther behind microprocessors every day. As a result, computer architects resort to **caches** to increase the average performance of memory systems. Although memory capacity is increasing steadily, program sizes are also increasing, and designers may not be willing to pay for all the memory demanded by an application.

**Memory management units (MMUs)** perform address translations that provide a larger virtual memory space in a small physical memory. **Memory protection units (MPUs)** provide memory protection mechanisms. In this section, we review both types of caches: MMUs, and MPUs.

### 3.5.1 Caches

Caches are widely used to speed up reads and writes in memory systems. Many microprocessor architectures include caches as part of their definitions. The cache speeds up the average memory access time when properly used. This increases the variability of memory access times. Accesses in the cache will be fast, but access to locations not cached will be slow. This variability in performance makes it especially important to understand how caches work so that we can better understand how to predict cache performance and factor these variations into system design.

#### Cache controllers

A cache is a small, fast memory that holds copies of some of the contents of the main memory. Because the cache is fast, it provides higher-speed access for the CPU, but because it is small, not all requests can be satisfied by the cache, forcing the system to wait for the slower main memory. Caching makes sense when the CPU uses only a relatively small set of memory locations at a time; the set of active locations is often called the **working set**.

Fig. 3.6 shows how the cache supports reads in the memory system. A **cache controller** mediates between the CPU and the memory system, which comprises the cache and main memory. The cache controller sends a memory request to the cache and the main memory. If the requested location is in the cache, the cache controller forwards the location's contents to the CPU and aborts the main memory request. This condition is a **cache hit**. If the location is not in the cache, the controller waits for the value from the main memory and forwards it to the CPU; this situation is a **cache miss**.

We can classify cache misses into several types, depending on the situation that generated them:

- A **compulsory miss** (i.e., **cold miss**) occurs the first time a location is used.

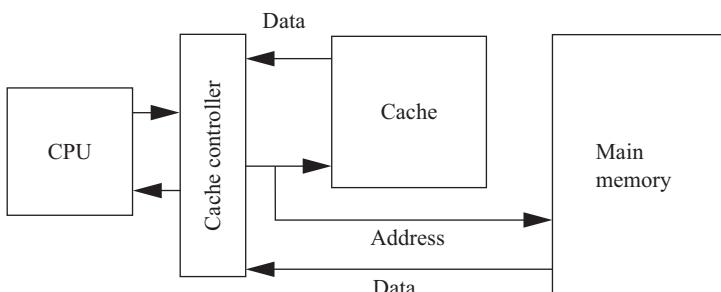


FIGURE 3.6

The cache in the memory system.

- A **capacity miss** is caused by a too-large working set.
- A **conflict miss** happens when two locations map to the same location in the cache.

### Memory system performance

Even before we consider ways to implement caches, we can write some basic formulas for memory system performance. Let  $h$  be the **hit rate**, the probability that a given memory location is in the cache. It follows that  $1-h$  is the **miss rate** or the probability that the location is not in the cache. Then, we can compute the average memory access time as

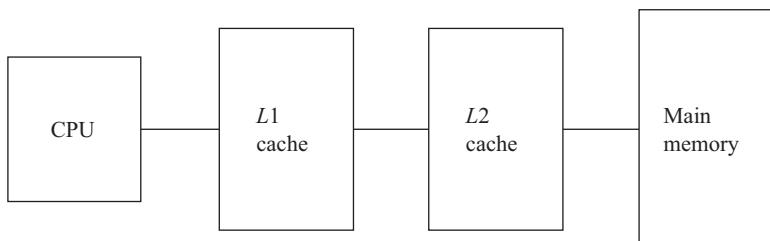
$$t_{av} = ht_{cache} + (1 - h)t_{main}, \quad (\text{Eq. 3.1})$$

where  $t_{cache}$  is the access time of the cache, and  $t_{main}$  is the main memory access time. The memory access times are basic parameters provided by the memory manufacturer. The hit rate depends on the program being executed and the cache organization, and is typically measured using simulators. The best-case memory access time, ignoring cache controller overhead, is  $t_{cache}$ , whereas the worst-case access time is  $t_{main}$ . Given that  $t_{main}$  is typically 50 to 75 ns, and  $t_{cache}$  is at most a few nanoseconds, the spread between worst- and best-case memory delays is substantial.

### Cache organization

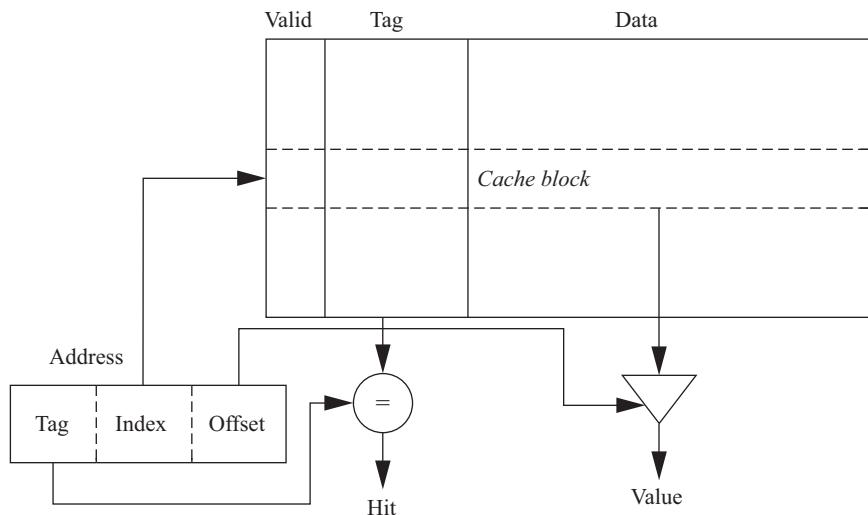
The simplest way to implement a cache is a **direct-mapped cache**, as shown in Fig. 3.8. The cache consists of cache **blocks**, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid. An address is divided into three sections. The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index. If the address tag matches the tag value in the block, that block includes the desired memory location. If the length of the data field is longer than the minimum addressable unit, the lowest bits of the address will be used as an offset to select the required value from the data field. Given the structure of the cache, only one block must be checked to see whether a location is in the cache—the index uniquely determines that block. If the access is a hit, the data value is read from the cache.

Writes are slightly more complicated than reads because we must update the main memory as well as the cache. There are several methods by which we can do this. The



**FIGURE 3.7**

A two-level cache system.

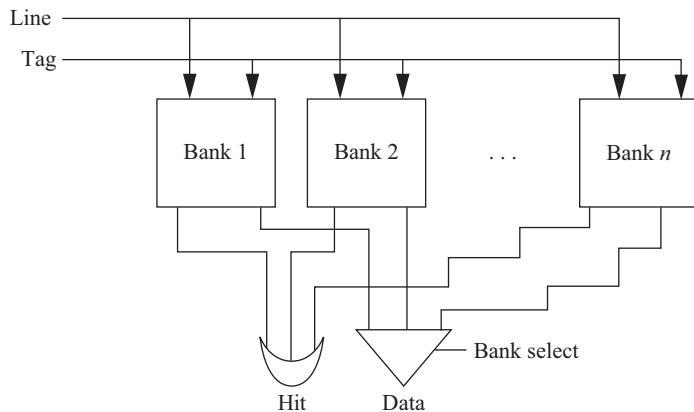
**FIGURE 3.8**

A direct-mapped cache.

simplest scheme is known as a **write-through**: every write changes both the cache and corresponding main memory location, usually through a write buffer. This scheme ensures that the cache and main memory are consistent but may generate additional main memory traffic. We can reduce the number of times we write to the main memory by using a **write-back** policy. If we write only when we remove a location from the cache, we eliminate the write when a location is written several times before it is removed from the cache.

The direct-mapped cache is both fast and relatively low cost, but it does have limits in its caching power to its simple scheme for mapping the cache onto the main memory. Consider a direct-mapped cache with four blocks in which locations 0, 1, 2, and 3 all map to different blocks. However, locations 4, 8, 12,... all map to the same block as location 0; locations 1, 5, 9, 13,... all map to a single block; and so on. If two popular locations in a program happen to map onto the same block, we will not gain the full benefits of the cache. As seen in Section 5.7, this can create program performance problems.

The limitations of the direct-mapped cache can be reduced by using the **set-associative** cache structure that is shown in Fig. 3.9. A set-associative cache is characterized by the number of **banks** or **ways** used, giving an  $n$ -way set-associative cache. A set is formed by all the blocks (one for each bank) that share the same index. Each set is implemented with a direct-mapped cache. A cache request is broadcast to all banks simultaneously. If any of the sets has the location, the cache reports a hit. Although memory locations map onto blocks using the same function, there are  $n$  separate blocks for each set of locations. Therefore, we can simultaneously cache several locations that

**FIGURE 3.9**

A set-associative cache.

happen to map onto the same cache block. The set-associative cache structure incurs a little extra overhead and is slightly slower than a direct-mapped cache, but the higher hit rates that it can provide often compensate.

The set-associative cache generally provides higher hit rates than the direct-mapped cache because conflicts between a small set of locations can be resolved within the cache. The set-associative cache is somewhat slower, so the CPU designer must be careful that it doesn't slow down the CPU's cycle time too much. A more important problem with set-associative caches for embedded program design is predictability. Because the time penalty for a cache miss is so severe, we often want to make sure that critical segments of our programs exhibit good behavior in the cache. It is relatively easy to determine when two memory locations conflict in a direct-mapped cache. Conflicts in a set-associative cache are more subtle; thus, the behavior of a set-associative cache is more difficult to analyze for both humans and programs.

Example 3.8 compares the behavior of direct-mapped and set-associative caches.

---

### Example 3.8 Direct-mapped vs. Set-associative Caches

For simplicity, let's consider a very simple caching scheme. We use two bits of the address as the tag. We compare a direct-mapped cache with four blocks and a two-way set-associative cache with four sets, and we use least-recently used (LRU) replacement to make it easy to compare the two caches.

Here are the contents of memory using a 3-bit address for simplicity.

Address	Data
000	0101
001	1111
010	0000
011	0110
100	1000
101	0001
110	1010
111	0100

We will give each cache the same pattern of addresses in binary to simplify picking out the index: 001, 010, 011, 100, 101, and 111. To understand how the direct-mapped cache works, let's see how its state evolves.

After 001 access:

Block	Tag	Data
00	—	—
01	0	1111
10	—	—
11	—	—

After 010 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	—	—

After 011 access:

Block	Tag	Data
00	—	—
01	0	1111
10	0	0000
11	0	0110

After 100 access (notice that the tag bit for this entry is 1):

Block	Tag	Data
00	1	1000
01	0	1111
10	0	0000
11	0	0110

After 101 access (overwrites the 01 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	0	0110

After 111 access (overwrites the 11 block entry):

Block	Tag	Data
00	1	1000
01	1	0001
10	0	0000
11	1	0100

We can use a similar procedure to determine what ends up in the two-way set-associative cache. The only difference is that we have some freedom when we have to replace a block with new data. To make the results easy to understand, we use a least-recently-used

replacement policy. For starters, let's make each bank the size of the original direct-mapped cache. The final state of the two-way set-associative cache follows:

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
00	1	1000	—	—
01	0	1111	1	0001
10	0	0000	—	—
11	0	0110	1	0100

Of course, this isn't a fair comparison for performance because the two-way set-associative cache has twice as many entries as the direct-mapped cache. Let's use a two-way, set-associative cache with two sets, giving us four blocks, the same number as in the direct-mapped cache. In this case, the index size is reduced to 1 bit, and the tag grows to 2 bits.

Block	Bank 0 tag	Bank 0 data	Bank 1 tag	Bank 1 data
0	01	0000	10	1000
1	10	0001	11	0100

In this case, the cache contents significantly differ from either the direct-mapped cache or the four-block, two-way set-associative cache.

The CPU knows when it is fetching an instruction (the PC is used to calculate the address, either directly or indirectly) or the data. We can therefore choose whether to cache instructions, data, or both. If cache space is limited, instructions are the highest priority for caching because they usually provide the highest hit rates. A cache that holds both instructions and data is called a **unified cache**.

Modern CPUs may use multiple levels of cache, as shown in Fig. 3.7. The **first-level cache** (commonly known as the **L1 cache**) is closest to the CPU, the **second-level cache (L2 cache)** feeds the first-level cache, and so on. In today's microprocessors, the first-level cache is often on-chip, and the second-level cache is off-chip, although we are starting to see on-chip second-level caches.

The second-level cache is much larger, but is also slower. If  $h_1$  is the first-level hit rate and  $h_2$  is the rate at which access hits the second-level cache, then the average access time for a two-level cache system is

$$t_{av} = h_1 t_{L1} + h_2 t_{L2} + (1 - h_1 - h_2)t_{main} \quad (\text{Eq. 3.2})$$

As the program's working set changes, we expect locations to be removed from the cache to make way for new locations. When set-associative caches are used, we must think about what happens when we throw out a value from the cache to make room for a new value. We do not have this problem in direct-mapped caches because every location maps onto a unique block. However, in a set-associative cache, we must decide

### First- and second-level cache

which set will have its block thrown out to make way for the new block. One possible replacement policy is **least recently used (LRU)**, that is, throw out the block that has been used farthest in the past. We can add relatively small amounts of hardware to the cache to keep track of the time since the last access for each block. Another policy is random replacement, which requires even less hardware to implement.

### 3.5.2 Memory management units and address translation

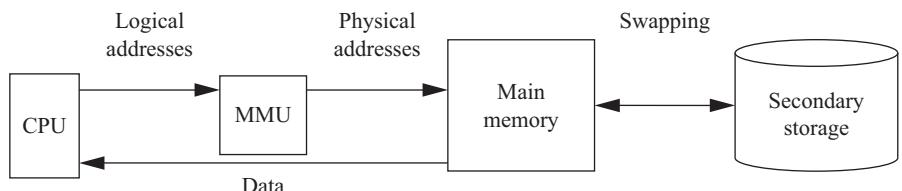
An MMU translates addresses between the CPU and physical memory. This translation process is often known as **memory mapping** because addresses are mapped from a logical space into a physical space. MMUs may not be used in processors for hard real-time applications because the memory mapping process introduces too much variation in execution time. However, many embedded systems with less stringent timing requirements include MMUs to support operating systems like Linux and rich sets of applications. It is helpful to understand the basics of MMUs for embedded systems that are complex enough to require them.

Many DSPs, including the C55x, don't use MMUs. Because DSPs are used for compute-intensive tasks, they often don't require hardware assist for logical address spaces.

Early computers used MMUs to compensate for limited address space in their instruction sets. When memory became cheap enough that physical memory could be larger than the address space defined by the instructions, MMUs allowed software to manage multiple programs in a single physical memory, each with its own address space.

MMUs are used to provide **virtual addressing**. As shown in Fig. 3.10, the MMU accepts logical addresses from the CPU. Logical addresses refer to the program's abstract address space, but do not correspond to actual RAM locations. The MMU translates them from tables to physical addresses that correspond to the RAM. By changing the MMU's tables, you can change the physical location at which the program resides without modifying the program's code or data. We must, of course, move the program in the main memory to correspond to the memory mapping change.

Furthermore, if we add a secondary storage unit, such as a solid state disk, we can eliminate parts of the program from the main memory. In a virtual memory system, the MMU keeps track of which logical addresses are resident in the main memory; those



**FIGURE 3.10**

A virtually addressed memory system.

**Memory mapping philosophy**

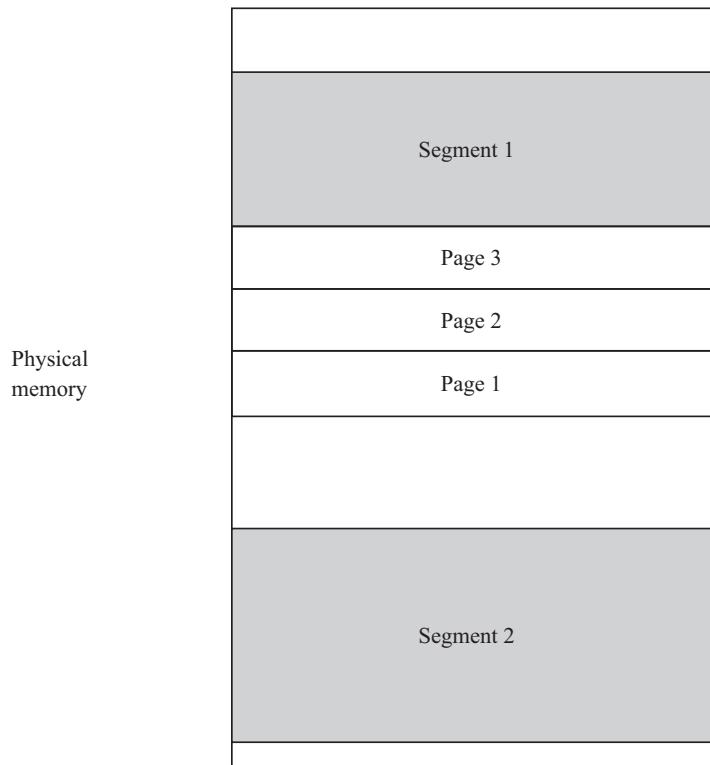
**Virtual addressing**

that do not reside in the main memory are kept on the secondary storage device. When the CPU requests an address that is not in the main memory, the MMU generates an exception called a **page fault**. The handler for this exception executes code that reads the requested location from the secondary storage device into the main memory. The program that generated the page fault is restarted by the handler only after:

- the required memory has been read back into the main memory, and
- the MMU's tables have been updated to reflect the changes.

Of course, loading a location into the main memory will usually require throwing something out of the main memory. The displaced memory is copied into secondary storage before the requested location is read in. As with caches, LRU is a good replacement policy.

There are two styles of address translation: **segmented** and **paged**. Each has advantages, and the two can be combined to form a segmented, paged addressing scheme. As illustrated in Fig. 3.11, segmenting is designed to support a large, arbitrarily sized region of memory, whereas pages describe small, equally sized regions.



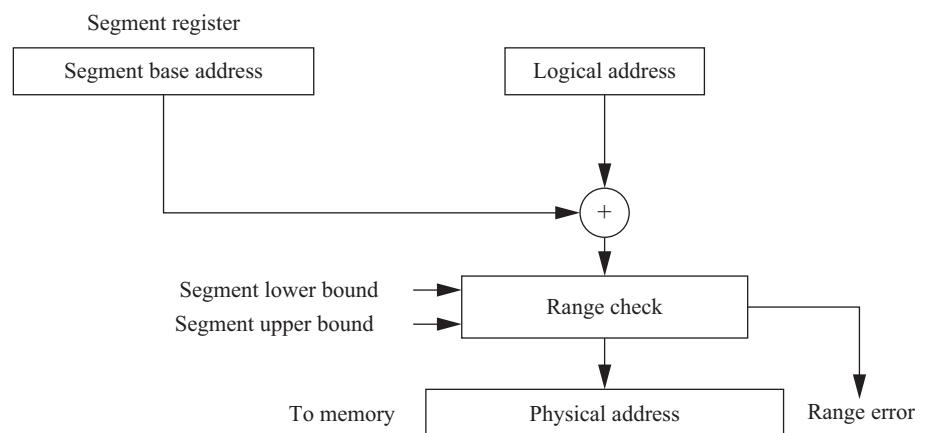
**FIGURE 3.11**

Segments and pages.

A segment is usually described by its start address and size, allowing different segments to be of different sizes. Pages are of uniform size, which simplifies the hardware required for address translation. A segmented, paged scheme is created by dividing each segment into pages and using two steps for address translation. Paging introduces the possibility of **fragmentation**, as program pages are scattered around physical memory.

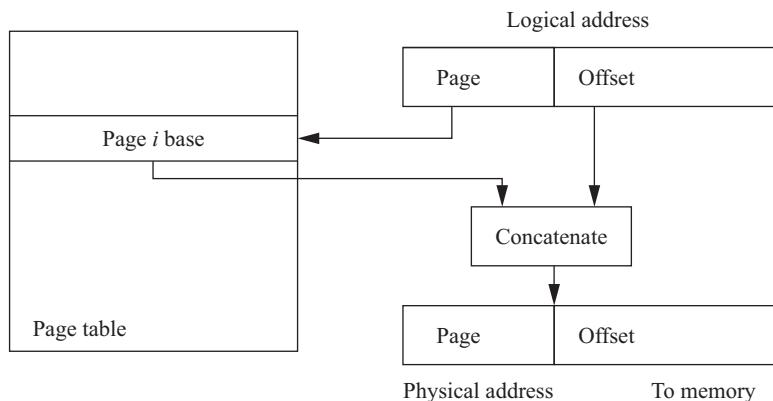
In a simple segmenting scheme, as shown in Fig. 3.12, the MMU maintains a segment register that describes the currently active segment. This register would point to the base of the current segment. The address extracted from an instruction or from any other source for addresses (e.g., a register) would be used as the offset for the address. The physical address is formed by adding the segment base to the offset. Most segmentation schemes also check the physical address against the upper limit of the segment by extending the segment register to include the segment size and comparing the offset to the allowed size.

The translation of paged addresses requires more MMU states, but a simpler calculation than is the case for segmented addressing. As shown in Fig. 3.13, the logical address is divided into two sections: a page number and an offset. The page number is used as an index in a page table, which stores the physical address for the start of each page. However, because all pages have the same size, and it is easy to ensure that page boundaries fall on the proper boundaries, the MMU simply needs to concatenate the top bits of the page starting address with the bottom bits from the page offset to form the physical address. Pages are small, typically between 512 bytes and 4 KB. As a result, an architecture with a large address space requires a large page table. The page table is normally kept in the main memory, which means that an address translation requires memory access.

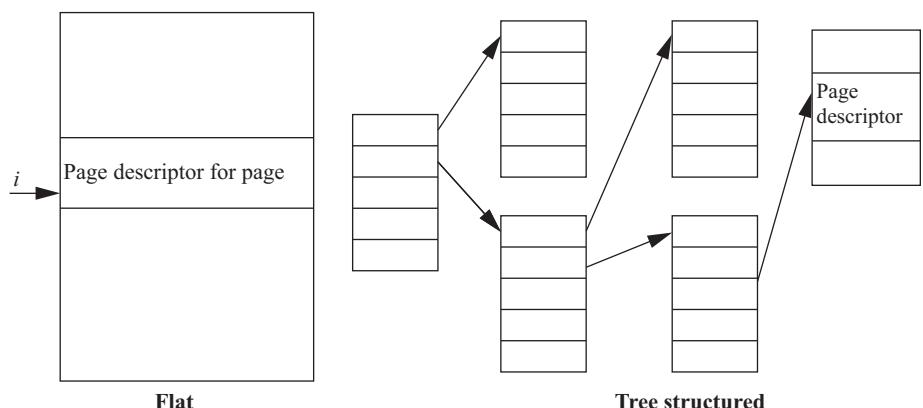


**FIGURE 3.12**

Address translation for a segment.

**FIGURE 3.13**

Address translation for a page.

**FIGURE 3.14**

Alternative schemes for organizing page tables.

The page table may be organized in several ways, as shown in Fig. 3.14. The simplest scheme is a flat table. The table is indexed by the page number, and each entry holds the page descriptor. A more sophisticated method is a tree. The root entry of the tree holds pointers to pointer tables at the next level of the tree; each pointer table is indexed by a part of the page number. We eventually, after three levels in this case, arrive at a descriptor table, which includes the page descriptor in which we are interested. A tree-structured page table incurs some overhead for the pointers, but it also allows us to build a partially populated tree. If some part of the address space is not used, we do not need to build the part of the tree that covers it.

The efficiency of paged address translation may be increased by caching page translation information. The cache for address translation is known as a **translation lookaside buffer (TLB)**. The MMU reads the TLB to check whether a page number is currently in the TLB cache, and if so, uses that value rather than reading from memory.

Virtual memory is typically implemented in a paging or segmented paged scheme so that only page-sized regions of memory need to be transferred on a page fault. Some extensions to both segmenting and paging are useful for virtual memory.

- At a minimum, a present bit is necessary to show whether the logical segment or page is currently in physical memory.
- A dirty bit shows whether the page/segment has been written. This bit is maintained by the MMU because it knows about every write performed by the CPU.
- Permission bits are often used. Some pages/segments may be readable, but not writable. If the CPU supports modes, pages/segments may be accessible by the supervisor, but not in user mode.

A data or instruction cache may operate either on logical or physical addresses, depending on where it is positioned relative to the MMU.

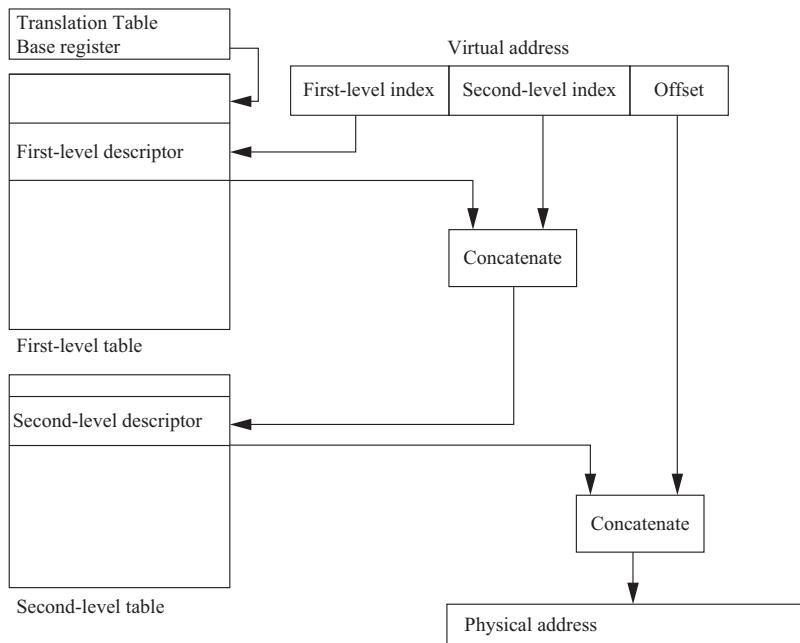
An MMU is an optional part of the Arm architecture. The Arm MMU supports both virtual address translation and memory protection; the architecture requires that the MMU be implemented when cache or write buffers are implemented. The Arm MMU supports the following types of memory regions for address translation:

- A **section** is a 1-MB block of memory.
- A **large page** is 64 KB.
- A **small page** is 4 KB.

An address is marked as a section mapped or page mapped. A two-level scheme is used to translate addresses. The first-level table, which is pointed to by the Translation Table Base register, holds descriptors for section translation and pointers to the second-level tables. The second-level tables describe the translations of both large and small pages. The basic two-level process for a large or small page is illustrated in Fig. 3.15. The details differ between large and small pages, such as the size of the second-level table index. The first- and second-level pages also contain access control bits for virtual memory and protection.

### 3.5.3 Memory protection units

MMUs can provide access protection for memory, but they incur a significant run-time penalty as well as power consumption. An MPU provides access control with a lower overhead. The next example looks at the Arm MPU.

**FIGURE 3.15**


---

ARM two-stage address translation.

---

### **Example 3.9 Arm Memory Protection Unit**

The Arm MPU allows privileged software, such as the operating system, to define up to 16 protected regions of memory. A protected region must be at least 32 bytes in size and no more than 4 GB; a region's size must be a multiple of 32 bytes and begin at a 32-byte aligned location. Normal memory can be assigned several attributes, including cacheability, shareability, and execute/never-execute. Memory associated with I/O devices may be given many attributes, including whether to gather multiple accesses into a single bus transaction, reordering/nonreordering, early/nonearly write acknowledgement.

---

## **3.6 CPU performance**

*Execution time* (how fast the CPU executes instructions) is a particularly important topic in embedded computing. In this section, we consider two factors that can substantially influence program performance: pipelining and caching.

### 3.6.1 Pipelining

Modern CPUs are designed as **pipelined** machines in which several instructions are executed in parallel. Pipelining greatly increases the CPU's efficiency. However, like any pipeline, a CPU pipeline works best when its contents flow smoothly. Some sequences of instructions can disrupt the flow of information in the pipeline and, perhaps temporarily, slow down the operation of the CPU.

#### Arm7 pipeline

The Arm7 has a three-stage pipeline:

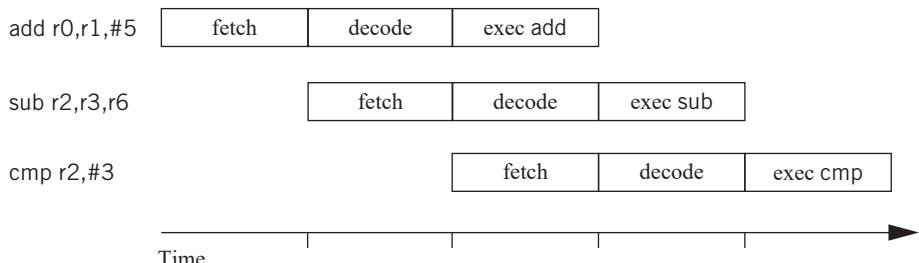
1. *Fetch*: The instruction is fetched from memory.
2. *Decode*: The instruction's opcode and operands are decoded to determine what function to perform.
3. *Execute*: The decoded instruction is executed.

Each of these operations requires one clock cycle for typical instructions. Thus, a normal instruction requires three clock cycles to execute completely. This is known as the **latency** of instruction execution. However, because the pipeline has three stages, an instruction is completed in every clock cycle. Thus, the pipeline has a **throughput** of one instruction per cycle. Fig. 3.16 illustrates the position of the instructions in the pipeline during execution using the notation introduced by Hennessy and Patterson [Hen06]. A vertical slice through the timeline shows all instructions in the pipeline at that time. By following an instruction horizontally, we can see the progress of its execution.

#### C55x pipeline

The C55x includes a seven-stage pipeline [Tex00B]:

1. *Fetch*,
2. *Decode*,
3. *Address*: computes data and branch addresses,
4. *Access 1*: reads data,
5. *Access 2*: finishes data read,
6. *Read stage*: puts operands onto internal busses, and
7. *Execute*: performs operations.



**FIGURE 3.16**

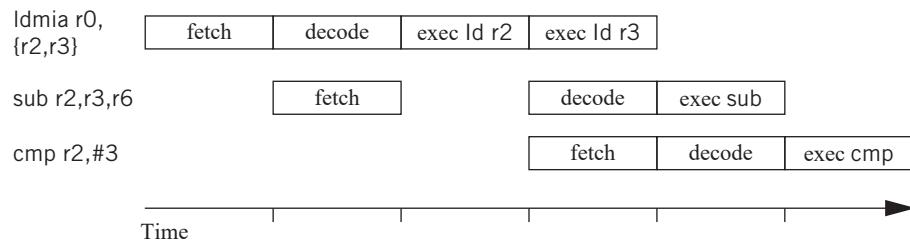
Pipelined execution of ARM instructions.

RISC machines are designed to keep the pipeline busy. Complex instruction-set computers may display a wide variation in instruction timing. Pipelined RISC machines typically have more regular timing characteristics, and most instructions that do not have pipeline hazards display the same latency.

#### Pipeline stalls

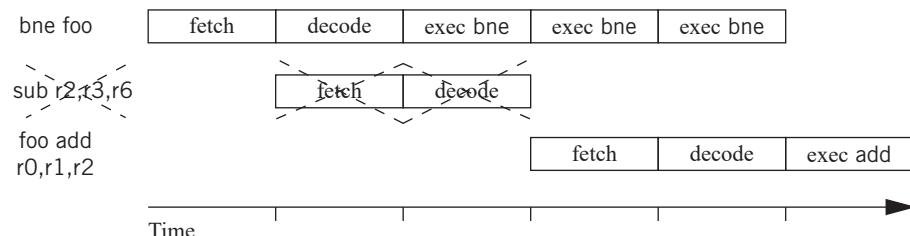
The one-cycle-per-instruction completion rate does not hold in every case. The simplest case for extended execution is when an instruction is too complex to complete the execution phase in a single cycle. A multiple-load instruction is an example of an instruction that requires several cycles in the execution phase. Fig. 3.17 illustrates a **data stall** in the execution of a sequence of instructions, starting with a load-multiple (`LDMIA`) instruction. Because there are two registers to load, the instruction must stay in the execution phase for two cycles. In a multiphase execution, the decode stage is also occupied because it must continue to remember the decoded instruction. As a result, the `SUB` instruction is fetched at the normal time, but is not decoded until the `LDMIA` is finished. This delays the fetching of the third instruction, the `CMP`.

Branches also introduce **control stall** delays into the pipeline, commonly referred to as a **branch penalty**, as shown in Fig. 3.18. The decision whether to take the conditional branch, `BNE`, is not made until the third clock cycle of that instruction's execution, which computes the branch target address. If the branch is taken, the succeeding



**FIGURE 3.17**

Pipelined execution of multicycle ARM instructions.



**FIGURE 3.18**

Pipelined execution of a branch in ARM.

instruction at PC+4 has been fetched and has started to be decoded. When the branch is taken, the branch target address is used to fetch the branch target instruction. Because we must wait for the execution cycle to complete before knowing the target, we must throw away two cycles of work on instructions in the path not taken. The CPU uses the two cycles between starting to fetch the branch target and starting to execute that instruction to finish housekeeping tasks related to the execution of the branch.

One way around this problem is to introduce a **delayed branch**. In this style of branch instruction, a fixed number of instructions directly after the branch is always executed, whether or not the branch is taken. This allows the CPU to keep the pipeline full during execution of the branch. However, some of those instructions after the delayed branch may be no-ops. Any instruction in the delayed branch window must be valid for both execution paths, whether or not the branch is taken. If there are not enough instructions to fill the delayed branch window, it must be filled with no-ops.

Let's use this knowledge of instruction execution time to develop two examples. First, we will look at execution times on the PIC16F; we will then evaluate the execution time of some C code on the more complex Arm.

### Example 3.10 Execution Time of a Loop on PIC16F

The PIC16F is pipelined but has relatively simple instruction timing [Mic07]. An instruction is divided into four Q cycles:

- Q1 decodes instructions;
- Q2 reads operands;
- Q3 processes the data;
- Q4 writes the data.

The time required for an instruction is  $T_{cy}$ . The CPU executes one Q cycle per clock period. Because instruction execution is pipelined, we generally refer to the execution time of an instruction as the number of cycles between it and the next instruction. The PIC16F does not have a cache.

Most instructions are executed in one cycle, but there are exceptions:

- Several flow-of-control instructions (CALL, GOTO, RETFIE, RETLW, and RETURN) always require two cycles.
- Skip-if instructions (DECFSZ, INCFSZ, BTFSC, and BTFSS) require two cycles if the skip is taken and one cycle if the skip is not taken. If the skip is taken, the next instruction remains in the pipeline but is not executed, causing a one-cycle pipeline bubble.

The PIC16F's very predictable timing allows real-time behavior to be encoded into a program. For example, we can set a bit on an I/O device for a data-dependent amount of time [Mic97B]:

```
        movf len, w ; get ready for computed goto
        addwf pcl, 1 ; computed goto (PCL is low byte of PC)
len3:  bsf x,1 ; set the bit at t-3
len2:  bsf x,1 ; set the bit at t-2
len1:  bsf x,1 ; set the bit at t-1
        bcf x,1 ; clear the bit at t
```

A **computed goto** is a general term for a branch to a location determined by a data value. In this case, the variable `len` determines the number of clock cycles for which the I/O device bit is set; this value is copied into the working register, `w`. We perform a computed goto by using the `addwf` register to add a jump value to `pc1`, the lower bits of the program counter. The working register, `w`, is an implicit argument of `addwf`. Its value is added to the value of `pc1`; the final argument `1` determines that the result will be stored back into `pc1`. If we want to set the bit for 3 cycles, we set `len` to 1 so that the computed goto jumps to `len3`. If we want to set the device bit for 2 cycles, we set `len` to two; to set the device bit for one cycle, we set `len` to three. The `bsf` (bit set) and `bcf` (bit clear) instructions take two arguments: the address of the word to be modified and the bit within that word. In this case, the I/O device's state is located at `x`, and we need to set/reset bit 1. Setting the device bit multiple times does not harm the operation of the I/O device. The computed goto allows us to dynamically vary the I/O device's operation time, while still maintaining very predictable timing.

---

### Example 3.11 Execution Time of a Loop on the Arm

We will use the C code for the FIR filter of Application Example 2.1:

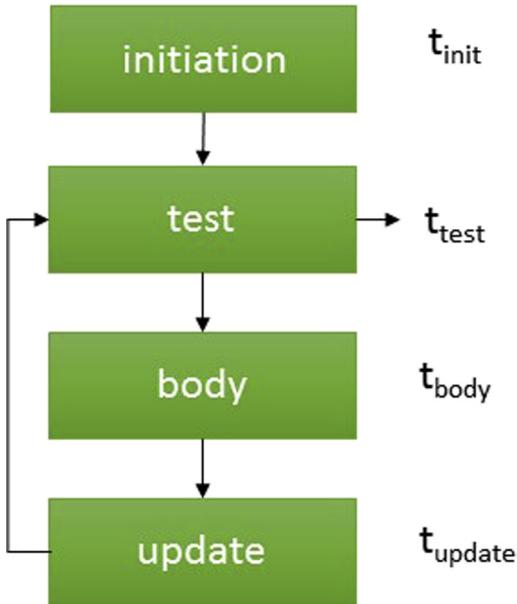
```
for (i = 0, f = 0; i > N; i++)
    f = f + c[i] * x[i];
```

We repeat the Arm code for this loop:

```
;loop initiation code
MOV r0,#0      ;use r0 for i, set to 0
MOV r8,#0      ;use a separate index for arrays
ADR r2,N       ;get address for N
LDR r1,[r2]     ;get value of N for loop termination test
MOV r2,#0      ;use r2 for f, set to 0
ADR r3,c       ;load r3 with address of base of c array
ADR r5,x       ;load r5 with address of base of x array
;test for exit
loop CMP r0,r1
    BGE loopend      ; if i >= N, exit loop
;loop body
    LDR r4,[r3,r8]   ;get value of c[i]
    LDR r6,[r5,r8]   ;get value of x[i]
    MUL r4,r4,r6     ;compute c[i]*x[i]
    ADD r2,r2,r4     ;add into running sum f
    ;update loop counter and array index
    ADD r8,r8,#4      ; add one word offset to array index
    ADD r0,r0,#1      ;add 1 to i
    B loop            ; continue loop
loopend ...
```

Inspection of the code shows that the only instruction that may take more than one cycle is the conditional branch in the loop test.

A block diagram of the code shows how it is broken into pieces for analysis:



Here are the number of instructions and the associated number of clock cycles in each block.

Block	Variable	# Instructions	# Cycles
Initiation	$t_{init}$	7	7
Test	$t_{body}$	2	2 best case, 4 worst case
Body	$t_{update}$	4	4
Update	$t_{test}$	3	4

The unconditional branch at the end of the update block always incurs a branch penalty of two cycles. The BGE instruction in the test block incurs a pipeline delay of two cycles when the branch is taken. That happens only in the last iteration, when the instruction has an execution time of  $t_{test,worst}$ ; in all other iterations, it executes in time  $t_{test,best}$ . We can write a formula for the total execution time of the loop in cycles as

$$t_{loop} = t_{initiation} + N(t_{body} + t_{update} + t_{test,best}) + t_{test,worst} \quad (\text{Eq. 3.3})$$

### 3.6.2 Cache performance

We have already discussed caches functionally. Although caches are invisible in the programming model, they have a profound effect on performance. We introduce caches because they substantially reduce memory access time when the requested location is in the cache. However, the desired location is not always in the cache because it is considerably smaller than the main memory. As a result, caches cause the time required to access memory to vary considerably. The extra time required to access a memory location not in the cache is often called the **cache miss penalty**. The amount of variation depends on several factors in the system architecture, but a cache miss is often several clock cycles slower than a cache hit.

The time required to access a memory location depends on whether the requested location is in the cache. However, as we have seen, a location may not be in the cache for several reasons.

- At a compulsory miss, the location has not been referenced before.
- At a conflict miss, two memory locations fight for the same cache line.
- At a capacity miss, the program's working set is simply too large for the cache.

The contents of the cache can change considerably over the course of the execution of a program. When we have several programs running concurrently on the CPU, we can have very dramatic changes in the cache contents. We need to examine the behavior of the programs running on the system to be able to accurately estimate performance when caches are involved. We consider this problem in more detail in Section 5.7.

---

## 3.7 CPU power consumption

Power consumption is, in some situations, as important as execution time. In this section, we study the characteristics of CPUs that influence power consumption and mechanisms provided by CPUs to control how much power they consume.

### Energy vs. power

First, we need to distinguish between **energy** and **power**. Power is, of course, energy consumption per unit time. Heat generation depends on power consumption. Battery life, on the other hand, most directly depends on energy consumption. Generally, we will use the term, *power*, as shorthand for energy and power consumption, distinguishing between them only when necessary.

### Power and energy

#### 3.7.1 CMOS power consumption

Power and energy are closely related, but they push different parts of the design. The energy required for a computation is independent of the speed at which we perform that work. Energy consumption is closely related to battery life. Power is energy per unit time. In some cases, such as vehicles that run from a generator, we may have limits on the total power consumption of the platform. However, the most

**CMOS power characteristics**

common limitation on power consumption comes from heat generation; more power burned means more heat.

The high-level power consumption characteristics of CPUs and other system components are derived from the circuits used to build those components. Today, virtually all digital systems are built with **complementary metal-oxide-semiconductor (CMOS)** circuits. The detailed circuit characteristics are best left to a study of very large-scale integration design [Wol08], but we can identify two important mechanisms of power consumption in CMOS:

- *Dynamic*: The traditional power consumption mechanism in CMOS circuit is dynamic; the logic uses most of its power when it is changing its output value. If the logic's inputs and outputs do not change, then it does not consume dynamic power. Thus, we can reduce dynamic power consumption by freezing the logic's inputs.
- *Static*: Modern CMOS processes also consume power statically; the nanometer-scale transistors used to make billion-transistor chips are subject to losses that are not important in older technologies with larger transistors. The most important static power consumption mechanism is **leakage**; the transistor draws current even when it is off. The only way to eliminate leakage current is to remove the power supply.

Dynamic and static power consumption require very different management methods. Dynamic power may be saved by running more slowly. Controlling static power requires turning off logic.

As a result, several power-saving strategies are used in CMOS CPUs:

- CPUs can be used at reduced voltage levels. For example, reducing the power supply from 1 V to 0.9 V causes the power consumption to drop by  $1^2/0.9^2 = 1.2 \times$ .
- The CPU can be operated at a lower clock frequency to reduce power (but not energy) consumption.
- The CPU may internally disable certain function units that are not required for the currently executing function. This reduces energy consumption.
- Some CPUs allow parts of the CPU to be totally disconnected from the power supply to eliminate leakage currents.

### 3.7.2 Power management modes

**Static vs. dynamic power management**

CPUs can provide two types of power management modes. A **static power management** mechanism is invoked by the user but does not otherwise depend on CPU activities. An example of a static mechanism is a **power-down mode** intended to save energy. This mode provides a high-level way to reduce unnecessary power consumption. The mode is typically entered with an instruction. If the mode stops the interpretation of instructions, then it clearly cannot be exited by the execution of another instruction. Power-down modes typically end upon receipt of an interrupt or another event. A **dynamic power management** mechanism takes action to control power

based on the dynamic activity in the CPU. For example, the CPU may turn off certain sections of the CPU when the instructions being executed do not require them.

A power-down mode provides the opportunity to greatly reduce power consumption because it will typically be entered for a substantial period. However, going into and especially out of a power-down mode is not free; it costs both time and energy. The power-down or power-up transition consumes time and energy to properly control the CPU's internal logic. Modern pipelined processors require complex controls that must be properly initialized to avoid corrupting data in the pipeline. Starting up the processor must also be done carefully to avoid power surges that could cause the chip to malfunction or even damage it.

Armv8-A processors provide two instructions for standby mode [ARM17]: wait for interrupt (WFI) and wait for event (WFE). Standby mode continues to apply power to the core but stops or gates most of the processor's clocks. The core can be woken up by an interrupt or external debug request in the case of WFI or by specified events in the case of WFE.

The modes of a CPU can be modeled using a **power state machine** [Ben00]. Each state in the machine represents a different mode, and every state is labeled with its average power consumption. The example machine has two states: run mode with power consumption  $P_{\text{run}}$  and sleep mode with power consumption  $P_{\text{sleep}}$ . Transitions show how the machine can go from state to state; each transition is labeled with the time required to go from the source to the destination state. In a more complex example, it may not be possible to go from a particular state to another particular state; traversing a sequence of states may be necessary.

Application Example 3.2 describes the power management modes of the NXP LPC1300.

---

### Application Example 3.2 Power Management Modes of the NXP LPC1311

The NXP LPC1311 [NXP12, NSP11] is an Arm Cortex-M3. It provides four power management modes:

Mode	CPU clock gated?	CPU logic powered?	Static RAM powered?	Peripherals powered?
Active	No	Yes	Yes	Yes
Sleep	Yes	Yes	Yes	Yes
Deep sleep	Yes	Yes	Yes	Most analog blocks shut down (power dip, watchdog remain powered)
Deep power-down	Shut down	No	No	No

In sleep mode, the peripherals remain active and can cause an interrupt that returns the system to run mode. Deep power-down mode is equivalent to a reset on restart.

Here are the static current consumption values for the LPC1311 in the power management states:

Mode	Current @ $V_{DD} = 3.3\text{ V}$
Active	17 mA @ 72 MHz system clock
Sleep	2 mA @ 12 MHz system clock
Deep sleep	30 $\mu\text{A}$
Deep power-down	220 nA

The sleep mode consumes 12% of the current consumed by the active mode. Deep sleep consumes 1.5% of the current required for sleep. The deep power-down mode consumes 0.7% of the current required by deep sleep.

### 3.7.3 Program-level power management

Two classic power management methods have been developed; one aimed primarily at dynamic power consumption and the other at static power consumption. One or a combination of both can be used depending on the characteristics of the technology in which the processor is fabricated.

**Dynamic voltage and frequency scaling (DVFS)** is designed to optimize dynamic power consumption by taking advantage of the relationship between speed and power consumption as a function of power supply voltage:

- The speed of the CMOS logic is proportional to the power supply voltage.
- The power consumption of the CMOS is proportional to the square of the power supply voltage ( $V^2$ ).

Therefore, by reducing the power supply voltage to the lowest level that provides the required performance, we can significantly reduce power consumption. DVFS controllers simultaneously adjust the power supply voltage and clock speed based on a command setting from software.

**Race-to-dark** (also called **race-to-sleep**) is designed to minimize static power consumption. If leakage current is very high, then the best strategy is to run as fast as possible and then shut down the CPU.

DVFS and race-to-dark can be used in combination by selecting a moderate clock speed that is between the values dictated by pure DVFS or race-to-dark. We can understand the trade-off strategy using a model for total energy consumption:

$$E_{tot} = \int_0^T P(t)dt = \int_0^T [P_{dyn}(t) + P_{static}(t)]dt \quad (\text{Eq. 3.4})$$

**Dynamic voltage and frequency scaling**

**Race-to-dark**

**Analysis**

The total energy consumed in an interval is the sum of the dynamic and static components. Static energy is roughly constant (ignoring any efforts by the CPU to temporarily turn off idle units), whereas dynamic power consumption depends on the clock rate. If we turn off the CPU, both components go to zero.

We must also consider the time required to change power supply voltage or clock speed. If mode changes take long enough, the energy lost during the transition may be greater than the savings given by the mode change.

## 3.8 Safety and security

Supervisor mode was an early form of protection for software because the supervisor had more capabilities than user-mode programs. Memory management also provided some security and safety-related features by preventing processes from interfering with each other. However, these mechanisms are viewed as insufficient for secure design.

Programs and hardware resources are both classified as either **trusted** or **untrusted**. A trusted program executes in **A Trusted Execution Environment (TEE)** and is allowed more privileges: the ability to change certain memory locations, access to I/O devices, and so forth. A trusted program can only be started within a trusted environment. Untrusted programs are also not allowed to directly execute trusted programs. If they could, they might be able to obtain a higher level of trust for themselves, which would allow the untrusted program to perform operations for which it does not have permission.

A **root-of-trust** provides a trusted environment for the execution of a well-defined set of programs. We can establish **chains-of-trust** based on this root-of-trust. A trusted execution should be able to trace its provenance back to the root-of-trust.

A hardware root-of-trust relies on both a trusted execution unit and memory that cannot be modified [Loi15]. A controller key is stored in ROM or in a one-time programmable memory. A ROM primary key would be programmed at the factory with a unique key for each device. That primary key is used to verify a public code verification key, also stored in a nonmodifiable memory. After verification, the code verification key can be used to verify software prior to execution. The trusted execution module performs the verification in a tamper-proof manner.

The next example looks at the Arm TEE.

---

### Application Example 3.3 Arm Trusted Execution Environment

Arm Cortex processors provide a set of technologies for secure and trusted computation [ARM13]. Security is based on a four-compartment model:

- *Normal world* includes user and system modes. These modes provide isolation through a combination of operating system and MMU.
- *Hypervisor mode* allows several operating systems to run on the processor as virtual machines. The virtual machine mechanism provides isolation.

- *Trusted World* uses Arm TrustZone to provide partitioning of the system into secure and nonsecure components.
- *SecurCore* provides physically separate chips that are protected against physical and software attacks.

A trusted execution environment provides a secure execution mode that ensures code and data come only from secure addresses; this protection mechanism also applies to peripherals in the secure state. Combining this mechanism with a hypervisor allows critical operations to be kept in the TEE to protect the virtual machines. The virtual machines can then provide a wider range of features.

Arm identifies several characteristics as important for secure operation of a computing platform:

- The processor security mechanisms should limit access to the system. Malicious software should not be able to bypass these mechanisms.
- DMA should provide mechanisms for secure access for both memory and I/O devices.
- Access by other parts of the platform to the central processor should be restricted.
- Trusted software should be able to identify processes that are consuming resources in the nontrusted world; the trusted software should be able to control those processes.
- Debugging functionality, such as a Joint (European) Test Access Group (JTAG) debugging port, should not be able to compromise security.
- I/O devices should support secure/nonsecure operation.
- Platforms should have their own unique, nonmodifiable key.
- The platform should provide secure, trusted storage for private keys.
- Boot processes and update should be secure and verifiable.
- Secure execution should ensure that the control flow of execution in secure mode cannot be subverted.
- The platform should provide security primitives.

The direct hardware support for trusted environments varies: some systems execute functions in hardware that cannot be modified; others put some trusted functions onto programmable processors to reduce cost and enhance flexibility. A trusted environment needs to be able to perform certain trusted functions, provide storage for encryption keys, and support encryption operations. It also requires an interface to the rest of the system that ensures untrusted operations cannot tamper with the trusted environment.

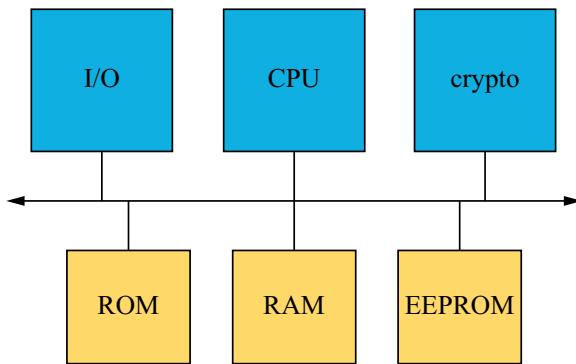
Arm TrustZone [ARM09] allows machines to be designed with many units capable of operating in one of two modes: normal or secure. CPUs with TrustZone have a status bit NS that determines whether it operates in secure or normal mode. Busses, DMA controllers, and cache controllers can also operate in secure mode.

**Smart cards** are widely used for transactions that involve money or other sensitive information. A smart card chip must satisfy several constraints: it must provide secure storage for information; it must allow some of that information to be changed; it must operate at very low energy levels; and it must be manufactured at very low cost.

Fig. 3.19 shows the architecture of a typical smart card [NXP14]. The smart card chip is designed to operate only when an external power source is applied. The I/O unit allows the chip to talk to an external terminal; both traditional electrical contacts and noncontact communication can be used. The CPU has access to RAM, but it also makes use of nonvolatile memory. A ROM may be used to store code that cannot be

#### TrustZone

#### Smart cards

**FIGURE 3.19**

Architecture of a typical smart card.

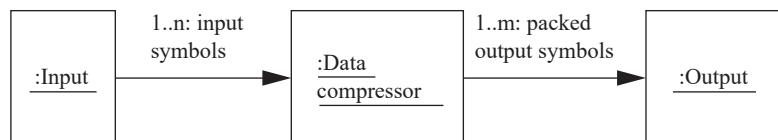
changed. The card may want to change some data or programs and to hold those values even when power is not applied. An electrically erasable programmable ROM (EEPROM) is often used for this nonvolatile memory owing to its very low cost. Specialized circuitry is used to allow the CPU to write to the EEPROM to ensure that write signals are stable even during the CPU operation [Ugo86]. A cryptography unit, coupled with a key that may be stored in the ROM or other permanent storage provides encryption and decryption.

### 3.9 Design example: Data compressor

Our design example for this chapter is a data compressor that takes in data with a constant number of bits per data element, and puts out a compressed data stream in which the data is encoded in variable-length symbols. Because this chapter concentrates on CPUs, we focus on the data compression routine itself. Some architectures add features to make it harder for programs to modify other programs.

#### 3.9.1 Requirements and algorithm

We use the **Huffman coding** technique, which is introduced in Application Example 3.4. We require some understanding of how our compression code fits into a larger system. Fig. 3.20 shows a collaboration diagram for the data compression process. The data compressor takes in a sequence of **input symbols** and then produces a stream of **output**

**FIGURE 3.20**

UML collaboration diagram for the data compressor.

**symbols.** Assume for simplicity that the input symbols are one byte in length. The output symbols are variable length; hence, we must choose a format in which to deliver the output data. Delivering each coded symbol separately is tedious because we would have to supply the length of each symbol and use external code to pack them into words. On the other hand, bit-by-bit delivery is almost certainly too slow. Therefore, we will rely on the data compressor to pack the coded symbols into an array. There is not a one-to-one relationship between the input and output symbols, and we may must wait for several input symbols before a packed output word comes out.

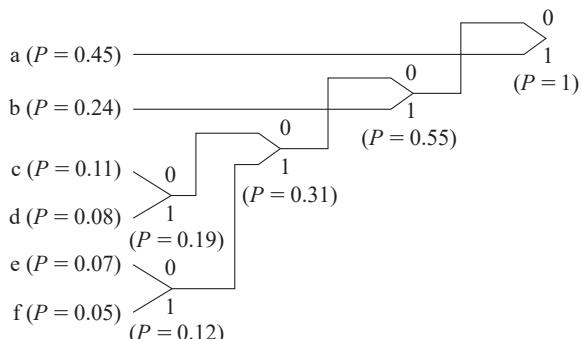
### Application Example 3.4 Huffman Coding for Text Compression

Text compression algorithms aim at statistical reductions in the volume of data. One commonly used compression algorithm is Huffman coding [Huf52], which makes use of information on the frequency of characters to assign variable-length codes to characters. If shorter bit sequences are used to identify more frequent characters, then the length of the total sequence will be reduced.

To decode the incoming bit string, the code characters must have unique prefixes: No code may be a prefix of a longer code for another character. As a simple example of Huffman coding, assume that these characters have the following probabilities,  $P$ , of appearance in a message:

Character	P
A	0.45
B	0.24
C	0.11
D	0.08
E	0.07
F	0.05

We build the code from the bottom up. After sorting the characters by probability, we create a new symbol by adding a bit. We then compute the joint probability of finding either one of those characters and re-sort the table. The result is a tree that we can read top down to find the character codes. Here is the coding tree for our example:



Reading the codes off the tree from the root to the leaves, we obtain the following coding of the characters:

Character	Code
A	1
B	01
C	0000
D	0001
E	0010
F	0011

After the code has been constructed, which in many applications is done offline, the codes can be stored in a table for encoding. This makes encoding simple, but, clearly, the encoded bit rate can vary significantly, depending on the input character sequence. On the decoding side, because we do not know *a priori* the length of a character's bit sequence, the computation time required to decode a character can vary significantly.

The data compressor, as discussed above, is not a complete system, but we can create at least a partial requirements list for the module as seen below. We used the abbreviation N/A for *not applicable* to describe some items that don't make sense for a code module.

Name	Data compression module
Purpose	Code module for Huffman data compression
Inputs	Encoding table, uncoded byte-size input symbols
Outputs	Packed compressed output symbols
Functions	Huffman coding
Performance	Requires fast performance
Manufacturing cost	N/A
Power	N/A
Physical size and weight	N/A

### 3.9.2 Specification

Let's refine the description of Fig. 3.20 to produce a more complete specification for our data compression module. The collaboration diagram concentrates on the

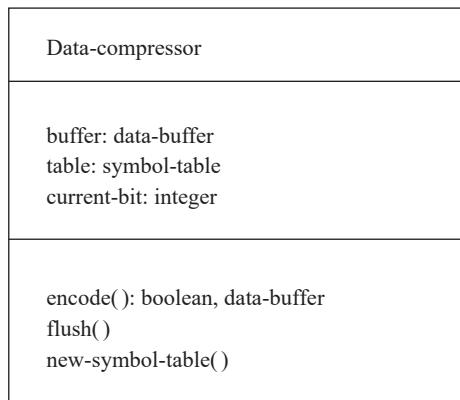
steady-state behavior of the system. For a fully functional system, we must provide some additional behavior:

- We must be able to provide the compressor with a new symbol table.
- We should be able to flush the symbol buffer to cause the system to release all pending symbols that have been partially packed. We may want to do this when we change the symbol table or in the middle of an encoding session to keep a transmitter busy.

A class description for this refined understanding of the requirements on the module is shown in Fig. 3.21. The class's *buffer* and *current-bit* behaviors keep track of the state of the encoding, and the *table* attribute provides the current symbol table. The class has three methods as follows:

- *Encode* performs the basic encoding function. It takes in a 1-B input symbol and returns two values: a Boolean showing whether it is returning a full buffer and, if the Boolean is true, the full buffer itself.
- *New-symbol-table* installs a new symbol table into the object and throws away the current contents of the internal buffer.
- *Flush* returns the current state of the buffer, including the number of valid bits in the buffer.

We also need to define classes for the data buffer and the symbol table. These classes are shown in Fig. 3.22. The *data-buffer* will be used to hold both packed symbols and unpacked ones, such as in the symbol table. It defines the buffer and its length. We must define a data type because the longest encoded symbol is longer than an input symbol. The longest Huffman code for an 8-bit input symbol is 256 bits. Producing a symbol this long happens only when the symbol probabilities have the proper values. The *insert* function packs a new symbol into the upper bits of the buffer; it



**FIGURE 3.21**

Definition of the data-compressor class.

Data-buffer	Symbol-table
databuf[databuflen]: character len: integer	symbols[nsymbols]: data-buffer
insert() length()	value(): symbol load()

**FIGURE 3.22**

Additional class definitions for the data compressor.

also puts the remaining bits in a new buffer if the current buffer is overflowed. The *Symbol-table* class indexes the encoded version of each symbol. The class defines an access behavior for the table; it also defines a *load* behavior to create a new symbol table. The relationships between these classes are shown in Fig. 3.23. A data compressor object includes one buffer and one symbol table.

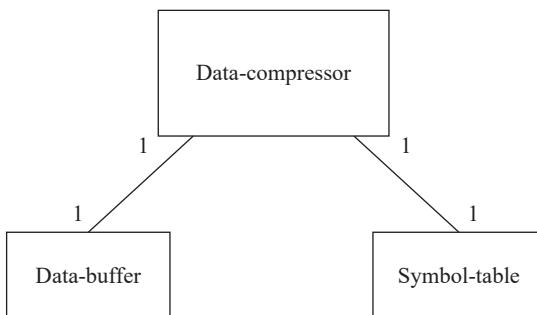
Fig. 3.24 shows a state diagram for the *encode* behavior. It shows that most of the effort goes into filling the buffers with variable-length symbols. Fig. 3.25 shows a state diagram for *insert*. It shows that we must consider two cases; the new symbol does not fill the current buffer or it does.

### 3.9.3 Program design

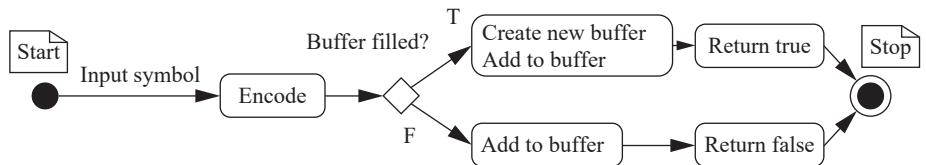
Because we are only building an encoder, the program is fairly simple. We will use this as an opportunity to compare object-oriented (OO) and non-OO implementations by coding the design in both C++ and C.

#### Object-oriented design in C++

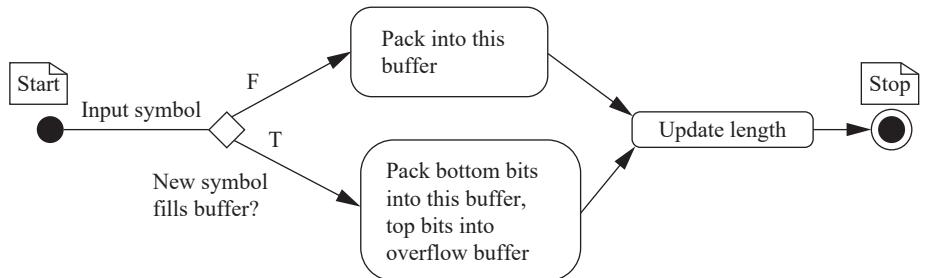
First is the object-oriented design using C++ because this implementation most closely mirrors the specification.

**FIGURE 3.23**

Relationships between classes in the data compressor.

**FIGURE 3.24**

State diagram for encode behavior.

**FIGURE 3.25**

State diagram for insert behavior.

The first step is to design the data buffer. The data buffer must be as long as the longest symbol. We also need to implement a function that lets us merge in another `data_buffer`, shifting the incoming buffer by the proper amount.

```

const int databuflen = 8; /* as long in bytes as longest symbol */
const int bitsperbyte = 8; /* definition of byte */
const int bytemask = 0xff; /* use to mask to 8 bits for safety */
const char lowbitsmask[bitsperbyte] = { 0, 1, 3, 7, 15, 31, 63, 127 };
/* used to keep low bits in a byte */
typedef char boolean; /* for clarity */
#define TRUE 1
#define FALSE 0

class data_buffer {
    char databuf[databuflen] ;
    int len;
    int length_in_chars() { return len/bitsperbyte; } /* length in
bytes rounded down--used in implementation */
public:
    void insert(data_buffer, data_buffer&);
    int length() { return len; } /* returns number of bits in
symbol */
    int length_in_bytes() { return (int)ceil(len/8.0); }
    void initialize(); /* initializes the data structure */
}
  
```

```
void data_buffer::fill(data_buffer, int); /* puts upper bits
of symbol into buffer */
void operator = (data_buffer&); /* assignment operator */
data_buffer() { initialize(); } /* C++ constructor */
~data_buffer() {} /* C++ destructor */
};
data_buffer empty_buffer; /* use this to initialize other
data_buffers */
void data_buffer::insert(data_buffer newval, data_buffer& newbuf) {
    /* This function puts the lower bits of a symbol (newval)
    into an existing buffer without overflowing the buffer. Puts
    spillover, if any, into newbuf. */
    int i, j, bitstoshift, maxbyte;
    /* precalculate number of positions to shift up */
    bitstoshift = length() - length_in_bytes() * bitsperbyte;
    /* compute how many bytes to transfer--can't run past end
    of this buffer */
    maxbyte = newval.length() + length() > databuflen * bitsperbyte ?
        databuflen : newval.length_in_chars();
    for (i = 0; i < maxbyte; i++) {
        /* add lower bits of this newval byte */
        databuf[i + length_in_chars()] |= (newval.databuf[i] <<
            bitstoshift) & bytemask;
        /* add upper bits of this newval byte */
        databuf[i + length_in_chars() + 1] |= (newval.databuf[i]
            >> (bitsperbyte - bitstoshift)) & lowbitsmask[bitsperbyte -
            bitstoshift];
    }
    /* fill up new buffer if necessary */
    if (newval.length() + length() > databuflen * bitsperbyte) {
        /* precalculate number of positions to shift down */
        bitstoshift = length() % bitsperbyte;
        for (i = maxbyte, j = 0; i++, j++; i <= newval.length_
            in_chars()) {
            newbuf.databuf[j] = (newval.databuf[i] >> bitsto-
                shift) & bytemask;
            newbuf.databuf[j] |= newval.databuf[i + 1] &
                lowbitsmask[bitstoshift];
        }
    }
    /* update length */
    len = len + newval.length() > databuflen * bitsperbyte ?
        databuflen * bitsperbyte : len + newval.length();
}
```

```

data_buffer& data_buffer::operator=(data_buffer& e) {
    /* assignment operator for data buffer */
    int i;
    /* copy the buffer itself */
    for (i = 0; i < databuflen; i++)
        databuf[i] = e.databuf[i];
    /* set length */
    len = e.len;
    /* return */
    return e;
}
void data_buffer::fill(data_buffer newval, int shiftamt) {
    /* This function puts the upper bits of a symbol (newval) into
     * the buffer. */
    int i, bitstoshift, maxbyte;
    /* precalculate number of positions to shift up */
    bitstoshift = length() - length_in_bytes() * bitsperbyte;
    /* compute how many bytes to transfer--can't run past end
     * of this buffer */
    maxbyte = newval.length_in_chars() > databuflen ? databuflen :
        newval.length_in_chars();
    for (i = 0; i < maxbyte; i++) {
        /* add lower bits of this newval byte */
        databuf[i + length_in_chars()] = newval.databuf[i] <<
            bitstoshift;
        /* add upper bits of this newval byte */
        databuf[i + length_in_chars() + 1] = newval.databuf[i] >>
            (bitsperbyte - bitstoshift);
    }
}
void data_buffer::initialize() {
    /* Initialization code for data_buffer. */
    int i;
    /* initialize buffer to all zero bits */
    for (i = 0; i < databuflen; i++)
        databuf[i] = 0;
    /* initialize length to zero */
    len = 0;
}

```

The code for `data_buffer` is relatively complex, and not all of its complexity was reflected in the state diagram of Fig. 3.25. That doesn't mean the specification was bad, but it does mean that it was written at a higher level of abstraction.

The symbol table code can be implemented relatively easily:

```

const int nsymbols = 256;
class symbol_table {
    data_buffer symbols[nsymbols];

```

```

public:
    data_buffer *value(int i) { return &(symbols[i]); }
    void load(symbol_table&);
    symbol_table() { } /* C++ constructor */
    ~symbol_table() { } /* C++ destructor */
};

void symbol_table::load(symbol_table& newsyms) {
    int i;
    for (i = 0; i < nsymbols; i++) {
        symbols[i] = newsyms.symbols[i];
    }
}

```

Now let's create the class definition for data\_compressor:

```

typedef char boolean; /* for clarity */
class data_compressor {
    data_buffer buffer;
    int current_bit;
    symbol_table table;
public:
    boolean encode(char, data_buffer&);
    void new_symbol_table(symbol_table newtable)
        { table = newtable; current_bit = 0;
          buffer = empty_buffer; }
    int flush(data_buffer& buf)
        { int temp = current_bit; buf = buffer;
          buffer = empty_buffer; current_bit = 0; return temp; }
    data_compressor() { } /* C++ constructor */
    ~data_compressor() { } /* C++ destructor */
};

```

Now let's implement the encode() method. The main challenge here is managing the buffer.

```

boolean data_compressor::encode(char isymbol, data_buffer& fullbuf) {
    data_buffer temp;
    int overlen;

    /* look up the new symbol */
    temp = *(table.value(isymbol)); /* the symbol itself */
    /* will this symbol overflow the buffer? */
    overlen = temp.length() + current_bit - buffer.length(); /*
       amount of overflow */
    if (overlen > 0) { /* we did in fact overflow */
        data_buffer nextbuf;
        buffer.insert(temp,nextbuf);
        /* return the full buffer and keep the next partial buffer */
        fullbuf = buffer;
    }
}

```

```

        buffer = nextbuf;
        return TRUE;
    } else { /* no overflow */
        data_buffer_no_overflow;
        buffer.insert(temp,no_overflow); /* won't use this argument */
        if (current_bit == buffer.length()) { /* return current buffer */
            fullbuf = buffer;
            buffer.initialize(); /* initialize the buffer */
            return TRUE;
        }
        else return FALSE; /* buffer isn't full yet */
    }
}

```

#### Non-object-oriented implementation

We may not have the luxury of coding the algorithm in C++. Although C is almost universally supported on embedded processors, support for languages that support object orientation, such as C++ or Java, is not universal. How would we structure C code to provide multiple instantiations of the data compressor? If we want to strictly adhere to the specification, we must be able to run several simultaneous compressors, because in the object-oriented specification we can create as many new *data-compressor* objects as we want. To run multiple data compressors simultaneously, we cannot rely on any global variables; all of the object state must be replicable. We can do this relatively easily, making the code only a little more cumbersome. We create a structure that holds the data part of the object as follows:

```

struct data_compressor_struct {
    data_buffer buffer;
    int current_bit;
    sym_table table;
}
typedef struct data_compressor_struct data_compressor,
    * data_compressor_ptr; /* data type declaration for convenience */

```

We would, of course, need to do something similar for the other classes. Depending on how strict we want to be, we may want to define data access functions to get to fields in the various structures we create. C would permit us to get to those struct fields without using the access functions, but using the access functions would give us a little extra freedom to modify the structure definitions later.

We then implement the class methods as C functions, passing in a pointer to the *data\_compressor* object we want to operate on. Here is the beginning of the modified encode method, showing how we make explicit all references to the data in the object.

```

typedef char boolean; /* for clarity */
#define TRUE 1
#define FALSE 0
boolean data_compressor_encode(data_compressor_ptr mycmprs,
    char isymbol, data_buffer * fullbuf) {

```

```

    data_buffer temp;
    int len, overlen;
    /* look up the new symbol */
    temp = mycmprs->table[isymbol].value; /* the symbol itself */
    len = mycmprs->table[isymbol].length; /* its value */
    ...

```

(For C++ aficionados, the above amounts to making explicit the C++ *this* pointer.)

If, on the other hand, we didn't care about the ability to run multiple compressions simultaneously, we could make the functions a little more readable by using global variables for the class variables:

```

static data_buffer buffer;
static int current_bit;
static sym_table table;

```

We have used the C `static` declaration to ensure that these globals are not defined outside the file in which they are defined; this gives us a little added modularity. We would, of course, have to update the specification so that it makes clear that only one compressor object can be run at a time. The functions that implement the methods can then operate directly on the globals, as seen below.

```

boolean data_compressor_encode(char isymbol,
    data_buffer& fullbuf){
    data_buffer temp;
    int len, overlen;
    /* look up the new symbol */
    temp = table[isymbol].value; /* the symbol itself */
    len = table[isymbol].length; /* its value */
    ...

```

Notice that this code does not need the structure pointer argument, causing it to resemble the C++ code a little more closely. However, horrible bugs will ensue if we try to run two different compressions at the same time through this code.

What can we say about the efficiency of this code? Efficiency has many aspects covered in more detail in [Chapter 5](#). For the moment, let's consider instruction selection, that is, how well the compiler does in choosing the right instructions to implement the operations. Bit manipulations, such as those made here, often raise concerns about efficiency. However, if we have a good compiler, and we select the right data types, instruction selection is usually not a problem. If we use data types that don't require data type transformations, a good compiler can select the right instructions to efficiently implement the required operations.

### 3.9.4 Testing

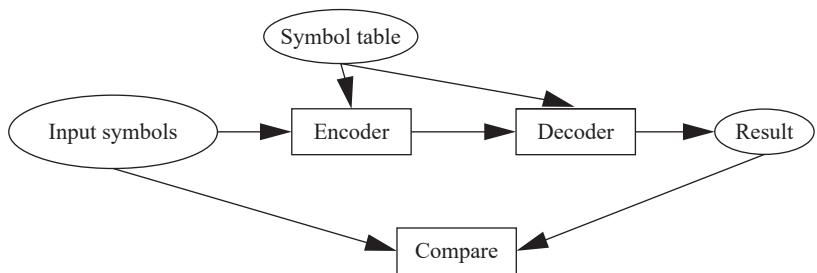
How do we test this program module to be sure it works? We consider testing much more thoroughly in Section 5.11. In the meantime, we can use common sense to come up with some testing techniques.

One way to test the code is to run it and look at the output without considering how the code is written. In this case, we can load up a symbol table, run some symbols through it, and see whether we get the correct result. We can get the symbol table from outside sources or by writing a small program to generate it ourselves. We should test several different symbol tables. We can get an idea of how thoroughly we are covering the possibilities by looking at the encoding trees. If we choose several very different looking encoding trees, we are likely to cover more of the functionality of the module. We also want to test enough symbols for each symbol table. One way to help automate testing is to write a Huffman decoder. As illustrated in Fig. 3.26, we can run a set of symbols through the encoder, and then, through the decoder, simply making sure that the input and output are the same. If they are not, we must check both the encoder and decoder to locate the problem. However, because most practical systems will require both in any case, this is a minor concern.

Another way to test the code is to examine it and try to identify potential problem areas. When we read the code, we should look for places where data operations take place to see that they are performed properly. We also want to look at the conditionals to identify different cases that need to be exercised. Here are some of the issues that we need to consider in testing:

- Is it possible to run past the end of the symbol table?
- What happens when the next symbol does not fill up the buffer?
- What happens when the next symbol exactly fills up the buffer?
- What happens when the next symbol overflows the buffer?
- Do very long encoded symbols work properly? How about very short ones?
- Does `flush()` work properly?

Testing the internals of code often requires building **scaffolding code**. For example, we may want to test the insert method separately, which would require building a program that calls the method with the proper values. If our programming language comes with an interpreter, building such scaffolding is easier because we don't need to create a complete executable. However, we often want to automate such tests even with interpreters because we will usually execute them several times.



**FIGURE 3.26**

A test of the encoder.

### 3.10 Summary

Numerous mechanisms must be used to implement complete computer systems. For example, interrupts have little direct visibility in the instruction set, but they are very important to input and output operations. Similarly, memory management is invisible to most of the program, but is very important to creating a working system.

Although we are not directly concerned with the details of computer architecture, characteristics of the underlying CPU hardware have a major impact on programs. When designing embedded systems, we are typically concerned about characteristics such as execution speed or power consumption. Having some understanding of the factors that determine performance and power will help you later as you develop techniques for optimizing programs to meet these criteria.

---

### What we learned

- The two major styles of I/O are polled and interrupt driven.
- Interrupts may be vectorized and prioritized.
- Supervisor mode helps protect the computer from program errors and provides a mechanism for controlling multiple programs.
- An exception is an internal error; a trap or software interrupt is explicitly generated by an instruction. Both are handled similarly to interrupts.
- A cache provides fast storage for a small number of main memory locations. Caches may be direct mapped or set associative.
- A memory management unit translates addresses from logical to physical addresses.
- Coprocessors provide a way to optionally implement certain instructions in hardware.
- Program performance can be influenced by pipelining, superscalar execution, and the cache. Of these, the cache introduces the most variability into instruction execution time.
- CPUs may provide static (independent of program behavior) or dynamic (influenced by currently executing instructions) methods for managing power consumption.

---

### Further reading

As with instruction sets, the Arm and C55x manuals provide good descriptions of exceptions, memory management, and caches for those processors. Patterson and Hennessy [Pat98] provide a thorough description of computer architecture, including pipelining, caches, and memory management.

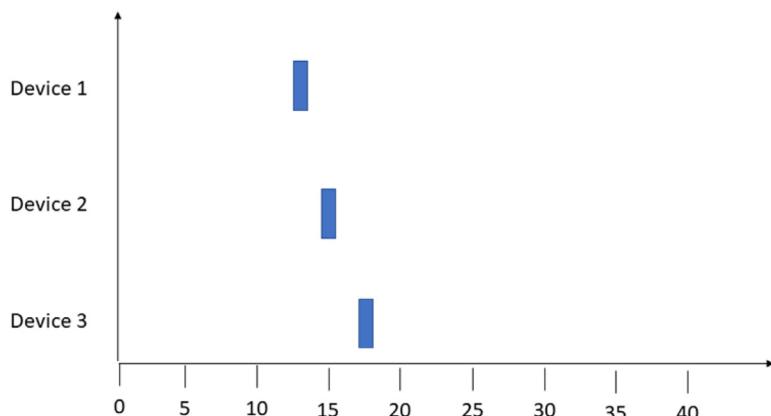
---

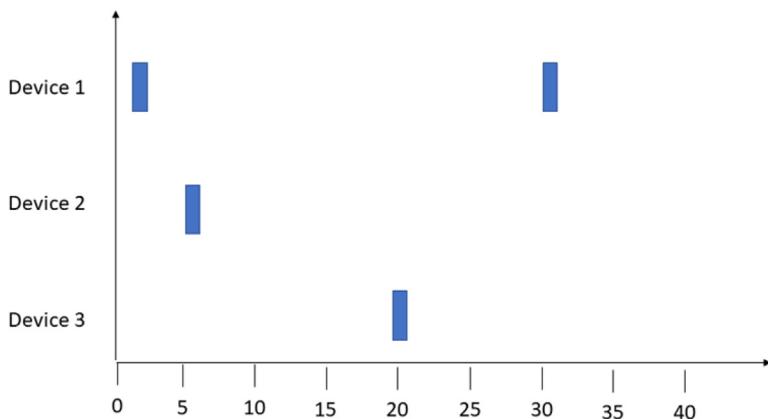
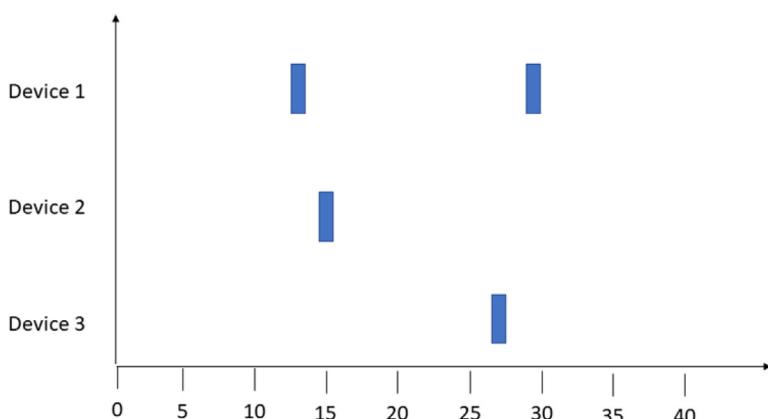
## Questions

- Q3-1** Why do most computer systems use memory-mapped I/O?
- Q3-2** Why do most programs use interrupt-driven I/O over busy/wait?
- Q3-3** Write Arm code that tests a register at location `ds1` and continues execution only when the register is nonzero.
- Q3-4** Write Arm code that waits for the low-order bit of device register `ds1` to become 1, and then, reads a value from register `dd1`.
- Q3-5** Implement `peek()` and `poke()` in assembly language for Arm.
- Q3-6** Draw a UML sequence diagram for a busy-wait read of a device. The diagram should include the program running on the CPU and the device.
- Q3-7** Draw a UML sequence diagram for a busy-wait write of a device. The diagram should include the program running on the CPU and the device.
- Q3-8** Draw a UML sequence diagram for copying characters from an input to an output device using busy-wait I/O. The diagram should include the two devices and the two busy-wait I/O handlers.
- Q3-9** When would you prefer to use busy-wait I/O over interrupt-driven I/O?
- Q3-10** Draw UML diagrams for the read of one character from an 8251 UART. To read the character from the UART, the device needs to read from the data register and to set the serial port status register bit 1 to 0.
  - Draw a sequence diagram that shows the foreground program, the driver, and the UART.
  - Draw a state diagram for the interrupt handler.
- Q3-11** If you could only have one of vectors or priorities in your interrupt system, which would you rather have?
- Q3-12** Draw a UML state diagram for software processing of a vectored interrupt. The vector handling is performed by software (a generic driver) that executes as the result of an interrupt. Assume that the vector handler can read the interrupting device's vector by reading a standard location. Your state diagram should show how the vector handler determines what driver should be called for the interrupt based on the vector.
- Q3-13** Draw a UML sequence diagram for an interrupt-driven read of a device. The diagram should include the background program, the handler, and the device.
- Q3-14** Draw a UML sequence diagram for an interrupt-driven write of a device. The diagram should include the background program, the handler, and the device.

- Q3-15** Draw a UML sequence diagram for a vectored interrupt-driven read of a device. The diagram should include the background program, the interrupt vector table, the handler, and the device.
- Q3-16** Draw a UML sequence diagram for copying characters from an input to an output device using interrupt-driven I/O. The diagram should include the two devices and the two I/O handlers.
- Q3-17** Draw a UML sequence diagram of a higher-priority interrupt that happens during a lower-priority interrupt handler. The diagram should include the device, the two handlers, and the background program.
- Q3-18** Draw a UML sequence diagram of a lower-priority interrupt that happens during a higher-priority interrupt handler. The diagram should include the device, the two handlers, and the background program.
- Q3-19** Draw a UML sequence diagram of a nonmaskable interrupt that happens during a low-priority interrupt handler. The diagram should include the device, the two handlers, and the background program.
- Q3-20** Draw a UML state diagram for the steps performed by an Arm7 when it responds to an interrupt.
- Q3-21** Three devices are attached to a microprocessor: Device 1 has highest priority, and device 3 has lowest priority. Each device's interrupt handler takes five time units to execute. Show what interrupt handler (if any) is executing at each time given these sequences of device interrupts:

a.



**b.****c.**

- Q3-22** Draw a UML sequence diagram that shows how an Arm processor goes into supervisor mode. The diagram should include the supervisor mode program and the user mode program.
- Q3-23** Give three examples of typical types of exceptions handled by CPUs.
- Q3-24** What are traps used for?
- Q3-25** Draw a UML sequence diagram that shows how an Arm processor handles a floating-point exception. The diagram should include the user program, the exception handler, and the exception handler table.

- Q3-26** Provide examples of how each of the following can occur in a typical program:
- compulsory miss
  - capacity miss
  - conflict miss
- Q3-27** You are given a memory system with a main memory access time of 70 ns.
- Plot average memory access time over a range of hit rates [0.94 %, 0.99%] for a cache access time of 3 ns.
  - Plot average memory access time over a range of cache access times [1 ns, 5 ns] for a cache hit rate of 98%.
- Q3-28** If we want an average memory access time of 8 ns, when our cache access time is 3 ns and our main memory access time is 80 ns, what cache hit rate must we achieve?
- Q3-29** In the two-way, set-associative cache with four banks of Example 3.8, show the state of the cache after each memory access, as was done for the direct-mapped cache. Use an LRU replacement policy.
- Q3-30** The following code is executed by an Arm processor with each instruction executed exactly once:

```

MOV r0,#0      ; use r0 for i , set to 0
LDR r1,#10    ; get value of N for loop termination test
MOV r2,#0      ; use r2 for f , set to 0
ADR r3,c      ; load r3 with address of base of c array
ADR r5,x      ; load r5 with address of base of x array
; loop test
loop CMP r0,r1
BGE loopend   ; if i >= N, exit loop
; loop body
LDR r4,[r3,r0] ; get value of c[i]
LDR r6,[r5,r0] ; get value of x[i]
MUL r4,r4,r6  ; compute c[i]*x[i]
ADD r2,r2,r4   ; add into running sum f
; update loop counter
ADD r0,r0,#1   ; add 1 to i
B loop        ; unconditional branch to top of loop

```

Show the contents of the instruction cache for these configurations, assuming each line holds one Arm instruction:

- direct-mapped, two lines
- direct-mapped, four lines
- two-way set-associative, two lines per set

- Q3-31** Show a UML state diagram for a paged address translation using a flat page table.
- Q3-32** Show a UML state diagram for a paged address translation using a three-level, tree-structured page table.
- Q3-33** What are the stages in an Arm7 pipeline?
- Q3-34** What are the stages in the C55x pipeline?
- Q3-35** What is the difference between latency and throughput?
- Q3-36** Draw a pipeline diagram for an Arm7 pipeline's execution of three fictional instructions: aa, bb, and cc. The aa instruction always requires two cycles to complete the execute stage. The cc instruction takes two cycles to complete the execute stage if the previous instruction was a bb instruction. Draw the pipeline diagram for these instruction sequences:
- bb, aa, cc;
  - cc, bb, cc;
  - cc, aa, bb, cc, bb.
- Q3-37** Draw two pipeline diagrams showing what happens when an Arm BZ instruction is:
- taken
  - not taken
- Q3-38** Name two mechanisms by which a CMOS microprocessor consumes power.
- Q3-39** Provide a user-level example of
- static power management
  - dynamic power management
- Q3-40** Can a trusted program start execution of an untrusted program? Explain.

---

## Lab exercises

- L3-1** Write a simple loop that lets you exercise the cache. By changing the number of statements in the loop body, you can vary the cache hit rate of the loop as it executes. If your microprocessor fetches instructions from off-chip memory, you should be able to observe changes in the speed of execution by observing the microprocessor bus.
- L3-2** If your CPU has a pipeline that gives different execution times when a branch is taken or not taken, write a program in which these branches take different amounts of time. Use a CPU simulator to observe the behavior of the program.
- L3-3** Measure the time required to respond to an interrupt.

# Computing Platforms

# 4

## CHAPTER POINTS

---

- CPU buses, I/O devices, and interfacing.
  - The CPU system as a framework for understanding design methodology.
  - System-level performance and power consumption.
  - Security features of platforms.
  - Development environments and debugging.
  - Design examples: Alarm clock, jet engine controller.
- 

## 4.1 Introduction

In this chapter, we concentrate on **computing platforms** created using microprocessors, I/O devices, and memory components. The microprocessor is an important element of the embedded computing system, but it cannot do its job without memories and I/O devices. We need to understand how to interconnect microprocessors and devices using the CPU bus. The application also relies on software that is closely tied to the platform hardware. Luckily, there are many similarities between the platforms required for different applications, so we can extract some generally useful principles by examining a few basic concepts.

The next section surveys the landscape of computing platforms, including both hardware and software. [Section 4.3](#) discusses CPU buses. [Section 4.4](#) describes memory components and systems, while [Section 4.5](#) introduces some useful I/O devices. [Section 4.6](#) considers how to design with computing platforms. [Section 4.7](#) discusses how to design a system using an embedded platform. [Section 4.8](#) develops methods to analyze performance at the platform level, and [Section 4.9](#) considers the power management of the platform. [Section 4.10](#) discusses security support in the platform. We close with two design examples: an alarm clock design example in [Section 4.11](#) and a wearable pedometer in [Section 4.12](#).

---

## 4.2 Basic computing platforms

Although some embedded systems require sophisticated platforms, many can be built around the variations of a generic computer system, ranging from 4-bit microprocessors

through complex systems-on-chips. The platform provides an environment in which we can develop our embedded application. It encompasses both hardware and software components—one without the other is generally not effective.

### 4.2.1 Platform hardware components

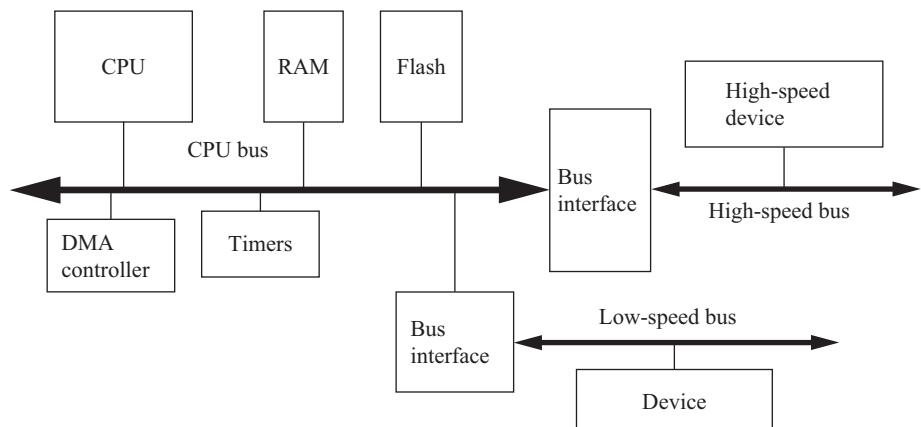
We are familiar with the CPU and memory as an idealized computer system. A practical computer needs additional components. As shown in Fig. 4.1, a typical computing platform includes several major hardware components:

- The CPU provides basic computational facilities.
- RAM is used for program and data storage.
- Flash (also known as ROM) holds the code and data that do not change.
- A direct memory access (DMA) controller provides DMA capabilities.
- Timers are used by the operating system for a variety of purposes.
- A high-speed bus connected to the CPU bus through a bridge allows fast devices to communicate efficiently with the rest of the system.
- A low-speed bus provides an inexpensive way to connect simpler devices and may be necessary for backward compatibility.

#### Buses

The bus provides a common connection between all the components in the computer: the CPU, memories, and I/O devices. We discuss buses in more detail in Section 4.3; the bus transmits addresses, data, and control information so that one device on the bus can read or write to/from another device.

While simple systems will have only one bus, more complex platforms may have several buses or interconnection networks. Buses are often classified by their overall performance: low-speed, high-speed, and so on. Multiple buses serve two purposes. First, devices on different buses will interact far less than those on the same bus. Dividing devices among buses can help reduce the overall load and increase the utilization of the buses. Second, low-speed buses usually provide simpler and cheaper



**FIGURE 4.1**

Hardware architecture of a typical computing platform.

interfaces than high-speed buses. A low-speed device may not benefit from the effort required to connect it to a high-speed bus.

Wide ranges of buses are used in computer systems. The USB, for example, is a bus that uses a small bundle of serial connections. For a serial bus, the USB provides high performance. However, complex buses, such as Peripheral Component Interconnect Express (PCIE), may use many parallel connections and other techniques to provide higher absolute performance.

#### Access patterns

Data transfers may occur between many pairs of components: CPU to/from memory, CPU to/from I/O device, memory to memory, or I/O to I/O device. Because the bus connects all these components (possibly through a bridge), it can mediate all types of transfers. However, basic data transfer requires executing instructions on the CPU. We can use a direct memory access (DMA) unit to offload some of the basic transfers. We discuss DMA in more detail in [Section 4.3](#).

#### Single-chip platforms

We can also put all the components for a basic computing platform on a single chip. A single-chip platform makes the development of certain types of embedded systems much easier, providing rich software development of a PC with the low cost of a single-chip hardware platform. The ability to integrate a CPU and devices on a single chip has allowed manufacturers to provide single-chip systems that do not conform to board-level standards.

#### Microcontrollers

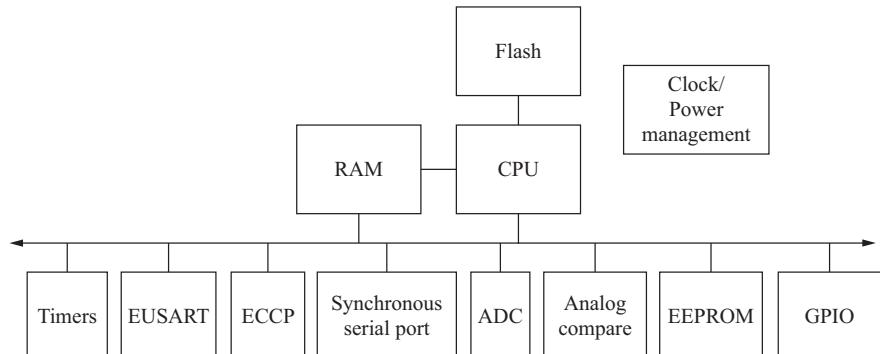
A microcontroller is a single chip that includes a CPU, memory, and I/O devices. The term was originally used for platforms based on small 4- and 8-bit processors, but it can also refer to single-chip systems using large processors.

The next two examples look at two different single-chip systems. Application Example 4.1 looks at PIC16F882, whereas Application Example 4.2 describes Cypress PSoC 6.

---

### Application Example 4.1: System Organization of PIC16F882

Here is a block diagram of the PIC16F882 (as well as 883 and 886) microcontroller [Mic09]:



PIC is a Harvard architecture; the flash memory used for instructions is accessible only to the CPU. The flash memory can be programmed using separate mechanisms. The microcontroller

includes several devices: timers; a universal synchronous/asynchronous receiver/transmitter (USART); capture-and-compare modules; a primary synchronous serial port; an analog-to-digital converter (ADC); analog comparators and references; an electrically erasable programmable ROM; and general-purpose I/O (GPIO).

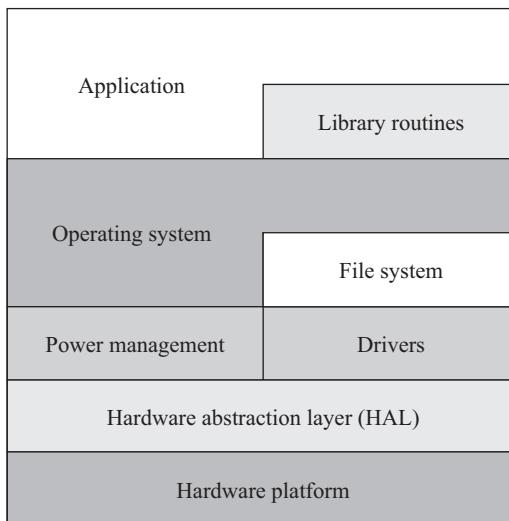
### Application Example 4.2: System Organization of Cypress PSoC 6

Cypress PSoC 6 CY8C62x8 and CY8C62xA [Cyp20] provide dual CPUs: a 150-MHz Arm Cortex-M4F with a single-cycle multiplier and floating-point and memory protection units; and a 100-MHz Cortex-M0+ with a single-cycle multiplier and memory protection unit. The memory system includes flash, static RAM (SRAM), and one-time programmable memory. The power management system provides six power modes. I/O includes several types of serial communication, an audio system, timing and pulse-width modulation, and programmable analog functions.

#### 4.2.2 Platform software components

Hardware and software are inseparable; each needs the other to perform its function. Much of the software in an embedded system comes from outside sources. Hardware vendors generally provide a basic set of software platform components to encourage the use of their hardware. These components range across many layers of abstraction.

Layer diagrams are often used to describe the relationships between different software components in a system. Fig. 4.2 shows a layer diagram of an embedded system. The hardware abstraction layer (HAL) provides a basic level of abstraction from hardware. Device drivers often use the HAL to simplify their structures. Similarly, the



**FIGURE 4.2**

Software layer diagram for an embedded system.

power management module must have low-level access to hardware. The operating system and file system provide the basic abstractions required to build complex applications. Because many embedded systems are algorithm-intensive, we often make use of library routines to perform complex kernel functions. These routines may be developed internally and reused, or in many cases, they come from the manufacturer and are heavily optimized for the hardware platform. The application makes use of all these layers, either directly or indirectly.

## 4.3 The CPU bus

The **bus** is the mechanism by which the CPU communicates with memory and devices. A bus is, at a minimum, a collection of wires, but it also defines a protocol by which the CPU, memory, and devices communicate. One of the major roles of the bus is to provide an interface to memory. Of course, I/O devices also connect to the bus. Based on an understanding of the bus, we can study the characteristics of the memory components in this section, focusing on DMA. We also look at how buses are used in computer systems.

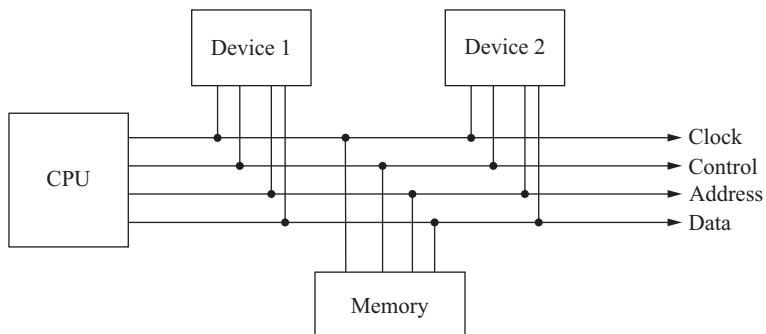
The term *bundle* is used to describe a collection of signals without an associated protocol. We use the terms “controller” and “responder” to describe roles in protocols; the Open Compute Project documents guidelines for inclusive and open terminology [Car20].

### 4.3.1 Bus organization and protocol

A bus is a common connection between components in a system. As shown in Fig. 4.3, the CPU, memory, and I/O devices are all connected to the bus. The signals comprising the bus provide the necessary communication: the data, addresses, a clock, and control signals.

**Bus controller**

In a typical bus system, the CPU serves as the **bus controller** and initiates all transfers. If any device could request a transfer, then other devices might be starved



**FIGURE 4.3**

Organization of a bus.

of bus bandwidth. Via a bus controller, the CPU reads and writes data and instructions from memory. It also initiates all reads or writes on I/O devices. DMA allows other devices to temporarily become the bus controller and transfer data without the CPU's involvement.

#### Four-cycle handshake

The basic building block of most bus protocols is the **four-cycle handshake**, as illustrated in Fig. 4.4. The handshake ensures that, when two devices want to communicate, one is ready to transmit, and the other is ready to receive. The handshake uses a pair of wires dedicated to the handshake: **enq** (meaning enquiry) and **ack** (meaning acknowledge). Extra wires are used for the data transmitted during the handshake. Each step in the handshake is identified by a transition on *enq* or *ack*:

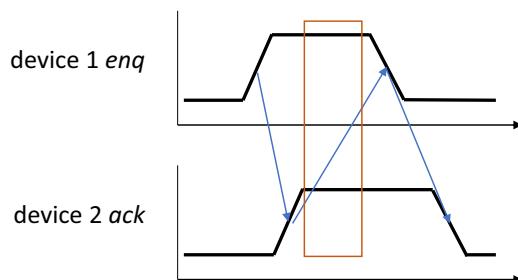
1. *Device 1* raises its *enq* output to signal an enquiry, which tells *device 2* that it should get ready to listen for data.
2. When *device 2* is ready to receive, it raises its *ack* output to signal an acknowledgment.
3. Once the data transfer is complete, *device 1* lowers its *enq* output.
4. After seeing that *enq* has been released, *device 2* lowers its *ack* output.

At the end of the handshake, both handshaking signals are low, just as they were at the start of the handshake. The system thus returns to its original state in readiness for another handshake-enabled data transfer.

#### Bus signals

Microprocessor buses build on the handshake for communication between the CPU and other system components. The term *bus* is used in two ways. The most basic use is as a set of related wires, such as address wires. However, the term may also mean a protocol for communicating between components. To avoid confusion, we will use the term **bundle** to refer to a set of related signals. The fundamental bus operations are reading and writing. The major components on a typical bus include:

- *Clock* provides synchronization to the bus components.
- *R/W* is true when the bus is reading and false when the bus is writing.
- *Address* is an *a*-bit bundle of signals that transmits the address for access.
- *Data* is an *n*-bit bundle of signals that can carry data to or from the CPU.
- *Data-ready* signals when the values on the data bundle are valid.



**FIGURE 4.4**

The four-cycle handshake.

All transfers on this basic bus are controlled by the CPU. The CPU can read or write a device or memory but devices and memory cannot initiate transfers. This is reflected by the fact that *R/W* and address are unidirectional signals because only the CPU can determine the address and direction of a transfer.

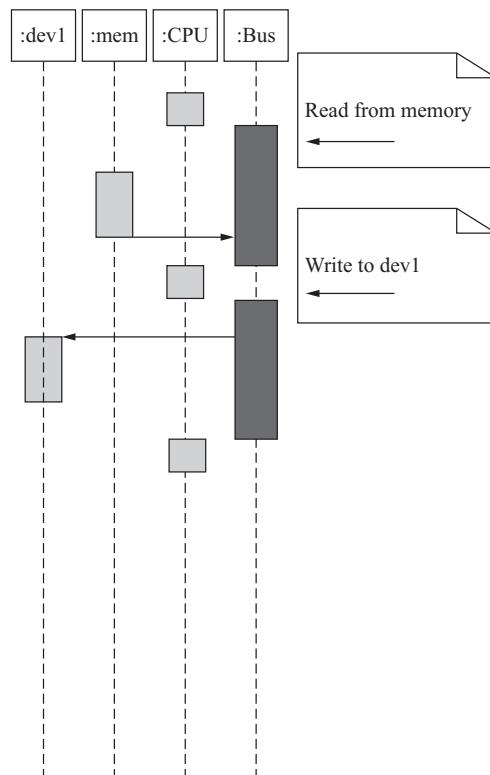
### Bus transactions

### Bus reads and writes

We refer to a read or a write on a bus as a **transaction**. The operation of a bus transaction is governed by the bus protocol. Most modern buses use a clock to synchronize the operations of devices on the bus. The bus clock frequency does not have to match that of the CPU, and in many cases, the bus runs significantly slower than the CPU does.

[Fig. 4.5](#) shows a sequence diagram for a read followed by a write. The CPU first reads a location from the memory, and then, writes it to dev1. The bus mediates each transfer. The bus operates under a protocol that determines when components on the bus can use certain signals and what those signals mean. The details of bus protocols are not important here, but it is important to keep in mind that bus operations take time, since the clock frequency of the bus is often much lower than that of the CPU. We will see how to analyze platform-level performance in [Section 4.7](#).

Sequence diagrams don't give us enough detail to fully understand the hardware. To provide the required details, the behavior of a bus is most often specified as a

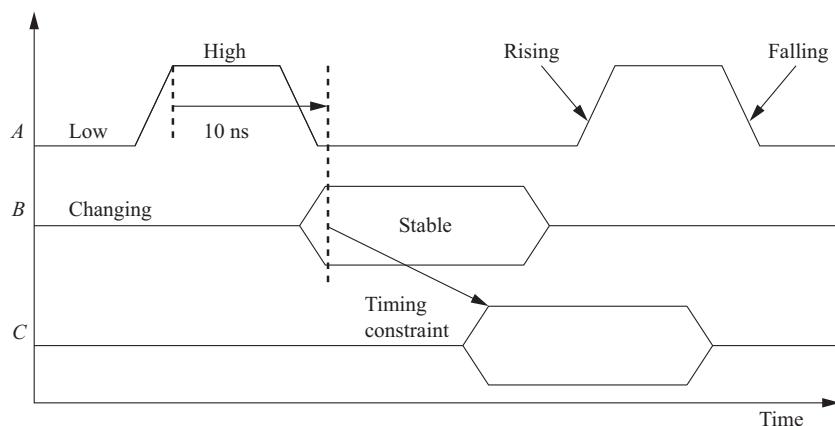


**FIGURE 4.5**

A typical sequence diagram for bus operations.

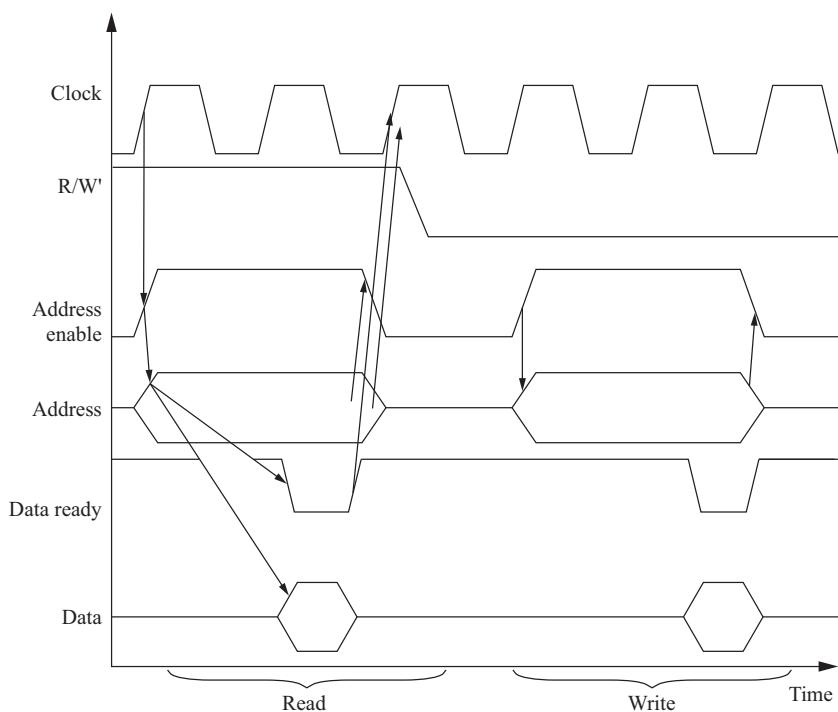
**timing diagram.** A timing diagram shows how the signals on a bus vary over time. Because signals such as address and data can take on many values, some standard notation is used to describe signals, as shown in Fig. 4.6. A's value is known at all times; thus, it is shown as a standard waveform that changes between zero and one. B and C alternate between **changing** and **stable** states. A stable signal has, as the name implies, a stable value that can be measured by an oscilloscope, but the exact value of that signal does not matter for the purposes of the timing diagram. For example, an address bus may be shown to be stable when the address is present, but the bus's timing requirements are independent of the exact address on the bus. A signal can go between a known 0/1 state and a stable/changing state. A changing signal does not have a stable value. Changing signals should not be used for computation. Timing diagrams sometimes show **timing constraints** to describe the precise relationships in time between events on the signals. We draw timing constraints in two different ways depending on whether we are concerned with the amount of time between events or only the order of events. The timing constraint from A to B, for example, shows that A must go high before B becomes stable. The constraint from A to B also has a time value of 10 ns, indicating that A goes high for at least 10 ns before B becomes stable.

Fig. 4.7 shows a timing diagram for the example bus. The diagram shows a read followed by a write. Timing constraints are shown only for the read operation, but similar constraints apply to the write operation. The bus is normally in read mode because that does not change the state of any of the devices or memories. The CPU can then ignore the bus data lines until it wants to use the results of a read. Notice also that the direction of data transfer on bidirectional lines is not specified in the timing diagram. During a read, the external device or memory sends a value on the data lines, whereas during a write, the CPU controls the data lines.



**FIGURE 4.6**

Timing diagram notation.

**FIGURE 4.7**

Timing diagram for read and write on the example bus.

With practice, we can see the sequence of operations for a read on the timing diagram:

- A read or write is initiated by setting the address enable high after the clock starts to rise. We set  $R/W' = 1$  to indicate a read, and the address lines are set to the desired address.
- One clock cycle later, the memory or device is expected to assert the data value at that address on the data lines. Simultaneously, the external device specifies that the data are valid by pulling down the *data-ready* line. This line is **active low**, meaning that a logically true value is indicated by a low voltage to provide increased immunity from electrical noise.
- The CPU is free to remove the address at the end of the clock cycle and must do so before the beginning of the next cycle. The external device has a similar requirement for removing the data value from the data lines.

The write operation has a similar timing structure. The read/write sequence illustrates that timing constraints are required for the transition of the  $R/W'$  signal between read and write states. The signal must, of course, remain stable within a read or write.

As a result, there is a restricted time window in which the CPU can change between the read and write modes.

The handshake that tells the CPU and devices when data are to be transferred is formed by *data-ready* for the acknowledge side but is implicit for the enquiry side. Because the bus is normally in read mode, *enq* does not need to be asserted, but the acknowledgment must be provided by *data-ready*.

The *data-ready* signal allows the bus to be connected to devices that are slower than the bus. As shown in Fig. 4.8, the external device does not need to immediately assert *data-ready*. The cycles between the minimum time at which data can be asserted and when they are actually asserted are known as **wait states**. Wait states are commonly used to connect slow, inexpensive memories to buses.

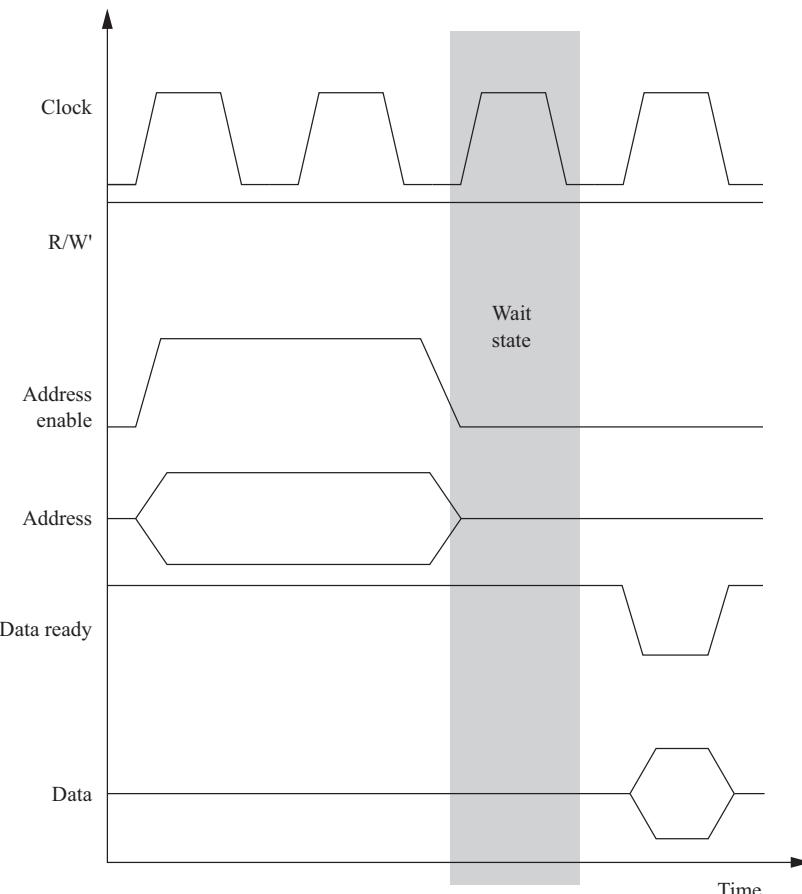


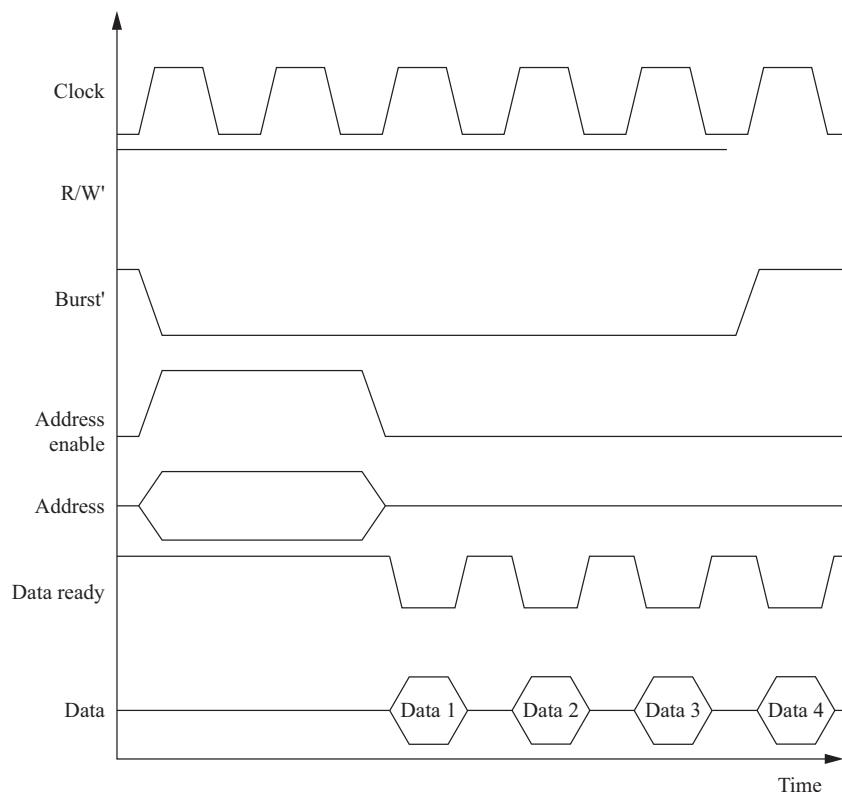
FIGURE 4.8

A wait state on a read operation.

We can also use bus handshaking signals to perform **burst transfers**, as illustrated in Fig. 4.9. In this burst read transaction, the CPU sends one address but receives a sequence of data values. We add an extra line to the bus, called *burst'* here, which signals when a transaction is actually a burst. Releasing the *burst'* signal tells the device that enough data have been transmitted. To stop receiving data after the end of *data 4*, the CPU releases the *burst'* signal at the end of *data 3* because the device requires some time to recognize the end of the burst. These values come from successive memory locations starting at the given address.

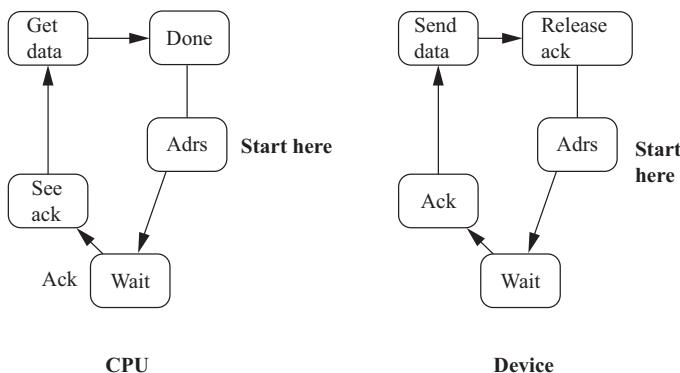
Some buses provide **disconnected transfers**. In these buses, the requests and responses are separate. The first operation requests a transfer. The bus can then be used for other operations. The transfer is completed later, when the data are ready.

The state machine view of the bus transaction is also helpful and a useful complement to the timing diagram. Fig. 4.10 shows the CPU and device state machines for the read operation. As with a timing diagram, we do not show all possible values of



**FIGURE 4.9**

A burst read transaction.

**FIGURE 4.10**

State diagrams for the bus read transaction.

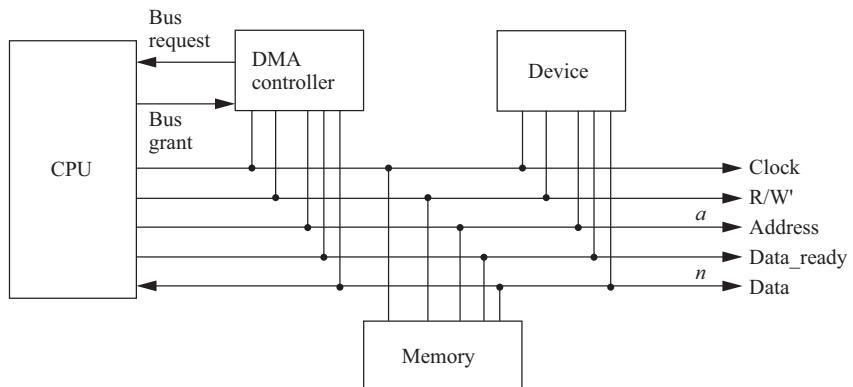
address and data lines; instead, we concentrate on the transitions of control signals. When the CPU decides to perform a read transaction, it moves to a new state, sending bus signals that cause the device to behave appropriately. The device's state transition graph captures its side of the protocol.

Some buses have data bundles that are smaller than the natural word size of the CPU. Using fewer data lines reduces the cost of the chip. Such buses are easiest to design when the CPU is natively addressable. A more complicated protocol hides the smaller data sizes from the instruction execution unit in the CPU. Byte addresses are sequentially sent over the bus, receiving one byte at a time; the bytes are assembled inside the CPU's bus logic before being presented to the CPU proper.

### 4.3.2 Direct memory access

Standard bus transactions require the CPU to be in the middle of every read and write transaction. However, there are certain types of data transfers in which the CPU does not need to be involved. For example, a high-speed I/O device may want to transfer a block of data into the memory. Although it is possible to write a program that alternately reads the device and writes to memory, it would be faster to eliminate the CPU's involvement and allow the device and memory communicate directly. This capability requires that some unit other than the CPU be able to control operations on the bus.

**Direct memory access (DMA)** is a bus operation that allows reads and writes not controlled by the CPU. A DMA transfer is controlled by a **DMA controller**, which requests control of the bus from the CPU. After gaining control, the DMA controller performs read and write operations directly between the devices and the memory.

**FIGURE 4.11**

A bus with a direct memory access controller.

[Fig. 4.11](#) shows the configuration of a bus with a DMA controller. The DMA requires the CPU to provide two additional bus signals:

- The **bus request** is an input to the CPU through which the DMA controllers ask for ownership of the bus.
- The **bus grant** signals that the bus has been granted to the DMA controller.

The DMA controller can act as a bus controller. It uses the bus request and bus grant signal to gain control of the bus using a classic four-cycle handshake. A bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready. The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the other bus signals: R/W', address, and so on. Upon becoming a bus controller, the DMA controller has control of all bus signals (except, of course, for bus requests and bus grants).

Once the DMA controller is a bus controller, it can perform reads and writes using the same bus protocol as with any CPU-driven bus transaction. Memory and devices do not know whether a read or write is performed by the CPU or by a DMA controller. After the transaction is finished, the DMA controller returns the bus to the CPU by de-asserting the bus request, causing the CPU to de-assert the bus grant.

The CPU controls the DMA operation through registers in the DMA controller. A typical DMA controller includes the following three registers:

- A starting address register specifies where the transfer is to begin.
- A length register specifies the number of words to be transferred.
- A status register allows the DMA controller to be operated by the CPU.

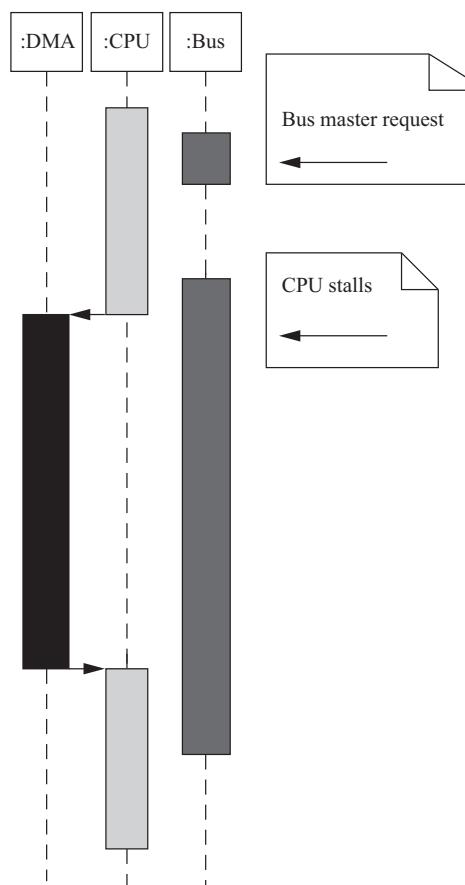
The CPU initiates a DMA transfer by setting the starting address and length registers appropriately, and then, writing the status register to set its start transfer bit.

After the DMA operation is complete, the DMA controller interrupts the CPU and tells it that the transfer is done.

#### Concurrency during DMA

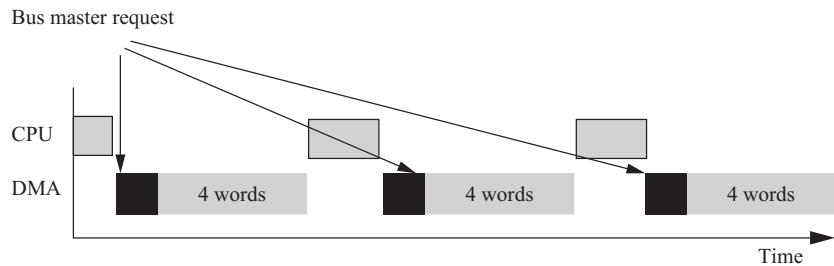
Because the CPU cannot use the bus during a DMA transfer, what does it do? As illustrated in Fig. 4.12, if the CPU has enough instructions and data in the cache and registers, it may be able to continue doing useful work for quite some time and may not notice the DMA transfer. However, when the CPU needs the bus, it stalls until the DMA controller returns bus controllership to the CPU.

To prevent the CPU from idling too long, most DMA controllers implement modes that occupy the bus for only a few cycles at a time. For example, the transfer may be made 4, 8, or 16 words at a time. As illustrated in Fig. 4.13, after each block, the DMA controller returns control of the bus to the CPU and goes to sleep for a preset period, after which it requests the bus again for the next block transfer.



**FIGURE 4.12**

UML sequence of system activity around a direct memory access transfer.

**FIGURE 4.13**

Cyclic scheduling of a direct memory access request.

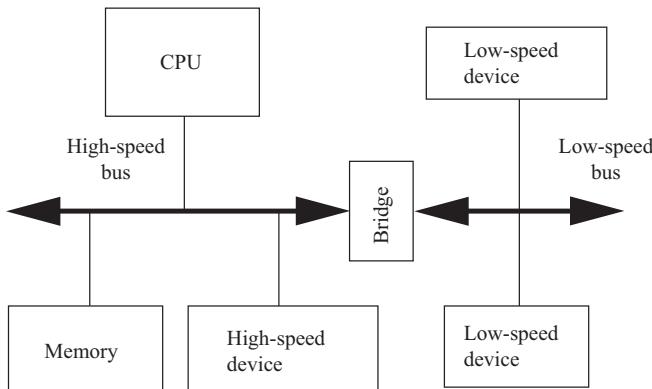
### 4.3.3 System bus configurations

A microprocessor system often has more than one bus. As shown in Fig. 4.14, high-speed devices may be connected to a high-performance bus, whereas lower-speed devices may be connected to a different bus. A small block of logic, known as a **bridge**, allows the buses to connect to each other. There are three reasons to do this:

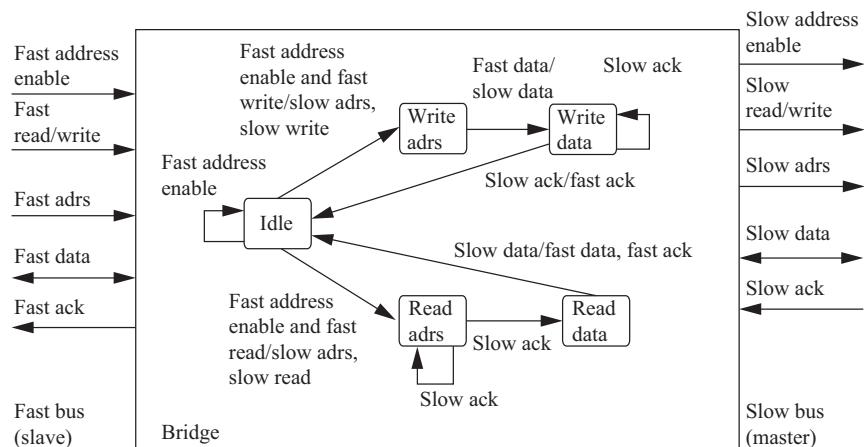
- Higher-speed buses may provide wider data connections.
- A high-speed bus usually requires more expensive circuits and connectors. The cost of low-speed devices can be reduced by using a lower-speed, lower-cost bus.
- The bridge may allow buses to operate independently, thereby providing some parallelism in I/O operations.

#### Bus bridges

Let's consider the operation of a bus bridge between what we call a fast bus and a slow bus, as illustrated in Fig. 4.15. The bridge is a responder on the fast bus and the controller of the slow bus. The bridge takes commands from the fast bus, upon which it is a responder, and issues those commands on the slow bus. It also returns the results

**FIGURE 4.14**

A multiple bus system.

**FIGURE 4.15**

Unified modeling language state diagram of bus bridge operation.

from the slow bus to the fast bus. For example, it returns the results of a read on the slow bus to the fast bus.

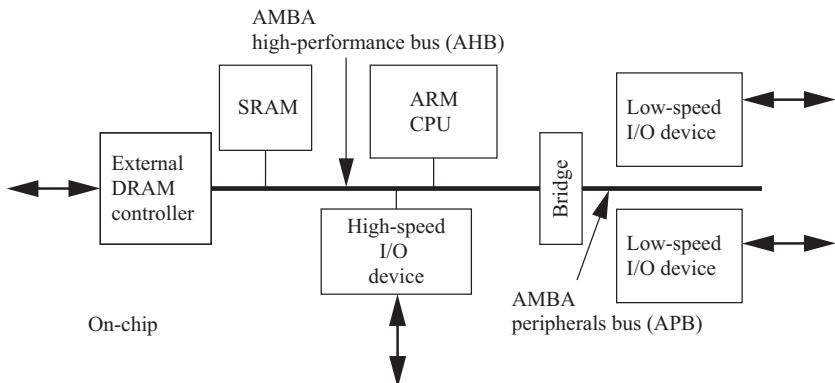
The upper sequence of states handles a write from the fast bus to the slow bus. These states must read the data from the fast bus and set up the handshake for the slow bus. Operations on the fast and slow sides of the bus bridge should overlap as much as possible to reduce the latency of bus-to-bus transfers. Similarly, the bottom sequence of states reads from the slow bus and writes the data to the fast bus.

The bridge also serves as a protocol translator between the two bridges. If the bridges are very close in protocol operation and speed, a simple-state machine may be enough. If there are larger differences in the protocol and timing between the two buses, the bridge may need to use registers to hold some data values temporarily.

#### Arm bus

Because the Arm CPU is manufactured by many different vendors, the bus provided off-chip can vary from chip to chip. ARM has created a separate bus specification for single-chip systems. The Advanced Microcontroller Bus Architecture (AMBA) bus [ARM99A] supports CPUs, memories, and peripherals integrated into a system-on-silicon. As shown in Fig. 4.16, the AMBA specification includes two buses. The AMBA High-Performance Bus (AHB) is optimized for high-speed transfers and is directly connected to the CPU. It supports several high-performance features: pipelining, burst transfers, split transactions, and multiple bus controllers.

A bridge can be used to connect the AHB to an AMBA Peripheral Bus (APB). This bus is designed to be simple and easy to implement; it also consumes relatively little power. The APB assumes that all peripherals act as responders, simplifying the logic required for both the peripherals and the bus controller. It also does not perform pipelined operations, which simplifies bus logic.

**FIGURE 4.16**


---

Elements of the Arm AMBA bus system.

---

## 4.4 Memory devices and systems

RAMs can be both read and written. They are called random access because, unlike magnetic disks, addresses can be read in any order. Most bulk memory in modern systems is **dynamic RAM (DRAM)**. DRAM is very dense; however, it does require that its values be **refreshed** periodically because the values inside the memory cells decay over time.

### Basic DRAM organization

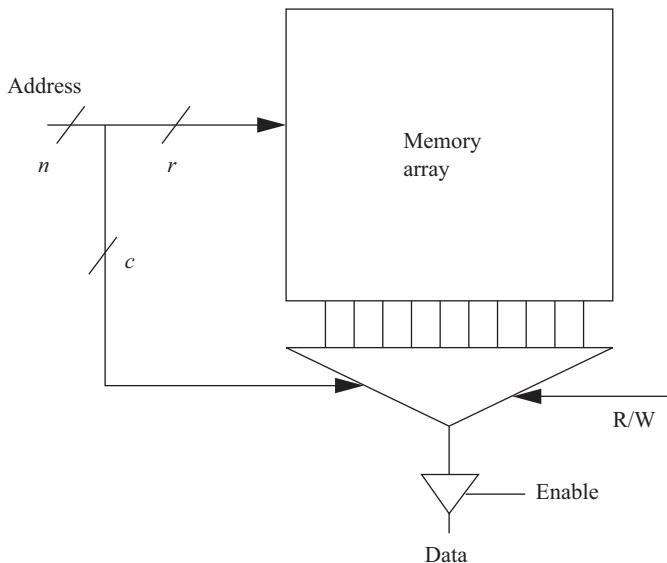
Although the basic organization of memories is simple, several variations exist that provide trade-offs [Cup01]. As shown in Fig. 4.17, a simple memory is organized as a two-dimensional array. Assume, for the moment, that the memory is accessed one bit at a time. The address for that bit is split into two sections: rows and columns. Together, they form the complete address in the array. If we want to access more than one bit at a time, we can use fewer bits in the column part of the address to select several columns simultaneously. The division of an address into rows and columns is important because it is reflected at the pins of the memory chip and is thus visible to the rest of the system. In a traditional DRAM, the row is sent first, followed by the column. Two control signals tell the DRAM when those address bits are valid: row address select or RAS and column address select or CAS.

### Refreshing

The DRAM must be refreshed periodically to retain its values. Rather than refreshing the entire memory at once, DRAMs refresh a part of the memory at a time. When a section of the memory is being refreshed, it can't be accessed until the refresh is complete. The DRAM cycles through the memory, refreshing it section by section.

### Bursts and page mode

Memories may offer some special modes that reduce the time required for access. Bursts and page mode accesses are both more efficient forms of access, but they differ in how they work. Burst transfers perform several accesses in sequence using a single

**FIGURE 4.17**

Organization of a basic memory.

address and possibly a single CAS signal. Page mode, in contrast, requires a separate address for each data access.

The most common type of DRAM is synchronous DRAM (SDRAM). A separate family of standards provides synchronous DRAM for graphics applications.

SDRAMs use RAS and CAS signals to break the address into two parts, which select the proper row and column in the RAM array. Signal transitions are relative to the SDRAM clock, which allows the internal SDRAM operations to be pipelined. As shown in Fig. 4.18, transitions in the control signals are related to a clock [Mic00]. SDRAMs include registers that control the mode in which the SDRAM operates. SDRAMs support burst modes that allow several sequential addresses to be accessed by sending only one address. SDRAMs generally also support an interleaved mode that exchanges pairs of bytes.

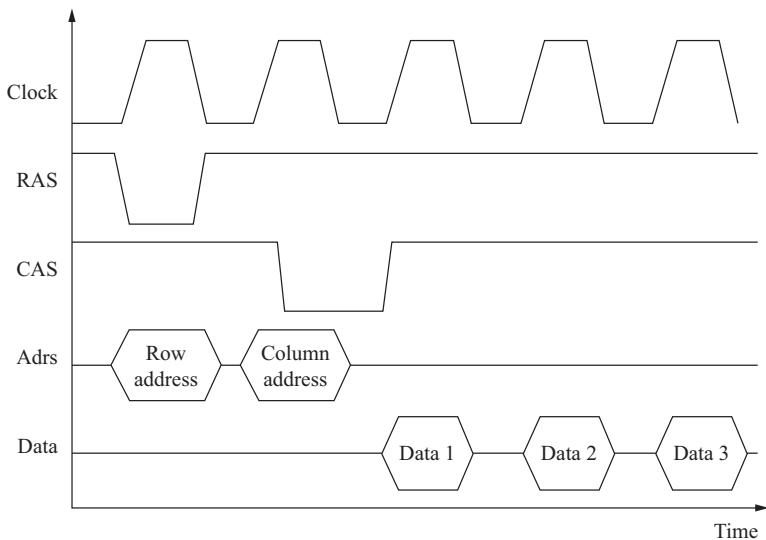
Memory for PCs is generally purchased as **single in-line memory modules (SIMMs)** or **double in-line memory modules (DIMMs)**. A SIMM or DIMM is a small circuit board that fits into a standard memory socket. A DIMM has two sets of leads compared with a SIMM's one. Memory chips are soldered to the circuit board to supply the desired memory.

**ROMs** are preprogrammed with fixed data and are useful in embedded systems because a great deal of the code and perhaps some data do not change over time. **Flash memory** is the dominant form of ROM. Flash memory can be erased and rewritten using standard system voltages, allowing it to be reprogrammed inside a typical system. This allows applications, such as automatic distribution of upgrades, wherein the

#### Types of DRAM

#### Synchronous dynamic RAM

#### Memory packaging

**FIGURE 4.18**

An synchronous dynamic RAM read operation.

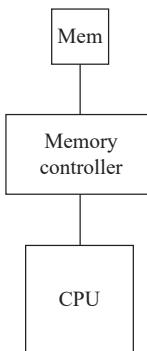
flash memory can be reprogrammed while downloading the new memory contents from a telephone line. Early flash memories had to be erased in their entirety; modern devices allow memory to be erased in blocks. Most flash memories today allow certain blocks to be protected. A common application is to keep the boot-up code in a protected block while allowing updates to other memory blocks on the device. As a result, this form of flash is commonly known as **boot-block flash** [Int03]. **One-time programmable** (OTP) memory is based on a different storage cell (an anti-fuse) that can be programmed once. OTP memories are typically small in capacity.

#### 4.4.1 Memory system organization

Modern memory is more than a one-dimensional array of bits. Memory chips have surprisingly complex organizations that allow us to make useful optimizations. For example, memories are usually divided into several smaller memory arrays.

##### Memory controllers

Modern computer systems use a memory controller as the interface between the CPU and the memory components. As shown in Fig. 4.19, the memory controller shields the CPU from knowledge of the detailed timing of different memory components. If the memory also consists of several different components, the controller will manage all accesses to all memories. Memory accesses must be scheduled. The memory controller receives a sequence of requests from the processor. However, it may not be possible to execute them as quickly as they are received if the memory component is already processing access. When faced with more accesses than the resources

**FIGURE 4.19**

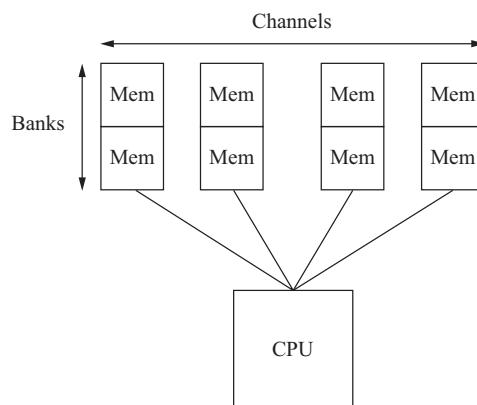
The memory controller in a computer system.

available to complete them, the memory controller will determine the order in which they will be handled and schedule the accesses accordingly.

#### Channels and banks

Channels and banks are two ways to add parallelism to the memory system. A channel is a connection to a group of memory components. If the CPU and memory controller can support multiple channels that operate concurrently, we can perform multiple independent accesses using the different channels. We may also divide the complete memory system into banks. Banks can perform accesses in parallel because each bank has its own memory arrays and addressing logic. By properly arranging memory into banks, we can overlap some of the access times for these locations and reduce the total time required for the complete set of accesses.

[Fig. 4.20](#) shows a memory system organized into channels and banks. Each channel has its own memory components and connection to the processor. Channels operate completely separately. The memory in each channel can be subdivided into banks. The banks in a channel can be accessed separately. Channels are, in general, more

**FIGURE 4.20**

Channels and banks in a memory system.

expensive than banks. A two-channel memory system, for example, requires twice as many pins and wires connecting the CPU and memory as a one-channel system does. Memory components are often separated internally into banks, providing that access to the outside is less expensive.

## 4.5 I/O devices

While a wide range of I/O devices can be built for computer systems, a few basic types provide important functionalities for embedded computing systems.

### Counter

A **counter** is a register with logic to count. It can be used to count a wide range of inputs and events. A counter may provide both up and down counts. Most counters will provide a reset input to reset the count to a known value, typically zero.

### Timer

A **timer** is a counter whose count is driven by a periodic signal. Timers are used to time many different types of system activities. In [Chapter 6](#), we will see how timers help with the operation of a real-time operating system.

### Real-time clock

A **real-time clock (RTC)** is a counter whose output corresponds to the clock time. The RTC typically keeps track of time relative to a fixed time standard. Software can then be used to convert the RTC value to human-readable time.

### General-purpose I/O

**General-purpose I/O (GPIO)** pins are just what the name implies; they can be used for a wide range of input or output functions. A GPIO pin can be configured as either input or output; it may also be disabled or put into a high-impedance mode. A more sophisticated GPIO interface may allow the pin to be configured to different logic levels. Software can write a value to the GPIO pin when it is in output mode, and the value is held until it is changed. When the pin is in input mode, the software can read the pin's value.

### Data conversion

**Data** conversion refers to the translation between analog and digital values. Two important characteristics of data conversion are as follows:

- the **rate** at which conversion can be performed;
- the **precision** in bits of the conversion; and
- the **accuracy** of the conversion.

### Analog-to-digital converter

An analog-to-digital converter (**ADC or A/D converter**) translates an analog value into a digital value. Many different techniques for ADC have been developed that provide a wide range of speeds, precisions, and costs. Generally, a high-speed, high-precision analog-to-digital conversion is expensive.

The next two examples compare ADCs intended for use in different applications.

---

### Example 4.1: Texas Instruments ADC12xJ1600-Q1

Texas Instruments ADC12xJ1600-Q1 [Tex20A] can be used for automotive radar systems. It provides a maximum analog/digital sample rate of 1.6 gigasamples per second (GSPS) at 12 bits of resolution. Its signal-to-noise ratio at 100 MHz is 57.4 dB. The full-scale input voltage is 800 mV. A JESD204C serial data interface is used to retrieve data.

---

---

### Example 4.2: Texas Instruments ADS126x

Texas Instruments ADS126x [Tex21] is useful in applications that require high accuracy: weight scales, thermocouples, strain-gauge sensors, and so on. It provides two levels of analog-to-digital conversion: a 32-bit precision converter and a 24-bit auxiliary converter. Both precision and auxiliary converters use sigma-delta architecture. The converter operates from 2.5 to 38.4K samples per second. It provides high accuracy: low offset and gain drift, low noise, and high linearity.

---

**Digital-to-analog converter**

A **digital-to-analog converter (DAC or D/A converter)** converts a digital value to an analog value that may be expressed as a voltage or current. Digital-to-analog conversion is conceptually more straightforward than the analog-to-digital case, but high-accuracy DACs are generally more expensive.

The next two examples compare digital-to-analog converters intended for different applications.

---

### Example 4.3: Texas Instruments AFE7422

Texas Instruments AFE7422 [Tex19] is designed for radio-frequency operations, such as phased array radar; it combines both DAC and ADC. It provides two 14-bit, 9 GSPS DACs and two 14-bit, 3 GSPS ADCs. It operates over a frequency range of 10 MHz to 6 GHZ. Both the transmit and receive signals paths provide configurable digital processing.

---

**Buses**

---

### Example 4.4: Texas Instruments DACx1001

Texas Instruments DACx1001 [Tex20B] provides high accuracy and low noise for instrumentation, such as semiconductor testing and medical radiology. Three versions are available in 20-, 18-, and 16-bit resolution. A four-wire serial interface provides digital access.

---

Directly connecting an I/O device to a high-speed CPU bus is not always a cost-effective solution. Lower-speed, lower-cost buses can be used to connect to simple, low-cost devices. The next example describes a simple bus used for audio data.

---

### Example 4.5: I<sup>2</sup>S Bus

The I<sup>2</sup>S bus [Phi96] is a standard interface for digital audio. An audio system may perform several processing steps on an audio signal using several different chips. I<sup>2</sup>S provides a standard, low-cost interface.

I<sup>2</sup>S is designed to carry stereo data with the two channels alternating. The I<sup>2</sup>S bus uses three signals:

- SCK is the system clock. A bus master provides the clock signal.
- WS is word select. The word select line determines which channel of audio is on the serial data line: low for left audio and high for right audio.

- Serial data provide the data encoded as two's complement with the most significant bit first. The standard does not define the number of bits in a sample; the end of a sample is signaled by a change in WS.

## 4.6 Designing with computing platforms

In this section, we concentrate on creating a working embedded system based on a computing platform. We first look at some example platforms and what they include. We then consider how to choose a platform for an application and how to make effective use of that platform.

### 4.6.1 Example platforms

The design complexity of the hardware platform can vary greatly, from a totally off-the-shelf solution to a highly customized design. A platform may comprise one chip or even dozens.

#### Open-source platforms

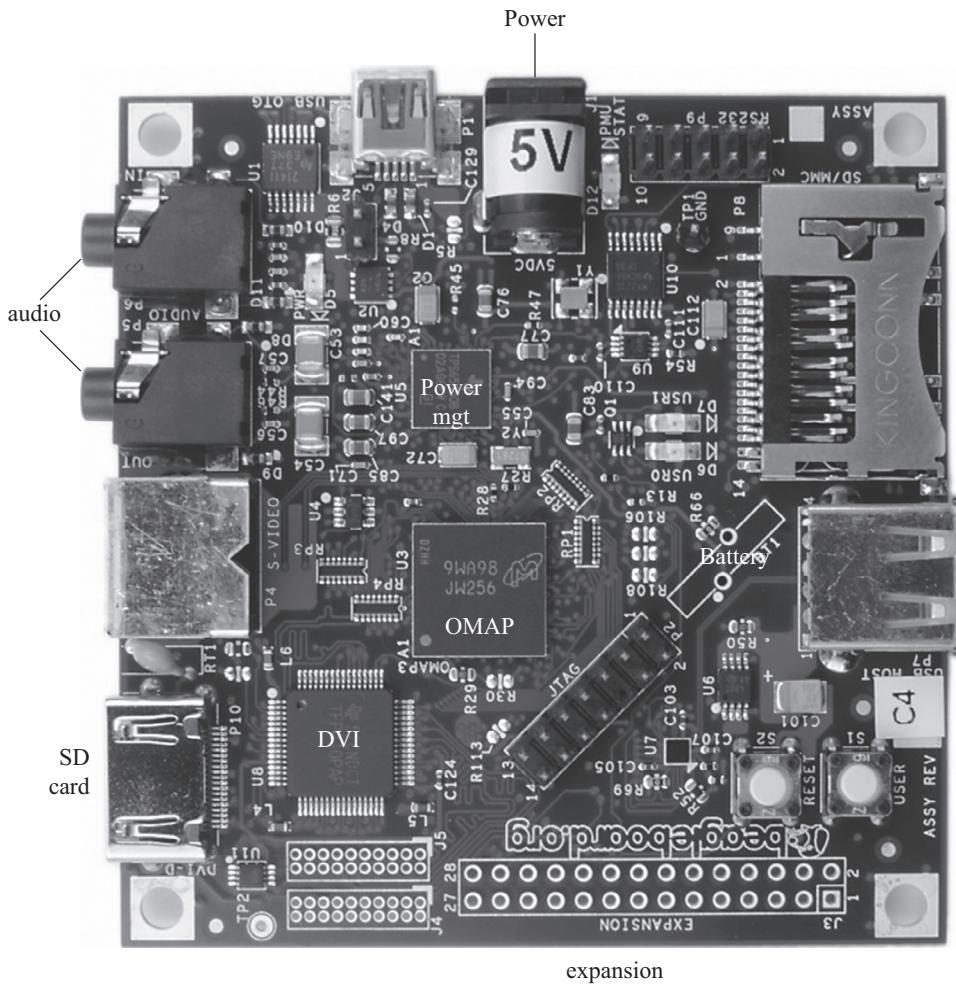
[Fig. 4.21](#) shows a BeagleBoard [Bea11], an open-source, low-cost platform for embedded system projects. The processor is an Arm Cortex-A8, which also comes with several built-in I/O devices. The board includes many connectors and supports a variety of I/O: flash memory, audio, video, and so forth. The support environment provides basic information about the board design, such as schematics, a variety of software development environments, and many example projects built with the BeagleBoard.

#### Developer and evaluation boards

Chip vendors often provide their own evaluation boards or modules for their chips. The evaluation board may be a complete solution, or it may provide what you need, with only slight modifications. The hardware design (e.g., netlist and board layout) is typically available from the vendor; companies provide this information to make it easy for customers to use their microprocessors. If the evaluation board does not completely meet your needs, you can modify the design using the netlist and board layout without starting from scratch. Vendors generally do not charge royalties for hardware board design.

[Fig. 4.22](#) shows a Jetson Nano developer kit [Fra19]. The processor includes a quad-core Arm core, a 128-core NVIDIA Maxwell GPU, and video encoder and decoder. I/O includes four USB 3.0 ports, a MIPI camera port, an HDMI port, and Gigabit Ethernet. The system boots off a microSD card. A Linux-based development system can be run on the board.

[Fig. 4.23](#) shows an Arm evaluation module. Like the BeagleBoard, this evaluation module includes a basic platform chip and a variety of I/O devices. However, the main purpose of BeagleBoard is as an end-use, low-cost board, and the evaluation module is primarily intended to support software development and serve as a starting point for a more refined product design. As a result, this evaluation module includes some features that would not appear in the final product, such as the connections to the processor's pins that surround the processor chip itself.

**FIGURE 4.21**

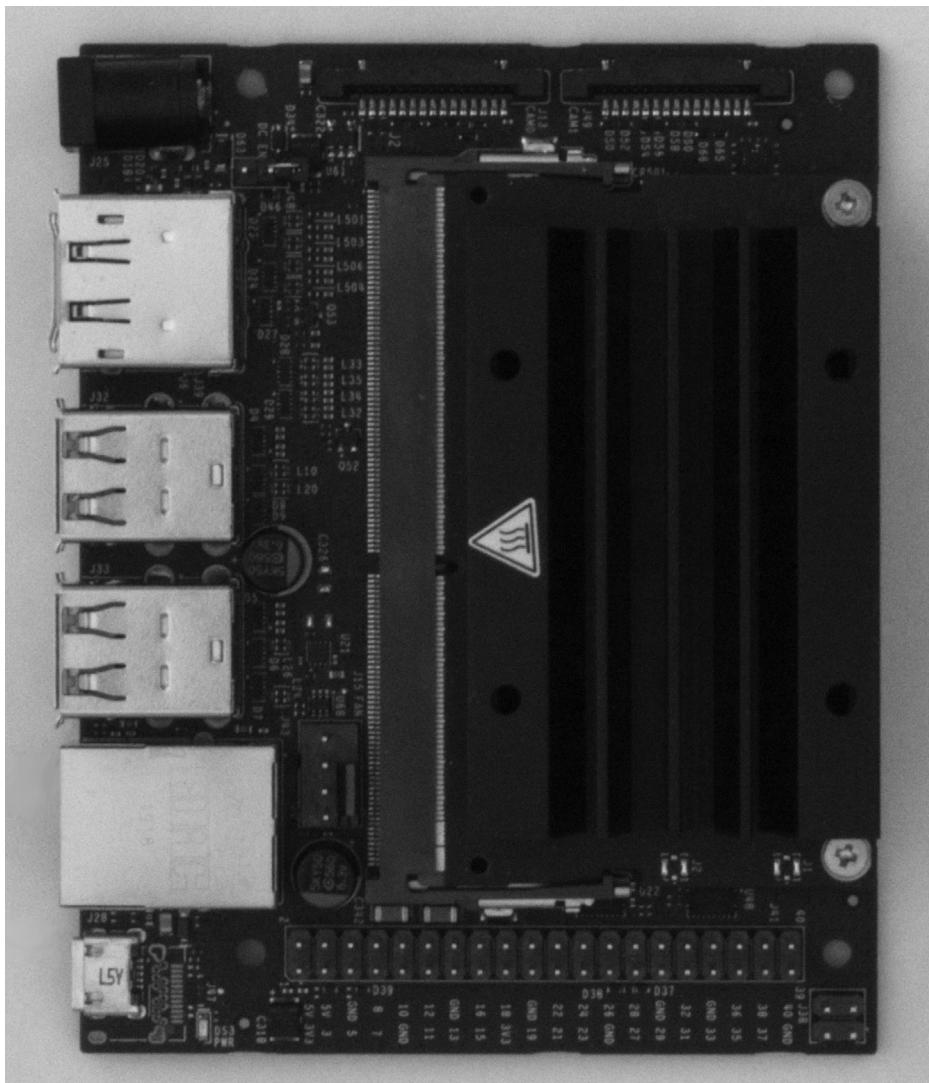
A BeagleBoard.

### 4.6.2 Choosing a platform

We will probably not design the platform for our embedded system from scratch. We may assemble hardware and software components from several sources, and we may also acquire a complete hardware/software platform package. Several factors will contribute to your decision to use a particular platform.

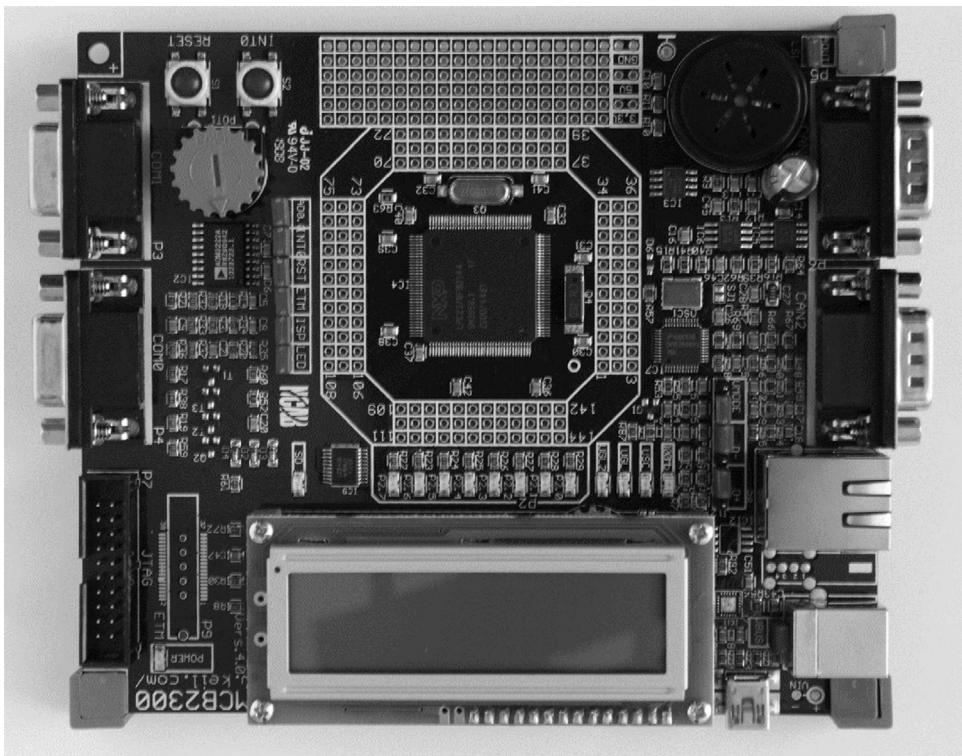
#### Hardware

The hardware architecture of the platform is the more obvious manifestation of the architecture, because you can touch it and feel it. The various components may all play a role in the suitability of the platform.

**FIGURE 4.22**

Jetson Nano Developer Kit.

- **CPU:** An embedded computing system clearly contains a microprocessor. But which one? There are many different architectures, and even within an architecture, we can select from among models that vary in clock speed, bus data width, integrated peripherals, and so on. The choice of CPU is one of the most important decisions, but it cannot be made without considering the software that will be executed on the machine.

**FIGURE 4.23**

An Arm evaluation module.

- *Bus:* The choice of a bus is closely tied to that of a CPU because the bus is an integral part of the microprocessor. However, in applications that make intensive use of the bus owing to I/O or other data traffic, the bus may be more of a limiting factor than the CPU. Attention must be paid to the required data bandwidths to ensure that the bus can handle the traffic.
- *Memory:* Once again, the question is not whether the system will have memory, but the characteristics of that memory. The most obvious characteristic is the total size, which depends on both the required data volume and the size of the program instructions. The ratio of ROM to RAM and the selection of DRAM versus SRAM can have a significant influence on the cost of the system. The speed of the memory plays a large part in determining system performance.
- *Input and output devices:* If we use a platform built out of many low-level components on a printed circuit board, we may have a great deal of freedom in the I/O devices connected to the system. Platforms based on highly integrated chips only come with certain combinations of I/O devices. The combination of I/O devices

**Software**

may be a prime factor in platform selection. We may need to choose a platform that includes some I/O devices we do not need to get the devices that we do need.

When we think about the software components of the platform, we generally think about both the run-time components and the support components. Run-time components become part of the final system: the operating system, code libraries, and so on. Support components include the code development environment, debugging tools, and so on.

Run-time components are a critical part of the platform. An operating system is required to control the CPU and its multiple processes. A file system is used in many embedded systems to organize internal data and as an interface with other systems. Many complex libraries—digital filtering and fast Fourier transform—provide highly optimized versions of complex functions.

Support components are critical to making use of complex hardware platforms. Without proper code development and operating systems, the hardware itself is useless. Tools may come directly from the hardware vendor, from third-party vendors, or from developer communities.

### 4.6.3 Intellectual property

Intellectual property (IP) is something that we can own, but not touch: software, netlists, and so on. Just as we need to acquire hardware components to build our system, we also need to acquire IP to make that hardware useful. Here are some examples of the wide range of IP that we use in embedded system design:

- run-time software libraries;
- software development environments; and
- schematics, netlists, and other hardware design information.

IP can come from many different sources. We may buy IP components from vendors. For example, we may buy a software library to perform certain complex functions and incorporate that code into our system. We may also obtain it from developer communities online.

Example 4.6 looks at the IP available for the BeagleBoard.

---

#### Example 4.6: BeagleBoard IP

The BeagleBoard Web site (<http://www.beagleboard.org>) contains both hardware and software IP. The hardware IP includes:

- schematics for the printed circuit board;
- artwork files (known as Gerber files) for the printed circuit board; and
- a bill of materials that lists the required components.

Software IP includes:

- a compiler for the processor
  - a version of Linux for the processor
-

#### 4.6.4 Development environments

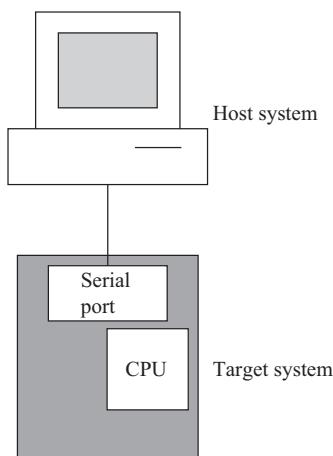
Although we may use an evaluation board, much of the software development for an embedded system is done on a PC or workstation known as the **host**, as illustrated in Fig. 4.24. The hardware upon which the code will finally run is known as the **target**. The host and target are frequently connected by a USB link, but a higher-speed link, such as Ethernet, can be used.

The target must include a small amount of software to talk to the host system. That software will take up some memory, interrupt vectors, and so on, but it should generally leave the smallest possible footprint in the target to avoid interfering with the application software. The host should be able to do the following:

- load programs into the target;
- start and stop program execution on the target; and
- examine memory and CPU registers.

A **cross-compiler** runs on one type of machine but generates code for another. After compilation, the executable code is typically downloaded to the embedded system via USB. We also often make use of host-target debuggers, in which the basic hooks for debugging are provided by the target, and a more sophisticated user interface is created by the host.

We often create a **testbench program** that can be built to help debug the embedded code. The testbench generates inputs to stimulate a piece of code and compares the outputs against expected values, providing valuable early debugging help. The embedded code may need to be slightly modified to work with the testbench, but careful coding, such as using the `#ifdef` directive in C, can ensure that the changes can be undone easily and without introducing bugs.



**FIGURE 4.24**

Connecting a host and target system.

### 4.6.5 Watchdog timers

The **watchdog** is a useful technique for monitoring the system during operation. Watchdogs, when used properly, can help improve the system's safety and security.

The most basic form of a watchdog is a watchdog timer [Koo10]. This technique uses a timer to monitor the correct operation of the software. As shown in Fig. 4.25, the timer's rollover output (set to high when the count reaches zero) is connected to the system reset. The software is modified so that it reinitializes the counter, setting it back to its maximum count. The software must be modified so that every execution path reinitializes the timer frequently, such that the counter never rolls over and resets the system. The timer's period determines how frequently the software must be designed to reinitialize the timer. A reset during operation indicates some type of software error or fault.

More generally, a watchdog processor monitors the operation of the main processor [Mah88]. For example, a watchdog processor can be programmed to monitor the control flow of a program executed on the main processor [Lu82].

Example 4.7 discusses the use of a watchdog timer on the Ingenuity Mars helicopter.

---

#### Example 4.7: Watchdog Timer Expiration on Ingenuity Mars Helicopter

The expiration of a watchdog timer on the Ingenuity Mars helicopter caused a delay in its first flight [NAS21A]. The watchdog expired during a high-speed spin test of the rotors on Sol 49 (April 9, 2021). The flight computer was in the midst of a transition from preflight to flight mode. This issue was addressed with a software update to modify the flight controller boot process [NAS21B].

---

### 4.6.6 Debugging techniques

A good deal of software debugging can be done by compiling and executing the code on a PC or workstation. However, at some point, running code on the embedded hardware platform is necessary. Embedded systems are usually less friendly programming environments than PCs. Nonetheless, the resourceful designer has several options available for debugging the system.

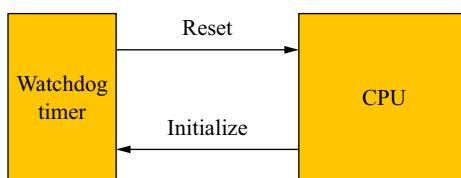


FIGURE 4.25

A watchdog timer in a system.

The USB port found on most evaluation boards is one of the most important debugging tools. In fact, it is often a good idea to design a USB port into an embedded system even if it will not be used in the final product; USB can be used not only for development debugging, but also for diagnosing problems in the field or field upgrades of software.

Another important debugging tool is the **breakpoint**. The simplest form of a breakpoint is for the user to specify an address at which the program's execution is to break. When the PC reaches that address, the control is returned to the monitor program. From the monitor program, the user can examine and/or modify the CPU registers, after which execution can be continued. Implementing breakpoints does not require using exceptions or external devices.

Programming Example 4.1 shows how to use instructions to create breakpoints.

---

### Programming Example 4.1: Breakpoints

A breakpoint is a location in the memory at which a program stops executing and returns to the debugging tool or monitor program. Implementing breakpoints is very simple; you simply replace the instruction at the breakpoint location with a subroutine call to the monitor. In the following code, to establish a breakpoint at location 0x40c in some Arm code, we've replaced the branch (B) instruction normally held at that location with a subroutine call (BL) to the breakpoint handling routine:

When the breakpoint handler is called, it saves all registers, and can then display the CPU state to the user and take commands.

To continue execution, the original instruction must be replaced in the program. If the breakpoint can be erased, the original instruction can simply be replaced, and control is returned to that instruction. This will normally require fixing the subroutine return address, which will point to the instruction after the breakpoint. If the breakpoint is to remain, then the original instruction can be replaced, and a new temporary breakpoint is placed at the next instruction (taking jumps into account, of course). When the temporary breakpoint is reached, the monitor puts back the original breakpoint, removes the temporary one, and resumes execution.

The Unix *dbx* debugger shows the program being debugged in source code form, but that capability is too complex to fit into most embedded systems. Very simple monitors will require you to specify the breakpoint as an absolute address, which will require you to know how the program was linked. A more sophisticated monitor will read the symbol table and allow you to use labels in the assembly code to specify locations.

---

#### LEDs as debugging devices

Never underestimate the importance of light-emitting diodes (LEDs) in debugging. As with serial ports, it is often a good idea to design in a few to indicate the system state, even if they will not normally be seen in use. LEDs can be used to show error conditions, when the code enters certain routines, or to show idle-time activity. LEDs can be entertaining as well; a simple flashing LED can provide a great sense of accomplishment when it first starts to work.

#### In-circuit emulation

When software tools are insufficient to debug the system, hardware aids can be deployed to give a clearer view of what is happening when the system is running.

The **microprocessor in-circuit emulator (ICE)** is a specialized hardware tool that can help to debug software in a working embedded system. At the heart of an ICE is a special version of the microprocessor that allows its internal registers to be read out when they are stopped. The ICE surrounds this specialized microprocessor with additional logic that allows the user to specify breakpoints and examine and modify the CPU state. The CPU provides as much debugging functionality as a debugger within a monitor program, but it does not take up any memory. The main drawback of in-circuit emulation is that the machine is specific to a particular microprocessor, even down to the pinout. If you use several microprocessors, maintaining a fleet of ICES to match can be exorbitant.

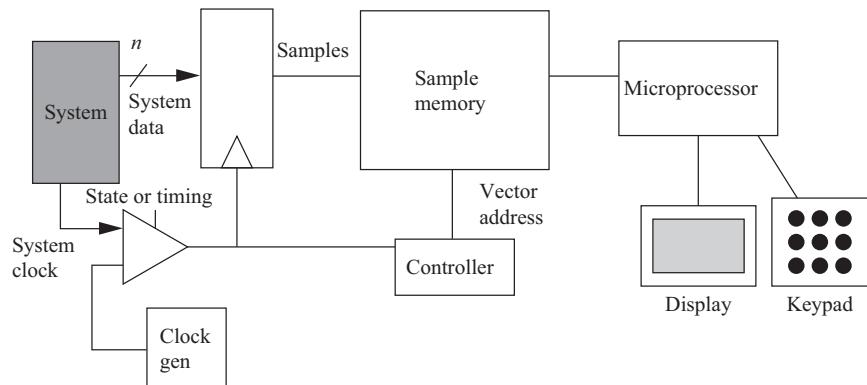
### Logic analyzers

The **logic analyzer** [Ald73] is the other major piece of instrumentation in the embedded system designer's arsenal. Think of a logic analyzer as an array of inexpensive oscilloscopes; the analyzer can sample many different signals simultaneously (tens to hundreds), but can display only 0, 1, or changing values for each. All these logic analysis channels can be connected to the system to record the activity on many signals simultaneously. The logic analyzer records the values of the signals into an internal memory, and then, displays the results on a display once the memory is full or the run is aborted. The logic analyzer can capture thousands or even millions of samples of data on all of these channels, providing a much larger time window into the operation of the machine than is possible with a conventional oscilloscope.

A typical logic analyzer can acquire data in either of two modes, which are typically called **state** and **timing modes**. To understand why the two modes are useful and the difference between them, it is important to remember that a logic analyzer trades reduced resolution on the signals for the longer time window. The measurement resolution of each signal is reduced in both voltage and time dimensions. The reduced voltage resolution is accomplished by measuring logic values (0, 1, x) rather than analog voltages. The reduction in timing resolution is accomplished by sampling the signal, rather than capturing a continuous waveform, as in an analog oscilloscope.

The state and timing modes represent different ways of sampling the values. Timing mode uses an internal clock that is fast enough to take several samples per clock period in a typical system. State mode, on the other hand, uses the system's own clock to control sampling; thus, it samples each signal only once per clock cycle. As a result, the timing mode requires more memory to store a given number of system clock cycles. On the other hand, it provides a greater resolution in the signal for detecting glitches. The timing mode is typically used for glitch-oriented debugging, whereas the state mode is used for sequentially oriented problems.

The internal architecture of a logic analyzer is shown in Fig. 4.26. The system's data signals are sampled at a latch within the logic analyzer; the latch is controlled by either the system clock or the internal logic analyzer sampling clock, depending on whether the analyzer is being used in the state or timing mode. Each sample is copied into a vector memory under the control of a state machine. The latch, timing circuitry, sample memory, and controller must be designed to run at high speed because several samples per system clock cycle may be required in timing mode.

**FIGURE 4.26**


---

Architecture of a logic analyzer.

After the sampling is complete, an embedded microprocessor takes over to control the display of the data captured in the sample memory.

Logic analyzers typically provide several formats for viewing data. One format is a timing diagram. Many logic analyzers allow not only customized displays, such as giving names to signals, but also more advanced display options. For example, an inverse assembler can be used to turn vector values into microprocessor instructions. The logic analyzer does not provide access to the internal state of the components, but it does give a particularly good view of externally visible signals. This information can be used for both functional and timing debugging.

Some modern logic analyzers are structured as USB devices; the device includes the data acquisition circuitry, while the host computer serves as the user interface.

#### 4.6.7 Debugging challenges

Logical errors in software can be hard to track, but errors in real-time code can create problems that are even harder to diagnose. Real-time programs are required to finish their work within a certain amount of time; if they run too long, they can create unexpected behavior.

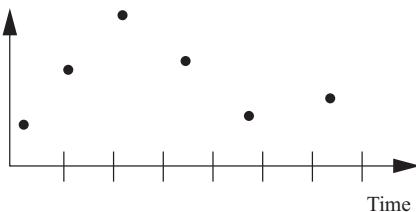
Example 4.8 demonstrates one of the problems that can arise.

---

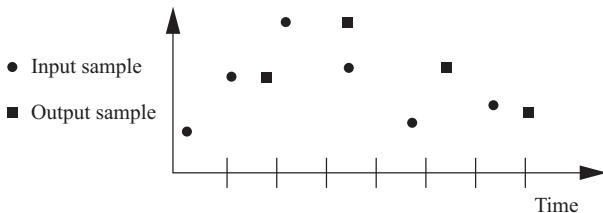
#### Example 4.8: A Timing Error in Real-time Code

Let's consider a simple program that periodically takes an input from an analog/digital converter, does some computations on it, and then, outputs the result to a digital/analog converter. To make it easier to compare input to output and see the results of the bug, we assume that the computation produces an output equal to the input but that a bug causes the computation to

run 50% longer than its given time interval. A sample input to the program over several sample periods looks like this:



If the program ran fast enough to meet its deadline, the output would simply be a time-shifted copy of the input. However, when the program runs over its allotted time, the output will become quite different. Exactly what happens depends, in part, on the behavior of the A/D and D/A converters. Therefore, let's make some assumptions. First, the A/D converter holds its current sample in a register until the next sample period, and the D/A converter changes its output whenever it receives a new sample. Next, a reasonable assumption about interrupt systems is that, when an interrupt is not satisfied and the device interrupts again, the device's old value will disappear and be replaced by the new value. The basic situation that develops when the interrupt routine runs too long is something like this:



1. The A/D converter is prompted by the timer to generate a new value, saves it in the register, and requests an interrupt.
2. The interrupt handler runs too long from the last sample.
3. The A/D converter gets another sample in the next period.
4. The interrupt handler finishes its first request, and then, immediately responds to the second interrupt. It never sees the first sample and only gets the second one.

Thus, assuming that the interrupt handler takes 1.5 times longer than it should, here is how it would process the sample input:

The output waveform is seriously distorted because the interrupt routine grabs the wrong samples and puts the results out at the wrong times.

The exact results of missing real-time deadlines depend on the detailed characteristics of the I/O devices and the nature of the timing violation. This makes debugging real-time problems especially difficult. Unfortunately, the best advice is that, if a system exhibits truly unusual behavior, missed deadlines should be suspected. ICES, logic analyzers, and even LEDs can be useful tools for checking the execution time of real-time code to determine whether it in fact meets its deadline.

## 4.7 Embedded file systems

### Flash memory

Many consumer electronics devices use **flash memory** for mass storage. Flash memory is a type of semiconductor memory that, unlike DRAM and SRAM, provides permanent storage. Values are stored in the flash memory cell as an electric charge using a specialized capacitor that can store the charge for years. Flash memory cell does not require an external power supply to maintain its value. Furthermore, the memory can be written electrically, and unlike previous generations of electrically erasable semiconductor memory, can be written using standard power supply voltages, and thus, does not need to be disconnected during programming.

### SSDs

A solid-state drive (SSD) is a storage device built from flash memory or another form of nonvolatile solid-state device. An SSD is an I/O device with its own controller, which acts as an interface between the drive storage and the rest of the system.

### Flash file systems

A flash memory has one important limitation that must be considered. Writing a flash memory cell causes mechanical stress that eventually wears out the cell. Today's flash memories can reliably be written a million times, but will, at some point, fail. Although a million write cycles may sound like a lot, creating a single file may require many write operations, particularly for the part of the memory that stores the directory information.

A wear-leveling flash file system [Ban95] manages the use of flash memory locations to equalize wear, while maintaining compatibility with existing file systems. A simple model of a standard file system has two layers: the bottom layer handles physical reads and writes on the storage device, and the top layer provides a logical view of the file system. A flash file system imposes an intermediate layer that allows the logical-to-physical mapping of files to be changed. This layer keeps track of how frequently different sections of the flash memory have been written and allocates data to equalize wear. It may also move the location of the directory structure while the file system is operating. Because the directory system receives the most wear, keeping it in one place may cause part of the memory to wear out before the rest, unnecessarily reducing the useful life of the memory device. Several flash file systems have been developed, such as the Yet Another Flash Filing System (YAFFS) [Yaf11].

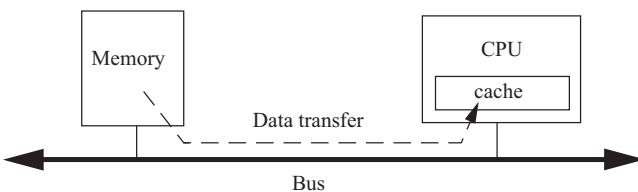
### DOS file systems

DOS file allocation table (**FAT**) file systems refer to the file system developed by Microsoft for early versions of the DOS [Mic00]. FAT32 file systems can interoperate with a wide range of operating systems and can be implemented in a relatively small amount of code. Wear-leveling algorithms for flash memory can be implemented without disturbing the basic operation of the file system.

---

## 4.8 Platform-level performance analysis

Bus-based systems add another layer of complications to performance analysis. Platform-level performance involves much more than the CPU. We often focus on the CPU because it processes instructions, but any part of the system can affect the

**FIGURE 4.27**

Platform-level data flows and performance.

total system performance. More precisely, the CPU provides an upper bound on performance, but any other part of the system can slow down the CPU. Merely counting instruction execution times is not enough.

Consider the simple system shown in Fig. 4.27. We want to move data from the memory to the CPU to process it. To get the data from the memory to the CPU, we must:

- read from the memory;
- transfer over the bus to the cache; and
- transfer from the cache to the CPU.

The time required to transfer from the cache to the CPU is included in the instruction execution time, but the other two times are not included.

The most basic measure of performance we are interested in is **bandwidth**—the rate at which we can move data. Ultimately, if we are interested in real-time performance, we are interested in real-time performance measured in seconds. However, often, the simplest way to measure performance is in units of clock cycles. However, different parts of the system run at different clock rates. We must ensure that we apply the right clock rate to each part of the performance estimate when we convert from clock cycles to seconds.

Bandwidth questions often arise when we are transferring large blocks of data. For simplicity, let's start by considering the bandwidth provided by only one system component: the bus. Consider an image of  $1920 \times 1080$  pixels with each pixel composed of 3 bytes of data. This gives a grand total of 6.2 megabytes. If these images are video frames, we want to check whether we can push one frame through the system within the 0.033 s that we have to process a frame before the next one arrives.

Let us assume that we can transfer 1 byte of data every microsecond, which implies a bus speed of 100 MHz. In this case, we would require 0.062 s to transfer one frame, or about half the rate required.

We can increase bandwidth in two ways: we can increase the clock rate of the bus, or we can increase the amount of data transferred per clock cycle. For example, if we increased the bus to carry 4 bytes or 32 bits per transfer, we would reduce the transfer time to 0.015 s at the original 100 MHz clock rate. Alternatively, if we could increase

**Bandwidth as performance**

**Bus bandwidth**

**Bus bandwidth characteristics****Bus bandwidth formulas**

the bus clock rate to 200 MHz, then we would reduce the transfer time to 0.031 s, which is within our time budget for the transfer.

How do we know how long it takes to transfer one unit of data? To determine this, we must look at the data sheet for the bus. A bus transfer generally takes more than one clock cycle. Burst transfers, which move blocks of data to contiguous locations, may be more efficient per byte. We also need to know the width of the bus—how many bytes per transfer. Finally, we need to know the bus clock period, which will generally be different from the CPU clock period.

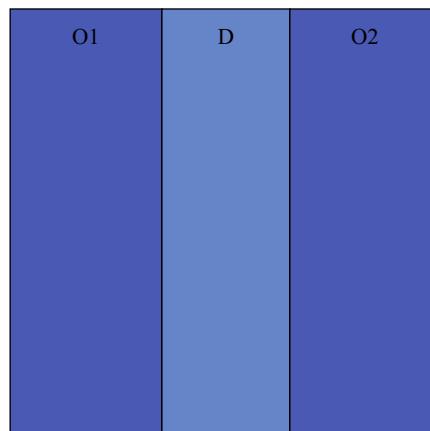
We can write formulas that tell us how long a transfer of  $N$  words will take, given a bus with a transfer width of one word. We will write our basic formulas in units of bus cycles,  $T$ , and then, convert those bus cycle counts to real time  $t$  using the bus clock period,  $P$ :

$$t = TP \quad (\text{Eq. 4.1})$$

First, consider transferring one word at a time with nonburst bus transactions. As shown in Fig. 4.28, a basic bus transfer of  $N$  bytes transfers one word per bus transaction. A single transfer takes  $D$  clock cycles. Ideally,  $D = 1$ , but a memory that introduces wait states is one example of a transfer that could require  $D > 1$  cycles. Addresses, handshaking, and other activities constitute overhead that may occur before ( $O_1$ ) or after ( $O_2$ ) the data. For simplicity, we lump the overhead into  $O = O_1 + O_2$ . This gives a total transfer time in clock cycles of

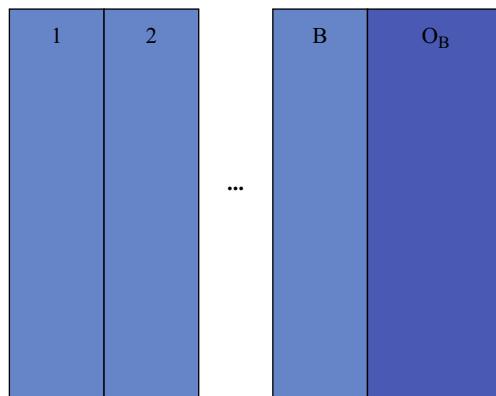
$$T_{\text{basic}}(N) = (O + D)N \quad (\text{Eq. 4.2})$$

Now consider a burst transaction with a burst transaction length of  $B$  words, as shown in Fig. 4.29. As before, each of those transfers will require  $D$  clock cycles, including any wait states. The bus also introduces  $O_B$  cycles of overhead per burst. This gives



**FIGURE 4.28**

Times and data volumes in a basic bus transfer.

**FIGURE 4.29**

Times and data volumes in a burst bus transfer.

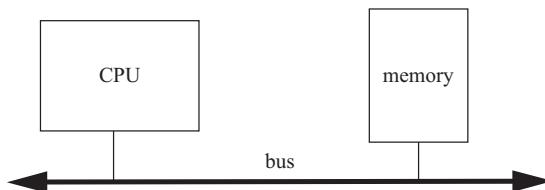
$$T_{burst}(N) = \frac{N}{B} (BD + O_B) \quad (\text{Eq. 4.3})$$

The next example applies our bus performance models to a simple example.

---

### Example 4.9: Performance Bottlenecks in a Bus-based System

Consider a simple bus-based system.



We want to transfer data between the CPU and the memory over the bus. We need to be able to read an HDTV  $1920 \times 1080$ , 3 bytes per pixel video frame into the CPU at the rate of 30 frames/s, for a total of 6.2 MB/s. Which will be the bottleneck and limit system performance: the bus or the memory?

Let's assume that the bus has a 100 MHz clock rate (period of  $10^{-8}$  s) and is two bytes wide, with  $D = 1$  and  $O = 3$ . The 2-B bus allows us to cut the number of bus operations in half. This gives a total transfer time of

$$T_{basic}(1920 \times 1080) = (3 + 1) \cdot \left( \frac{6.2 \times 10^6}{2} \right) = 12.4 \times 10^6 \text{ cycles}$$

$$t_{basic} = T_{basic} P = 0.124 \text{ s}$$

Because the total time to transfer 1-s worth of frames is more than 1 s, the bus is not fast enough for our application.

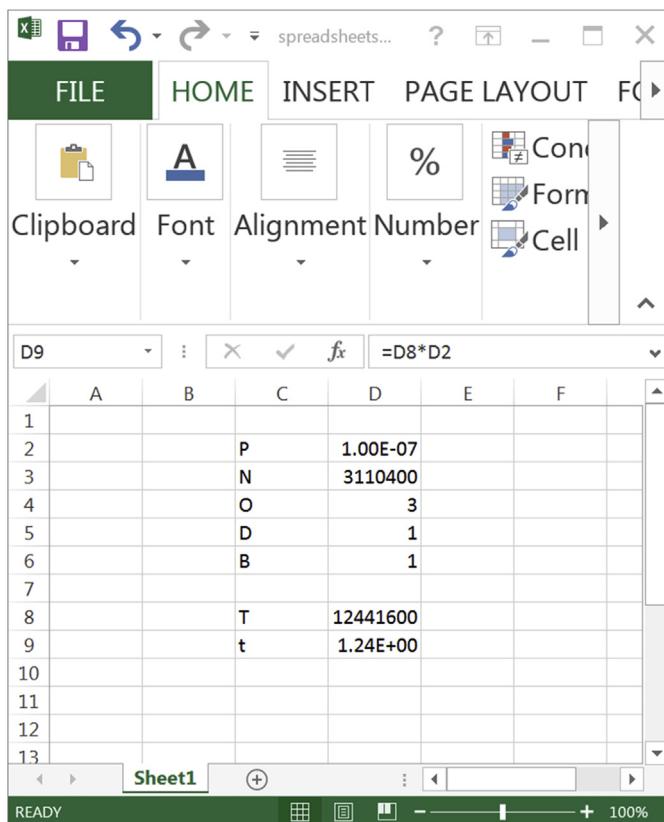
Alternatively, consider a memory that provides a burst mode with  $B = 4$  and is two bytes wide. For this memory,  $D = 1$  and  $O = 4$ , and assume that it runs at the same clock speed. The access time for this memory is 10 ns. Then,

$$T_{basic}(1920 \times 1080) = \frac{(6.2 \times 10^6 / 2)}{4} (4 \cdot 1 + 4) = 6.2 \times 10^6 \text{ cycles}$$

$$t_{basic} = T_{basic}P = 0.062s$$

The memory is fast enough—it requires less than 1 s to transfer the 30 frames that must be transmitted in 1 s.

One way to explore design trade-offs is to build a spreadsheet with our bandwidth formulas.



We can change values, such as bus width and clock rate, and instantly see their effects on available bandwidth.

### Component bandwidth

Bandwidth questions also come up in situations that we don't normally think of as communications. Transferring data into and out of components also raises questions of bandwidth. The simplest illustration of this problem is memory.

The width of a memory determines the number of bits that we can read from the memory in one cycle. This is a form of data bandwidth. We can change the types of memory components we use to change the memory bandwidth; we may also be able to change the format of our data to accommodate the memory components.

#### Memory aspect ratio

A single memory chip is not solely specified by the number of bits it can hold. As shown in Fig. 4.30, memories of the same size can have different **aspect ratios**. For example, a 1-Gbit memory that is 1-bit wide will use 30 address lines to present  $2^{30}$  locations, each with 1-b of data. The same-size memory in a 4-bit-wide format will have 26 address lines, and an 8-bit-wide memory will have 22 address lines.

Memory chips do not come in extremely wide aspect ratios, but we can build wider memories by using several memories in parallel. By organizing memory chips into the proper aspect ratio for our application, we can build a memory system with the total amount of storage that we want, which presents the data width that we need.

The memory system width may also be determined by the memory modules we use. Rather than buying memory chips individually, we may buy memory as SIMMs or DIMMs. These memories are wide, but generally only come in standard widths.

Which aspect ratio is preferable for the overall memory system depends, in part, on the format of the data that we want to store in the memory and the speed with which it must be accessed, giving rise to bandwidth analysis.

We must also consider the time required to read or write a memory. Once again, we refer to the component data sheets to find these values. Access times depend quite a bit on the type of memory chip used. Page modes operate similarly to burst modes in buses. If the memory is not synchronous, we can still refer the times between events back to the bus clock cycle to determine the number of clock cycles required for access.

#### Memory access times and bandwidth

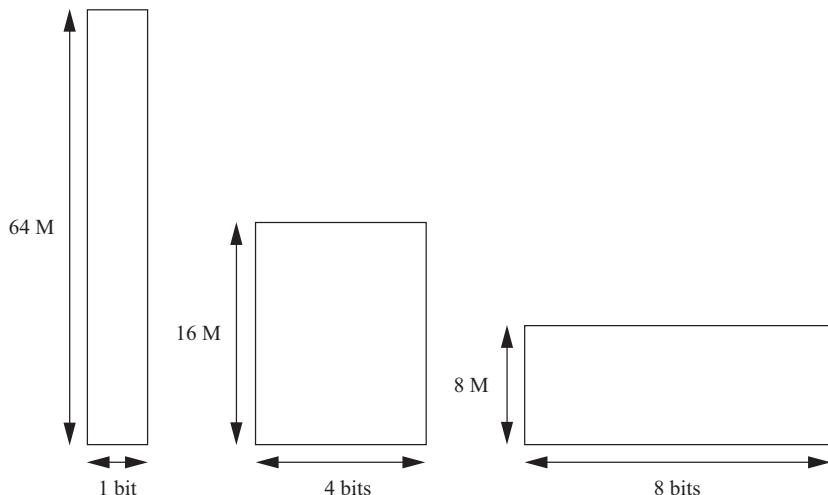


FIGURE 4.30

Memory aspect ratios.

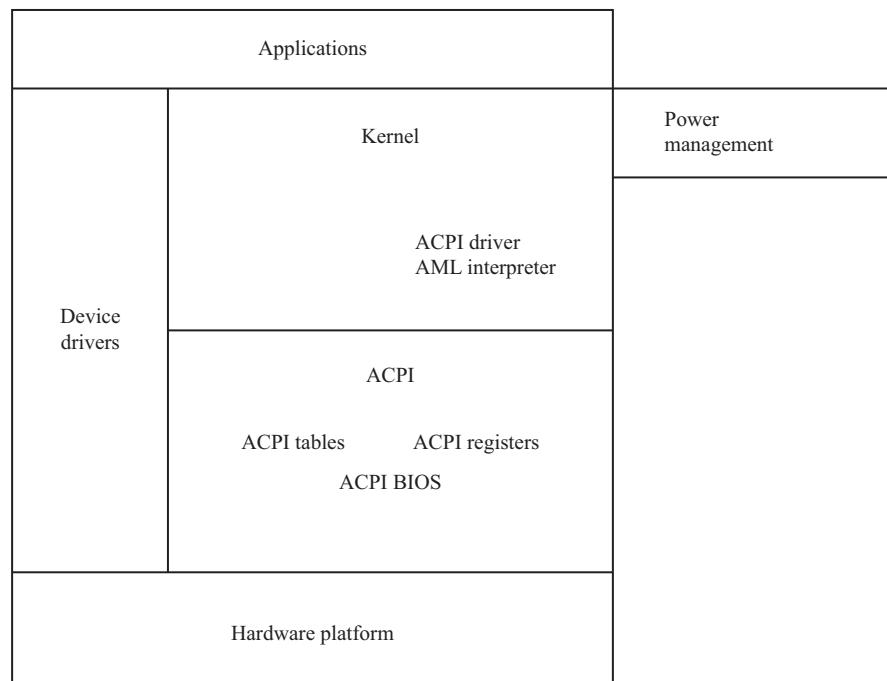
**ACPI**

## 4.9 Platform-level power management

The **Advanced Configuration and Power Interface (ACPI)** [ACP13] is an open-industry standard for power management services. Initially targeted at PCs, it is designed to be compatible with a wide variety of operating systems. The role of ACPI in the system is illustrated in Fig. 4.31. The ACPI provides some basic power management facilities and abstracts the hardware layer. The operating system has its own power management module that determines the policy, and the operating system then uses ACPI to send the required controls to the hardware and to observe the hardware's state as input to the power manager.

The ACPI supports several basic global power states:

- G3, the mechanical off state, in which the system consumes no power;
- G2, the soft off state, which requires a full operating system reboot to restore the machine to working conditions;
- G1, the sleeping state, in which the system appears to be off, and the time required to return to working conditions is inversely proportional to power consumption;
- G0, the working state, in which the system is fully usable;



**FIGURE 4.31**

The Advanced Configuration and Power Interface and its relationship to a complete system.

- S4, a nonvolatile sleep, in which the system state is written to nonvolatile memory for later restoration; and
- the legacy state, in which the system does not follow the ACPI.

The power manager typically includes an observer, who receives messages through the ACPI that describe the system behavior. It also includes a decision module that determines power management actions based on these observations.

## 4.10 Platform security

Security and safety cannot be bolted on; they must be baked in. A basic understanding of security issues is important for every embedded system designer. This section discusses some basic security techniques and their use in the computing platform. We discuss programs and security in [Section 5.11](#).

### Cryptography

#### Secret key cryptography

We make use of a few basic concepts in cryptography throughout the book [Sch96]: cryptography, public-key cryptography, hashing, and digital signatures.

Cryptography is designed to encode a message so that it cannot be read directly by someone who intercepts the message; the code should also be difficult to break. Traditional techniques are known as **secret key cryptography** because they rely on the secrecy of the key used to encrypt the messages. The advanced encryption standard (AES) is a widely used encryption algorithm [ISO10]. It encrypts data in blocks of 128 bits and can use keys of three different sizes: 128, 192, or 256 bits. The SIMON block cipher [Bea13] was developed as a lightweight cipher. It operates on blocks of several sizes ranging from 32 to 128 bits and with keys ranging from 64 to 256 bits.

**Public-key cryptography** splits the key into two pieces: a private key and a public key. The two are related such that a message encrypted with the private key can be decrypted using the public key, but the private key cannot be inferred from the public key. Because the public key does not disclose information about how the message is encoded, it can be kept in a public place for anyone to use. The Rivest–Shamir–Adleman (RSA) algorithm is one widely used public-key algorithm.

A **cryptographic hash function** has a somewhat different purpose. It is used to generate a **message digest** from a message. The message digest is generally shorter than the message itself and doesn't directly reveal the message contents. The hash function is designed to minimize *collisions* so that two different messages should be very unlikely to generate the same message digest. As a result, the hash function can be used to generate short versions of more lengthy messages. SHA-3 [Dwo15] is the latest in a series of SHA standards.

We can use a combination of public-key cryptography and hash functions to create a **digital signature**, a message that authenticates a message coming from a particular sender. The sender uses her own private key to sign either the message itself or the message digest. The receiver of the message then uses the sender's public key to decrypt the signed message. A digital signature ensures the identity of the person who encrypts the message; the signature is unforgeable and unalterable. We can

### Hash functions

### Digital signatures

also combine digital signatures with message encryption. In this case, both the sender and receiver have private and public keys. The sender first signs the message with her private key, and then, encrypts the signed message with the *receiver's* public key. Upon receipt, the receiver first decrypts using her private key, and then, verifies the signature using the *sender's* public key.

Cryptographic functions may be implemented in software or hardware. The next example describes an embedded processor with hardware security accelerators.

---

### Application Example 1.1: Texas Instruments TM4C129x Microcontroller

Texas Instruments TM4C [Tex14] is designed for applications that require floating-point computation and low-power performance, such as building automation, security and access control, and data acquisition. The CPU is an Arm Cortex-M4. It includes accelerators for AES, data encryption standard, SHA, MD5 (message digest), and cyclic redundancy check (CRC).

---

#### Attestation

Signed code allows the platform to know that the code comes from a trusted source. A computing platform often interacts with other platforms, either over networks or over other types of links. When Alice talks to Bill, Bill needs to know that Alice has not been compromised before sharing secure information with Alice. **Attestation** [Cok11] provides a mechanism for Alice to demonstrate integrity to Bill and for Bill to **appraise** Alice's security. For example, Alice may generate a hash of certain important, predefined pieces of code or portions of memory, sign the hash, and send it to Bill. Because those hashed elements are known to Bill, Bill expects a certain set of hash codes. If the hash codes differ, Bill should assume that Alice has been compromised. If the received codes match the expected codes, Alice can be judged to be secure with high confidence.

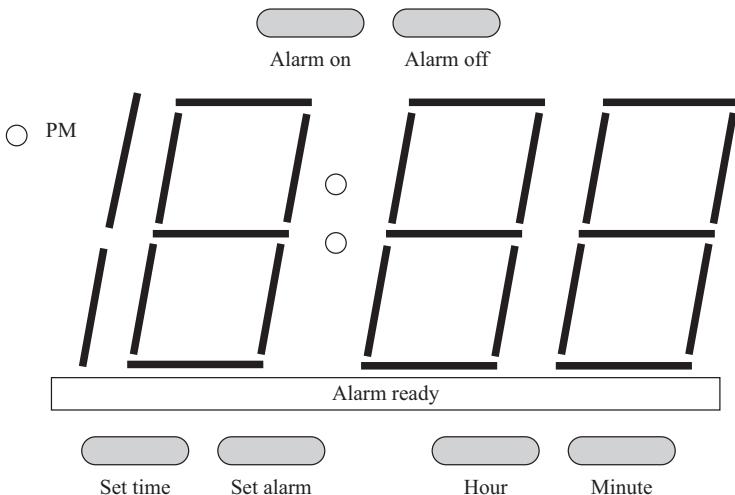
---

## 4.11 Design example: alarm clock

Our first system design example is an alarm clock. We use a microprocessor to read the clock's buttons and update the time display. Because we now better understand I/O, we work through the steps of the methodology to go from a concept to a completed and tested system.

### 4.11.1 Requirements

The basic functions of an alarm clock are well understood and easy to enumerate. Fig. 4.32 illustrates the front panel design for the alarm clock. The time is shown as four digits in 12-h format; we use light to distinguish between AM and PM. We use several buttons to set the clock and alarm times. When we press the *hour* and *minute* buttons, we advance the hour and minute each by one. When setting the time, we must hold down the *set time* button while we hit the *hour* and *minute* buttons; the

**FIGURE 4.32**

Front panel of the alarm clock.

*set alarm* button works in a similar fashion. We turn the alarm on and off with the *alarm on* and *alarm off* buttons. When the alarm is activated, the *alarm ready* light is on. A separate speaker provides the audible alarm.

We are now ready to create the following requirements table:

Name	Alarm clock
Purpose	A 24-h digital clock with a single alarm.
Inputs	Six pushbuttons: set time; set alarm, hour, minute, alarm on, alarm off.
Outputs	Four-digit, clock-style output. PM indicator light. Alarm ready light. Buzzer.
Functions	<p>Default mode: The display shows the current time. PM light is on from noon to midnight.</p> <p>Hour and minute buttons are used to advance time and alarm, respectively. Pressing one of these buttons increments the hour/minute once.</p> <p>Depress set time button: This button is held down while hour/minute buttons are pressed to set time. New time is automatically shown on the display.</p> <p>Depress set alarm button: While this button is held down, display shifts to current alarm setting; depressing hour/minute buttons set alarm value in a manner like setting time.</p> <p>Alarm on: Puts clock in alarm-on state, causes clock to turn on buzzer when current time reaches alarm time, turns on alarm ready light.</p> <p>Alarm off: Turns off buzzer, takes clock out of alarm-on state, turns off alarm ready light.</p>

*Continued*

*—Continued*

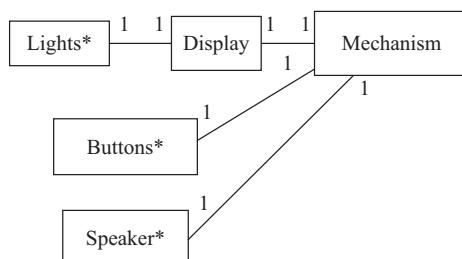
Name	Alarm clock
Performance	Displays hours and minutes, but not seconds. Should be accurate within the accuracy of a typical microprocessor clock signal. (Excessive accuracy may unreasonably drive up the cost of generating an accurate clock.)
Manufacturing cost	Consumer product range. Cost will be dominated by the microprocessor system, not the buttons or displays.
Power	Powered by AC through a standard power supply.
Physical size and weight	Small enough to fit on a nightstand with expected weight for an alarm clock.

### 4.11.2 Specification

The basic function of the clock is simple, but we need to create some classes and associated behaviors to clarify exactly how the user interface works.

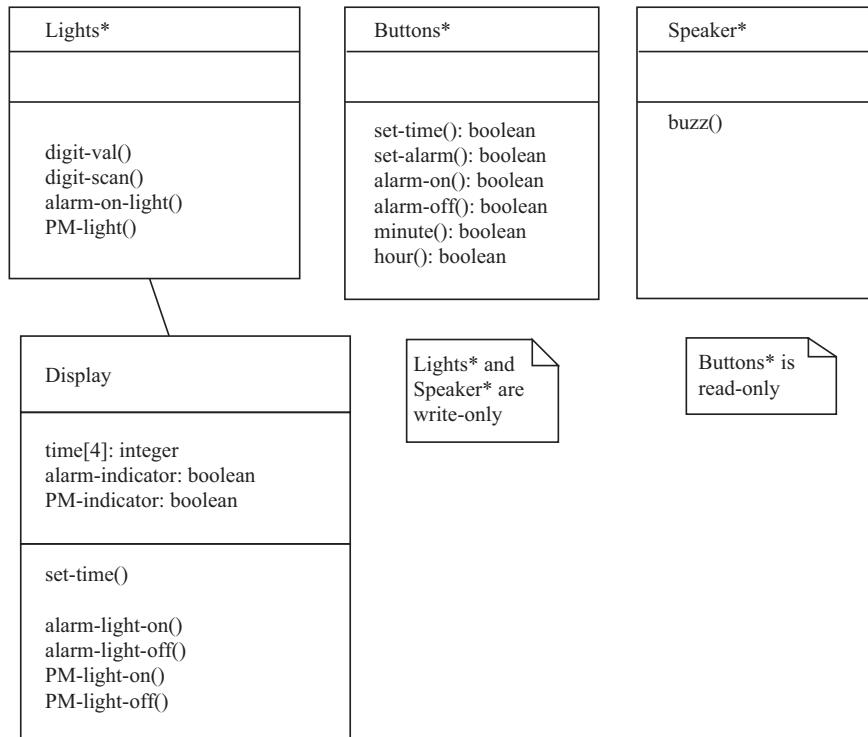
Fig. 4.33 shows the basic classes for the alarm clock. Borrowing a term from mechanical watches, we call the class that handles the basic clock operation the *Mechanism* class. We have three classes that represent physical elements: *Lights\** for all the digits and lights, *Buttons\** for all the buttons, and *Speaker\** for the sound output. The *Buttons\** class can easily be used directly by *Mechanism*. As discussed below, the physical display must be scanned to generate the digit output, so we introduce the *Display* class to abstract the physical lights.

The details of the low-level user interface classes are shown in Fig. 4.34. The *Buttons\** class provides read-only access to the current state of the buttons. The *Lights\** class allows us to drive the lights. However, to save pins on the display, *Lights\** provides signals for only one digit, along with a set of signals to indicate which digit is currently being addressed. We generate the display by scanning the digits periodically. That function is performed by the *Display* class, which makes the display appear as an unscanned, continuous display to the rest of the system.



**FIGURE 4.33**

Class diagram for the alarm clock.

**FIGURE 4.34**

Details of user interface classes for the alarm clock.

The *Mechanism* class is presented in Fig. 4.35. This class keeps track of the current time, the current alarm time, whether the alarm has been turned on, and whether it is currently buzzing. The clock shows the time only to the minute, but it keeps the internal time to the second. The time is kept as discrete digits rather than as a single integer to simplify transferring the time to the display. The class provides two behaviors, both of which run continuously. First, the *scan-keyboard* handles looking at the inputs and updating the alarm and other functions as requested by the user. Second, *update-time* keeps the current time accurate.

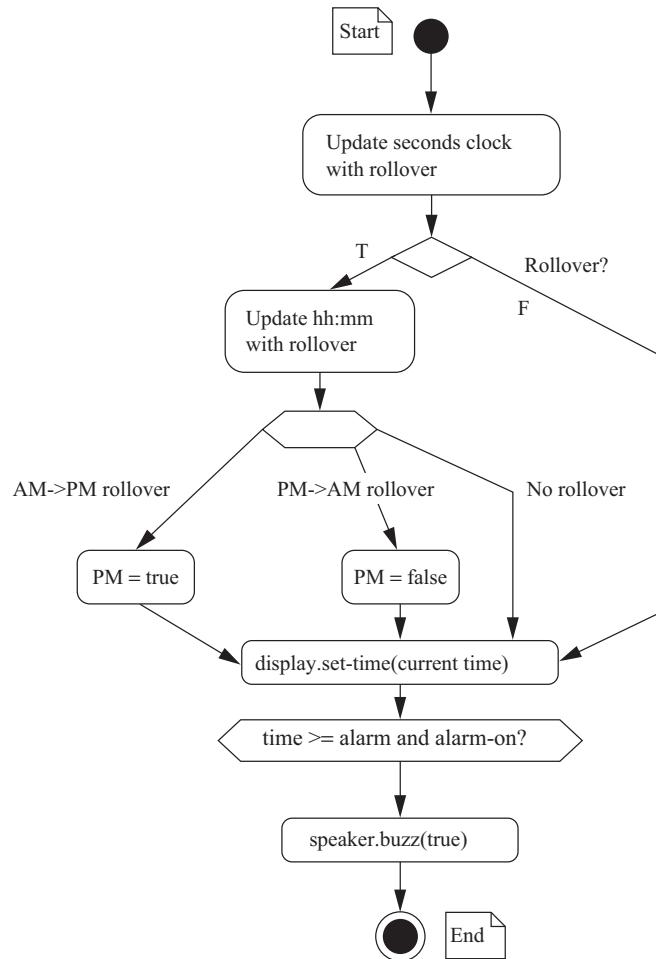
Fig. 4.36 shows the state diagram for *update-time*. This behavior is straightforward, but it must do several things. It is activated once per second and must update the second clock. If it has counted 60 seconds, it must then update the displayed time; when it does so, it must roll over between digits and keep track of AM-to-PM and PM-to-AM transitions. It sends the updated time to the display object. It also compares the time with the alarm setting and sets the alarm buzzing under the proper conditions.

The state diagram for the *scan-keyboard* is shown in Fig. 4.37. This function is called periodically, frequently enough, so that all the user's button presses are

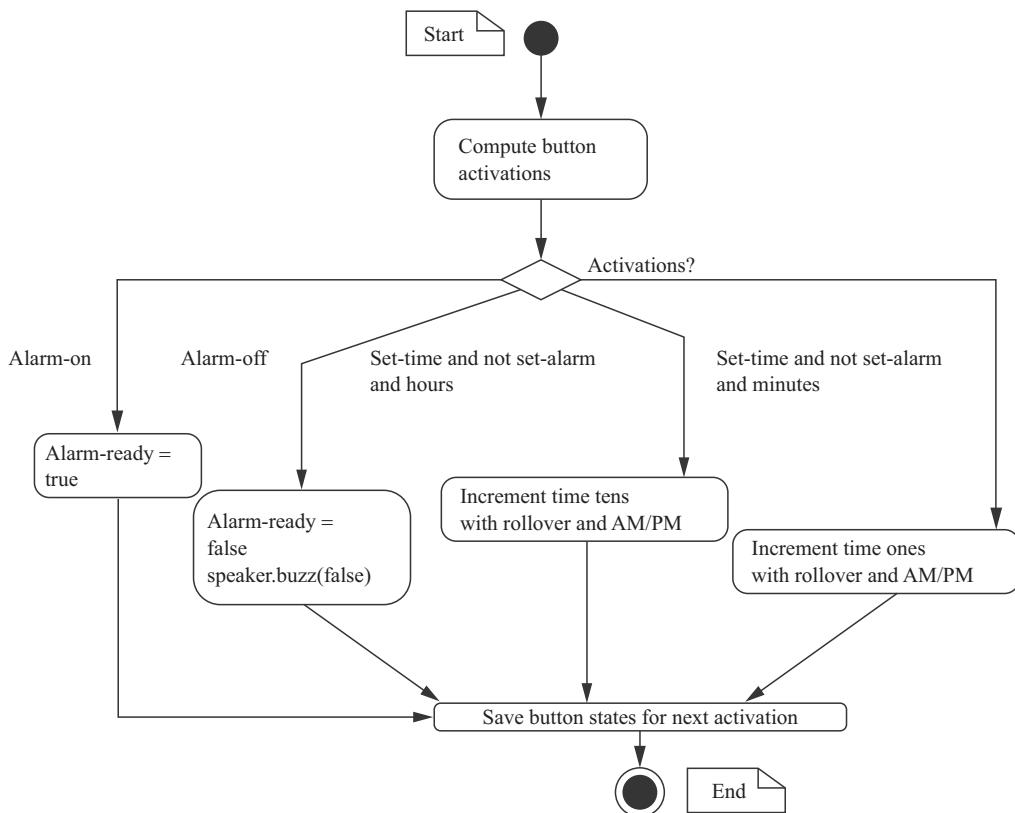
Mechanism
scan-keyboard runs periodically
update-time runs once per second
seconds: integer PM: boolean tens-hours, ones-hours: integer tens-minutes, ones-minutes: integer alarm-ready: boolean  alarm-tens-hours, alarm-ones-hours: integer alarm-tens-minutes, alarm-ones-minutes: integer  scan-keyboard() update-time()

**FIGURE 4.35**

The mechanism class.

**FIGURE 4.36**

State diagram for update-time.

**FIGURE 4.37**

State diagram for scan-keyboard.

caught by the system. Because the keyboard will be scanned several times per second, we don't want to register the same button press several times. If, for example, we advanced the minutes count on every keyboard scan when the *set time* and *minutes* buttons were pressed, the time would be advanced much too fast. To make the buttons respond more reasonably, the function computes button activations; it compares the current state of the button to the button's value on the last scan, and it considers the button activated only when it is on for this scan but was off for the last scan. Once computing the activation values for all the buttons, it looks at the activation combinations and takes the appropriate actions. Before exiting, it saves the current button values for computing activations the next time this behavior is executed.

### 4.11.3 System architecture

The software and hardware architectures of a system are always hard to completely separate, but let's first consider the software architecture, and then, its implications for the hardware.

The system has both periodic and aperiodic components; the current time must obviously be updated periodically, and the button commands occur occasionally.

It seems reasonable to have the following two major software components:

- An interrupt-driven routine can update the current time. The current time will be kept in a variable in the memory. A timer can be used to interrupt periodically and update the time. As seen in the subsequent discussion of the hardware architecture, the display must send the new value when the minute value changes. This routine can also maintain the PM indicator.
- A foreground program can poll the buttons and execute their commands. Because buttons are changed at a relatively slow rate, it makes no sense to add the hardware required to connect the buttons to interrupts. Instead, the foreground program will read the button values, and then, use simple conditional tests to implement the commands, including setting the current time, setting the alarm, and turning off the alarm. Another routine called by the foreground program will turn the buzzer on and off based on the alarm time.

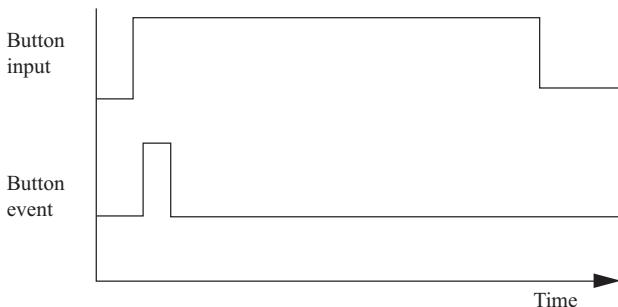
An important question for the interrupt-driven current time handler is how often the timer interrupts occur. A one-minute interval would be very convenient for the software, but a one-minute timer would require many counter bits. It is more realistic to use a 1-s timer and a program variable to count the seconds in a minute.

The foreground code will be implemented as a while loop.

```
while (TRUE) {
    read_buttons(button_values); /*read inputs*/
    process_command(button_values); /*do commands*/
    check_alarm(); /*decide whether to turn on the alarm*/
}
```

The loop first reads the buttons using `read_buttons()`. In addition to reading the current button values from the input device, this routine must preprocess the button values so that the user interface code responds properly. The buttons will remain depressed for many sample periods, because the sample rate is much faster than any person can push and release buttons. We want to make sure that the clock responds to this as a single depression of the button, not as one depression per sample interval. As shown in Fig. 4.38, this can be done by performing a simple edge detection on the button input; the button event value is one for one sample period when the button is depressed, and then, goes back to zero and does not return to one until the button is depressed, and then, released. This can be accomplished using a simple two-state machine.

The `process_command()` function handles responding to button events. The `check_alarm()` function checks the current time against the alarm time and decides

**FIGURE 4.38**

Preprocessing button inputs.

when to turn on the buzzer. This routine is kept separate from the command processing code because the alarm must go on when the proper time is reached, independent of the button inputs.

We have determined from the software architecture that we will need a timer connected to the CPU. We will also need logic to connect the buttons to the CPU bus. In addition to performing edge detection on the button inputs, we must, of course, debounce the buttons.

The final step before starting to write code and build hardware is to draw the state transition graph for the clock's commands. This diagram will be used to guide the implementation of the software components.

#### 4.11.4 Component design and testing

The two major software components, the interrupt handler and the foreground code, can be implemented relatively straightforwardly. Because most of the functionality of the interrupt handler is in the interruption process itself, that code is best tested on the microprocessor platform. The foreground code can be more easily tested on the PC or workstation used for code development. We can create a testbench for this code that generates button depressions to exercise the state machine. We will also need to simulate the advancement of the system clock. Trying to directly execute the interrupt handler to control the clock is probably a bad idea; not only would that require some type of emulation of interrupts, but it would also require us to count interrupts second by second. A better testing strategy is to add testing code that updates the clock, perhaps once per four iterations of the foreground `while` loop.

The timer will probably be a stock component, so we would then focus on implementing logic to interface with the buttons, display, and buzzer. The buttons will require debouncing logic. The display will require a register to hold the current display value to drive the display elements.

### 4.11.5 System integration and testing

Because this system has a small number of components, system integration is relatively easy. The software must be checked to ensure that the debugging code has been turned off. Three types of tests can be performed. First, the clock's accuracy can be checked against a reference clock. Second, commands can be exercised from the buttons. Finally, the buzzer's functionality should be verified.

---

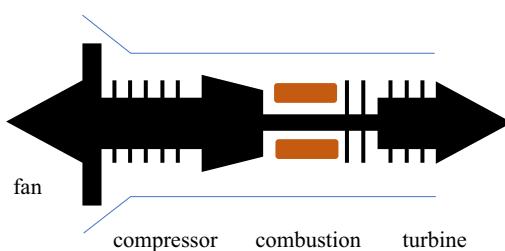
## 4.12 Design example: jet engine controller

A jet engine requires real-time controls to ensure that it responds properly to the pilot's actions. Balancing the computations required for control with sensing and actuation using the I/O devices gives us a good example of performance analysis in bus-based systems.

### 4.12.1 Theory of operation and requirements

A jet engine's operation is remarkably simple [Rol15]. As shown in Fig. 4.39, air enters the engine through a fan, which pushes the air through a compressor. The highly compressed air flows into a combustion chamber, where fuel is applied. The burning exhaust flows out the back of the engine through a turbine. The turbine is turned by the expelled air. Shafts couple that rotation back to the compressor and fan to keep the engine fed with air. The fan also provides airflow that bypasses the combustion chamber; this airflow provides additional thrust. The compressor runs continuously.

The digital controller for a jet engine is known as **full authority digital electronic control (FADEC)**. A traditional jet engine has one control: the throttle [Gar, Liu12]. Engine thrust cannot be directly sensed, but can be inferred. Thrust is a function of shaft speed that can be directly measured. Sensors can also be installed to measure variables related to the health of the engine, such as pressure and temperature.



**FIGURE 4.39**

Cross-section of a jet engine.

### 4.12.2 Specifications

The pilot's command input to the engine is the throttle. The throttle position is known as the **throttle resolver angle (TRA)**. A typical engine uses a pair of shafts, one for low-pressure (N1) and another for high-pressure (N2) sections. Sensors can directly measure the rotational speed of these shafts. These measurements allow for feedback control of the engine; the shaft speeds, and therefore, the thrust can be compared to the commanded thrust.

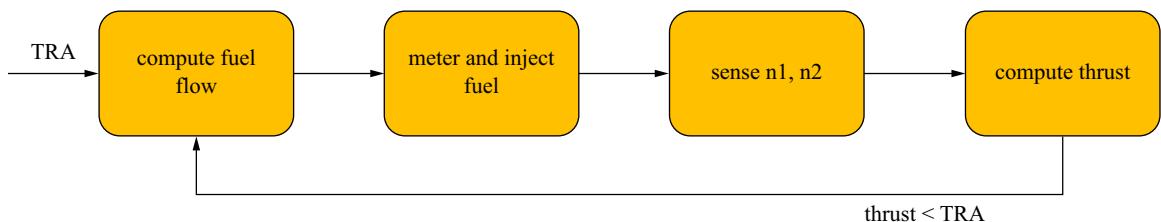
We will concentrate on a very simple control algorithm for the jet engine shown in Fig. 4.40:

- The throttle resolver angle is the pilot's command input to the jet engine.
- Fuel flow (WF) is computed based on TRA and the current computed thrust; that WF is sent to the fuel system.
- The shaft speeds N1 and N2 are sensed.
- Thrust is computed from the shaft speeds. The results are stored for the next control cycle.

A typical sample rate for the engine controller is in the  $10 - 100 \text{ Hz}$  range.

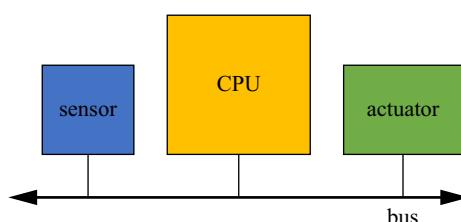
### 4.12.3 System architecture

As shown in Fig. 4.41, the computing platform for a FADEC typically consists of a CPU and a separate I/O bus such as the controller area network (CAN) bus that



**FIGURE 4.40**

Jet engine control algorithm.



**FIGURE 4.41**

Simple computing platform for jet engine control.

will be discussed in [Chapter 10](#). The CPU bus is not used directly to connect to devices; the CPU bus is connected to the CAN bus, which in turn connects to the sensors and actuators. However, the scheduling principles for this architecture are the same as those for a CPU bus.

As shown in [Fig. 4.42](#), the control computation is divided into two tasks: computing WF from the TRA and computing thrust from shaft speeds. The WF computation cannot begin until after the TRA has been received. However, the bus can read the shaft speeds N1 and N2, while the CPU performs that computation. Similarly, the bus can send WF to the fuel system, while the thrust is computed from the shaft speeds.

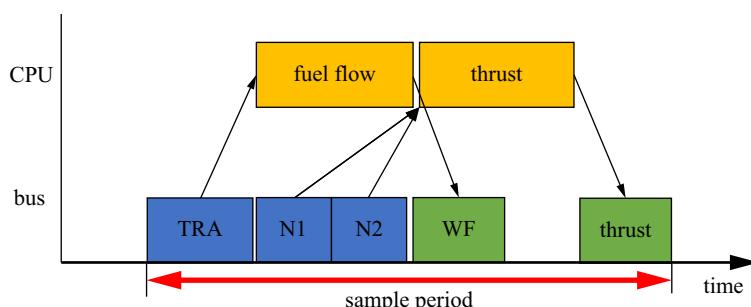
#### 4.12.4 Component design

Control algorithms can be evaluated using a model of the jet engine. C-MAPSS [Liu12], for example, is a Simulink model of a jet engine. FADEC software will often be coded in C language. The numerical behavior of the control algorithms is important for proper operation. The required numerical precision can be evaluated using jet engine models, and the execution time required for various number formats can be determined by software performance analysis.

#### 4.12.5 System integration and testing

A test harness could be built to test the software on the target computing platform. The harness would provide sensor data and log actuator data. The results could be evaluated after each run to assess the software's correctness.

Jet engine test stands must be carefully designed. A test stand, for example, may allow the engine to operate remotely. The test bay can be separated from the interior of the building by a wall and a shatter-resistant observation window. A large set of louvers fill most of the outside wall of the test bay; in case of an explosion, the louvers blow open and vent to the outside.



**FIGURE 4.42**

Schedule for I/O and computation in a jet engine controller.

---

### 4.13 Summary

The microprocessor is only one component in an embedded computing system; memory and I/O devices are equally important. The microprocessor bus serves as the glue that binds all these components together. Hardware platforms for embedded systems are often built around common platforms with appropriate amounts of memory and I/O devices added on; low-level monitor software also plays an important role in these systems.

---

### What we learned

- CPU buses are built on handshaking protocols.
- A variety of memory components are available, which vary widely in speed, capacity, and other capabilities.
- An I/O device uses logic to interface to the bus so that the CPU can read and write the device's registers.
- Embedded systems can be debugged using a variety of hardware and software methods.
- System-level performance depends not just on the CPU, but the memory and bus as well.
- Security features of the platform enable the development of secure applications.

---

### Further reading

Shanley and Anderson [Min95] describe the PCI bus in detail. Dahlin [Dah00] describes how to interface to a touchscreen. Collins [Col97] describes the design of microprocessor ICEs. Earnshaw et al. [Ear97] describe an advanced debugging environment for the Arm architecture.

---

### Questions

- Q4-1** Name three major hardware components of a generic computing platform.
- Q4-2** Name three major software components of a generic computing platform.
- Q4-2** What role does the HAL play in the platform?
- Q4-4** Draw UML state diagrams for device 1 and device 2 in a four-cycle handshake.

- Q4-5** Describe the role of these signals in a bus:
- R/W'
  - data-ready
  - clock
- Q4-6** Draw a UML sequence diagram that shows a four-cycle handshake between a bus controller and a device.
- Q4-7** Define these signal types in a timing diagram:
- changing
  - stable
- Q4-8** Draw a timing diagram with the following signals (where  $[t_1, t_2]$  is the time interval, starting at  $t_1$  and ending at  $t_2$ ):
- Signal A is stable [0,10], changing [10,15], stable [15,30].
  - Signal B is 1 [0,5], falling [5,7], 0 [7,20], changing [20,30].
  - Signal C is changing [0,10], 0 [10,15], rising [15,18], 1 [18,25], changing [25,30].
- Q4-9** Draw a timing diagram for a write operation with no wait states.
- Q4-10** Draw a timing diagram for a read operation on a bus in which the read includes two wait states.
- Q4-11** Draw a timing diagram for a write operation on a bus in which the write takes two wait states.
- Q4-12** Draw a timing diagram for a burst write operation that writes four locations.
- Q4-13** Draw a UML state diagram for a burst read operation with wait states. One state diagram is for the bus controller and the other is for the device being read.
- Q4-14** Draw a UML sequence diagram for a burst read operation with wait states.
- Q4-15** Draw timing diagrams for
- a device becoming bus controller; and
  - the device returning control of the bus to the CPU.
- Q4-16** Draw a timing diagram that shows a complete DMA operation, including handing off the bus to the DMA controller, performing the DMA transfer, and returning bus control back to the CPU.
- Q4-17** Draw UML state diagrams for a bus controllership transaction, in which one side shows the CPU as the default bus controller and the other shows the device that can request bus controllership.
- Q4-18** Draw a UML sequence diagram for a bus controllership request, grant, and return.

- Q4-19** Draw a UML sequence diagram that shows a DMA bus transaction and concurrent processing on the CPU.
- Q4-20** Draw a UML sequence diagram for a complete DMA transaction, including the DMA controller requesting the bus, the DMA transaction itself, and returning control of the bus to the CPU.
- Q4-21** Draw a UML sequence diagram showing a read operation across a bus bridge.
- Q4-22** Draw a UML sequence diagram showing a write operation with wait states across a bus bridge.
- Q4-23** Draw a UML sequence diagram for a read transaction that includes a DRAM refresh operation. The sequence diagram should include the CPU, the DRAM interface, and the DRAM internals to show the refresh itself.
- Q4-24** Draw a UML sequence diagram for a DRAM read operation. Show the activity of each of the DRAM signals.
- Q4-25** What is the role of a memory controller in a computing platform?
- Q4-26** What hardware factors might be considered when choosing a computing platform?
- Q4-27** What software factors might be considered when choosing a computing platform?
- Q4-28** Write ARM assembly language code that handles a breakpoint. It should save the necessary registers, call a subroutine to communicate with the host, and upon return from the host, cause the breakpointed instruction to be properly executed.
- Q4-29** Assume an A/D converter is supplying samples at 44.1 kHz.
- How much time is available per sample for CPU operations?
  - If the interrupt handler executes 100 instructions obtaining the sample and passing it onto the application routine, how many instructions can be executed on a 20-MHz RISC processor that executes 1 instruction per cycle?
- Q4-30** If an interrupt handler executes for too long and the next interrupt occurs before the last call to the handler has finished, what happens?
- Q4-31** Consider a system in which an interrupt handler passes on samples to a finite impulse response (FIR) filter program that runs in the background.
  - If the interrupt handler takes too long, how does the FIR filter's output change?
  - If the FIR filter code takes too long, how does its output change?

- Q4-32** Assume that your microprocessor implements an ICE instruction that asserts a bus signal that causes a microprocessor ICE to start. Additionally, assume that the microprocessor allows all internal registers to be observed and controlled through a boundary scan chain. Draw a UML sequence diagram of the ICE operation, including execution of the ICE instruction, uploading the microprocessor state to the ICE, and returning control to the microprocessor's program. The sequence diagram should include the microprocessor, the microprocessor ICE, and the user.
- Q4-33** Why might an embedded computing system want to implement a DOS-compatible file system?
- Q4-34** Name two example embedded systems that implement a DOS-compatible file system.
- Q4-35** You are given a memory system with an overhead  $O=2$  and a single-word transfer time of 1 (no wait states.) You will use this memory system to perform a transfer of 1,024 locations. Plot total number of clock cycles  $T$  as a function of burst size  $B$  for  $1 <= B <= 8$ .
- Q4-36** You are given a bus that supports single-word and burst transfers. A single transfer takes one clock cycle (no wait states). The overhead of the single-word transfer is 1 clock cycles ( $O = 1$ ). The overhead of a burst is 3 clock cycles ( $O_B = 3$ ) Which performs a two-word transfer faster: a pair of single transfers or a single burst of two words?
- Q4-37** You are given a 2-byte wide bus that supports single-byte, dual word (same clock cycle) and burst transfers of up to 8 bytes (four byte pairs per burst). The overhead of each of these types of transfers is 1 clock cycle ( $O = O_B = 1$ ) and a data transfer takes one clock cycle per single or dual word ( $D=1$ ). You want to send a 1080P video frame at a resolution of  $1920 \times 1080$  pixels with 3 bytes per pixel. Compare the difference in bus transfer times if the pixels are packed vs. sending a pixel as a 2-byte followed by a single-byte transfer.
- Q4-38** Determine the design parameters for an audio system:
- Determine the total bytes/s required for an audio signal of 16 bits/sample per channel, two channels, sampled at 44.1 kHz.
  - Given a clock period  $P = 20$  MHz for a bus, determine the bus width required assuming that nonburst mode transfers are used and  $D = O = 1$ .
  - Given a clock period  $P = 20$  MHz for a bus, determine the bus width required assuming burst transfers of length four and  $D = O_B = 1$ .
  - Assume the data signal now contains both the original audio signal and a compressed version of the audio at a bit rate of 1/10 the input audio signal. Assume bus bandwidth for burst transfers of length four with  $P = 20$  MHz and  $D = O_B = 1$ . Will a bus of width 1 be sufficient to handle the combined traffic?

- Q4-39** You are designing a system a bus-based computer: the input device I1 sends its data to program P1; P1 sends its output to device O1. Is there any way to overlap bus transfers and computations in this system?
- Q4-40** What hardware modules might be used to create a digital signature?
- 

## Lab exercises

- L4-1** Use a logic analyzer to view system activity on your bus.
- L4-2** If your logic analyzer is capable of on-the-fly disassembly, use it to display bus activity in the form of instructions, rather than simply ones and zeroes.
- L4-3** Attach LEDs to your peripheral bus so that you can monitor its activity. For example, use an LED to monitor the read/write line on the bus.
- L4-4** Design logic to interface an I/O device to your microprocessor.
- L4-5** Have someone else deliberately introduce a bug into one of your programs, and then use the appropriate debugging tools to find and correct the bug.
- L4-6** Identify the different bus transaction types in your platform. Compute the best-case bus bandwidth.
- L4-7** Construct a simple program to access memory in widely separated places. Measure the memory system bandwidth and compare to the best-case bandwidth.
- L4-8** Construct a simple program to perform some memory accesses. Use a logic analyzer to study the bus activity. Determine what types of bus modes are used for the transfers.

# Program Design and Analysis

# 5

## CHAPTER POINTS

---

- Some useful components for embedded software.
- Models of programs, such as data flow and control flow graphs.
- An introduction to compilation methods.
- Analyzing and optimizing programs for performance, size, and power consumption.
- How to test programs to verify their accuracy.
- Design examples: software modem and digital still camera (DSC).

---

## 5.1 Introduction

In this chapter, we study in detail the process of creating programs for embedded processors. The creation of embedded programs is at the heart of embedded system design. If you are reading this book, you almost certainly know how to write code, but designing and implementing embedded programs is different and more challenging than writing typical workstation or PC programs. Embedded code must not only provide rich functionality, but also often run at a required rate to meet system deadlines, fit into the allowed amount of memory, and meet power consumption requirements. Designing code that simultaneously meets multiple design constraints is a considerable challenge, but luckily there are techniques and tools that we can use to help us through the design process. Ensuring that the program works is also a challenge, but once again methods and tools come to our aid.

Throughout the discussion we concentrate on high-level programming languages, specifically C. High-level languages were once shunned as too inefficient for embedded microcontrollers, but better compilers, more compiler friendly architectures, and faster processors and memory have made high-level language programs common. Some sections of a program may still need to be written in assembly language if the compiler doesn't provide sufficiently good results, but even when coding in assembly language, it is often helpful to think about the program's functionality in high-level form. Many of the analysis and optimization techniques that we study in this chapter are equally applicable to programs that are written in assembly language.

The next section talks about some software components that are commonly used in embedded software. Section 5.3 introduces the control/data flow graph as a model for high-level language programs (which can also be applied to programs originally written in assembly language). Section 5.4 reviews the assembly and linking process, and Section 5.5 introduces some compilation techniques. Section 5.6 introduces methods for analyzing the performance of programs. We talk about optimization techniques that are specific to embedded computing in the next three sections: performance in Section 5.7, energy consumption in Section 5.8, and size in Section 5.9. In Section 5.10, we discuss techniques for ensuring that the programs you write are correct. In Section 5.11, we consider the related problem of program design for safety and security. We close with two design examples: a software modem in Section 5.12 and a DSC in Section 5.13.

---

## 5.2 Components for embedded programs

In this section, we consider code for three structures or components that are commonly used in embedded software: the state machine, circular buffer, and queue. State machines are well suited to **reactive systems** such as user interfaces; circular buffers and queues are useful in digital signal processing.

### 5.2.1 State machines

#### State machine style

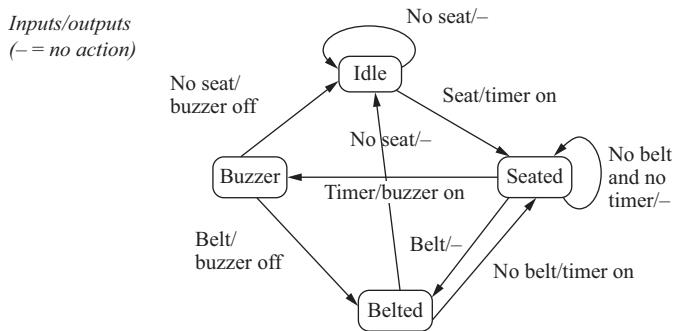
When inputs appear intermittently rather than as periodic samples, it is often convenient to think of the system as reacting to those inputs. The reaction of most systems can be characterized in terms of the input received and the current state of the system. This naturally leads to a **finite-state machine** style of describing the reactive system's behavior. Moreover, if the behavior is specified in that way, it is natural to write the program implementing that behavior in a state machine style. The state machine style of programming is also an efficient implementation of such computations. Finite-state machines are usually first encountered in the context of hardware design.

Programming Example 5.1 shows how to write a finite-state machine in a high-level programming language.

---

### Programming Example 5.1: A State Machine in C

The behavior we want to implement is a simple seat belt controller [Chi94]. The controller's job is to turn on a buzzer if a person sits in a seat and does not fasten the seat belt within a fixed amount of time. This system has three inputs and one output. The inputs are a sensor for the seat to know when a person has sat down, a seat belt sensor that tells when the belt is fastened, and a timer that goes off when the required time interval has elapsed. The output is the buzzer. Here is a state diagram that describes the seat belt controller's behavior:



The idle state is in force when there is no person in the seat. When the person sits down, the machine goes into the seated state and turns on the timer. If the timer goes off before the seat belt is fastened, the machine goes into the buzzer state. If the seat belt goes on first, it enters the belted state. When the person leaves the seat, the machine goes back to idle.

To write this behavior in C, we will assume that we have loaded the current values of all three inputs (seat, belt, timer) into variables and will similarly hold the outputs in variables temporarily (timer\_on, buzzer\_on). We will use a variable named state to hold the current state of the machine and a switch statement to determine which action to take in each state. Here is the code:

```

#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3

switch(state) { /*check the current state */
    case IDLE:
        if (seat){ state = SEATED; timer_on = TRUE; }
        /*default case is self-loop */
        break;
    case SEATED:
        if (belt) state = BELTED; /*won't hear the buzzer */
        else if (timer) state = BUZZER; /*didn't put on
        belt in time */
        /*default case is self-loop */
        break;
    case BELTED:
        if (!seat) state = IDLE; /* person left */
        else if (!belt) state = SEATED; /* person still
        in seat */
        break;
    case BUZZER:
        if (belt) state = BELTED; /*belt is on--turn off
        buzzer */
        else if (!seat) state = IDLE; /* no one in seat--
        turn off buzzer */
        break;
}
  
```

This code takes advantage of the fact that the state will remain the same unless explicitly changed; this makes self-loops back to the same state easy to implement. This state machine may be executed forever in a `while (TRUE)` loop or periodically called by some other code. In either case, the code must be executed regularly so that it can check on the current value of the inputs, and if necessary, go into a new state.

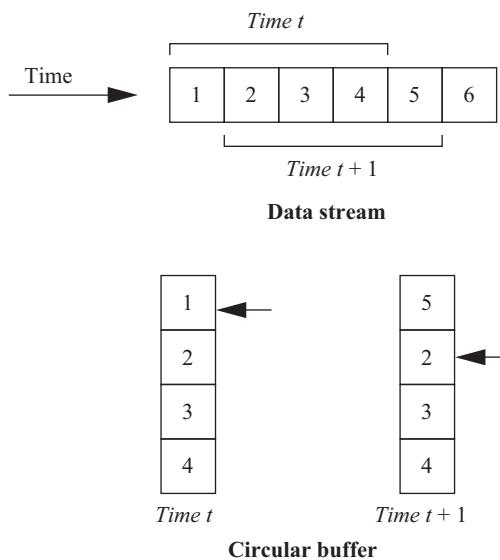
### 5.2.2 Circular buffers and stream-oriented programming

#### Data stream style

The data stream style makes sense for data that come in regularly and must be processed on the fly. The finite impulse response (FIR) filter of Application Example 2.1 is a classic example of stream-oriented processing. For each sample, the filter must emit one output that depends on the values of the last  $n$  inputs. In a typical workstation application, we would process the samples over a given interval by reading them all in from a file, and then, computing the results all at once in a batch process. In an embedded system, we must not only emit outputs in real time, but we must also do so using a minimum amount of memory.

#### Circular buffer

The **circular buffer** is a data structure that lets us handle streaming data in an efficient way. Fig. 5.1 illustrates how a circular buffer stores a subset of the data stream. At each point in time, the algorithm needs a subset of the data stream that forms a window into the stream. The window slides with time as we throw out old values that are no longer needed and add new values. Because the size of the window does not



**FIGURE 5.1**

A circular buffer.

change, we can use a fixed-sized buffer to hold the current data. To avoid constantly copying data within the buffer, we will move the head of the buffer in time. The buffer points to the location at which the next sample will be placed; every time we add a sample, we automatically overwrite the oldest sample, which is the one that needs to be thrown out. When the pointer reaches the end of the buffer, it wraps around to the top.

#### **Instruction set support**

#### **High-level language implementation**

Many digital signal processors provide addressing modes to support circular buffers. For example, the C55x [Tex04] provides five circular buffer start address registers (their names start with BSA). These registers allow circular buffers to be placed without alignment constraints.

In the absence of specialized instructions, we can write our own C code for a circular buffer. This code also helps us to understand the operation of the buffer. Programming Example 5.2 provides an efficient implementation of a circular buffer.

### **Programming Example 5.2: A Circular Buffer in C**

Once we build a circular buffer, we can use it in a variety of ways. We will use an array as the buffer:

```
#define CMAX 6 /*filter order */

int circ[CMAX]; /*circular buffer */
int pos; /*position of current sample */
```

The variable pos holds the position of the current sample. As we add new values to the buffer, this variable moves.

Here is the function that adds a new value to the buffer:

```
void circ_update(int xnew) {
    /*add the new sample and push off the oldest one */

    /*compute the new head value with wraparound; the pos
     pointer moves from 0 to CMAX-1 */
    pos = ((pos == CMAX-1) ? 0 : (pos+1));
    /*insert the new value at the new head */
    circ[pos] = xnew;
}
```

The assignment to pos takes care of wraparound; when pos hits the end of the array, it returns to zero. We then put the new value into the buffer at the new position. This overwrites the old value that was there. Note that as we go to higher index values in the array, we march through the older values.

We can now write an initialization function, which sets the buffer values to zero. More importantly, it sets pos to the initial value. For ease of debugging, we want the first data

element to go into circ[0]. To do this, we set pos to the end of the array so that it is set to zero before the first element is added:

```
void circ_init() {
    int i;

    for (i=0; i<CMAX; i++) /* set values to 0 */
        circ[i] = 0;
    pos=CMAX-1; /*start at tail so first element will be at 0 */
}
```

We can also make use of a function to get the  $i^{\text{th}}$  value of the buffer. This function has to translate the index in temporal order, with zero being the newest value, to its position in the array:

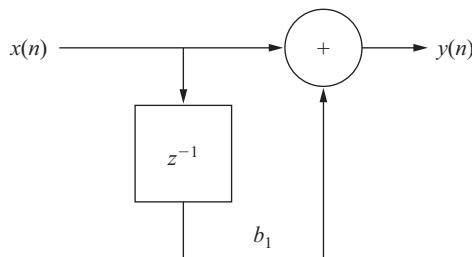
```
int circ_get(int i) {
    /*get the ith value from the circular buffer */
    int ii;
    /*compute the buffer position */
    ii = (pos - i) % CMAX;
    /*return the value */
    return circ[ii];
}
```

---

We can now write C code for a digital filter. To help us to understand the filter algorithm, we can introduce a widely used representation for filter functions.

#### Signal flow graph

The FIR filter is only one type of digital filter. We can represent many different filtering structures using a **signal flow graph**, as shown in Fig. 5.2. The filter operates at a sample rate, with inputs arriving and outputs generated at the sample rate. The inputs  $x[n]$  and  $y[n]$  are sequences indexed by  $n$ , which corresponds to the sequence of samples. Nodes in the graph can represent either arithmetic operators or delay operators. The  $+$  node adds its two inputs and produces the output  $y[n]$ . The box labeled  $z^{-1}$  is a delay operator. The  $z$  notation comes from the  $z$  transform that is used in digital signal processing; the  $-1$  superscript means that the operation performs a time delay of one sample period. The edge from the delay operator to the



**FIGURE 5.2**

A signal flow graph.

**Filters and buffering**

addition operator is labeled with  $b_1$ , meaning that the output of the delay operator is multiplied by  $b_1$ .

The code to produce one FIR filter output looks like this:

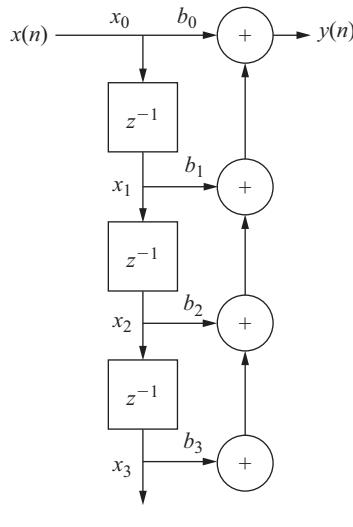
```
for (i=0, y=0.0; i<N; i++)
    y += x[i] * b[i];
```

However, the filter takes in a new sample in every sample period. The new input becomes  $x_1$ , the old  $x_1$  becomes  $x_2$ , and so on.  $x_0$  is stored directly in the circular buffer, but must be multiplied by  $b_0$  before being added to the output sum. Early digital filters were built-in hardware, where we could build a shift register to perform this operation. If we used an analogous operation in software, we would move every value in the filter in every sample period. We can avoid this with a circular buffer, moving the head without moving the data elements.

The next example uses our circular buffer class to build a FIR filter.

### Programming Example 5.3: An FIR Filter in C

Here is a signal flow graph for an FIR filter:



The delay elements running vertically hold the input samples, with the most recent sample at the top and the oldest one at the bottom. Unfortunately, the signal flow graph doesn't explicitly label all of the values that we use as inputs to operations, so the figure also shows the values ( $x_i$ ) that we need to operate on in our FIR loop.

When we compute the filter function, we want to match the  $b_i$ 's and  $x_i$ 's. We will use our circular buffer for the  $x$ 's, which change over time. We will use a standard array for the  $b$ 's that don't change. In order for the filter function to be able to use the same  $i$  value for both sets of data, we need to put the  $x$  data in the proper order. We can put the  $b$  data in a standard

array, with  $b_0$  being the first element. When we add a new  $x$  value, it becomes  $x_0$  and replaces the oldest data value in the buffer. This means that the buffer head moves from higher to lower values, and not from lower to higher as we might expect.

Here is the modified version of `circ_update()` that puts a new sample into the buffer in the desired order:

```
void circ_update(int xnew) {
    /*add the new sample and push off the oldest one */

    /*compute the new head value with wraparound; the pos
pointer moves from CMAX-1 down to 0 */
    pos = ((pos == 0) ? CMAX-1 : (pos-1));
    /*insert the new value at the new head */
    circ[pos] = xnew;
}
```

We also need to change `circ_init()` to set `pos = 0` initially. We don't need to change `circ_get()`.

Given these functions, the filter itself is simple. Here is our code for the FIR filter function:

```
int fir(int xnew) {
    /*given a new sample value, update the queue and compute the
filter output */
    int i;

    int result; /*holds the filter output */

    circ_update(xnew); /*put the new value in */
    for (i=0, result=0; i<CMAX; i++) /* compute the filter
function */
        result += b[i] * circ_get(i);
    return result;
}
```

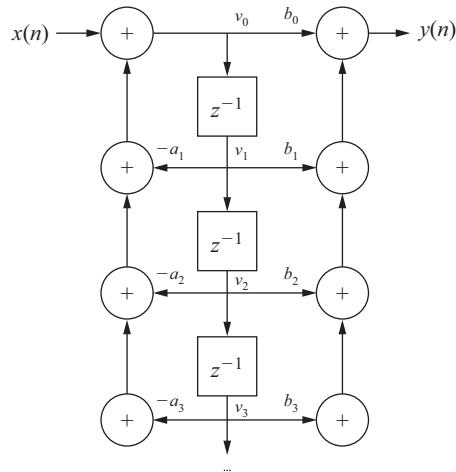
---

There is only one major structure for FIR filters, but several possible structures for infinite impulse response (IIR) filters, depending on the application requirements. One of the important reasons for so many different IIR forms is numerical properties; depending on the filter structure and coefficients, one structure may give significantly less numerical noise than another. However, numerical noise is beyond the scope of our discussion, so let's concentrate on one form of the IIR filter that highlights buffering issues. The next example looks at one form of the IIR filter.

---

### Programming Example 5.4: A Direct Form II IIR Filter Class in C

Here is what is known as the direct form II of an IIR filter:



This structure is designed to minimize the amount of buffer space required. Other forms of the IIR filter have other advantages, but require more storage. We will store the  $v_i$  values as in the FIR filter. In this case,  $v_0$  does not represent the input, but rather the left-hand sum. However,  $v_0$  is stored before multiplication by  $b_0$  so that we can move  $v_0$  to  $v_1$  in the following sample period.

We can use the same `circ_update()` and `circ_get()` functions that we used for the FIR filter. We need two coefficient arrays: one for  $a$ s and one for  $b$ s; as with the FIR filter, we can use standard C arrays for the coefficients because they don't change over time. Here is the IIR filter function:

```
int iir2(int xnew) {
    /*given a new sample value, update the queue and compute
    the filter output*/
    int i, aside, bside, result;

    for (i=0, aside=0; i<ZMAX; i++)
        aside += -a[i+1] * circ_get(i);
    for (i=0, bside=0; i<ZMAX; i++)
        bside += b[i+1] * circ_get(i);
    result = b[0] *(xnew + aside) + bside;
    circ_update(xnew); /*put the new value in */
    return result;
}
```

---

### 5.2.3 Queues and producer/consumer systems

Queues are also used in signal processing and event processing. Queues are used whenever data may arrive and depart at somewhat unpredictable times, or when variable amounts of data may arrive. A queue is often referred to as an **elastic buffer**. We saw how to use elastic buffers for I/O in [Chapter 3](#).

One way to build a queue is with a linked list. This approach allows the queue to grow to an arbitrary size. However, in many applications, we are unwilling to pay the price of dynamically allocating memory. Another way to design the queue is to use an array to hold all of the data. Although some writers use both circular buffer and queue to mean the same thing, we use the term *circular buffer* to refer to a buffer that always has a fixed number of data elements, while a *queue* may have varying numbers of elements.

Programming Example 5.5 gives the C code for a queue that is built from an array.

#### Programming Example 5.5: An Array-Based Queue

The first step in designing the queue is to declare the array that we will use for the buffer:

```
#define Q_SIZE 5 /*your queue size may vary */
#define Q_MAX (Q_SIZE-1) /*this is the maximum index value into the
array */

int q[Q_SIZE]; /*the array for our queue */
int head, tail; /*indexes for the current queue head and tail */
```

The variables `head` and `tail` keep track of the two ends of the queue.

Here is the initialization code for the queue:

```
void queue_init() {

    /*initialize the queue data structure */

    head = 0;
    tail = 0;
}
```

We initialize the `head` and `tail` to the same position. As we add a value to the tail of the queue, we increment `tail`. Similarly, when we remove a value from the `head`, we increment `head`. The value of `head` is always equal to the location of the first element of the queue (except when the queue is empty). The value of `tail`, in contrast, points to the location in which the next queue entry will go. When we reach the end of the array, we must wrap around these values; for example, when we add a value into the last element of `q`, the new value of `tail` becomes the zeroth entry of the array.

We need to check for two error conditions: removing from an empty queue and adding to a full queue. In the first case, we know the queue is empty if `head == tail`. In the second case, we know the queue is full if incrementing `tail` will cause it to equal `head`. Testing for fullness, however, is a little harder because we must worry about wraparound.

Here is the code for adding an element to the tail of the queue, which is known as **enqueueing**:

```
void enqueue(int val) {
    /*check for a full queue */
    if (((tail+1) % Q_SIZE) == head) error("enqueue onto full
queue",tail);
    /*add val to the tail of the queue */
    q[tail] = val;
    /*update the tail */
    if (tail == Q_MAX)
        tail = 0;
    else
        tail++;
}
```

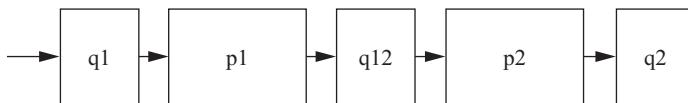
And here is the code for removing an element from the head of the queue, known as **dequeueing**:

```
int dequeue() {
    int returnval; /*use this to remember the value that you will
return */
    /*check for an empty queue */
    if (head == tail) error("dequeue from empty queue",head);
    /*remove from the head of the queue */
    returnval = q[head];
    /*update head */
    if (head == Q_MAX)
        head = 0;
    else
        head++;
    /*return the value */
    return returnval;
}
```

Digital filters always take in the same amount of data in each time period. Many systems, even signal processing systems, don't fit that mold. Rather, they may take in varying amounts of data over time and produce varying amounts. When several of these systems operate in a chain, the variable-rate output of one stage becomes the variable-rate input of another stage.

#### Producer/consumer

[Fig. 5.3](#) shows a block diagram of a simple **producer/consumer system**. p1 and p2 are the two blocks that perform algorithmic processing. The data are fed to them by queues that act as elastic buffers. The queues modify the flow of control in the system and store data. If, for example, p2 runs ahead of p1, it will eventually run out of data in its q12 input queue. At that point, the queue will return an empty signal to p2. At this point, p2 should stop working until more data are available. This sort of complex control is easier to implement in a multitasking environment, as we will see in [Chapter 6](#),

**FIGURE 5.3**

A producer/consumer system.

but it is also possible to make effective use of queues in programs that are structured as nested procedures.

#### Data structures in queues

The queues in a producer/consumer may hold either uniform-sized data elements or variable-sized data elements. In some cases, the consumer needs to know how many of a given type of data element are associated together. The queue can be structured to hold a complex data type. Alternatively, the data structure can be stored as bytes or integers in the queue with, for example, the first integer holding the number of successive data elements.

---

## 5.3 Models of programs

In this section, we develop models for programs that are more general than source code. Why not use the source code directly? First, there are many different types of source code, such as assembly languages and C code, but we can use a single model to describe all of them. Once we have such a model, we can perform many useful analyses on the model more easily than we could on the source code.

Our fundamental model for programs is the control/data flow graph (**CDFG**). We can also model hardware behavior with the CDFG. As the name implies, the CDFG has constructs that model both data operations (arithmetic and other computations) and control operations (conditionals). Part of the power of the CDFG comes from its combination of control and data constructs. To understand the CDFG, we start with pure data descriptions, and then, extend the model to control.

### 5.3.1 Data flow graphs

A **data flow graph** is a model of a program with no conditionals. In a high-level programming language, a code segment with no conditionals—more precisely, with only one entry and exit point—is known as a basic block. Fig. 5.4 shows a simple basic block. As the C code is executed, we would enter this basic block at the beginning and execute all of the statements.

Before we are able to draw the data flow graph for this code, we need to modify it slightly. There are two assignments to the variable `x`; it appears twice on the left side of an assignment. We need to rewrite the code in **single-assignment form**, in which a variable appears only once on the left side. Because our specification is C code, we

```
w = a + b;
x = a - c;
y = x + d;
x = a + c;
z = y + e;
```

**FIGURE 5.4**

A basic block in C.

```
w = a + b;
x1 = a - c;
y = x1 + d;
x2 = a + c;
z = y + e;
```

**FIGURE 5.5**

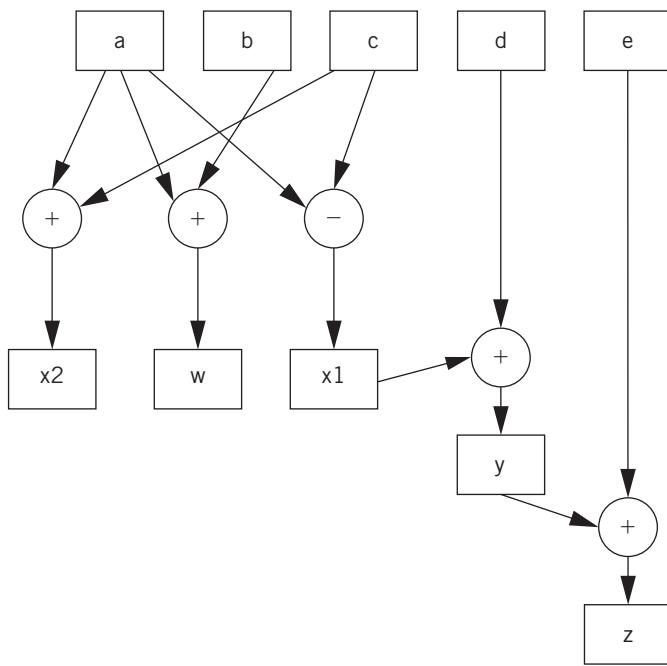
The basic block in single-assignment form.

assume that the statements are executed sequentially, so that any use of a variable refers to its latest assigned value. In this case,  $x$  is not reused in this block (it is presumably used elsewhere), so we must eliminate the multiple assignments to  $x$ . The result is shown in Fig. 5.5, where we have used the names  $x_1$  and  $x_2$  to distinguish the separate uses of  $x$ .

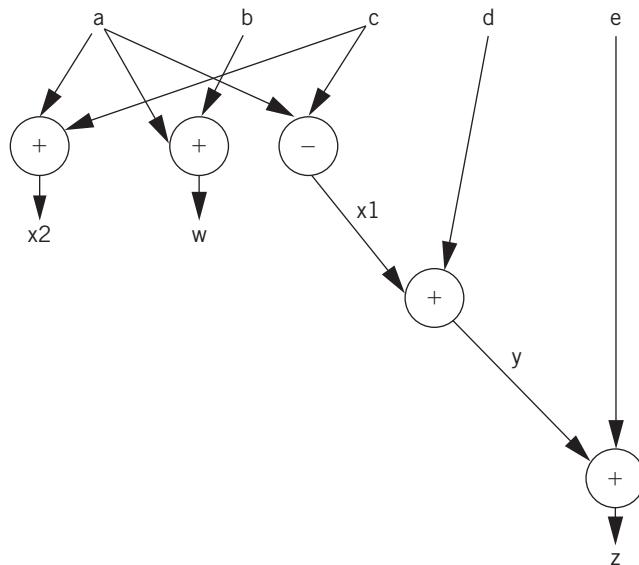
The single-assignment form is important because it allows us to identify a unique location in the code where each named location is computed. As an introduction to the data flow graph, we use two types of nodes in the graph: round nodes denote operators and square nodes represent values. The value nodes may be either inputs to the basic block, such as  $a$  and  $b$ , or variables that are assigned to within the block, such as  $w$  and  $x_1$ . The data flow graph for our single-assignment code is shown in Fig. 5.6. The single-assignment form means that the data flow graph is acyclic; if we assigned to  $x$  multiple times, the second assignment would form a cycle in the graph including  $x$  and the operators used to compute  $x$ . Keeping the data flow graph acyclic is important in many types of analyses that we want to do on the graph. Of course, it is important to know whether the source code assigns to a variable multiple times, because some of those assignments may be mistakes. We consider the analysis of source code for proper use of assignments in Section 5.5.

The data flow graph is generally drawn in the form that is shown in Fig. 5.7. Here, the variables are not explicitly represented by nodes. Instead, the edges are labeled with the variables that they represent. As a result, a variable can be represented by more than one edge. However, the edges are directed and all of the edges for a variable must come from a single source. We use this form for its simplicity and compactness.

The data flow graph for the code makes the order in which the operations are performed in the C code much less obvious. This is one of the advantages of the data flow graph. We can use it to determine feasible reorderings of the operations, which may help us to reduce pipeline or cache conflicts. We can also use it when the exact order of operations simply doesn't matter. The data flow graph defines a partial ordering of the operations in the basic block. We must ensure that a value is computed before it is used, but generally, there are several possible orderings of evaluating expressions that satisfy this requirement.

**FIGURE 5.6**

An extended data flow graph for our sample basic block.

**FIGURE 5.7**

Standard data flow graph for our sample basic block.

### 5.3.2 Control/data flow graphs

A CDFG uses a data flow graph as an element, adding constructs to describe control. In a basic CDFG, we have two types of nodes: **decision nodes** and **data flow nodes**. A data flow node encapsulates a complete data flow graph to represent a basic block. We can use one type of decision node to describe all types of control in a sequential program. The jump/branch is, after all, the way that we implement all of those high-level control constructs.

[Fig. 5.8](#) shows a portion of C code with control constructs and the CDFG constructed from it. The rectangular nodes in the graph represent the basic blocks. The basic blocks in the C code have been represented by function calls for simplicity. The diamond-shaped nodes represent the conditionals. The node's condition is given by the label, and the edges are labeled with the possible outcomes of evaluating the condition.

Building a CDFG for a `while` loop is straightforward, as shown in [Fig. 5.9](#). The `while` loop consists of both a test and a loop body, each of which we know how to represent in a CDFG. We can represent `for` loops by remembering that, in C, a `for` loop is defined in terms of a `while` loop [Ker88]. This `for` loop

```
for (i = 0; i < N; i++) {
    loop_body();
}
```

is equivalent to

```
i = 0;
while (i < N) {
    loop_body();
    i++;
}
```

#### Hierarchical representation

For a complete CDFG model, we can use a data flow graph to model each data flow node. Thus, the CDFG is a hierarchical representation; a data flow CDFG can be expanded to reveal a complete data flow graph.

An execution model for a CDFG is very much like the execution of the program that it represents. The CDFG does not require the explicit declaration of variables and we assume that the implementation has sufficient memory for all variables. We can define a state variable that represents a program counter in a CPU. (When studying a drawing of a CDFG, a finger works well for keeping track of the program counter state.) As we execute the program, we either execute the data flow node or compute the decision in the decision node and follow the appropriate edge, depending on the type of node that the program counter points to. Even though the data flow nodes may specify only a partial ordering on the data flow computations, the CDFG is a sequential representation of the program. There is only one program counter in our execution model of the CDFG, and operations are not executed in parallel.

```

if (cond1)
    basic_block_1();
else
    basic_block_2();
basic_block_3();
switch (test1) {
    case c1: basic_block_4(); break;
    case c2: basic_block_5(); break;
    case c3: basic_block_6(); break;
}

```

C code

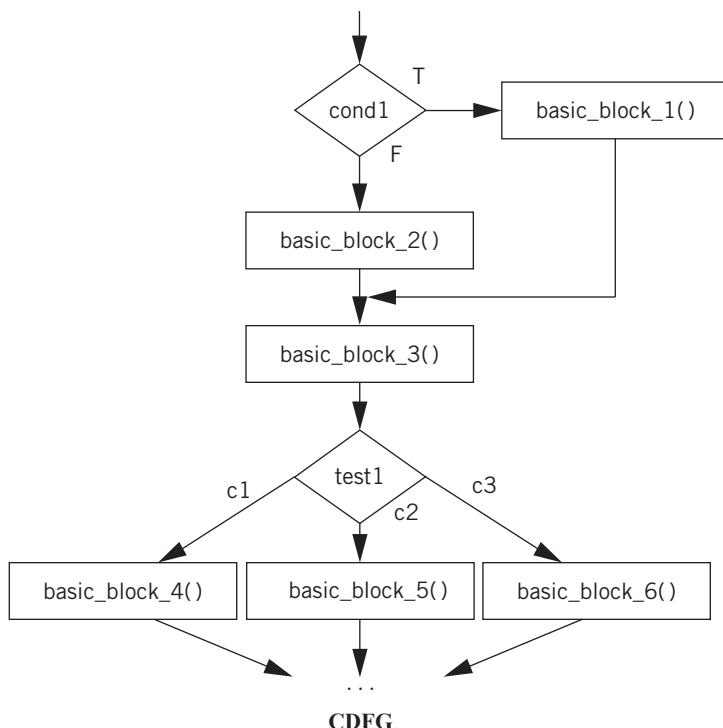
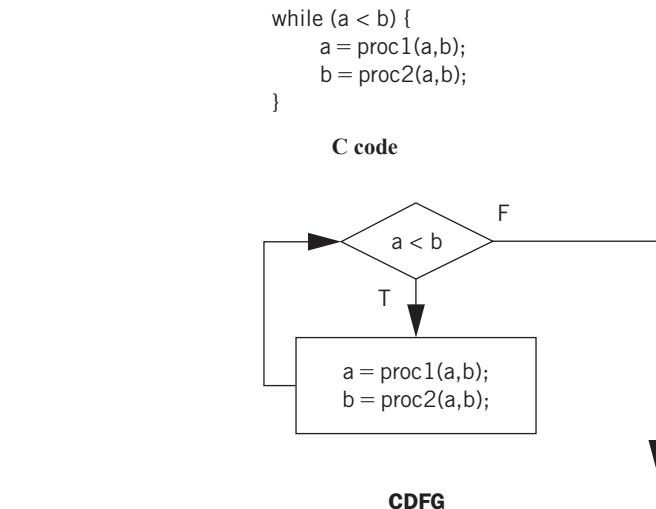


FIGURE 5.8

C code and its control/data flow graph.

The CDFG is not necessarily tied to high-level language control structures. We can also build a CDFG for an assembly language program. A jump instruction corresponds to a nonlocal edge in the CDFG. Some architectures, such as ARM and many VLIW processors, support the predicated execution of instructions, which may be represented by special constructs in the CDFG.

**FIGURE 5.9**

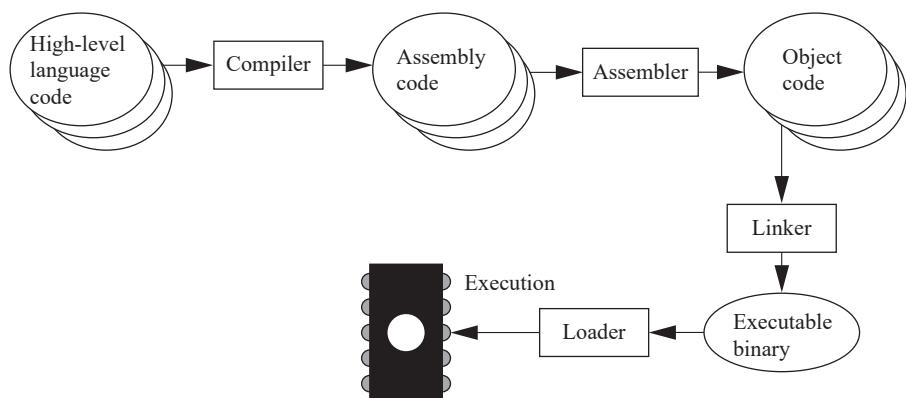
A while loop and its CDFG.

## 5.4 Assembly, linking, and loading

Assembly and linking are the last steps in the compilation process. They change a list of instructions into an image of the program's bits in memory. Loading puts the program into memory so that it can be executed. In this section, we survey the basic techniques required for assembly linking to help us to understand the complete compilation and loading process.

### Program generation workflow

[Fig. 5.10](#) highlights the role of assemblers and linkers in the compilation process. This process is often hidden from us by compilation commands that do everything that is required to generate an executable program. As the figure shows, most compilers do not directly generate machine code, but instead create the instruction-level program in the form of human-readable assembly language. Generating assembly language rather than binary instructions frees the compiler writer from details that are extraneous to the compilation process, including the instruction format as well as the exact addresses of instructions and data. The assembler's job is to translate symbolic assembly language statements into bit-level representations of instructions that are known as **object code**. The assembler takes care of instruction formats and does part of the job of translating labels into addresses. However, because the program may be built from many files, the final steps in determining the addresses of instructions and data are performed by the linker, which produces an **executable binary** file. However, this file may not necessarily be in the CPU's memory, unless the linker happens to

**FIGURE 5.10**

Program generation from compilation through loading.

create the executable directly in RAM. The program that brings the program into memory for execution is called a **loader**.

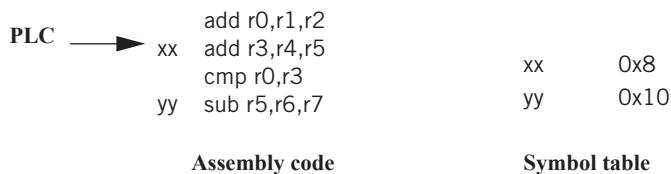
#### Absolute and relative addresses

The simplest form of the assembler assumes that the starting address of the assembly language program has been specified by the programmer. The addresses in such a program are known as **absolute addresses**. However, in many cases, particularly when we are creating an executable out of several component files, we do not want to specify the starting addresses for all of the modules before assembly. If we did, we would have to determine not only the length of each program in memory, but also the order in which they would be linked into the program before assembly. Therefore, most assemblers allow us to use **relative addresses** by specifying at the start of the file that the origin of the assembly language module is to be computed later. Addresses within the module are then computed relative to the start of the module, and the linker is responsible for translating relative addresses into addresses.

### 5.4.1 Assemblers

When translating assembly code into object code, the assembler must translate op-codes, format the bits in each instruction, and translate labels into addresses. In this section, we review the translation of assembly language into binary.

Labels make the assembly process more complex, but they are the most important abstraction provided by the assembler. Labels let the programmer (a human programmer or a compiler generating assembly code) avoid worrying about the locations of instructions and data. Label processing requires making two passes through the assembly source code:

**FIGURE 5.11**

Symbol table processing during assembly.

1. The first pass scans the code to determine the address of each label.
2. The second pass assembles the instructions using the label values computed in the first pass.

#### Symbol table

As shown in Fig. 5.11, the name of each symbol and its address are stored in a **symbol table** that is built during the first pass. The symbol table is built by scanning from the first instruction to the last. For the moment, we assume that we know the address of the first instruction in the program. During scanning, the current location in memory is kept in a **program location counter (PLC)**. Despite the similarity in name to the program counter, the PLC is not used to execute the program, but only to assign memory locations to labels. For example, the PLC always makes exactly one pass through the program, whereas the program counter makes many passes over code in a loop. Thus, at the start of the first pass, the PLC is set to the program's starting address and the assembler looks at the first line. After examining the line, the assembler updates the PLC to the next location (because ARM instructions are four bytes long, the PLC would be incremented by four) and looks at the next instruction. If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC. At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass. During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the instruction.

However, how do we know the starting value of the PLC? The simplest case is addressing. In this case, one of the first statements in the assembly language program is a pseudo-op that specifies the **origin** of the program; that is, the location of the first address in the program. A common name for this pseudo-op (e.g., the one used for the ARM) is the **ORG** statement

```
ORG 2000
```

which puts the start of the program at location 2000. This pseudo-op accomplishes this by setting the PLC's value to its argument's value, which is 2000 in this case. Assemblers generally allow a program to have many **ORG** statements in case the instructions or data must be spread around various spots in memory.

Example 5.1 illustrates the use of the PLC in generating the symbol table.

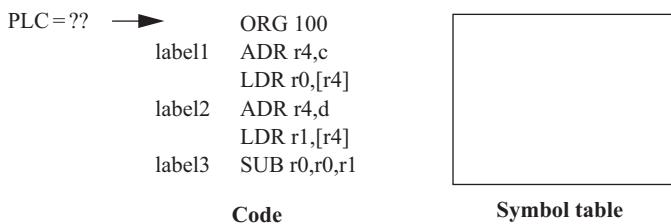
---

### Example 5.1: Generating a Symbol Table

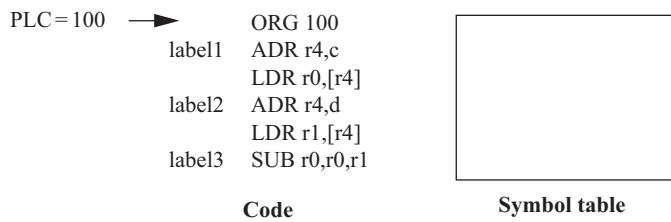
Let's use the following simple example of ARM assembly code:

```
ORG 100
label1 ADR r4,c
        LDR r0,[r4]
label2 ADR r4,d
        LDR r1,[r4]
label3 SUB r0,r0,r1
```

The initial ORG statement tells us the starting address of the program. To begin, let's initialize the symbol table to an empty state and put the PLC at the initial ORG statement:

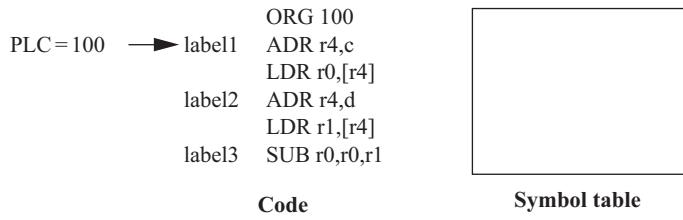


The PLC value shown is at the beginning of this step before we have processed the ORG statement. The ORG tells us to set the PLC value to 100.



To process the next statement, we move the PLC to point to the next statement. However, because the last statement was a pseudo-op that generates no memory values, the PLC value remains at 100.

Because there is a label in this statement, we add it to the symbol table, taking its value from the current PLC value.



To process the next statement, we advance the PLC to point to the next line of the program and increment its value by the length in memory of the last line, namely 4.

Code	Symbol table
PLC = 104 → label1 ORG 100 label1 ADR r4,c label1 LDR r0,[r4] label2 ADR r4,d label2 LDR r1,[r4] label3 SUB r0,r0,r1	label1 100

We continue this process as we scan the program until we reach the end, at which the state of the PLC and symbol table are as shown below.

Code	Symbol table
PLC = 116 → label3 label1 ORG 100 label1 ADR r4,c label1 LDR r0,[r4] label2 ADR r4,d label2 LDR r1,[r4] label3 SUB r0,r0,r1	label1 100 label2 108 label3 116

Assemblers allow labels to be added to the symbol table without occupying space in the program memory. A typical name for this pseudo-op is EQU for equate. For example, in the code,

```

ADD r0,r1,r2
FOO EQU 5
BAZ SUB r3,r4,#FOO
  
```

the EQU pseudo-op adds a label named FOO with value 5 to the symbol table. The value of the BAZ label is the same as if the EQU pseudo-op were not present, because EQU does not advance the PLC. The new label is used in the subsequent SUB instruction as the name for a constant. EQUs can be used to define symbolic values to help to make the assembly code more structured.

#### ARM ADR pseudo-op

The ARM assembler supports one pseudo-op that is particular to the ARM instruction set. In other architectures, an address would be loaded into a register (e.g., for an indirect access) by reading it from a memory location. ARM does not have an instruction that can load an effective address, so the assembler supplies the ADR pseudo-op to create the address in the register. It does so by using ADD or SUB instructions to generate the address. The address to be loaded can be register relative, program relative, or

**Object code formats**

numeric, but it must assemble to a single instruction. More complicated address calculations must be programmed explicitly.

The assembler produces an object file that describes the instructions and data in binary format. A commonly used object file format, which was originally developed for Unix but is now used in other environments as well, is known as the Common Object File Format (COFF). The object file must describe the instructions, data, and any addressing information, and also usually carries along the symbol table for later use in debugging.

Generating relative code rather than code introduces some new challenges into the assembly language process. Rather than using an `ORG` statement to provide the starting address, the assembly code uses a pseudo-op to indicate that the code is in fact relocatable. Relative code is the default for the ARM assembler. Similarly, we must mark the output object file as being relative code. We can initialize the PLC to 0 to ensure that addresses are relative to the start of the file. However, we must be careful when we generate code that makes use of those labels, because we do not yet know the actual value that must be put into the bits. Instead, we must generate relocatable code. We use extra bits in the object file format to mark the relevant fields as relocatable, and then, insert the label's relative value into the field. The linker must therefore modify the generated code; when it finds a field that is marked as relative, it uses the addresses that it has generated to replace the relative value with a correct value for the address. To understand the details of changing relocatable code into executable code, we must understand the linking process that is described in the next section.

### 5.4.2 Linking

Many assembly language programs are written as several smaller pieces rather than as a single large file. Breaking a large program into smaller files helps to delineate program modularity. If the program uses library routines, those will already be preassembled, and assembly language source code for the libraries may not be available for purchase. A **linker** allows a program to be stitched together out of several smaller pieces. The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files.

Some labels will be both defined and used in the same file. Other labels will be defined in a single file but used elsewhere, as illustrated in Fig. 5.12. The place in the file where a label is defined is known as an **entry point**. The place in the file where the label is used is called an **external reference**. The main job of the loader is to *resolve* external references based on available entry points. As a result of the need to know how definitions and references connect, the assembler passes not only the object file, but also the symbol table, to the linker. Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points. External references are identified in the object code by their relative symbol identifiers.

**Linking process**

The linker proceeds in two phases. First, it determines the address of the start of each object file. The order in which object files are to be loaded is given by the user,

label1	LDR r0,[r1]	label2	ADR var1
...	...	...	...
	ADR a		B label3
	...		...
	B label2	x	% 1
	...	y	% 1
var1	% 1	a	% 10

External references	Entry points
a	label1
label2	var1

File 1

External references	Entry points
var1	label2
label3	x
	y
	a

File 2

**FIGURE 5.12**

External references and entry points.

either by specifying parameters when the loader is run or by creating a **load map** file that gives the order in which the files are to be placed in memory. Given the order in which files are to be placed in memory and the length of each object file, it is easy to compute the starting address of each file. At the start of the second phase, the loader merges all symbol tables from the object files into a single large table. It then edits the object files to change relative addresses into addresses. This is typically performed by having the assembler write extra bits into the object file to identify the instructions and fields that refer to labels. If a label cannot be found in the merged symbol table, it is undefined, and an error message is sent to the user.

Controlling where code modules are loaded into memory is important in embedded systems. Some data structures and instructions, such as those used to manage interrupts, must be put at precise memory locations for them to work. In other cases, different types of memory may be installed at different address ranges. For example, if we have flash in some locations and DRAM in others, we want to make sure that the locations to be written are placed in the DRAM locations.

Workstations and PCs provide **dynamically linked libraries**, and certain sophisticated embedded computing environments may provide them as well. Rather than link a separate copy of commonly used routines such as I/O to every executable program on the system, dynamically linked libraries allow them to be linked in at the start of program execution. A brief linking process is run just before the execution of the program begins; the dynamic linker uses code libraries to link in the required routines. This not only saves storage space, but also allows the programs that use those libraries

### Dynamically linked libraries

to be easily updated. However, it does introduce a delay before the program starts executing.

### 5.4.3 Object code design

We have to take several issues into account when designing object code. In a timesharing system, many of these details are taken care of for us. When designing an embedded system, we may need to handle some of them ourselves.

#### Memory map design

As we have seen, the linker allows us to control where object code modules are placed in memory. We may need to control the placement of several types of data:

- Interrupt vectors and other information for I/O devices must be placed in specific locations.
- Memory management tables must be set up.
- Global variables that are used for communication between processes must be put in locations that are accessible to all users of those data.

We can give these locations symbolic names so that, for example, the same software can work on different processors that put these items at different addresses. However, the linker must be given the proper absolute addresses to configure the program's memory.

#### Reentrancy

Many programs should be designed to be **reentrant**. A program is reentrant if can be interrupted by another call to the function without changing the results of either call. If the program changes the values of global variables, it may give a different answer when it is called recursively. Consider this code:

```
int foo = 1;

int task1() {
    foo = foo + 1;
    return foo;
}
```

In this simple example, the variable `foo` is modified; thus, `task1()` gives a different answer on every invocation. We can avoid this problem by passing `foo` in as an argument:

```
int task1(int foo) {
    return foo+1;
}
```

---

## 5.5 Compilation techniques

Although we don't write our own assembly code in most cases, we still care about the characteristics of the code that our compiler generates: its speed, its size, and its power consumption. Understanding how a compiler works will help us to write code and

direct the compiler to get the assembly language implementation that we want. We will start with an overview of the compilation process, followed by some basic compilation methods, and conclude with some more advanced optimizations.

### 5.5.1 The compilation process

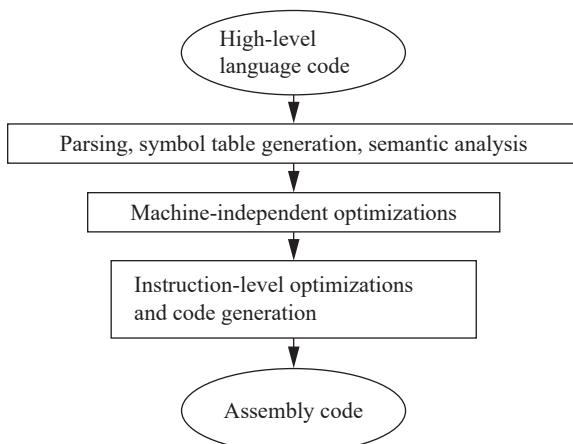
It is useful to understand how a high-level language program is translated into instructions, interrupt handling instructions, place data and instructions in memory, and so on. Understanding how the compiler works can help you to know when you cannot rely on the compiler. Next, because many applications are also performance sensitive, understanding how code is generated can help you to meet your performance goals, either by writing high-level code that gets compiled into the instructions you want or by recognizing when you must write your own assembly code.

We can summarize the compilation process with a formula:

$$\text{compilation} = \text{translation} + \text{optimization}$$

The high-level language program is translated into the lower-level form of instructions; optimizations try to generate better instruction sequences than would be possible if the brute force technique of independently translating source code statements were used. Optimization techniques focus on more of the program to ensure that compilation decisions that appear to be good for one statement are not necessarily problematic for other parts of the program.

The compilation process is outlined in Fig. 5.13. Compilation begins with high-level language code such as C or C++ and generally produces assembly code. Directly producing object code simply duplicates the functions of an assembler, which is a very desirable standalone program to have. The high-level language program is



**FIGURE 5.13**

The compilation process.

parsed to break it into statements and expressions. In addition, a symbol table is generated, which includes all of the named objects in the program. Some compilers may then perform higher-level optimizations that can be viewed as modifying the high-level language program input without reference to instructions.

Simplifying arithmetic expressions is one example of a machine-independent optimization. Not all compilers do such optimizations, and compilers can vary widely regarding which combinations of machine-independent optimizations they do perform. Instruction-level optimizations are aimed at generating code. They may work directly on real instructions or on a pseudo-instruction format that is later mapped onto the instructions of the target CPU. This level of optimization also helps to modularize the compiler by allowing code generation to create simpler code that is later optimized. For example, consider this array access code:

```
x[i] = c*x[i];
```

A simple code generator would generate the address for `x[i]` twice: once for each appearance in the statement. The later optimization phases can recognize this as an example of common expressions that need not be duplicated. While it would be possible to create a code generator that never generated the redundant expression in this simple case, taking into account every such optimization at code generation time is very difficult. We get better code and more reliable compilers by generating simple code first, and then, optimizing it.

## 5.5.2 Basic compilation methods

### Statement translation

### Procedures

In this section, we consider the basic job of translating a high-level language program with little or no optimization.

Procedures (or functions, as they are known in C) require specialized code: the code that is used to call and pass parameters is known as **procedure linkage**; procedures also need a way to store local variables. Generating code for procedures is relatively straightforward once we know the procedure linkage that is appropriate for the CPU. At the procedure definition, we generate the code to handle the procedure call and return. At each call of the procedure, we set up the procedure parameters and make the call.

The CPU's subroutine call mechanism is usually not sufficient to support procedures directly in modern programming languages. We introduced the procedure stack and procedure linkages in Section 2.3.3. The linkage mechanism provides a way for the program to pass parameters into the program and for the procedure to return a value. It also provides help in restoring the values of registers that the procedure has modified. All procedures in a given programming language use the same linkage mechanism (although different languages may use different linkages). The mechanism can also be used to call handwritten assembly language routines from compiled code.

The information for a call to a procedure is known as a **frame**; the frames are stored on a stack to keep track of the order in which the procedures have been called.

Procedure stacks are typically built to grow down from high addresses. A **stack pointer** (sp) defines the end of the current frame, whereas a **frame pointer** (fp) defines the end of the last frame. The fp is technically necessary only if the stack frame can be grown by the procedure during execution. The procedure can refer to an element in the frame by addressing relative to sp. When a new procedure is called, the sp and fp are modified to push another frame onto the stack. In addition to allowing parameters and return values to be passed, the frame also holds the locally declared variables. When accessing a local variable, the compiled code must do so by referencing a location within the frame, which requires it to perform address arithmetic.

As we saw in [Chapter 2](#), the ARM Procedure Call Standard (APCS) [Slo04] is the recommended procedure linkage for ARM processors. r0–r3 are used to pass the first four parameters into the procedure. r0 is also used to hold the return value.

The next example looks at compiler-generated procedure linkage code.

### Programming Example 5.6: Procedure Linkage in C

Here is a procedure definition:

```
int p1(int a, int b, int c, int d, int e) {
    return a + e;
}
```

This procedure has five parameters, so we would expect that one of them would be passed through the stack while the rest are passed through registers. It also returns an integer value, which should be returned in r0. Here is the code for the procedure generated by the ARM gcc compiler with some handwritten comments:

```
mov    ip, sp          ; procedure entry
stmfd sp!, {fp, ip, lr, pc}
sub    fp, ip, #4
sub    sp, sp, #16
str    r0, [fp, #-16]   ; put first four args on stack
str    r1, [fp, #-20]
str    r2, [fp, #-24]
str    r3, [fp, #-28]
ldr    r2, [fp, #-16]   ; load a
ldr    r3, [fp, #4]      ; load e
add    r3, r2, r3       ; compute a + e
mov    r0, r3           ; put the result into r0 for return
ldmea fp, {fp, sp, pc} ; return
```

Here is a call to that procedure:

```
y = p1(a,b,c,d,x);
```

Here is the ARM gcc code with handwritten comments:

```
ldr    r3, [fp, #-32]   ; get e
str    r3, [sp, #0]      ; put into p1()'s stack frame
ldr    r0, [fp, #-16]   ; put a into r0
ldr    r1, [fp, #-20]   ; put b into r1
```

```

ldr r2, [fp, #-24] ; put c into r2
ldr r3, [fp, #-28] ; put d into r3
bl p1; call p1()
mov r3, r0          ; move return value into r3
str r3, [fp, #-36] ; store into y in stack frame

```

We can see that the compiler sometimes makes additional register moves but it does follow the APCS standard.

A large amount of the code in a typical application consists of arithmetic and logical expressions. Understanding how to compile a single expression, as described in the next example, is a good first step in understanding the entire compilation process.

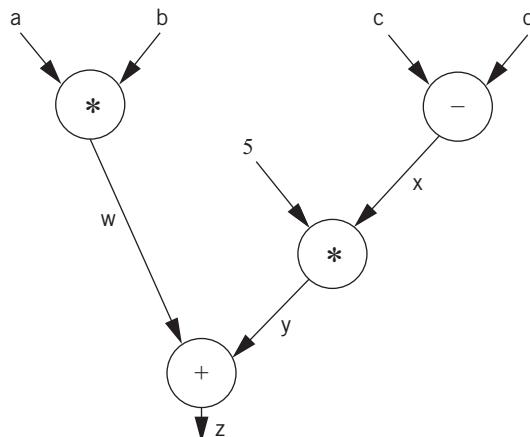
### Example 5.2: Compiling an Arithmetic Expression

Consider this arithmetic expression:

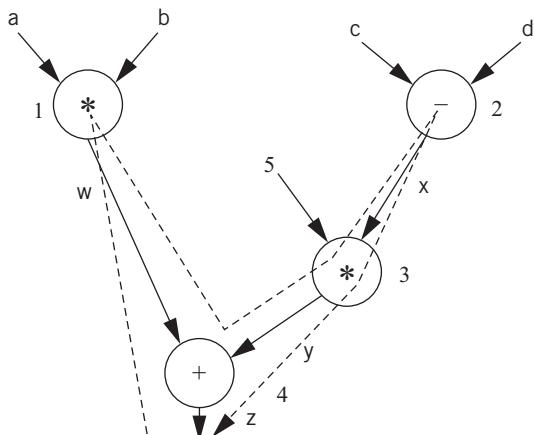
$$x = a * b + 5 * (c - d)$$

The expression is written in terms of program variables. In some machines, we may be able to perform memory-to-memory arithmetic directly on the locations corresponding to those variables. However, in many machines, such as the ARM, we must first load the variables into registers. This requires selecting which registers receive not only the named variables, but also intermediate results such as  $(c - d)$ .

The code for the expression can be built by walking the data flow graph. Here is the data flow graph for the expression:



The temporary variables for the intermediate values and final result have been named  $w$ ,  $x$ ,  $y$ , and  $z$ . To generate code, let's walk from the tree's root (where  $z$ , the final result, is generated) by traversing the nodes in post order. During the walk, we generate instructions to cover the operation at every node. Here is the path:



The nodes are numbered in the order in which code is generated. Because every node in the data flow graph corresponds to an operation that is directly supported by the instruction set, we simply generate an instruction at every node. Because we are making an arbitrary register assignment, we can use up the registers in order, starting with r1. Here is the resulting ARM code:

```

; operator 1 (*)
ADR r4,a           ; get address for a
MOV r1,[r4]          ; load a
ADR r4,b           ; get address for b
MOV r2,[r4]          ; load b
ADD r3,r1,r2        ; put w into r3
; operator 2 (-)
ADR r4,c           ; get address for c
MOV r4,[r4]          ; load c
ADR r4,d           ; get address for d
MOV r5,[r4]          ; load d
SUB r6,r4,r5        ; put z into r6
; operator 3 (*)
MUL r7,r6,#5        ; operator 3, puts y into r7
; operator 4 (+)
ADD r8,r7,r3        ; operator 4, puts x into r8
; assign to x
ADR r1,x            ; assigns to x location
STR r8,[r1]          ; assigns to x location

```

One obvious optimization is to reuse a register whose value is no longer needed. In the case of the intermediate values  $w$ ,  $y$ , and  $z$ , we know that they cannot be used after the end of the expression (e.g., in another expression) because they have no name in the C program. However, the final result  $z$  may in fact be used in a C assignment, and the value can be reused later in the program. In this case, we would need to know when the register is no longer needed to determine its best use.

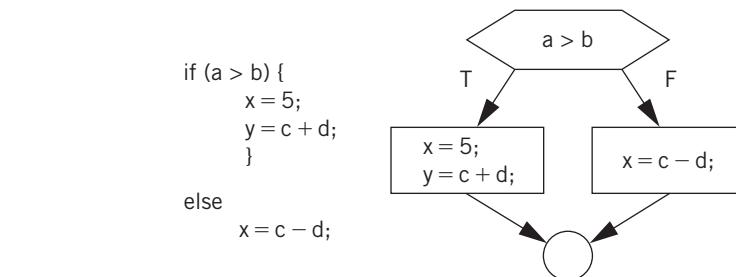
For comparison, here is the code generated by the ARM gcc compiler with handwritten comments:

```
ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
mul r1, r3, r2          ; multiply
ldr r2, [fp, #-24]
ldr r3, [fp, #-28]
rsb r2, r3, r2          ; subtract
mov r3, r2
mov r3, r3, asl #2
add r3, r3, r2          ; add
add r3, r1, r3          ; add
str r3, [fp, #-32]      ; assign
```

In the previous example, we made an arbitrary allocation of variables to registers for simplicity. When we have large programs with multiple expressions, we must allocate registers more carefully, because CPUs have a limited number of registers. We will consider register allocation in more detail below.

We also need to be able to translate control structures. Because conditionals are controlled by expressions, the code generation techniques of the last example can be used for those expressions, leaving us with the task of generating code for the flow of control itself. Fig. 5.14 shows a simple example of changing the flow of control in C—an if statement, in which the condition controls whether the true or false branch of the if is taken. Fig. 5.14 also shows the control flow diagram for the if statement.

The next example illustrates how to implement conditionals in assembly language.



**FIGURE 5.14**

Flow of control in C and control flow diagrams.

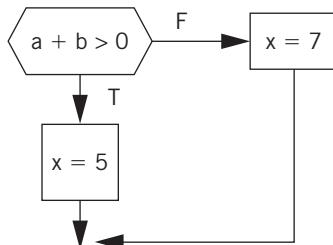
---

**Example 5.3: Generating Code for a Conditional**

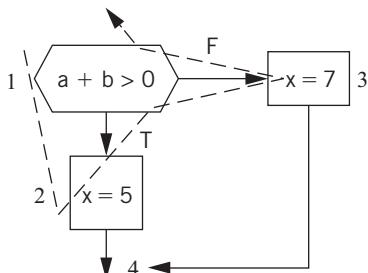
Consider this C statement:

```
if (a + b > 0)
    x = 5;
else
    x = 7;
```

The CDFG for the statement appears below.



We know how to generate the code for the expressions. We can generate the control flow code by walking the CDFG. One ordered walk through the CDFG follows:



To generate code, we must assign a label to the first instruction at the end of a directed edge and create a branch for each edge that does not go to the immediately following instruction. The exact steps to be taken at the branch points depend on the target architecture. On some machines, evaluating expressions generates condition codes that we can test in subsequent branches, and on other machines, we must use test-and-branch instructions. ARM allows us to test condition codes, so we get the following ARM code for the 1–2–3 walk:

```

ADR r5,a      ; get address for a
LDR r1,[r5]   ; load a
ADR r5,b      ; get address for b
LDR r2,b      ; load b
ADD r3,r1,r2
BLE label3    ; true condition falls through branch
; true case
    LDR r3,#5    ; load constant
    ADR r5,x
    STR r3,[r5]  ; store value into x
    B stmtend    ; done with the true case

```

```

; false case
label3 LDR r3,#7    ; load constant
        ADR r5,x      ; get address of x
        STR r3,[r5]   ; store value into x
stmtend

```

The 1–2 and 3–4 edges do not require a branch and label because they are straight-line code. In contrast, the 1–3 and 2–4 edges require a branch and a label for the target.

For comparison, here is the code generated by the ARM gcc compiler with some hand-written comments:

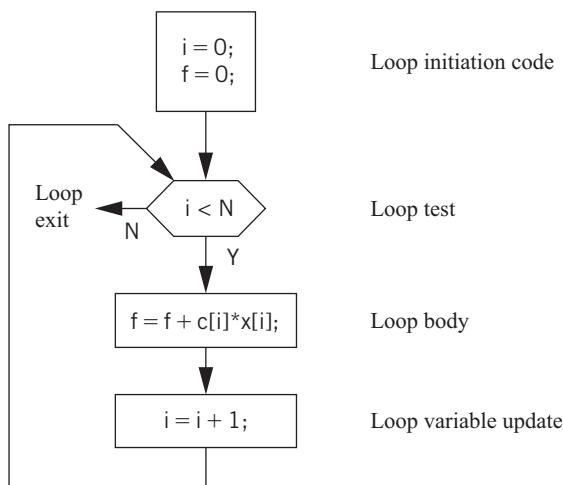
```

ldr r2, [fp, #-16]
ldr r3, [fp, #-20]
add r3, r2, r3
cmp r3, #0          ; test the branch condition
ble .L3             ; branch to false block if <=
mov r3, #5          ; true block
str r3, [fp, #-32]
b .L4              ; go to end of if statement
.L3:               ; false block
    mov r3, #7
    str r3, [fp, #-32]
.L4:               ; true block

```

Because expressions are generally created as straight-line code, they typically require careful consideration of the order in which the operations are executed. We have much more freedom when generating conditional code, because the branches ensure that the flow of control goes to the correct block of code. If we walk the CDFG in a different order and lay out the code blocks in a different order in memory, we still get valid code as long as we place the branches properly.

Drawing a control flow graph based on the while form of the loop helps us to understand how to translate it into instructions:



C compilers can generate assembler source (using the `-s` flag), which some compilers inter-splice with the C code. Such code is a very good way to learn about both assembly language programming and compilation.

### Data structures

The compiler must also translate references to data structures into references to raw memories. In general, this requires address computations. Some of these computations can be done at compile time, whereas others must be done at run time.

Arrays are interesting because the address of an array element must be computed at run time in general, because the array index may change. Let us first consider a one-dimensional array:

$$a[i]$$

The layout of the array in memory is shown in Fig. 5.15: the zeroth element is stored as the first element of the array, the first element directly below, and so on. We can create a pointer for the array that points to the array's head, namely  $a[0]$ . If we call that pointer `aptr` for convenience, we can rewrite the reading of  $a[i]$  as

$$\ast(aptr + i)$$

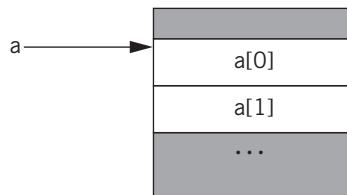
Two-dimensional arrays are more challenging. There are multiple possible ways to lay out a two-dimensional array in memory, as shown in Fig. 5.16. In this form, which is known as **row major**, the rightmost variable of the array ( $j$  in  $a[i][j]$ ) varies the most quickly. Fortran uses the opposite organization, known as *column major*, in which the leftmost array index varies the fastest. Two-dimensional arrays also require more sophisticated addressing; in particular, we must know the size of the array. Let us consider the row major form. If the  $a[][]$  array is of size  $M \times N$ , we can change the two-dimensional array access into a one-dimensional array access. Thus,

$$a[i][j]$$

becomes

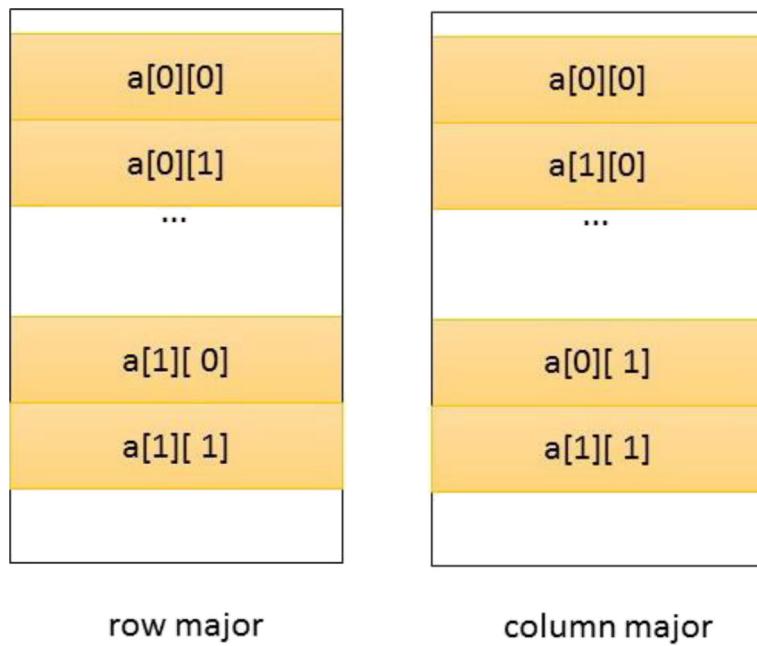
$$a[i \ast M + j]$$

A C struct is easier to address. As shown in Fig. 5.16, a structure is implemented as a contiguous block of memory. Fields in the structure can be accessed using constant



**FIGURE 5.15**

Layout of a one-dimensional array in memory.

**FIGURE 5.16**

Memory layout for two-dimensional arrays.

offsets to the base address of the structure. In this example, if field1 is four bytes long, field2 can be accessed as

`*(aptr + 4)`

This addition can usually be done at compile time, requiring only the indirection itself to fetch the memory location during execution.

### 5.5.3 Compiler optimizations

Basic compilation techniques can generate inefficient code. Compilers use a wide range of algorithms to optimize the code that they generate.

#### Inlining

**Function inlining** replaces a subroutine call to a function with equivalent code to the function body. By substituting the function call's parameters into the body, the compiler can generate a copy of the code that performs the same operations, but without the subroutine overhead. C++ provides an *inline* qualifier that allows the compiler to substitute an inline version of the function. In C, programmers can perform inlining manually or by using a preprocessor macro to define the code body.

**Outlining** is the opposite of inlining: a set of similar sections of code is replaced with calls to an equivalent function. Although inlining eliminates the function call overhead, it also increases the program size. Inlining also inhibits sharing of the

function code in the cache; because the inlined copies are distinct pieces of code, they cannot be represented by the same code in the cache. Outlining is sometimes useful to improve the cache behavior of common functions.

### Loop transformations

Loops are important program structures; although they are compactly described in the source code, they often use a large fraction of the computation time. Many techniques have been designed to optimize loops.

A simple but useful transformation is known as **loop unrolling**, as illustrated in the next example. Loop unrolling is important because it helps to expose parallelism that can be used by later stages of the compiler.

### Example 5.4: Loop Unrolling

Here is a simple C loop:

```
for (i = 0; i < N; i++) {
    a[i] = b[i] * c[i];
}
```

This loop is executed a fixed number of times, namely  $N$ . A straightforward implementation of the loop would create and initialize the loop variable  $i$ , update its value at every iteration, and test it to see whether to exit the loop. However, because the loop is executed a fixed number of times, we can generate more direct code.

If we let  $N = 4$ , then we can substitute this straight-line code for the loop:

```
a[0] = b[0]*c[0];
a[1] = b[1]*c[1];
a[2] = b[2]*c[2];
a[3] = b[3]*c[3];
```

This unrolled code has no loop overhead code at all; that is, no iteration variable and no tests. However, the unrolled loop has the same problems as the inlined procedure; it may interfere with the cache and expands the amount of code required.

We do not, of course, have to unroll loops fully. Rather than unroll the above loop four times, we could unroll it twice. Unrolling produces this code:

```
for (i = 0; i < 2; i++) {
    a[i*2] = b[i*2]*c[i*2];
    a[i*2 + 1] = b[i*2 + 1]*c[i*2 + 1];
}
```

In this case, because all operations in the two lines of the loop body are independent, later stages of the compiler may be able to generate code that allows them to be executed efficiently on the CPU's pipeline.

**Loop fusion** combines two or more loops into a single loop. For this transformation to be legal, two conditions must be satisfied. First, the loops must iterate over the same values. Second, the loop bodies must not have dependencies that would be violated if they are executed together; for example, if the second loop's  $i$ th iteration depends on the results of the  $i + 1$ th iteration of the first loop, the two loops cannot

be combined. **Loop distribution** is the opposite of loop fusion; that is, decomposing a single loop into multiple loops.

#### Dead code elimination

**Dead code** is code that can never be executed. Dead code can be generated by programmers, either inadvertently or purposefully. Dead code can also be generated by compilers. Dead code can be identified by **reachability analysis**, finding the other statements or instructions from which it can be reached. If a given piece of code cannot be reached or if it can be reached only by a piece of code that is unreachable from the main program, it can be eliminated. **Dead code elimination** analyzes code for reachability and trims away dead code.

#### Register allocation

**Register allocation** is a very important compilation phase. Given a block of code, we want to select assignments of variables (both declared and temporary) to registers to minimize the total number of required registers.

The next example illustrates the importance of proper register allocation.

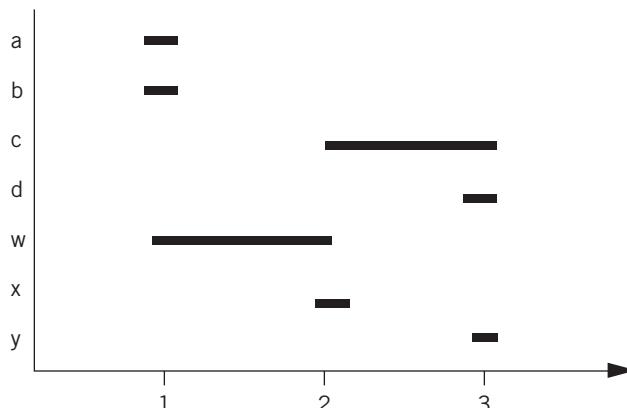
### Example 5.5: Register Allocation

To keep the example small, we assume that we can use only four of the ARM's registers. In fact, such a restriction is not unthinkable. Programming conventions can reserve certain registers for special purposes and significantly reduce the number of general-purpose registers that are available.

Consider this C code:

```
w = a + b; /*statement 1 */
x = c + w; /*statement 2 */
y = c + d; /*statement 3 */
```

A naive register allocation, assigning each variable to a separate register, would require seven registers for the seven variables in the above code. However, we can do much better by reusing a register once the value stored in the register is no longer needed. To understand how to do this, we can draw a **lifetime graph** that shows the statements on which each statement is used. Here is a lifetime graph in which the x axis is the statement number in the C code, and the y axis shows the variables:



A horizontal line stretches from the first statement where the variable is used to the last use of the variable; a variable is said to be **live** during this interval. At each statement, we can

determine every variable that is currently in use. The maximum number of variables in use at any statement determines the maximum number of registers that are required. In this case, statement two requires three registers: *c*, *w*, and *x*. This fits within the four-register limitation. By reusing registers once their current values are no longer needed, we can write code that requires no more than four registers. Here is one register assignment:

a	r0
b	r1
c	r2
d	r0
w	r3
x	r0
y	r3

Here is the ARM assembly code that uses the above register assignment:

```

LDR r0,[p_a]      ; load a into r0 using pointer to a (p_a)
LDR r1,[p_b]      ; load b into r1
ADD r3,r0,r1      ; compute a + b
STR r3,[p_w]      ; w = a + b
LDR r2,[p_c]      ; load c into r2
ADD r0,r2,r3      ; compute c + w, reusing r0 for x
STR r0,[p_x]      ; x = c + w
LDR r0,[p_d]      ; load d into r0
ADD r3,r2,r0      ; compute c + d, reusing r3 for y
STR r3,[p_y]      ; y = c + d

```

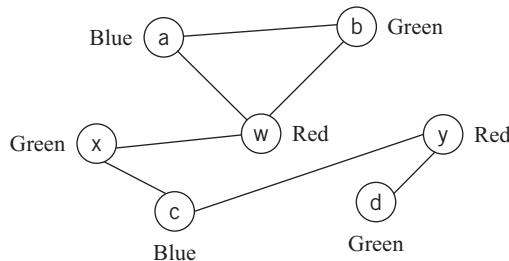
---

If a section of code requires more registers than are available, we must **spill** some of the values out to memory temporarily. After computing some values, we write the values to temporary memory locations, reuse those registers in other computations, and then, reread the old values from the temporary locations to resume work. Spilling registers is problematic in several respects: it requires extra CPU time, and uses up both instruction and data memory. Putting effort into register allocation to avoid unnecessary register spills is worth your time.

We can solve register allocation problems by building a **conflict graph** and solving a graph coloring problem. As shown in Fig. 5.17, each variable in the high-level language code is represented by a node. An edge is added between two nodes if they are both live at the same time. The graph coloring problem is to use the smallest number of distinct colors to color all of the nodes, such that no two nodes are directly connected by an edge of the same color. The figure shows a satisfying coloring that uses three colors. Graph coloring is NP-complete, but there are efficient heuristic algorithms that can give good results on typical register allocation problems.

Lifetime analysis assumes that we have already determined the order in which we will evaluate operations. In many cases, we have freedom in the order in which we do things. Consider this expression:

$$(a + b) * (c - d)$$

**FIGURE 5.17**

Using graph coloring to solve the problem of Example 5.5.

We must perform the multiplication last, but we can do either the addition or the subtraction first. Different orders of loads, stores, and arithmetic operations may also result in different execution times on pipelined machines. If we can keep values in registers without having to reread them from main memory, we can save on execution time and reduce the code size as well.

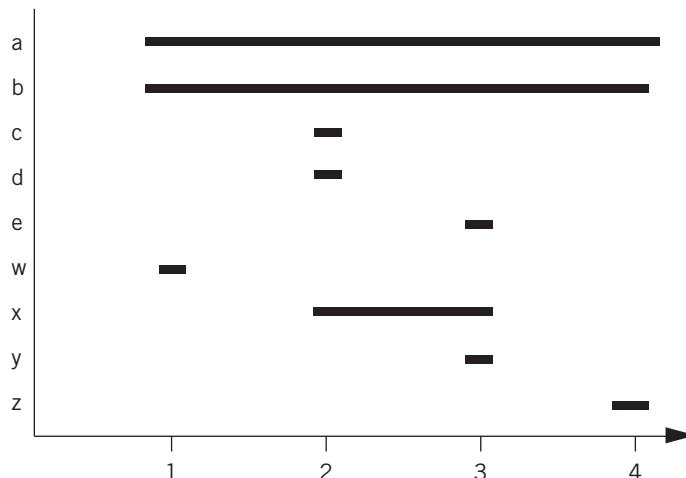
The next example shows how proper **operator scheduling** can improve register allocation.

### Example 5.6: Operator Scheduling for Register Allocation

Here is a sample C code fragment:

```
w = a + b; /* statement 1 */
x = c + d; /* statement 2 */
y = x + e; /* statement 3 */
z = a - b; /* statement 4 */
```

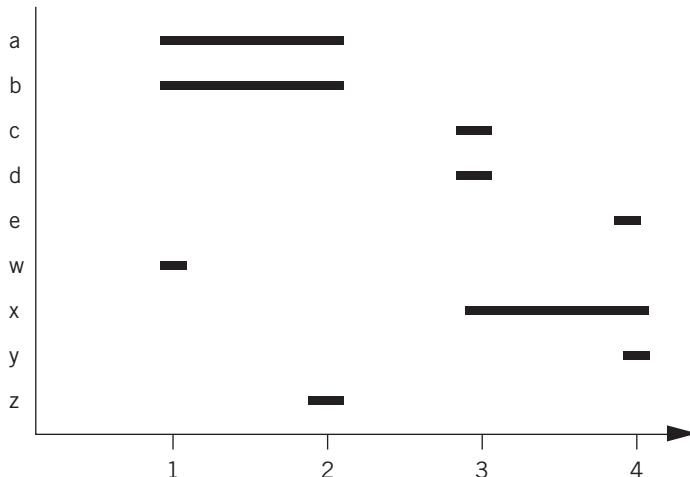
If we compile the statements in the order in which they were written, we get this register graph:



Because  $w$  is needed until the last statement, we need five registers at statement 3, even though only three registers are needed for the statement at line 3. If we swap statements 3 and 4 (renumbering them 39 and 49), we reduce our requirements to three registers. Here is the modified C code:

```
w = a + b; /*statement 1 */
z = a - b; /* statement 29 */
x = c + d; /*statement 39 */
y = x + e; /*statement 49 */
```

Additionally, here is the lifetime graph for the new code:



Compare the ARM assembly code for the two code fragments. We have written both assuming that we have only four free registers. In the *before* version, we do not have to write out any values, but we must read  $a$  and  $b$  twice. The *after* version allows us to retain all values in registers for as long as we need them.

#### *Before version*

```
LDR r0,a
LDR r1,b
ADD r2,r0,r1
STR r2,w ; w = a + b
LDR r0,c
LDR r1,d
ADD r2,r0,r1
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
LDR r0,a ; reload a
LDR r1,b ; reload b
SUB r2,r1,r0
STR r2,z ; z = a - b
```

#### *After version*

```
LDR r0,a
LDR r1,b
ADD r2,r1,r0
STR r2,w ; w = a + b
SUB r2,r0,r1
STR r2,z ; z = a - b
LDR r0,c
LDR r1,d
ADD r2,r1,r0
STR r2,x ; x = c + d
LDR r1,e
ADD r0,r1,r2
STR r0,y ; y = x + e
```

**Scheduling**

We have some freedom to choose the order in which operations will be performed. We can use this to our advantage; for example, we may be able to improve the register allocation by changing the order in which operations are performed, thereby changing the lifetimes of the variables.

We can solve scheduling problems by keeping track of resource utilization over time. We do not have to know the exact microarchitecture of the CPU. All we have to know is that, for example, instruction types 1 and 2 both use resource A, while instruction types 3 and 4 use resource B. CPU manufacturers generally disclose enough information about the microarchitecture to allow us to schedule instructions even when they do not provide a detailed description of the CPU's internals.

We can keep track of CPU resources during instruction scheduling using a **reservation table** [Kog81]. As illustrated in Fig. 5.18, the rows in the table represent the instruction execution time slots and the columns represent resources that must be scheduled. Before scheduling an instruction to be executed at a particular time, we check the reservation table to determine whether all resources that are needed by the instruction are available at that time. Upon scheduling the instruction, we update the table to note all resources that are used by that instruction. Various algorithms can be used for the scheduling itself, depending on the types of resources and instructions involved, but the reservation table provides a good summary of the state of an instruction scheduling problem that is in progress.

We can also schedule instructions to maximize performance. As we know from Section 3.6, when an instruction that takes more cycles than normal to finish is in the pipeline, pipeline bubbles appear that reduce the performance. **Software pipelining** is a technique for reordering instructions across several loop iterations to reduce pipeline bubbles. Some instructions take several cycles to complete; if the value that is produced by one of these instructions is needed by other instructions in the loop iteration, they must wait for that value to be produced. Rather than pad the loop with no-ops, we can start instructions from the next iteration. The loop body then contains instructions that manipulate values from several different loop iterations; some of the instructions are working on the early part of iteration  $n + 1$ , others are working on iteration  $n$ , and still others are finishing iteration  $n - 1$ .

Selecting the instructions to use to implement each operation is not trivial. There may be several different instructions that can be used to accomplish the same goal, but

**Instruction selection**

Time	Resource A	Resource B
$t$	X	
$t + 1$	X	X
$t + 2$	X	
$t + 3$		X

**FIGURE 5.18**

A reservation table for instruction scheduling.

they may have different execution times. Moreover, using one instruction for one part of the program may affect the instructions that can be used in adjacent code. Although we can't discuss all of the problems and methods for code generation here, a little bit of knowledge helps us to envision what the compiler is doing.

One useful technique for generating code is **template matching**, as illustrated in Fig. 5.19. We have a directed acyclic graph (DAG) that represents the expression for which we want to generate code. To help to match up instructions and operations, we represent instructions using the same DAG representation. We shade the instruction template nodes to distinguish them from code nodes. Each node has a cost, which may be simply the execution time of the instruction, or may include factors for size, power consumption, and so on. In this case, we have shown that each instruction takes the same amount of time, and thus, all have a cost of 1. Our goal is to cover all nodes in the code DAG with instruction DAGs; until we have covered the code DAG, we haven't generated code for all of the operations in the expression. In this case, the lowest-cost covering uses the multiply-add instruction to cover both nodes. If we first tried to cover the bottom node with the multiply instruction,

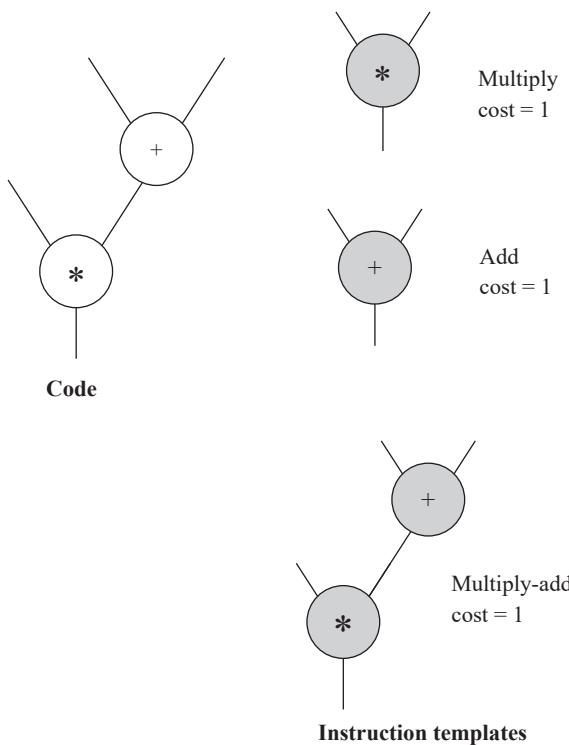


FIGURE 5.19

Code generation by template matching.

**Understanding your compiler**

we would find ourselves blocked from using the multiply–add instruction. Dynamic programming can be used to find the lowest-cost covering of trees efficiently and heuristics can extend the technique to DAGs.

Clearly, the compiler can vastly transform your program during the creation of assembly language. However, compilers are also substantially different in terms of the optimizations they perform. Understanding your compiler can help you get the best code out of it.

Studying the assembly language output of the compiler is a good way to learn about what the compiler does. Some compilers will annotate sections of code to help you to make the correspondence between the source and assembler output. Starting with small examples that exercise only a few types of statements will help. You can experiment with different optimization levels (the `-O` flag on most C compilers). You can also try writing the same algorithm in several ways to see how the compiler’s output changes.

If you can’t get your compiler to generate the code that you want, you may need to write your own assembly language. You can do this by writing it from scratch or by modifying the output of the compiler. If you write your own assembly code, you must ensure that it conforms to all compiler conventions, such as procedure call linkage. If you modify the compiler output, you should be sure that you have the algorithm correct before you start writing code so that you don’t have to edit the compiler’s assembly language output repeatedly. You also need to document the fact that the high-level language source is, in fact, not the code used in the system.

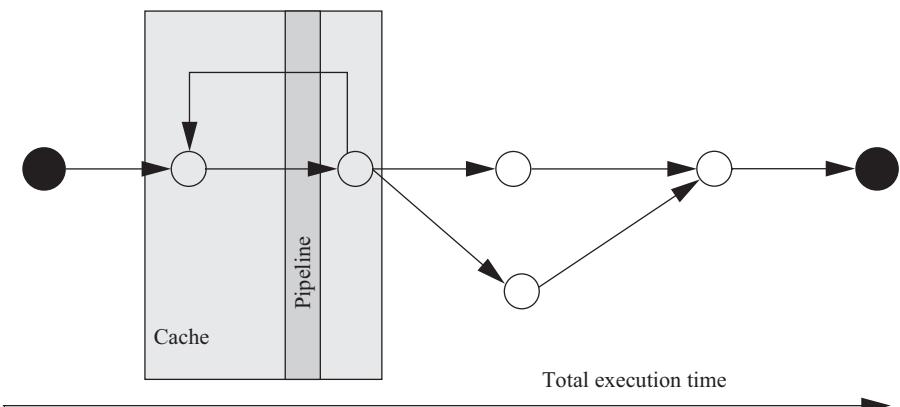
## 5.6 Program-level performance analysis

Because embedded systems must perform functions in real time, we often need to know how fast a program runs. The techniques that we use to analyze program execution time are also helpful in analyzing properties such as power consumption. In this section, we study how to analyze programs to estimate their run times. We also examine how to optimize programs to improve their execution times; of course, optimization relies on analysis.

It is important to keep in mind that CPU performance is not judged in the same way as program performance. Certainly, the CPU clock rate is a very unreliable metric for program performance. However, more importantly, the fact that the CPU executes part of our program quickly doesn’t mean that it will execute the entire program at the desired rate. As illustrated in Fig. 5.20, the CPU pipeline and cache act as windows into our program. To understand the total execution time of our program, we must look at execution paths, which are far longer than the pipeline and cache windows in general. The pipeline and cache influence the execution time, but the execution time is a global property of the program.

While we might hope that the execution time of programs could be precisely determined, this is in fact difficult to do in practice:

- The execution time of a program often varies with the input data values, because those values select different execution paths in the program. For example, loops



**FIGURE 5.20**

Execution time is a global property of a program.

may be executed a varying number of times and different branches may execute blocks of varying complexity.

- The cache has a major effect on the program performance, and once again, the cache's behavior depends in part on the data values that are input into the program.
- Execution times may vary even at the instruction level. Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations. In general, the execution time of an instruction in a pipeline depends not only on that instruction, but also on the instructions around it in the pipeline.

#### Measuring execution speed

We can measure program performance in several ways:

- Some microprocessor manufacturers supply simulators for their CPUs. The simulator runs on a workstation or PC, takes an executable as input for the microprocessor along with input data, and simulates the execution of that program. Some of these simulators go beyond functional simulation to measure the execution time of the program. Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution. Be careful: some microprocessor performance simulators are not 100% accurate, and simulation of I/O-intensive code may be difficult.
- A timer that is connected to the microprocessor bus can be used to measure the performance of executing sections of code. The code to be measured would reset and start the timer at its start and stop the timer at the end of execution. The length of the program that can be measured is limited by the accuracy of the timer.
- A logic analyzer can be connected to the microprocessor bus to measure the start and stop times of a code segment. This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution.

The length of code that can be measured is limited by the size of the logic analyzer's buffer.

We are interested in the following three different types of performance measures on programs:

- **Average-case execution time:** This is the typical execution time that we would expect for typical data. Clearly, the first challenge is defining typical inputs.
- **Worst-case execution time:** The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases, it is not.
- **Best-case execution time:** This measure can be important in multirate real-time systems, as seen in [Chapter 6](#).

First, we look at the fundamentals of program performance in more detail. We then consider trace-driven performance based on executing the program and observing its behavior.

### 5.6.1 Analysis of program performance

The key to evaluating the execution time is breaking the performance problem into parts. The program execution time [Sha89] can be seen as

$$\text{execution time} = \text{program path} + \text{instruction timing}$$

The path is the sequence of instructions executed by the program (or its equivalent in the high-level language representation of the program). The instruction timing is determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behavior, and caching. Luckily, these two problems can be solved relatively independently.

Although we can trace the execution path of a program through its high-level language specification, it is difficult to get accurate estimates of the total execution time from a high-level language program. This is because there is no direct correspondence between program statements and instructions. The number of memory locations and variables must be estimated, and results may be either saved for reuse or recomputed on the fly, among other effects. These problems become more challenging as the compiler puts more and more effort into optimizing the program. However, some aspects of program performance can be estimated by looking directly at the C program. For example, if a program contains a loop with a large, fixed-iteration bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time consuming segments of the program.

Of course, a precise estimate of performance also relies on the instructions to be executed, because different instructions take different amounts of time. In addition, to make life even more difficult, the execution time of one instruction can depend on the instructions executed before and after it.

The next example illustrates data-dependent program paths.

---

### Example 5.5: Data-dependent Paths in *if* Statements

Here is a pair of nested *if* statements:

```
if (a || b) { /* test 1 */
    if (c) /*test 2 */
        { x = r *s + t; /* assignment 1 */
        else { y = r + s; /*assignment 2 */
        z = r + s + u; /*assignment 3 */
    } else {
        if (c) /*test 3 */
            { y = r - t; /* assignment 4 */
    }
}
```

The conditional tests and assignments are labeled within each *if* statement to make it easier to identify paths. Which execution paths may be exercised? One way to enumerate all of the paths is to create a truth table-like structure. The paths are controlled by the variables in the *if* conditions, namely, *a*, *b*, and *c*. For any given combination of values of those variables, we can trace through the program to see which branch is taken at each *if* and which assignments are performed. For example, when *a* = 1, *b* = 0, and *c* = 1, *test 1* is true and *test 2* is true. This means that we first perform *assignment 1*, and then, *assignment 3*.

Here are the results for all of the controlling variable values:

<b>a</b>	<b>b</b>	<b>c</b>	<b>Path</b>
0	0	0	test 1 false, test 3 false: no assignments
0	0	1	test 1 false, test 3 true: assignment 4
0	1	0	test 1 true, test 2 false: assignments 2, 3
0	1	1	test 1 true, test 2 true: assignments 1, 3
1	0	0	test 1 true, test 2 false: assignments 2, 3
1	0	1	test 1 true, test 2 true: assignments 1, 3
1	1	0	test 1 true, test 2 false: assignments 2, 3
1	1	1	test 1 true, test 2 true: assignments 1, 3

Note that there are only four distinct cases: no assignment, assignment 4, assignments 2 and 3, or assignments 1 and 3. These correspond to the possible paths through the nested *ifs*; the table adds value by telling us which variable values exercise each of these paths.

---

Enumerating the paths through a fixed-iteration *for* loop is seemingly simple. In the code,

```
for (i = 0; i < N; i++)
    a[i] = b[i]*c[i];
```

the assignment in the loop is performed exactly  $N$  times. However, we can't forget the code that is executed to set up the loop and to test the iteration variable.

Example 5.6 illustrates how to determine the path through a loop.

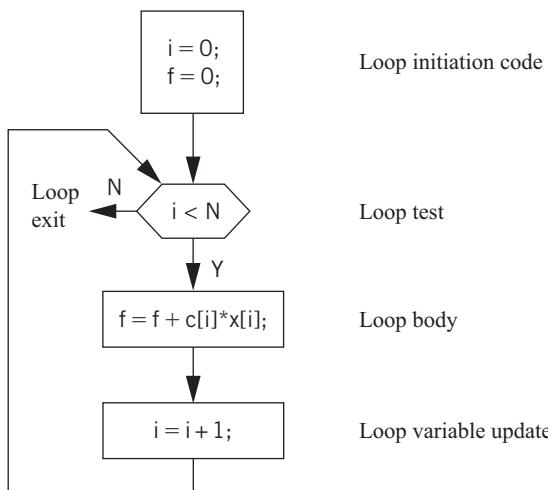
---

### Example 5.6: Paths in a Loop

Here is the loop code for the FIR filter of Application Example 2.1:

```
for (i = 0, f = 0; i < N; i++)
    f = f + c[i] * x[i];
```

By examining the CDFG for the code, we can more easily determine how many times various statements are executed. Here is the CDFG once again:



The CDFG makes it clear that the loop initiation block is executed once, the test is executed  $N + 1$  times, and the body and loop variable update are each executed  $N$  times.

---

Example 5.6 has very simple behavior: the loop executes for a fixed number of iterations and the loop body contains no conditionals. The next example makes a small change to the FIR filter that results in substantially more complex behavior.

---

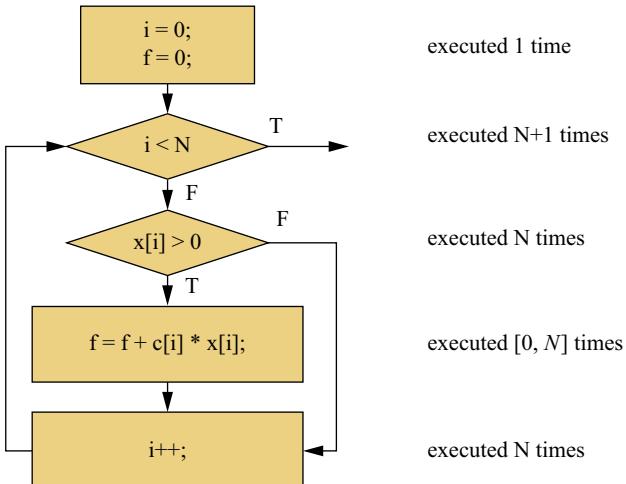
### Example 5.7: Conditional Behavior in a Loop

Here is a slightly more complex version of the FIR filter:

```
for (i=0, f=0; i<N; i++) {
    if (x[i] > 0)
        f = f + c[i] * x[i];
}
```

The result is updated conditionally based on the value of  $x[i]$ .

The CDFG for this code is more complex than that for the basic FIR filter:



The body of the *if* statement that tests  $x[i] > 0$  is executed a variable number of times. We can say that it is executed no more than  $N$  times, as it can be executed at most once per loop iteration. However, the exact number of times that it will be executed depends on the data values that are input into the program. Without knowing something about the behavior of the data values, we cannot bound the number of times that this statement is executed more precisely. As a result, we can only write the execution time of the code as being bounded by its best case (no values of  $x[i]$  are greater than zero) and its worst case (all values of  $x[i]$  are greater than zero).

### Instruction timing

Once we know the execution path of the program, we have to measure the execution time of the instructions that are executed along that path. The simplest estimate is to assume that every instruction takes the same number of clock cycles, which means that we need only count the instructions and multiply by the per-instruction execution time to obtain the program's total execution time. However, even when ignoring cache effects, this technique is simplistic for the reasons summarized below.

- *Not all instructions take the same amount of time.* RISC architectures tend to provide uniform instruction execution times to keep the CPU's pipeline full. However, as we saw in [Chapter 3](#), even very simple RISC architectures like the PIC16F take different amounts of time to execute certain instructions. Floating-point instructions show especially wide variations in execution time; while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.
- *The execution times of instructions are not independent.* The execution time of one instruction depends on the instructions around it. For example, many CPUs use register bypassing to speed up instruction sequences when the result of one

instruction is used in the next instruction. As a result, the execution time of an instruction may depend on whether its destination register is used as a source for the next operation (or vice versa).

- *The execution time of an instruction may depend on operand values.* This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result. Other specialized instructions can, for example, perform a data-dependent number of integer operations.

We can handle the first two problems more easily than the third. We can look up instruction execution times in a table; the table will be indexed by opcode and possibly by other parameter values such as the registers used. To handle interdependent execution times, we can add columns to the table to consider the effects of nearby instructions. Because these effects are generally limited by the size of the CPU pipeline, we know that we need to consider a relatively small window of instructions to handle such effects. Handling variations due to operand values is difficult without executing the program using a variety of data values, given the large number of factors that can affect value-dependent instruction timing. Luckily, these effects are often small. Even in floating-point programs, most of the operations are typically additions and multiplications, whose execution times have small variances.

#### Caching effects

Thus far, we have not considered the effect of the cache. Because the access time for main memory can be 10–100 times larger than the cache access time, caching can have huge effects on the instruction execution time by changing both the instruction and data access times. Caching performance is inherently dependent on the program’s execution path because the cache’s contents depend on the history of accesses.

The next example studies the effects of caching on the FIR filter.

---

### Example 5.8: Caching Effects on the FIR Filter

Here is the loop code for our FIR filter:

```
for (i = 0, f = 0; i < N; i++)
    f = f + c[i] * x[i];
```

The value of  $f$  is kept in a register so it doesn’t have to be fetched from memory. However,  $c[i]$  and  $x[i]$  must be fetched during every iteration. For simplicity, let’s assume that the cache has  $L = 4$  words per line:

Line 0	Word 0	Word 1	Word 2	Word 3
Line 1	Word 4	Word 5	Word 6	Word 7

We also assume that the  $c$  and  $x$  arrays are placed in memory such that they do not interfere with each other in the cache. In this situation, the time required to read the next value of  $c$  or  $x$  depends on that word’s position in the cache line. An array entry corresponding to the first entry in the cache line will result in a cache miss and will require  $t_{miss}$  cycles. The other entries will result in cache hits and require  $t_{hit}$  cycles. Remember that the loop body has four instructions, two of which are loads. As the loop body is executed  $N$  times, we can write the total execution time for all  $N$  iterations as

$$t_{loop} = 2N + \frac{N}{L}t_{miss} + N\left(1 - \frac{1}{L}\right)t_{hit}$$

This formula assumes that the number of words in the cache line evenly divides the number of loop iterations; it is slightly more cumbersome in the general case.

---

### 5.6.2 Measurement-driven performance analysis

The most direct way to determine the execution time of a program is by measuring it. This approach is appealing, but it does have some drawbacks. First, to cause the program to execute its worst-case execution path, we must provide the proper inputs for it. Determining the set of inputs that will guarantee the worst-case execution path is infeasible. Furthermore, to measure the program's performance on a particular type of CPU, we need the CPU or its simulator.

Despite these problems, measurement is the most commonly used way to determine the execution time of embedded software. Worst-case execution time analysis algorithms have been used successfully in some areas, such as flight control software, but many system design projects determine the execution time of their programs by measurement.

Most methods of measuring program performance combine the determination of the execution path and the timing of that path: as the program executes, it chooses a path and we observe the execution time along that path. We refer to the record of the execution path of a program as a **program trace** (or more succinctly, a **trace**). Traces can be valuable for other purposes, such as analyzing the cache behavior of the program.

Perhaps the biggest problem in measuring program performance is determining a useful set of inputs to give the program. This problem has two aspects. First, we have to determine the actual input values. We may be able to use benchmark data sets or data that are captured from a running system to help us to generate typical values. For simple programs, we may be able to analyze the algorithm to determine the inputs that cause the worst-case execution time. The software testing methods of [Section 5.10](#) can help us to generate some test values and determine how thoroughly we have exercised the program.

The other problem with input data is the **software scaffolding** that we may need to feed data into the program and get data out. When we are designing a large system, it may be difficult to extract out part of the software and test it independently of the other parts of the system. We may need to add new testing modules to the system software to help us to introduce testing values and to observe the testing outputs.

We can measure program performance either directly on the hardware or by using a simulator. Each method has its advantages and disadvantages.

**Profiling** is a simple method for analyzing software performance. A profiler does not measure the execution time; instead it counts the number of times that procedures or basic blocks in the program are executed. There are two major ways to profile a

Program traces

Measurement issues

Profiling

program: we can modify the executable program by adding instructions that increment a location every time the program passes that point in the program, or we can sample the program counter during execution and keep track of the distribution of PC values. Profiling adds relatively little overhead to the program and it gives us some useful information about where the program spends most of its time.

#### Physical performance measurement

Physical measurement requires some sort of hardware instrumentation. The most direct method of measuring the performance of a program would be to watch the program counter's value: start a timer when the PC reaches the program's start and stop the timer when it reaches the program's end. Unfortunately, it generally isn't possible to observe the program counter directly. However, in many cases, it is possible to modify the program so that it starts a timer at the beginning of execution and stops the timer at the end. While this doesn't give us direct information about the program trace, it does give us the execution time. If we have several timers available, we can use them to measure the execution time of different parts of the program.

A logic analyzer or an oscilloscope can be used to watch for signals that mark various points in the execution of the program. However, because logic analyzers have a limited amount of memory, this approach doesn't work well for programs with extremely long execution times.

#### Hardware traces

Some CPUs have hardware facilities for automatically generating trace information. For example, the Pentium family microprocessors generate a special bus cycle, namely a branch trace message, that shows the source and/or destination address of a branch [Col97]. If we record only traces, we can reconstruct the instructions that are executed within the basic blocks, while greatly reducing the amount of memory required to hold the trace.

#### Simulation-based performance measurement

The alternative to physical measurement of the execution time is simulation. A CPU simulator is a program that takes a memory image for a CPU as input and performs the operations on that memory image that the actual CPU would perform, leaving the results in the modified memory image. For purposes of performance analysis, the most important type of CPU simulator is the **cycle-accurate simulator**, which performs a sufficiently detailed simulation of the processor's internals that it can determine the exact number of clock cycles required for execution. A cycle-accurate simulator is built with detailed knowledge of how the processor works, so that it can take into account all of the possible behaviors of the microarchitecture that may affect the execution time. Cycle-accurate simulators are slower than the processor itself, but a variety of techniques can be used to make them surprisingly fast, running only hundreds of times slower than the hardware itself. A simulator that functionally simulates instructions but does not provide timing information is known as an **instruction-level simulator**.

A cycle-accurate simulator has a complete model of the processor, including the cache. It can therefore provide valuable information about why the program runs too slowly. The next example discusses a simulator that can be used to model many different processors.

---

### Example 5.9: Cycle-accurate Simulation

SimpleScalar (<http://www.simplescalar.com>) is a framework for building cycle-accurate CPU models. Some aspects of the processor can be configured easily at run time. For more complex changes, we can use the SimpleScalar toolkit to write our own simulator.

We can use SimpleScalar to simulate the FIR filter code. SimpleScalar can model a number of different processors; we will use a standard ARM model here.

We want to include the data as part of the program so that the execution time doesn't include file I/O. File I/O is slow and the time that it takes to read or write data can change substantially from one execution to another. We get around this problem by setting up an array that holds the FIR data. Furthermore, because the test program will include some initialization and other miscellaneous code, we execute the FIR filter many times in a row using a simple loop. Here is the complete test program:

```
#define COUNT 100
#define N 12

int x[N] = {8,17,3,122,5,93,44,2,201,11,74,75};
int c[N] = {1,2,4,7,3,4,2,2,5,8,5,1};

main() {
    int i, k, f;
    for (k=0; k<COUNT; k++) { /* run the filter */
        for (i=0; i<N; i++)
            f += c[i]*x[i];
    }
}
```

To start the simulation process, we compile our test program using a special compiler:

```
% arm-linux-gcc firtest.c
```

This gives us an executable program (by default, a.out) that we use to simulate our program:

```
% arm-outorder a.out
```

SimpleScalar produces a large output file with a great deal of information about the program's execution. Because this is a simple example, the most useful piece of data is the total number of simulated clock cycles that are required to execute the program:

```
sim_cycle 25854 × total simulation time in cycles
```

To ensure that we can ignore the effects of program overhead, we will execute the FIR filter for several different values of COUNT and compare. This run uses COUNT = 100; when we also run COUNT = 1,000 and COUNT = 10,000, we get these results:

COUNT	Total simulation time in cycles	Simulation time for one filter execution
100	25,854	259
1000	155,759	156
10,000	1,451,840	145

Because the FIR filter is so simple and runs in so few cycles, we have to execute it a number of times to wash out all the other overhead of program execution. However, the times for 1000 and 10,000 filter executions are within 10% of each other, so these values are reasonably close to the actual execution time of the FIR filter itself.

---

### Performance counters

Some CPUs provide hardware **performance counters** to keep track of performance-related events during execution. These registers can be used to count the number of events such as cache and pipeline operations. The ARM Performance Monitoring Unit (PMU) is an example of a performance counter system [Arm21].

## 5.7 Software performance optimization

In this section, we will look at several techniques for optimizing software performance, including basic loop and cache-oriented loop optimizations as well as more generic strategies.

### 5.7.1 Basic loop optimizations

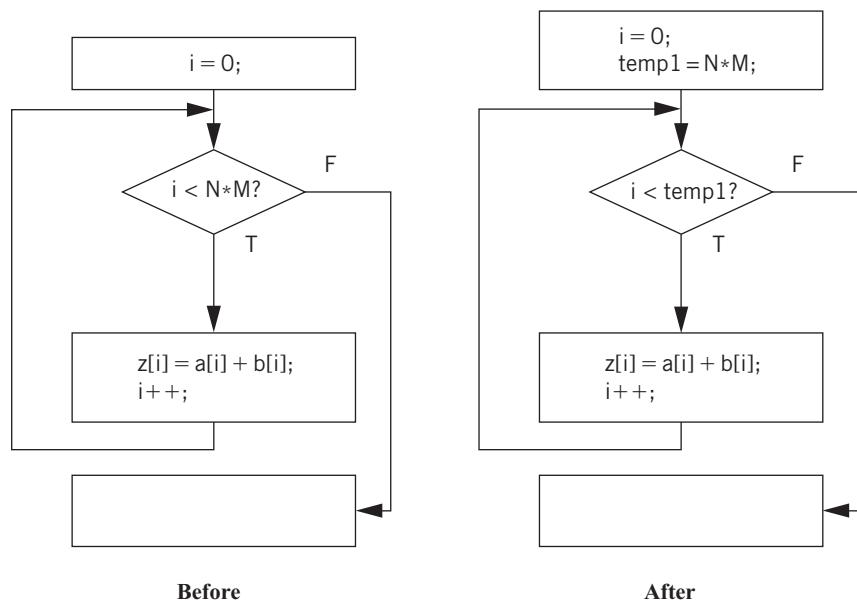
Loops are important targets for optimization, because programs with loops tend to spend a lot of time executing those loops. There are three important techniques in optimizing loops: **code motion**, **induction variable elimination**, and **strength reduction**.

Code motion lets us move unnecessary code out of a loop. If a computation's result does not depend on the operations that are performed in the loop body, we can safely move it out of the loop. Code motion opportunities can arise because programmers may find some computations clearer and more concise when put in the loop body, even though they are not strictly dependent on the loop iterations. A simple example of code motion is also common. Consider this loop:

```
for (i = 0; i < N*M; i++) {
    z[i] = a[i] + b[i];
}
```

The code motion opportunity becomes more obvious when we draw the loop's CDFG, as shown in Fig. 5.21. The loop bound computation is performed on every iteration during the loop test, even though the result never changes. We can avoid  $N \times M - 1$  unnecessary executions of this statement by moving it before the loop, as shown in the figure.

An **induction variable** is a variable whose value is derived from the loop iteration variable's value. The compiler often introduces induction variables to help it to implement the loop. When properly transformed, we may be able to eliminate some variables and apply strength reduction to others.

**FIGURE 5.21**

Code motion in a loop.

A nested loop is a good example of the use of induction variables. Here is a simple nested loop:

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        z[i][j] = b[i][j];
```

The compiler uses induction variables to help it to address the arrays. Let us rewrite the loop in C using induction variables and pointers. Later, we use a common induction variable for the two arrays, even though the compiler would probably introduce separate induction variables, and then, merge them.

```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++) {
        zbinduct = i*M + j;
        *(zptr + zbinduct) = *(bptr + zbinduct);
    }
```

In the above code, `zptr` and `bptr` are pointers to the heads of the `z` and `b` arrays, and `zbinduct` is the shared induction variable. However, we do not need to compute

`zbinduct` afresh each time. Because we are stepping through the arrays sequentially, we can simply add the update value to the induction variable:

```

zbinduct = 0;
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        *(zptr + zbinduct) = *(bptr + zbinduct);
        zbinduct++;
    }
}

```

This is a form of strength reduction because we have eliminated the multiplication from the induction variable computation.

Strength reduction helps us to reduce the cost of a loop iteration. Consider this assignment is

```
y = x * 2;
```

In integer arithmetic, we can use a left shift rather than a multiplication by 2 (as long as we properly keep track of overflows). If the shift is faster than the multiply, we probably want to perform the substitution. This optimization can often be used with induction variables because loops are often indexed with simple expressions. Strength reduction can often be performed with simple substitution rules because there are relatively few interactions between the possible substitutions.

### 5.7.2 Cache-oriented loop optimizations

A **loop nest** is a set of loops, one inside the other. Loop nests occur when we process arrays. A large body of techniques has been developed for optimizing loop nests. Many of these methods are designed to improve the cache performance. Rewriting a loop nest changes the order in which array elements are accessed. This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve the cache performance. In this section, we concentrate on the analysis of loop nests for cache performance.

The next example looks at two cache-oriented loop nest optimizations.

#### Programming Example 5.7: Data Realignment and Array Padding

Assume that we want to optimize the cache behavior of the following code:

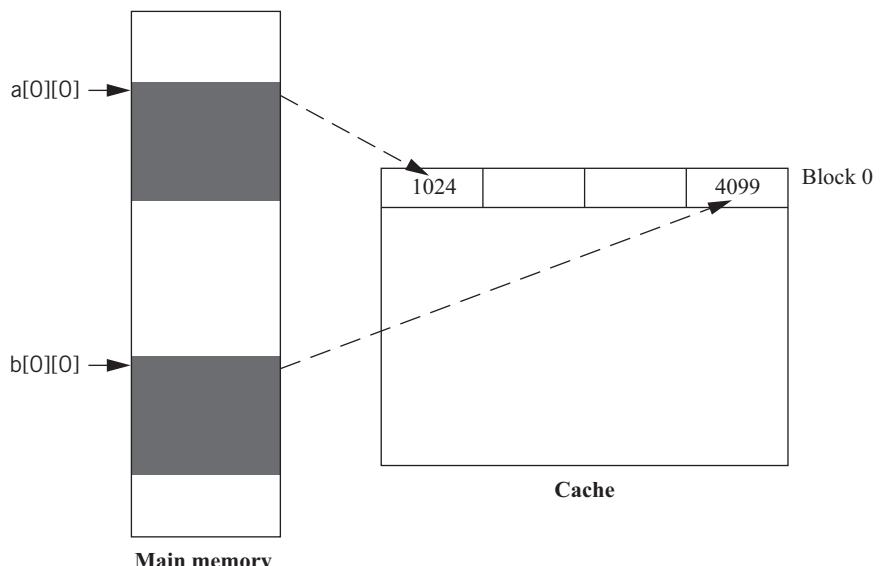
```

for (j = 0; j < M; j++)
    for (i = 0; i < N; i++)
        a[j][i] = b[j][i] * c;

```

Let us also assume that the `a` and `b` arrays are sized with `M` at 265 and `N` at 4 and a 256-line, four-way set-associative cache with four words per line. Even though this code does not reuse any data elements, cache conflicts can cause serious performance problems because they interfere with spatial reuse at the cache line level.

Assume that the starting location for  $a[]$  is 1024 and the starting location for  $b[]$  is 4099. Although  $a[0][0]$  and  $b[0][0]$  do not map to the same word in the cache, they do map to the same block.



As a result, we see the following scenario in execution:

- The access to  $a[0][0]$  brings in the first four words of  $a[]$ .
- The access to  $b[0][0]$  replaces  $a[0][0]$  through  $a[0][3]$  with  $b[0][3]$  and the contents of the three locations before  $b[]$ .
- When  $a[0][1]$  is accessed, the same cache line is again replaced with the first four elements of  $a[]$ .

Once the  $a[0][1]$  access brings that line into the cache, it remains there for the  $a[0][2]$  and  $a[0][3]$  accesses, because the  $b[]$  accesses are now on the next line. However, the scenario repeats itself at  $a[1][0]$  and every four iterations of the cache.

One way to eliminate the cache conflicts is to move one of the arrays. We do not have to move it far. If we move  $b$ 's start to 4100, we eliminate the cache conflicts.

However, this fix won't work in more complex situations. Moving one array may only introduce cache conflicts with another array. In such cases, we can use another technique called padding. If we extend each of the rows of the arrays to have four elements rather than three, with the padding word placed at the beginning of the row, we eliminate the cache conflicts. In this case,  $b[0][0]$  is located at 4100 by the padding. Although padding wastes memory, it substantially improves memory performance. In complex situations with multiple arrays and sophisticated access patterns, we must use a combination of techniques, namely relocating and padding arrays, to be able to minimize cache conflicts.

---

**Loop tiling** breaks a loop up into a set of nested loops, with each inner loop performing the operations on a subset of the data. Loop tiling changes the order in which array elements are accessed, thereby allowing us to control the behavior of the cache better during loop execution. The next example illustrates the use of loop tiling.

---

### Programming Example 5.8: Loop Tiling

Here is a nest of two loops, with each loop walking through an array:

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        z[i][j] = x[i] * y[j];
    }
}
```

Every iteration of the outer *i* loop makes use of every value of *y*[ ]. The values of *y*[ ] may conflict with *x*[ ][ ] in the cache.

We can improve the cache behavior of *y*[ ] by dividing it into tiles of size TILE, each of which can fit into the cache:

```
for (j=0; j < N; j += TILE) {
    for (i=0; i<N; i++) {
        for (jj=0; j<TILE; jj++) {
            z[i][j + jj] = x[i] * y[j + jj];
        }
    }
}
```

In this code, each iteration of the *i* loop makes use of one tile of *y*[ ]. The new, outermost loop iterates over the tiles to ensure that the entire array is covered.

---

### 5.7.3 Performance optimization strategies

Let's look more generally at how to improve the program execution time. First, we must make sure that the code really needs to run faster. Performance analysis and measurement will give you a baseline for the execution time of the program. Knowing the overall execution time tells you how much it needs to be improved. Knowing the execution time of various pieces of the program helps you to identify the correct locations for changes to the program.

You may be able to redesign your algorithm to improve efficiency. Examining asymptotic performance is often a good guide to efficiency. Doing fewer operations is usually the key to performance. However, in a few cases, brute force may provide a better implementation. A seemingly simple high-level-language statement may in fact hide a very long sequence of operations that slows down the algorithm. Using dynamically allocated memory is one example, because managing the heap takes time but it is hidden from the programmer. For example, a sophisticated algorithm that uses dynamic storage may be slower in practice than an algorithm that performs more operations on statically allocated memory.

Finally, you can look at the implementation of the program itself. Here are a few hints on program implementation:

- Try to use registers efficiently. Group accesses to a value together so that the value can be brought into a register and kept there.

- Make use of page mode accesses in the memory system whenever possible. Page mode reads and writes eliminate one step in the memory access. You can increase the use of page mode by rearranging your variables so that more can be referenced contiguously.
- Analyze the cache behavior to find major cache conflicts. Restructure the code to eliminate as many of these as you can, as follows:
  - For instruction conflicts, if the offending code segment is small, try to rewrite the segment to make it as small as possible so that it better fits into the cache. Writing in assembly language may be necessary. For conflicts across larger spans of code, try moving the instructions or padding with NOPs.
  - For scalar data conflicts, move the data values to different locations to reduce conflicts.
  - For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.

---

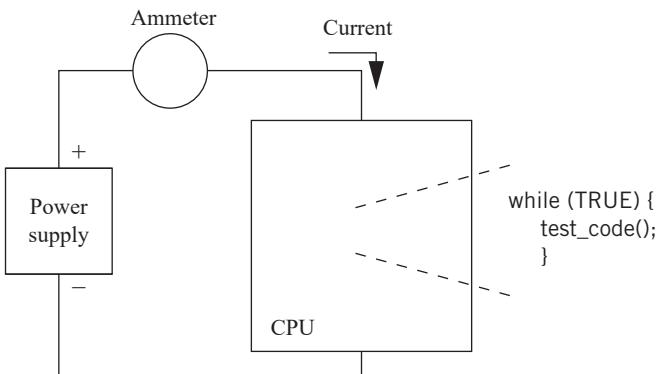
## 5.8 Program-level energy, and power analysis and optimization

Power consumption is a particularly important design metric for battery-powered systems because the battery has a very limited lifetime. However, power consumption is increasingly important in systems that run off the power grid. Fast chips run hot and controlling power consumption is an important element of increasing reliability and reducing the system cost.

How much control do we have over power consumption? Ultimately, we must consume the energy that is required to perform necessary computations. However, there are opportunities for saving power:

- We may be able to replace the algorithms with others that do things in clever ways that consume less power.
- Memory accesses are a major component of power consumption in many applications. By optimizing memory accesses, we may be able to reduce the power significantly.
- We may be able to turn off parts of the system—such as subsystems of the CPU, chips in the system, and so on—when we don’t need them in order to save power.

The first step in optimizing a program’s energy consumption is knowing how much energy the program consumes. It is possible to measure the power consumption for an instruction or a small code fragment [Tiw94]. The technique, which is illustrated in Fig. 5.22, executes the code under test repeatedly in a loop. By measuring the current flowing into the CPU, we are measuring the power consumption of the complete loop, including both the body and other code. By separately measuring the power consumption of a loop with no body (ensuring, of course, that the compiler hasn’t optimized away the empty loop), we can calculate the power consumption of the loop body code as the difference between the full loop and the bare loop energy cost of an instruction.

**FIGURE 5.22**

Measuring energy consumption for a piece of code.

Several factors contribute to the energy consumption of the program:

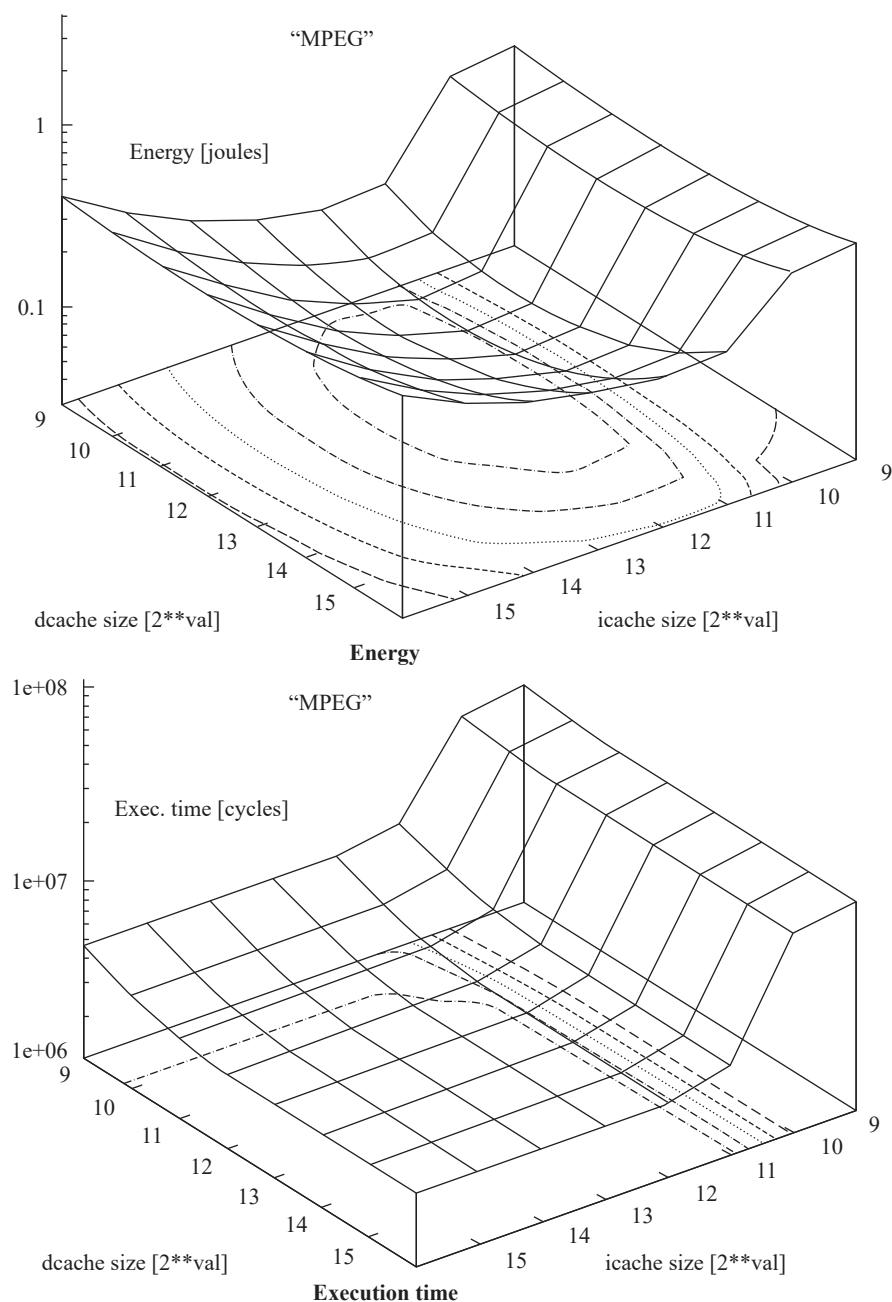
- Energy consumption varies somewhat from instruction to instruction.
- The sequence of instructions has some influence.
- The opcode and locations of the operands also matter.

Choosing which instructions to use can make some difference in a program's energy consumption, but concentrating on the instruction opcodes has limited payoffs in most CPUs. The program must do a certain amount of computation to perform its function. While there may be some clever ways to perform that computation, the energy cost of the basic computation will change by only a small amount compared to the total system energy consumption, and usually only after a great deal of effort. We are further hampered in our ability to optimize instruction-level energy consumption because most manufacturers do not provide detailed, instruction-level energy consumption figures for their processors.

#### Memory effects

In many applications, the biggest payoff in energy reduction for a given amount of designer effort comes from concentrating on the memory system. Memory transfers are by far the most expensive type of operation performed by a CPU [Cat98]; a memory transfer requires tens or hundreds of times more energy than an arithmetic operation. As a result, the biggest payoffs in energy optimization come from properly organizing the instructions and data in memory. Accesses to registers are the most energy efficient; cache accesses are more energy efficient than main memory accesses.

Caches are an important factor in energy consumption. A cache hit saves a costly main memory access and the cache itself is relatively power hungry because it is built from SRAM, not DRAM. If we can control the size of the cache, we want to choose the smallest cache that provides us with the necessary performance. Li and Henkel [Li98] measured the influence of caches on energy consumption in detail. Fig. 5.23 breaks down the energy consumption of a computer running MPEG (a video encoder)

**FIGURE 5.23**

Energy and execution time vs. instruction/data cache size for a benchmark program [Li98].

into several components: the software running on the CPU, main memory, data cache, and instruction cache.

As the instruction cache size increases, the energy cost of the software on the CPU decreases, but the instruction cache starts to dominate the energy consumption. Experiments like this on several benchmarks show that many programs have sweet spots in energy consumption. If the cache is too small, the program runs slowly and the system consumes a lot of power owing to the high cost of main memory accesses. If the cache is too large, the power consumption is high, without a corresponding payoff in performance. At intermediate values, the execution time and power consumption are both good.

#### **Energy optimization**

How can we optimize a program for low power consumption? The best overall advice is that *high performance = low power*. In general, making the program run faster also reduces the energy consumption.

Clearly, the biggest factor that can be reasonably well controlled by the programmer is the memory access patterns. If the program can be modified to reduce instruction or data cache conflicts, for example, the energy required by the memory system can be significantly reduced. The effectiveness of changes such as reordering instructions or selecting different instructions depends on the processor involved, but they are generally less effective than cache optimizations.

A few optimizations mentioned previously for performance are also often useful for improving energy consumption:

- Try to use registers efficiently. Group accesses to a value together so that the value can be brought into a register and kept there.
- Analyze cache behavior to find major cache conflicts. Restructure the code to eliminate as many of these as you can:
  - For instruction conflicts, if the offending code segment is small, try to rewrite the segment to make it as small as possible so that it better fits into the cache. Writing in assembly language may be necessary. For conflicts across larger spans of code, try moving the instructions or padding with NOPs.
  - For scalar data conflicts, move the data values to different locations to reduce conflicts.
  - For array data conflicts, consider either moving the arrays or changing your array access patterns to reduce conflicts.
- Make use of page mode accesses in the memory system whenever possible. Page mode reads and writes eliminate one step in the memory access, saving a considerable amount of power.

Metha et al. [Met97] present some additional observations about energy optimization:

- Moderate loop unrolling eliminates some loop control overhead. However, when the loop is unrolled too much, the power increases owing to the lower hit rates of straight-line code.

- Software pipelining reduces pipeline stalls, thereby reducing the average energy per-instruction.
- Eliminating recursive procedure calls where possible saves power by getting rid of function call overhead. Tail recursion can often be eliminated; some compilers do this automatically.

---

## 5.9 Analysis and optimization of program size

The memory footprint of a program is determined by the size of its data and instructions. Both must be considered to minimize the program size.

Data provide an excellent opportunity for minimizing size because the data are most highly dependent on the programming style. Because inefficient programs often keep several copies of data, identifying and eliminating duplications can lead to significant memory savings, usually with little performance penalty. Buffers should be sized carefully; rather than defining a data array to a large size that the program will never attain, determine the actual maximum amount of data that is held in the buffer and allocate the array accordingly. Data can sometimes be packed, such as by storing several flags in a single word and extracting them by using bit-level operations.

A very low-level technique for minimizing data is to reuse values. For instance, if several constants happen to have the same value, they can be mapped to the same location. Data buffers can often be reused at several different points in the program. However, this technique must be used with extreme caution, because subsequent versions of the program may not use the same values for the constants. A more generally applicable technique is to generate data on the fly rather than to store it. Of course, the code required to generate the data takes up space in the program, but when complex data structures are involved, there may be some net space savings from using code to generate data.

Minimizing the size of the instruction text of a program requires a mix of high-level program transformations and careful instruction selection. Encapsulating functions in subroutines can reduce the program size when done carefully. Because subroutines have overhead for parameter passing that is not obvious from the high-level language code, there is a minimum-sized function body for which a subroutine makes sense. Architectures that have variable-sized instruction lengths are particularly good candidates for careful coding to minimize the program size, which may require assembly language coding of key program segments. There may also be cases in which one or a sequence of instructions is much smaller than alternative implementations; for example, a multiply–accumulate instruction may be both smaller and faster than separate arithmetic operations.

When reducing the number of instructions in a program, one important technique is the proper use of subroutines. If the program performs identical operations repeatedly, these operations are natural candidates for subroutines. Even if the operations vary somewhat, you may be able to construct a properly parameterized subroutine

that saves space. Of course, when considering the code size savings, the subroutine linkage code must be considered in the equation. There is extra code not only in the subroutine body, but also in each call to the subroutine that handles parameters. In some cases, proper instruction selection may reduce the code size; this is particularly true in CPUs that use variable-length instructions.

Some microprocessor architectures support **dense instruction sets**, which are specially designed instruction sets that use shorter instruction formats to encode the instructions. The ARM Thumb instruction set and the MIPS-16 instruction set for the MIPS architecture are two examples of this type of instruction set. In many cases, a microprocessor that supports the dense instruction set also supports the normal instruction set, although it is possible to build a microprocessor that executes only the dense instruction set. Special compilation modes produce the program in terms of the dense instruction set. Of course, the program size varies with the type of program, but programs using the dense instruction set are often 70%–80% of the size of the standard instruction set equivalents.

---

## 5.10 Program validation and testing

Complex systems need testing to ensure that they work as they are intended. However, bugs can be subtle, particularly in embedded systems, where specialized hardware and real-time responsiveness make programming more challenging. Fortunately, there are many available techniques for software testing that can help us generate a comprehensive set of tests to ensure that our system works properly. We examine the role of validation in the overall design methodology in [Section 9.6](#). In this section, we concentrate on nuts-and-bolts techniques for creating a good set of tests for a given program.

The first question we must ask ourselves is how much testing is enough. Clearly, we cannot test the program for every possible combination of inputs. Because we cannot implement an infinite number of tests, we naturally ask ourselves what a reasonable standard of thoroughness is. One of the major contributions of software testing is to provide us with standards of thoroughness that make sense. Following these standards does not guarantee that we will find all bugs. However, by breaking the testing problem into subproblems and analyzing each subproblem, we can identify testing methods that provide reasonable amounts of testing, while keeping the testing time within reasonable bounds.

We can use various combinations of two major types of testing strategies:

- **Black-box** methods generate tests without looking at the internal structure of the program.
- **Clear-box** (also known as **white-box**) methods generate tests based on the program structure.

In this section, we cover both types of tests, which complement each other by exercising programs in very different ways.

### 5.10.1 Clear-box testing

The control/data flow graph extracted from a program's source code is an important tool in developing clear-box tests for the program. To test the program adequately, we must exercise both its control and data operations.

In order to execute and evaluate these tests, we must be able to control the variables in the program and observe the results of computations, much as in manufacturing testing. In general, we may need to modify the program to make it more testable. By adding new inputs and outputs, we can usually substantially reduce the effort required to find and execute the test. No matter what we are testing, we must accomplish the following three things in a test:

- Provide the program with inputs that exercise the test that we are interested in.
- Execute the program to perform the test.
- Examine the outputs to determine whether the test was successful.

Example 5.10 illustrates the importance of observability and controllability in software testing.

### Example 5.10: Controlling and Observing Programs

Let's first consider controllability by examining the following FIR filter with a limiter:

```
firout = 0.0; /*initialize filter output */
/*compute buff*c in bottom part of circular buffer */
for (j = curr, k = 0; j < N; j++, k++)
    firout += buff[j] *c[k];
/*compute buff*c in top part of circular buffer */
for (j = 0; j < curr; j++, k++)
    firout += buff[j] *c[k];
/*limit output value */
if (firout >100.0) firout = 100.0;
if (firout <-100.0) firout = -100.0;
```

The above code computes the output of a FIR filter from a circular buffer of values, and then, limits the maximum filter output (much as an overloaded speaker will hit a range limit). If we want to test whether the limiting code works, we must be able to generate two out-of-range values for `firout`: positive and negative. To do this, we must fill the FIR filter's circular buffer with  $N$  values in the proper range. Although there are many sets of values that will work, it will still take time for us to set up the filter output for each test properly.

This code also illustrates an observability problem. If we want to test the FIR filter itself, we look at the value of `firout` before the limiting code executes. We could use a debugger to set breakpoints in the code, but this is an awkward way to perform a large number of tests. If we want to test the FIR code independently of the limiting code, we need to add a mechanism to observe `firout` independently.

Being able to perform this process for many tests entails some amount of drudgery, which can be alleviated with good program design that simplifies controllability and observability.

The next task is to determine the set of tests to be performed. We need to perform many different types of tests to be confident that we have identified a large fraction of the existing bugs. Even if we thoroughly test the program using one criterion, that criterion ignores other aspects of the program. Over the next few pages, we will describe several very different criteria for program testing.

#### Execution paths

The most fundamental concept in clear-box testing is the path of execution through a program. Previously, we considered paths for performance analysis; we are now concerned with making sure that a path is covered and determining how to ensure that the path is in fact executed. We want to test the program by forcing the program to execute along the chosen paths. We force the execution of a path by giving it inputs that cause it to take the appropriate branches. The execution of a path exercises both the control and data aspects of the program. The control is exercised as we take branches; both the computations leading up to the branch decision and other computations that are performed along the path exercise the data aspects.

Is it possible to execute every complete path in an arbitrary program? The answer is no, because the program may contain a `while` loop that is not guaranteed to terminate. The same is true for any program that operates on a continuous stream of data, because we cannot arbitrarily define the beginning and end of the data stream. If the program always terminates, there are indeed a finite number of complete paths that can be enumerated from the path graph. This leads us to the next question: Does it make sense to exercise every path? The answer to this question is *no* for most programs, because the number of paths, especially for any program with a loop, is extremely large. However, the choice of an appropriate subset of paths to test requires some thought.

Example 5.11 illustrates the consequences of two different choices of testing strategies.

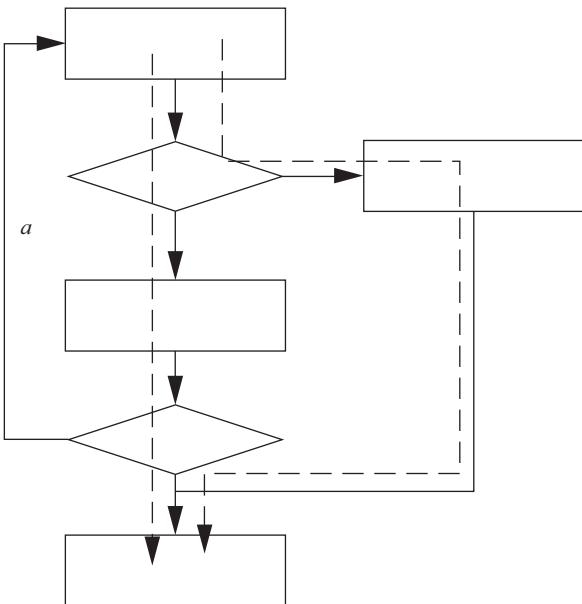
#### Example 5.11: Choosing the Paths to Test

We have at least two reasonable ways to choose a set of paths in a program to test:

- Execute every statement at least once.
- Execute every direction of a branch at least once.

These conditions are equivalent for structured programming languages without gotos, but are not the same for unstructured code. Most assembly language is unstructured, and state machines may be coded in high-level languages with gotos.

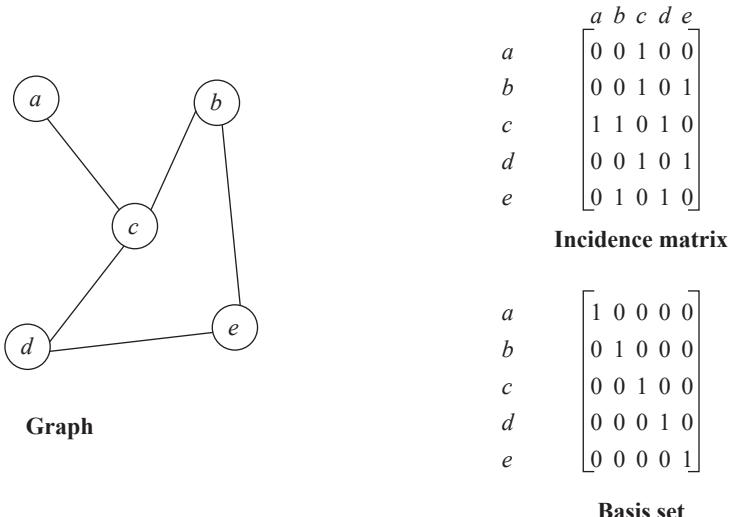
To understand the difference between statement and branch coverage, consider this CDFG:



We can execute every statement at least once by executing the program along two distinct paths. However, this leaves branch *a* out of the lower conditional uncovered. To ensure that we have executed along every edge in the CDFG, we must execute a third path through the program. This path does not test any new statements, but it does cause *a* to be exercised.

How do we choose a set of paths that adequately covers the program's behavior? Intuition tells us that a relatively small number of paths should be able to cover most practical programs. Graph theory helps us to get a quantitative handle on the different paths required. In an undirected graph, we can form any path through the graph from combinations of **basis paths**. Unfortunately, this property does not strictly hold for directed graphs such as CDFGs, but this formulation still helps us to understand the nature of selecting a set of covering paths through a program. The term "basis set" comes from linear algebra. Fig. 5.24 shows how to evaluate the basis set of a graph. The graph is represented as an **incidence matrix**. Each row and column represents a node; 1 is entered for each node pair connected by an edge. We can use standard linear algebra techniques to identify the basis set of the graph. Each vector in the basis set represents a primitive path. We can form new paths by adding the vectors modulo 2. In general, there is more than one basis set for a graph.

The basis set property provides a metric for test coverage. If we cover all of the basis paths, we can consider the control flow to be adequately covered. Although the basis set measure is not entirely accurate because the directed edges of the

**FIGURE 5.24**

The matrix representation of a graph and its basis set.

CDFG may make some combinations of paths infeasible, it does provide a reasonable and justifiable measure of the test coverage.

A simple measure known as **cyclomatic complexity** [McC76] allows us to measure the control complexity of a program. Cyclomatic complexity is an upper bound on the size of the basis set. If  $e$  is the number of edges in the flow graph,  $n$  is the number of nodes, and  $p$  is the number of components in the graph, the cyclomatic complexity is given by

$$M = e - n + 2p \quad (5.1)$$

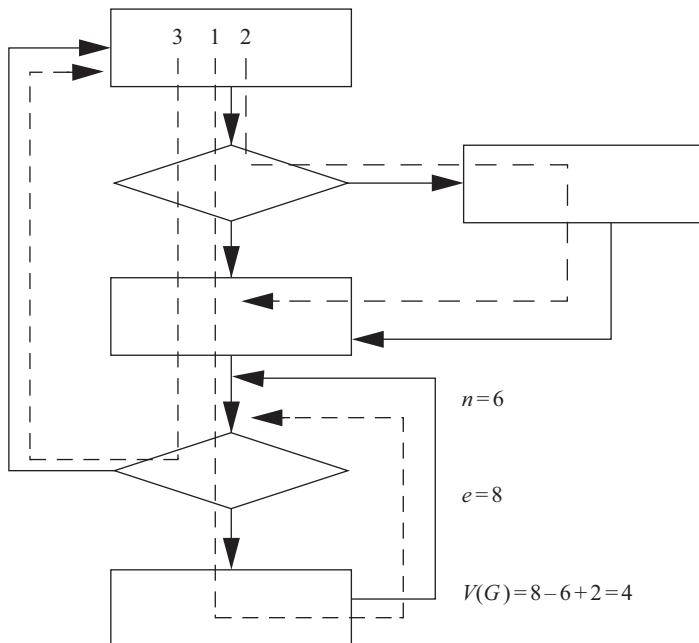
For a structured program,  $M$  can be computed by counting the number of binary decisions in the flow graph and adding 1. If the CDFG has higher-order branch nodes,  $b - 1$  is added for each  $b$ -way branch. In the example of Fig. 5.25, the cyclomatic complexity evaluates to 4. Because there are actually only three distinct paths in the graph, the cyclomatic complexity in this case is an overly conservative bound.

Cyclomatic complexity can be used to identify code that will be difficult to test. A maximum cyclomatic complexity of 10 is widely used [McC76, Wat96].

Another way of looking at control flow-oriented testing is to analyze the conditions that control the conditional statements. Consider the following `if` statement:

```
if ((a == b) || (c >= d)) { ... }
```

This complex condition can be exercised in several different ways. If we want to exercise the paths through this condition fully, it is prudent to exercise the conditional's elements in ways relating to their own structure and not just the structure

**FIGURE 5.25**

Cyclomatic complexity.

of the paths through them. A simple condition testing strategy is known as **branch testing** [Mye79]. This strategy requires the true and false branches of a conditional and every simple condition in the conditional's expression to be tested at least once.

Example 5.12 illustrates branch testing.

---

### Example 5.12: Condition Testing with the Branch Testing Strategy

Assume that the code below is what we meant to write.

```
if (a || (b >= c)) { printf("OK\n"); }
```

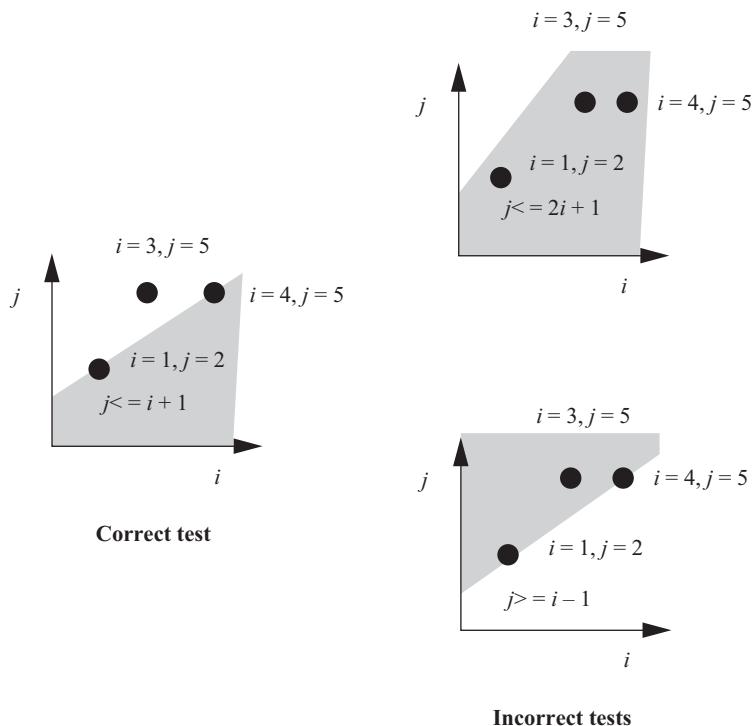
The code that we mistakenly wrote instead follows:

```
if (a && (b >= c)) { printf("OK\n"); }
```

If we apply branch testing to the code that we wrote, one of the tests will use these values:  $a = 0$ ,  $b = 3$ ,  $c = 2$  (making  $a$  false and  $b \geq c$  true). In this case, the code should print the OK term [ $0 \mid (3 \geq 2)$  is true], but instead doesn't print [ $0 \&\& (3 \geq 2)$  evaluates to false]. That test picks up the error.

---

Another more sophisticated strategy for testing conditionals is known as **domain testing** [How82], as illustrated in Fig. 5.26. Domain testing concentrates on linear

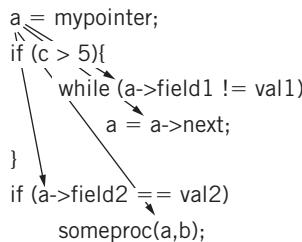
**FIGURE 5.26**

Domain testing for a pair of values.

inequalities. In the figure, the inequality that the program should use for the test is  $j \leq i + 1$ . We test the inequality with three test points—two on the boundary of the valid region and a third outside the region, but between the  $i$  values of the other two points. When we make some common mistakes in typing the inequality, these three tests are sufficient to uncover them, as shown in the figure.

A potential problem with path coverage is that the paths that are chosen to cover the CDFG may not have any important relationship to the program's function. Another testing strategy known as **data flow testing** makes use of **def-use analysis** (short for definition-use analysis). It selects paths that have some relationship to the program's function.

The terms "def" and "use" come from compilers, which use def-use analysis for optimization [Aho06]. A variable's value is **defined** when an assignment is made to the variable; it is **used** when it appears on the right side of an assignment (sometimes called a **C-use** for computation use) or in a conditional expression (sometimes called **P-use** for predicate use). A **def-use pair** is a definition of a variable's value and the use of that value. Fig. 5.27 shows a code fragment and all of the def-use pairs for the first assignment to `a`. Def-use analysis can be performed on a program using iterative

**FIGURE 5.27**

Definitions and uses of variables.

algorithms. Data flow testing chooses tests that exercise selected def-use pairs. The test first causes a certain value to be assigned at the definition, and then, observes the result at the use point to be sure that the desired value arrived there. Frankl and Weyuker [Fra88] defined criteria for choosing which def-use pairs to exercise to satisfy a well-behaved adequacy criterion.

### Testing loops

We can write some specialized tests for loops. Because loops are common and often perform important steps in the program, it is worth developing loop-centric testing methods. If the number of iterations is fixed, testing is relatively simple. However, many loops have bounds that are executed at run time.

Consider first the case of a single loop:

```

for (i = 0; i < terminate(); i++)
    proc(i,array);

```

It would be too expensive to evaluate the above loop for all possible termination conditions. However, there are several important cases that we should try at a minimum:

1. skipping the loop entirely [if possible, such as when `terminate()` returns 0 on its first call];
2. one loop iteration;
3. two loop iterations;
4. if there is an upper bound  $n$  on the number of loop iterations (which may come from the maximum size of an array), a value that is significantly below that maximum number of iterations; and
5. tests near the upper bound on the number of loop iterations, that is,  $n - 1$ ,  $n$ , and  $n + 1$ .

We can also have nested loops, like this:

```

for (i = 0; i < terminate1(); i++)
    for (j = 0; j < terminate2(); j++)
        for (k = 0; k < terminate3(); k++)
            proc(i,j,k,array);

```

There are many possible strategies for testing nested loops. One thing to keep in mind is which loops have fixed versus variable numbers of iterations. Beizer [Bei90] suggested an inside-out strategy for testing loops with multiple variable iteration bounds. First, concentrate on testing the innermost loop as above; the outer loops should be controlled to their minimum numbers of iterations. After the inner loop has been thoroughly tested, the next outer loop can be tested more thoroughly, with the inner loop executing a typical number of iterations. This strategy can be repeated until the entire loop nest has been tested. Clearly, nested loops can require a large number of tests. It may be worthwhile to insert testing code to allow greater control over the loop nest for testing.

### 5.10.2 Black-box testing

Black-box tests are generated without knowledge of the code being tested. When used alone, black-box tests have a low probability of finding all of the bugs in a program. However, when used in conjunction with clear-box tests, they help to provide a well-rounded test set, because black-box tests are likely to uncover errors that are unlikely to be found by tests that are extracted from the code structure. Black-box tests can really work. For instance, when asked to test an instrument whose front panel was run by a microcontroller, one acquaintance of the author used his hand to depress all of the buttons simultaneously. The front panel immediately locked up. This situation could occur in practice if the instrument were placed face-down on a table, but the discovery of this bug would be very unlikely via clear-box tests.

One important technique is to take tests directly from the specification for the code under design. The specification should state which outputs are expected for certain inputs. Tests should be created that provide specified outputs and evaluate whether the results also satisfy the inputs.

We can't test every possible input combination, but some rules of thumb help us to select reasonable sets of inputs. When an input can range across a set of values, it is a very good idea to test at the ends of the range. For example, if an input must be between 1 and 10, 0, 1, 10, and 11 are all important values to test. We should be sure to consider tests both within and outside the range, such as testing values within the range and outside the range. We may want to consider tests well outside the valid range as well as boundary condition tests.

**Random tests** form one category of black-box testing. Random values are generated with a given distribution. The expected values are computed independently of the system, and then, the test inputs are applied. A large number of tests must be applied for the results to be statistically significant, but the tests are easy to generate.

Another scenario is to test certain types of data values. For example, integer-valued inputs can be generated at interesting values such as 0, 1, and values near the maximum end of the data range. Illegal values can be tested as well.

**Regression tests** form an extremely important category of tests. When tests are created during earlier stages in the system design or for previous versions of the system, those tests should be saved to apply to the later versions of the system. Clearly,

#### Random tests

#### Regression tests

**Numerical accuracy**

unless the system specification has changed, the new system should be able to pass the old tests. In some cases, old bugs can creep back into systems, such as when an old version of a software module is inadvertently installed. In other cases, regression tests simply exercise the code in different ways than would be done for the current version of the code, and therefore, possibly exercise different bugs.

Some embedded systems, particularly digital signal processing systems, lend themselves to numerical analysis. Signal processing algorithms are frequently implemented with limited-range arithmetic to save hardware costs. Aggressive data sets can be generated to stress the numerical accuracy of the system. These tests can often be generated from the original formulas without reference to the source code.

### 5.10.3 Evaluating functional tests

How much testing is enough? Horgan and Mathur [Hor96] evaluated the coverage of two well-known programs, namely *TeX* and *awk*. They used functional tests for these programs that had been developed over several years of extensive testing. Upon applying those functional tests to the programs, they obtained the code coverage statistics shown in Fig. 5.28. The columns refer to various types of test coverage: *block* refers to basic blocks, *decision* refers to conditionals, *P-use* refers to the use of a variable in a predicate (decision), and *C-use* refers to variable use in a nonpredicate computation. These results are at least suggestive that functional testing does not fully exercise the code and that techniques that explicitly generate tests for various pieces of code are necessary to obtain adequate levels of code coverage.

Methodological techniques are important for understanding the quality of your tests. For example, if you keep track of the number of bugs that are tested each day, the data that you collect over time should show you some trends on the number of errors per page of code to expect on average, how many bugs are caught by certain types of tests, and so on. We address methodological approaches to quality control in more detail in Chapter 7.

One interesting method for analyzing the coverage of your tests is **error injection**. First, take your existing code and add bugs to it, keeping track of where the bugs were added. Thereafter, run your existing tests on the modified program. By counting the number of added bugs that your tests found, you can get an idea of how effective the tests are in uncovering the bugs that you haven't yet found. This method assumes that you can deliberately inject bugs that are of similar varieties to those created

	Block	Decision	P-use	C-use
TeX	85%	72%	53%	48%
awk	70%	59%	48%	55%

**FIGURE 5.28**

Code coverage of functional tests for TeX and awk (after Horgan and Mathur [Hor96]).

naturally by programming errors. If the bugs are too easy or too difficult to find, or simply require different types of tests, the bug injection's results will not be relevant. Of course, it is essential that you finally use the correct code and not the code with added bugs.

---

## 5.11 Safety and security

Software testing and eliminating bugs are important aspects of producing secure code. However, some methods that are important for secure software go beyond testing and some security-related bugs are important enough to warrant special attention. A complete discussion of secure program design is beyond our scope, having been the sole topic of several books [Gra03, Che07], but we can identify a few important techniques.

### Buffer overflows

Buffer overflows are a widely available avenue for attacks. If a program reads data from an external source in such a way as to overflow the space that is allocated for the buffer, that external source can change other parts of memory. These changes can be used to insert instructions into the program that are later executed and allow the attacker to take over the program. Transfers into arrays should always check the bounds of the buffer and enforce its limit.

### Buffer initialization

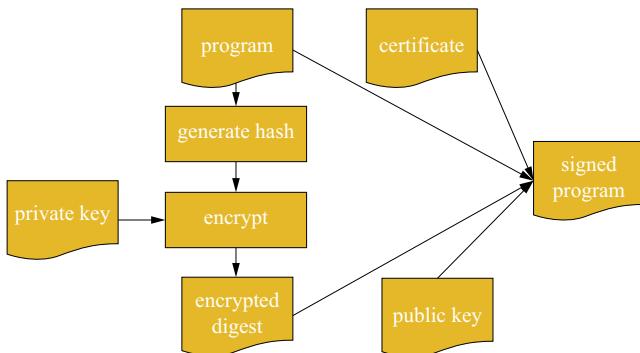
Buffers that have not been initialized to a known value, such as zero, can also result in security leaks. The memory that is used for a buffer is often recycled from some other use during execution. The former values of that memory may contain information that is useful to attackers. If the buffer is not initialized before use, its information may become available to attackers. Graff and van Wyck [Gra03] describe one such incident: a series of bugs caused an uninitialized buffer to be filled with part of the system's password file and for those data to be written out to a software distribution. Although no known problems resulted from this error, it made back-door attacks possible that would allow attackers to intrude into systems.

### Certificates

We often want to install code onto an embedded platform from outside sources. We want to be sure that the code comes from a trusted source, as installing code from a malicious agent would allow malware onto the platform. Establishing the trustworthiness of a source is managed by a **certification authority** [Koh78]. A source requests a **certificate** from the authority; the source may be required to pay the authority for the certificate. The certificate includes an identifier for the source and its encryption key; the certificate may also come with an expiration date. That certificate can then be used to distribute code: the code makes it difficult for an adversary to lie about where the code was produced. The recipient of a certificate can check the source identifier to be sure that it comes from the expected source, and then, use the associated key to decrypt code or messages.

### Code signing

**Code signing** combines the signing certificate with a **digital signature** of the code itself to create a code module with a verifiable source. The code signing process is outlined in Fig. 5.29. First, a **cryptographic hash function** is used to create a **hash** or **digest** of the program. That hash is a compact form of the program generated by a carefully designed function that makes it easy to generate the digest, but difficult

**FIGURE 5.29**

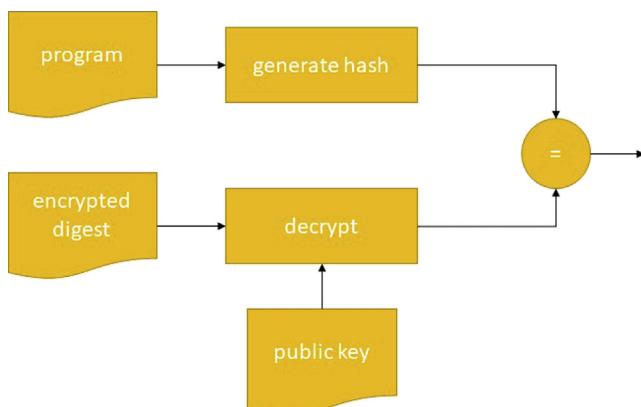
Code signing.

to reconstruct the original input from that digest. The SHA hash algorithms are examples of cryptographic hash functions [NIS15]. That digest is then encrypted using public key cryptography. The signer uses a private key to generate the encrypted version; a public key, which is available to everyone, can be used to decrypt. The original program, its encrypted digest, the public key used to generate the digest, and the signing certificate are then packaged to create a signed program.

The signature of a program can be checked before execution, as shown in Fig. 5.30. The encrypted digest is decrypted using the public key. It should be the same as the hash generated from the program. If the two do not match, the code has been modified after it was signed by the sender and should not be trusted.

### Passwords

Some programs may use passwords to authenticate users; for example, requiring login before allowing some system parameters to be changed. Passwords should be stored only in encrypted form.

**FIGURE 5.30**

Signature verification.

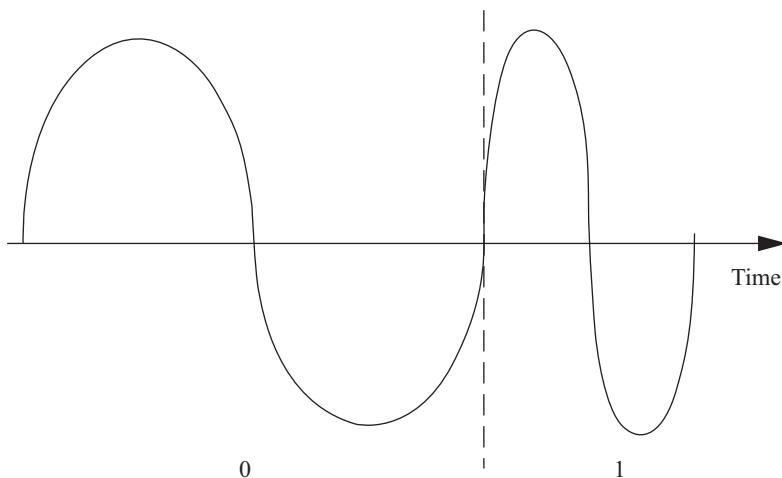
## 5.12 Design example: software modem

In this section, we design a simple modem such as we might use in a basic Internet-of-Things (IoT) node. Before jumping into the modem design itself, we discuss the principles of how to transmit a digital data communications link such as a radio signal or a wire. We will then go through the specification and discuss architecture, module design, and testing.

### 5.12.1 Theory of operation and requirements

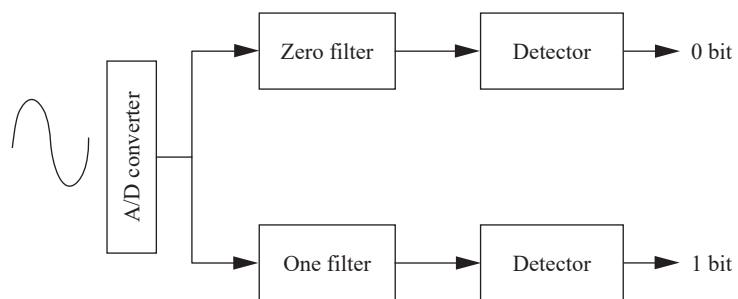
The modem will use **frequency-shift keying (FSK)**, which is a technique used in 1200-baud modems. Keying alludes to Morse code–style keying. As shown in Fig. 5.31, the FSK scheme transmits sinusoidal tones, with 0 and 1 assigned to different frequencies. Sinusoidal tones are much better suited to transmission over analog phone lines than are the traditional high and low voltages of digital circuits. The 01 bit patterns create the chirping sound that is characteristic of modems. Higher-speed modems are backward compatible with the 1200-baud FSK scheme and begin a transmission with a protocol to determine which speed and protocol should be used.

The scheme that is used to translate the audio input into a bit stream is illustrated in Fig. 5.32. The analog input is sampled and the resulting stream is sent to two digital filters (such as a FIR filter). One filter passes frequencies in the range that represents a 0 and rejects the 1-band frequencies, whereas the other filter does the converse. The outputs of the filters are sent to detectors, which compute the average value of the signal over the past  $n$  samples. When the energy goes above a threshold value, the appropriate bit is detected.



**FIGURE 5.31**

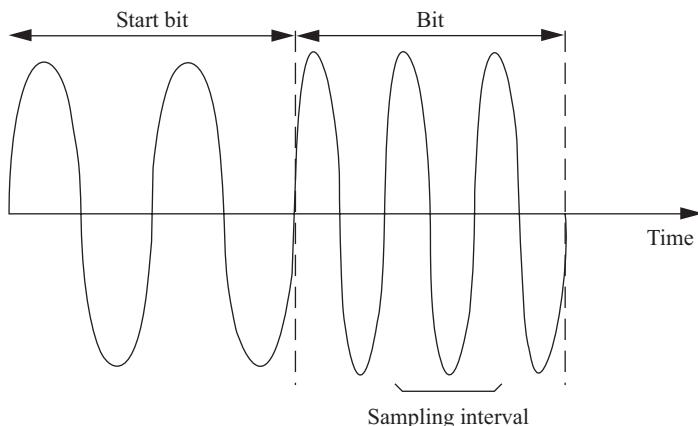
Frequency-shift keying.

**FIGURE 5.32**

The FSK detection scheme.

We will send data in units of 8-bit bytes. The transmitting and receiving modems agree in advance on the length of time during which a bit will be transmitted (otherwise known as the baud rate). However, the transmitter and receiver are physically separated and are therefore not synchronized in any way. The receiving modem does not know when the transmitter has started to send a byte. Furthermore, even when the receiver does detect a transmission, the clock rates of the transmitter and receiver may vary somewhat, causing them to fall out of sync. In both cases, we can reduce the chances of error by sending the waveforms for a longer time.

The receiving process is illustrated in Fig. 5.33. The receiver will detect the start of a byte by looking for a start bit, which is always 0. By measuring the length of the start bit, the receiver knows where to look for the start of the first bit. However, because the receiver may have slightly misjudged the start of the bit, it does not immediately try to detect the bit. Instead, it runs the detection algorithm at the predicted middle of the bit.

**FIGURE 5.33**

Receiving bits in the modem.

The modem will not include the physical layer interface such as the radio. We will assume that we have analog audio inputs and outputs for sending and receiving. The modem's job is to generate a waveform that can be transmitted. We will also run at a much slower bit rate than 1200 baud to simplify the implementation. Furthermore, we will not implement a serial interface to a host, but rather put the transmitter's message in memory and save the receiver's result in memory as well. Given those understandings, let's fill out the requirements table.

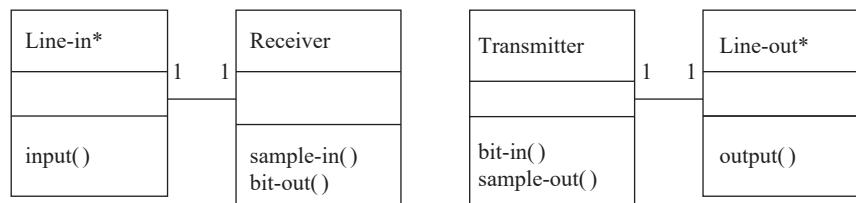
Name	Modem
Purpose	A fixed baud rate, frequency-shift keyed modem.
Inputs	Analog sound input, reset button.
Outputs	Analog sound output, LED bit display.
Functions	Transmitter: Sends data stored in microprocessor memory in 8-bit bytes. Sends start bit for each byte equal in length to one bit. Receiver: Automatically detects bytes and stores results in main memory.
Performance	1200 baud.
Manufacturing cost	Dominated by microprocessor and radio link.
Power	Compatible with IoT node.
Physical size and weight	Compatible with IoT node.

### 5.12.2 Specification

The basic classes for the modem are shown in Fig. 5.34. The classes include physical classes for line-in and line-out plus classes for the receiver and transmitter.

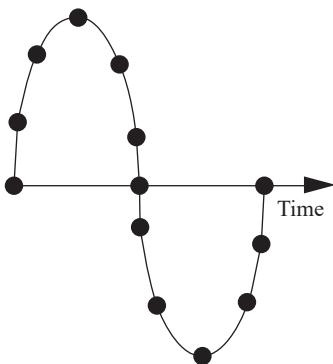
### 5.12.3 System architecture

The modem consists of one small subsystem (the interrupt handlers for the samples) and two major subsystems (transmitter and receiver). Two sample interrupt handlers



**FIGURE 5.34**

Class diagram for the modem.



```
float sine_wave[N_SAMP] =  
{ 0.0, 0.5, 0.866, 1,  
0.866, 0.5, 0.0, -0.5,  
0.866, -1.0, -0.866, -0.5,  
0};
```

Table

Analog waveform and samples

FIGURE 5.35

Waveform generation by table lookup.

are required: one for input and another for output, but they are very simple. The transmitter is simpler, so let's consider its software architecture first.

The best way to generate waveforms that retain the proper shape over long intervals is **table lookup**. Software oscillators can be used to generate periodic signals, but numerical problems limit their accuracy. Fig. 5.35 shows an analog waveform with sample points and the C code for these samples. Table lookup can be combined with interpolation to generate high-resolution waveforms without excessive memory costs, which is more accurate than oscillators because no feedback is involved. The required number of samples for the modem can be found by experimentation with the analog/digital converter and sampling code.

The structure of the receiver is considerably more complex. The filters and detectors of Fig. 5.32 can be implemented with circular buffers. However, that module must feed a state machine that recognizes the bits. The recognizer state machine must use a timer to determine when to start and stop computing the filter output average based on the starting point of the bit. It must then determine the nature of the bit at the proper interval. It must also detect the start bit and measure it using the counter. The receiver sample interrupt handler is a natural candidate to double as the receiver timer, because the receiver's time points are relative to the samples.

The hardware architecture is relatively simple. In addition to the analog/digital and digital/analog converters, a timer is required. The amount of memory that is required to implement the algorithms is relatively small.

#### 5.12.4 Component design and testing

The transmitter and receiver can be tested relatively thoroughly on the host platform because the timing-critical code only delivers data samples. The transmitter's output is quite easy to verify, particularly if the data are plotted. A testbench can be

constructed to feed the receiver code sinusoidal inputs and to test its bit recognition rate. It is a good idea to test the bit detectors first before testing the complete receiver operation. One potential problem in host-based testing of the receiver is encountered when library code is used for the receiver function. If a DSP library for the target processor is used to implement the filters, a substitute must be found or built for the host processor testing. The receiver must then be retested when it is moved to the target system to ensure that it still functions properly with the library code.

Care must be taken to ensure that the receiver does not run too long and miss its deadline. Because the bulk of the computation is in the filters, it is relatively simple to estimate the total computation time early in the implementation process.

### 5.12.5 System integration and testing

There are two ways to test the modem system: by having the modem's transmitter send bits to its receiver and by connecting two different modems. The ultimate test is to connect two different modems, particularly modems that are designed by different people, to be sure that incompatible assumptions or errors were not made. However, single-unit testing, which is called **loop-back** testing in the telecommunications industry, is simpler and a good first step. Loop-back can be performed in two ways. First, a shared variable can be used to pass data directly from the transmitter to the receiver. Second, an audio cable can be used to plug the analog output into the analog input. In this case, it is also possible to inject analog noise to test the resiliency of the detection algorithm.

---

## 5.13 Design example: digital still camera

In this section, we design a simple digital still camera (DSC). Video cameras share some similarities with DSCs but are different in several ways, most notably their emphasis on streaming media. We will study a design example for one subsystem of a video camera in [Chapter 8](#).

### 5.13.1 Theory of operation and requirements

To understand the DSC better, we will first consider the digital photography process. A modern digital camera performs a great many steps:

- Determine the exposure and focus.
- Capture the image.
- Develop the image.
- Compress the image.
- Generate and store the image as a file.

In addition to taking the photo, the camera must perform several other important operations, often simultaneously with the picture-taking process. For example, it

must update the camera's electronic display. It must also listen for button presses from the user, which may command operations that modify the current operation of the camera. The camera should also provide a browser with which the user can review the stored images.

#### Imaging terminology

Much of this terminology comes largely from film photography; for example, the decisions that are made while developing a digital photo are similar to those made in the development of a film image, even though the steps to carry out those decisions are very different. Many variations are possible on all of these steps, but the basic process is common to all DSCs. A few basic terms are useful: the image is divided into **pixels**; a pixel's brightness is referred to as its **luminance**; a color pixel's brightness in a particular color is known as **chrominance**.

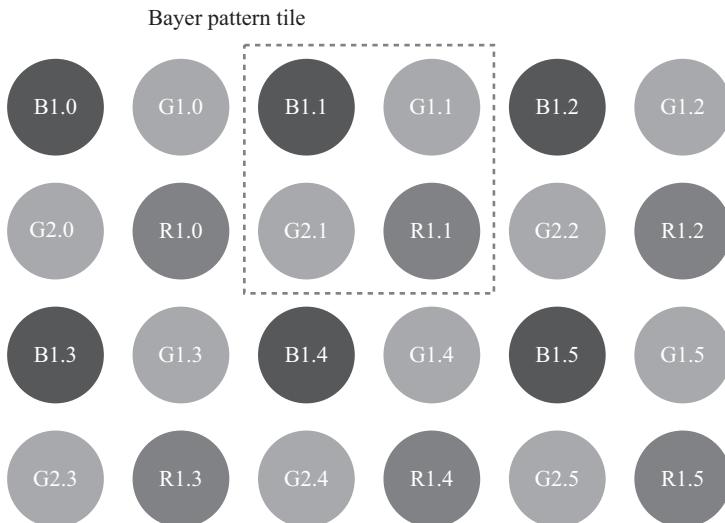
#### Imaging algorithms

The camera can use the image sensor data to drive the exposure-setting process. Several algorithms are possible, but all involve starting with a test exposure and using some search algorithm to select the final exposure. Exposure setting may also use any of several different metrics to judge the exposure. The simplest metric is the average luminance of a pixel. The camera may evaluate some function of several points in the image. It may also evaluate the image's **histogram**. The histogram is composed by sorting the pixels into bins by luminance; 256 bins is a common choice for the resolution of the histogram. The histogram gives us more information than does a single average. For instance, we can tell whether many pixels are overexposed or underexposed by looking at the bins at the extremes of luminance.

Three major approaches are used to determine focus: active rangefinding, phase detection, and contrast detection. Active rangefinding uses a pulse that is sent out, and the time-of-flight of the reflected and returned pulse is measured to determine the distance. Ultrasound was used in one early autofocus system, the Polaroid SX-70, but infrared pulses are more commonly used today. Phase detection compares light from opposite sides of the lens, creating an optical rangefinder. Contrast detection relies on the fact that out-of-focus edges do not display the sharp changes in luminance of in-focus edges. The simplest algorithms for contrast detection evaluate the focus at predetermined points in the image, which may require the photographer to move the camera to place a suitable edge at one of the autofocus spots.

Two major types of image sensors are used in modern cameras [Nak05]: charged-coupled devices (CCDs) and CMOS. For our purposes, the operations of these two in a digital still camera are equivalent.

Developing the image involves both putting the image into a usable form and improving its quality. The most basic operation required for a color image is to interpolate a full color value for each pixel. Most image sensors capture color images using a **color filter array**—color filters that cover a single pixel. The filters usually capture one of the primary colors, namely red, green, and blue, and are arranged in a two-dimensional pattern. The first such color filter array was proposed by Bayer [Bay76]. His pattern is still widely used and is known as the **Bayer pattern**. As shown in Fig. 5.36, the Bayer pattern is a  $2 \times 2$  array with two green, one blue, and one red pixels. More green pixels are used because the eye is most sensitive to green, which Bayer viewed as a simple luminance signal. Because each image

**FIGURE 5.36**

A color filter array arranged in a Bayer pattern.

sensor pixel captures only one of the primaries, the other two primaries must be interpolated for each pixel. This process is known as **Bayer pattern interpolation** or **demosaicing**. The simplest interpolation algorithm is a simple average, which can also be used as a low-pass filter. For example, we can interpolate the missing values from the green pixel G2.1 as

$$\begin{aligned} R2.1 &= (R1.0 + R1.1)/2 \\ B2.1 &= (B1.1 + B1.4)/2 \end{aligned}$$

We can use more information to interpolate missing green values. For example, the green value that is associated with red pixel R1.1 can be interpolated from the four nearest green pixels:

$$G1.1 = (G1.1 + G2.1 + G2.2 + G1.4)/4$$

However, this simple interpolation algorithm introduces color fringes at edges. More sophisticated interpolation algorithms exist to minimize color fringing while maintaining other aspects of image quality.

Development also determines and corrects for **color temperature** in the scene. Different light sources can be composed of light of different colors; color can be described as the temperature of a black body that will emit radiation of that frequency. The human visual system automatically corrects for color temperature. For example, we do not notice that fluorescent lights emit a greenish cast. However, a photo that is taken without color temperature correction will display a cast from the color of light used to illuminate the scene. A variety of algorithms exists to determine the color

temperature. A set of chrominance histograms is often used to judge color temperature. Once a color correction is determined, it can be applied to the pixels in the image.

Many cameras also apply **sharpening algorithms** to reduce some of the effects of pixilation in digital image capture. Once again, various sharpening algorithms are used, which apply filters to sets of adjacent pixels.

Some cameras offer a RAW mode of image capture. The resulting RAW file holds the pixel values without processing. RAW capture allows the user to develop the image manually and offline using sophisticated algorithms and programs. Although generic RAW file formats exist, most camera manufacturers use proprietary RAW formats. Some uncompressed image formats also exist, such as TIFF [Ado92].

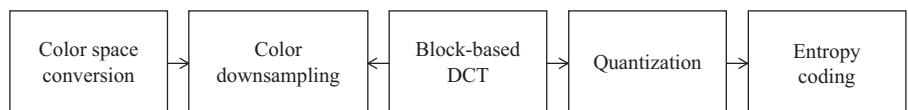
#### Image compression

Compression reduces the amount of storage that is required for the image. Some **lossless compression** methods are used that do not throw away information from the image. However, most images are stored using **lossy compression**. A lossy compression algorithm throws away information in the image, so that the decompression process cannot reproduce an exact copy of the original image. A number of compression techniques have been developed to reduce the storage space required for an image substantially, without noticeably affecting the image quality. The most common family of compression algorithms is **JPEG** [CCI92] (JPEG stands for **Joint Photographic Experts Group**). The JPEG standard was extended as JPEG 2000, but classic JPEG is still widely used. The JPEG standard itself contains a large number of options, not all of which need to be implemented to conform to the standard.

As shown in Fig. 5.37, the typical compression process used for JPEG images has five main steps:

- color space conversion
- color downsampling
- block-based discrete cosine transform (DCT)
- quantization
- entropy coding

(We call this typical because the standard allows for several variations.) The first step puts the color image into a form that allows optimizations that are less likely to reduce image quality. Color can be represented by a number of different **color spaces** or combinations of colors that, when combined, form the full range of colors. We saw the red/green/blue (RGB) color space in the color filter array. For compression, we convert into the  $Y' C_R C_B$  color space:  $Y'$  is a luminance channel,  $C_R$  is a red channel,



**FIGURE 5.37**

The typical JPEG compression process.

and  $C_B$  is a blue channel. The conversion between these two color spaces is defined by the JPEG File Interchange Format (JFIF) standard [Ham92]:

$$\begin{aligned} y' &= + (0.299 * R'_D) \quad + (0.587 * G'_D) \quad + (0.114 * B'_D) \\ C_R &= 128 - (0.168736 * R'_D) \quad - (0.331264 * R'_D) \quad + (0.5 * R'_D) \\ C_B &= 128 - (0.5 * R'_D) \quad - (0.418688 * R'_D) \quad - (0.081312 * R'_D) \end{aligned}$$

Once in  $Y' C_R C_B$  form,  $C_R$  and  $C_B$  are generally reduced by downsampling. The **4:2:2** method downsamples both  $C_R$  and  $C_B$  to half of their normal resolution only in the horizontal direction. The **4:2:0** method downsamples both  $C_R$  and  $C_B$  both horizontally and vertically. Downsampling reduces the amount of data that are required, and is justified by the human visual system's lower acuity in chrominance. The three **color channels**,  $Y'$ ,  $C_R$ , and  $C_B$ , are processed separately.

The color channels are next separately broken into  $8 \times 8$  **blocks** (the term block specifically refers to an  $8 \times 8$  array of values in JPEG). The DCT is applied to each block. DCT is a frequency transform that produces an  $8 \times 8$  block of **transform coefficients**. It is reversible, so that the original values can be reconstructed from the transform coefficients. DCT does not itself reduce the amount of information in a block. Many highly optimized algorithms exist to compute the DCT, particularly for an  $8 \times 8$  DCT.

The lossiness of the compression process occurs in the quantization step. This step changes the DCT coefficients with the aim to do so in a way that allows them to be stored in fewer bits. Quantization is applied on the DCT rather than on the pixels because some image characteristics are easier to identify in the DCT representation. Specifically, because the DCT breaks up the block according to **spatial frequencies**, quantization often reduces the high frequency content of the block. This strategy tends to reduce the data set size during compression significantly while causing less noticeable visual artifacts.

The quantization is defined by an  $8 \times 8$  **quantization matrix**  $Q$ . A DCT coefficient  $G_{i,j}$  is quantized to a value  $B_{i,j}$  using the  $Q_{i,j}$  value from the quantization matrix:

$$B_{i,j} = \text{round} \left( \frac{G_{i,j}}{Q_{i,j}} \right)$$

The JPEG standard allows for different quantization matrices but gives a typical matrix that is widely used:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

This matrix tends to zeros in the lower right coefficients. Higher spatial frequencies (and therefore, finer detail) are represented by the coefficients in the lower and right parts of the matrix. Putting these to zero eliminates some fine detail from the block.

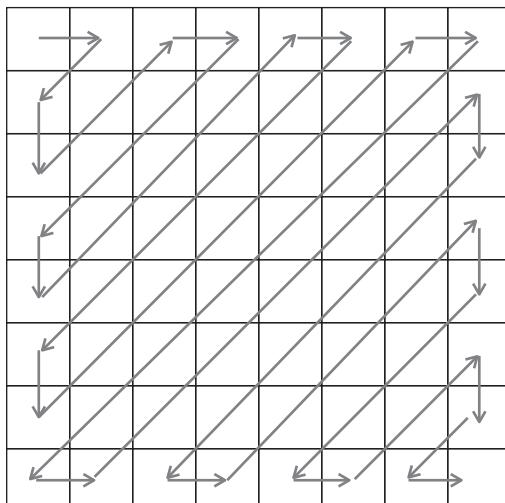
Quantization does not directly provide for a smaller representation of the image. Entropy coding (lossless encoding) recodes the quantized blocks in a form that requires fewer bits. JPEG allows multiple entropy coding algorithms to be used. The most common algorithm is Huffman coding. The encoding can be represented as a table that maps a fixed number of bits into a variable number of bits. This step encodes the difference between the current and previous coefficients, not the coefficient itself. Several different styles of encoding are possible: **baseline sequential** codes one block at a time; **baseline progressive** encodes corresponding coefficients of every block.

Coefficients are read from the coefficient matrix in the zig-zag pattern, as shown in Fig. 5.38. This pattern reads along diagonals from the upper left to the lower right, which corresponds to reading from the lowest to highest spatial frequency. If fine detail is reduced equally in both the horizontal and vertical dimensions, then zero coefficients are also arranged diagonally. The zig-zag pattern increases the length of sequences of zero coefficients in such cases. Long strings of zeros can be coded in a very small number of bits.

The requirements for the digital still camera are given in Fig. 5.39.

### 5.13.2 Specification

A digital still camera must comply with a number of standards. These standards generally govern the format of the output generated by the camera. Standards in



**FIGURE 5.38**

Zig-zag pattern for reading coefficients.

Name	Digital still camera
Purpose	Digital still camera with JPEG compression
Inputs	Image sensor, shutter button
Outputs	Display, flash memory
Functions	Determine exposure and focus, capture image, perform Bayer pattern interpolation, JPEG compression, store in flash file system
Performance	Take one picture in 2 sec.
Manufacturing cost	Approximately \$75
Power	Two AA batteries
Physical size and weight	Approx 4 in × 4 in × 1 in, less than 4 ounces.

**FIGURE 5.39**

Requirements for the digital still camera.

general allow for some freedom of implementation in certain aspects while still adhering to the standard.

#### File formats

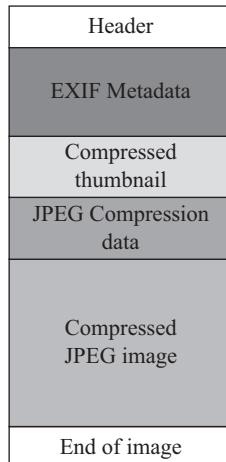
The **Tagged Image File Format (TIFF)** [Ado92] is often used to store uncompressed images, although it also supports several compression methods. Baseline TIFF specifies a basic format that also provides flexibility on the image size, bits per pixel, compression, and other aspects of image storage.

The JPEG standard itself allows many different options that can be followed in various combinations. A set of operations that is used to generate a compressed image is known as a **process**. The JFIF standard [Ham92] is a widely used file interchange format. It is compatible with the JPEG standard but specifies a number of items in more detail.

The **Exchangeable Image File Format (EXIF)** standard is widely used to extend the information stored in an image file further. As shown in Fig. 5.40, an EXIF file holds several types of data:

- The metadata section provides a wide range of information: date, time, location, and so on. The metadata are defined as attribute/value pairs. An EXIF file need not contain all possible attributes.
- A **thumbnail** is a smaller version of a file that is used for quick display. Thumbnails are widely used both by cameras to display the image on a small screen and on desktop computers to display a sample of the image more quickly. Storage of the thumbnail avoids the need to regenerate the thumbnail each time, saving both computation time and energy.
- JPEG compression data include tables, such as entropy coding and quantization tables, that are used to encode the image.
- The compressed JPEG image itself takes up the bulk of the space in the file.

The entire image storage process is defined by yet another standard, namely the Design rule for Camera File (DCF) standard [CIP10]. The DCF specifies three major

**FIGURE 5.40**


---

Structure of an EXIF file.

steps: JPEG compression, EXIF file generation, and DOS file allocation table (FAT) image storage. DCF specifies a number of aspects of file storage:

- The DCF image root directory is kept in the root directory and DCIM (for “digital camera images”).
- The directories within DCIM have names of eight characters, the first three of which are numbers between 100 and 999, giving the directory number. The remaining five characters are required to be upper-case alphanumeric.
- File names in DCF are eight characters long. The first four characters are upper-case alphanumeric, followed by a four-digit number between 0001 and 9999.
- Basic files in DCF are in EXIF version 2 format. The standard specifies a number of properties of these EXIF files.

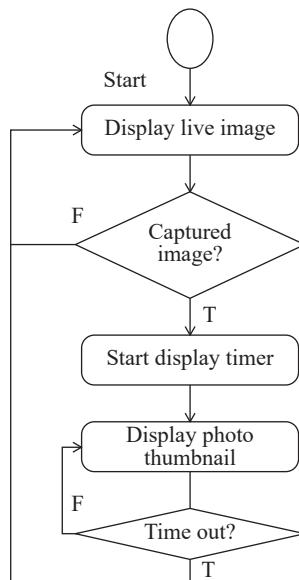
The Digital Print Order Format (DPOF) standard [DPO00] provides a standard way for camera users to order prints of selected photographs. Print orders can be captured in the camera, a home computer, or other devices, and transmitted to a photo-finisher or printer.

#### **Camera operating modes**

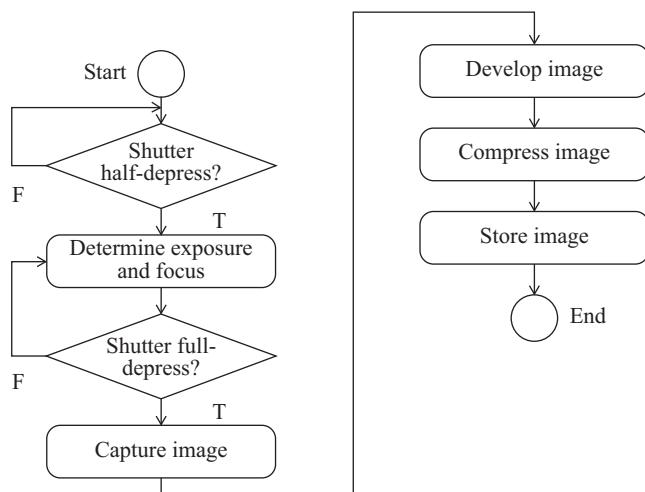
Even a simple point-and-shoot camera provides a number of options and modes. Two basic operations are fundamental: display a live view of the current image and capture an image.

[Fig. 5.41](#) shows a state diagram for the display. In normal operation, the camera repeatedly displays the latest image from the image sensor. After an image is captured, it is briefly shown on the display.

[Fig. 5.42](#) shows a state diagram for the picture-taking process. Depressing the shutter button halfway signals the camera to evaluate the exposure and focus. Once the shutter is fully depressed, the image is captured, developed, compressed, and stored.

**FIGURE 5.41**

State diagram for display operation.

**FIGURE 5.42**

State diagram for picture taking.

### 5.13.3 System architecture

The basic architecture of a digital still camera uses a controller to provide the basic sequencing for picture taking and camera operation. The controller calls on a number of other units to perform the various steps in each process.

Fig. 5.43 shows the basic classes in a digital still camera. The controller class implements the state diagrams for camera operation. The buttons and display classes provide an abstraction of the physical user interface. The image sensor abstracts the operation of the sensor. The picture developer provides algorithms for mosaicing, sharpening, and so on. The compression unit generates compressed image data. The file generator takes care of aspects of the file generation beyond compression. The file system performs basic file functions. Cameras may also provide other communication ports such as USB or Firewire.

Figure 5.44 shows a typical block diagram for digital still camera hardware. While some cameras will use the same processor for both system control and image processing, many cameras rely on separate processors for these two tasks. The DSP may be programmable or custom hardware.

A sequence diagram for taking a photo is shown in Figure 5.45. This sequence diagram maps the basic operations onto the units in the hardware architecture.

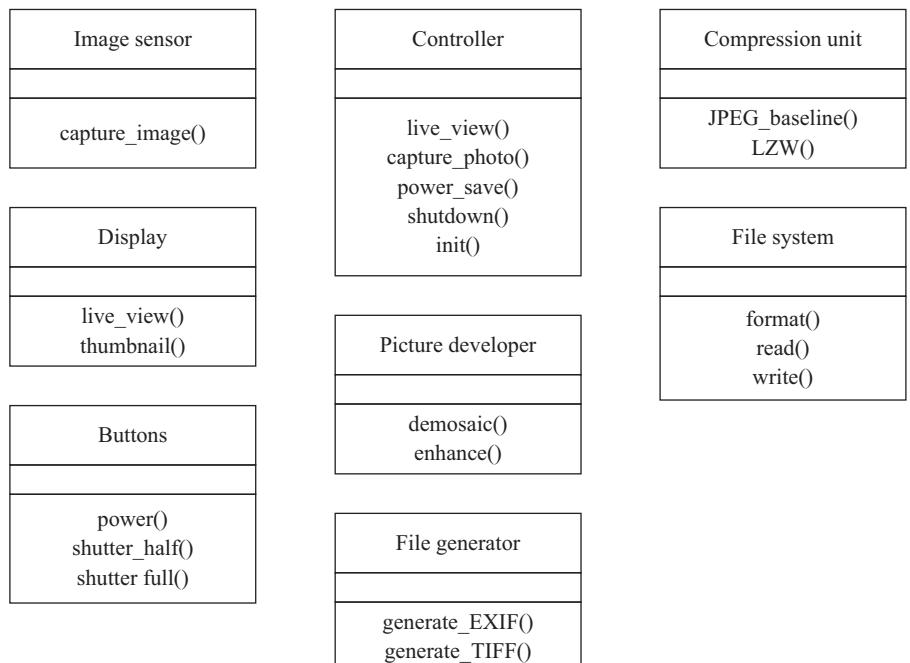
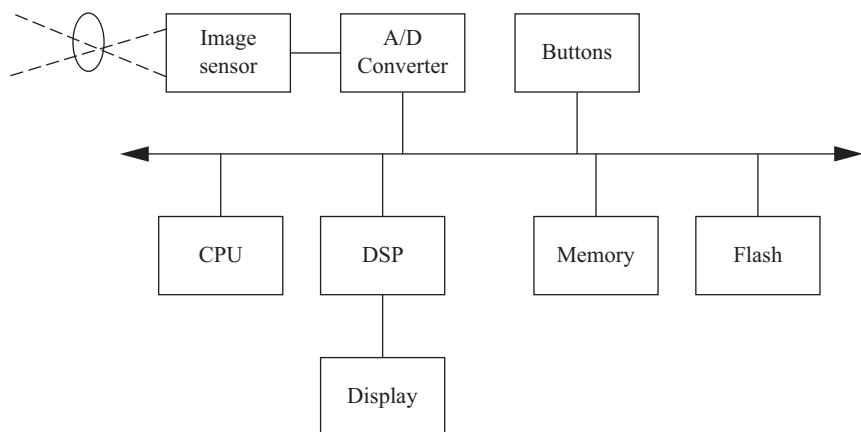


FIGURE 5.43

Basic classes in the digital still camera.

**FIGURE 5.44**

Computing platform for a digital still camera.

The design of buffering is very important in a digital still camera. Buffering affects the rate at which pictures can be taken, the energy consumed, and the cost of the camera. The image exists in several versions at different points in the process: the raw image from the image sensor; the developed image; the compressed image data; and the file. Most of these data is buffered in system RAM. However, displays may have their own memory to ensure adequate performance.

#### **5.13.4 Component design and testing**

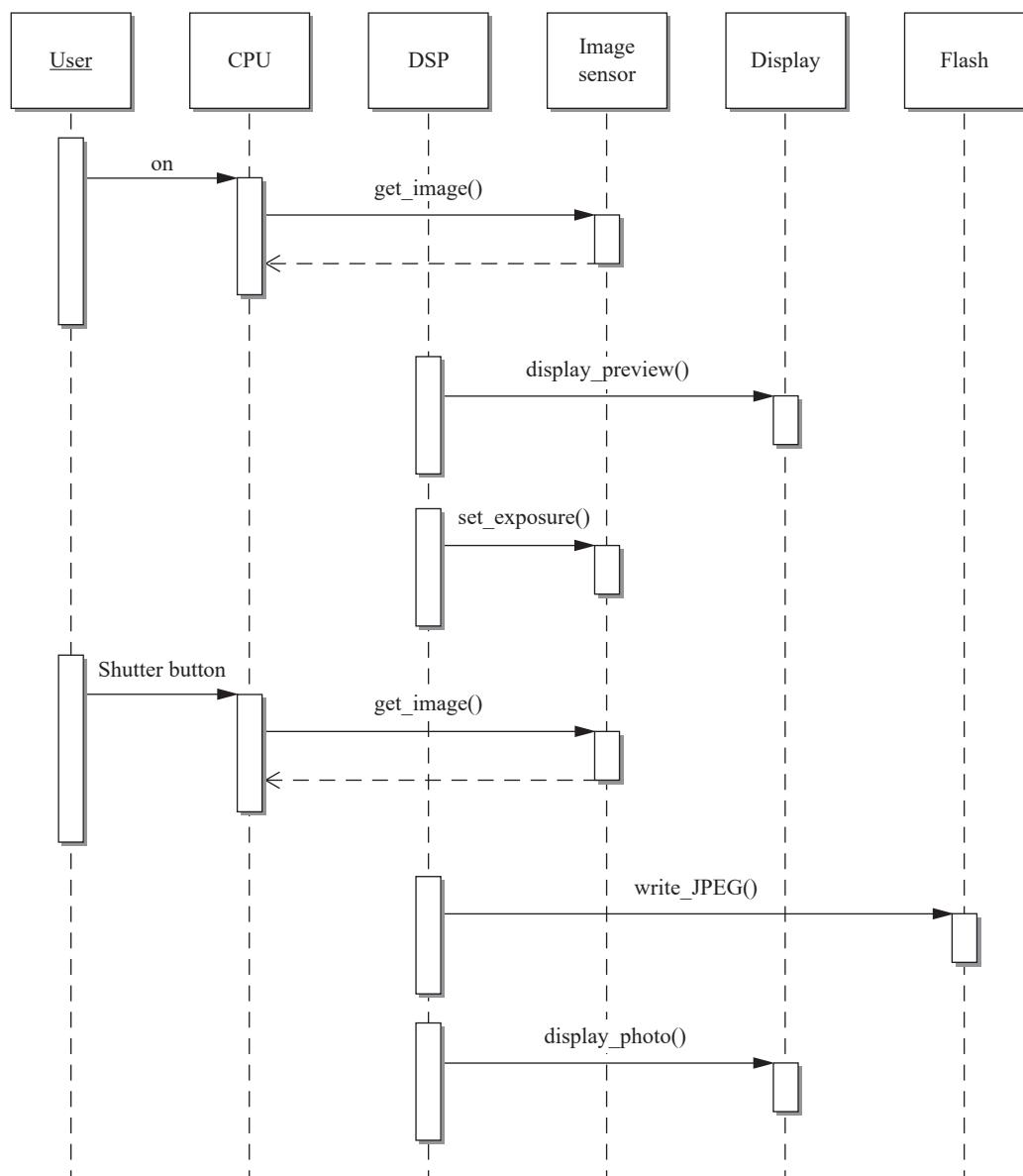
Components based on standards, such as JPEG or FAT, may be implemented using modules developed elsewhere. JPEG compression in particular may be implemented with special-purpose hardware. DCT accelerators are common. Some digital still camera engines use hardware units that produce a complete JFIF file from an image.

The multiple buffering points in the picture-taking process can help to simplify testing. Test file inputs can be introduced into the buffer by test scaffolding, and then, run with results in the output buffer checked against reference output.

#### **5.13.5 System integration and testing**

Buffers help to simplify system integration, although care must be taken to ensure that the buffers do not overlap in main memory.

Some tests can be performed by substituting pixel value streams for the sensor data. Final tests should make use of a target image so that qualities such as sharpness and color fidelity can be judged.

**FIGURE 5.45**

Sequence diagram for taking a picture with a digital still camera.

## 5.14 Summary

The program is a very fundamental unit of embedded system design and it usually contains tightly interacting code. Because we care about more than just functionality, we need to understand how programs are created. Because today's compilers do not take directives such as "compile this to run in less than 1 microsecond," we must be able to optimize programs ourselves for speed, power, and space. Our earlier understanding of computer architecture is critical to our ability to perform these optimizations. We also need to test programs to make sure they do what we want. Some of our testing techniques can also be useful in exercising the programs for performance optimization.

---

## What we learned

- We can use data flow graphs to model straight-line code and CDFGs to model complete programs.
- Compilers perform numerous of tasks, such as generating control flow, assigning variables to registers, creating procedure linkages, and so on.
- Remember the performance optimization equation: *execution time = program path + instruction timing*
- Memory and cache optimizations are very important to performance optimization.
- Optimizing for power consumption often goes hand in hand with performance optimization.
- Optimizing programs for size is possible, but don't expect miracles.
- Programs can be tested as black boxes (without knowing the code) or as clear boxes (by examining the code structure).
- Code signing can be used to authenticate software.

---

## Further reading

Aho, Sethi, and Ullman [Aho06] wrote a classic text on compilers, and Muchnick [Muc97] describes advanced compiler techniques in detail. A paper on the ATOM system [Sri94] provides a good description of instrumenting programs for gathering traces. Cramer et al. [Cra97] describe the Java just-in-time compiler. Li and Malik [Li97D] describe a method for statically analyzing program performance. Banerjee [Ban93, Ban94] describes loop transformations. Two books by Beizer, one on fundamental functional and structural testing techniques [Bei90] and the other on system-level testing [Bei84], provide comprehensive introductions to software testing, and as a bonus, are well written. Lyu [Lyu96] provides a good advanced survey of software reliability. Walsh [Wal97] describes a software modem implemented on an Arm processor.

---

## Questions

- Q5-1** Write C code for a state machine that implements a four-cycle handshake.
- Q5-2** Use the circular buffer functions to write a C function that accepts a new data value, puts it into the circular buffer, and then, returns the average value of all the data values in the buffer.
- Q5-3** Write C code for a producer/consumer program that takes one value from one input queue, another value from another input queue, and puts the sum of those two values into a separate queue.
- Q5-4** For each basic block given below, rewrite it in single-assignment form, and then, draw the data flow graph for that form.
- a.**

```
x = a + b;
y = c + d;
z = x + e;
```
  - b.**

```
r = a + b - c;
s = 2 *r;
t = b - d;
r = d + e;
```
  - c.**

```
a = q - r;
b = a + t;
a = r + s;
c = t - u;
```
  - d.**

```
w = a - b + c;
x = w - d;
y = x - z;
w = a + b - c;
z = y + d;
y = b *c;
```
- Q5-5** Draw the CDFG for the following code fragments:
- a.**

```
if (y == 2) { r = a + b; s = c - d; }
else r = a - c
```
  - b.**

```
x = 1;
if (y == 2) { r = a + b; s = c - d; }
else { r = a - c; }
```
  - c.**

```
x = 2;
while (x < 40) {
    x = foo[x];
}
```
  - d.**

```
for (i = 0; i < N; i++)
    x[i] = a[i]*b[i];
```

```

e. for (i = 0; i < N; i++) {
    if (a[i] == 0)
        x[i] = 5;
    else
        x[i] = a[i]*b[i];
}

```

**Q5-6** Show the contents of the assembler's symbol table at the end of code generation for each line of the following programs:

a. ORG 200  
 p1: ADR r4,a  
 LDR r0,[r4]  
 ADR r4,e  
 LDR r1,[r4]  
 ADD r0,r0,r1  
 CMP r0,r1  
 BNE q1  
 p2: ADR r4,e

b. ORG 100  
 p1: CMP r0,r1  
 BEQ x1  
 p2: CMP r0,r2  
 BEQ x2  
 p3: CMP r0,r3  
 BEQ x3

c. ORG 200  
 S1: ADR r2,a  
 LDR r0,[r2]  
 S2: ADR r2,b  
 LDR r2,a  
 ADD r1,r1,r2

d. ORG 100  
 L1: ADR r1,a  
 LDR r0,[r1]  
 L2: ADR r1,b  
 LDR r1,a  
 L3: CMP r0, r2  
 L4: BEQ L19

**Q5-7** Your linker uses a single pass through the set of given object files to find and resolve external references. Each object file is processed in the order given, all external references are found, and then, the previously loaded files are searched for labels that resolve those references. Will this linker be able to successfully load a program with these external references and entry points?

Object file	Entry points	External references
o1	a, b, c, d	s, t
o2	r, s, t	w, y, d
o3	w, x, y, z	a, c, d

**Q5-8** Determine whether each of these programs is reentrant.

a. 

```
int p1(int a, int b) {
    return(a + b);
}
```

b. 

```
int x, y;
int p2(int a) {
    return a + x;
}
```

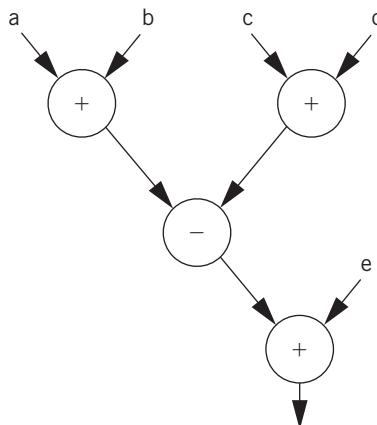
c. 

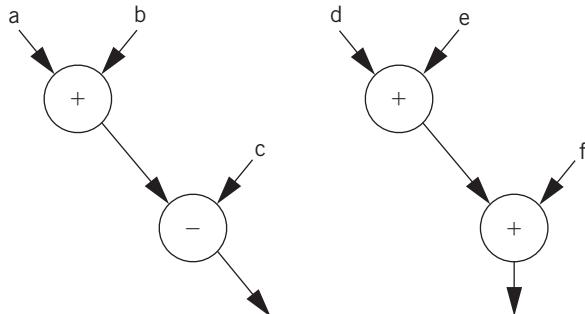
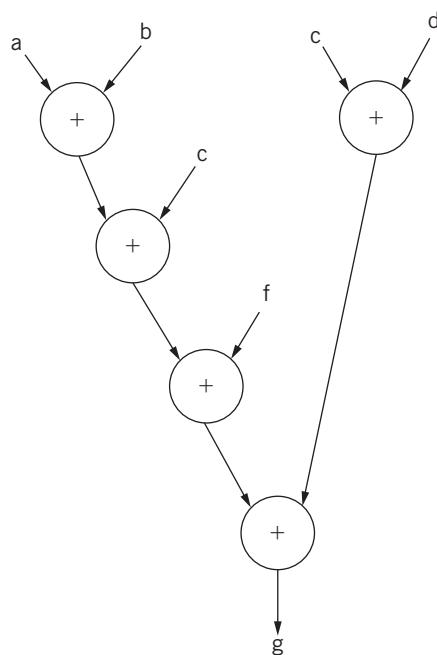
```
int x, y;
int p3(int a, int b) {
    if (a > 0)
        x = b;
    return a + b;
}
```

**Q5-9** Is the code for the FIR filter of Programming Example 5.3 reentrant? Explain.

**Q5-10** Provide the required order of execution of operations in these data flow graphs. If several operations can be performed in arbitrary order, show them as a set:  $\{a + b, c - d\}$ .

a.



**b.****c.**

**Q5-11** Draw the CDFG for the following C code before and after applying dead code elimination to the if statement:

```
#define DEBUG 0
proc1();
if (DEBUG) debug_stuff();
```

```

switch (foo) {
    case A: a_case();
    case B: b_case();
    default: default_case();
}

```

**Q5-12** Unroll the loop below:

- a. two times
- b. three times

```

for (i = 0; i < 32; i++)
    x[i] = a[i] * c[i];

```

**Q5-13** Apply loop fusion or loop distribution to these code fragments as appropriate. Identify the technique you use and write the modified code.

- a. 

```

for (i=0; i<N; i++)
    z[i] = a[i] + b[i];
for (i=0; i<N; i++)
    w[i] = a[i] - b[i];
```
- b. 

```

for (i=0; i<N; i++) {
    x[i] = c[i]*d[i];
    y[i] = x[i] *e[i];
}
```
- c. 

```

for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
        c[i][j] = a[i][j] + b[i][j];
        x[j] = x[j] *c[i][j];
    }
    y[i] = a[i] + x[j];
}
```

**Q5-14** Can you apply code motion to the following example? Explain.

```

for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        z[i][j] = a[i] *b[i][j];

```

**Q5-15** For each of the basic blocks of Q5-4, determine the minimum number of registers required to perform the operations when they are executed in the order shown in the code. You can assume that all computed values are used outside the basic blocks, so that no assignments can be eliminated.

**Q5-16** For each of the basic blocks of Q5-4, determine the order of execution of operations that gives the smallest number of required registers. Next, state the number of registers required in each case. You can assume that all computed values are used outside the basic blocks, so that no assignments can be eliminated.

**Q5-17** Draw a data flow graph for the code fragment of Example 5.6. Assign an order of execution to the nodes in the graph so that no more than four registers are required. Explain how you arrived at your solution using the structure of the data flow graph.

**Q5-18** Determine the longest path through each code fragment, assuming that all statements can be executed in equal time and that all branch directions are equally probable.

```

a. if (i < CONST1) { x = a + b; }
    else { x = c - d; y = e + f; }
b. for (i = 0; i < 32; i++)
    if (a[i] < CONST2)
        x[i] = a[i] * c[i];
c. if (a < CONST3) {
    if (b < CONST4)
        w = r + s;
    else {
        w = r - s;
        x = s + t;
    }
} else {
    if (c > CONST5) {
        w = r + t;
        x = r - s;
        y = s + u;
    }
}

```

**Q5-19** For each code fragment, list the sets of variable values required to execute each assignment statement at least once. Reaching all assignments may require multiple independent executions of the code.

```

a. if (a > 0)
    x = 5;
else {
    if (b < 0)
        x = 7;
}
b. if (a == b) {
    if (c > d)
        x = 1;
else
    x = 2;
x = x + 1;
}

```

```

c. if (a + b > 0) {
        for (i=0; i<a; i++)
            x = x + 1;
    }
d. if (a - b == 5)
        while (a > 2)
            a = a - 1;
    
```

- Q5-20** Determine the shortest path through each code fragment, assuming that all statements can be executed in equal time and that all branch directions are equally probable. The first branch is always taken.

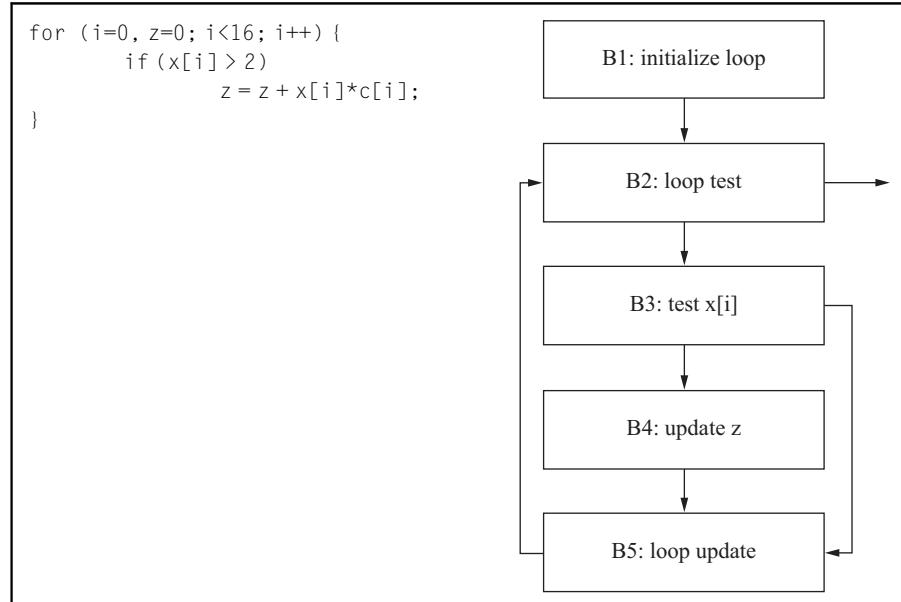
```

a. if (a > 0)
        x = 5;
    else {
        if (b < 0)
            x = 7;
    }
b. if (a == b) {
        if (c > d)
            x = 1;
        else
            x = 2;
        x = x + 1;
    }
    
```

- Q5-21** You are given this program and its flowchart:

```

for (i=0, z=0; i<16; i++) {
    if (x[i] > 2)
        z = z + x[i]*c[i];
}
    
```



The execution time of the blocks is: B1 = 6 cycles, B2 = 2 if branch taken, 5 if not taken, B3 = 3 if branch taken, 6 if not taken, B4 = 7, B5 = 1

- a. What is the maximum number of times that each block in your flowchart executed?
- b. What is the minimum number of times that each block in your flowchart executed?
- c. What is the maximum execution time of the program in clock cycles?  
What is the minimum execution time of the program in clock cycles?

**Q5-22** You are given this program:

```
for (i=0, z=0; i<16; i++) {
    z = z + x[i]*c[i];
}
```

A cache miss costs 6 clock cycles and a cache hit costs 2 clock cycles.

Assume that x and c do not interfere in the cache and that z and i are held in registers. If the cache line can hold W words, plot  $T_a$ , the total number of cycles required for the array accesses (x and c) during all 16 loop iterations for the values  $2 \leq W \leq 8$ .

**Q5-23** Write the branch tests for each conditional.

- a. if ((a > 0) && (b < 0)) f1();
- b. if ((a == 5) && !c) f2();
- c. if ((b || c) && (a != d)) f3();

**Q5-24** The loop appearing below is executed on a machine that has a 1-K-word data cache with four words per cache line.

- a. How must x and a be placed relative to each other in memory to produce a conflict miss every time the inner loop's body is executed?
- b. How must x and a be placed relative to each other in memory to produce a conflict miss one out of every four times the inner loop's body is executed?
- c. How must x and a be placed relative to each other in memory to produce no conflict misses?

```
For (i = 0; i < 50; i++)
    for (j = 0; j < 4; j++)
        x[i][j] = a[i][j] * c[i];
```

**Q5-25** Explain why the person generating clear-box program tests should not be the person who wrote the code being tested.

**Q5-26** Find the cyclomatic complexity of the CDFGs for each of the code fragments given below.

```

a. if (a < b) {
    if (c < d)
        x = 1;
    else
        x = 2;
} else {
    if (e < f)
        x = 3;
    else
        x = 4;
}

b. switch (state) {
    case A:
        if (x = 1) { r = a + b; state = B; }
        else { s = a - b; state = C; }
        break;
    case B:
        s = c + d;
        state = A;
        break;
    case C:
        if (x < 5) { r = a - f; state = D; }
        else if (x == 5) { r = b + d; state = A; }
        else { r = c + e; state = D; }
        break;
    case D:
        r = r + 1;
        state = D;
        break;
}
c. for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        x[i][j] = a[i][j] * c[i];

```

**Q5-27** Use the branch condition testing strategy to determine a set of tests for each of the following statements.

```

a. if (a < b || ptr1 == NULL) proc1();
    else proc2();
b. switch (x) {
    case 0: proc1(); break;
    case 1: proc2(); break;
    case 2: proc3(); break;

```

```

        case 3: proc4(); break;
        default; dproc(); break;
    }
c. if (a < 5 && b > 7) proc1();
    else if (a < 5) proc2();
    else if (b > 7) proc3();
    else proc4();

```

**Q5-28** Find all the def-use pairs for each code fragment given below.

```

a. x = a + b;
    if (x < 20) proc1();
    else {
        y = c + d;
        while (y < 10)
            y = y + e;
    }
b. r = 10;
    s = a - b;
    for (i = 0; i < 10; i++)
        x[i] = a[i] * b[s];
c. x = a - b;
    y = c - d;
    z = e - f;
    if (x < 10) {
        q = y + e;
        z = e + f;
    }
    if (z < y) proc1();

```

**Q5-29** For each of the code fragments of Q5-28, determine values for the variables that will cause each def-use pair to be exercised at least once.

**Q5-30** Assume you want to use random tests on an FIR filter program. How would you know when the program under test is executing correctly?

**Q5-31** What role does a certificate play in signed code?

## Lab exercises

**L5-1** Compare the source code and assembly code for a moderate-size program. Most C compilers will provide an assembly language listing with the -S flag. Can you trace the high-level language statements in the assembly code? Can you see any optimizations that can be done on the assembly code?

- L5-2** Write C code for an FIR filter. Measure the execution time of the filter, either using a simulator or by measuring the time on a running microprocessor. Vary the number of taps in the FIR filter and measure execution time as a function of the filter size.
- L5-3** Write C++ code for an FIR filter using a class to implement the filter. Implement as many member functions as possible as inline functions. Measure the execution time of the filter and compare to the C implementation.
- L5-4** Generate a trace for a program using software techniques. Use the trace to analyze the program's cache behavior.
- L5-5** Use a cycle-accurate CPU simulator to determine the execution time of a program.
- L5-6** Measure the power consumption of your microprocessor on a simple block of code.
- L5-7** Use software testing techniques to determine how well your input sequences to the cycle-accurate simulator exercise your program.
- L5-8** Generate a set of functional tests for a moderate-size program. Evaluate your test coverage in one of two ways: Have someone else independently identify bugs and see how many of those bugs your tests catch (and how many tests they catch that were not found by the human inspector); or inject bugs into the code and see how many of those are caught by your tests.

# Processes and Operating Systems

# 6

## CHAPTER POINTS

---

- Process abstraction.
- Switching contexts between programs.
- Real-time operating systems (RTOSs).
- Interprocess communication.
- Task-level performance analysis and power consumption.
- Design example: engine control unit.

---

## 6.1 Introduction

Although simple applications can be programmed on a microprocessor by writing a single piece of code, many applications are sophisticated enough that writing one large program does not suffice. When multiple operations must be performed at widely varying times, a single program can easily become too complex and unwieldy. The result is spaghetti code that is too difficult to verify for either performance or functionality.

This chapter studies two fundamental abstractions that allow us to build complex applications on microprocessors: the **process** and the **operating system (OS)**. Together, these abstractions allow us to switch the state of the processor between tasks. The process cleanly defines the state of an executing program, whereas the operating system provides the mechanism for switching the execution between processes.

Together these two mechanisms allow us to build applications with more complex functionality and much greater flexibility to satisfy timing requirements. The need to satisfy complex timing requirements—events happening at distinctive rates, intermittent events, and so on—causes us to use processes and operating systems to build embedded software. Satisfying complex timing tasks can introduce extremely complex control into programs. Using processes to compartmentalize functions and encapsulating in the operating system, the control required to switch between processes makes it much easier to satisfy timing requirements with relatively clean control within the processes.

We are particularly interested in the **real-time operating system (RTOS)**, an operating system that provide facilities for satisfying real-time requirements. An RTOS allocates resources using algorithms that take real time into account. General-purpose operating systems, in contrast, generally allocate resources using other criteria, such as fairness. Trying to allocate the CPU equally to all processes without regard to time can easily cause processes to miss deadlines.

In the next section, we introduce the concept of a process. [Section 6.2](#) motivates our need for multiple processes. [Section 6.3](#) looks at how the RTOS implements processes. [Section 6.4](#) develops algorithms for scheduling those processes to meet real-time requirements. [Section 6.6](#) introduces some basic concepts of interprocess communication. [Section 6.7](#) considers the performance of real-time operating systems. [Section 6.8](#) surveys various RTOSs. We study an engine control unit as a design example in [Section 6.9](#).

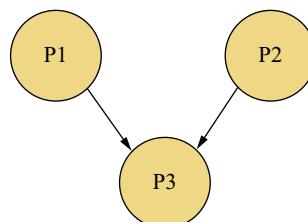
## 6.2 Multiple tasks and processes

We studied the design of the programs in [Chapter 5](#). An RTOS allows us to run several programs concurrently. The RTOS helps build more complex systems using several programs that run concurrently. Processes and tasks are the building blocks of multitasking systems, much as C functions and code modules are the building blocks of programs.

### Tasks

Many (if not most) embedded computing systems do more than one thing—the environment can cause mode changes that in turn cause the embedded system to behave quite differently. For example, when designing a telephone answering machine, we can define recording a phone call and operating the user's control panel as distinct tasks because they perform logically distinct operations that must be performed at distinctive rates.

The term **task** is used in several different ways in real-time computing. We use it to mean a set of real-time programs that can communicate. [Fig. 6.1](#) shows a task that consists of three subtasks; arrows in the **task graph** show data dependencies. P3 cannot start before it receives the results of P1 and P2, and P1 and P2 can be executed in parallel.



**FIGURE 6.1**

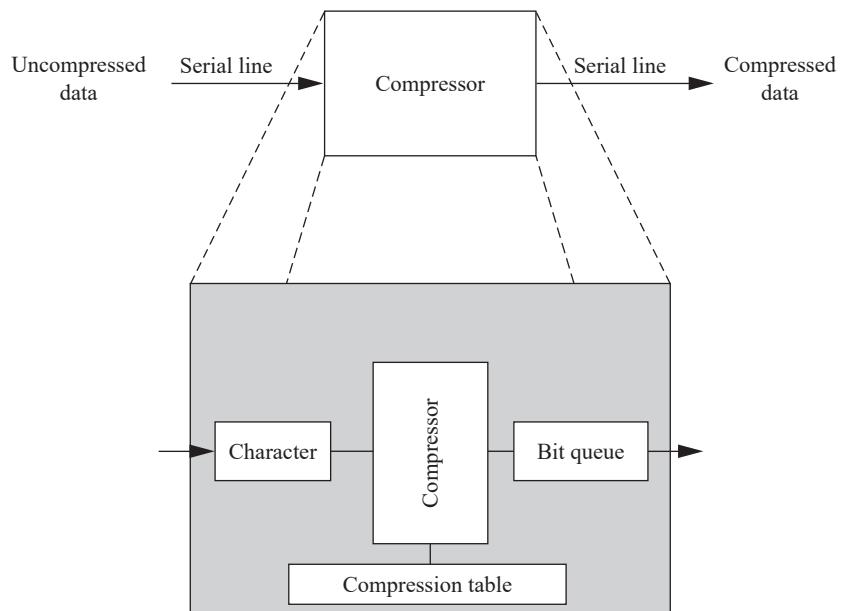
A task made up of three subtasks.

A **process** is a single execution of a program. If we run the same program at two different times, we have created two different processes. Each process has its own state that includes not only its registers but also all its memory. In some operating systems, the memory management unit is used to keep each process in a separate address space. In others, particularly lightweight RTOSs, the processes run in the same address space. Processes that share the same address space are often called **threads**.

To understand why the separation of an application into tasks may be reflected in the program structure, consider how we would build a stand-alone text compression unit based on the compression algorithm implemented in [Section 3.8](#). As shown in [Fig. 6.2](#), this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line based on a predefined compression table. Such a box may be used, for example, to compress the data being sent to a modem.

#### Variable data rates

There is a need to receive and send data at different rates. For example, the program may emit two bits for the first byte and then seven bits for the second byte. This routine will obviously find itself reflected in the structure of the code. It is possible to create irregular, ungainly code to solve this problem, but a more elegant solution is to create a queue of output bits, with those bits being removed from the



**FIGURE 6.2**

The text compression engine as a multirate system.

queue and sent to the serial port in eight-bit sets. However, beyond the need to create a clean data structure that simplifies the control structure of the code, we must also ensure that we process the inputs and outputs at the proper rates. For example, if we spend too much time packaging and emitting output characters, we may drop an input character. Solving timing problems is a challenging problem.

#### Asynchronous input

The text compression box provides a simple example of rate-control problems. A control panel on a machine provides an example of a different type of rate-control problem: the **asynchronous input**. The control panel of the compression box may, for example, include a compression mode button that disables or enables compression so that the input text is passed through unchanged when compression is disabled. We certainly don't know when the user will push the button for compression mode, but the button may be depressed asynchronously relative to the arrival of characters for compression.

We do know, however, that the button will be depressed at a much lower rate than the characters will be received because it is not physically possible for a person to repeatedly depress a button at even slow serial line rates. Keeping up with the input and output data while checking the button can introduce some very complex control code into the program. Sampling the button's state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently can cause the machine to incorrectly compress data. One solution is to introduce a counter into the main compression loop so that a subroutine, to check the input button, is called once every  $n$  times the compression loop is executed. However, this solution doesn't work when either the compression loop or the button-handling routine has highly variable execution times. If the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost. We must be able to keep track of these two tasks separately by applying different timing requirements to each. This is the sort of control that processes allow.

#### Events

An asynchronous input is an example of an **event**. We consider the timing analysis of events in [Section 6.5.8](#).

#### Timing and programming

These two examples illustrate how requirements for timing and execution rate can create major problems in programming. When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution quickly become very complex. Worse, such complex control is usually quite difficult to verify for either functional or timing properties.

---

## 6.3 Multirate systems

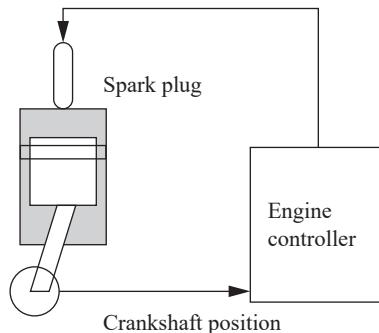
Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled. **Multirate** embedded computing systems, including automobile engines, printers, and cell phones, are very common.

In these systems, certain operations must be executed periodically, and each operation is executed at its own rate.

Application Example 6.1 describes why automobile engines require multirate control.

### Application Example 6.1: Automotive Engine Control

The simplest automotive engine controllers, such as the ignition controller for a basic motorcycle engine, perform only one task, timing the firing of the spark plug, which takes the place of a mechanical distributor. The spark plug must be fired at a certain point in the combustion cycle, but to obtain better performance, the phase relationship between the piston's movement and the spark should change as a function of engine speed. Using a microcontroller that senses the engine crankshaft position allows the spark timing to vary with engine speed. Firing the spark plug is a periodic process, but note that the period depends on the engine's operating speed.



The control algorithm for a modern automobile engine is much more complex, making the need for microprocessors much greater. Automobile engines must meet strict requirements, as mandated by law in the United States, for both emissions and fuel economy. On the other hand, engines must satisfy customers not only in terms of performance but also in terms of ease of starting in extreme cold and heat, low maintenance, and so on.

Automobile engine controllers use additional sensors, including a gas-pedal position and an oxygen sensor, which is used to control emissions. They also used a multimode control scheme. For example, one mode may be used for engine warm-up, another for cruise, another for climbing steep hills, and so forth. A larger number of sensors and modes increases the number of discrete tasks that must be performed. The highest-rate task is firing the spark plugs. The throttle setting must be sampled and acted upon regularly, although not as frequently as the crankshaft setting and spark plugs. The oxygen sensor responds much more slowly than does the throttle, so adjustments to the fuel/air mixture suggested by the oxygen sensor can be computed at a much lower rate.

The engine controller takes a variety of inputs that determine the state of the engine. It then controls two basic engine parameters: spark-plug firings and the fuel/air mixture. Engine control is computed periodically, but the periods of the different inputs and outputs range over several orders of magnitude of time. An early paper on automotive electronics

by Marley [Mar78] described the rates at which engine inputs and outputs must be handled:

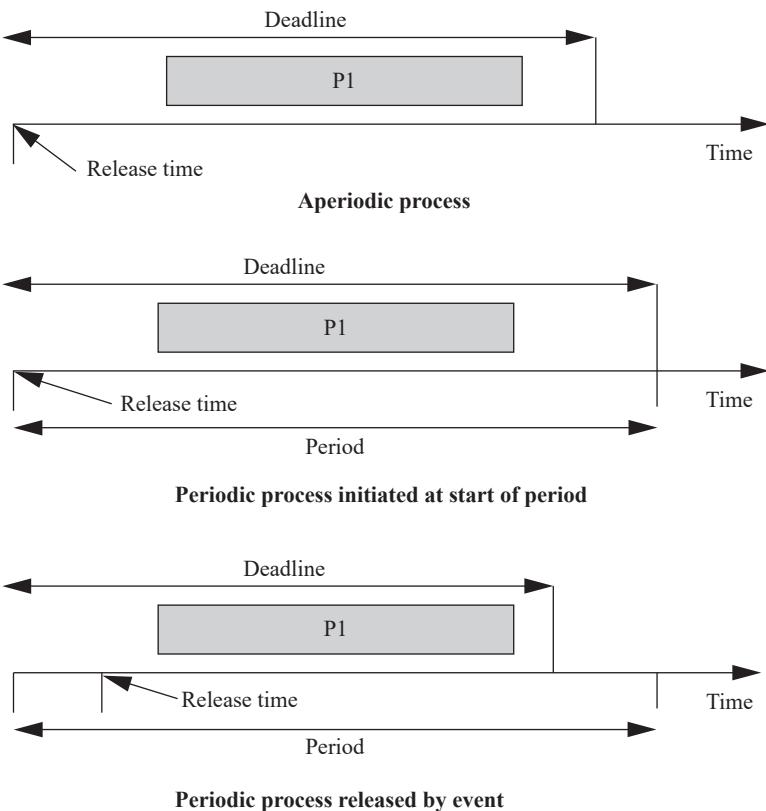
Variable	Time to move full range (ms)	Update period (ms)
Engine spark timing	300	2
Throttle	40	2
Air flow	30	4
Cranking battery voltage	80	4
Fuel flow	250	10
Percentage recycled exhaust gas	500	25
Set of status switches	100	50
Air temperature	Seconds	500
Barometric pressure	Seconds	1,000
Spark/dwell	10	1
Fuel adjustments	80	4
Carburetor adjustments	500	25
Mode actuators	100	100

The fastest rate that the engine controller must handle is 2 ms, and the slowest rate is 4 s, a range of three orders of magnitude.

### 6.3.1 Timing requirements for processes

Processes can have several different types of timing requirements imposed on them by the application. The timing requirements of a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling in more detail, we outline the types of process timing requirements that are useful in embedded system design.

Fig. 6.3 illustrates different ways in which we can define two important requirements for processes: **initiation time** and **deadline**. The initiation time is the time at which the process goes from the waiting state to the ready state. An aperiodic process is, by definition, initiated by an event, such as the arrival of external data or data computed by another process. The initiation time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself. For a periodically executed process, there are two common possibilities. In simpler systems, the process may be ready at the beginning of the period.

**FIGURE 6.3**

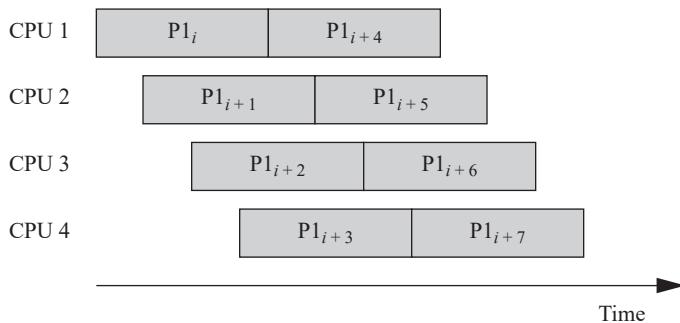
Example definitions of initiation times and deadlines.

More sophisticated systems may set the initiation time at the arrival time of certain data at a time after the start of the period.

A deadline specifies when a computation must be finished. The deadline for an aperiodic process is generally measured from the initiation time because that is the only reasonable time reference. The deadline for a periodic process may, in general, occur at some time other than the end of the period. As we will see in [Section 6.5](#), some scheduling policies simplify the assumption that the deadline occurs at the end of the period.

Rate requirements are also common. A rate requirement specifies how quickly processes must be initiated. The **period** of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between successive input samples. The process's **rate** is the inverse of its period. In a multirate system, each process executes at its own distinct rate.

The most common requirement for periodic processes is for the **initiation interval** to be equal to the period. However, the pipelined execution of processes allows the

**FIGURE 6.4**

A sequence of processes with a high initiation rate.

initiation interval to be less than the period. Fig. 6.4 illustrates the process execution in a system with four CPUs. The various execution instances of program P1 have been subscripted to distinguish their initiation times. In this case, the initiation interval is equal to one-fourth of the period. It is possible for a process to have an initiation rate less than the period, even in single-CPU systems. If the process execution time is significantly less than the period, it may be possible to initiate multiple copies of a program at slightly offset times.

#### Hyperperiod

When we consider a set of processes, we often talk about the **hyperperiod** of the set of processes—the least-common multiple (LCM) of the periods of the processes. We will see later that the hyperperiod tells us the length of the time interval over which we must analyze the schedule.

#### Response time

Although a period is a specification of the expected behavior of a task, we also want to talk about its actual behavior. We define the **response time** of a process as the time at which the process finishes. If the schedule meets its requirements, the response time will be before the end of the process's period. The response time depends only in part on its **computation time**: how long it takes to execute. In a multi-tasking system, the process may be interrupted to allow other processes to run. The response time accounts for all the CPU time allocated to other processes.

#### Jitter

We may also be concerned with the **jitter** of a task, which is the allowable variation in its completion. Jitter can be important to a variety of applications: the playback of multimedia data to avoid audio gaps or jerky images and the control of machines to ensure that the control signal is applied at the right time.

#### Missing a deadline

What happens when a process misses a deadline? The practical effects of a timing violation depend on the application; the results can be catastrophic in an automotive control system, whereas a missed deadline in a telephone system may cause a temporary silence on the line. The system can be designed to take a variety of actions when a deadline is missed. Safety-critical systems may try to take compensatory measures, such as approximating data or switching into a special safety mode. Systems for which safety is not as important may take simple measures to avoid propagating bad data, such as inserting silence in a phone line, or they may completely ignore the failure.

Even if the modules are functionally correct, their timing behavior can introduce major execution errors. Application Example 6.2 describes a timing problem in space shuttle software that caused a delay in the first launch of the shuttle.

---

### Application Example 6.2: A Space Shuttle Software Error

Garman [Gar81] described a software problem that delayed the first launch of the US space shuttle. No one was hurt, and the launch proceeded after the computers were reset. However, this bug was serious and unanticipated.

The shuttle's primary control system is known as the Primary Avionics Software System (PASS). It used four computers to monitor events, with the four machines voting to ensure fault tolerance. Four computers allowed one machine to fail while still leaving three operating machines to vote, such that a majority vote would still be possible to determine operating procedures. If at least two machines failed, control was to be turned over to a fifth computer called the Backup Flight Control System (BFS). The BFS used the same computer, requirements, programming language, and compiler, but it was developed by a different organization than the one that built the PASS to ensure that methodological errors did not cause the simultaneous failure of both systems. The switchover from PASS to BFS was controlled by the astronauts.

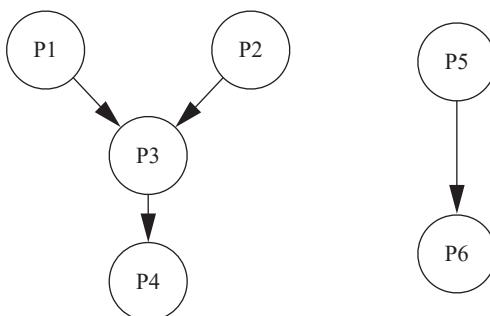
During normal operation, the BFS would listen to the operation of the PASS computers so that they could keep track of the state of the shuttle. However, BFS would stop listening when it thought PASS was compromising data fetching. This would prevent PASS failures from inadvertently destroying the state of the BFS. PASS uses an asynchronous, priority-driven software architecture. If high-priority processes take too much time, the operating system can skip or delay lower-priority processing. The BFS, in contrast, used a time slot system that allocated a fixed amount of time to each process. Because the BFS monitored the PASS, it could be confused by temporary overloads on the primary system. As a result, the PASS was changed late in the design cycle to make its behavior more amenable to the backup system.

On the morning of the launch attempt, the BFS failed to synchronize itself with the primary system. It saw the events in the PASS system as inconsistent and therefore stopped listening to PASS behavior. It turned out that all PASS and BFS processing had been running late relative to telemetry data. This occurred because the system incorrectly calculated its start time.

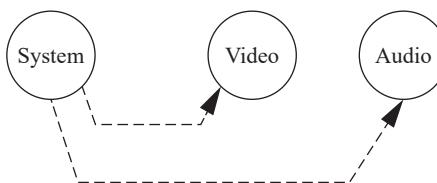
After much analysis of system traces and software, it was determined that a few minor changes to the software had caused the problem. First, about two years before the incident, a subroutine used to initialize the data bus was modified. Because this routine was run prior to calculating the start time, it introduced an additional unnoticed delay into that computation. About a year later, a constant was changed in an attempt to fix that problem. As a result of these changes, there was a 1 in 67 probability for a timing problem. When this occurred, almost all computations on the computers would occur a cycle late, leading to the observed failure. The problems were difficult to detect in testing because they required running through all the initialization code; many tests start with a known configuration to save the time required to run the setup code. The changes to the programs were also not obviously related to the final changes in timing.

---

The timing constraints between processes may be constrained when the processes pass data among each other. Fig. 6.5 shows a set of processes with data dependencies among them. Before a process can become ready, all the processes on which it depends must complete and send their data to it. The data dependencies define a partial ordering on process execution. P1 and P2 can execute in any order (or in interleaved fashion), but both must complete before P3, and P3 must complete before P4. All processes must be finished before the end of the period. The data dependencies must form

**FIGURE 6.5**

Data dependencies among processes.

**FIGURE 6.6**

Communication among processes at different rates.

a directed acyclic graph; a cycle in the data dependencies is difficult to interpret in a periodically executed system.

Communication among processes that run at different rates cannot be represented by data dependencies because there is no one-to-one relationship between data coming out of the source process and going into the destination process. Nevertheless, communication among processes of different rates is very common. Fig. 6.6 illustrates the communication required among the three elements of an MPEG audio/video decoder. Data come into the decoder in the system format, which multiplexes audio and video data. The system decoder process demultiplexes the audio and video data and distributes them to the appropriate processes. Multirate communication is necessarily one way. For example, the system process writes data to the video process, but a separate communication mechanism must be provided for communication from the video process back to the system process.

### 6.3.2 CPU usage metrics

In addition to the application characteristics, we must have a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is **utilization**:

$$U = \frac{\text{CPU time for useful work}}{\text{total available CPU time}} \quad (\text{Eq. 6.1})$$

available CPU time. This ratio ranges between zero and one, with one meaning that all the available CPU time is being used for system purposes. Utilization is often expressed as a percentage, and it is typically calculated over the hyperperiod of the task set.

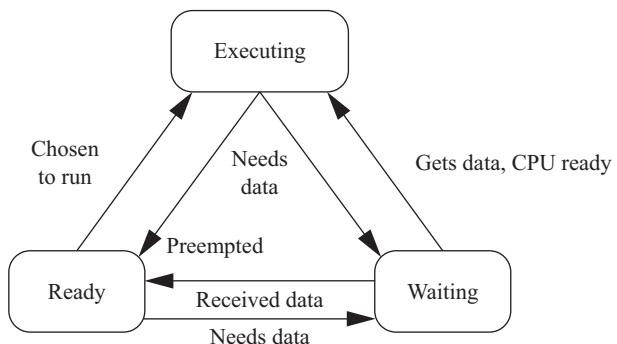
### 6.3.3 Process state and scheduling

The first job of the operating system is to determine the process that runs next. The work of choosing the order of running processes is known as scheduling.

The operating system considers a process to be in one of three basic **scheduling states**: **waiting**, **ready**, or **executing**. At most, there is one process executing on the CPU at any time. If there is no useful work to be done, an idling process may be used to perform null operations. Any process that could be executed is in the ready state; the operating system chooses among the ready processes to select the next executing process. A process may not, however, always be ready to run. For instance, a process may be waiting for data from an I/O device or another process, or it may be set to run from a timer that has not yet expired. Such processes are in a waiting state. Fig. 6.7 shows the possible transitions between the states available to a process. A process goes into the waiting state when it needs data that it has not yet received or when it has finished all work for the current period. A process goes into the ready state when it receives its required data and when it enters a new period. A process can go into the executing state only when it has all its data and is ready to run, and the scheduler selects the process as the next process to run.

A **scheduling policy** defines how processes are selected for promotion from the ready state to the running state. Every multitasking operating system implements some type of scheduling policy. Choosing the right scheduling policy not only ensures that the system will meet all its timing requirements, but it also has a profound influence on the CPU horsepower required to implement the system's functionality.

Scheduling policies vary widely in terms of the generality of the timing requirements they can handle and the efficiency with which they use the CPU. Utilization



**FIGURE 6.7**

Scheduling states of a process.

is one of the key metrics in evaluating a scheduling policy. We will see that some types of timing requirements for a set of processes imply that we cannot utilize 100% of the CPU's execution time on useful work, even ignoring context switching overhead. However, some scheduling policies can deliver higher CPU utilization than others, even for the same timing requirements. The best policy depends on the required timing characteristics of the scheduled processes.

In addition to utilization, we must also consider **scheduling overhead**—the execution time required to choose the next execution process, which is incurred in addition to any context switching overhead. Generally, the more sophisticated the scheduling policy, the more CPU time it takes during system operation to implement it. Moreover, we generally achieve higher theoretical CPU utilization by applying more complex scheduling policies with higher overheads. The final decision regarding a scheduling policy must account for both theoretical utilization and practical scheduling overhead.

### 6.3.4 Running periodic processes

We must find a programming technique that allows us to run periodic processes, ideally at different rates. For the moment, let's think of a process as a subroutine; we call them `p1()`, `p2()`, etc., for simplicity. Our goal is to run these subroutines at rates determined by the system designer.

**First step: while loop**

Here is a very simple program that runs our process subroutines repeatedly:

```
while (TRUE) {
    p1();
    p2();
}
```

This program has several problems. First, it doesn't control the rate at which processes execute; the loop runs as quickly as possible, starting a new iteration as soon as the previous iteration has finished. Second, all processes run at the same rate.

**A timed loop**

Before worrying about multiple rates, let's first make the processes run at a controlled rate. One could imagine controlling the execution rate by carefully designing the code. By determining the execution time of the instructions executed during an iteration, we could pad the loop with useless “no operations” to make the execution time of an iteration equal the desired period. Although some video games were designed this way in the 1970s, this technique should be avoided. Modern processors make it difficult to accurately determine execution time, and even simple processors have nontrivial timing, as we saw in [Chapter 3](#). Conditionals anywhere in the program make it even harder to be sure that the loop consumes the same amount of execution time for every iteration. Furthermore, if any part of the program is changed, the entire timing scheme must be reevaluated.

A timer is a much more reliable way to control the execution of the loop. We would probably use the timer to generate periodic interrupts. Let's assume for the

moment that the `pall()` function is called by the timer's interrupt handler. Then, this code will execute each process once after a timer interrupt:

```
void pall() {
    p1();
    p2();
}
```

What happens when a process runs too long? The timer's interrupt will cause the CPU's interrupt system to mask its interrupts (at least on a reasonable processor), so the interrupt won't occur until after the `pall()` routine returns. As a result, the next iteration will start late. This is a serious problem, but we will have to wait for further refinements before we can fix it.

#### Multiple timers

Our next problem is executing different processes at different rates. If we have several timers, we can set each timer at a different rate. We could then use a function to collect all the processes that run at that rate:

```
void pA() {
    /* processes that run at rate A */
    p1();
    p3();
}
void pB() {
    /* processes that run at rate B */
    p2();
    p4();
    p5();
}
...
...
```

This works, but it requires multiple timers, and we may not have enough timers to support all rates required by a system.

#### Timer plus counters

An alternative is to use counters to divide the counter rate. If, for example, process `p2()` must run at 1/3 the rate of `p1()`, then we can use this code:

```
static int p2count = 0; /* use this to remember count across timer
                        interrupts */
void pall() {
    p1();
    if (p2count >= 2) { /* execute p2() and reset count */
        p2();
        p2count = 0;
    }
    else p2count++; /* just update count in this case */
}
```

This solution allows us to execute processes at rates that are simple multiples of each other. However, when the rates aren't related by a simple ratio, the counting process becomes more complex and likelier to contain bugs.

The next example illustrates an approach to cooperative multitasking in PIC16F.

---

### Programming Example 6.1: Cooperative Multitasking in PIC16F887

We can establish a time base using Timer 0. The period of the timer is set to the period for the execution of all tasks. The flag TOIE enables interrupts for Timer 0. When the timer finishes, it causes an interrupt, and TOIF is set. The interrupt handler for the timer tells us when the timer has ticked using the global variable `timer_flag`:

```
void interrupt timer_handler() {
    if (TOIE && TOIF) { /* timer 0 interrupt */
        timer_flag = 1; /* tell main that the next time period has
                           started */
        TOIF = 0; /* clear timer 0 interrupt flag */
    }
}
```

The main program first initializes the timer and interrupt system, including setting the desired period for the timer. It then uses a while loop to run the tasks at the start of each period:

```
main() {
    init(); /* initialize system, timer, etc. */
    while (1) { /* do forever */
        if (timer_flag) { /* now do the tasks */
            task1();
            task2();
            task3();
            timer_flag = 0; /* reset timer flag */
        }
    }
}
```

---

Why not just put the tasks into the timer handler? We want to ensure that an iteration of the tasks is completed. If the tasks were executed in `timer_handler()` but ran past the period, the timer would interrupt again and stop execution of the previous iteration. The interrupted task may be left in an inconsistent state.

---

## 6.4 Preemptive Real-Time Operating Systems

A preemptive RTOS solves the fundamental problems of a cooperative multitasking system. It executes processes based on timing requirements provided by the system designer. The most reliable way to meet timing requirements accurately is to build a **preemptive** operating system and use **priorities** to control what process runs at any given time. We use these two concepts to build a basic RTOS. We will use FreeRTOS

[Bar07] as our example operating system. This operating system runs on many different platforms.

### 6.4.1 Two basic concepts

To make our operating system work, we must introduce two basic concepts simultaneously. First, we introduce preemption as an alternative to the C-function call to control execution. Second, we introduce priority-based scheduling as a way for the programmer to control the order in which the processes run. We will explain these ideas one at a time as general concepts, and then, go on in the next sections to see how they are implemented in FreeRTOS.org.

#### Preemption

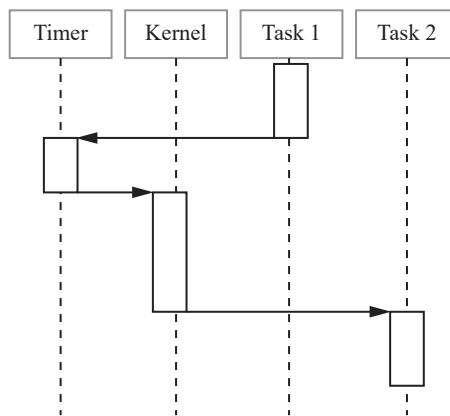
To take full advantage of the timer, we must change our notion of a process. We must, in fact, break the assumptions of our high-level programming language. We will create new routines that allow us to jump from one subroutine to another at any point in the program. This, together with the timer, will allow us to move between functions whenever necessary based upon the system's timing constraints.

[Fig. 6.8](#) shows an example of the preemptive execution of an operating system. We want to share the CPU across the two processes. The **kernel** is the part of the operating system that determines what process is running. The timer periodically activates the kernel. The length of the timer period is known as the **time quantum** because it is the smallest increment in which we can control CPU activity. The kernel determines what process will run next and causes that process to run. On the next timer interrupt, the kernel may pick the same process or another process to run.

Note that this use of the timer is distinctive from our use of the timer in the previous section. Previously, we used the timer to control loop iterations, with one loop iteration including the execution of several complete processes. Here, the time quantum is generally smaller than the execution time of any of the processes.

#### Process priorities

How does the kernel determine what process will run next? We want a mechanism that executes quickly so that we don't spend all our time in the kernel and starve out



**FIGURE 6.8**

Sequence diagram for preemptive execution.

the processes that do useful work. If we assign each task a numerical priority, then the kernel can simply look at the processes and their priorities, see which ones actually want to execute (some may be waiting for data or for some event), and select the highest-priority process that is ready to run. This mechanism is flexible and fast.

### 6.4.2 Processes and context

#### Context switching mechanism

How does the kernel switch between processes before the process is completed? We can't rely on C-level mechanisms to do so; a process is not a C-function or a subroutine. However, we can use assembly language to switch between processes. The timer interrupt causes control to change from the currently executing process to the kernel; assembly language can be used to save and restore registers. We can similarly use assembly language to restore registers not from the process that was interrupted by the timer, and to use registers from any process we want. The set of registers that defines a process is known as its **context**, and switching from one process's register set to another is known as **context switching**. The data structure that holds the state of the process is known as the **record**.

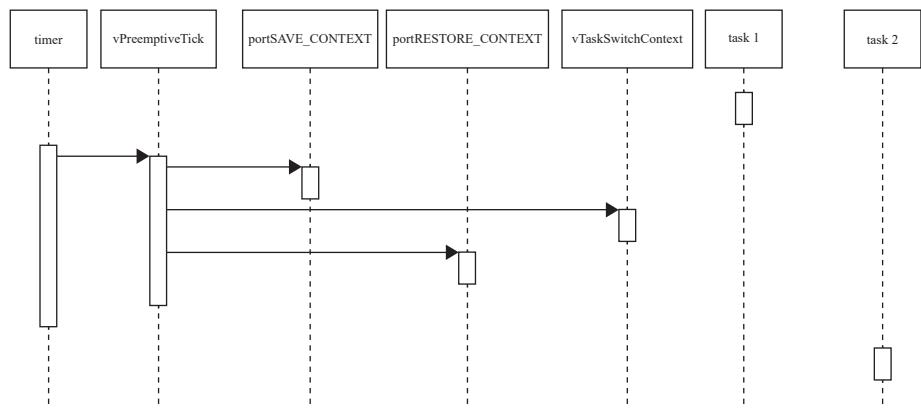
#### RTOS implementation

The best way to understand processes and context is to dive into an RTOS implementation. We use the [FreeRTOS.org](#) kernel as an example; in particular, we use version 7.0.1 for the ARM7 AVR32 platform.

A process is known in [FreeRTOS.org](#) as a *task*.

Let's start with the simplest case, namely, steady state: everything has been initialized, the operating system is running, and we are ready for a timer interrupt. Fig. 6.9 shows a sequence diagram in [FreeRTOS.org](#). This diagram shows the application tasks, the hardware timer, and all the functions in the kernel involved in the context switch:

- `vPreemptiveTick()` is called when the timer ticks.
- `SIG_OUTPUT_COMPARE1A` responds to the timer interrupt and uses `portSAVE_CONTEXT()` to swap out the current task context.



**FIGURE 6.9**

Sequence diagram for a [FreeRTOS.org](#) context switch.

- `vTaskIncrementTick()` updates the time and `vTaskSwitchContext` chooses a new task.
- `portRESTORE_CONTEXT()` swaps in the new context.

Here is the code for `vPreemptiveTick()` in the file `portISR.c`:

```
void vPreemptiveTick( void )
{
    /* Save the context of the current task. */
    portSAVE_CONTEXT();

    /* Increment the tick count - this may wake a task. */
    vTaskIncrementTick();

    /* Find the highest priority task that is ready to run. */
    vTaskSwitchContext();

    /* End the interrupt in the AIC. */
    AT91C_BASE_AIC->AIC_EOICR = AT91C_BASE_PITC->PITC_PIVR;;

    portRESTORE_CONTEXT();
}
```

The first thing that this routine must do is save the context of the task that was interrupted. To do this, it uses the routine `portSAVE_CONTEXT()`. `vTaskIncrementTick()`; then, it performs housekeeping, such as incrementing the tick count. It then determines which task to run next using the routine `vTaskSwitchContext()`. After some more housekeeping, it uses `portRESTORE_CONTEXT()` to restore the context of the task that was selected by `vTaskSwitchContext()`. The action of `portRESTORE_CONTEXT()` causes control to transfer to that task without using the standard C return mechanism.

The code for `portSAVE_CONTEXT()` in the file, `portmacro.h`, is defined as a macro function and not as a C-function. Let's look at the assembly code that is actually executed.

```
push r0
in r0, __SREG__
cli
push r0
push r1
clr r1
push r2
; continue pushing all the registers
push r31
lds r26, pxCurrentTCB
lds r27, pxCurrentTCB + 1
in r0, __SP_L__
st x+, r0
in r0, __SP_H__
st x+, r0
```

The context includes the 32 general-purpose registers, PC, status registers, and stack pointers SPH and SPL. Register `r0` is saved first because it is used to save the status register. Compilers assume that `r1` is set to zero, so the context switch does

so after saving the old value of r1. Most of the routine simply consists of pushing the registers; we have commented out some of those register pushes for clarity. Next, the kernel stores the stack pointer.

Here is the code for `vTaskSwitchContext()`, which is defined in the file `tasks.c` (minus some preprocessor directives that include some optional code):

```
void vTaskSwitchContext( void )
{
    if( uxSchedulerSuspended != ( unsigned portBASE_TYPE ) pdFALSE )
    {
        /* The scheduler is currently suspended - do not allow a
        context switch. */
        xMissedYield = pdTRUE;
    }
    else
    {
        traceTASK_SWITCHED_OUT();

        taskFIRST_CHECK_FOR_STACK_OVERFLOW();
        taskSECOND_CHECK_FOR_STACK_OVERFLOW();

        /* Find the highest priority queue that contains ready tasks. */
        while( listLIST_IS_EMPTY( &( pxReadyTasksLists
            [ uxTopReadyPriority ] ) ) )
        {
            configASSERT( uxTopReadyPriority );
            --uxTopReadyPriority;
        }

        /* listGET_OWNER_OF_NEXT_ENTRY walks through the list, so
        the tasks of the
        same priority get an equal share of the processor time. */
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,
            &( pxReadyTasksLists [ uxTopReadyPriority ] ) );

        traceTASK_SWITCHED_IN();
        vWriteTraceToBuffer();
    }
}
```

This function is relatively straightforward; it walks down the list of tasks to identify the highest-priority task.

As with `portSAVE_CONTEXT()`, the `portRESTORE_CONTEXT()` routine is also defined in `portmacro.h` and is implemented as a macro with embedded assembly language. Here is the underlying assembly code:

```
lds r26, pxCurrentTCB
lds r27, pxCurrentTCB + 1
ld r28, x+
```

```

    out  __SP_L__, r28
    ld   r29, x+
    out  __SP_H__, r29
    pop  r31
; pop the registers
    pop  r1
    pop  r0
    out  __SREG__, r0
    pop  r0

```

This code first loads the address for the new task's stack pointer, then gets the stack pointer register values, and finally restores the general-purpose and status registers.

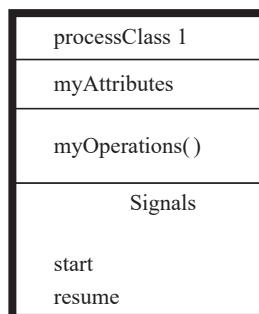
### 6.4.3 Processes and object-oriented design

We need to design systems with processes as components. In this section, we survey the ways in which we can describe processes using UML and how we can use processes as components in object-oriented design.

#### UML active objects

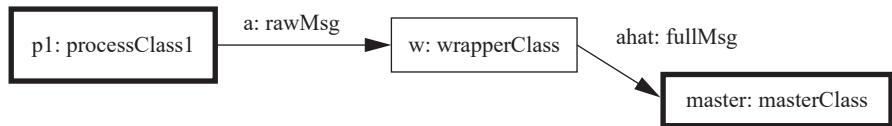
UML often refers to processes as **active objects**, that is, objects that have independent threads of control. The class that defines an active object is known as an **active class**. Fig. 6.10 shows an example of an active UML class. It has all the normal characteristics of a class, including a name, attributes, and operations. It also provides a set of signals that can be used to communicate with the process. A signal is an object that is passed between processes for asynchronous communication. We describe the signals in more detail in Section 6.6.

We can mix active objects and normal objects when describing a system. Fig. 6.11 shows a simple collaboration diagram in which an object is used as an interface between two processes; p1 uses the *w* object to manipulate its data before the data are sent to the *master* process.



**FIGURE 6.10**

An active class in UML.

**FIGURE 6.11**

A collaboration diagram with active and normal objects.

## 6.5 Priority-based scheduling

The operating system's fundamental job is to allocate resources in the computing system to programs that request them. Naturally, the CPU is the scarcest resource, so scheduling the CPU is the operating system's most important job. In this section, we consider the structure of operating systems, how they schedule processes to meet performance requirements, shared resources, and other problems in scheduling, scheduling for low power, and the assumptions underlying our scheduling algorithms.

### Round-robin scheduling

A common scheduling algorithm in general-purpose operating systems is **round-robin**. All processes are kept on a list and scheduled, one after the other. This is generally combined with preemption so that one process does not consume all the CPU time. Round-robin scheduling provides a form of fairness in which all processes get a chance to execute. However, it does not guarantee the completion time of any task; as the number of processes increases, the response time of all processes increases. Real-time systems, in contrast, require their notion of fairness to include timeliness and the satisfaction of deadlines.

### Process priorities

A common way to choose the next executing process in an RTOS is based on process priorities. Each process is assigned a priority, an integer-valued number. The next process to be chosen for execution is the process in the set of ready processes that has the highest-valued priority. Example 6.1 shows how priorities can be used to schedule processes.

### Example 6.1: Priority-driven Scheduling

For this example, we adopt the following simple rules:

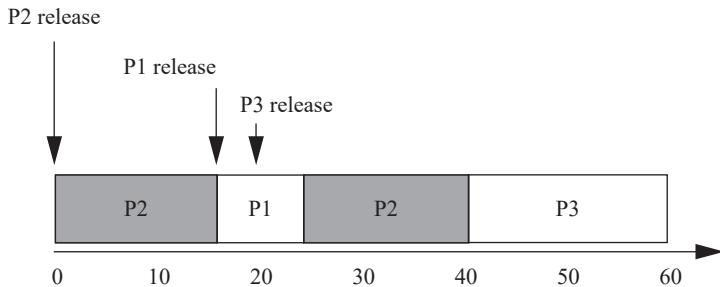
- Each process has a fixed priority that does not vary during the course of execution. More sophisticated scheduling schemes do, in fact, change the priorities of processes to control what happens next.
- The ready process with the highest priority (with one as the highest priority of all) is selected for execution.
- A process continues until it is completed or is preempted by a higher-priority process.

Let's define a simple system with three processes, as seen below.

Process	Priority	Execution time
P1	1	10
P2	2	30
P3	3	20

In addition to describing the properties of the processes in general, we need to know the environmental setup. We assume that P2 is ready to run when the system is started, P1's data arrive at time 15, and P3's data arrive at time 18.

Once we know the process properties and the environment, we can use the priorities to determine which process is running throughout the complete execution of the system.



When the system begins execution, P2 is the only ready process, so it is selected for execution. At time 15, P1 becomes ready; it preempts P2 and begins execution because it has a higher priority. Because P1 is the highest-priority process in the system, it is guaranteed to execute until it finishes. P3's data arrive at time 18, but it cannot preempt P1. Even when P1 finishes, P3 is not allowed to run. P2 is still ready and has a higher priority than P3. Only after both P1 and P2 finish can P3 execute.

### 6.5.1 Rate-monotonic scheduling

**Rate-monotonic scheduling (RMS)**, introduced by Liu and Layland [Liu73], was one of the first scheduling policies developed for real-time systems and is still very widely used. We say that RMS is a **static scheduling policy** because it assigns fixed priorities to processes. It turns out that these fixed priorities are sufficient to efficiently schedule processes in many situations.

The theory underlying RMS is known as **rate-monotonic analysis (RMA)**. This theory, as summarized below, uses a relatively simple model of the system.

- All processes run periodically on a single CPU.
- Context switching time is ignored.
- There are no data dependencies between processes.
- The execution time for a process is constant.
- All deadlines are at the end of their periods.
- The highest-priority ready process is always selected for execution.

The major result of RMA is that a relatively simple scheduling policy is optimal. Priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority. This fixed-priority scheduling policy is the optimum assignment of static priorities to processes, in that it provides the highest CPU utilization while ensuring that all processes meet their deadlines. Example 6.2 illustrates RMS.

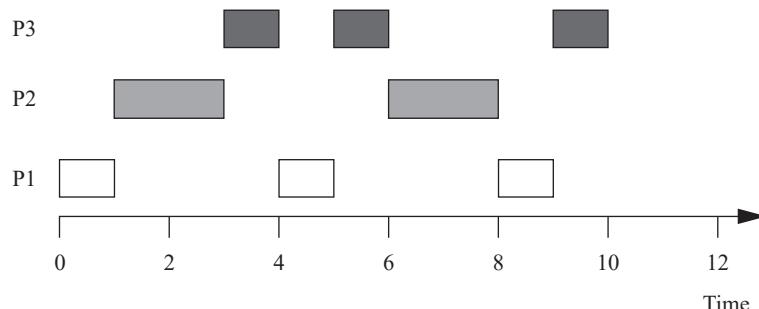
---

**Example 6.2: Rate-monotonic Scheduling**

Here is a simple set of processes and their characteristics.

Process	Execution time	Period
P1	1	4
P2	2	6
P3	3	12

Applying the principles of RMA, we give P1 the highest priority, P2 the middle priority, and P3 the lowest priority. To understand all the interactions between the periods, we must construct a timeline equal in length to the least-common multiple of the process periods (12 in this case). The complete schedule for the least-common multiple of the periods is called the **unrolled schedule**.



All three periods start at time zero. P1's data arrive first. Because P1 is the highest-priority process, it can start to execute immediately. After one time unit, P1 finishes and goes out of the ready state until the start of the next period. At time 1, P2 starts executing as the highest-priority ready process. At time 3, P2 finishes and P3 starts executing. P1's next iteration starts at time 4, at which point it interrupts P3. P3 gets one more time unit of execution between the second iterations of P1 and P2, but P3 doesn't get to finish until after the third iteration of P1.

Consider this different set of execution times for these processes, keeping the same deadlines.

Process	Execution time	Period
P1	2	4
P2	3	6
P3	3	12

In this case, we can show that there is no feasible assignment of priorities that guarantees scheduling. Even though each process alone has an execution time significantly less than its period, combinations of processes can require more than 100% of the available CPU cycles.

For example, during one 12-time-unit interval, we must execute P1 three times, requiring six units of CPU time; P2 two times, costing six units of CPU time; and P3 one time, requiring three units of CPU time. The total of  $6 + 6 + 3 = 15$  units of CPU time is more than the 12 time units available, clearly exceeding the available CPU capacity.

Liu and Layland [Liu73] proved that the RMA priority assignment is optimal using critical-instant analysis. The **critical instant** for a process is defined as the instant during execution at which the task has the largest response time; the **critical interval** is the complete interval for which the task has the largest response time. It is easy to prove that the critical instant for any process  $P$ , under the RMA model, occurs when it is ready and all higher-priority processes are also ready; if we change any higher-priority process to waiting, then  $P$ 's response time can only go down.

We can use critical-instant analysis to determine whether there is a feasible schedule for the system. In the case of the second set of execution times, there was no feasible schedule. Critical-instant analysis also implies that priorities should be assigned in order of periods. Let the periods and computation times of two processes, P1 and P2, be  $\tau_1, \tau_2$  and  $T_1, T_2$ , respectively, with  $\tau_1 < \tau_2$ . We can generalize the result of Example 6.2 to show the total CPU requirements for the two processes in two cases. In the first case, let P1 have higher priority. In the worst case, we then execute P2 once during its period and as many iterations of P1 that can fit in the same interval. Because there are  $\lfloor \tau_2/\tau_1 \rfloor$  iterations of P1 during a single period of P2, the required constraint on CPU time, ignoring context switching overhead, is

$$\left\lfloor \frac{\tau_2}{\tau_1} \right\rfloor T_1 + T_2 \leq \tau_2 \quad (\text{Eq. 6.2})$$

If, on the other hand, we give higher priority to P2, then critical-instant analysis tells us that we must execute all of P2 and all of P1 in one of P1's periods in the worst case:

$$T_1 + T_2 \leq \tau_1 \quad (\text{Eq. 6.3})$$

There are cases in which the first relationship can be satisfied, and the second cannot, but there are no cases in which the second relationship can be satisfied and the first cannot. We can inductively show that a process with a shorter period should always be given higher priority for process sets of arbitrary size. It is also possible to prove that RMS always provides a feasible schedule if such a schedule exists.

The bad news is that, although RMS is the optimal static-priority schedule, it does not allow the system to use 100% of the available CPU cycles. The total CPU **utilization** for a set of  $n$  tasks is

$$U = \sum_{i=1}^n \frac{T_i}{\tau_1} \quad (\text{Eq. 6.4})$$

It is possible to show that, for a set of two tasks under RMS scheduling, the CPU utilization,  $U$ , has a least upper bound of  $2(2^{1/2} - 1) \cong 0.83$ . In other words, the CPU will be idle at least 17% of the time. This idle time is because priorities are assigned

statically; we see in the next section that more aggressive scheduling policies can improve the CPU utilization. When there are  $m$  tasks and the ratio between any two periods is less than two, the maximum processor utilization is

$$U = m(2^{1/m} - 1) \quad (\text{Eq. 6.5})$$

As  $m$  approaches infinity, the CPU utilization (with the factor-of-two restriction on the relationship between periods) asymptotically approaches  $\ln 2 = 0.69$ ; the CPU will be idle 31% of the time. We can use processor utilization  $U$  as an easy measure of the feasibility of an RMS scheduling problem.

Consider an example of an RMS schedule for a system in which P1 has a period of 4 and an execution time of 2 and P2 has a period of 7 and an execution time of 1; these tasks satisfy the factor-of-two restriction on relative periods. The hyperperiod of the processes is 28, so the CPU utilization of this set of processes is  $[(2 \times 7) + (1 \times 4)]/28 = 0.64$ , which is less than our bound of  $\ln 2$ .

### Implementation

The implementation of RMS is easy. Fig. 6.12 shows the C code for an RMS scheduler run at the operating system's timer interrupt. The code merely scans through the list of processes in priority order and selects the highest-priority ready process to run. Because the priorities are static, the processes can be sorted by priority before the system starts executing. As a result, this scheduler has an asymptotic complexity of  $O(n)$ , where  $n$  is the number of processes in the system. This code assumes that processes are not created dynamically. If dynamic process creation is required, the array can be replaced by a linked list of processes, but the asymptotic complexity remains the same. The RMS scheduler has both low asymptotic

```

/* processes[] is an array of process activation records,
   stored in order of priority, with processes[0] being
   the highest-priority process */
Activation_record processes[NPROCESSES];

void RMA(int current) { /* current = currently executing
   process */
    int i;
    /* turn off current process (may be turned back on) */
    processes[current].state = READY_STATE;
    /* find process to start executing */
    for (i = 0; i < NPROCESSES; i++)
        if (processes[i].state == READY_STATE) {
            /* make this the running process */
            processes[i].state == EXECUTING_STATE;
            break;
        }
}

```

**FIGURE 6.12**

C code for rate-monotonic scheduling.

complexity and low actual execution time, which help minimize the discrepancies between the zero-context switch assumption of RMS and the actual execution of an RMS system.

### 6.5.2 Earliest-deadline-first scheduling

**Earliest-deadline-first (EDF)** is another well-known scheduling policy that was also studied by Liu and Layland [Liu73]. It is a dynamic priority scheme; it changes process priorities during execution based on initiation times. As a result, it can achieve higher CPU utilization than RMS.

The EDF policy is also easy, but in contrast to RMS, EDF updates the priorities of processes at every time quantum. In EDF, priorities are assigned in order of deadline: the highest-priority process is the one whose deadline is nearest in time, and the lowest-priority process is the one whose deadline is farthest away. Once EDF has recalculated priorities, the scheduling procedure is the same as that for RMS; the highest-priority ready process is chosen for execution.

Example 6.3 illustrates EDF scheduling in practice.

#### Example 6.3: Earliest-deadline-first Scheduling

Consider the following processes:

Process	Execution time	Period
P1	1	3
P2	1	4
P3	2	5

The least-common multiple of the periods is 60, and the utilization is  $1/3 + 1/4 + 1/5 = 0.9833333$ . This utilization is too high for RMS, but it can be handled with the EDF schedule. Here is the schedule:

Time	Running process	Deadlines
0	P1	
1	P2	
2	P3	P1
3	P3	P2
4	P1	P3
5	P2	P1
6	P1	
7	P3	P2

*Continued*

—Continued

Time	Running process	Deadlines
8	P3	P1
9	P1	P3
10	P2	
11	P3	P1, P2
12	P3	
13	P1	
14	P2	P1, P3
15	P1	P2
16	P2	
17	P3	P1
18	P3	
19	P1	P2, P3
20	P2	P1
21	P1	
22	P3	
23	P3	P1, P2
24	P1	P3
25	P2	
26	P3	P1
27	P3	P2
28	P1	
29	P2	P1, P3
30	P1	
31	P3	P2
32	P3	P1
33	P1	
34	P2	P3
35	P3	P1, P2
36	P1	
37	P2	
38	P3	P1
39	P1	P2, P3

—Continued

Time	Running process	Deadlines
40	P2	
41	P3	P1
42	P1	
43	P3	P2
44	P3	P1, P3
45	P1	
46	P2	
47	P3	P1, P2
48	P3	
49	P1	P3
50	P2	P1
51	P1	P2
52	P3	
53	P3	P1
54	P2	P3
55	P1	P2
56	P2	P1
57	P3	
58	P3	
59	Idle	P1, P2, P3

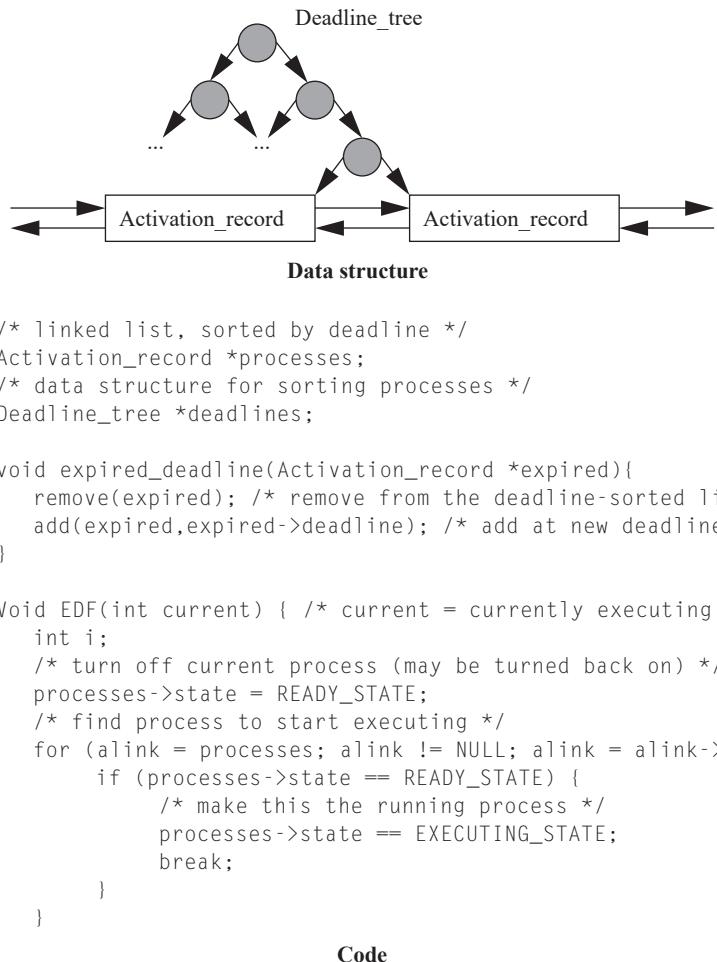
There is one time slot left at the end of this unrolled schedule, which is consistent with our earlier calculation that the CPU utilization is 59/60.

---

Liu and Layland showed that EDF can achieve 100% utilization. A feasible schedule exists if the CPU utilization (calculated in the same way as that for RMA) is less than or equal to one.

### Implementation

The implementation of EDF is more complex than RMS code. Fig. 6.13 outlines one way to implement EDF. The major problem is keeping the processes sorted by time to deadline. Because the times to deadlines for the processes change during execution, we cannot presort the processes into an array, as we could for RMS. To avoid re-sorting the entire set of records at every change, we can build a binary tree to keep the sorted records and incrementally update the sort. At the end of each period, we can move the record to its new place in the sorted list by deleting it from the tree, and then, adding it back to the tree using standard tree manipulation

**FIGURE 6.13**

C code for earliest-deadline-first scheduling.

techniques. We must update process priorities by traversing them in sorted order, so the incremental sorting routines must also update the linked list pointers that let us traverse the records in deadline order. The linked list lets us avoid traversing the tree to go from one node to another, which would require more time. After putting in the effort to build the sorted list of records, selecting the next executing process is done in a manner like that of RMS. However, dynamic sorting adds complexity to the entire scheduling process. Each update of the sorted list requires  $O(n \log n)$  steps. The EDF code is also significantly more complex than the RMS code.

### 6.5.3 RMS vs. EDF

EDF can schedule some task sets that RMS cannot, as shown in the following example.

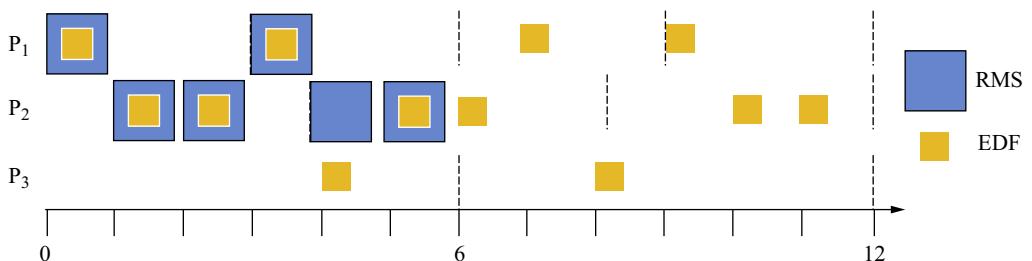
---

#### Example 6.4 RMS vs. EDF Scheduling

Consider this example:

	C	T
P1	1	3
P2	2	4
P3	1	6

The hyperperiod of this task set is 12. Here is an attempt to construct RMS and EDF schedules for these tasks:



EDF successfully schedules tasks with a utilization of 100%. In contrast, RMS misses P3's deadline at Time 6.

---

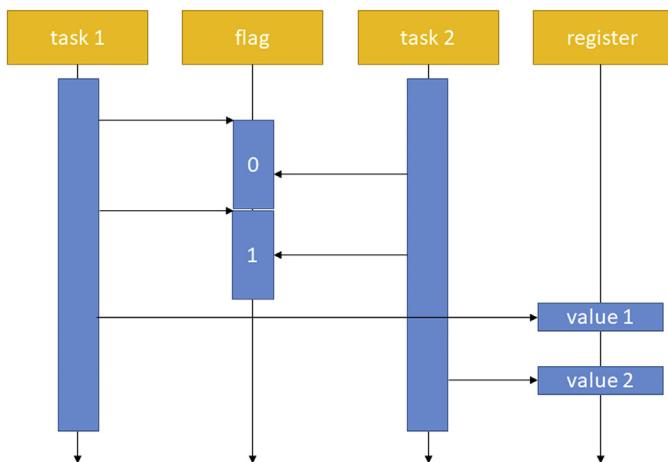
### 6.5.4 Shared resources, mutexes, and semaphores

A process may need to do more than read and write values to and from memory. For example, it may need to communicate with an I/O device. It may also use shared memory locations to communicate with other processes. When dealing with **shared resources**, special care must be taken.

#### Race condition

Consider the case in which an I/O device has a flag that must be tested and modified by a process. Problems can arise when other processes also want to access the device. If combinations of events from the two tasks operate on the device in the wrong order, we may create a **critical timing race** or **race condition** that causes erroneous operation. Consider the situation illustrated in Fig. 6.14:

1. Task 1 reads the flag location and sees that it is 0.
2. Task 2 reads the flag location and sees that it is 0.

**FIGURE 6.14**

A race condition.

3. Task 1 sets the flag location to 1 and writes the data to the I/O device’s data register.
4. Task 2 also sets the flag to one and writes its own data to the device data register, overwriting the data from Task 1.

In this case, both devices thought they were able to write to the device, but Task 1’s write was never completed because it was overridden by Task 2.

#### Critical sections

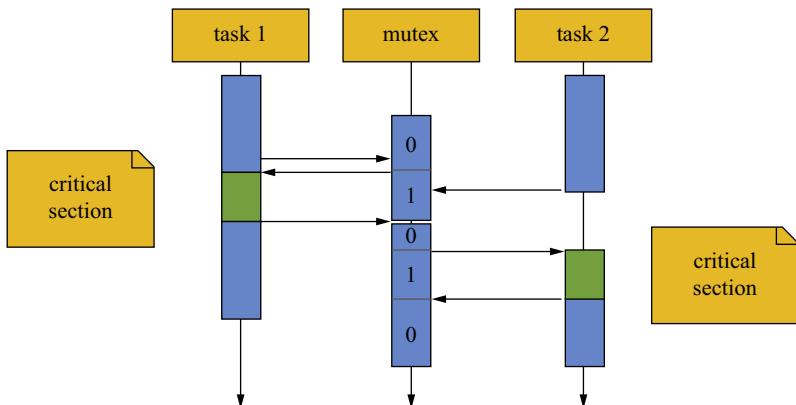
To prevent this type of problem, we need to control the order in which some operations occur. For example, we need to be sure that a task finishes an I/O operation before allowing another task to start its own operation on that I/O device. We do so by enclosing sensitive sections of code in a critical section that executes without interruption.

#### Mutex

We use a **mutex** (for mutual exclusion) to protect a critical section, as illustrated in Fig. 6.15. The mutex is called before each task enters its critical section. In this example, Task 1 calls the mutex first and receives it; the mutex changes state to record that it has been reserved. By the time Task 2 asks for the mutex, Task 1 has already reserved it, so Task 2 waits until Task 1 releases the mutex. At that point, Task 2 proceeds with its critical section, and then, releases the mutex when it is done. Many operating systems allow several mutexes with different names or identifiers to be created to allow multiple critical sections to be managed.

#### Semaphores

A **semaphore** is useful when several copies of a resource are available. The term “semaphore” was derived from railroads, in which a shared section of the track is guarded by signal flags that use semaphores to signal when it is safe to enter the track.

**FIGURE 6.15**

Example application of a mutex to protect critical sections.

A semaphore has two properties: a name allows us to create more than one semaphore in the system, and a count keeps track of the number of units of that type of resource currently in use. The semaphore names are, by tradition,  $P()$  and  $V()$ . The  $P()$  operation is used to gain access. If a unit of the resource is available, that resource's count is incremented. If the count equals the maximum number of units available,  $P()$  waits until one is released using the  $V()$  operator, which decrements the count.

#### Test-and-set

To implement mutexes and semaphores, the microprocessor bus must support an **atomic read/write** operation, which is available on some microprocessors. These types of instructions first read a location, and then, set it to a specified value, returning the results of the test. If the location was already set, then the additional set has no effect, but the instruction returns a false result. If the location was not set, the instruction returns true, and the location is in fact set. The bus supports this as an atomic operation that cannot be interrupted.

Programming Example 6.2 describes the advanced reduced instruction set architecture machine (ARM) atomic read/write operation in more detail.

---

### Programming Example 6.2 Compare-and-swap operations

The SWP (swap) instruction is used in the ARM to implement atomic compare-and-swap:

SWP Rd, Rm, Rn

The SWP instruction takes three operands. The memory location pointed to by  $Rn$  is loaded and saved into  $Rd$ , and the value of  $Rm$  is then written into the location pointed to by  $Rn$ . When  $Rd$  and  $Rn$  are the same register, the instruction swaps the register's value and the value stored at the address pointed to by  $Rd/Rn$ .

---

```

        ADR r0, SEMAPHORE      ; get semaphore address
        LDR r1, #1
GETFLAG    SWP r1,r1,[r0]      ; test-and-set the flag
        BNZ GETFLAG          ; no flag yet, try again
HASFLAG    ...

```

For example, the code sequence first loads the constant, 1, into r1 and the address of the semaphore FLAG1 into register r2. It then reads the semaphore into r0 and writes the 1 value into the semaphore. The code then tests whether the semaphore fetched from memory is zero; if it was, the semaphore was not busy, and we can enter the critical region that begins with the HASFLAG label. If the flag is nonzero, we loop back to try to get the flag once again.

---

The test-and-set allows us to implement semaphores. The P() operation uses a test-and-set to repeatedly test a location that holds a lock on the memory block. The P() operation does not exit until the lock is available; once it is available, the test-and-set automatically sets the lock. Once past the P() operation is passed, the process can work on the protected memory block. The V() operation resets the lock, allowing other processes to access the region by using the P() function.

#### Critical sections and timing

Critical sections pose some problems for real-time systems. Because the interrupt system is shut off during the critical section, the timer cannot interrupt, and other processes cannot start to execute. The kernel may also have its own critical sections that keep interrupts from being serviced and other processes from executing.

### 6.5.5 Priority inversion

Shared resources cause a new and subtle scheduling problem: a low-priority process blocks the execution of a higher-priority process by keeping hold of its resource, a phenomenon known as **priority inversion**. Example 6.5 illustrates the problem.

---

#### Example 6.5 Priority Inversion

A system with three processes: P1 has the highest priority, P3 has the lowest priority, and P2 has a priority between those of P1 and P3. P1 and P3 both use the same shared resource. Processes become ready in this order:

- P3 becomes ready and enters its critical region, reserving the shared resource.
- P2 becomes ready and preempts P3.
- P1 becomes ready. It will preempt P2 and start to run, but only until it reaches its critical section for the shared resource. At that point, it will stop executing.

For P1 to continue, P2 must finish, allowing P3 to resume and finish its critical section. Only when P3 is finished with its critical section can P1 resume.

---

#### Priority inheritance

The most common method for dealing with priority inversion is **priority inheritance**, which promotes the priority of any process when it requests a resource from the operating system. The priority of the process temporarily becomes higher than

that of any other process that may use the resource. This ensures that the process will continue executing once it has the resource so that it can finish its work with the resource, return it to the operating system, and allow other processes to use it. Once the process is finished with the resource, its priority is demoted to its normal value.

### 6.5.6 Scheduling for low power

We can adapt RMS and EDF to consider power consumption. Although the race-to-dark case is more challenging, we have well-understood methods for using **dynamic voltage and frequency scaling (DVFS)** in conjunction with priority-based real-time scheduling [Qua07].

Using DVFS with EDF is relatively straightforward. The critical interval determines the worst case that must be handled. We first set the clock speed to meet the performance requirements in the critical interval. We then select the second-most critical interval and set the clock speed, continuing until the entire hyperperiod has been covered.

Using DVFS with RMS is, unfortunately, NP-complete. However, heuristics can be used to compute a good schedule that meets deadlines and reduces power consumption.

### 6.5.7 A closer look at our modeling assumptions

Our analyses of RMS and EDF made some strong assumptions. These assumptions have made the analyses much more tractable, but the predictions of the analysis may not hold up in practice. Because a misprediction may cause a system to miss a critical deadline, it is important to understand the consequences of these assumptions.

RMS assumes that there are no data dependencies between processes. Example 6.6 shows that knowledge of data dependencies can help to use the CPU more efficiently.

---

#### Example 6.6: Data Dependencies and Scheduling

Data dependencies imply that certain combinations of processes can never occur. Consider this simple example [Mal96]:

**Task graph**

Task	Deadline
1	10
2	8

**Task rates**

Process	CPU time
P1	2
P2	1
P3	4

**Execution times**

We know that P1 and P2 cannot execute at the same time because P1 must finish before P2 can begin. Furthermore, we know that, because P3 has a higher priority, it will not preempt both P1 and P2 in a single iteration. If P3 preempts P1, then P3 will complete before P2 begins; if P3 preempts P2, then it will not interfere with P1 in that iteration. Because we know that some combinations of processes cannot be ready at the same time, we know that our worst-case CPU requirements are less than would be required if all processes could be ready simultaneously.

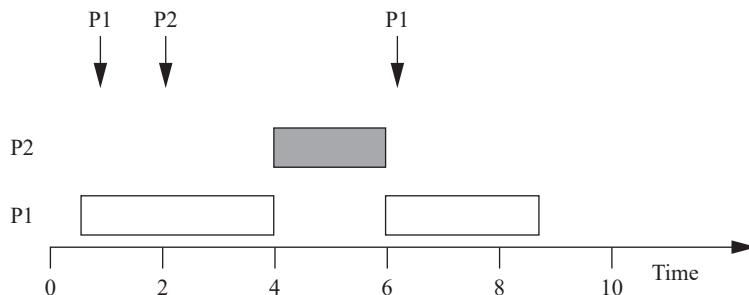
One important simplification we have made is that contexts can be switched in zero time. On the one hand, this is clearly wrong. We must execute instructions to save and restore context, and we must execute additional instructions to implement the scheduling policy. On the other hand, context switching can be implemented efficiently; context switching need not kill performance. The effects of nonzero context switching time must be carefully analyzed in the context of a particular implementation to be sure that the predictions of an ideal scheduling policy are sufficiently accurate. Example 6.7 shows that context switching can, in fact, cause a system to miss a deadline.

### Example 6.7: Scheduling and Context Switching Overhead

Appearing below is a set of processes and their characteristics.

Process	Execution time	Deadline
P1	3	5
P2	3	10

First, let us try to find a schedule assuming that context switching time is zero. The following is a feasible schedule for a sequence of data arrivals that meets all the deadlines:



Now, let us assume that the total time to initiate a process, including context switching and scheduling policy evaluation, is one time unit. It is easy to see that there is no feasible schedule for the above data arrival sequence because we require a total of  $2TP_1 + TP_2 = 2 \times (1 + 3) + (1 + 3) = 12$  time units to execute one period of P2 and two periods of P1.

In most real-time operating systems, a context switch requires only a few hundred instructions, with only slightly more overhead for a simple real-time scheduler, like RMS. These small overhead times are not likely to cause serious scheduling problems. Problems are most likely to manifest themselves in the highest-rate processes, which are often the most critical in any case. Completely checking that all deadlines will be met with nonzero context switching time requires checking all possible schedules for processes and including the context switch time at each preemption or process initiation. However, assuming an average number of context switches per process and computing CPU utilization can provide at least an estimate of how close the system is to CPU capacity.

Rhodes and Wolf [Rho97] developed a computer-assisted design algorithm for implementing processes that compute exact schedules to accurately predict the effects of context switching. Their algorithm selects interrupt-driven and polled implementations for processes on a microprocessor. Polled processes introduce less overhead, but they do not respond to events as quickly as interrupt-driven processes. Furthermore, because adding interrupt levels to a microprocessor usually incurs some cost in added logic, we do not want to use interrupt-driven processes when they are unnecessary. Their algorithm computes precise schedules for the processes, including the overhead for polling or interrupts, as appropriate, and then, uses heuristics to select implementations for the processes. The major heuristic starts with all processes implemented as polled mode, and then, changes processes that miss deadlines to use interrupts. Some iterative improvement steps try various combinations of interrupt-driven processes to eliminate deadline violations. These heuristics minimize the number of processes implemented with interrupts.

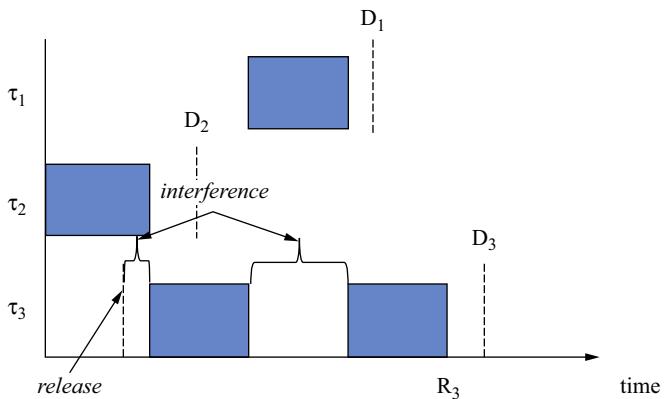
Another important assumption we have made is that the process execution time is constant. As seen in [Section 5.6](#), this is not the case; both data-dependent behavior and caching effects can cause large variations in run times. The techniques for bounding the cache-based performance of a single program do not work when multiple programs are in the same cache. The state of the cache depends on the product of the states of all programs executing in the cache, making the state space of the multiple-process system exponentially larger than that of a single program. We discuss this problem in more detail in [Section 6.7](#).

### 6.5.8 Events and sporadic tasks

An event, such as a nonperiodic input, requires processing by a **sporadic task**. That task will, in general, be active, along with a mix of periodic and other sporadic tasks. The first question to ask about the scheduling of a sporadic task is its the response time [Aud93].

An example is shown in [Fig. 6.16](#). The sporadic task in this example is  $\tau_3$ , which in this case has the lower priority than either  $\tau_1$  or  $\tau_2$ . The deadlines of the three tasks are  $D_1$ ,  $D_2$ ,  $D_3$ .  $\tau_3$  is released while  $\tau_2$  is running, causing interference. After  $\tau_3$  starts to execute,  $\tau_1$  interferes with its execution. The total response time of  $\tau_3$  is

$$R_3 = C_3 + B_3 + I_3 \quad (\text{Eq. 6.6})$$

**FIGURE 6.16**

Response time of a sporadic task.

where  $R_3$  is the response time of  $\tau_3$ ,  $C_3$  is its computation time,  $B_3$  is the time that  $\tau_3$  is blocked by tasks owing to priority inheritance, and  $I_3$  is its total interference time. While  $C_3$  is known,  $B_3$  and  $I_3$  depend on the schedule of the higher-priority tasks. We can analyze the components of the response time given an interval during which the event can occur and the corresponding deadline for finishing the computation associated with that deadline. This analysis does not provide a worst case for any possible event timing.

As we saw with RMA, the calculation of interference time for a task  $\tau_i$  depends on how many times it executes during the interval of interest. If we let  $hp(i)$  represent the tasks with higher priority than  $\tau_i$ , then

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j \quad (\text{Eq. 6.7})$$

where  $T_j$  is the execution time of  $\tau_j$ . This gives a response time of

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j \quad (\text{Eq. 6.8})$$

The blocking time,  $B_i$ , can be determined from the length of the longest critical section of a lower-priority task that may run under priority inheritance during the interval. The response time formula can be extended to consider jitter in the task's release time.

## 6.6 Interprocess communication mechanisms

Processes often need to communicate with each other. **Interprocess communication** mechanisms are provided by the operating system as part of the process abstraction.

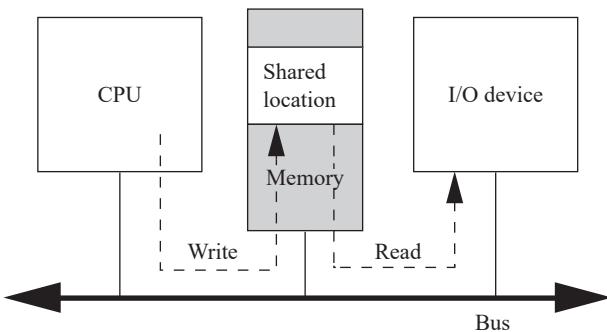


FIGURE 6.17

Shared memory communication implemented on a bus.

In general, a process can send a communication in one of two ways: **blocking** or **nonblocking**. After sending a blocking communication, the process goes into the waiting state until it receives a response. Nonblocking communication allows the process to continue execution after sending the communication. Both types of communication are useful.

There are two major styles of interprocess communication: **shared memory** and **message passing**. The two are logically equivalent; given one, you can build an interface that implements the other. However, some programs may be easier to write using one rather than the other. Moreover, the hardware platform may make it easier to implement or more efficient than the other.

### 6.6.1 Shared memory communication

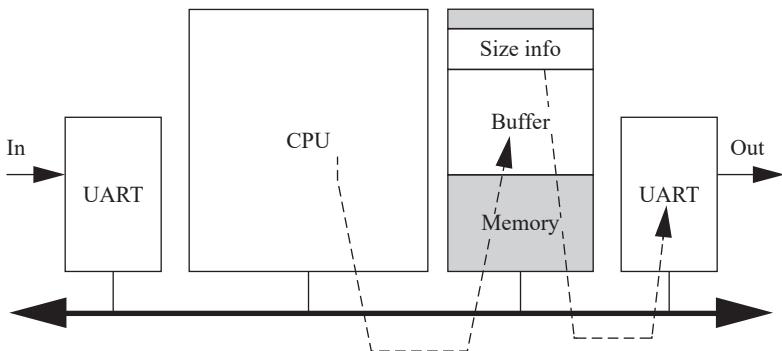
Fig. 6.17 illustrates how shared memory communication works in a bus-based system. Two components, a CPU and an I/O device, communicate through a shared memory location. The software on the CPU has been designed to know the address of the shared location; the shared location has also been loaded into the proper register of the I/O device. If, as in the figure, the CPU wants to send data to the device, it writes them to the shared location. The I/O device then reads the data from that location. The read and write operations are standard and can be encapsulated in a procedural interface.

Example 6.8 describes the use of shared memory as a practical communication mechanism.

---

#### Example 6.8: Elastic Buffers as Shared Memory

The text compressor of Application Example 3.4 provides a good example of a shared memory. As shown below, the text compressor uses the CPU to compress incoming text, which is then sent on a serial line by a Universal Asynchronous Receiver/Transmitter (UART).



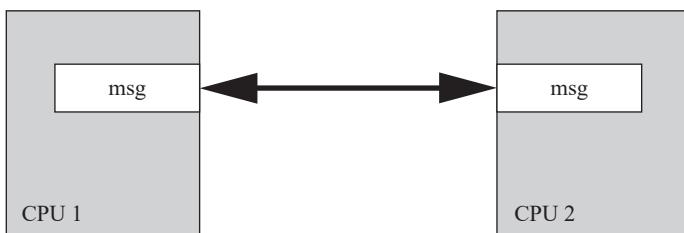
The input data arrive at a constant rate and are easy to manage. However, because the output data are consumed at a variable rate, these data require an elastic buffer. The CPU and output UART share a memory area; the CPU writes compressed characters into the buffer and the UART removes them as necessary to fill the serial line. Because the number of bits in the buffer changes constantly, the compression and transmission processes require additional size information. In this case, coordination is simple; the CPU writes at one end of the buffer and the UART reads at the other end. The only challenge is to ensure that the UART does not overrun the buffer.

### 6.6.2 Message passing

Message-passing communication complements the shared memory model. As shown in Fig. 6.18, each communicating entity has its own message send/receive unit. The message is not stored on the communications link but rather at the senders/receivers at the endpoints. In contrast, shared memory communication can be seen as a memory block used as a communication device in which all the data are stored in the communication link/memory.

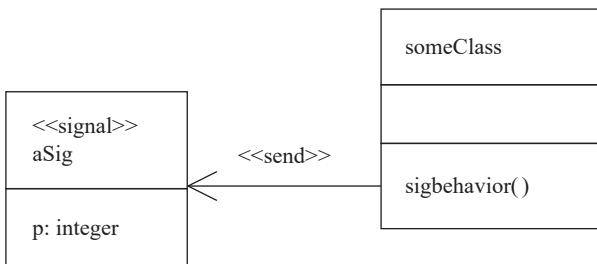
#### Messages

Applications in which units operate relatively autonomously are natural candidates for message-passing communication. For example, a home control system has one microcontroller per household device (e.g., lamp, thermostat, faucet, and appliance). The devices must communicate infrequently; furthermore, their physical



**FIGURE 6.18**

Message-passing communication.

**FIGURE 6.19**

Use of a UML signal.

separation is large enough that we would not naturally think of them as sharing a central pool of memory. Passing communication packets among devices is a natural way to describe coordination between these devices. Message passing is the natural implementation of communication in many 8-bit microcontrollers that do not normally operate with external memory.

### Queues

A **queue** is a common mechanism for keeping track of messages. The queue uses first in-first out (FIFO) discipline and holds records that represent messages. The [FreeRTOS.org](#) system provides a set of queue functions. It allows queues to be created and deleted so that the system may have as many queues as necessary. A queue is described by the data type `xQueueHandle` and created using `xQueueCreate`:

```

xQueueHandle q1;
q1 = xQueueCreate(MAX_SIZE,sizeof(msg_record)); /* maximum number of
records in queue, size of each record */
if (q1 == 0) /* error */
...
  
```

The queue is created using the `vQueueDelete()` function.

A message is put into the queue using `xQueueSend()` and received using `xQueueReceive()`:

```

xQueueSend(q1,(void *)msg,(portTickType)0); /* queue, message to
send, final parameter controls timeout */
if (xQueueReceive(q2,&(in_msg),0); /* queue, message received,
timeout */
  
```

The final parameter in these functions determines how long the queue waits to finish. In the case of a send, the queue may have to wait for something to leave to make room. In the case of the receive, the queue may have to wait for the data to arrive.

### 6.6.3 Signals

Another form of interprocess communication commonly used in Unix is the **signal**. A signal is simple because it does not pass data beyond the existence of the signal itself.

A signal is analogous to an interrupt, but it is entirely a software creation. A signal is generated by a process and transmitted to another process by the operating system.

A UML signal is actually a generalization of the Unix signal. While a Unix signal carries no parameters other than a condition code, a UML signal is an object. As such, it can carry parameters as object attributes. Fig. 6.16 shows the use of a signal in UML. The *sigbehavior()* behavior of the class handles throwing the signal, as indicated by <<send>>. The signal object is indicated by the <<signal>> stereotype.

#### 6.6.4 Mailboxes

A mailbox is a simple mechanism for asynchronous communication. Some architectures define mailbox registers. These mailboxes have a fixed number of bits and can be used for small messages. We can also implement a mailbox using semaphores with the mailbox messages held in the main memory. A very simple version of a mailbox, one that holds only one message at a time, illustrates some important principles of inter-process communication.

For the mailbox to be most useful, we want it to contain two items: the message itself and a mail-ready flag. The flag is true when a message has been put into the mailbox and cleared when the message is removed. This assumes that each message is destined for exactly one recipient. Here is a simple function to put a message into the mailbox, assuming that the system supports only one mailbox used for all messages:

```
void post(message *msg) {
    P(mailbox.sem); /* wait for the mailbox */
    copy(mailbox.data,msg); /* copy the data into the mailbox */
    mailbox.flag = TRUE; /* set the flag to indicate a message
                           is ready */
    V(mailbox.sem); /* release the mailbox */
}
```

Here is a function to read from the mailbox:

```
boolean pickup(message *msg) {
    boolean pickup = FALSE; /* local copy of the ready flag */
    P(mailbox.sem); /* wait for the mailbox */
    pickup = mailbox.flag; /* get the flag */
    mailbox.flag = FALSE; /* remember that this message was
                           received */
    copy(msg,mailbox.data); /* copy the data into the caller's
                           buffer */
    V(mailbox.sem); /* release the flag---can't get the mail if
                     we keep the mailbox */
    return(pickup); /* return the flag value */
}
```

Why do we need to use semaphores to protect the read operation? If we don't, a pickup could receive the first part of one message and the second part of another. The semaphores in `pickup()` ensure that a `post()` cannot interleave between the memory reads of the pickup operation.

## 6.7 Evaluating operating system performance

The scheduling policy does not tell us all that we would like to know about the performance of real system running processes. Our analysis of scheduling policies makes some simplifying assumptions:

- We assumed that context switches require zero time. Although it is often reasonable to neglect context switch time when it is much smaller than the process execution time, context switching can add significant delays in some cases.
- We have largely ignored interrupts. The latency from when an interrupt is requested to when the device's service is complete is a critical parameter of real-time performance.
- We assumed that we know the execution time of the processes. In fact, we learned in [Section 5.7](#) that program time is not a single number but can be bounded by worst-case and best-case execution times.
- We probably determined the worst-case or best-case times for the processes in isolation. However, in fact, they interact with each other in the cache. Cache conflicts among processes can drastically degrade the process execution time.

We must examine the validity of all these assumptions.

### Context switching time

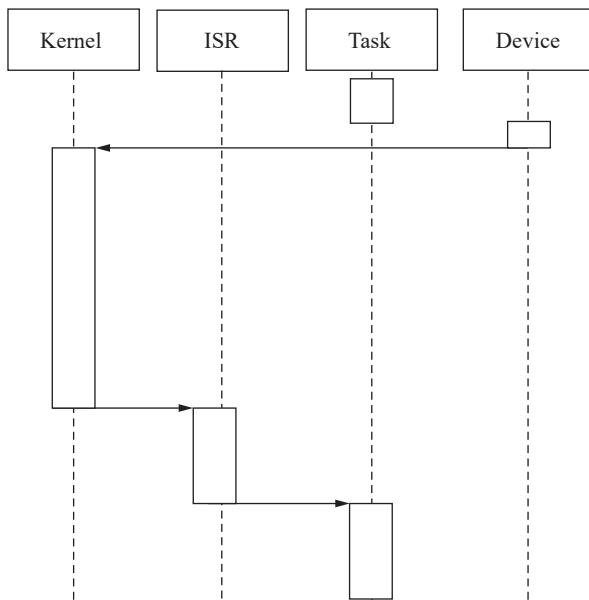
Context switching time depends on several factors:

- the amount of CPU context that must be saved; and
- scheduler execution time.

The execution time of the scheduler can, of course, be affected by coding practices. However, the choice of scheduling policy also affects the time required by the schedule to determine the next process to run. Scheduling complexity can be classified as a function of the number of tasks to be scheduled. For example, round-robin scheduling, although it doesn't guarantee deadline satisfaction, is a constant-time algorithm whose execution time is independent of the number of processes. Round-robin is often referred to as an  $O(1)$  scheduling algorithm because its execution time is constant independent of the number of tasks. EDF scheduling, in contrast, requires sorting deadlines, which is an  $O(n \log n)$  activity.

### Interrupt latency

**Interrupt latency** for an RTOS is the duration of time from the assertion of a device interrupt to the completion of the device's requested operation. In contrast, when we discussed CPU interrupt latency, we were concerned only with the time the hardware took to start the execution of the interrupt handler. Interrupt latency is critical because data may be lost when an interrupt is not serviced in a timely fashion.

**FIGURE 6.20**

Sequence diagram for RTOS interrupt latency.

**Fig. 6.20** shows a sequence diagram of RTOS interrupt latency. A task is interrupted by a device. The interrupt goes to the kernel, which may need to finish a protected operation. Once the kernel can process the interrupt, it calls the interrupt service routine (ISR), which performs the required operations on the device. Once the ISR is completed, the task can resume execution.

Several factors in both hardware and software affect interrupt latency:

- the processor interrupt latency;
- the execution time of the interrupt handler; and
- delays due to RTOS scheduling.

The processor interrupt latency was chosen when the hardware platform was selected; this is often not the dominant factor in overall latency. The execution time of the handler depends on the device operation required, assuming that the interrupt handler code is not poorly designed. This leaves RTOS scheduling delays, which can be the dominant component of RTOS interrupt latency, particularly if the operating system is not designed for low interrupt latency.

The RTOS can delay the execution of an interrupt handler in two ways. First, critical sections in the kernel prevent the RTOS from taking interrupts. A critical section may not be interrupted, so the semaphore code must turn off interrupts. Some operating systems have extensive critical sections that disable interrupt handling for extensive periods. Linux is an example of this phenomenon. Linux was not originally

designed for real-time operation, and interrupt latency was not a major concern. Longer critical sections can improve performance for some types of workloads because they reduce the number of context switches. However, long critical sections cause major problems for interrupts.

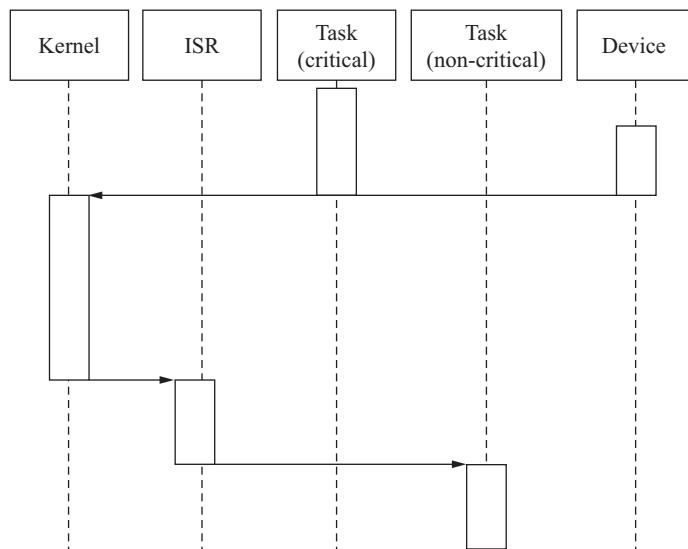
[Fig. 6.21](#) shows the effect of critical sections on interrupt latency. If a device interrupts during a critical section, that critical section must finish before the kernel can handle the interrupt. The longer the critical section, the greater the potential delay. Critical sections are an important source of scheduling jitter because a device may interrupt at different points in the execution of processes and hit critical sections at different points.

#### Interrupt priorities and interrupt latency

Second, a higher-priority interrupt may delay a lower-priority interrupt. A hardware interrupt handler runs as part of the kernel, not as a user thread. The priorities for interrupts are determined by hardware, not by the RTOS. Furthermore, any interrupt handler preempts all user threads because interrupts are part of the CPU's fundamental operation. We can reduce the effects of hardware preemption by dividing interrupt handling into two pieces of code. First, a very simple piece of code, usually called an **interrupt service handler (ISH)**, performs the minimal operations required to respond to the device. The rest of the required processing, which may include updating user buffers or other more complex operations, is performed by a user-mode thread known as an **interrupt service routine (ISR)**. Because the ISR runs as a thread, the RTOS can use its standard policies to ensure that all the tasks in the system receive their required resources.

#### RTOS performance evaluation tools

Some RTOSs provide simulators or other tools that allow you to view the operation of the processes in the system. These tools will show not only abstract events, such as



**FIGURE 6.21**

Interrupt latency during a critical section.

**Caches and RTOS performance**

processes, but also context switching time, interrupt response time, and other overheads. This sort of view can be helpful in both functional and performance debugging.

Many real-time systems have been designed based on the assumption that there is no cache present, even though one actually exists. This grossly conservative assumption is made because system architects lack tools that permit them to analyze the effect of caching. Because they do not know where caching will cause problems, they are forced to retreat to the simplifying assumption that there is no cache. The result is extremely overdesigned hardware, which has much more computational power than is necessary. However, just as experience tells us that a well-designed cache provides significant performance benefits for a single program, a properly sized cache can allow a microprocessor to run a set of processes much more quickly. By analyzing the effects of the cache, we can make much better use of the available hardware.

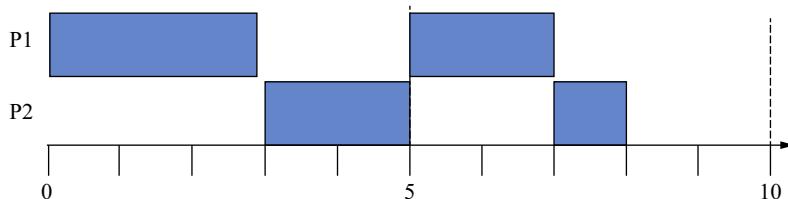
Li and Wolf [Li99] developed a model for estimating the performance of multiple processes that share a cache. In the model, some processes can be given reservations in the cache, such that only a particular process can inhabit a reserved section of the cache; other processes are left to share the cache. We generally want to use cache partitions only for performance-critical processes because cache reservations are wasteful of limited cache space. Performance is estimated by constructing a schedule that considers not just the execution time of the processes, but also the state of the cache. Each process in the shared section of the cache is modeled by a binary variable: 1 if present in the cache and 0 if not. Each process is also characterized by three total execution times: assuming no caching, with typical caching, and with all code always residing in the cache. The always-resident time is unrealistically optimistic, but it can be used to find a lower bound on the required schedule time. During the construction of the schedule, we can look at the current cache state to see whether the no-cache or typical-caching execution time should be used at this point in the schedule. We can also update the cache state if the cache is needed for another process. Although this model is simple, it provides much more realistic performance estimates than assuming that the cache either is nonexistent or is perfect. Example 6.9 shows how cache management can improve CPU utilization.

### **Example 6.9: Effects of Scheduling on the Cache**

Consider a system containing the following three processes:

Process	Worst-case CPU time	Average-case CPU time	Period
P1	3	2	5
P2	2	1	5

Each process runs slower when it is not resident in the cache, for example, on its first execution. It runs faster when it is cache-resident. If we can arrange the memory addresses of the processes so that they do not interfere in the cache, then their execution looks like this:



The first execution of each process runs at the worst-case execution time. In their second executions, each process is in the cache, and so runs in less time, leaving extra time at the end of the period.

## 6.8 POSIX real-time operating systems

In this section, we look at the Portable Operating System Interface (POSIX) standard for Unix-style operating systems and its support for real-time systems.

### Posix

POSIX is a version of the Unix operating system created by the IEEE Computer Society. POSIX-compliant operating systems are source-code compatible. An application can be compiled and run without modification on a new POSIX platform, assuming that the application uses only POSIX-standard functions. Although Unix was not originally designed as an RTOS, POSIX has been extended to support real-time requirements. Many RTOSs are POSIX-compliant, and it serves as a good model for basic RTOS techniques. The POSIX standard has many options, and particular implementations do not have to support all options. The existence of features is determined by C preprocessor variables; for example, the `FOO` option is available if the `_POSIX_FOO` preprocessor variable is defined. All these options are defined in the system, including file `unistd.h`.

The POSIX standard covers a huge expanse of operating systems and applications. The features that are useful for one application may not be applicable to another. Because features generally incur costs in memory size and performance, the appropriate choice of features is important for resource-limited devices. POSIX supports two mechanisms that are important to real-time systems [Gal92]: threads and real-time scheduling.

### POSIX threads

POSIX supports multiple processes, which is the way that many larger systems operate. However, these processes incur significant memory management overhead. For smaller real-time applications, POSIX provides a thread mechanism [Ope18] under the `pthread.h` include file. An application can use a single process to execute several concurrent threads that share the same memory space.

### Real-time scheduling in POSIX

POSIX supports two mechanisms for real-time scheduling: one for processes and another for threads [Har03]. Both processes and threads can use any of three different scheduling policies: `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`. Both `SCHED_FIFO` and

SCHED\_RR provide fixed-priority, preemptive scheduling. Unfortunately, the name of SCHED\_FIFO is misleading. It is a strict priority-based scheduling scheme in which a process runs until it is preempted or terminated. The term FIFO simply refers to the fact that, within a priority, processes run in first-come, first-served order.

---

Processes and threads may use different schedulers, causing a phenomenon known as **contention scope**. A global scheduling scope ignores process scheduling directives and schedules entirely at the thread level; a mixed scheduling scope first schedules processes and global threads, then local threads.POSIX mutexes

---

POSIX supports mutexes in the `pthread.h` include file [Ope18]. A mutex is created using `pthread_mutex_init()`, which returns a pointer to a struct of type `pthread_mutex_t`. The mutex can be locked using `pthread_mutex_lock()`, which will block if the mutex is already locked. The function `pthread_mutex_trylock()` will return immediately if the mutex is already locked. The function `pthread_mutex_unlock()` is used to unlock the mutex.

## Linux

The Linux operating system has become increasingly popular as a platform for embedded computing. Linux is a POSIX-compliant operating system that is available as an open source. However, Linux was not originally designed for real-time operations [Yag08, Hal11]. Some versions of Linux may exhibit long interrupt latencies, primarily owing to large critical sections in the kernel that delay interrupt processing. Two methods have been proposed to improve interrupt latency. A **dual-kernel** approach uses a specialized kernel, the **co-kernel**, for real-time processes and a standard kernel for non-real-time processes. All interrupts must go through the co-kernel to ensure that real-time operations are predictable. The other method is a kernel patch that provides priority inheritance to reduce the latency of many kernel operations. These features are enabled using the PREEMPT\_RT mode.

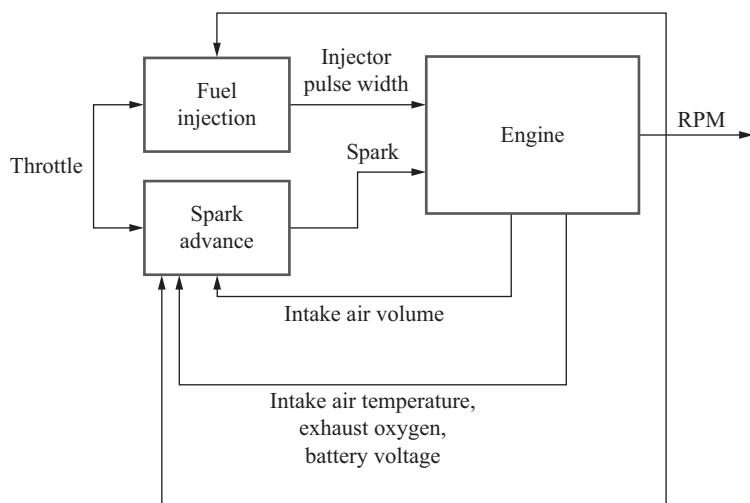
---

## 6.9 Design example: engine control unit

In this section, we design a simple engine control unit (ECU). This unit controls the operation of a fuel-injected engine based on several measurements taken from the running engine.

### 6.9.1 Theory of operation and requirements

We design a basic engine controller for a simple fuel-injected engine [Toy]. As shown in Fig. 6.22, the throttle is the command input. The engine measures throttle, revolutions per minute (RPM), air volume intake, and other variables. The engine controller computes the injector pulse width and spark. This design doesn't compute all the outputs required by a real engine; we only concentrate on a few essentials. We also ignore the different modes of engine operation: warm-up, idle, cruise, and so on. Multimode

**FIGURE 6.22**

Engine block diagram.

control is one of the principal advantages of engine control units, but we will concentrate here on a single mode to illustrate basic concepts in multirate control.

Our requirements chart for the ECU is shown in Fig. 6.23.

### 6.9.2 Specification

As we saw in Application Example 6.1, the engine controller must deal with processes that happen at different rates. Fig. 6.24 shows the update periods for the different signals.

Name	ECU
Purpose	Engine controller for fuel-injected engine
Inputs	Throttle, RPM, intake air volume, intake manifold pressure
Outputs	Injector pulse width, spark advance angle
Functions	Compute injector pulse width and spark advance angle as a function of throttle, RPM, intake air volume, intake manifold pressure
Performance	Injector pulse updated at 2-ms period, spark advance angle updated at 1-ms period
Manufacturing cost	Approximately \$50
Power	Powered by engine generator
Physical size and weight	Approx 4 in × 4 in, less than 1 pound.

**FIGURE 6.23**

Requirements for the engine controller.

Signal	Variable name	In/out	Update period (ms)
Throttle	T	input	2
RPM	NE	input	2
Intake air volume	VS	input	25
Injector pulse width	PW	output	2
Spark advance angle	S	output	1
Intake air temperature	THA	input	500
Exhaust oxygen	OX	input	25
Battery voltage	+B	input	4

**FIGURE 6.24**

Periods for data in the engine controller.

We will use  $\Delta NE$  and  $\Delta T$  to represent the change in RPM and throttle position, respectively. Our controller computes two output signals: injector pulse width PW and spark advance angle, S [Toy]. It first computes the initial values for these variables:

$$PW = \frac{2.5}{2NE} \times VS \times \frac{1}{10 - K_1 \Delta T} \quad (\text{Eq. 6.9})$$

$$S = k_2 \times \Delta NE - k_3 VS \quad (\text{Eq. 6.10})$$

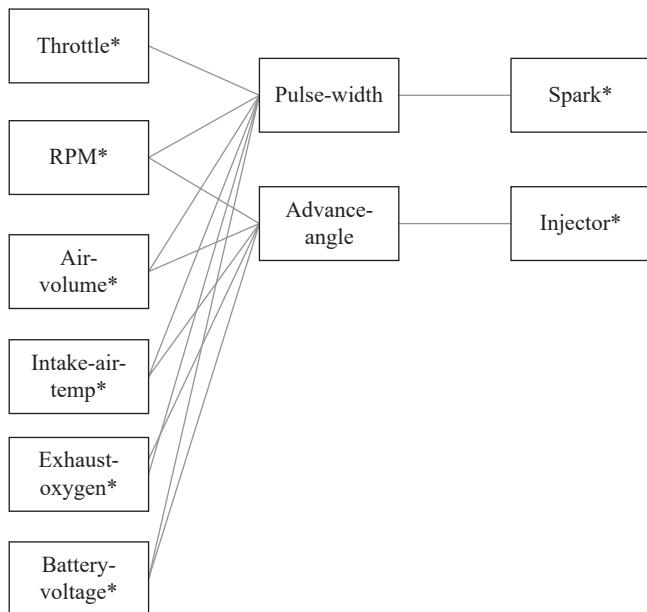
The controller then applies corrections to these initial values:

- As the intake air temperature (THA) increases during engine warm-up, the controller reduces the injection duration.
- As the throttle opens, the controller temporarily increases the injection frequency.
- The controller adjusts the duration up or down based upon readings from the exhaust oxygen sensor (OX).
- The injection duration increases as the battery voltage (+B) drops.

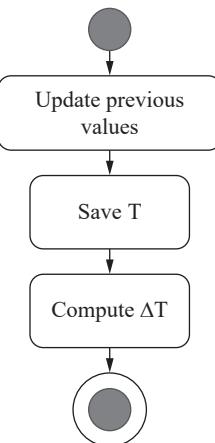
### 6.9.3 System architecture

Fig. 6.25 shows the class diagram for the engine controller. The two major processes, PW and advance angle, compute the control parameters for the spark plugs and injectors.

The control parameters rely on changes in some of the input signals. We use the physical sensor classes to compute these values. Each change must be updated at the variable's sampling rate. The update process is simplified by performing it in a task that runs at the required update rate. Fig. 6.26 shows the state diagram for throttle sensing, which saves both the current value and the change in value of the throttle. We can use a similar control flow to compute changes to the other variables.

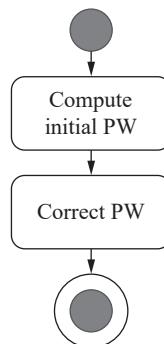
**FIGURE 6.25**

Class diagram for the engine controller.

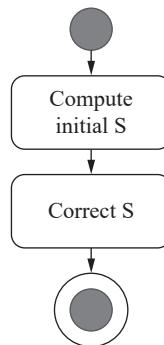
**FIGURE 6.26**

State diagram for throttle position sensing.

[Fig. 6.27](#) shows the state diagram for injector pulse width, and [Fig. 6.28](#) shows the state diagram for spark advance angle. In each case, the value is computed in two stages: first, an initial value, followed by a correction.

**FIGURE 6.27**

State diagram for injector pulse width.

**FIGURE 6.28**

State diagram for spark advance angle.

The pulse width and advance angle processes do not, however, generate the waveforms to drive the spark and injector waveforms. These waveforms must be carefully timed to the engine's current state. Each spark plug and injector must fire at exactly the right time in the engine cycle, accounting for the engine's current speed, as well as the control parameters.

Some engine controller platforms provide hardware units that generate high-rate, changing waveforms. One example is MPC5602 D [Fre11]. The main processor is a PowerPC processor. The enhanced modular I/O subsystem (eMIOS) provides 28 input and output channels controlled by timers. Each channel can perform a variety of functions. The output pulse width and frequency modulation buffered mode automatically generate a waveform whose period and duty cycle can be varied by writing registers in the eMIOS. The details of the waveform timing are then handled by the output channel hardware.

Because these objects must be updated at different rates, their execution will be controlled by an RTOS. Depending on the RTOS latency, we can separate the I/O functions into interrupt service handlers and threads.

#### 6.9.4 Component design and testing

The various tasks must be coded to satisfy the requirements of RTOS processes. Variables maintained across task execution, such as the change-of-state variables, must be allocated and saved in appropriate memory locations. The RTOS initialization phase is used to set up the task periods.

Because some of the output variables depend on changes in state, these tasks should be tested with multiple input variable sequences to ensure that both the basic and adjustment calculations are performed correctly.

The Society of Automotive Engineers (SAE) has several standards for automotive software: J2632 for coding practices for C code, J2516 for software development lifecycle, J2640 for software design requirements, and J2734 for software verification and validation.

#### 6.9.5 System integration and testing

Engines generate huge amounts of electrical noise that can cripple digital electronics. They also operate over vast temperature ranges: hot during engine operation, and potentially very cold before the engine is started. Any testing performed on an actual engine must be conducted using an engine controller that has been designed to withstand the harsh environment of the engine compartment.

---

## 6.10 Summary

The process abstraction is forced on us by the need to satisfy complex timing requirements, particularly for multirate systems. Writing a single program that simultaneously satisfies deadlines at multiple rates is too difficult because the control structure of the program becomes unintelligible. The process encapsulates the state of a computation, allowing us to easily switch among different computations.

The operating system encapsulates complex control to coordinate the process. The scheme used to determine the transfer of control among processes is known as a scheduling policy. A good scheduling policy is useful across many different applications, while also providing efficient utilization of the available CPU cycles.

However, it is difficult to achieve 100% utilization of the CPU for complex applications. Because of variations in data arrivals and computation times, reserving some cycles to meet worst-case conditions is necessary. Some scheduling policies achieve higher utilizations than others, but often at the cost of unpredictability; they may not guarantee that all deadlines are met. Knowledge of the characteristics

of an application can be used to increase CPU utilization, while also complying with deadlines.

---

## What we learned

- A process is a single thread of execution.
- Preemption is the act of changing the CPU's execution from one process to another.
- A scheduling policy is a set of rules that determines the process to run.
- Rate-monotonic scheduling is a simple but powerful scheduling policy.
- Interprocess communication mechanisms allow data to be passed reliably between processes.
- Scheduling analysis often ignores certain real-world effects. Cache interactions between processes are the most important effects to consider when designing a system.

---

## Further reading

Gallmeister [Gal95] provides a thorough and very readable introduction to POSIX in general and its real-time aspects in particular. Liu and Layland [Liu73] introduced RMS; their paper became the foundation for real-time systems analysis and design. The book by Liu [Liu00] provides a detailed analysis of real-time scheduling.

---

## Questions

- Q6-1** Identify activities that operate at different rates in
- a DVD player
  - a laser printer
  - an airplane
- Q6-2** Name an embedded system that requires both periodic and aperiodic computation.
- Q6-3** An audio system processes samples at a rate of 44.1 kHz. At what rate could we sample the system's front panel to both simplify the analysis of the system schedule and provide adequate response to the user's front panel requests?
- Q6-4** Draw a UML class diagram for a process in an operating system. The process class should include the necessary attributes and behaviors required for a typical process.

- Q6-5** Draw a task graph in which P1 and P2 each process separate inputs, and then, pass their results onto P3 for further processing.
- Q6-6** Compute the utilization for these task sets:
- P1: period = 1 s, execution time = 10 ms; P2: period = 100 ms, execution time = 10 ms.
  - P1: period = 100 ms; execution time = 25 ms; P2: period = 80 ms; execution time = 15 ms; P3: period = 40 ms; execution time = 5 ms.
  - P1: period = 10 ms; execution time = 1 ms; P2: period = 1 ms; execution time = 0.2 ms; P3: period = 0.2 ms; execution time = 0.05 ms.
- Q6-7** What factors provide a lower bound on the period at which the system timer interrupts for preemptive context switching?
- Q6-8** What factors provide an upper bound on the period at which the system timer interrupts for preemptive context switching?
- Q6-9** What is the distinction between the ready and waiting states of process scheduling?
- Q6-10** A set of processes changes state, as shown over the interval [0,1 ms]. P1 has the highest priority, and P3 has the lowest priority. Draw a UML sequence diagram showing the state of all the processes during this interval.

<b>t</b>	<b>Process states</b>
0	P1 = waiting, P2 = waiting, P3 = executing
0.1	P1 = ready
0.15	P2 = ready
0.2	P1 = waiting
0.3	P1 = ready, P3 = ready
0.4	P1 = waiting
0.5	P2 = waiting
0.6	P3 = waiting
0.8	P2 = ready, P3 = ready
0.9	P2 = waiting

- Q6-11** Provide examples of
- blocking interprocess communication
  - nonblocking interprocess communication
- Q6-12** For the following periodic processes, what is the shortest interval we must examine to see all combinations of deadlines?

a.

Process	Deadline
P1	2
P2	5
P3	10

b.

Process	Deadline
P1	2
P2	4
P3	5
P4	10

c.

Process	Deadline
P1	3
P2	4
P3	5
P4	6
P5	10

- Q6-13** Consider the following system of periodic processes executing on a single CPU:

Process	Execution time	Deadline
P1	4	200
P2	1	10
P3	2	40
P4	6	50

Can we add another instance of P1 to the system and meet all the deadlines using RMS?

- Q6-14** Given the following set of periodic processes running on a single CPU (P1 has highest priority), what is the maximum execution time  $x$  of P3 for which all the processes will be schedulable using EDF?

Process	Execution time	Deadline
P1	1	10
P2	3	25
P3	$x$	50
P4	10	100

- Q6-15** A set of periodic processes is scheduled using RMS; P1 has the highest priority. For the process execution times and periods shown below, show the state of the processes at the critical instant for each process.

- a. P1
- b. P2
- c. P3

Process	Time	Deadline
P1	1	4
P2	1	5
P3	1	10

- Q6-16** For the given periodic process execution times and periods (P1 has the highest priority), show how much CPU time of higher-priority processes will be required during one period of each of the following processes:

- a. P1
- b. P2
- c. P3
- d. P4

Process	Time	Deadline
P1	1	5
P2	2	10
P3	2	25
P5	5	50

**Q6-17** For the periodic processes shown below:

- Schedule the processes using an RMS policy.
- Schedule the processes using an EDF policy.

In each case, compute the schedule for an interval equal to the least-common multiple of the periods of the processes. P1 has the highest priority, and time starts at  $t = 0$ .

Process	Time	Deadline
P1	1	3
P2	1	4
P3	1	12

**Q6-18** For the periodic processes shown below:

- Schedule the processes using an RMS policy.
- Schedule the processes using an EDF policy.

In each case, compute the schedule for an interval equal to the least-common multiple of the periods of the processes. P1 has the highest priority and time starts at  $t = 0$ .

Process	Time	Deadline
P1	1	3
P2	1	4
P3	2	6

**Q6-19** For the periodic processes shown below:

- Schedule the processes using an RMS policy.
- Schedule the processes using an EDF policy.

In each case, compute the schedule for an interval equal to the least-common multiple of the periods of the processes. P1 has the highest priority and time starts at  $t = 0$ .

Process	Time	Deadline
P1	1	2
P2	1	3
P3	2	10

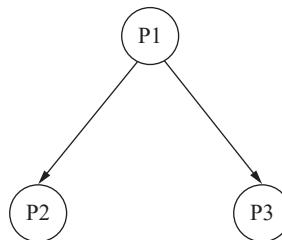
**Q6-20** For the given set of periodic processes, all of which share the same deadline of 12:

- Schedule the processes for the given arrival times using standard RMS (no data dependencies).
- Schedule the processes, taking advantage of the data dependencies. By how much is the CPU utilization reduced?

Process	Execution time
P1	2
P2	1
P3	2

**Q6-21** For the periodic processes given below, find a valid schedule

- using standard RMS; and
- adding one unit of overhead for each context switch.



Process	Time	Deadline
P1	2	30
P2	5	40
P3	7	120
P4	5	60
P5	1	15

**Q6-22** For the periodic processes and deadlines given below:

- Schedule the processes using RMS.
- Schedule using EDF and compare the number of context switches required for EDF and RMS.

Process	Time	Deadline
P1	1	5
P2	1	10
P3	2	20
P4	10	50
P5	7	100

- Q6-23** If you wanted to reduce the cache conflicts between the most computationally intensive parts of two processes, what are two ways that you could control the locations of the processes' cache footprints?
- Q6-24** A system has two processes P1 and P2, with P1 having higher priority. They share an I/O device ADC. If P2 acquires the ADC from the RTOS and P1 becomes ready, how does the RTOS schedule the processes using priority inheritance?
- Q6-25** Explain the roles of IRSs and interrupt service handlers in interrupt handling.
- Q6-26** Briefly explain the dual-kernel approach to RTOS design.

## Lab exercises

- L6-1** Using your favorite operating system, write code to spawn a process that writes "Hello, world!" to the screen or flashes a light-emitting diode (LED), depending on your available output devices.
- L6-2** Build a small serial port device that lights LEDs based on the last character written to the serial port. Create a process that will light LEDs based on keyboard input.
- L6-3** Write a driver for an I/O device.
- L6-4** Write context switch code for your favorite CPU.
- L6-5** Measure context switching overhead on an operating system.
- L6-6** Using a CPU that runs an operating system that uses RMS, try to get the CPU utilization up to 100%. Vary the data arrival times to test the robustness of the system.
- L6-7** Using a CPU that runs an operating system that uses EDF, try to get the CPU utilization as close to 100% as possible without failing. Try a variety of data arrival times to determine how sensitive your process set is to environmental variations.

- L6-8** Measure the effect of cache conflicts on real-time execution time. First, set up your system to measure the execution time of your real-time process. Next, add a background process to the system. One version of the background process should do nothing; another should do some work that will invalidate as many of the cache entries as possible.

# System Design Techniques

7

## CHAPTER POINTS

- A deeper look into design methodologies, requirements, specifications, and system analysis.
- System modeling.
- Formal and informal methods for system specification.
- Dependability, security, and safety.

## 7.1 Introduction

In this chapter, we consider the techniques required to create complex embedded systems. Thus far, our design examples have been small, so important concepts can be conveyed simply. However, most embedded system designs are inherently complex, given that their functional specifications are rich, and they must obey multiple other requirements on cost, performance, and so on. We require methodologies to help guide our design decisions when designing large systems.

In [Section 7.2](#), we look at design methodologies in more detail. [Section 7.3](#) studies use cases and requirement analysis, which capture informal descriptions of what a system must do, and [Section 7.4](#) considers techniques for more formally specifying system modeling and functionality. [Section 7.5](#) proceeds to system analysis and architecture design. [Section 7.6](#) focuses on safety and security as aspects of dependability.

## 7.2 Design methodologies

This section considers the complete **design methodology** (a **design process**) for embedded computing systems. We start with the rationale for design methodologies, and then we look at several different methodologies.

### 7.2.1 Why design methodologies?

The process is important because without it we can't reliably deliver the products we want to create. Thinking about the sequence of steps necessary to build something

**Product metrics**

may seem superfluous, but the fact is that everyone has their own design process, even if they don't articulate it. If you are designing embedded systems in your basement by yourself, having your own work habits is fine. However, when several people work together on a project, they need to agree on who will do things and how those things will be done. Being explicit about the process is important when people work together. Because many embedded computing systems are too complex to be designed and built by one person, we must consider design processes.

The obvious goal of a design process is to create a product that does something useful. Typical specifications for a product will include functionality (e.g., wearable health monitor), manufacturing cost (e.g., must have a retail price below \$200), performance (e.g., must power up within 2 s), power consumption (e.g., must run for 12 h without recharge), and other properties. Of course, a design process has several important goals beyond function, performance, and power:

- *Time-to-market.* Customers always want new features. A product that comes out first can win the market, even while setting customer preferences for future generations of the product. The profitable market life for some products is 3–6 months. If you are 3 months late, you will never make money. In some categories, the competition is against the calendar, not just against competitors. Calculators, for example, are disproportionately sold just before school starts in the fall. If you miss your market window, you must wait a year for another sales season.
- *Design cost.* Many consumer products are very cost sensitive. Industrial buyers are also increasingly concerned about costs. The costs of designing the system are distinct from the manufacturing cost. The cost of engineers' salaries, computers used in design, and so on must be spread across the units sold. In some cases, only one or a few copies of an embedded system may be built; hence, design costs can dominate manufacturing costs. Design costs can also be important for high-volume consumer devices when time-to-market pressures cause teams to swell in size.
- *Quality.* Customers not only want their products fast and cheap, but they also want them to be right. A design methodology that cranks out shoddy products will eventually be forced out of the marketplace. Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job, to obtain a high-quality product at the end.

Design processes evolve over time owing to external and internal forces. Customers change, requirements change, products change, and available components change. Internally, people learn how to do things better, people move on to other projects, others come in, and companies are bought and sold to merge and shape corporate cultures.

Software engineers have spent a great deal of time thinking about software design processes. Much of this thinking has been motivated by mainframe software such as databases. However, embedded applications have also inspired some important thinking about software design.

A good methodology is critical to building systems that work properly. Delivering buggy systems to customers always causes dissatisfaction. However, in some

applications, such as medical and automotive systems, bugs create serious safety problems that can endanger users' lives. We discuss quality in more detail in [Section 7.6.1](#). As an introduction, the next three examples discuss software errors that affect space missions.

---

### Example 7.1: Loss of the Mars Climate Observer

In September 1999, the Mars Climate Observer, an unmanned U.S. spacecraft designed to study Mars, was lost. It most likely exploded as it heated up in the atmosphere of Mars after approaching the planet too closely. This occurred because of a series of problems, according to an analysis by *IEEE Spectrum* and contributing editor James Oberg [Obe99]. From an embedded systems perspective, the first problem is best classified as a requirement problem. The contractors who built the spacecraft at Lockheed Martin calculated the values for the flight controllers at the Jet Propulsion Laboratory (JPL). JPL did not specify the physical units to be used, but it expected them to be in Newtons. The Lockheed Martin engineers returned values in units of pound force. This discrepancy resulted in trajectory adjustments being 4.45 times larger than they should have been. The error was not caught by a software configuration process, nor was it caught by manual inspections. Although there were concerns about the spacecraft's trajectory, errors in the calculation of the spacecraft's position were not caught in time.

---

---

### Example 7.2: New Horizons Communications Blackout

New Horizons, a NASA probe, experienced an 81-min radio communications blackout while approaching its Jupiter fly-by [Klo15]. The blackout was traced to a timing bug in New Horizons's command sequence.

---

---

### Example 7.3: LightSail File Overflow Bug

A file overflow bug caused the LightSail satellite to malfunction in orbit [Cha15]. A telemetry data file was allowed to grow beyond its allocated memory. The bug was not found in pre-flight testing because the tests were not run for a sufficient period. The bug occurred only after about 40 h of operation. In this case, after 8 days, cosmic rays caused the computer to reset, allowing the machine to be restarted. The machine was then restarted once a day to avoid activating the bug again.

---

## 7.2.2 Design methodologies for embedded computing

A design methodology (**design flow**) is a sequence of steps to be followed during design. Some steps can be performed by tools, such as compilers or CAD systems; other steps can be performed by hand. In this section, we look at the basic characteristics of design flows.

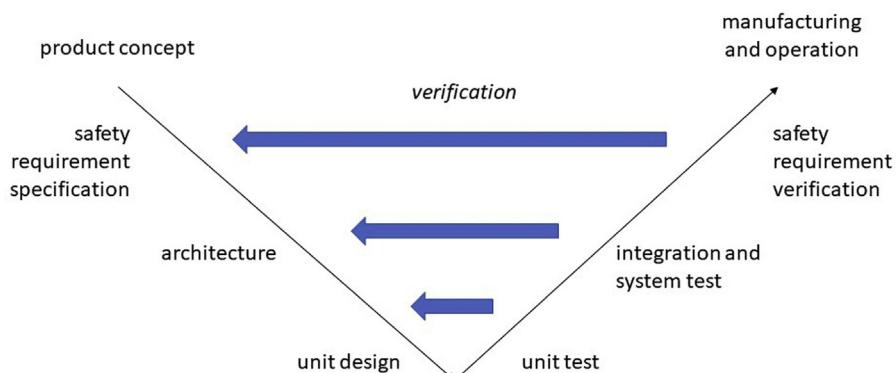
Design methodologies created for other domains, such as Web sites or financial transactions, may not provide all characteristics and assurances required for embedded computing systems. The **waterfall** model was an early software design methodology that moved from requirements to architecture, coding, testing, and maintenance. Only local interactions between adjacent stages were allowed. The waterfall model is regarded as not sufficiently flexible for large modern software systems. **Agile** models are often used to design consumer-oriented software; they do not generally afford the documentation and testing regimens required for real-time systems.

### V model

The **V model** [ISO18B] was defined to support safety methodologies for automobiles, and has been widely adopted for real-time embedded systems. As shown in Fig. 7.1, this methodology uses a top-down design and bottom-up verification. The bottom-up verification phases correspond to the top-down design phases, so that the design is verified from smallest to largest units. In the spirit of hierarchical design flows, the V model is also applied to both hardware and software design processes within the overall system design. We discuss ISO 26262, the standard that introduced the V model, in Section 7.6.5.

### Hierarchical design flows

Many complex embedded systems are built with smaller designs. The complete system may require the design of significant software components, field-programmable gate arrays (FPGAs), and so on, and these in turn may be built from smaller components that need to be designed. The design flow follows the levels of abstraction in the system, from complete system designs to individual components. The implementation phase of a flow is a complete flow from specification through testing. In such a large project, each flow will be handled by separate people or teams. The teams must rely on each other's results. The component teams take their requirements from the team handling the next higher level of abstraction, and the higher-level team relies on the quality of the design and testing performed by the component team. Good communication is vital in such large projects.



**FIGURE 7.1**

ISO 26262 V model [ISO18B].

## 7.3 Requirements analysis and specification

Before designing a system, we need to know what we are designing. We need to gather information from a variety of sources to determine the desired characteristics of the system. We also need to capture these characteristics in ways that can be used by the rest of the design team.

### Requirements and specifications

A **requirement** is a description of a desired characteristic of the system: some aspect of its behavior, its responsiveness, cost, and so on. A **specification** is a complete set of requirements for a system. Requirements and specifications are, however, directed toward the outward behavior of the system, not its internal structure.

### Functional and nonfunctional requirements

There are two types of requirements: **functional** and **nonfunctional**. A functional requirement states what the system must do, such as computing an FFT. A nonfunctional requirement can be any number of other attributes, including physical size, cost, power consumption, design time, reliability, and so on.

We often start with a more informal process to determine the basic requirements of the system. We then refine those characteristics to form a complete specification.

### 7.3.1 Requirements capture

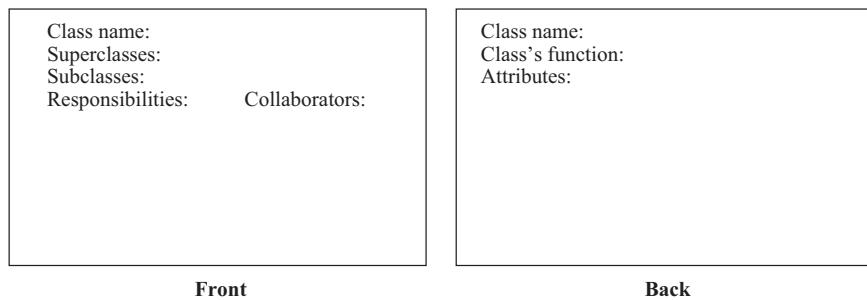
How do you determine the requirements? If the product is a continuation of a series, then many of the requirements will be well understood. However, even with the most modest upgrade, talking to the customer is valuable. In a large company, marketing or sales departments may do most of the work of asking customers what they want, but a surprising number of companies have designers talk directly with customers. Direct customer contact gives the designer an unfiltered sample of what the customer needs. It also helps build empathy with the customer, which often pays off in cleaner, easier-to-use customer interfaces. Talking to the customer may also include conducting surveys, organizing focus groups, or asking selected customers to test a mockup or prototype.

### Informal requirements capture

Although the final architecture must be complete and correct, informal methods of requirement analysis can help start the process. The **CRC card** methodology is a well-known and useful way to help analyze a system. The acronym *CRC* stands for these three major items that the methodology tries to identify:

- *Classes* define the logical groupings of data and functionality.
- *Responsibilities* describe what the classes do.
- *Collaborators* are the other classes with which a given class works.

The CRC card methodology has people write on index cards. In the United States, the standard size for index cards is 3×5 in. Hence, these cards are often called “3×5 cards.” An example card is shown in Fig. 7.2; it has space to write down the class name, its responsibilities and collaborators, and other information. The essence of the CRC card methodology is to have people write on these cards, talk about them, and update them until they are satisfied with the descriptions.

**FIGURE 7.2**

Layout of a CRC card.

This technique may seem like a primitive way to design computer systems. However, it has several important advantages. First, it is easy to get noncomputer people to create CRC cards. Getting the advice of domain experts (e.g., automobile designers for automotive electronics or human factors experts for appliance design) is vital to system design. The CRC card methodology is informal enough that it will not intimidate noncomputer specialists and will allow you to capture their input. Second, it aids computer specialists by encouraging them to work in groups to analyze scenarios. The walkthrough process used with CRC cards is especially useful in scoping out a design and determining which parts of a system are poorly understood. This informal technique is valuable for tool-based design and coding. If you still feel a need to use tools to help you practice the CRC methodology, software engineering tools are available that automate the creation of CRC cards.

Before going through the methodology, let's review the CRC concepts in a little more detail. We are familiar with classes; they encapsulate functionality. A class may represent a real-world object or it may describe an object that has been created solely to help architect a system. A class has both an internal state and a functional interface; the functional interface describes the class's capabilities. The responsibility set is an informal way of describing the functional interface. The responsibilities provide the class's interface, not its internal implementation. Unlike describing a class in a programming language, the responsibilities may be described informally in English (or in your favorite language). The collaborators of a class are simply the classes to which it talks: classes that use its capabilities or that it calls upon to help it do its work.

The class terminology is a little misleading when an object-oriented programmer looks at CRC cards. In the methodology, a class is used more like an object in an object-oriented programming language. The CRC card class is used to represent a real actor in the system. However, the CRC card class is easily transformed into a class definition in an object-oriented design.

CRC card analysis is performed by a team of people. It is possible to use it by yourself, but a lot of the benefit of the method comes from talking about developing classes with others. Before beginning the process, you should create many copies of

blank CRC cards using the basic format shown in Fig. 7.2. As you work in a group, you will write on these cards and will discard many just to rewrite them as the system evolves. The CRC card methodology is informal, but you should go through the following steps when using it to analyze a system:

1. *Develop an initial list of classes.* Write down the class name and perhaps a few words on what it does. A class may represent a real-world object or an architectural object. Identifying which category the class falls into, perhaps by putting a star next to the name of a real-world object, is helpful. Each person can be responsible for handling a part of the system, but team members should talk during this process to ensure that no classes are missed and that duplicate classes are not created.
2. *Write an initial list of responsibilities and collaborators.* The responsibilities list helps describe in more detail what the class does. The collaborators' list should be built from obvious relationships between the classes. Both the responsibilities and collaborators will be refined in later stages.
3. *Create some usage scenarios.* These scenarios describe what the system does. Scenarios begin with some type of outside stimulus, which is an important reason for identifying relevant real-world objects.
4. *Walk through the scenarios.* This is the heart of the methodology. During the walkthrough, each person on the team represents one or more classes. The scenario should be simulated by action. Team members may announce what their class does, ask other classes to perform operations, and so on. Moving around, for example, to show the transfer of data, team members may visualize the system's operation. During the walkthrough, all information created thus far is targeted for updating and refinement, including the classes, responsibilities, collaborators, and usage scenarios. Classes may be created, destroyed, or modified during this process. You will also probably find many holes in the scenario itself.
5. *Refine the classes, responsibilities, and collaborators.* Some of this will be done during the walkthrough, while adding a second pass after the scenarios. A longer perspective will help you make more global changes to the CRC cards.
6. *Add class relationships.* When CRC cards have been refined, subclass and superclass relationships should become clearer and can be added.

The CRC cards must then be used to help drive the implementation. In some cases, it may work best to use the CRC cards as direct source material for the implementors; this is particularly true if you can get the designers involved in the CRC card process. In other cases, you may want to write a more formal description in UML or other information captured during the CRC card analysis. This formal description is then used to build the design document for the system implementors. Example 7.5 illustrates the use of the CRC card methodology.

---

### Example 7.4: CRC Card Analysis of an Elevator System

Let's perform a CRC card analysis of an elevator system. First, we need the following basic set of classes:

- *Real-world classes*: elevator car, passenger, floor control, car control, and car sensor.
- *Architectural classes*: car state, floor control reader, car control reader, car control sender, and scheduler.

For each class, we need the following initial set of responsibilities and collaborators. An asterisk is used to remind ourselves which classes represent real-world objects.

Class	Responsibilities	Collaborators
Elevator car*	Moves up and down	Car control, car sensor, car control sender
Passenger*	Pushes floor control and car control buttons	Floor control, car control
Floor control*	Transmits floor requests	Passenger, floor control reader
Car control*	Transmits car requests	Passenger, car control reader
Car sensor*	Senses car position	Scheduler
Car state	Records current position of car	Scheduler, car sensor
Floor control reader	Interface between floor control and rest of system	Floor control, scheduler
Car control reader	Interface between car control and rest of system	Car control, scheduler
Car control sender	Interface between scheduler and car	Scheduler, elevator car
Scheduler	Sends commands to cars based upon requests	Floor control reader, car control reader, car control sender, car state

Several usage scenarios define the basic operation of the elevator system as well as some unusual scenarios:

1. One passenger requests a car on a floor, gets in the car when it arrives, requests another floor, and gets out when the car reaches that floor.
2. One passenger requests a car on the floor, gets in the car when it arrives, and requests the floor that the car is currently on.
3. A second passenger requests a car, while another passenger is riding in the elevator.
4. Two people push the floor buttons on different floors at the same time.
5. Two people push car control buttons in different cars at the same time.

At this point, we must walk through the scenarios and make sure they are reasonable. Therefore, we should find a set of people and walk through these scenarios. Do the classes, responsibilities, collaborators, and scenarios make sense? How would you modify them to improve the system specifications?

---

### Characteristics of specifications

#### 7.3.2 From requirements to specification

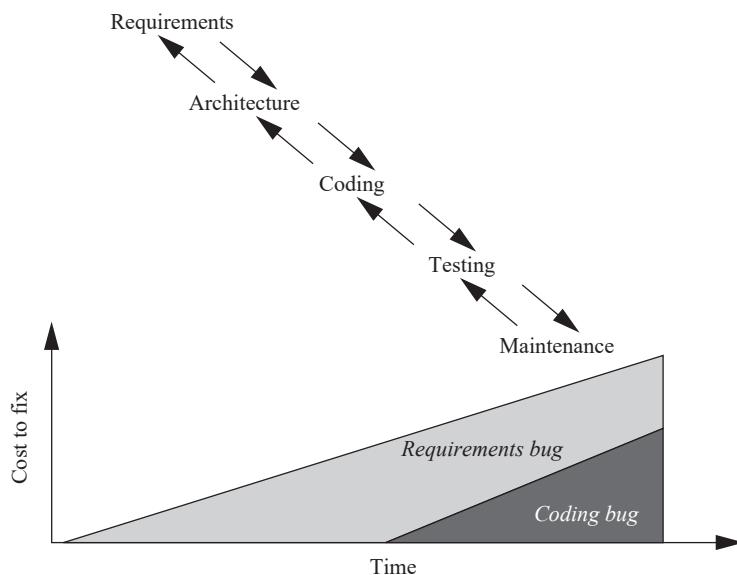
A good specification should meet several tests [Dav90]:

- *Correctness*: The requirements should not mistakenly describe what the customer wants. Part of correctness is avoiding over-requiring; the requirements should not add unnecessary conditions.
- *Unambiguousness*: The requirements document should be clear and have only one plain language interpretation.
- *Completeness*: All requirements should be included.
- *Verifiability*: There should be a cost-effective way to ensure that each requirement is satisfied in the final product. For example, a requirement that the system package be “attractive” would be hard to verify without some agreed upon definition of attractiveness.
- *Consistency*: One requirement should not contradict another requirement.
- *Modifiability*: The requirements document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, and so on.
- *Traceability*: Each requirement should be traceable in the following ways:
  - We should be able to trace backward from the requirements to know why each requirement exists.
  - We should be able to trace forward from documents created before the requirements (e.g., marketing memos) to understand how they relate to the final requirements.
  - We should be able to trace forward and understand how each requirement is satisfied in the implementation.
  - We should also be able to trace backward from the implementation to know which requirements they were intended to satisfy.

#### 7.3.3 Validating the specification

The specification is generated early in the design process. The goal of validation is to ensure that the specification properly captures the user’s needs.

Validating the specification is important for the simple reason that bugs in the requirements or specifications can be extremely expensive to fix later. Fig. 7.3 shows how the cost of fixing bugs grows over the course of the design process. We use the waterfall model as a simple example, but the same holds for any design flow. The longer a bug survives in the system, the more expensive it will be to fix it. A coding bug, if not found until after system deployment, will cost money to recall and reprogram, among other things. However, a bug introduced earlier in the flow and not discovered until the same point will accrue all costs and more. A bug was introduced in the specification and left until maintenance could force an entire redesign of the product, not just the distribution of a software update. Discovering bugs early is crucial because it prevents them from being released to customers, minimizes design costs, and reduces design time. Although some specification bugs will become apparent in the detailed design stages

**FIGURE 7.3**

Long-lived bugs are more expensive to fix.

(e.g., as the consequences of certain requirements are better understood), it is possible and desirable to weed out as many bugs as possible during the generation of the requirements and specifications.

The goal of validating the specification is to ensure that it satisfies the criteria originally applied in [Section 7.3.2](#): correctness, completeness, consistency, and so on. Validation is part of the larger requirements and specification process. Various techniques can be applied, while they are being created, to help you understand the requirements and specifications, whereas others can be applied to a draft with the results used to modify the specifications.

#### Prototyping

**Prototypes** are useful when dealing with end users. Rather than simply describing the system to them in broad technical terms, a prototype can let them see, hear, and touch at least some of the important aspects of the system. Of course, the prototype will not be fully functional, because the design work has not yet been done. However, user interfaces are well suited for prototyping and user testing. Canned or randomly generated data can be used to simulate the system's internal operation. A prototype can help the end user critique numerous functional and nonfunctional requirements, such as data displays, speed of operation, size, weight, and so forth. Certain programming languages, sometimes called **prototyping languages** or **specification languages**, are especially well suited to prototyping. Very high-level languages, such as MATLAB in the signal processing domain, may be able to perform functional attributes, such as the mathematical function to be performed, but not nonfunctional attributes, such as the speed of execution. **Preeexisting systems** can also be used to

help end users articulate their needs. Specifying what someone does or doesn't like about an existing machine is much easier than having them talk about the new system in the abstract. In some cases, it may be possible to construct a prototype of the new system from the preexisting system.

Auditing tools may be useful in verifying consistency, completeness, and so forth. Working through **usage scenarios** often helps designers fill out the details of a specification and ensure its completeness and correctness.

#### Formal methods

In some cases, **formal methods** (that is, design techniques that make use of mathematical proofs) may be useful. Proofs may be done either manually or automatically. In some cases, proving that a particular condition can or cannot occur according to the specification is important. Automated proofs are particularly useful in certain types of complex systems that can be specified succinctly, but whose behavior over time is complex. For example, complex protocols have been successfully formally verified in this fashion.

## 7.4 System modeling

In this section, we look at some advanced techniques for specification and how they can be used. [Section 7.4.1](#) looks at a model-based design, and [Section 7.4.2](#) studies two UML dialects for modeling.

### 7.4.1 Model-based design

#### Model-based design

**Model-based design** [Kar03] has emerged as an important methodology for cyber-physical systems. This approach is more holistic than traditional design flows because it takes a unified view of both the physical plant and the embedded computing system.

Model-based design makes use of a set of tools that compile a description of cyber-physical systems into a set of implementations: software, computing platform, and physical plant components. Given the wide range of cyber-physical systems, we should not expect a single toolset to serve all applications. Instead, a set of tools is created to serve a particular class of designs.

#### Domain-specific modeling languages

Designers create their designs in a **domain-specific modeling language (DSML)**. This language is designed to be used by a domain expert, such as an aircraft or automobile designer. The DSML captures both the functional and nonfunctional characteristics of the system. This description can be used in several ways:

- The cyber-physical system specification can be simulated, for example, by generating a Simulink model.
- The implementation design space can be explored under a range of design parameters.
- Given the choice of design parameters, an implementation can be synthesized.

The integration of the tools ensures that the implementation is consistent with the high-level simulation and the original specifications.

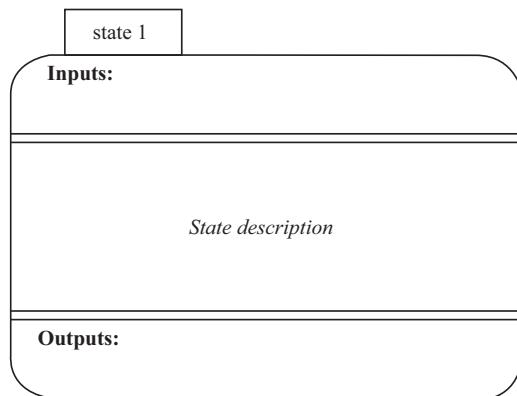
Example 7.5 describes the specification of a real-world, safety-critical system used in aircraft. The specification techniques developed to ensure the correctness and safety of this system can also be used in many applications, particularly in systems where much of the complexity goes into the control structure.

---

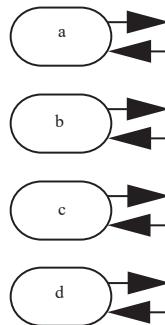
### Example 7.5: Traffic Alert and Collision Avoidance System II Specification

TCAS II (Traffic Alert and Collision Avoidance System) is a collision avoidance system for aircraft. Based on a variety of information, a TCAS unit in an aircraft keeps track of the position of other nearby aircraft. If TCAS decides that a mid-air collision may be likely, it uses audio commands to suggest evasive action. For example, a prerecorded voice may warn “DESCEND! DESCEND!” if TCAS believes that an aircraft above poses a threat and there is room to maneuver below. TCAS makes sophisticated decisions in real time and is clearly safety-critical. On the one hand, it must detect as many potential collision events as possible within the limits of its sensors. On the other hand, it must generate as few false alarms as possible because the extreme maneuvers it recommends are themselves potentially dangerous.

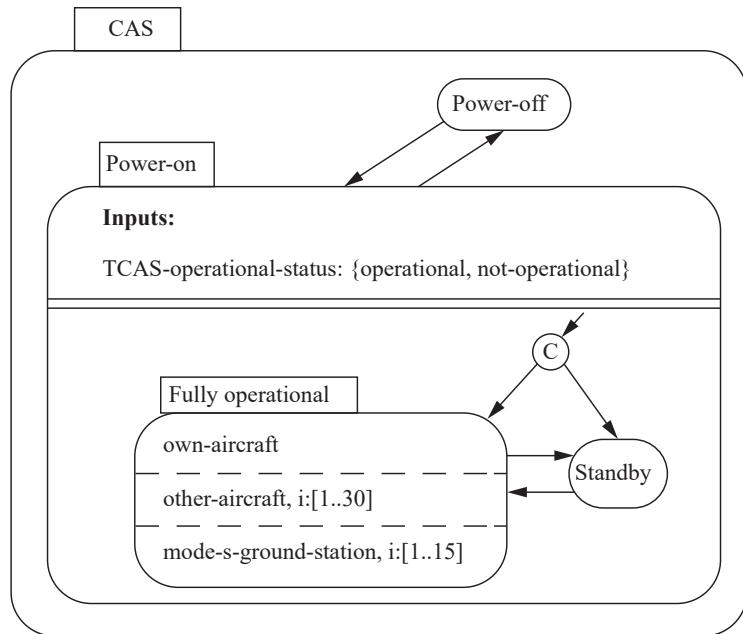
Leveson et al. [Lev94] developed a specification for the TCAS II system. We won’t cover the entire specification here, but just enough to provide its flavor. The TCAS II specification was written in root system markup language. They use a modified version of state chart notation for specifying states, in which the inputs to and outputs from the state are made explicit. The basic state notation looks like this:



They also use a transition bus to show sets of states in which there are transitions between all (or almost all) states. In this example, there are transitions from a, b, c, or d to any of the other states:



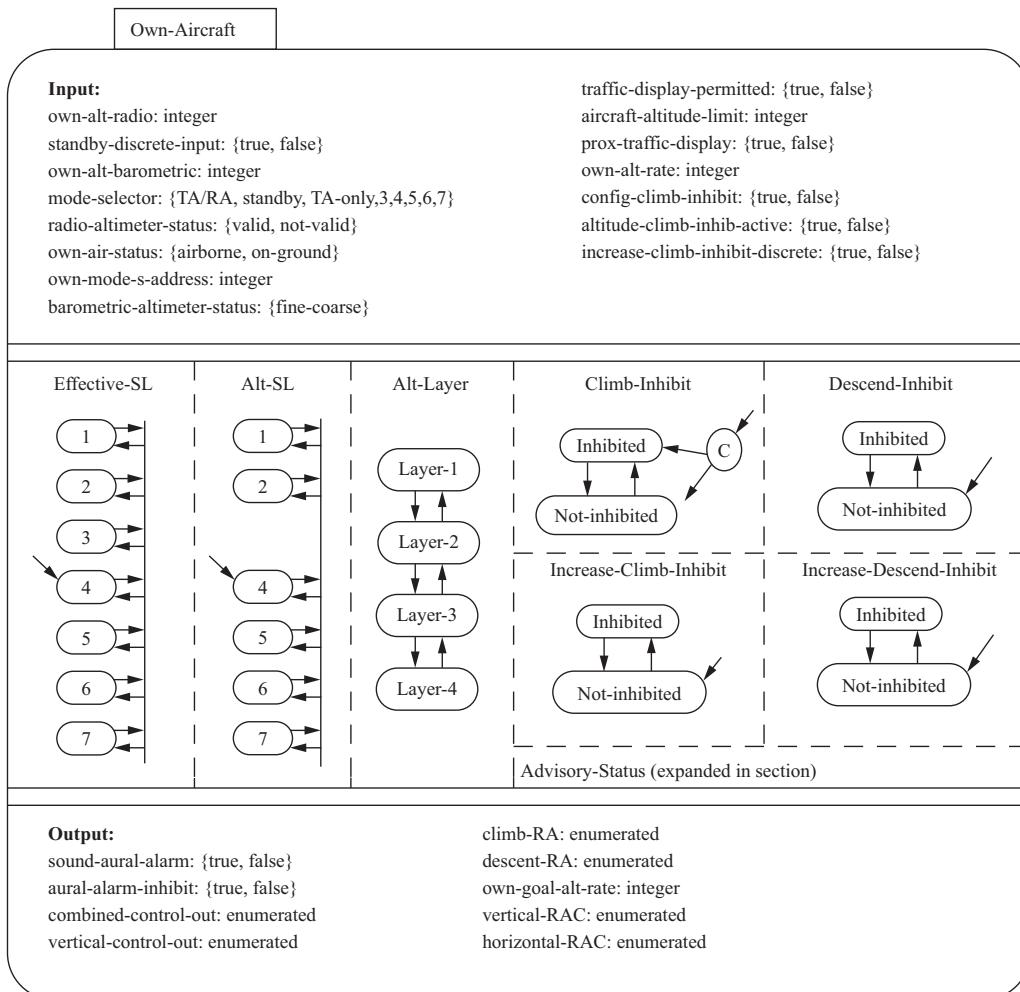
The top-level description of the collision avoidance system is relatively simple:



This diagram specifies that the system has *Power-off* and *Power-on* states. In the *Power-on* state, the system may be in *Standby* or *Fully operational* mode. In the *Fully operational* mode, three components operate in parallel, as specified by the AND state: the *own-aircraft* subsystem, a subsystem to keep track of up to 30 other aircraft, and a subsystem to keep track of up to 15 Mode S ground stations, which provide radar information.

The next diagram shows a specification of the *Own-Aircraft* AND state. Once again, the behavior of *Own-Aircraft* is an AND composition of several subbehaviors. The *Effective-SL* and *Alt-SL* states reflect two ways to control the sensitivity level (SL) of the system, with each state representing a different sensitivity level. Differing sensitivities are required depending on the distance from the ground and other factors. The *Alt-Layer* state divides the vertical airspace into layers, with this state keeping track of the current layer. *Climb-Inhibit* and *Descent-Inhibit* states are used to selectively inhibit climbs) which may be difficult at high

altitudes) or descents (clearly dangerous near the ground), respectively. Similarly, the *Increase-Climb-Inhibit* and *Increase-Descend-Inhibit* states can inhibit high-rate climbs and descents. Because the *Advisory-Status* state is complicated, its details are not shown here.



### 7.4.2 UML dialects for modeling

Several UML profiles have been defined for use in system modeling and real-time embedded computing system design. These dialects provide standard ways to talk

**SysML**

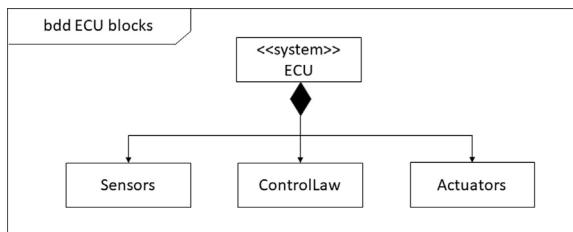
about systems and embedded concepts. We will use a simple engine control unit to illustrate some characteristics of these dialects.

The **Systems Modeling Language (SysML)** [OMG17] is based on UML and is designed as a broad-spectrum systems engineering language. SysML defines nine types of diagrams, some adopted from the base UML definition and others newly defined. Frames are optional parts of UML diagrams, but are required by SysML; each SysML diagram has a defined abbreviation. The sequence (**sd**), state machine (**STM**), package (**pkg**), and use case (**uc**) diagrams are all used in a manner consistent with UML.

SysML supports two types of block diagrams: the **block definition diagram (bdd)** defines the types of blocks used in the structure, and the **internal block diagram (ibd)** shows the connections between blocks. Fig. 7.4 shows a simple bbd for the engine control unit with three blocks or subsystems: engine sensors, the control law computation block, and engine actuators. Fig. 7.5 shows the associated ibd.

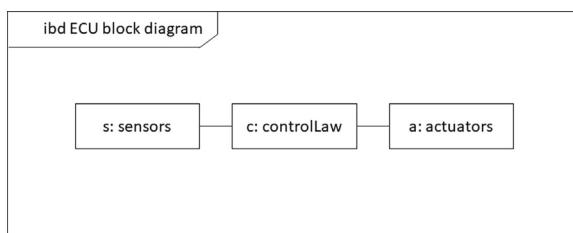
The **activity diagram (act)** combines object flows with object data flows. It is used to describe the behavior of a block. Fig. 7.6 shows a simple act. Control flow is described using state machine constructs, while an object for the engine temperature value shows the flow of information.

The **requirement diagram (req)** and **parametric diagram (par)** are new to SysML. Fig. 7.7 shows a simple req: one requirement is functional, and the other is



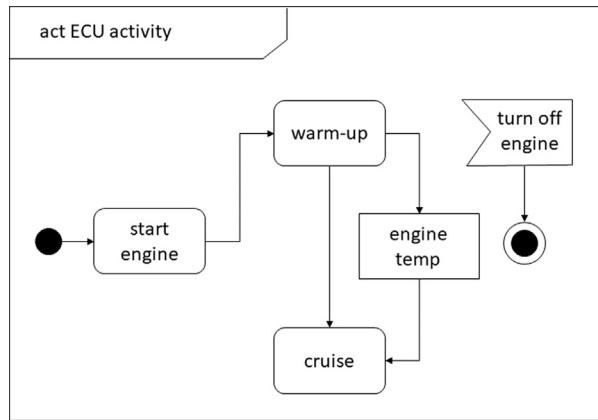
**FIGURE 7.4**

A SysML block definition diagram.

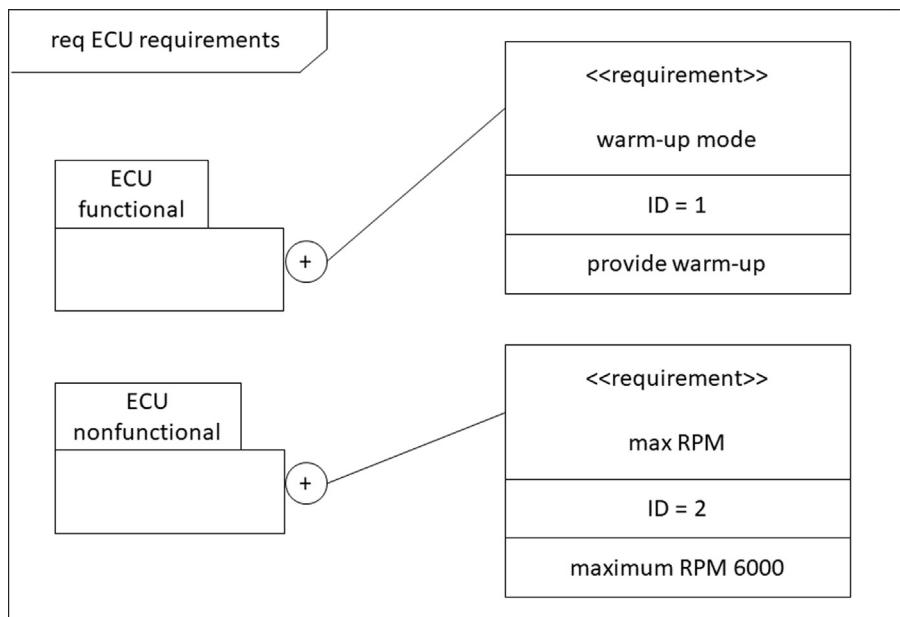


**FIGURE 7.5**

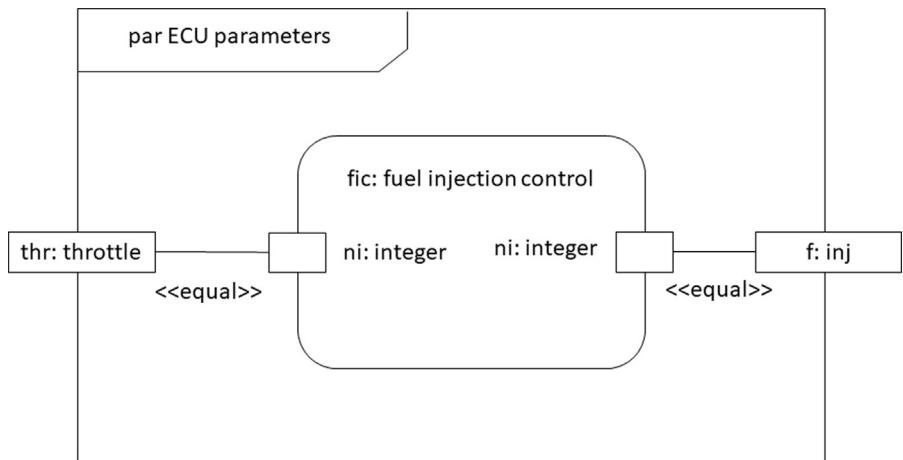
A SysML internal block diagram.

**FIGURE 7.6**

A SysML activity diagram.

**FIGURE 7.7**

A SysML requirement diagram.

**FIGURE 7.8**

A SysML parametric diagram.

nonfunctional. A **par** is shown in Fig. 7.8. A block is used to define the relationship between the parameters that are represented as inputs and outputs.

**MARTE**

SysML also supports tables to describe the relationships between elements.

MARTE [OMG19] stands for *modeling and analysis of real-time and embedded systems*. It provides several types of diagrams specific to that design domain.

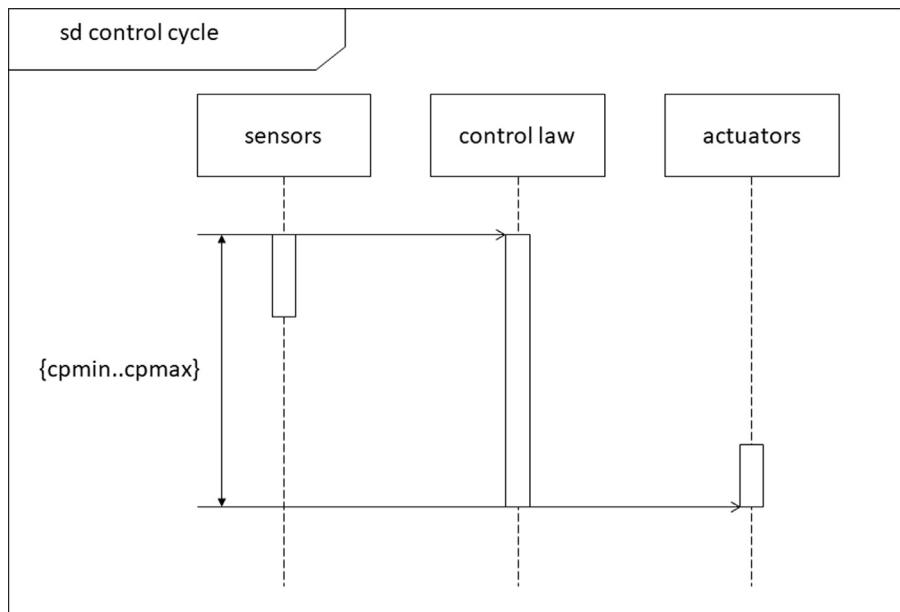
The **Non-Functional Properties Modeling (NFP)** module is used to capture the nonfunctional properties of the system. The **Time Modeling (Time)** module models both chronometric and sequential time. Fig. 7.9 shows a simple sequence diagram with an MARTE timing constraint. The constraint is given as a range of values. Two types of time can be described: an untimed causal model, a partially timed synchronous model, and a real-time physical model.

MARTE also supports the description of allocation decisions.

The Allocation Modeling (Alloc) module supports the allocation of functions and operations to resources. Fig. 7.10 shows an object diagram with allocation constraints at two levels: from the functional blocks to the RTOS abstraction level and from there to the computing platform.

The **High-Level Application Modeling (HLAM)** profile provides for the description of both quantitative and qualitative features. The **Detailed Resource Modeling (DRM)** profile provides both **software resource modeling (SRM)** and **hardware resource modeling (HRM)**.

The **Generic Quantitative Analysis Modeling (GQAM)** profile describes two types of evaluation: schedulability analysis for software task sets, and performance analysis, typically of a statistical nature. The MARTE schedulability analysis profile supports the analysis of both temporal constraints and schedulability, as well as sensitivity analysis of timing to system design parameters. Schedulability can describe a

**FIGURE 7.9**

Sequence diagram with MARTE timing constraint.

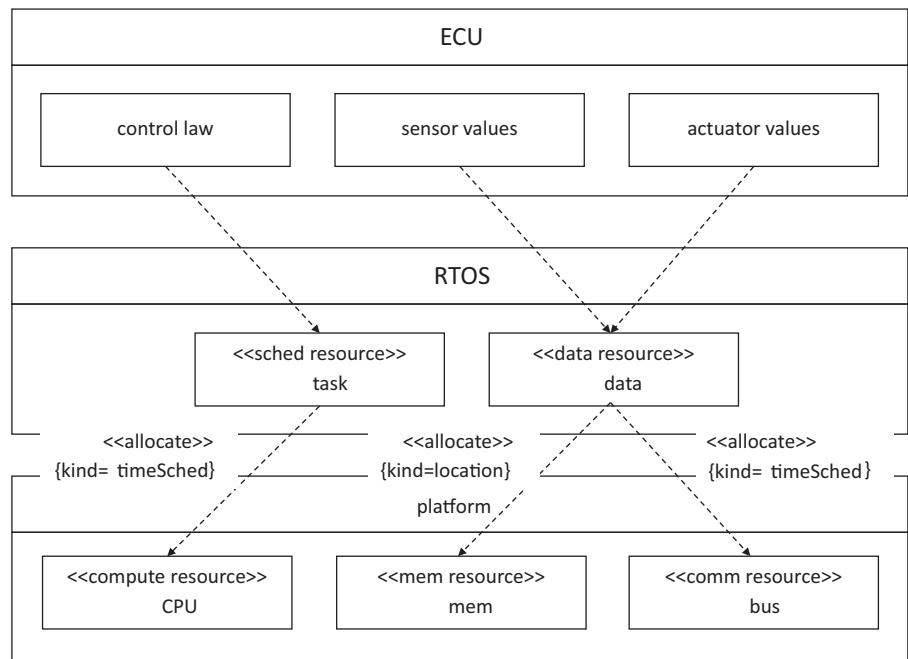
workload, timed observers to capture constraints and timing results, and resources, such as RTOS schedulers. The **Performance Analysis Modeling (PAM)**\_profile is used to describe best-effort and soft real-time systems.

## 7.5 System analysis and architecture design

In this section, we consider how to turn a specification into an architectural design. We already have several techniques for making specific decisions. Here, we look at how to get a handle on the overall system architecture: [Section 7.5.1](#) looks at common design patterns for embedded computing systems, and [Section 7.5.2](#) considers the transaction-level modeling of architectures.

### 7.5.1 Design patterns

A **design pattern** is a solution to a given type of recurring engineering problem. They capture best practices and help us identify structure in our designs. A design pattern for an embedded architecture might consider several elements: computing platform, allocation of operations, schedules for computations, and data movement.

**FIGURE 7.10**

Object diagram MARTE allocation annotations.

The microcontroller provides one of the most basic design patterns in embedded computing. A single processor connects to memory and I/O devices over a bus. Threads run under the control of an RTOS. The single-chip nature of a microcontroller simplifies many design decisions: we choose the entire configuration of CPU, memory, and I/O as a unit; on-chip communication is generally less costly in time and energy than off-chip communication.

A heterogeneous multiprocessor system-on-chip (MPSoC) provides more sophisticated variation. The MPSoC includes several processors, some of which may be accelerators with limited programmability. The chip may include several internal communication networks.

A **networked control system** typically refers to a single-bus system. It shares the same structure as a microcontroller (CPU, memory, I/O, bus), but these components are embedded across multiple chips and may be spread over a significant physical space. The schedule of data movements in the network is an important aspect of the design process.

Several other design patterns for networked control have more complex topologies. For example, a unit topology places the processing for a particular operation within its physical housing. A functional architecture groups the processing of several related functions. A shared architecture combines several different functions into a

processing unit. A zone architecture groups computations based on their physical location. We explore these architectures and their uses in automobiles and airplanes in [Chapter 9](#).

### 7.5.2 Transaction-level modeling

Refining a specification into an architecture requires the consideration of several factors: functionality, real-time performance, power consumption, and so on. Intermediate levels of abstraction help us manage the design process and competing constraints. A traditional level of abstraction in hardware design is **register transfer design**, a clock cycle-accurate model. We used threads as a model for software design. An intermediate model is a **transaction-level model** [Cai03]. A **transaction** is a communication between concurrent entities (either software threads or hardware units). Transactions can be modeled at several levels of detail, providing a path for design refinement:

- Untimed models (causal in MARTE terminology) preserve the partial ordering of transactions but do not provide information about execution time. Untimed models support functional debugging.
- Bus-accurate models (synchronous in MARTE terminology) provide refined timing models of the transactions, typically an estimate of the bus transaction determined from the bus specifications and estimates of software execution time.
- Cycle-accurate models (also synchronous in MARTE terminology) model communication, software, and hardware to clock cycle accuracy.

SystemC [IEE12] is a C++ library that supports the simulation and synthesis of concurrent processes, such as embedded software and hardware. The OCCN framework [Cop04] is a modeling and simulation environment built on top of SystemC.

---

## 7.6 Dependability, safety, and security

The quality of a product or service can be judged by how well it satisfies its intended function. A product can be of low quality for several reasons, such as if it were manufactured shoddily. Its components may have been improperly designed, its architecture was poorly conceived, and the product's requirements were poorly understood.

When we evaluate modern embedded computing systems, the traditional notions of quality, which are shaped in large part by consumer satisfaction, are no longer sufficient. **Dependability**, **safety**, and **security** are all vital aspects of a satisfactory system. These concepts are related but distinct. Dependability is a quantitative term that measures the length of time a system can operate without defects [Sie98]. Safety and security can be measured using dependability metrics. All terms are related to the general concept of quality.

The software testing techniques described in [Section 5.10](#) constitute one way to help meet quality-related goals. However, the pursuit of dependability, security, and safety must extend throughout the design flow. For example, settling on the proper requirements and specifications is an important determinant of quality. If the system is too difficult to design, it will probably be difficult to keep it working properly. Customers may desire features that sound nice, but, in fact, don't add much to the overall usefulness of the system. In many cases, having too many features only makes the design more vulnerable to both design and implementation errors, as well as attacks from hostile sources.

In this section, we review these related concepts in more detail. We start with a discussion of quality assurance processes. [Section 7.6.2](#) discusses verifying requirements and specifications. [Section 7.6.3](#) introduces design reviews as a method for quality management. [Section 7.6.4](#) introduces several safety-oriented methodologies. [Section 7.6.5](#) considers methodologies for safety-critical systems.

### 7.6.1 Quality assurance techniques

**Quality assurance** refers to both informal and more rigorous forms of product reliability, safety, security, and other aspects of fitness for use. The International Standards Organization (ISO) has created a set of quality standards known as **ISO 9000**. ISO 9000 was created to apply to a broad range of industries, including but not limited to embedded hardware and software. A standard developed for a particular product, such as wooden construction beams, could specify criteria particular to that product, such as the load that a beam must be able to carry. However, a wide-ranging standard, such as ISO 9000, cannot specify the detailed standards for every industry. Consequently, ISO 9000 concentrates on the processes used to create the product or service. The processes used to satisfy ISO 9000 affect the entire organization, as well as the individual steps taken during design and manufacturing.

A detailed description of ISO 9000 is beyond the scope of this book; several books [Sch94, Jen95] describe ISO 9000's applicability to software development. We can, however, make the following observations about quality management based on ISO 9000:

- *Process is crucial.* Haphazard development leads to low-quality haphazard products. Knowing what steps are to be followed to create a high-quality product is essential to ensuring that all the necessary steps are followed.
- *Documentation is important.* Documentation has several roles: The creation of the documents describing processes helps those involved understand the processes; documentation helps internal quality monitoring groups to ensure that the required processes are being followed; and documentation helps outside groups (e.g., customers, auditors, and so on) understand the processes and how they are being implemented.
- *Communication is important.* Quality ultimately relies on people. Good documentation helps people understand the total quality process. The people in the

organization should understand not only their specific tasks but also how their jobs can affect overall system quality.

Many types of techniques can be used to verify system designs and ensure quality. Techniques can be either *manual* or *tool-based*. Manual techniques are surprisingly effective in practice. In [Section 7.6.4](#), we discuss *design reviews*, which are simply meetings at which the design is discussed; such meetings are prosperous in identifying bugs. Many software testing techniques described in [Chapter 5](#) can be applied manually by tracing through the program to determine the required tests. Tool-based verification helps considerably in managing large quantities of information that may be generated in a complex design. Test generation programs can automate much of the drudgery of creating test sets for programs. Tracking tools can help ensure that various steps have been performed. Design flow tools automate the process of running design data through other tools.

Metrics are important to the quality control process. To determine whether we have achieved high levels of quality, we must be able to measure aspects of the system and our design process. We can measure certain aspects of the system itself, such as the execution speed of programs or the coverage of test patterns. We can also measure aspects of the design process, such as the rate at which bugs are found.

Tool-driven and manual techniques must fit into an overall process. The details of that process will be determined by several factors, including the type of product being designed (e.g., video game, laser printer, or air traffic control system), the number of units to be manufactured, the time allowed for design, the existing practices in the company into which any new processes must be integrated, and many other factors. An important role of ISO 9000 is to help organizations study their total processes, not just particular segments that may appear to be important at a particular time.

One well-known way of measuring the quality of an organization's software development process is the **Capability Maturity Model (CMM)**, developed by Carnegie Mellon University's Software Engineering Institute [SEI99]. The CMM provides a model for judging an organization. It defines the following five levels of maturity:

1. *Initial*. A poorly organized process with limited well-defined processes. The success of a project depends on the efforts of individuals, not the organization itself.
2. *Repeatable*. This level provides basic tracking mechanisms that allow management to understand cost, scheduling, and how well the systems under development meet their goals.
3. *Defined*. The management and engineering processes are documented and standardized. All projects make use of documented and approved standard methods.
4. *Managed*. This phase makes detailed measurements of the development process and product quality.
5. *Optimizing*. At the highest level, feedback from detailed measurements is used to continually improve an organization's processes.

The Software Engineering Institute has found limited organizations anywhere in the world that meet the highest level of continuous improvement and quite a few organizations that operate under the chaotic processes of the initial level. However, the CMM provides a benchmark by which organizations can judge themselves and use that information for improvement.

### 7.6.2 Design reviews

The **design review** [Fag76] is a critical component of any quality assurance process. The design review is a simple, low-cost way to catch bugs early in the design process. A design review is simply a meeting in which team members discuss a design and review how a component of the system works. Some bugs are caught simply by preparing for the meeting, as the designer is forced to think through the design in detail. Other bugs are caught by people attending the meeting who will notice problems that may not be caught by the unit's designer. By catching bugs early and not allowing them to propagate into the implementation, we reduce the time required to get a working system. We can also use the design review to improve the quality of the implementation and make future changes easier to implement.

#### Design review format

A design review is held to review a particular component of the system. A design review team has several types of members:

- The *designers* of the component being reviewed are, of course, central to the design process. They present their design to the rest of the team for review and analysis.
- The *review leader* coordinates the pre-meeting activities, the design review, and the post-meeting follow-up.
- The *review scribe* records the minutes of the meeting so that designers and others know which problems need to be fixed.
- The *review audience* studies the component. Audience members will naturally include other members of the project for which this component is being designed. Audience members from other projects often add valuable perspectives and may notice problems that team members have missed.

The design review process begins before the meeting itself. The design team prepares a set of documents (e.g., code listings, flowcharts, and specifications) that will be used to describe each component. These documents are distributed to other members of the team in advance of the meeting so that everyone has time to become familiar with the material. The review leader coordinates the meeting time, the distribution of handouts, and so forth.

During the meeting, the leader handles ensuring that the meeting runs smoothly while the scribe takes notes about what happens. The designers are responsible for presenting the component design. A top-down presentation often works well, beginning with the requirements and interface description, followed by the overall structure

of the component, the details, and then the testing strategy. The audience should look for all types of problems at every level of detail:

- Is the design team's view of the component's specification consistent with the overall system specification, or has the team misinterpreted something?
- Is the interface specification correct?
- Does the component's internal architecture work well?
- Are there coding errors in the component?
- Is the testing strategy adequate?

The notes taken by the scribe are used in the meeting follow-up. The design team should correct bugs and address the concerns raised at the meeting. While doing so, the team should keep notes describing what they did. The design review leader coordinates with the design team both to make sure that the changes are made and to distribute the change results to the audience. If the changes are straightforward, then a written report is adequate. If the errors found during the review caused major reworking of the component, a new design review meeting for the new implementation using as many of the original team members as possible may be useful.

### 7.6.3 Safety-oriented methodologies

We start with an example that describes the serious safety problems of one computer-controlled medical system. Medical equipment, like aviation electronics, is a safety-critical application; unfortunately, this medical equipment caused deaths before its design errors were properly understood. This example also allows us to use specification techniques to understand software design problems.

---

#### Example 7.6: The Therac-25 Medical Imaging System

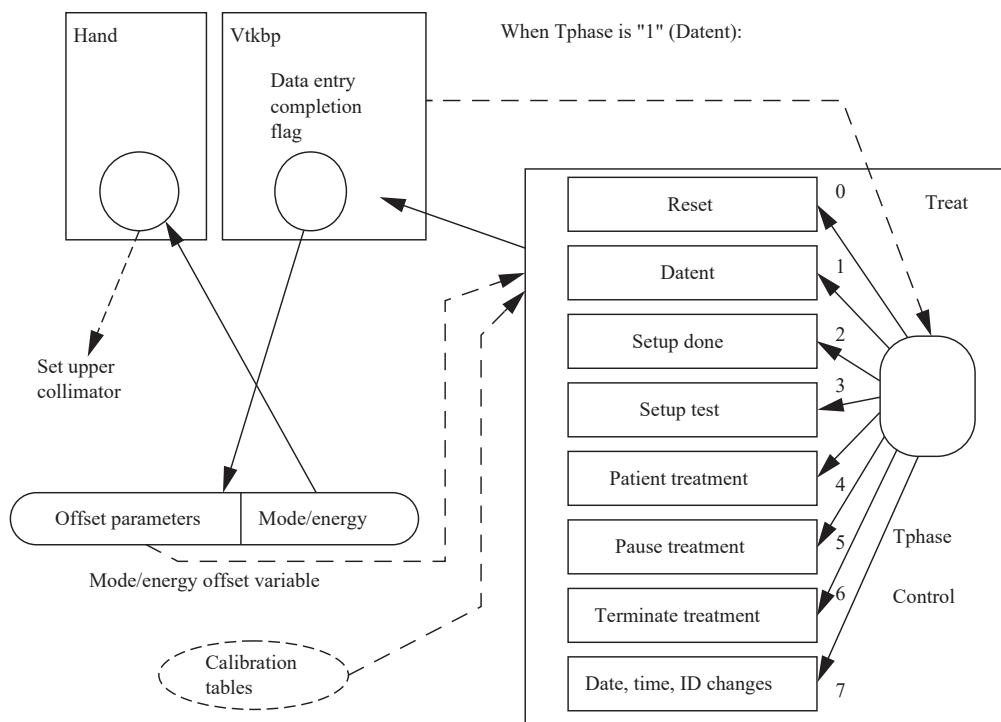
The Therac-25 medical imaging system caused what Leveson and Turner called the most serious computer-related accidents to date (at least nonmilitary and admitted) [Lev93]. During six known accidents, these machines delivered massive radiation overdoses, causing deaths and serious injuries. Leveson and Turner analyzed the Therac-25 system and the causes of these accidents.

Therac-25 was controlled by a programmed data processor (PDP)-11 minicomputer. The computer was responsible for controlling a radiation gun that delivered a dose of radiation to the patient. It also runs a terminal that presents the main user interface. The machine's software was developed by a single programmer in PDP-11 assembly language over several years. The software includes four major components: stored data, a scheduler, a set of tasks, and interrupt services. The three major critical tasks in the system are as follows:

- A treatment monitor controls and monitors the setup and delivery of the treatment in eight phases.
- A servo task controls the radiation gun, machine motions, and so on.
- A housekeeper task takes care of system status interlocks and limit checks. A limit check determines whether some system parameters have gone beyond preset limits.

The code was relatively crude: the software allowed several processes access to shared memory, there was no synchronization mechanism aside from shared variables, and the test-and-set for shared variables were not indivisible operations.

Let's examine the software problems responsible for one series of accidents. Leveson and Turner reverse-engineered a specification for the relevant software as follows:



- *Treat* is a treatment monitor task that is divided into eight subroutines (*Reset*, *Datent*, and others).
- *Tphase* is a variable that controls which of these subroutines are currently executing. *Treat* reschedules itself after the execution of each subroutine.
- The *Datent* subroutine communicates with the keyboard entry task via the data entry completion flag, which is a shared variable. *Datent* looks at this flag to determine when it should leave the data entry mode and go to the *Setup test* mode.
- The *Mode/energy offset* variable is a shared variable: The top byte holds offset parameters used by the *Datent* subroutine, and the low-order byte holds mode and energy offset used by the *Hand* task.

When the machine is run, the operator is forced to enter the mode and energy (there is one mode in which the energy is set to a default). However, the operator can later edit the mode and energy separately. The software's behavior is timing-dependent. If the keyboard handler sets the completion variable before the operator changes the *Mode/energy* data, the *Datent* task will not detect the change. Once *Treat* leaves *Datent*, it will not enter that subroutine again during treatment. However, the *Hand* task, which runs concurrently, will see the new *Mode/energy* information. Apparently, the software included no checks to detect incompatible data.

After the *Mode/energy* data are set, the software sends parameters to a digital/analog converter and then calls a *Magnet* subroutine to set the bending magnets. Setting the magnets takes about 8 s, and a subroutine, *Ptime*, is used to introduce a time delay. Owing to the way that *Datent*, *Magnet*, and *Ptime* are written, it is possible that changes to the parameters made by the user can be shown on the screen but will not be sensed by *Datent*. One accident occurred when the operator initially entered *Mode/energy*, went to the command line, changed *Mode/energy*, and returned to the command line within 8 s. The error therefore depended on the typing speed of the operator. Because operators become faster and more skillful with the machine over time, this error is likelier to occur with experienced operators.

Leveson and Turner emphasized that several poor design methodologies and flawed architectures were at the root of the bugs that led to the accidents:

- The designers performed a limited safety analysis. For example, low probabilities were assigned to certain errors with no apparent justification.
- Mechanical backups were not used to check the operation of the machine (e.g., testing beam energy), although such backups were employed in earlier models of the machine.
- Programmers created overly complex programs based on unreliable coding styles.

In summary, the designers of Therac-25 relied on system testing with insufficient module testing or formal analysis.

---

Several methodologies have been developed to specifically address the design of safety-critical systems. Some of these methodologies have been codified in standards that have been adopted by various industrial organizations. Safety-oriented methodologies supplement other aspects of methodology to apply techniques that have been demonstrated to improve the safety of the resulting systems.

#### Availability

Because dependability often depends on hidden characteristics that can only be modeled stochastically, **availability** is the term for the probability of a system's correct operation over time. A detailed discussion of dependability is beyond our scope here, but a common stochastic model for reliability is an exponential distribution:

$$R(t) = e^{-\lambda t} \quad (\text{Eq. 7.1})$$

The parameter,  $\lambda$ , is the number of failures per unit time.

#### Safety analysis phases

- **Hazard analysis** determines the types of safety-related problems that may occur.
- **Risk assessment** analyzes the effects of hazards, such as the severity or likelihood of injury.
- **Risk mitigation** modifies the design to improve the system's response to identified hazards.

Several standards have been developed for the design of safety-critical systems. Many of these standards focus on applications, such as aviation or automotive. Standards may also cover different levels of abstraction, with some covering earlier phases of design, while others concentrate on coding.

#### ISO 26262

ISO 26262 [ISO18A] is a standard that governs functional safety management for automotive electrics and electronics. The standard describes how to assess hazards, identify methods to reduce risks, and track safety requirements through to the

delivered product. Hazard analysis and risk assessment result in an Automotive Safety Integrity Level (ASIL) [ISO18C]. Events are assigned a severity classification for the degree of injury expected, an exposure classification describing how likely a person is to be exposed to the event, and a controllability classification for how well people can react to control the hazard.

**DO-178C**

DO-178C [RTC11] recommends safety-related procedures for airborne software. Hazard analysis results in the assignment of each failure condition to a **software level**: A catastrophic, B hazardous, C major, D minor, or E no effect. The standard requires that the generated safety-related requirements be traceable in software implementation and testing.

**SAE standards**

The Society of Automotive Engineers (SAE) has several standards for automotive software: J2632 for coding practices for C code, J2516 for software development lifecycle, J2640 for software design requirements, and J2734 for software verification and validation.

**Coding standards**

Several coding standards have been developed to specify specific ways of writing software that increase software reliability. Coding standards inevitably target specific programming languages, but a consideration of some specific examples of coding standards helps us understand the role they play in reliability and quality assurance.

**MISRA**

The Motor Industry Software Reliability Association (MISRA) formulated a series of standards for software coding in automotive and other critical systems. MISRA C:2012 [MIS13] is an updated version of the coding standards for the C programming language while MISRA C++ [MIS08] is a set of standards for C++. Both standards provide directives and rules for how programs in these languages are to be written. They also place these guidelines in the context of overall development methodology. The guidelines are to be used as part of a documented software process.

The MISRA C standard gives a set of general directives, some of which are general (e.g., traceability of code to requirements), and others are more specific (e.g., code should compile without errors). It also provides a set of more detailed rules. For example, a project should not contain unreachable or dead code. (Unreachable code can never be executed, whereas dead code can be executed, but has no effect.) As another example, all function types are required to be in prototype form. Early versions of C did not require the program to declare the types of a function's arguments or its return. Later versions of C created *function prototypes* that included type information. This rule requires that the prototype form always be used.

These standards are intended to be enforced with the help of tools. Relying on manual methods, such as design reviews, to enforce a large number of very detailed rules in these standards would be unwieldy. Commercial tools have been developed that specifically check for these rules and generate reports that can be used to document the design process.

**CERT C and 17961**

CERT C [Sea14] is a standard for coding in C-language programs. It does not directly target embedded computing. Its rules can be divided into 14 categories, including topics such as memory management, expressions, integers, floating point,

arrays, strings, and error handling. ISO/IEC TS 17961 [ISO13] is a standard for secure coding in C.

#### 7.6.4 Security

**Security** Safety-critical designs must also be secure. Devices with user accounts and Internet access provide obvious avenues for attacks. However, attacks may also come from more indirect sources.

**Air gaps** The **air gap myth** is a continuing source of vulnerabilities. The notion that we can build a system with an *air gap* (no direct Internet connection) is naïve and unrealistic in modern embedded computing systems. A variety of devices and techniques can be used to carry infections onto embedded processors.

The next example discusses the first cyber-physical attack, Stuxnet. The attacked system was air-gapped, but was still successfully attacked.

---

#### Example 7.7: Stuxnet

The name “Stuxnet” was given to a series of attacks on Iranian nuclear processing facilities [McD13, Fal10, Fal11]. The facilities were not directly connected to the Internet, but were air-gapped. However, the facility’s computers were infected by workers who used USB devices that had been infected while connected to outside machines; those workers were likely to use software tools carried on USB memory devices to conduct standard software operations.

The attacks targeted a particular type of programmable logic controller (PLC) used to control centrifuges that were part of the nuclear processing equipment. The PLCs were programmed using PCs. A PC infected with Stuxnet executed two dynamically linked libraries to attack the PLC software: one that identified PLC code for attack (a process known as **fingerprinting**) and another that modified the PLC’s programming.

The PLC software was modified to improperly operate the centrifuges. The attack code had some knowledge of the structure of the nuclear processing equipment and which centrifuges to attack. Stuxnet used **replay** to hide its attacks: it first recorded the centrifuge’s output during nonfaulty behavior, and then replayed that behavior while it maliciously operated the centrifuge. In addition, Stuxnet modified the PLC software to hide the existence of PLC modifications.

These attacks caused extensive damage to the nuclear processing facilities and seriously compromised their ability to operate.

---

Graff and van Wyck [Gra03] recommended several methodological steps to help improve the security of a program. Assessing threats and the risks posed by those threats is an important early step in program design. The **attack surface** of a program is the set of program locations and use cases in which it can be attacked. Some applications have naturally small attack surfaces, whereas others inherently expose much larger attack surfaces. As we will see in [Chapter 9](#), remote telematics interfaces to cars provide large attack surfaces. Once risks are identified, several methods can be used to mitigate them, such as avoiding using the application in certain circumstances, performing checks to limit risks, and so forth. Finding unusual metaphors for a program’s operations may help identify potential weaknesses.

---

## 7.7 Summary

System design takes a comprehensive view of the application and the system under design. To ensure that we design an acceptable system, we must understand the application and its requirements. Numerous techniques, such as object-oriented design, can be used to create useful architecture from the system's original requirements. Along the way, by measuring our design processes, we can gain a clearer understanding of where bugs are introduced, how to fix them, and how to avoid introducing them in the future.

---

## What we learned

- Design methodologies and flows can be organized in many different ways.
- A variety of methods can be used to elicit requirements.
- System modeling can capture both the functional and nonfunctional aspects of a system.
- Dependability, safety, and security are related attributes of embedded systems and concerns during design.

---

## Further reading

Pressman [Pre97] provides a thorough introduction to software engineering. Davis [Dav90] provides a good survey of software requirements. Beizer [Bei84] surveys system-level testing techniques. Leveson [Lev86] provides a good introduction to software safety. Schmauch [Sch94] and Jenner [Jen95] both described ISO 9000 for software development. A tutorial edited by Chow [Cho85] includes many important early papers on software quality assurance. Cusumano [Cus91] provides a fascinating account of software factories in both the United States and Japan.

---

## Questions

**Q7-1** Provide realistic examples of how a requirements document may be:

- a. ambiguous
- b. incorrect
- c. incomplete
- d. unverifiable

**Q7-2** How can poor specifications lead to poor quality code—do aspects of a poorly constructed specification necessarily lead to bad software?

**Q7-3** What are the main phases of a design review?

- Q7-4** What is an example dependability measure?
- Q7-5** What are example types of functional, safety-related bugs were found in Therac-25?
- Q7-6** What are example types of nonfunctional, safety-related bugs were found in Therac-25?
- Q7-7** What are example types of methodological, safety-related problems were found in Therac-25?
- Q7-8** How could a replay attack be used against an automobile?

---

### Lab exercises

- L7-1** Draw a diagram showing the developmental steps of one of the projects you recently designed.
- L7-2** Find a detailed description of a system of interest to you. Write your own description of what it does and how it works.

# Internet-of-Things Systems

# 8

## CHAPTER POINTS

---

- Internet-of-Things (IoT) = sensors + wireless networks + database.
  - Wireless networking for IoT devices.
  - Database design for IoT systems.
  - Design example: smart home.
- 

## 8.1 Introduction

The **Internet-of-Things** (IoT, or *Internet-of-Everything*) is a new name for a concept that has evolved over several decades. Mark Weiser of Xerox PARC coined the term *ubiquitous computing* to describe a range of smart and interconnected devices, as well as the applications to which these devices could be put. Moore's Law has allowed us to improve those early devices in several important ways: modern IoT devices provide more computing and storage; they operate at lower energy levels; and they are cheaper. Advances in wireless communications have also allowed these devices to communicate much more effectively with each other and with traditional communication systems. The result of these advances is an explosion in devices, systems, and applications.

This chapter describes the basic concepts underlying IoT system design. We start with a survey of applications that make use of IoT technologies in [Section 8.2](#), followed by a discussion of IoT system architectures in [Section 8.3](#). [Section 8.4](#) studies several networking technologies, a central concept for IoT system design. In [Section 8.5](#), we look at two types of data structures that can be used to organize information in IoT systems: databases and timewheels. We conclude with the example of an IoT smart home in [Section 8.6](#).

---

## 8.2 IoT system applications

An Internet-of-Things system is a soft real-time networked embedded computing system. An IoT system always includes input devices: tags, sensors, and so on.

One may also include output devices: motor controllers, electronic controllers, and displays. The devices comprise data processing devices (displays, buttons) and cyber-physical devices (temperature sensors, cameras).

#### Application examples

IoT systems can be used for several different purposes:

- A computer-readable identification code for a physical object can allow a computer system to keep track of an inventory of items.
- A complex device (e.g., an appliance) can be controlled via a user interface on a cell phone or computer.
- A set of sensors can monitor activity with data analysis algorithms extracting useful information from the sensor data. Sports sensor systems, for example, can monitor and analyze the activity of an athlete or team of athletes. Smart building systems can monitor and adjust the temperature and air quality of a building.

#### Devices

We can identify three types of devices for people: **implanted** devices are within the body; **wearable** devices are worn outside the body; and **environmental** sensors are located entirely off the body (e.g., mounted on the wall). These categories also fit well with the objects. We use one set of terms for both cases: **interior**, **exterior**, and **environmental**.

Various interior or implanted devices are used for people, including heart rate sensors, neurological sensors, and so forth. For objects, sensors play an equivalent role: engine temperature sensors, and more. A smart band is an example of an exterior sensor for people. Environmental sensors range across many modalities, including door sensors, cameras, and weather sensors.

#### Example 8.1: Wall-Mountable Camera

An **Internet Protocol (IP) camera** is a video camera that transmits digital video over an Internet connection. Older video cameras transmitted analog video over cables. These cameras often transmit video in H.264 format; they may also use motion JPEG, a sequence of still frames. Many cameras also provide a still-image mode. A **pan-tilt-zoom (PTZ)** camera can move horizontally (pan) and vertically (tilt), as well as zoom its lens in and out. Some cameras use a semi-spherical reflector to capture panoramic images.

#### RFID

**Radio-frequency identification (RFID)** is an important class of IoT devices. An **RFID tag** can be used to provide an identification number for a physical object and possibly for other information as well. Many RFID tags are read-only; they can be programmed before installation using a separate machine. However, in use, they only reply to a request by transmitting their preprogrammed identification tag. Some tags can be written in use under the control of radio transmission.

We can identify two common use cases for RFID tag communication. The first is known as **passive** because the tag transmits only when it receives a request. An **active** tag will respond to requests but will also transmit on its own periodically.

We also use **passive** to describe RFID tags that have no internal power source; they receive all their energy from the outside. A battery-assistive passive device uses

passive communications but makes use of a battery. An RFID tag can use its antenna to receive radio-frequency energy. It can store some energy in a capacitor and use that energy to operate the radio to receive and transmit data.

RFID tags were designed to operate in several different frequency bands and at several different distance ranges. Some tags operate only within a few centimeters, whereas others can be read from dozens of meters away.

Some RFID tags use the electronic product code (EPC) [GS114] as an identifier. An EPC can be used to assign a unique name to a physical object.

## 8.3 IoT system architectures

### Edge and cloud

We often divide an IoT system into **edge** and **cloud** components. An edge device is one of the system's application devices; data from the edge devices may be processed remotely at Internet servers referred to as "the cloud."

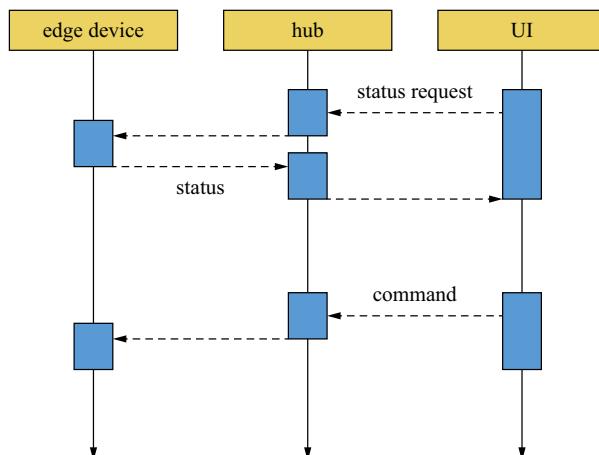
### IoT use cases

We can identify several formulas for IoT design, each with its own associated use cases.

The simplest design formula is a smart appliance:

$$\text{IoT smart appliance} = \text{connected appliance} + \text{network} + \text{UI}$$

In this case, a communication node allows the appliance to connect to a user interface (UI) through the network, giving the user access to the controls and status of the appliance. [Fig. 8.1](#) shows a UML sequence diagram for a smart appliance. In this scenario, a user interface runs on a device, such as a smart phone. The UI can interact



**FIGURE 8.1**

Use case for an IoT smart appliance.

with the smart appliance by sending and receiving messages via the hub. The UI can check the status of the smart appliance or give it commands.

A more sophisticated version applies to distributed sensor systems:

IoT monitoring system = sensors + network + database + dashboard

Examples of IoT monitoring systems include smart homes and buildings or connected cities. [Fig. 8.2](#) shows a sample use case. The sensors feed data into a database via their hubs. A data analysis program runs in the cloud to extract useful information from sensor data streams. Those results are given to the user on a **dashboard** that provides a summary of the system's status, important events, and so on.

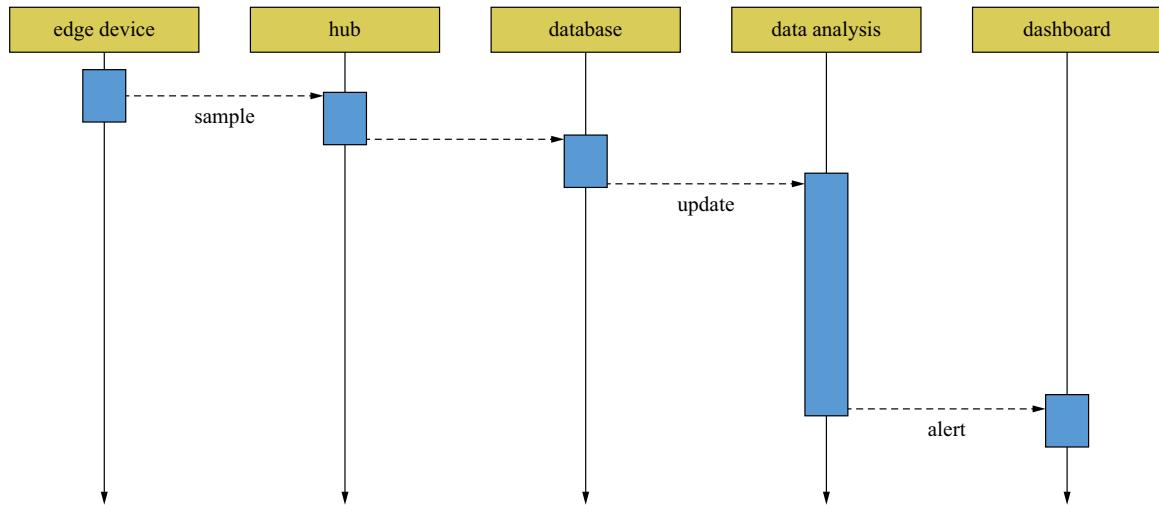
IoT networks can also be used for control, as illustrated in [Fig. 8.3](#).

IoT control system = sensors + network + database + controller + actuator

A wireless sensor network can make sensor measurements that are sent to a cloud-based controller, which then sends a command to an actuator in the edge network. The control algorithm may be a traditional periodic algorithm, for example, a motor controller. It can also be an event-driven controller, as is typical for home or building energy management.

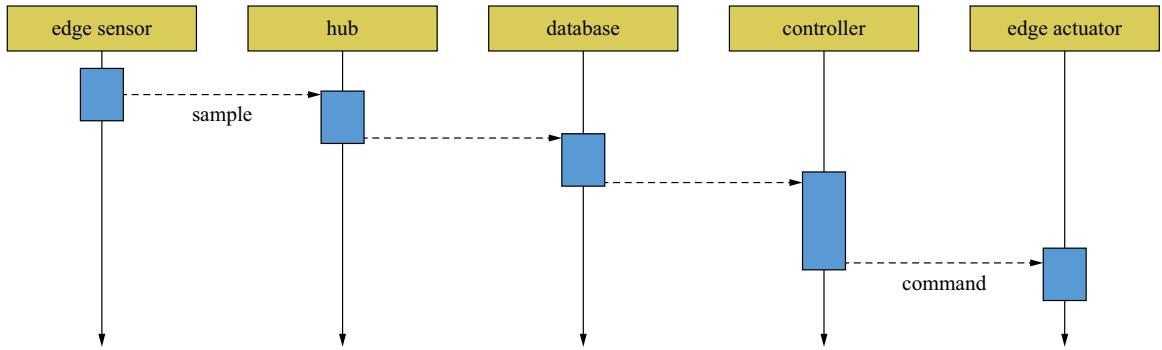
## 8.4 Networks for IoT

Networking is a critical component of an IoT system; wireless networking allows for a much wider range of sensor applications than is possible with wired networks. We



**FIGURE 8.2**

Use case for an IoT monitoring system.

**FIGURE 8.3**

Use case for an IoT control system.

start with a review of the fundamental concepts in networking: the **Open Systems Interconnection (OSI)** model and the Internet Protocol. We then describe some basic concepts in wireless networks for IoT, followed by the specifics of four widely used wireless networks: Bluetooth/Bluetooth Low Energy (BLE), IEEE 802.15.4/ZigBee, Wi-Fi, and LoRa.

### 8.4.1 The open systems interconnection model

Networks are complex systems. Ideally, they provide high-level services while hiding many of the details of data transmission from the other components in the system. To help understand (and design) networks, the International Standards Organization has developed a seven-layer model for networks known as OSI [Sta97A]. Understanding the OSI layers will help us understand the details of real networks.

The seven layers of the OSI model, as shown in Fig. 8.4, are intended to cover a broad spectrum of networks and their uses. Some networks may not need the services of one or more layers. Higher layers may be missing, or an intermediate layer may not be necessary. However, any data network should fit into the OSI model.

#### OSI layers

The OSI model includes seven levels of abstraction, known as **layers**:

- **Physical (PHY):** The PHY layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires), electrical properties, the basic functions of the electrical and physical components, and the basic procedures for exchanging bits.
- **Data link:** The primary purpose of this layer is error detection and control across a single link. However, if the network requires multiple hops over several data links, the data link layer does not define the mechanism for data integrity between hops but only within a single hop. The data link layer can be divided into two sublayers: **medium access control (MAC)** to control access permissions and the **logical link control (LLC)** layer, which encapsulates network protocols as well as managing error checking and frame synchronization.

Application	End-use interface
Presentation	Data format
Session	Application dialog control
Transport	Connections
Network	End-to-end service
Data link	Reliable data transport
Physical	Mechanical, electrical

**FIGURE 8.4**


---

The open systems interconnection model layers.

- **Network (NWK):** This layer defines the basic end-to-end data transmission service. The network layer is particularly important in multihop networks.
- **Transport:** The transport layer defines connection-oriented services that ensure that data are delivered in the proper order and without errors across multiple links. This layer may also try to optimize network resource utilization.
- **Session:** A session provides mechanisms for controlling the interaction of end-user services across a network, such as data grouping and checkpointing.
- **Presentation:** This layer defines data exchange formats and provides transformation utilities to application programs.
- **Application:** The application layer provides the application interface between the network and the end-user programs.

Although it may seem that embedded systems would be too simple to require the use of the OSI model, the model is in fact quite useful. Even relatively simple embedded networks provide physical, data links, and network services. An increasing number of embedded systems provide Internet services that require implementing the full range of functions in the OSI model.

### 8.4.2 IP

The Internet Protocol (IP) [Los97, Sta97A] is the fundamental protocol on the **Internet**. It provides connectionless, packet-based communication. Industrial automation has long been a good application area for Internet-based embedded systems. Information appliances that use the Internet are rapidly becoming another use of IP in embedded computing. The term *Internet* generally refers to the global network of

**Internetworking**

computers connected by the IP. However, it is possible to build an isolated network not connected to the global Internet that uses IP.

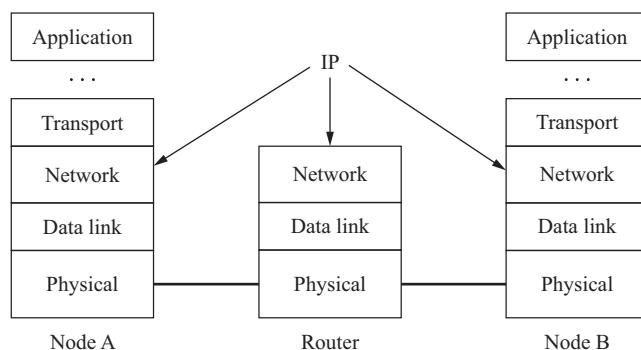
IP is not defined over a particular physical implementation; it is an **internetworking** standard. Internet packets are assumed to be carried by some other network, such as Ethernet. Generally, an Internet packet travels over several different networks from source to destination. The IP allows data to flow seamlessly through these networks from one end user to another. The relationship between IP and individual networks is illustrated in Fig. 8.5. IP works at the NWK layer. When node A wants to send data to node B, the application's data pass through several layers of the protocol stack to get to the Internet Protocol, which then creates packets for routing to the destination. These are then sent to the *data link* and *PHY* layers. A node that transmits data among different types of networks is known as a **router**. The router's functionality must go up to the IP layer, but because it does not run applications, it does not need to go to higher levels of the OSI model. In general, a packet may go through several routers to reach its destination. At the destination, the IP layer provides data to the transport layer and ultimately to the receiving application. As the data pass through several layers of the protocol stack, the IP packet data are encapsulated in packet formats appropriate to each layer.

**IP packets**

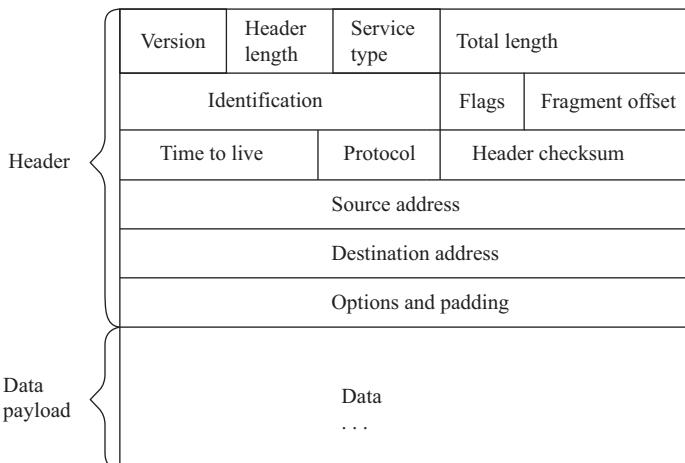
The basic format of an IP packet is shown in Fig. 8.6. The header and data payload are both variable in length. The maximum total length of the header and data payload is 65,535 bytes.

An Internet address is a number (32 bits in early versions of IP, 128 bits in IPv6). The IP address is typically written in the form xxx.xxx.xxx.xxx. The names by which users and applications typically refer to Internet nodes, such as [foo.baz.com](#), are translated into IP addresses via calls to a **Domain Name Server (DNS)**, one of the higher-level services built on top of IP.

The fact that IP works at the network layer tells us that it does not guarantee that a packet is delivered to its destination. Furthermore, packets that do arrive may come out of order. This is referred to as **best-effort routing**. Because routes for data may change quickly with subsequent packets being routed along different paths with

**FIGURE 8.5**

Protocol utilization in Internet communication.

**FIGURE 8.6**

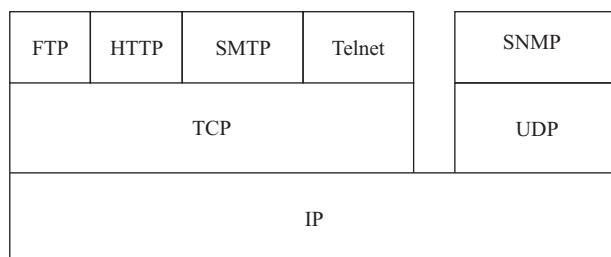
Internet protocol packet structure.

different delays, the real-time performance of IP can be hard to predict. When a small network is contained totally within the embedded system, performance can be evaluated through simulation or other methods because the possible inputs are limited. Because the performance of the Internet may depend on worldwide usage patterns, its real-time performance is inherently harder to predict.

#### IP services

The Internet also provides higher-level services built on top of IP. The **Transmission Control Protocol (TCP)** is one such example. It provides a connection-oriented service that ensures that data arrive in the appropriate order, and it uses an acknowledgment protocol to ensure that packets arrive. Because many higher-level services are built on top of TCP, the basic protocol is often referred to as TCP/IP.

[Fig. 8.7](#) shows the relationships between IP and higher-level Internet services. Using IP as the foundation, TCP is used to provide **File Transport Protocol (FTP)** for batch file transfers, Hypertext Transport Protocol (**HTTP**) for World Wide Web service, **Simple Mail Transfer Protocol (SMTP)** for email, and Telnet for virtual

**FIGURE 8.7**

Internet service stack.

terminals. A separate transport protocol, the **User Datagram Protocol (UDP)**, is used as the basis for the network management services provided by the **Simple Network Management Protocol (SNMP)**.

### 8.4.3 IoT networking concepts

Not everything is connected to the Internet. Although the Internet is a good match for computer systems, it is not always well suited to other types of devices. Many IoT devices communicate over non-IP networks, which are sometimes called **edge networks**. As shown in Fig. 8.8, devices can link to the Internet using a **gateway** that translates between the IoT network and the Internet.

#### Ad hoc networks

IoT networks differ from traditional Internet in that they do not need to be explicitly set up and managed. An **ad hoc network** is created by the self-organization of a set of nodes. The nodes route messages to each other; they do not rely on separate routers or other networking equipment.

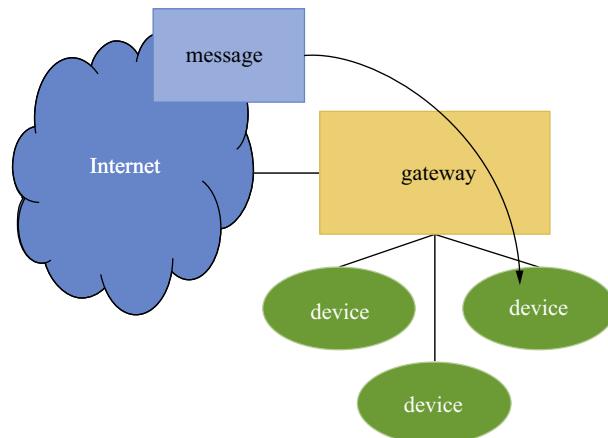
We can evaluate an IoT network based on both its functional and nonfunctional characteristics:

- Does it provide adequate security and privacy?
- How much energy is required for communication? Many IoT network devices are designed to operate from a button battery for an extended period. We refer to these networks as **ultra-low energy (ULE)**.
- How much does the network cost to add to a device?

#### Services

An ad hoc network should supply several services:

- **Authentication** determines whether a node is eligible to be connected to the network.



**FIGURE 8.8**

Gateway between the Internet and a personal area network.

- **Authorization** checks whether a given node should be able to access a piece of information on the network.
- **Encryption and decryption** help provide security.

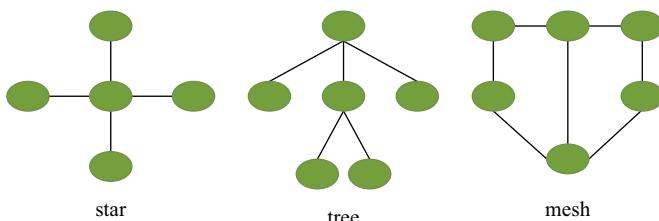
The **topology** of a network describes the structure of communication within the network. In the OSI model, a link is a direct connection between two nodes. Communication between two nodes that are not directly connected requires several hops. The topology of a wireless IoT network is influenced by several factors, including the range of radios and the complexity of the management of the topology. Fig. 8.9 shows several examples of IoT network topologies. The **star network** uses a central hub through which all other nodes communicate. A **tree network** provides a more complex structure, but still only provides one path between a pair of nodes. A **mesh network** is a general structure.

Once the network authenticates a node, it must perform housekeeping functions to incorporate a new node into the network. **Routing discovery** determines the routes that will be used by packets that travel to and from other nodes to the new node. Routing discovery starts by searching the network for paths to the destination node. A node will broadcast a message requesting routing discovery services and record the response it receives. The recipient nodes will then broadcast their own routing discovery request, with the process continuing until the destination node is reached. Once a set of routes has been identified, the network evaluates the cost of choosing one (or perhaps more than one) path. The cost computation for a path can include the number of hops, the transmission energy required, and the signal quality on each link.

Routing discovery produces a routing table for each node, as illustrated in Fig. 8.10. When a node wants to send a message to another node, it consults its routing table to determine the first node on the path. That node consults its own routing table to determine the next hop; the process continues until the packet reaches its destination.

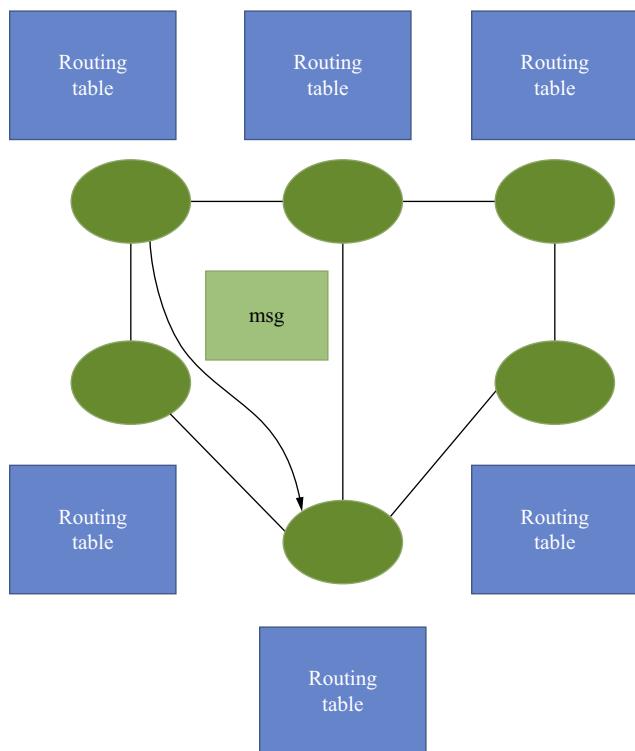
## QoS

Many IoT networks support **synchronous** and **asynchronous** communication. Synchronous communication is periodic, for example, voice or sampled data. We often use the term **quality-of-service (QoS)** to describe the bandwidth and periodicity characteristics of synchronous data. To provide synchronous data with QoS



**FIGURE 8.9**

Example of IoT network topologies.

**FIGURE 8.10**

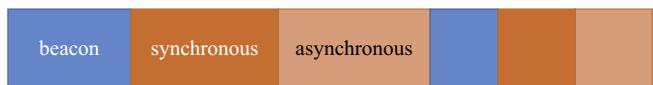
Routing packets through a network.

characteristics, the network needs to reserve bandwidth for that communication. Many networks perform **admission control** to process a request for synchronous transmission and to determine whether the network has the bandwidth available to support the request. A request may be rejected if, for example, too many existing synchronous communication streams do not leave sufficient bandwidth to support the requested connection.

#### Synchronization

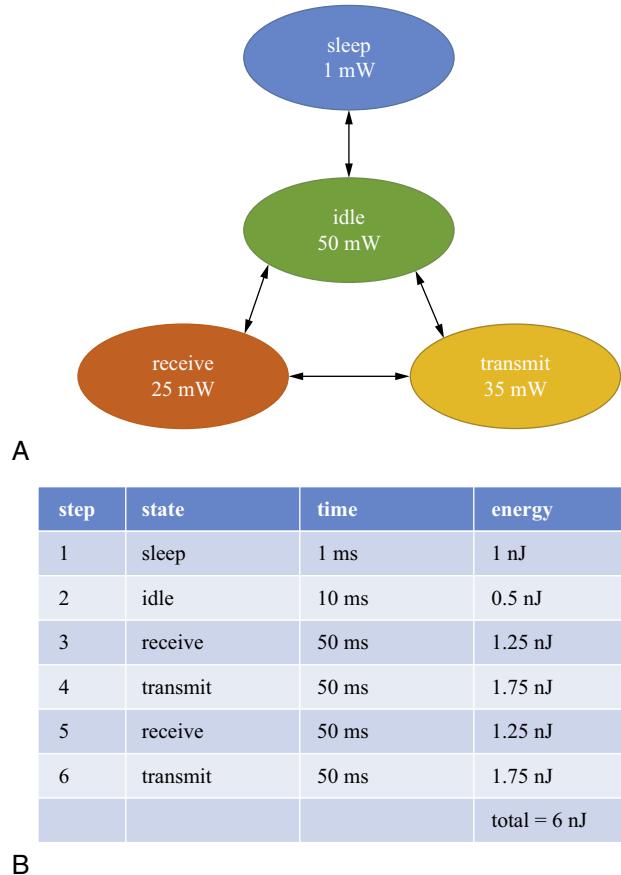
One challenge in synchronous communication over wireless networks is synchronizing the nodes. Many wireless networks provide synchronous communication using **beacons**. As illustrated in Fig. 8.11, the beacon is a transmission from a node that marks the beginning of a communication interval. The time between beacons is usually divided into two segments, one for synchronous packets and the other for asynchronous packets. A synchronous communication can be assigned its own time slot in the synchronous segment.

The energy required for communication is a key concern for wireless battery-operated networks. Energy requirements may be stated in two styles: either Joules or Watts. Charge is expressed in amp-hours. The amp-hour metric can be converted to energy using the power supply voltage.

**FIGURE 8.11**

Beacon transmissions.

Energy consumption is generally evaluated for particular use cases that consider idle time, transmission length, or other factors. We can use the power state machine in Section 3.7.2 to help us compute wireless energy consumption. A use case describes a path through a power state machine. As shown in Fig. 8.12, given the time spent in each state and the power consumption in those states, we can compute the energy consumption for a use case.

**FIGURE 8.12**

Example of radio energy consumption analysis. (a) Radio power state machine. (b) Use-case-based energy analysis.

#### 8.4.4 Bluetooth and Bluetooth Low Energy

**Bluetooth** was introduced in 1999, originally for telephony applications, such as wireless headsets for cell phones. It is now used to connect a wide range of devices to host systems. **Bluetooth Low Energy (BLE)** shares a name but has a quite different design.

##### Classic Bluetooth

**Classic Bluetooth** [Mil01], as the original standard has come to be known, is designed to operate in a radio band known as the **instrumentation, scientific, and medical (ISM) band**. The ISM band is in the 2.4-GHz frequency range and no license is required to operate in the ISM band throughout the world. There are, however, some restrictions on how it can be used, such as the 1 MHz bandwidth channels and frequency-hopping spread spectrum.

Bluetooth networks are often called **piconets** thanks to their small physical size. A piconet consists of a master and several slaves. A slave can be active or parked. A device can be a slave on more than one piconet.

The Bluetooth stack is divided into three groups: **transport protocol, middleware protocol, and application**.

The transport protocol group has several constituents:

- The radio provides physical data transport.
- The baseband layer defines the Bluetooth air interface.
- The link manager performs device pairing, encryption, and negotiation of link properties.
- The **LLC and adaptation protocol (L2CAP)** layer provides a simplified abstraction of transport for higher levels. It breaks large packets into Bluetooth packets. It negotiates the QoS required and performs admission control.

The middleware group has several members:

- The RFCOMM layer provides a serial port-style interface.
- The service discovery protocol (SDP) provides a directory for network services.
- IP and IP-oriented services, such as TCP and UDP.
- A variety of other protocols, such as IrDA for infrared and telephony controls.

The application group includes the various applications that use Bluetooth.

Every Bluetooth device is assigned a 48-bit Bluetooth device address. Every Bluetooth device also has its own Bluetooth clock that is used to synchronize the radios on a piconet, as required for frequency-hopping spread spectrum communication. When a Bluetooth device becomes a part of a piconet, it adjusts its operation to the clock of the master.

Transmissions on the network alternate between master and slave directions. The baseband supports two types of packets:

- **Synchronous connection-oriented (SCO)** packets are used for QoS-oriented traffic, such as voice and audio.

- **Asynchronous connectionless (ACL)** packets are used for non-QoS traffic.

SCO traffic has a higher priority than ACL traffic.

### Bluetooth Low Energy

BLE [Hey13] is, as the name implies, designed to support very low energy radio operation. A radio operated by a button-sized battery for an extended period is an example scenario of BLE usage. BLE is part of the Bluetooth standard, but it differs in some fundamental ways from Classic Bluetooth. For example, BLE uses a different modulation scheme in the PHY layer than does Classic Bluetooth. BLE does, however, share some features and components of Classic Bluetooth, such as the L2CAP layer.

Minimizing the amount of time the radio is on is critical to low-energy operation. BLE is designed to minimize radio on-time in several ways. At the link level, packets are designed to be relatively small. BLE is also designed to support communications that do not require long-lived connections.

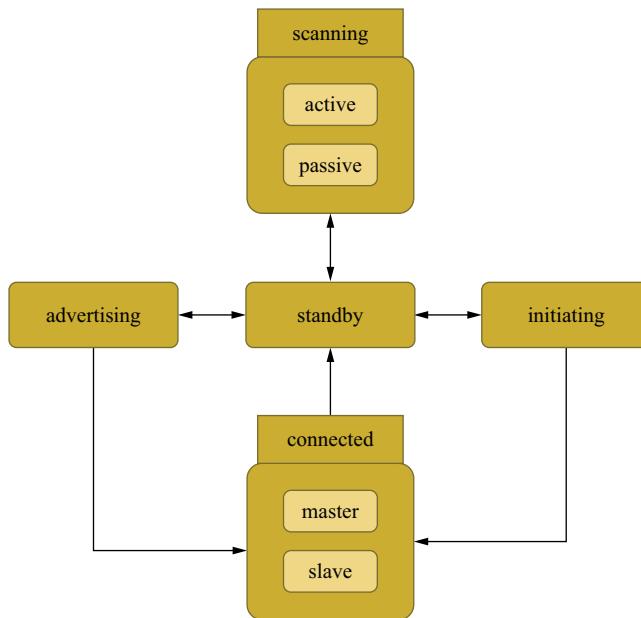
**Advertising** is one form of communication that is designed to support low-energy operation. A device can transmit advertising packets; devices can also listen to advertising packets. Advertising can be used to discover devices or broadcast information. Some short communications may be possible entirely through advertising. If longer communications are required between devices, BLE also supports the establishment of connections.

Fig. 8.13 shows the BLE link-level state machine. The scanning state allows the device to listen to advertising packets from other devices: passive scanning only listens, while active scanning may also send requests for additional information. The advertising state corresponds to a device transmitting advertising packets. A connection may be entered through either the initiating or advertising states. As with Classic Bluetooth, a device in a connection is either a master or a slave. The standby state is reachable from all other states.

The BLE Host Controller Interface (HCI) provides several interfaces to the host: UART provides simple communication facilities; three-wire UART adds capabilities to UART for link establishment and acknowledgment; USB provides high-speed communication; and Secure Digital Input Output (SDIO) is designed for medium-speed communication.

The **Attribute Protocol Layer** provides a mechanism to allow devices to create application-specific protocols for their particular data communication and management needs. An **attribute** has three components: its **handle**, which functions as the name of the attribute; its type, which is chosen from the set of **Universally Unique Identifiers (UUIDs)**; and its **value**. Most UUIDs used in BLE devices are part of a restricted set built around the **Bluetooth-based UUID** and come in several types: service UUIDs, units, attribute types, characteristic descriptors, and characteristic types.

Attributes are collected in an **attribute database** maintained in an attribute server. An attribute client can use the **Attribute Protocol** to query a database. Each device has only one attribute database. An attribute has permissions: readable, writable, or both. Attributes can also be protected with authentication and authorization.

**FIGURE 8.13**

Bluetooth Low Energy link-level state machine.

A set of attributes can be used to define a state machine for a protocol. The states and transitions in the state machine can be stored as attributes; the current state, inputs, and outputs can also be represented by attributes.

The **Generic Attribute Profile Layer** (GATT) defines a basic set of attributes for all BLE devices. The main purpose of the Generic Attribute Profile is to define the procedures for discovery and for the interactions of clients and servers.

BLE provides mechanisms for security. Two devices communicating for the first time is known as **pairing**. This process uses a **short term key** to send a **long term key** to the device. The long-term key is stored in the database, a process known as **bonding**. Data sent as part of a connection may be encrypted using the advanced encryption standard.

#### 8.4.5 802.15.4 and ZigBee

ZigBee is a widely used personal area network (PAN) based on the IEEE 802.15.4 standard (sorry, this standard has no catchy name). 802.15.4 defines the MAC and PHY layers; ZigBee builds upon those definitions to provide several application-oriented standards.

802.15.4 [IEE06] can operate in several different radio bands, including ISM, but also others. The standard is designed for systems with either no battery or those that allow only a small current draw from the battery.

802.15.4 supports two types of devices: **full-function devices (FFDs)** and **reduced-function devices (RFDs)**. An FFD can serve as either a device, a coordinator, or a personal area network coordinator. An RFD can only be a device. Devices can form networks using either a star topology or a peer-to-peer topology. In the case of a star topology network, a PAN coordinator serves as a hub; peer-to-peer networks also have a PAN coordinator, but communications do not have to go through the PAN coordinator.

The basic unit of communication in an 802.15.4 network is the frame, which includes addressing information, error correction, and other information, as well as a data payload. A network can also make use of optional superframes. A superframe, which is divided into 16 slots, has an active portion, followed by an inactive portion. The first slot of a superframe is known as the **beacon**. It synchronizes the nodes in the network and carries network identification information. To support QoS guarantees and low-latency operations, the PAN coordinator may dedicate parts of a superframe to those high-QoS or low-latency operations. Those slots do not have contention.

The PHY layer activates the radio, manages the radio link, sends and receives packets, and other functions. It has two major components: the PHY data service and the PHY management service. The interface to the PHY layer is known as the *physical layer management entity service access point (PLME-SAP)*. The standard uses *carrier sense multiple access with collision avoidance (CSMA-CA)*.

The MAC layer processes frames and other functions. It also provides encryption and other mechanisms that can be used by applications to provide security functions. The MAC layer consists of the MAC data service and MAC management service; its interface is known as the *MLME-SAP*.

ZigBee [Far08] defines two layers above the 802.15.4 PHY and MAC layers: the **NWK** layer provides network services, and the **APL** layer provides application-level services.

The ZigBee NWK layer forms networks, manages the entry and exit of devices to and from the network, and manages routing. The NWK layer has two major components. The **NWK Layer Data Entity (NLDE)** provides data transfer services, and the **NWK Layer Management Entity (NLME)** provides management services. A **Network Information Base (NIB)** holds a set of constants and attributes. The NWK layer also defines the network address for the device.

The NWK layer provides three types of communication: broadcast, multicast, and unicast. A broadcast message is received by every device on the broadcast channel. Multicast messages are sent to a set of devices. A unicast message, the default type of communication, is sent to a single device.

The devices in a network may be organized in many different topologies. A network topology may be determined, in part, by which nodes can physically communicate with each other, but the topology may be dictated by other factors. A message may, in general, travel through multiple hops in the network to its destination. A ZigBee coordinator or router performs a routing process to determine the route through a network used to communicate with a device. The choice of a route can be guided by

several factors: the number of hops or link quality. The NWK layer limits the number of hops that a given frame is allowed to travel.

The ZigBee APL layer includes an **application framework**, an **application support sublayer (APS)**, and a **ZigBee Device Object (ZDO)**. Several **application objects** may be managed by the application framework, each for a different application. The APS provides a services interface from the NWK layer to the application objects. The ZDO provides additional interfaces between the APS and the application framework.

ZigBee defines several **application profiles** that define a particular application. The ZigBee Alliance issues the **application identifier**. The application profile includes a set of **device descriptions** that provide the characteristics and state of the device. One element of the device description also points to a **cluster** that consists of a set of attributes and commands.

#### 8.4.6 Wi-Fi

The 802.11 standard, known as Wi-Fi [IEE97], was originally designed for portable and mobile applications such as laptops. The original standard has been extended several times to include higher-performance links in several different bands. It was designed before ultra-low-energy networking became an important goal. However, a new generation of Wi-Fi designs is designed for efficient power management and operates at significantly lower power levels.

Wi-Fi supports ad hoc networking. A **basic service set (BSS)** consists of two or more 802.11 nodes that communicate with each other. A **distribution system (DS)** interconnects BSSs. More expansive links are provided by an **extended service set (ESS)** network. BSS related by an ESS can overlap or be physically separate. A **portal** connects the wireless network to other networks.

A network provides a set of services. The most basic service is the **distribution** of messages from source to destination. **Integration** delivers a message to a portal for distribution by another network. **Association** refers to the relationship of a station to an access point; **reassociation** allows an association to be moved to a different access point; and **disassociation** allows an association to be terminated. Every station must provide authentication, deauthentication, privacy, and MAC service data unit (MSDU) delivery. A DSS must provide association, disassociation, distribution, integration, and reassociation.

The reference model for 802.11 breaks the PHY layer into two sublayers: **PHY Layer Convergence Protocol (PLCP)** and **Physical Medium Dependent (PMD)**. They communicate with a PHY sublayer management entity. The MAC sublayer communicates with a MAC sublayer management entity. Both management entities communicate with the station management entity.

MAC provides several services:

- Asynchronous data service. This service is connectionless and best-effort.

- Security. Security services include confidentiality, authentication, and access control.
- StrictlyOrdered service. A variety of effects can cause frames to arrive out of order. This service ensures that higher levels see the frames in the strict order in which they are transmitted.

The next example describes a low-power Wi-Fi device.

---

### Example 8.2: Qualcomm QCA4004 Low-Power Wi-Fi

The QCA4004 [Qua15] is a Wi-Fi device designed for low-energy operation. It operates in both 2.4- and 5-GHz bands. Low-energy features include power-saving modes with fast wakeup times. The chip can be interfaced with devices using GPIO or I<sup>2</sup>C.

---

#### 8.4.7 LoRa

**LoRa** stands for *long range*. This network is designed for both low-power operations suitable for IoT systems and coverage over wide areas. The term *LoRa* primarily refers to the PHY layer, which is based on a form of spread spectrum. LoRaWAN is a protocol developed to take advantage of the LoRa PHY layer. An asynchronous protocol is used to allow devices to send data when required, while reducing the power consumption for idle intervals.

---

## 8.5 Databases and timewheels

In this section, we consider mechanisms that can be used to organize information in IoT systems. **Databases** are used in many applications to store collections of information. Since IoT systems often operate devices in real time, a **timewheel** can be used to manage the temporal behavior of the system.

### 8.5.1 Databases

IoT networks use databases to manage and analyze data from IoT devices. IoT databases are often kept in the cloud; this is particularly true when we want to not just store data, but also compute on it. To understand how to use databases, we first need to consider how to put data into the database and then how to extract data from the database.

The traditional database model is the **relational database management system (RDBMS)** [Cod70]. The term relational comes from mathematics: a relation is a Cartesian product of a set of domain values with a set of range values.

As shown in Fig. 8.14, data in a relational database are organized into tables. The rows of the table represent **records** (sometimes called **tuples**). The columns of the table are known as **fields** or **attributes**. One column of the table (or sometimes a set of

**Relational database**

**Data representation**

devices			
name	id (primary key)	address	type
door	234	10.113	binary
record	refrigerator	4326	signal
table	213	11.039	MV
chair	4325	09.423	binary
faucet	2	11.324	signal

device_data			
signature (primary key)	device	time	value
256423	234	11:23:14	1
252456	4326	11:23:47	40
663443	234	11:27:55	0

**FIGURE 8.14**

Tables in a database.

columns) is used as a **primary key**; each record has a unique value for its primary key field, and each key value uniquely identifies the values of the other columns. The *devices* table in the example defines a set of devices. The *id* field serves as the primary key for the devices. The *device\_data* table records timestamped data from the devices. Each of these records has its own primary key, given the name *signature*. Because those records also contain the *id* for the device that recorded the data, we can use a device's *id* field to look up records in the *device\_data* table.

A database contains, in general, more than one table. The set of table definitions in a database is known as its **schema**. *Requirements analysis* is the process of determining what data are required for a given application, as in object-oriented program design.

From a logical perspective, eliminating redundancy is key to maintaining the data in the database. If a piece of data is stored in two different tables (or two different columns in one table), then any change to the data must be recorded in all of its copies. If some copies of the data are changed and some are not, then the value returned will depend on which copy was accessed. In practice, redundant values may improve access times; database management systems can perform this type of optimization without requiring database designers to use redundant schemas.

The database designer's description of the tables does not necessarily reflect how the data are organized in memory or on a disk. The database management system may perform some optimizations to reduce storage requirements or improve access speed. The relational model does not order the records in the table, which helps to

give the database management system more freedom to store the data in the most efficient format.

#### Normal forms

Database **normal forms** are rules that help us create databases without redundancies and other types of problems that may cause problems in database management. Many different normalization rules have been created, some of which we consider here. The **first normal form** is a schema in which every cell contains only a single value; every record has the same number of fields.

The **second normal form** obeys the first normal form, and the values of all the other cells in a record are unique to the key. If a database does not obey the second normal form, then we have duplicated the information in the database. For example, an example database in the second normal form database has two tables: the records in one table contain sensor name, network address, and physical location; records in another table contain sensor name, read time, and value. The name/time pair forms a key for the second table. If the second table's records also included the sensor's physical location, it would not be in second normal form because the sensor location does not depend on the name/time of a reading. Updating the physical location of a sensor would require updating multiple records.

The **third normal form** satisfies the second normal form (and therefore also the first); it also requires that the non-key columns be independent. A database in third normal form has two tables: one with sensor name and sensor model number and another with sensor model number and sensor type (motion, video, and so forth). If the database were modified to have a single record with sensor name, model number, and type, it would not be in the third normal form because the type can be inferred from the model number.

#### Queries

A request for information is known as a **query**. Users do not deal directly with tables. Instead, they formulate a request in a **query language** known as **structured query language (SQL)**. The result is the set of records that satisfy the query. A query may result in more than one record. In the example of Fig. 8.14, we can ask for all the *device\_data* records for the door using the query:

```
select from device_data where device = 234
```

The result would be two records.

#### Joins

A common type of query combines information from more than one table. This operation is known as a **join**. A join can be described mathematically as a Cartesian product of rows. Tables can be related to each other in several different ways:

- one-to-one, if a record in one table corresponds to exactly one record in another table, for example, a sensor and its network address;
- one-to-many, in which a record in one table is related to many records in another table, such as a sensor and a set of readings; and
- many-to-many, in which a set of records can be related to another set of records, such as a group of sensors and readings from that sensor group.

The database management system's job is to efficiently perform the logical operations described by the join. If one type of query is known to occur frequently, the

database management system may optimize the internal representation for that type of query; these sorts of optimizations do not change the schema but merely represent the data in a particular format that is hidden from the user.

Other types of relationships beyond join are possible. A **projection** eliminates some columns in a relation, for example, by paring down a lengthy record of a person to the fields requested by a query. A **restriction** eliminates some rows from a table, for example, by returning only fields with a last name, starting with “A.”

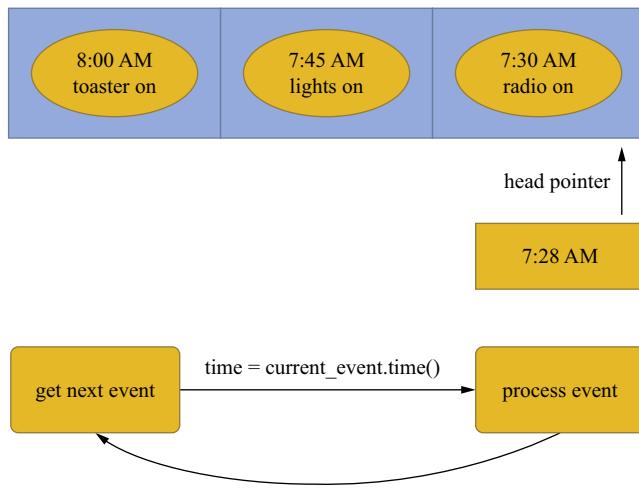
### Schemaless databases

An alternative to the relational model is the **schemaless** or **noSQL** database. The term *schemaless* is something of a misnomer; the data are stored as records, but there is no central knowledge of the format of those records. Data are stored in collections of arrays or tables. The database designer writes software methods to access and modify database records. **JavaScript Object Notation (JSON)** [ECM13] is often used to describe records in a schemaless database. JSON syntax builds objects out of two basic component data structures: name/value pairs and ordered lists of values.

### 8.5.2 Timewheels

The timewheel allows the IoT system to process events in the order in which they occur. This is particularly important when controlling devices—turning lights on and off at specified times, for example. Timewheels are used in event-driven simulators to control the order in which simulated events are processed. We can also use timewheels to manage the temporal behavior of devices in an IoT system [Coe14].

As shown in Fig. 8.15, the timewheel is a sorted list of input and output events. As input events arrive, they are put in sorted order into the queue. Similarly, when output events are scheduled, they are placed in the queue at the proper time order.



**FIGURE 8.15**

Timewheel organization. (a) Timewheel queue. (b) Unified modeling language state diagram.

[Fig. 8.15](#) also provides a UML state diagram for the operation of the timewheel. It pulls the head event from its queue. When the time specified in the event is reached, that event is processed.

A system may have one or more timewheels. A central timewheel can be used to manage activity throughout the entire IoT system. In larger networks, several timewheels can be distributed around the network, each of which keeps track of local activity.

## 8.6 Example: smart home

A **smart home** is a house equipped with sensors that monitor activity and help run the house. A smart home may provide several types of services:

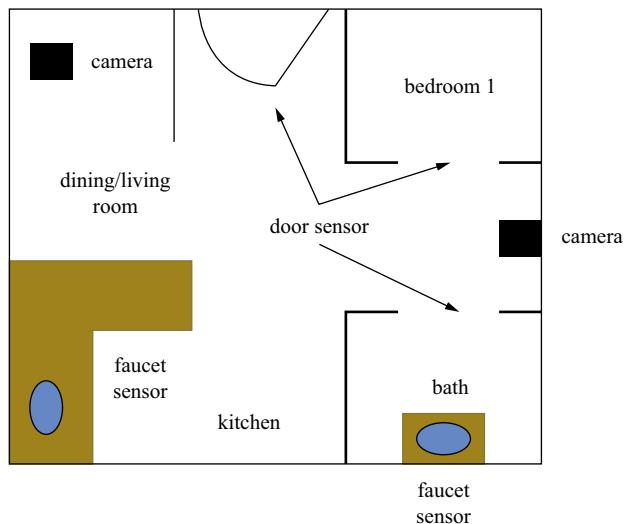
- remote or automatic operation of lights and appliances;
- energy and water management for efficient use of natural resources; and
- monitoring the activities of residents.

Smart homes can be particularly helpful to residents, such as senior citizens or people with special needs [Wol15]. The smart home can analyze activities to, for example, be sure that the resident is taking care of daily tasks. It can also provide summaries of the resident's activity to loved ones, caregivers, and health care professionals. Performing these tasks requires not only operating sensors, but also analyzing the sensor data to extract events and patterns. A smart home system can provide three types of outputs:

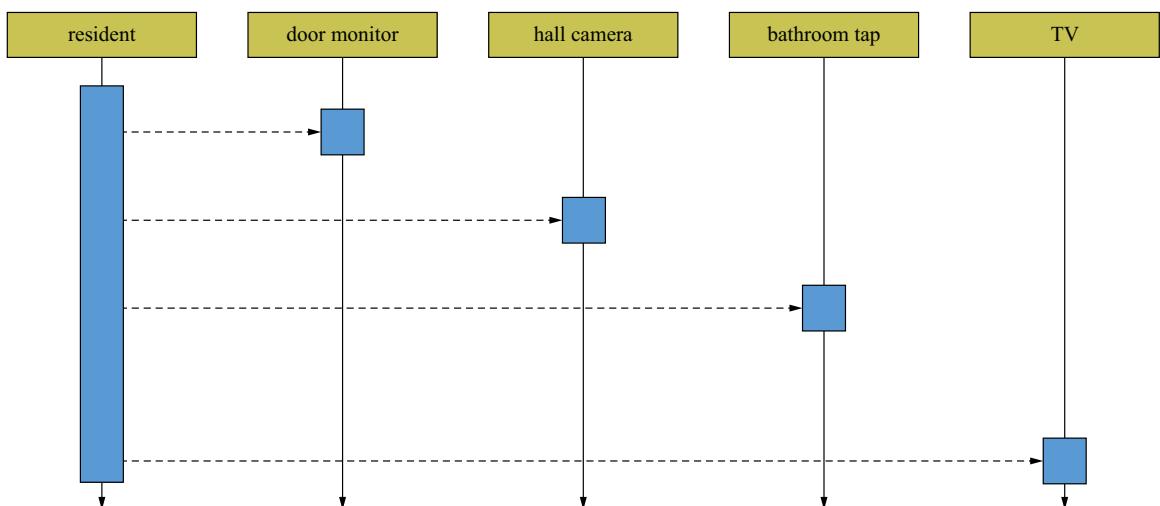
- reports on the activities of residents;
- alerts for out-of-the-ordinary activity; and
- recommendations to the residents and caregivers as to what actions may be taken.

[Fig. 8.16](#) shows the layout of a typical smart home. The home includes several cameras to monitor common areas. Other types of sensors can also help to keep track of activity: door sensors tell when someone has passed through a door, but does not give the person's identity or even whether they are entering or leaving; sensors on water faucets can tell when someone is using a bathroom or kitchen sink; and electrical outlet sensors can tell when someone is using an electrical appliance. Appliances and devices can also be controlled: lights can be turned on and off; heaters and air conditioners can be managed; sprinklers can be turned on and off; and so on.

[Fig. 8.17](#) shows how sensors can be used to monitor a resident's activity. The resident leaves her bedroom, walks through the hallway to the bathroom, uses the sink, then moves to the living room, and turns on the TV. Sensors can be used to monitor these actions. In the case of the hall camera, computer vision algorithms can identify the person in the hallway and track their movement from one room to another. However, the other sensors provide only indirect information. Analysis algorithms can use

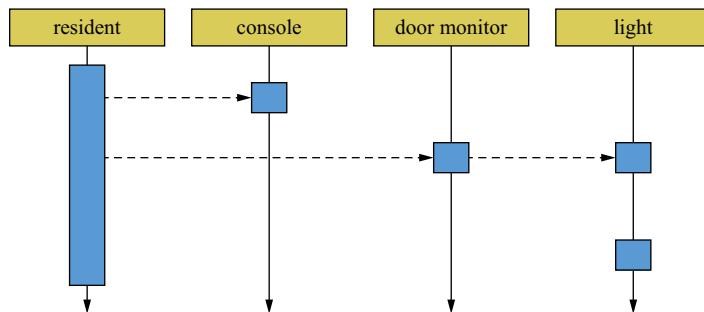
**FIGURE 8.16**

IoT network in a smart home.

**FIGURE 8.17**

Analyzing a resident's activity in a smart home.

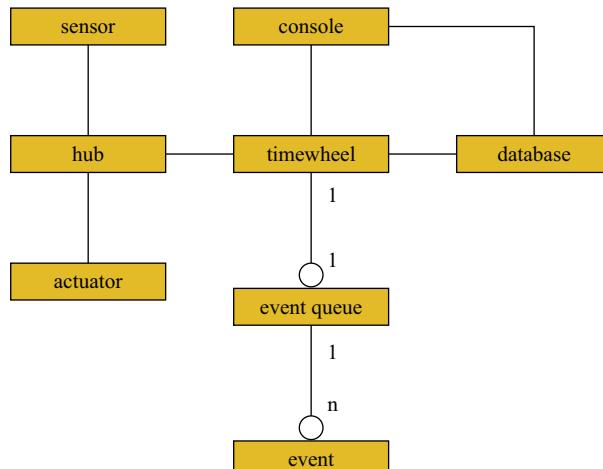
statistical methods to infer that the same person was likely to have caused all these events. For example, a person in another part of a house may not be able to change locations quickly enough to cause some of the events.

**FIGURE 8.18**

Light activation in a smart home.

[Fig. 8.18](#) shows how the smart home can control and monitor activities. The resident uses the console to set up two conditions in which the light should be turned on: whenever anyone goes through the resident's doorway, or at a specified time. In this use case, the resident later goes through the doorway, causing the light to turn on for a specified interval. The light also goes on automatically at the appointed time.

[Fig. 8.19](#) shows an object diagram of a smart home. Sensors and hubs form the network. The console is the main UI. Data are processed in two stages. Sensor readings and events are initially processed by a timewheel that manages the timely operation of devices in the house. Not all sensor readings are of long-term interest. Sensor events for long-term analysis are passed to a database. A database can reside in the cloud and allow access to a variety of analysis algorithms.

**FIGURE 8.19**

UML object diagram for a smart home.

---

## 8.7 Summary

IoT systems leverage low-cost, network-enabled devices to build sophisticated networks. Some different networks can be used to build IoT systems; many practical systems combine several networks. Databases are often used to manage information about IoT devices. Timewheels can be used to manage events in the system.

---

## What we learned

- An IoT system connects edge devices into a network and manages information about those devices in soft real time.
- Bluetooth, ZigBee, and Wi-Fi are all used to connect IoT devices to the rest of the network.
- Databases can be used to store and manage information about IoT devices. Timewheels can be used to manage time-oriented activities in the network.

---

## Further reading

Karl and Willig discuss wireless sensor networks [Kar06]. Serpanos and Wolf discuss IoT systems [Ser18]. Farahani [Far08] describes ZigBee networks. Heydon [Hey13] describes BLE networks.

---

## Questions

**Q8-1** Classify these Bluetooth layers using the OSI model:

- a) baseband
- b) L2CAP
- c) RFCOMM

**Q8-2** Use the power state machine in Fig. 8.12 to determine the energy used in these use cases:

- a) Idle 1 s; receive 10 ms; idle 0.1 s; transmit 5  $\mu$ s.
- b) Sleep 1 min; receive 50 ms; idle 0.1 s; receive 100 ms.
- c) Sleep 5 min; transmit 5  $\mu$ s; receive 10 ms; idle 0.1 s; transmit 10  $\mu$ s.

**Q8-3** Design the schema for a database table that records the activation times of a motion sensor.

**Q8-4** Design the schema for a single database table that records the activation times for several motion sensors.

- Q8-5** You are given a timewheel that is initially empty. The timewheel processes events, each of which has a generation time and a release time. Show the state of the timewheel (events and their order) after each of these events is received at its generation time. Times are given as mm:ss (minutes, seconds).
- a) e1: generation 00:05, release 00:06.
  - b) e2: generation 00:10, release 20:00.
  - c) e3: generation 01:15, release 10:00.
  - d) e4: generation 12:15, release 12:20.
  - e) e5: generation 12:16, release 12:18.

---

### Lab exercises

- L8-1** Use Bluetooth to connect a simple sensor, such as an electric eye, to a database.
- L8-2** Use a temperature sensor and a motion sensor to determine the average temperature in a room when a person is present.
- L8-3** Design a database schema for a smart classroom. Identify the features of the smart classroom and design the schema to support those use cases.

# Automotive and Aerospace Systems

9

## CHAPTER POINTS

- Networked control in cars and airplanes.
- Networks for vehicles.
- Security and safety of vehicles.

## 9.1 Introduction

Cars and airplanes are superb examples of complex embedded computing systems; we have real-life experience with them and understand what they do. They represent very large industries, and they are examples of safety-critical real-time distributed embedded systems.

We start with a discussion of use cases for vehicles. [Section 9.2](#) discusses networked control in cars and airplanes. [Section 9.3](#) describes in more detail several networks used in vehicles. [Section 9.4](#) considers the safety and security issues of vehicles.

## 9.2 Vehicular use cases

Vehicles have been important markets for embedded computers since the early days of microprocessors. The advent of autonomous vehicles has enhanced the role of embedded computing in vehicles.

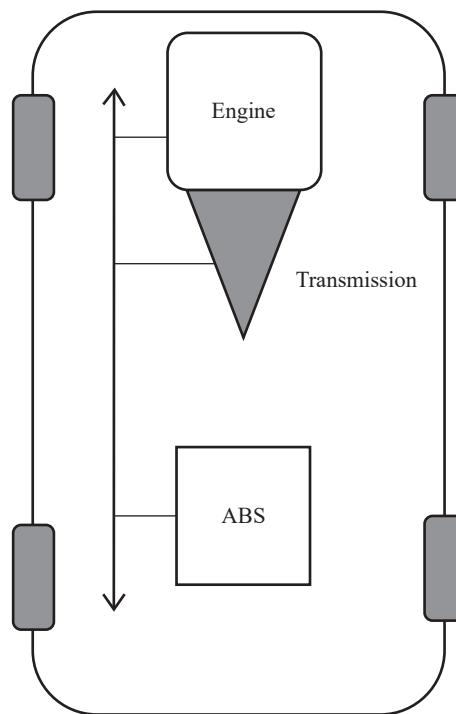
### 9.2.1 Vehicles as cyber-physical systems

Cars and airplanes are examples of cyber-physical systems; software provides real-time control for the physical plant.

[Fig. 9.1](#) shows a car network and three of the major subsystems in the car: the engine, the transmission, and the antilock braking system (ABS). Each is a mechanical system controlled by a processor. First, consider the roles of mechanical systems, all of which are mechanically coupled:

- The engine provides the power to drive the wheels.

#### Car subsystems

**FIGURE 9.1**

Major elements of an automobile network.

- The transmission mechanically transforms the engine's rotational energy into a form most useful by the wheels.
- The ABS system controls how the brakes are applied to each of the four wheels. It can separately control the brake on each wheel.

Now, consider the roles of the associated processors:

- The engine controller accepts commands from the driver via the gas pedal. It also takes several measurements. Based on the commands and measurements, it determines the spark and fuel timing of every engine cycle.
- The transmission controller determines when to change gears.
- The ABS system takes braking commands from the driver via the brake pedal. It also takes measurements from the wheels about their rotating speeds. It turns the brakes on and off each wheel to maintain traction on each wheel.

These subsystems must communicate with each other to do their jobs:

- The engine controller may change the spark timing during gear shifting to reduce shocks during shifting.

- The transmission controller must receive the throttle position from the engine controller to help determine the proper shifting pattern for the transmission.
- The ABS system tells the transmission when brakes are being applied in case the gear needs to be shifted.

None of these tasks needs to be performed at the highest rates in the system, which are the rates for spark timing. A relatively small amount of information can be exchanged to achieve the desired effect.

## Avionics

Aircraft electronics are known as **avionics**. The most fundamental difference between avionics and automotive electronics is **certification**. Anything that is permanently attached to the aircraft must be certified. The certification process for production aircraft is twofold. First, the design is certified in a process known as **type certification**; then, the manufacture of each aircraft is certified during production. The certification process is a prime reason why avionics architectures are more conservative than automotive electronics systems.

### 9.2.2 Driver assistance and autonomy

Driving can be assisted or automated with many levels of sophistication. A complete trip generally requires several different types of driving: leaving a parking space, low-speed driving on local streets, cruising, and parking. The term **advanced driver-assistance system (ADAS)** encompasses a broad range of functions for various parts of the trip. Some cars provide parking assistance by using sensors to warn of proximity to obstacles; other cars fully automate some types of parking. Adaptive cruise control adjusts cruise speed when vehicles ahead drive more slowly. Emergency braking can be performed to avoid collisions. Automated driving refers to the more complete operation of the vehicle. However, driving can be automated to several levels of sophistication. SAE International has defined the following levels of driving automation [SAE18]:

- Level 0, no driving automation.
- Level 1, driver assistance. Sustained execution of either a lateral or longitudinal motion control subtask but not both. The driver is expected to perform the rest of the driving task.
- Level 2, partial driving automation. Sustained execution of both lateral and longitudinal vehicular motion control. The driver watches for objects and events and supervises the driving automation system.
- Level 3, conditional driving automation. Sustained performance of a specific dynamic driving task. The user will respond to requests from the system to intervene and respond to performance-relevant system failures.
- Level 4, high driving automation. Sustained performance of an operation design domain of a driving task and fallback. The user is not expected to respond to a request to intervene.
- Level 5, full driving automation. The sustained and unconditional performance of a dynamic driving task and fallback, not limited to a particular operational design

domain. An operational design domain is the environment or situation in which an automated system is designed to operate properly. The user is not expected to respond to a request to intervene.

---

### 9.3 Networked control systems in cars and airplanes

Cars and airplanes are examples of **networked control systems**: computer networks with processors and I/O devices that perform control functions. Control systems require real-time responsiveness over a closed loop, from measurement back to control action. Although a simple control system may be built with a microprocessor and a few I/O devices, complex machines require network-based control. The network has several principal uses. First, a network allows more computing power to be applied to the system than would be possible with a single CPU. Second, many control applications require the controller to be physically near the controlled device. Machines with fast reaction rates require controllers to respond quickly. If the controller is placed physically distant from the machine, the communication time to and from the controller may interfere with its ability to properly control the plant. A network allows several controllers to be placed near the components they control (e.g., engines and brakes) while allowing them to cooperate in the overall control of the car.

Modern automobiles may contain over 100 processors that execute 100 million lines of code [Owe15]. Modern airplanes incorporate less software, in large part due to the demands of certification. However, modern airplanes still rely on computers and networks for flight operations. Data on vehicle networks serve a wide range of purposes, ranging from critical vehicle controls to navigation and passenger entertainment. Autonomous driving imposes additional requirements on a vehicle's computing platform [Liu17]. Autonomous vehicles generally use multiple sensors of several types. Perceptual tasks include localization, object detection, and object tracking. Based on these results, the autonomous system must predict the actions of surrounding moving objects, plan paths to move toward the destination, and avoid obstacles as the environment changes.

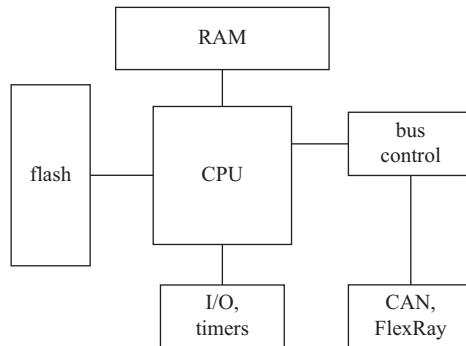
#### 9.3.1 Network devices

Different terms are used in automotive and aerospace systems for devices connected to a network. An **electronic control unit (ECU)** is widely used in automotive design. The acronym *ECU* originally referred to an “engine control unit,” but the meaning of the term was later expanded to any electronic unit in the vehicle. A **line replaceable unit (LRU)** is used in aircrafts for a unit that can be easily unplugged and replaced during maintenance.

##### ECUs

The next two examples describe ECUs designed for automotive systems: one is designed for body electronics, such as doors and lighting, and the other is designed for engine control.

---

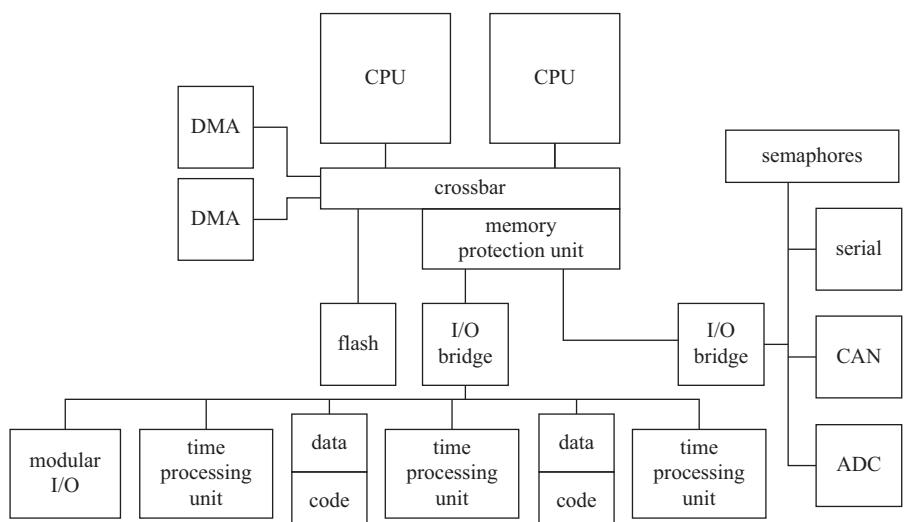
**Example 9.1: Infineon XC2200**


The XC2200 family [Inf12] covers a range of automotive applications. One of the members of that family is designed for body control (e.g., lighting, door locks, and wiper control). The Body Control Module includes a 16/32 bit processor [Inf08], electronically erasable programmable ROM and static RAM, analog-to-digital converters, pulse-width modulators, serial channels, light drivers, and network connections.

---



---

**Example 9.2: Freescale MPC5676R**


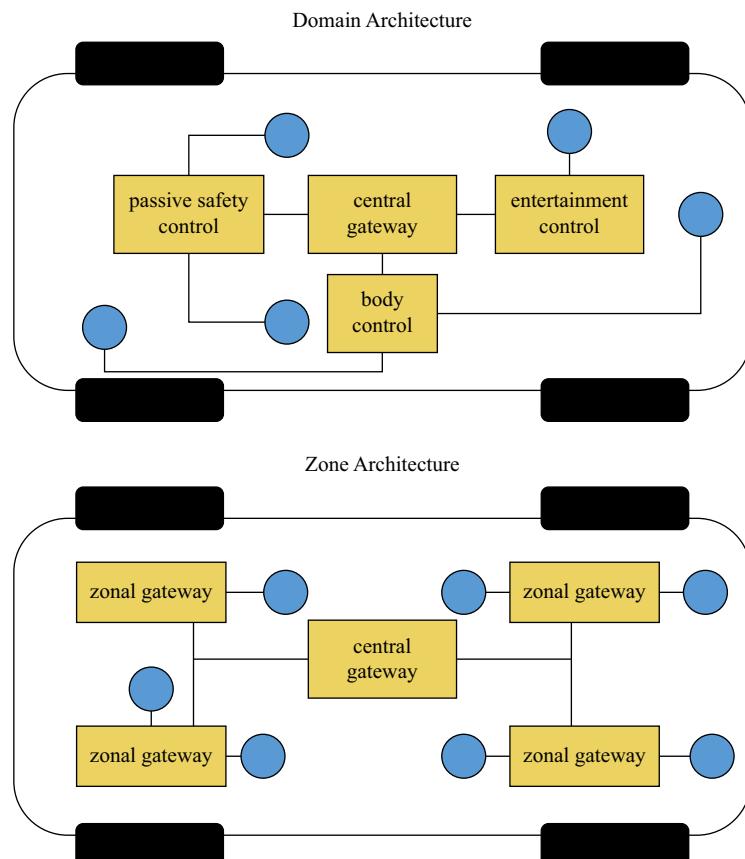
The MPC5676R [Fre11B] is a dual-processor platform for powertrain systems.

The two main processors are members of the Power Architecture Book E architecture and are user-mode compatible with PowerPC. They provide short vector instructions for use in signal processing. Each processor has its own 16 K data and instruction caches. The time-processing unit can be used to generate and read waveforms. Interfaces to the CAN, the LIN, and FlexRay networks are supported.

### 9.3.2 Vehicle network architectures

#### Automotive networks

[Fig. 9.2](#) shows two architectures for automotive networks. The traditional organization of networks in an automobile is known as **domain architecture** [Vem20]. ECUs are provided for various functions, including engine control, braking, entertainment, and so forth. Network connections are made to the ECU from each device for which it is responsible.



**FIGURE 9.2**

Domain and zone automotive network architectures.

The **zone architecture** has emerged as an alternative architecture. Several **zonal gateways** are positioned at various locations in the car. For example, a **central gateway** connects the zonal gateways. Each device connects to the zonal gateway to which it is closest. As a result, a given zonal gateway may perform several distinct types of functions.

Zone architectures have emerged as an approach to simplify automotive wiring. Wires in a car are typically organized into **harnesses**. The wiring harness of a car is typically its third heaviest component, after the body and engine [Kla19]. It is also the third-most expensive component in the car. Connecting devices to a nearby gateway can substantially reduce the size and cost of the wiring harness.

### Aircraft networks

The traditional architecture [Hel04] for an avionics system has a separate LRU for each function: artificial horizon, engine control, flight surfaces, and so on.

A more sophisticated system is the bus-based one. The Boeing 777 avionics [Mor07], for example, are built from a series of racks. Each rack is a set of core processor modules (CPMs), I/O modules, and power supplies. CPMs may implement one or more functions. A bus known as SAFEbus connects the modules. Cabinets are connected using a serial bus known as ARINC 6210.

A distributed approach to avionics is the **federated network**. In this architecture, a function or several functions have their own network. The networks share data necessary for the interaction of these functions. A federated architecture is designed so that failure in one network will not interfere with the operation of the other networks.

The Genesis Platform [Wal07] is a next-generation architecture for avionics and safety-critical systems; it is used on the Boeing 787 Dreamliner. Unlike federated architectures, it does not require a one-to-one correspondence between application groups and network units. In contrast, Genesis defines a virtual system for avionic applications that are then mapped onto a physical network that may have a different topology.

## 9.4 Vehicular networks

Vehicular networks often have relatively low bandwidth when compared to fixed networks, such as local area networks. However, the computations are organized so that each processor has to send only a relatively small amount of data to other processors to do the system's work. We will first look at the CAN bus, which is widely used in cars and sees some use in airplanes. We will then briefly consider other vehicular networks.

### 9.4.1 CAN bus

The **Controller Area Network** or **CAN bus** [Bos07] was designed for automotive electronics, and was first used in production cars in 1991. A CAN network consists of a set of electronic control units connected by the CAN bus; the ECUs pass messages to each other using the CAN protocol. The CAN bus is used for safety-critical operations, such as antilock braking. It is also used in less-critical applications,

such as passenger-related devices. CAN is well-suited to the strict requirements of automotive electronics: reliability, low power consumption, low weight, and low cost.

CAN comes in several variations. The version known as high-speed CAN uses bit-serial communication and runs at rates of up to 1 Mb/s over a twisted pair connection of 40 m. An optical link can also be used. The bus protocol supports multiple masters on the bus.

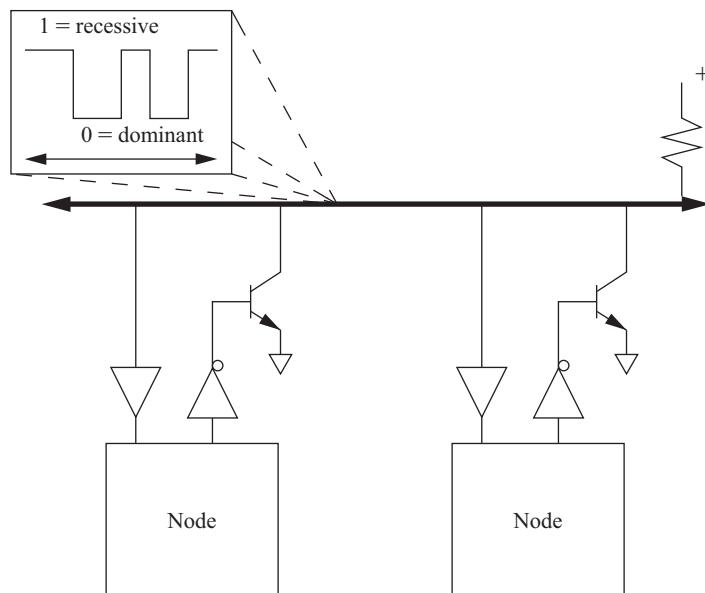
#### Physical layer

As shown in Fig. 9.3, each node in the CAN bus has its own electrical drivers and receivers that connect the node to the bus in a wired-AND fashion. The driving circuits on the bus cause the bus to be pulled down to 0 if any node on the bus pulls the bus down; this bus voltage state is known to be **dominant** [Wat17]. If no node pulls down the bus, the bus voltage remains high; this state is known as **recessive**. When all nodes are transmitting ones, the bus is said to be in the recessive state. When a node transmits a zero, the bus is in the dominant state. Data are sent on the network in packets known as **data frames**.

CAN is a synchronous bus; all transmitters must send at the same time for bus arbitration to work. Nodes synchronize themselves with the bus by listening to the bit transitions on the bus. The first bit of a data frame provides the first synchronization opportunity in a frame. The nodes must also continue to synchronize against later transitions in each frame.

#### Data frame

The format of a CAN data frame is shown in Fig. 9.4. A data frame starts with a dominant bit and ends with a string of seven recessive bits. There are at least three bit fields between the data frames. The first field in the packet contains the packet's



**FIGURE 9.3**

Physical and electrical organization of a CAN bus.

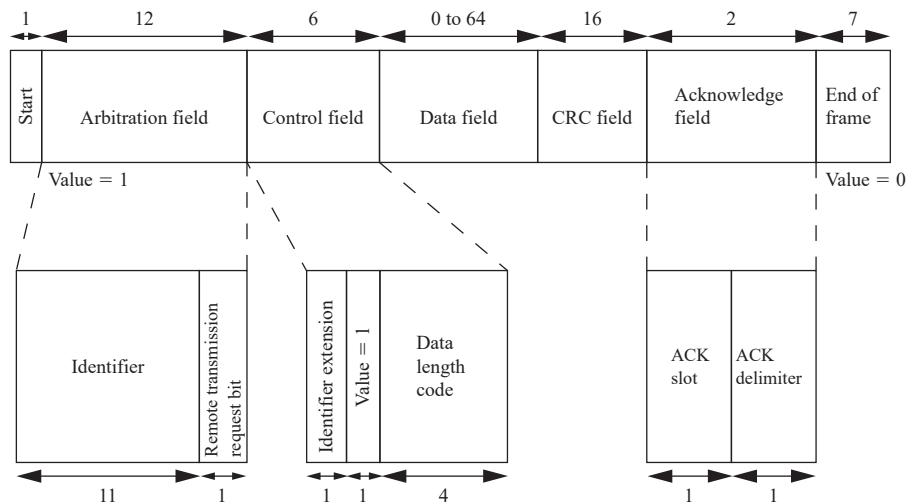


FIGURE 9.4

CAN data frame format.

destination address, which is known as the arbitration field. The destination identifier is 11 bits long. The trailing remote transmission request (RTR) bit is set to 0 if the data frame is used to request data from the device specified by the identifier. When RTR = 1, the packet is used to write the data to the destination identifier. The control field provides an identifier extension and a 4-bit length for the data field with a 1 in between. The data field is from 0 to 8 bytes, depending on the value given in the control field. A cyclic redundancy check (CRC) is sent after the data field for error detection. The acknowledge field is used to let the identifier signal whether the frame was correctly received: the sender places a recessive bit (1) in the acknowledgment (ACK) slot of the acknowledge field; if the receiver detects an error, it forces the value to a dominant (0) value. If the sender sees a 0 on the bus in the ACK slot, it knows it must retransmit. The ACK slot is followed by a single bit delimiter, followed by the end-of-frame field.

### Arbitration

Control of the CAN bus is arbitrated using a technique known as Carrier Sense Multiple Access with Arbitration on Message Priority (CSMA/AMP). CAN encourages a data-push programming style. Network nodes transmit synchronously, so they all start sending their identifier fields simultaneously. When a node hears a dominant bit in the identifier and tries to send a recessive bit, it stops transmitting. By the end of the arbitration field, only one transmitter will be left. The identifier field acts as a priority identifier, with the all-0 identifier having the highest priority.

### Remote frames

A remote frame is used to request data from another node. The requestor sets the RTR bit to a recessive bit to specify a remote frame; it also specifies zero data bits. The node specified in the identifier field will respond with a data frame that has the requested value. Note that there is no way to send parameters in a remote

**Error handling**

frame; for example, you cannot use an identifier to specify a device and provide a parameter to say which data value you want from that device. Instead, each possible data request must have its own identifier.

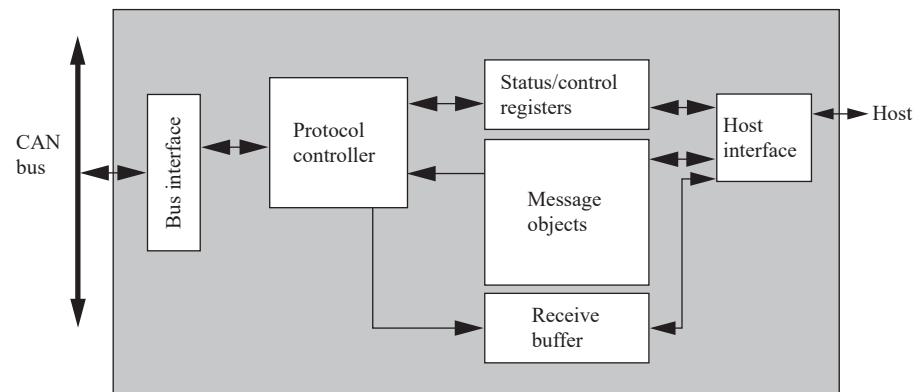
An error frame can be generated by any node that detects an error on the bus. Upon detecting an error, a node interrupts the current transmission with an error frame, which consists of an error flag field followed by an error delimiter field of eight recessive bits. The error delimiter field allows the bus to return to the quiescent state so that data frame transmission can resume. The bus also supports an overload frame, which is a special error frame sent during the interframe quiescent period. An overload frame signals that a node is overloaded and will not be able to handle the next message. The node can delay the transmission of the next frame with up to two overload frames in a row, hopefully giving it enough time to recover from its overload. The CRC field can be used to check a message's data field for correctness.

If a transmitting node does not receive an acknowledgment of a data frame, it should retransmit the data frame until the data are acknowledged. This action corresponds to the data link layer in the Open Systems Interconnection model.

[Fig. 9.5](#) shows the basic architecture of a typical CAN controller. The controller implements the physical and data link layers; because the CAN is a bus, it does not need network layer services to establish end-to-end connections. The protocol control block handles determining when to send messages, when a message must be resent because of arbitration losses, and when a message should be received.

**Time-triggered architectures****9.4.2 Other automotive networks**

The **time-triggered architecture** [Kop03] is an architecture for networked control systems that provides more reliable communication delays. Events in the time-triggered architecture are organized around real time. Since devices in the network need time to respond to communication events, time is modeled as a sparse system.

**FIGURE 9.5**

Architecture of a CAN controller.

Intervals of active communication are interspersed with idle periods. This model ensures that even if the clock's value varies somewhat from device to device, all devices on the network will be able to maintain the order of events in the system.

#### FlexRay

The **FlexRay network** [Nat09] has been designed as the next generation of system buses for cars. FlexRay provides high data rates—up to 10 Mbits/s—with deterministic communication. It is also designed to be fault tolerant. Communications on the bus are designed around a communication cycle. Part of the cycle, known as the *static segment*, is dedicated to events for which communication time has been guaranteed. The communication time for each of these events is determined by a schedule set up by the designer. Some devices may need only sporadic communication, and they can make use of the *dynamic segment* for these events.

#### LIN

The Local Interconnect Network (LIN) bus [Bos07] was created to connect components in a small area, such as a single door. The physical medium is a single wire that provides data rates of up to 20 kbytes/s for up to 16 bus subscribers. All transactions are initiated by the master and responded to by a frame. The software for the network is often generated from a LIN description file that describes the network subscribers, the signals to be generated, and the frames.

Several buses have come into use for passenger entertainment. Bluetooth is becoming the standard mechanism for cars to interact with consumer electronics devices, such as audio players or phones.

#### MOST

The Media-Oriented Systems Transport (MOST) bus [Bos07] was designed for entertainment and multimedia information. The basic MOST bus runs at 24.8 Mbytes/s and is known as MOST 25, and 50 and 150 Mbytes/s versions have been developed. MOST can support up to 64 devices. The network is organized as a ring.

Data transmission is divided into channels. A control channel transfers control and system management data. Synchronous channels are used to transmit multimedia data; MOST 25 provides up to 15 audio channels. An asynchronous channel provides high data rates, but without the quality-of-service guarantees of the synchronous channels.

#### 100 BASE-T1

*Automotive Ethernet* refers to any variation of the Ethernet used in automobiles. The 100BASE-T1 standard is one example of a commonly used version of Ethernet for automobiles. Signals are carried over an unshielded twisted pair of wires. Unlike many other forms of Ethernet, this standard allows full-duplex connections; devices at both ends of a connection can transmit and receive simultaneously over the twisted pair.

The next example looks at a controller designed to interconnect LIN and CAN buses.

---

### Example 9.3: Automotive Central Body Controllers

A central body controller [Tex11C] runs various devices that are part of the car body, including lights, locks, windows, and so forth. It includes a CPU that performs management, communications, and power management functions. The processor interfaces with the CAN bus and LIN bus transceivers. Devices such as remote car locks, lights, or wiper blades are connected to LIN buses. The processor transfers commands and data between the main CAN bus and the device-centric LIN buses, as appropriate.

---

## 9.5 Safety and security

Cars and airplanes pose important challenges for the design of safe and secure embedded systems. The large number of vehicles makes them potentially dangerous; their internal complexity makes them vulnerable to a wide variety of threats.

### Threat models

Vehicles are vulnerable to threats from many sources:

- **Maintenance.** Maintenance technicians must access the vehicle’s internals, including computers. They may maliciously modify components. Even if the technicians are not malicious, if the computers they use for their work have been compromised, they may act as conduits or gateways for attacks.
- **Component suppliers.** Components may be shipped with backdoors or other problems. The components may come from a corrupt supplier or may be the victim of unauthorized modification by an errant employee.
- **Passengers.** Modern vehicles supply network connections for passengers. These networks can easily serve as avenues for attackers to enter the vehicle’s core systems.
- **Passersby.** Wireless passenger networks may extend their range beyond the car, allowing others to attack. Wireless door lock mechanisms provide another avenue for attack. Some cars provide telematics services for remote access to the vehicle’s operation, providing another avenue of attack.

### Example attacks

These threat models are not hypothetical—they are realistic. The next example describes two experiments in car hacking. Afterwards, we present an example of a possible airplane hacking incident.

---

### Example 9.4: Experiments in Car Hacking

Researchers have demonstrated techniques for hacking automobiles [Kos10, Che11]. To demonstrate the seriousness of the vulnerabilities found, they concentrated only on techniques that provided them with control over all of the car’s systems. The team identified a variety of methods to access the car’s internals: using traditional hacking techniques to infect the diagnostic computers used by mechanics; using a specially coded CD to modify the code of the CD player and then using the CD player to infect other car devices; sending signals to the car’s telematics system to control it.

Computer security researchers demonstrated vulnerabilities by taking over a Jeep Cherokee driven by a journalist [Gre15]. The car’s telematics system was used to obtain entry into the vehicle’s computer systems. The entertainment system was then compromised, and its software modified; the computers did not check the validity of the software updates. The entertainment system was then used to send messages on the car’s CAN bus to control other parts of the car, such as killing the engine or disabling the brakes.

---

---

### Example 9.5: Airplane Hacking

A computer security researcher was arrested on suspicion of having hacked into Boeing 737 during a flight [Pag15]. An affidavit states that the person hacked into the in-flight entertainment system, and then, modified code in the Thrust Management Computer.

---

#### Safety

Not all problems are caused by malicious activity. Software bugs can result in serious safety problems, including accidents. The next example describes an airplane crash in which software problems are implicated.

---

### Example 9.6: Software Implicated in Airplane Crash

Software bugs are suspected in the crash of an Airbus A400M [Pag15B, Chi15]. Software in the ECUs is suspected to have caused three engines on an A400M to shut down during flight, causing a fatal crash.

---

The following example describes the issues raised in lawsuits on automotive software.

---

### Example 9.7: Design Errors Implicated in Car Crashes

A court in Oklahoma ruled that Toyota was liable in a case of unintended acceleration [Dun13]. An expert in the case testified that the electronic throttle control system source code was of unreasonably low quality, that software metrics predicted additional bugs, and that the car's fail-safe capabilities were both inadequate and defective. Koopman [Koo14] provided a detailed summary of the topics from the case. His summary states that the car's electronic throttle control system code contained 67 functions with a cyclomatic complexity over 50, and that the throttle angle function had a cyclomatic complexity of 146, with a cyclomatic complexity value over 50 considered "untestable."

---

In another case, a car manufacturer implemented a **defeat** on its own cars; it installed software that deactivated air pollution controls on its cars.

---

### Example 9.8: Volkswagen Diesel Defeat

In 2015, Volkswagen admitted to installing a **defeat** of its own software on its diesel cars [Tho15]. The software defeat was detected when the vehicle was being tested for emissions; in this case, the software enabled all emissions control features. When the car was not being tested, a variety of emissions controls was disabled. With disabled emissions controls, cars could emit up to 40 times more emissions.

---

Autonomy introduces new safety concerns. Example 9.9 describes a fatal collision involving a pedestrian and a vehicle controlled by a developmental automated driving system.

---

### **Example 9.9: Collision Between Vehicle Controlled by a Developmental Automated Driving System and Pedestrian**

The National Transportation Safety Board (NTSB) [Nat19] reported an accident that occurred on the evening of March 18, 2018, in Tempe, Arizona. A proprietary developmental automated driving system was operational in a test vehicle and active at the time of the crash. The vehicle struck and fatally injured a pedestrian crossing N. Mill Avenue outside a crosswalk. The NTSB's provided this probable cause statement:

*The National Transportation Safety Board determines that the probable cause of the crash in Tempe, Arizona, was the failure of the vehicle operator to monitor the driving environment and the operation of the automated driving system because she was visually distracted throughout the trip by her personal cell phone. Contributing to the crash were the Uber Advanced Technologies Group's (1) inadequate safety risk assessment procedures, (2) ineffective oversight of vehicle operators, and (3) lack of adequate mechanisms for addressing operators' automation complacency—all a consequence of its inadequate safety culture. Further factors contributing to the crash were (1) the impaired pedestrian's crossing of N. Mill Avenue outside a crosswalk, and (2) the Arizona Department of Transportation's insufficient oversight of automated vehicle testing.*

The vehicle's developmental automated driving system was designed to operate in autonomous mode only on designated and premapped routes. The automated driving system's sensors included a single light detection and ranging system, 8 dual-ranging radars, and 11 cameras.

The report describes that the automated driving system first detected the pedestrian 5.6 s before the crash, first as a vehicle, then as an unknown object and a bicyclist. The automated driving system continued to track the pedestrian until the crash. However, it did not correctly predict the pedestrian's path or reduce the vehicle's speed in response. At 1.2 s before impact, the automated driving system determined that a collision was imminent, and that the situation exceeded the response specifications of the automated driving system's braking system to avoid collision. The design of the vehicle relied on the operator to take control of the vehicle.

The NTSB report provides several recommendations:

- To the National Highway Traffic Safety Administration:

*Require entities who are testing or who intend to test a developmental automated driving system on public roads to submit a safety self-assessment report to your agency. (H-19-47)*

*Establish a process for the ongoing evaluation of the safety self-assessment reports as required in Safety Recommendation H-19-47 and determine whether the plans include appropriate safeguards for testing a developmental automated driving system on public roads, including adequate monitoring of vehicle operator engagement, if applicable. (H-110-48)*

- To the State of Arizona:

*Require developers to submit an application for testing automated driving system (ADS)-equipped vehicles that, at a minimum, details a plan to manage the risk associated with crashes and operator inattentiveness and establishes countermeasures to prevent crashes or mitigate crash severity within the ADS testing parameters. (H-19-49)*

*Establish a task group of experts to evaluate applications for testing vehicles equipped with automated driving systems, as described in Safety Recommendation H-19-49, before granting a testing permit. (H-19-50)*

- To the American Association of Motor Vehicle Administrators:

*Inform the states about the circumstances of the Tempe, Arizona, crash and encourage them to (1) require developers to submit an application for testing automated driving system (ADS)-equipped vehicles that, at a minimum, details a plan to manage the risk associated with crashes and operator inattentiveness and establishes countermeasures to prevent crashes or mitigate crash severity within the ADS testing parameters, and (2) establish a task group of experts to evaluate the application before granting a testing permit. (H-19-51)*

- To the Uber Technologies, Inc., Advanced Technologies Group:

*Complete the implementation of a safety management system for automated driving system testing that, at a minimum, includes safety policy, safety risk management, safety assurance, and safety promotion. (H-19-52)*

---

## 9.6 Summary

Automobiles and airplanes rely on embedded software, and they illustrate several important concepts in advanced embedded computing systems. They are organized as networked control systems with multiple processors communicating to coordinate real-time operations. They are safety-critical systems that demand the highest levels of design assurance.

---

## What we learned

- Cars and airplanes make use of networked control systems.
- The computing platforms for vehicles make use of heterogeneous sets of processors that communicate over heterogeneous networks.
- The complexity of vehicles creates challenges for secure and safe vehicle design.
- Engine controllers execute mathematical control functions at high rates to operate the engine.

---

## Further reading

Kopetz [Kop97] provided a thorough introduction to the design of distributed embedded systems. The book by Robert Bosch GmbH [Bos07] discusses automotive electronics in detail. The Digital Aviation Handbook [Spi07] describes the avionics systems of several aircraft.

## Questions

- Q9-1** Give examples of the component networks in a federated network for an automobile.
- Q9-2** Draw a UML sequence diagram for a use case of a passenger sitting in a car seat and buckling the seatbelt. The sequence diagram should include the passenger, the seat's passenger sensor, the seat belt fastening sensor, the seat belt controller, and the seat belt fastened indicator, which is on when the passenger is seated, but the seat belt is not fastened.
- Q9-3** Draw a UML sequence diagram for a use case for an attack on a car through its telematics unit. The attack first modifies the software on the telematics unit and then modifies software on the brake unit. The sequence diagram should include the telematics unit, the brake unit, and the attacker.
- 

## Lab exercises

- L9-1** Build an experimental setup that lets you monitor messages on an embedded network.
- L9-2** Build a CAN bus monitoring system.

# Embedded Multiprocessors

# 10

## CHAPTER POINTS

---

- Why we need networks and multiprocessors in embedded computing systems.
  - Embedded multiprocessor architectures.
  - System design for parallel and distributed computing.
  - Design example: video accelerator.
- 

### 10.1 Introduction

Many embedded computing systems require more than one CPU. To build such systems, we must use networks to connect processors, memory, and devices. We must then program the system to take advantage of the parallelism inherent in multiprocessing and to account for the communication delays incurred by networks. This chapter introduces some basic concepts in parallel and distributed embedded computing systems. Section 10.2 outlines the case for using multiprocessors in embedded systems. Section 10.3 examines the categories of multiprocessors. Section 10.4 considers shared memory multiprocessors and multiprocessor systems-on-chip (MPSoCs). Section 10.5 walks through the design of a video accelerator as an example of a specialized processing element (PE).

---

### 10.2 Why multiprocessors?

#### Definitions

Programming a single CPU is hard enough. Why make life more difficult by adding more processors? A **multiprocessor** is, in general, any computer system with two or more processors coupled together. Multiprocessors used for scientific or business applications tend to have regular architectures that include several identical processors that can access a uniform memory space. We use the term **processing element (PE)** to mean any unit responsible for computation, whether or not it is

**Why so many?**

programmable. We use the term **network** (or **interconnection network**) to describe the interconnections between the processing elements.

**Cost/performance**

Embedded system designers must take a more general view of the nature of multiprocessors. As we will see, embedded computing systems are built atop a complete spectrum of multiprocessor architectures. Why is there no single multiprocessor architecture for all types of embedded computing applications? And why do we need embedded processors at all? The reasons for multiprocessors are the same reasons that drive embedded system design: real-time performance, power consumption, and cost.

The first reason for using an embedded multiprocessor is that it can offer significantly better cost/performance—that is, functionality per dollar spent on the system—than would be obtained by spending the same amount of money on a uniprocessor system. The reason for this is that the processing element purchase price is a *nonlinear* function of performance [Wol08]. The cost of a microprocessor increases greatly as clock speed increases. We would expect this trend as a normal consequence of very large-scale integration (VLSI) fabrication and market economics. Clock speeds are normally distributed by normal variations in VLSI processes; because the fastest chips are rare, they command a high price in the marketplace.

Because the fastest processors are very costly, splitting the application so that it can be performed on several smaller processors is much cheaper. Even with the added costs of assembling components, the total system is less expensive. Of course, splitting the application across multiple processors entails higher engineering costs and lead times, which must be factored into the project.

**Real-time performance**

In addition to reducing costs, using multiple processors can also help with real-time performance. We can often meet deadlines and be responsive to interactions much more easily when we put those time-critical processes on separate processors. Given that scheduling multiple processes on a single CPU incurs overhead in most realistic scheduling models, as discussed in [Chapter 6](#), putting the time-critical processes on processing elements that have little or no time sharing reduces scheduling overhead. Because we pay for that overhead at a nonlinear rate for the processor, as illustrated in [Fig. 10.1](#), the savings by segregating time-critical processes can be large; it may take an extremely large and powerful CPU to provide the same responsiveness that can be had from a distributed system.

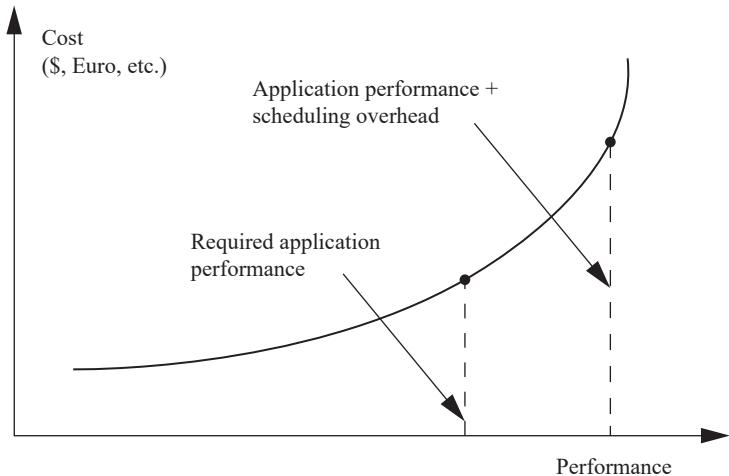
**Cyber-physical considerations**

We may also need to use multiple processors to put some of the processing power near the physical systems being controlled. Cars, for example, put control elements near the engine, brakes, and other major components. Analog and mechanical needs often dictate that critical control functions be performed very close to the sensors and actuators.

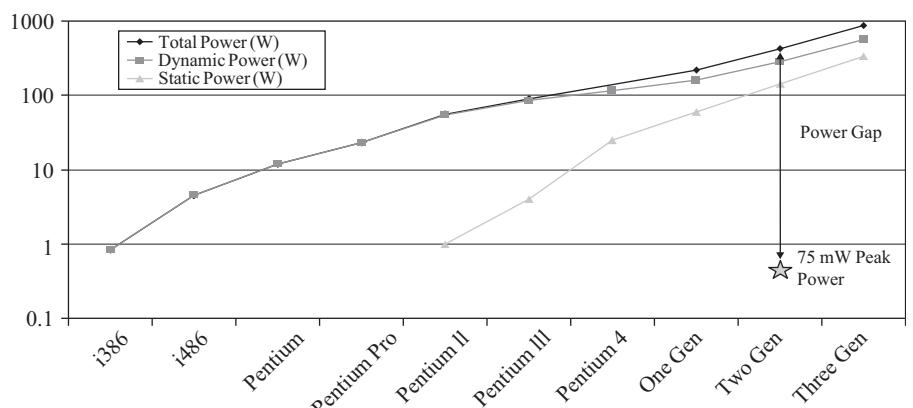
**Power**

Many of the technology trends that encourage us to use multiprocessors for performance also lead us to multiprocessing for low-power embedded computing. Several processors running at slower clock rates consume less power than a single large processor; performance scales linearly with power supply voltage, but power scales with  $V^2$ .

Austin et al. [Aus04] showed that general-purpose computing platforms aren't keeping up with the strict energy budgets of battery-powered embedded computing.

**FIGURE 10.1**

Scheduling overhead is paid for at a nonlinear rate.

**FIGURE 10.2**

Power consumption trends for desktop processors [Aus04] 2004 IEEE Computer Society.

**Fig. 10.2** compares the performance of the power requirements of desktop processors with available battery power. Batteries can provide only about 75 mW of power. Desktop processors require close to 1000 times the amount of power to run. This huge gap cannot be solved by tweaking processor architectures or software. Multiprocessors provide a way to break through this power barrier and build substantially more efficient embedded computing platforms.

### 10.3 Categories of multiprocessors

Multiprocessors in general-purpose computing have a long and rich history. Embedded multiprocessors have been widely deployed for several decades. The range of embedded multiprocessor implementations is also impressively broad. Multiprocessing has been used both for relatively low-performance systems and to achieve very high levels of real-time performance at very low energy levels.

#### Shared memory vs. message passing

There are two major types of multiprocessor architectures, as illustrated in Fig. 10.3:

- **Shared memory** systems have a pool of processors ( $P_1, P_2, \dots$ ) that can read and write a collection of memories ( $M_1, M_2, \dots$ ).
- **Message passing** systems have a pool of processors that can send messages to each other. Each processor has its own local memory.

Both shared memory and message-passing machines use an interconnection network; the details of these may vary considerably. These two types are functionally equivalent. We can turn a program written for one style of machine into an equivalent program for the other style. We may choose to build one or the other based on a variety of considerations, including performance, cost, and so on.

#### System-on-chip vs. distributed

The shared memory *vs.* message-passing distinction doesn't tell us everything we would like to know about a multiprocessor. The physical organization of the processing elements and memory play a large role in determining the characteristics of the system. We have already seen in Chapter 4 single-chip microcontrollers that include the processor, memory, and I/O devices. A multiprocessor system-on-chip (MPSoC) [Wol08B] is a system-on-chip with multiple processing elements. In contrast, we use the term **distributed system** for a multiprocessor in which the processing elements are physically separated. In general, the networks used for MPSoCs will be fast

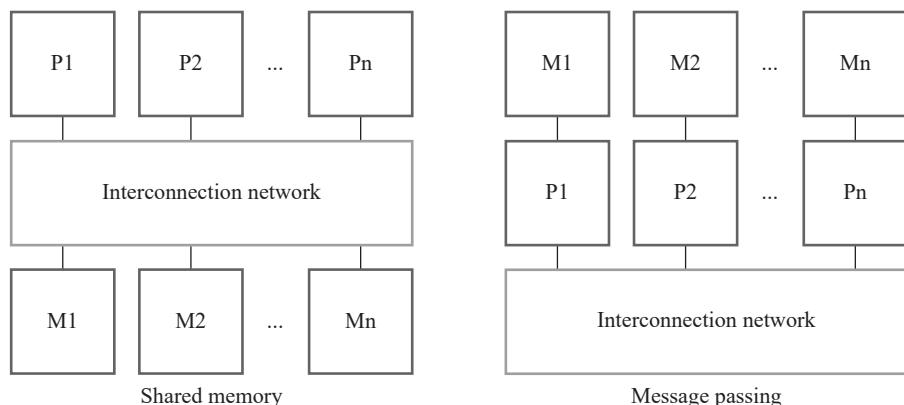


FIGURE 10.3

The two major multiprocessor architectures.

and will provide lower-latency communication between the PEs. The networks for distributed systems give higher latencies than are possible on a single chip, but many embedded systems require us to use multiple chips that may be physically very far apart. The differences in latencies between MPSOCs and distributed systems influence the programming techniques used for each.

### MPSOCs

Shared memory systems are very common in single-chip embedded multiprocessors. Shared memory multiprocessors show up in low-cost systems [Section 10.6](#). They also appear in higher-cost, high-performance systems. Shared memory systems offer relatively fast access to shared memory.

The next example describes a heterogeneous multicore embedded processor for smartphones, The Apple A15.

---

#### Example 10.1: Apple A15

Apple A15 [Fru21] is an SoC for smartphones. The CPU cluster includes two high-performance cores and four high-efficiency cores. Two variants of the chip offer somewhat different GPUs: four cores in one variant and five cores in the other. Accelerators include video encoding and decoding, an image signal processor, a display engine, and a neural engine.

---

---

## 10.4 MPSOCs and shared memory multiprocessors

Shared memory processors are well suited to applications that require a large amount of data to be processed. Signal processing systems stream data and can be well suited for shared memory processing. Most MPSOCs are shared memory systems.

Shared memory allows processors to communicate with varying patterns. If the pattern of communication is very fixed and if the processing of different steps is performed in different units, then a networked multiprocessor may be most appropriate. If the communication patterns between steps can vary, then shared memory provides flexibility. If one processing element is used for several different steps, then shared memory also allows for the required flexibility in communication.

### 10.4.1 Heterogeneous shared memory multiprocessors

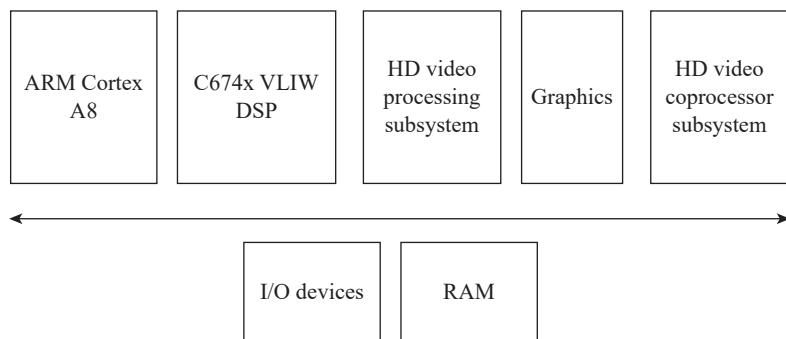
Many high-performance embedded platforms are heterogeneous multiprocessors. Different processing elements perform different functions. PEs may be programmable processors with different instruction sets or specialized accelerators that provide little or no programmability. In both cases, the motivation for using different types of PEs is efficiency. Processors with different instruction sets can perform different tasks faster and use less energy. Accelerators provide even faster and lower-power operations for a narrow range of functions.

The next example studies the TI TMS320DM816x DaVinci digital media processor.

---

**Example 10.2: TI TMS320DM816x DaVinci**

DaVinci 816x [Tex11, Tex11B] was designed for high-performance video applications. It includes a CPU, a digital signal processor (DSP), and several specialized units:



The 816x has two main programmable processors. The Arm Cortex A8 includes Neon multimedia instructions. It is an in-order dual-issue machine. The C674x is a very long instruction word DSP. It has six arithmetic logic units and 64 general-purpose registers.

The HD video coprocessor subsystem (HDVICP2) provides image and video acceleration. It natively supports several standards, such as H.264 (used in BluRay), MPEG-4, MPEG-2, and JPEG. It includes specialized hardware for major image and video operations, including transform and quantization, motion estimation, and entropy coding. It also has its own DMA engine. It can operate at resolutions up to 1080P/I at 60 fps. The HD video processing subsystem provides additional video processing capabilities. It can process up to three high-definition and one standard-definition video stream simultaneously. It can perform operations, such as scan rate conversion, chromakey, and video security. The graphics unit is designed for 3D graphics operations that can process up to 30 M triangles/s.

---

#### 10.4.2 Accelerators

One important category of PEs for embedded multiprocessors is the **accelerator**. Accelerators can provide large performance increases for applications with **computational kernels** that spend a great deal of time on a small section of code. Accelerators can also provide critical speedups for low-latency I/O functions.

The design of accelerated systems is one example of **hardware/software co-design**—the simultaneous design of hardware and software to meet system objectives. Thus far, we have taken the computing platform as a given; by adding accelerators, we can customize the embedded platform to better meet our application’s demands.

As illustrated in Fig. 10.4, a CPU accelerator is attached to the CPU bus. The CPU is often called the **host**. The CPU talks to the accelerator through the data and control registers in the accelerator. These registers allow the CPU to monitor the accelerator’s operation and give the accelerator commands.

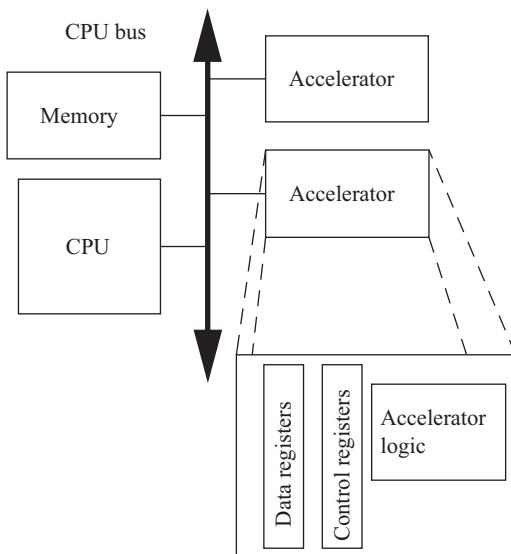


FIGURE 10.4

CPU accelerators in a system.

The CPU and accelerator may also communicate via shared memory. If the accelerator needs to operate on a large volume of data, it is usually more efficient to leave the data in memory and have the accelerator read and write memory directly, rather than to have the CPU shuttle data from memory to accelerator registers and back. The CPU and accelerator use synchronization mechanisms to ensure that they do not destroy each other's data.

An accelerator is not a co-processor. A co-processor is connected to the internals of the CPU and processes instructions. An accelerator interacts with the CPU through the programming model interface; it does not execute instructions. Its interface is functionally equivalent to an I/O device, although it usually does not perform input or output.

The first task in designing an accelerator is to determine that our system actually needs one. We must make sure that the function we want to accelerate will run more quickly on our accelerator than it will execute as software on a CPU. If our system CPU is a small microcontroller, the race may easily be won, but competing against a high-performance CPU is a challenge. We must also make sure that the accelerated function will speed up the system. If some other operation is in fact the bottleneck, or if moving data into and out of the accelerator is too slow, then adding the accelerator may not be a net gain.

Once we have analyzed the system, we need to design the accelerator itself. To identify our need for an accelerator, we must have a good understanding of the algorithm to be accelerated, which is often in the form of a high-level language program.

We must translate the algorithm description into a hardware design—a considerable task. We must also design the interface between the accelerator core and the CPU bus. The interface includes more than bus handshaking logic. For example, we must determine how the application software on the CPU will communicate with the accelerator and provide the required registers; we may have to implement shared memory synchronization operations; and we may have to add address generation logic to read and write large amounts of data from the system memory.

Finally, we will have to design the CPU-side interface for the accelerator. The application software will have to talk to the accelerator, providing it data, and telling it what to do. We have to somehow synchronize the operation of the accelerator with the rest of the application so that the accelerator knows when it has the required data, and the CPU knows when it has received the desired results.

Field-programmable gate arrays (FPGAs) provide a useful platform for custom accelerators. An FPGA has a **fabric** with both programmable logic gates and programmable interconnects that can be configured to implement a specific function. Most FPGAs also provide on-board memory that can be configured with different ports for custom memory systems. Some FPGAs provide on-board CPUs to run software that can talk to the FPGA fabric. Small CPUs can also be implemented directly in the FPGA fabric; the instruction sets of these processors can be customized for the required function.

The next example describes an MPSoC with both an on-board multiprocessor and an FPGA fabric.

---

### Example 10.3: Xilinx Zynq UltraScale+ MPSoCs

The Xilinx Zynq UltraScale+ family (<http://www.xilinx.com>) combines a multiprocessor, FPGA fabric, memory, and other system components. The chips include both a quad-core Arm Cortex-A53 and a dual-core Arm Cortex-R5, as well as a Mali graphics unit. A variety of dynamic and static memory interfaces is provided. I/O devices include PCIe, SATA, USB, CAN, SPI, and GPIO. Several security units are provided. The chips also include an array of combinational logic blocks and block RAM.

---

#### 10.4.3 Accelerator performance analysis

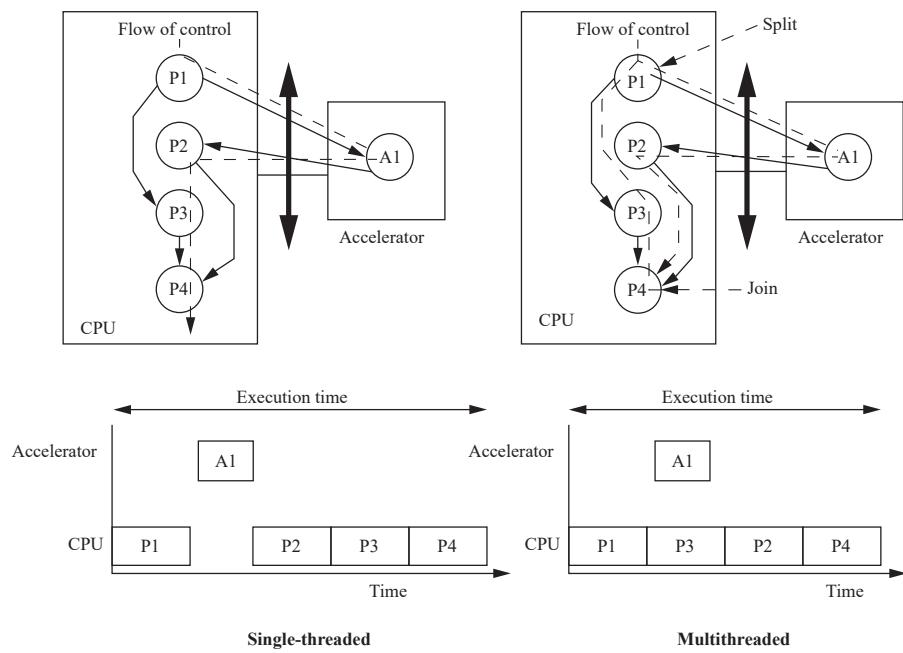
In this section, we are most interested in **speedup**: How much faster is the system with the accelerator than the system without it? We may, of course, be concerned with other metrics, such as power consumption and manufacturing cost. However, if the accelerator doesn't provide an attractive speedup, questions of cost and power will be moot.

Performance analysis of an accelerated system is a more complex task than what we have done thus far. In [Chapter 6](#), we found that performance analysis of a CPU with multiple processes was more complex than the analysis of a single program.

When we have multiple processing elements, performance analysis becomes even more difficult.

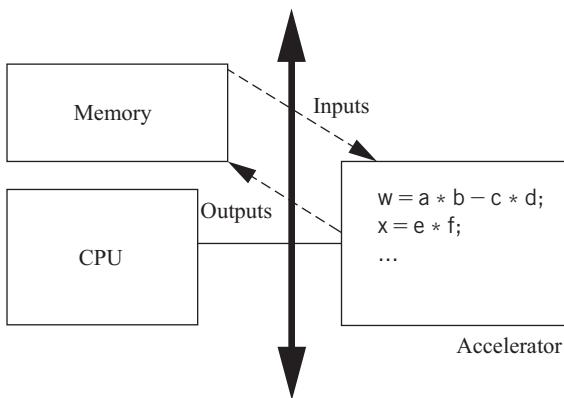
The speedup factor depends in part on whether the system is **single-threaded** or **multithreaded**, that is, whether the CPU sits idle while the accelerator runs in the single-threaded case, or the CPU can do useful work in parallel with the accelerator in the multithreaded case. Another equivalent description is **blocking** vs. **nonblocking**. Does the CPU's scheduler block other operations and wait for the accelerator call to complete, or does the CPU allow some other process to run in parallel with the accelerator? The possibilities are shown in Fig. 10.5. Data dependencies allow  $P_2$  and  $P_3$  to run independently on the CPU, but  $P_2$  relies on the results of the  $A_1$  process implemented by the accelerator. However, in the single-threaded case, the CPU blocks to wait for the accelerator to return the results of its computation. As a result, it doesn't matter whether  $P_2$  or  $P_3$  runs next on the CPU. In the multithreaded case, the CPU continues to do useful work while the accelerator runs, so the CPU can start  $P_3$  just after starting the accelerator and finishing the task earlier.

The first task is to analyze the performance of the accelerator. As illustrated in Fig. 10.6, the execution time for the accelerator depends on more than just the time required to execute the accelerator's function. It also depends on the time required to get the data into the accelerator and back out of it.



**FIGURE 10.5**

Single-threaded vs. multithreaded control of an accelerator.

**FIGURE 10.6**

Components of execution time for an accelerator.

Because the CPU's registers are probably not addressable by the accelerator, the data probably reside in the main memory.

#### Accelerator execution time

A simple accelerator reads all its input data, performs the required computation, and then, writes all its results. In this case, the total execution time may be written as

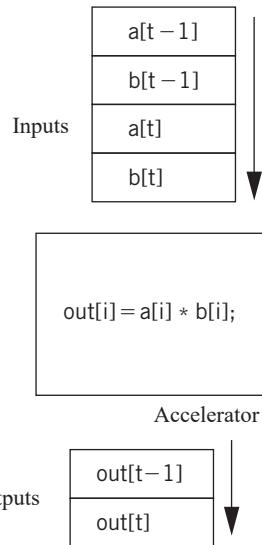
$$t_{\text{accel}} = t_{in} + t_x + t_{out} \quad (\text{Eq. 10.1})$$

where  $t_x$  is the execution time of the accelerator assuming all data are available, and  $t_{in}$  and  $t_{out}$  are the times required for reading and writing the required variables, respectively. The values for  $t_{in}$  and  $t_{out}$  must reflect the time required for bus transactions, including two factors:

- the time required to flush any register or cache values to the main memory, if those values are needed in the main memory to communicate with the accelerator; and
- the time required for the transfer of control between the CPU and accelerator.

Transferring data into and out of the accelerator may require the accelerator to become a bus controller. Because the CPU may delay bus controllership requests, some worst-case values for bus controllership acquisition must be determined based on the CPU characteristics.

A more sophisticated accelerator could try to overlap input and output with computation. For example, it could read a few variables and start computing on those values, while reading other values in parallel. In this case, the  $t_{in}$  and  $t_{out}$  terms would represent the nonoverlapped read/write times rather than the complete input and output times. One important example of overlapped I/O and computation is streaming data applications, such as digital filtering. As illustrated in Fig. 10.7, an accelerator may take in one or more streams of data and output a stream. A typical stream is sufficiently large that it cannot be fetched at once; a series of fetches and stores is required. Latency requirements generally require that outputs be produced on the

**FIGURE 10.7**

Streaming data into and out of an accelerator.

fly rather than storing all the data and then computing; furthermore, it may be impractical to store long streams at all. In this case, the  $t_{in}$  and  $t_{out}$  terms are determined by the amount of data read before starting computation and the length of time between the last computation and the last data output.

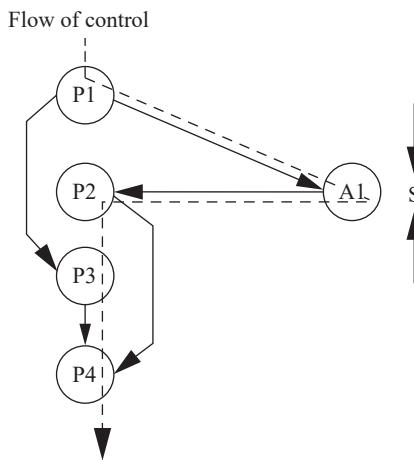
We are most interested in the speedup obtained by replacing the software implementation with the accelerator. The total speedup  $S$  for a kernel can be written as [Hen94]:

$$S = n(t_{CPU} - t_{accel}) = n[t_{CPU} - (t_{in} + t_x + t_{out})] \quad (\text{Eq. 10.2})$$

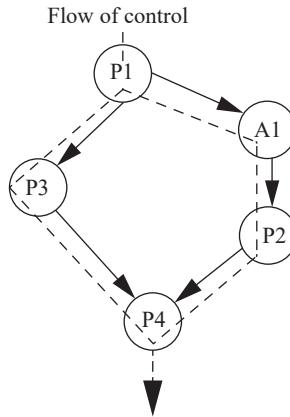
where  $t_{CPU}$  is the execution time of the equivalent function in the software on the CPU, and  $n$  is the number of times the function will be executed. We can use the techniques described in [Chapter 5](#) to determine the value of  $t_{CPU}$ . Clearly, the more times the function is evaluated, the more valuable the speedup provided by the accelerator becomes.

**System speedup** Ultimately, we don't care as much about the accelerator's speedup as the speedup for the complete system—how much faster is the entire application's execution? In a single-threaded system, the evaluation of the accelerator's speedup to the total system speedup is simple: the system execution time is reduced by  $S$ . The reason is illustrated in [Fig. 10.8](#); the single thread of control gives us a single path whose length we can measure to determine the new execution speed.

Evaluating system speedup in a multithreaded environment requires more subtlety. As shown in [Fig. 10.9](#), there is now more than one execution path. The total system

**FIGURE 10.8**

Evaluating system speedup in a single-threaded implementation.

**FIGURE 10.9**

Evaluating system speedup in a multithreaded implementation.

execution time depends on the **longest path**, from the beginning of execution to the end of execution. In this case, the system execution time depends on the relative speeds of  $P_3$  and  $P_2$  plus  $A_1$ . If  $P_2$  and  $A_1$  together take the most time,  $P_3$  will not play a role in determining the system execution time. If  $P_3$  takes longer, then  $P_2$  and  $A_1$  will not be factors. To determine the system execution time, we must label each node in the graph with its execution time. In simple cases, we can enumerate the paths, measure the length of each, and select the longest one as the system execution time. Efficient graph algorithms can also be used to compute the longest path.

This analysis shows the importance of selecting the proper functions to be moved to the accelerator. Clearly, if the function selected for speedup isn't a big portion of system execution time, taking the number of times it is executed into account, you won't see much system speedup. We also learned from [Equation 10.2](#) that, if too much overhead is incurred in getting data into and out of the accelerator, we won't see much speedup.

#### 10.4.4 Scheduling and allocation

When designing a distributed embedded system, we must deal with the **scheduling** and **allocation** design problems described below.

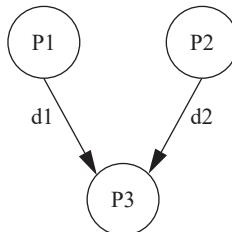
- We must *schedule* operations in time, including communication on the network and computations on the processing elements. The scheduling of operations on the PEs and the communications between the PEs are linked. If one PE finishes its computations too late, it may interfere with another communication on the network, as it tries to send its result to the PE that needs it. This is bad for both the PE that needs the results and the other PEs whose communication is interfered with.
- We must *allocate* computations to the processing elements. The allocation of computations to the PEs determines what communications are required; if a value computed on one PE is needed on another PE, it must be transmitted over the network.

Example 10.4 illustrates scheduling and allocation in accelerated embedded systems.

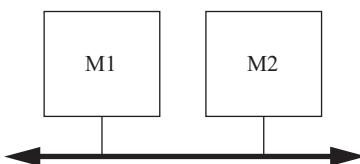
---

#### Example 10.4: Scheduling and Allocating Processes on a Distributed Embedded System

We can specify the system as a task graph. However, different processes may result in different PEs. Here is a task graph:



We have labeled the data transmissions on each arc so that we can refer to them later. We want to execute the task on this platform:



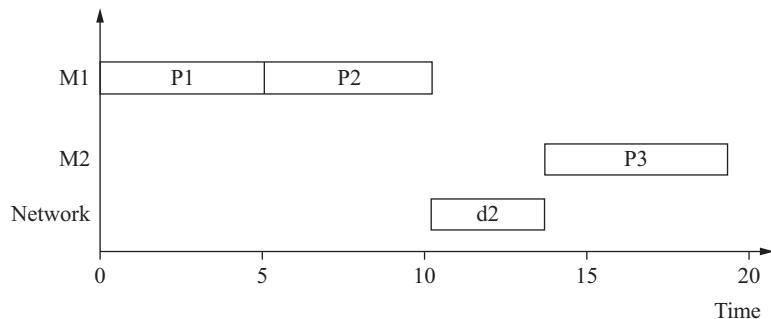
The platform has two PEs and a single bus connecting both PEs. To make decisions about where to allocate and when to schedule processes, we need to know how fast each process runs on each PE. Here are the process speeds:

	<b>M1</b>	<b>M2</b>
P1	5	5
P2	5	6
P3	—	5

The dash (—) entry signifies that the process cannot run on that type of processing element. In practice, a process may be excluded from some PEs for several reasons. If we use an ASIC to implement a special function, it will be able to implement only one process. A small CPU, such as a microcontroller, may not have enough memory for the process's code or data; it may also simply run too slowly to be useful. A process may run at different speeds on different CPUs for many reasons. Even when the CPUs run at the same clock rate, differences in the instruction sets can cause a process to be better suited to a particular CPU.

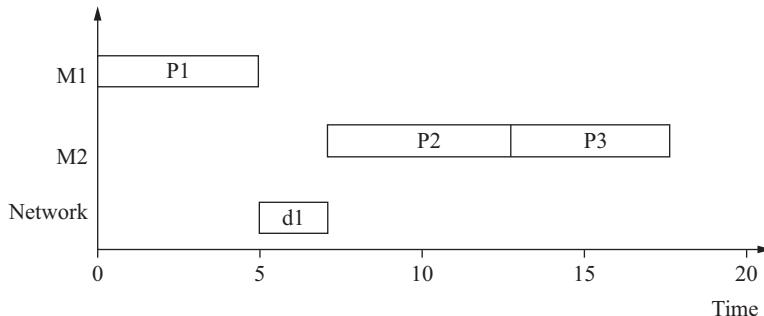
If two processes are allocated to the same PE, they can communicate using the processing element's internal memory and incur no network communication time. Each edge in the task graph corresponds to a data communication that must be carried over the network. Because all PEs communicate at the same rate, the data communication rate is the same for all transmissions between PEs. We need to know how long each communication takes. In this case,  $d_1$  is a short message requiring two time units, and  $d_2$  is a longer communication requiring four time units.

As an initial design, let us allocate  $P_1$  and  $P_2$  to  $M_1$  and  $P_3$  to  $M_2$ . This allocation would, on the surface, appear to be a good one, because  $P_1$  and  $P_2$  are both placed on the processor that runs them the fastest. This schedule shows what happens to all the PEs and the network:



The schedule has a length of 19. The  $d_1$  message is sent between the processes internal to  $P_1$  and does not appear on the bus.

Let's try a different allocation:  $P_1$  on  $M_1$  and  $P_2$  and  $P_3$  on  $M_2$ . This makes  $P_2$  run more slowly. Here is the new schedule:



The length of this schedule is 18 or one time units less than the other schedule. The increased computation time of  $P_2$  is more than made up for by being able to transmit a shorter message on the bus. If we had not taken communication into account when analyzing the total execution time, we could have made the wrong choice of which processes to put on the same PE.

### 10.4.5 System integration

The design of an accelerated system often requires combining several different types of components. Serial busses are often used for module-to-module communication, particularly for tasks such as initialization and configuration.

The **I<sup>2</sup>C bus** [Phi92] is a well-known bus commonly used to link microcontrollers and other modules into systems. It has even been used for the command interface in an MPEG-2 video chip [van97]; while a separate bus was used for high-speed video data, setup information was transmitted to the on-chip controller through an I<sup>2</sup>C bus interface.

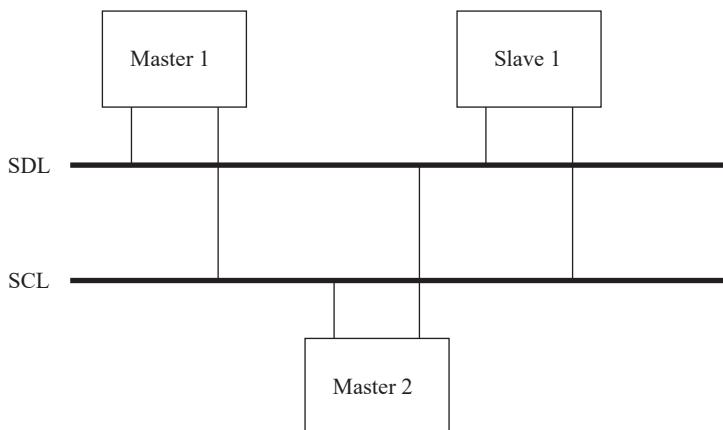
I<sup>2</sup>C is designed to be low cost, easy to implement, and of moderate speed (up to 100 kbit/s for the standard bus and up to 400 kbit/s for the extended bus). As a result, it uses only two lines: the **serial data line (SDL)** for data and the **serial clock line (SCL)**, which indicates when valid data are on the data line. Fig. 10.10 shows the structure of a typical I<sup>2</sup>C bus system. Every node in the network is connected to both SCL and SDL. Some nodes may be able to act as bus controllers, and the bus may have more than one controller. Other nodes may act as responders that respond only to requests from controllers.

The basic electrical interface to the bus is shown in Fig. 10.11. The bus does not define particular voltages to be used for high or low, so that either bipolar or MOS circuits can be connected to the bus. Both bus signals use open collector/open drain circuits.<sup>1</sup> A pull-up resistor keeps the default state of the signal high, and transistors

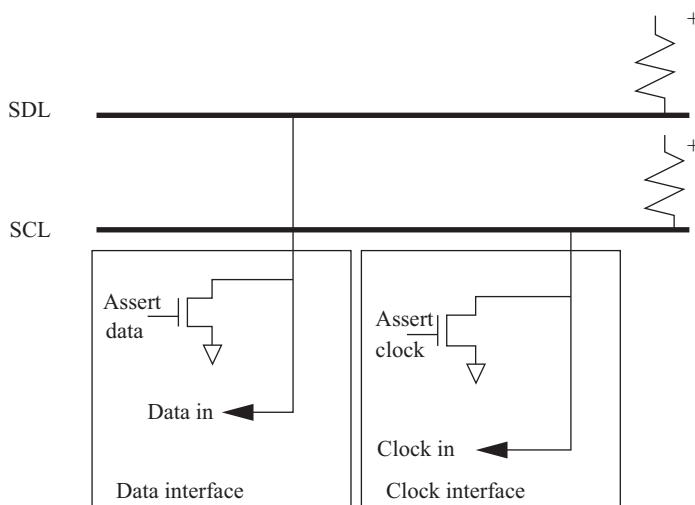
#### Physical layer

#### Electrical interface

<sup>1</sup>An open collector uses a bipolar transistor, while an open drain circuit uses an MOS transistor

**FIGURE 10.10**

Structure of an I<sup>2</sup>C bus system.

**FIGURE 10.11**

Electrical interface to the I<sup>2</sup>C bus.

are used in each bus device to pull down the signal when a zero is to be transmitted. Open collector/open drain signaling allows several devices to simultaneously write the bus without causing electrical damage.

The open collector/open drain circuitry allows a responder device to stretch a clock signal during a read from a responder. The controller handles generating the SCL clock, but the responder can stretch the low period of the clock (but not the high period) if necessary.

The I<sup>2</sup>C bus is designed as a multicontroller bus; any one of several different devices may act as the controller at various times. As a result, there is no global controller to generate the clock signal on the SCL. Instead, a controller drives both the SCL and SDL when it is sending data. When the bus is idle, both the SCL and SDL remain high. When two devices try to drive either SCL or SDL to different values, the open collector/open drain circuitry prevents errors, but each controller device must listen to the bus while transmitting to be sure that it is not interfering with another message. If the device receives a different value than it is trying to transmit, then it knows that it is interfering with another message.

#### Data link layer

Every I<sup>2</sup>C device has an address. The system designer determines the addresses of the devices, usually as part of the program for the I<sup>2</sup>C driver. The addresses must, of course, be chosen so that no two devices in the system have the same address. A device address is 7 bits in the standard I<sup>2</sup>C definition (the extended I<sup>2</sup>C allows 10-bit addresses). Address 0000000 is used to signal a **general call** or bus broadcast, which can be used to signal all devices simultaneously. Address 11110XX is reserved for the extended 10-bit addressing scheme; there are several other reserved addresses as well.

A **bus transaction** comprises a series of one-byte **transmissions** and an address followed by one or more data bytes. I<sup>2</sup>C encourages a data-push programming style. When a controller wants to write a responder, it transmits the responder's address, followed by the data. Because a responder cannot initiate a transfer, the controller must send a read request with the responder's address and let the responder transmit the data. Therefore, an address transmission includes the 7-bit address and one bit for data direction: 0 for writing from the controller to the responder and 1 for reading from the responder to the controller. (This explains the 7-bit addresses on the bus.) The format of the address transmission is shown in Fig. 10.12.

A bus transaction is initiated by a start signal and completed with an end signal:

- A start is signaled by leaving the SCL high and sending a one-to-zero transition on SDL.
- A stop is signaled by setting the SCL high and sending a zero-to-one transition on SDL.

However, starts and stops must be paired. A controller can write and then read (or read and then write) by sending a start after the data transmission, followed by another address transmission and then more data. The basic state transition graph for the controller's actions in a bus transaction is shown in Fig. 10.13.

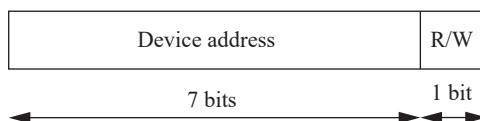
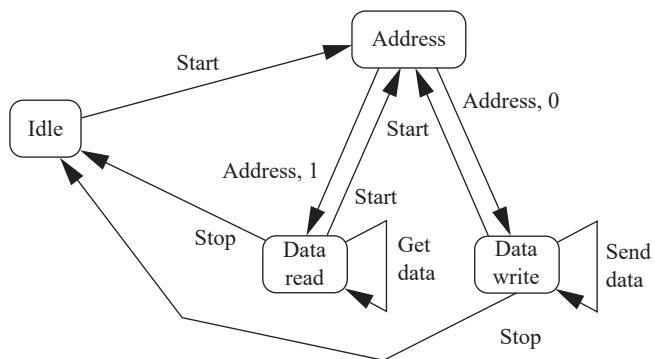


FIGURE 10.12

Format of an I<sup>2</sup>C address transmission.

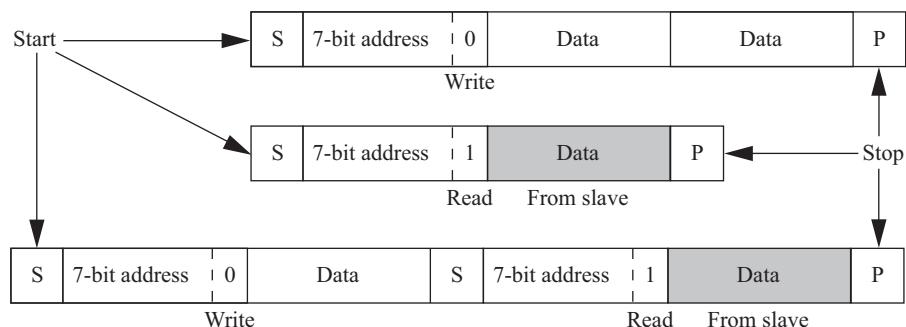
**FIGURE 10.13**

State transition graph for an I<sup>2</sup>C bus controller.

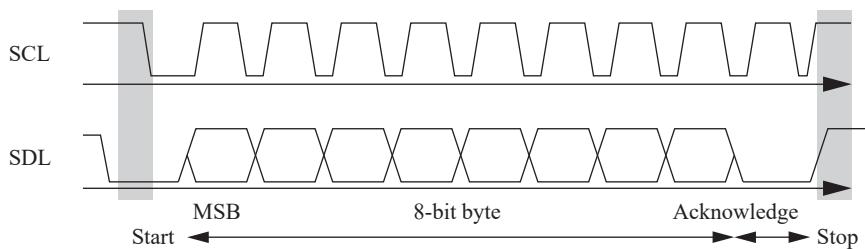
The formats of some typical complete bus transactions are shown in Fig. 10.14. In the first example, the controller writes two bytes to the addressed responder. In the second, the controller requests a read from a responder. In the third, the controller writes one byte to the responder, and then, sends another start to initiate a read from the responder.

#### Byte format

Fig. 10.15 shows how a data byte is transmitted on the bus, including start and stop events. The transmission starts when SDA is pulled low while SCL remains high. After this start condition, the clock line is pulled low to initiate the data transfer. At each bit, the clock line goes high, while the data line assumes its proper value of 0 or 1. An acknowledgment is sent at the end of every 8-bit transmission, whether it is an address or data. For acknowledgment, the transmitter does not pull down the SDA, allowing the receiver to set the SDA to zero if it properly receives the byte. After

**FIGURE 10.14**

Typical bus transactions on the I<sup>2</sup>C bus.

**FIGURE 10.15**

Transmitting a byte on the I<sup>2</sup>C bus.

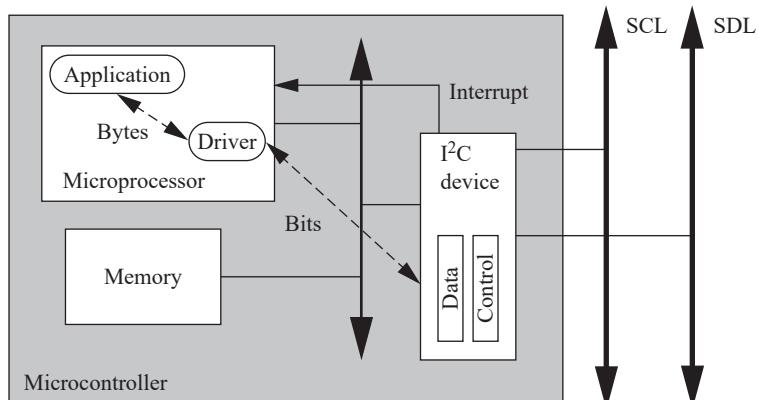
the acknowledgment, the SDL goes from low to high, while the SCL is high, signaling the stop condition.

#### Bus arbitration

The bus uses this feature to arbitrate each message. When sending, devices listen to the bus as well. If a device is trying to send logic 1 but hears a logic 0, it immediately stops transmitting and gives the other sender priority. The devices should be designed so that they can stop transmitting in time to allow a valid bit to be sent. In many cases, arbitration will be completed during the address portion of a transmission, but arbitration may continue into the data portion. If two devices are trying to send identical data to the same address, then of course they never interfere, and both succeed in sending their message. This form of arbitration is like the CAN bus arbitration seen in [Section 10.3.1](#).

#### Application interface

The I<sup>2</sup>C interface on a microcontroller can be implemented with varying percentages of functionality in software and hardware [Phi89]. As illustrated in [Fig. 10.16](#), a typical system has a one-bit hardware interface with routines for byte-level functions. The I<sup>2</sup>C device takes care to generate the clock and data. The application code calls

**FIGURE 10.16**

An I<sup>2</sup>C interface in a microcontroller.

for routines to send an address, send a data byte, and so on, which then generates the SCL and SDL, acknowledges, and so forth. One of the microcontroller's timers is typically used to control the length of bits on the bus. Interrupts may be used to recognize bits. However, when used in controller mode, polled I/O may be acceptable if no other pending tasks can be performed because controllers initiate their own transfers.

### 10.4.6 Debugging

It is generally good policy to separately debug the basic interface between the accelerator and the rest of the system before integrating the full accelerator into the platform.

Hardware/software co-simulation can be effective in accelerator design. Because the co-simulator allows you to run software relatively efficiently alongside a hardware simulation, it allows you to exercise the accelerator in a realistic but simulated environment. It is especially difficult to exercise the interface between the accelerator core and the host CPU without running the CPU's accelerator driver. It is much better to do so in a simulator before fabricating the accelerator rather than to have to modify the hardware prototype of the accelerator.

## 10.5 Design example: video accelerator

In this section, we consider the design of a video accelerator, specifically a motion estimation accelerator. Digital video is a computationally intensive task, so it is well suited to acceleration. Motion estimation engines are used in real-time search engines; we may want to have one attached to our PC to experiment with video processing techniques.

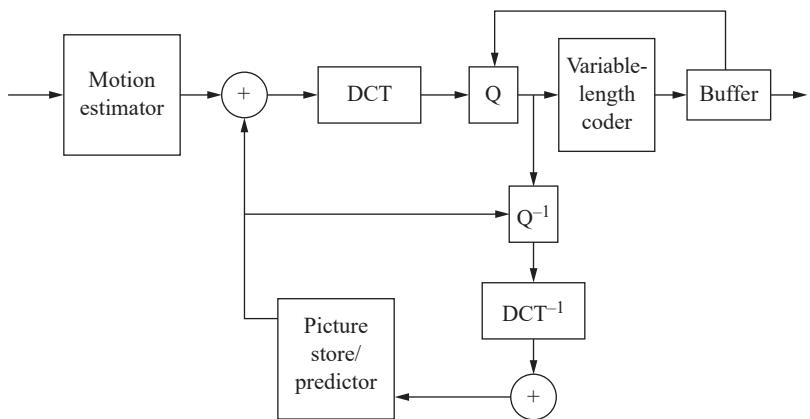
### 10.5.1 Video compression

Before examining the video accelerator itself, let's look at video compression algorithms to understand the role played by a motion estimation engine.

[Fig. 10.17](#) shows a block diagram for MPEG-2 video compression [Has97]. MPEG-2 forms the basis for U.S. HDTV broadcasting. This compression uses several component algorithms together in a feedback loop. The discrete cosine transform (DCT) used in JPEG also plays a key role in MPEG-2. As in still image compression, the DCT of a block of pixels is quantized for lossy compression, and then, subjected to lossless variable-length coding to further reduce the number of bits required to represent the block.

However, JPEG-style compression alone does not sufficiently reduce video bandwidth for many applications. MPEG uses motion to encode one frame in terms of another. Rather than sending each frame separately, as in motion JPEG, some frames are sent as modified forms of other frames using a technique known as **block motion estimation**. During encoding, the frame is divided into **macroblocks**. Macroblocks

#### Motion-based coding

**FIGURE 10.17**

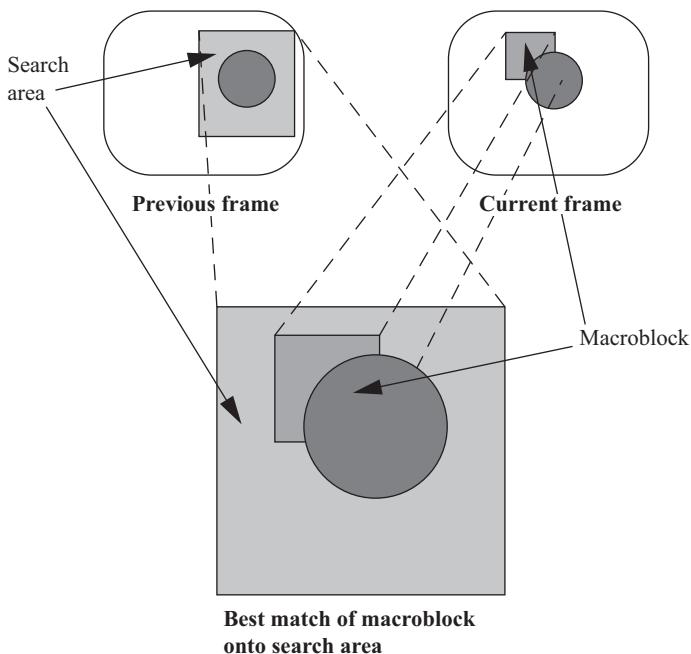
Block diagram of the MPEG-2 compression algorithm.

from one frame are identified in other frames using correlation. The frame can then be encoded using a vector that describes the motion of the macroblock from one frame to another without explicitly transmitting all the pixels. As shown in Fig. 10.17, the MPEG-2 encoder also uses a feedback loop to further improve image quality. This form of coding is lossy, and several different conditions can cause prediction to be imperfect: objects within a macroblock may move from one frame to the next or a macroblock may not be found by the search algorithm, etc. The encoder uses the encoding information to recreate the lossily-encoded picture, compares it to the original frame, and generates an error signal that can be used by the receiver to fix smaller errors. The decoder must keep some recently decoded frames in the memory so that it can retrieve the pixel values of the macroblocks. This internal memory saves a great deal of transmission and storage bandwidth.

The concept of block motion estimation is illustrated in Fig. 10.18. The goal is to perform a two-dimensional correlation to find the best match between the regions in the two frames. We divide the current frame into  $16 \times 16$  macroblocks. For every macroblock in the frame, we want to find the region in the previous frame that most closely matches the macroblock. Searching over the entire previous frame would be too expensive, so we usually limit the search to a given area, centered around the macroblock and larger than the macroblock. We try the macroblock at various offsets in the search area. We measure similarity using the following sum-of-differences measure:

$$\sum_{1 \leq i,j \leq n} |M(i,j) - S(i - o_x, j - o_y)| \quad (\text{Eq. 10.3})$$

where  $M(i,j)$  is the intensity of the macroblock at pixel  $i,j$ ,  $S(i,j)$  is the intensity of the search region,  $n$  is the size of the macroblock in one dimension, and  $\langle o_x, o_y \rangle$  is the

**FIGURE 10.18**


---

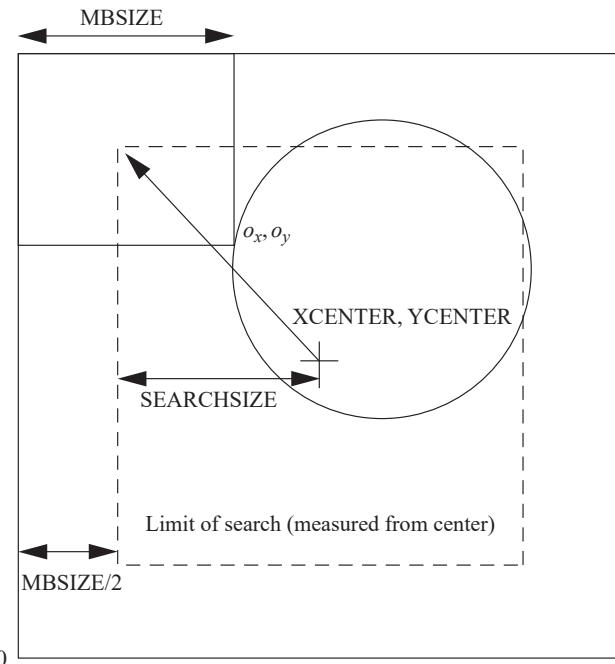
Block motion estimation.

offset between the macroblock and the search region. Intensity is measured as an 8-bit luminance that represents a monochrome pixel; color information is not used in motion estimation. We choose the macroblock position relative to the search area that gives us the smallest value for this metric. The offset at this chosen position describes a vector from the search area center to the macroblock's center, which is called the **motion vector**.

### 10.5.2 Algorithm and requirements

For simplicity, we will build an engine for a full search that compares the macroblock and search area at every possible point. Because this is an expensive operation, several methods have been proposed for conducting a sparser search of the search area. More advanced algorithms are used in practice. We choose a full-motion search here to concentrate on some basic issues in the relationship between the accelerator and the rest of the system.

A good way to describe the algorithm is in C. Some basic parameters of the algorithm are illustrated in Fig. 10.19. The image being searched includes some features such as a circle; in this case, parts of the circle lie outside the search area. Here is the C

**FIGURE 10.19**

Block motion search parameters.

code for a single search, which assumes that the search region does not extend past the boundary of the frame.

```

bestx = 0; besty = 0; /*initialize best location--none yet */
bestsad = MAXSAD; /*best sum-of--difference thus far */
for (ox = -SEARCHSIZE; ox < SEARCHSIZE; ox++) {
    /*x search ordinate */
    for (oy = -SEARCHSIZE; oy < SEARCHSIZE; oy++) {
        /*y search ordinate */
        int result = 0;
        for (i = 0; i < MBSIZE; i++) {
            for (j = 0; j < MBSIZE; j++) {
                result = result + iabs(mb[i][j] -
                    search[i - ox + XCENTER][j - oy + YCENTER]);
            }
        }
        if (result <= bestsad) { /* found better match */
            bestsad = result;
            bestx = ox; besty = oy;
        }
    }
}

```

The arithmetic for each pixel is simple, but we must process a lot of pixels. If MBSIZE is 16 and SEARCHSIZE is eight, and remembering that the search distance in each dimension is  $8 + 1 + 8$ , then we must perform

$$n_{ops} = (16 \times 16) \times (17 \times 17) = 73,984 \quad (\text{Eq. 10.4})$$

difference operations to find the motion vector for a single macroblock, which requires looking at twice as many pixels; one from the search area and one from the macroblock. We can now see an interest in algorithms that do not require a full search. To process the video, we will have to perform this computation on every macroblock of every frame. Adjacent blocks have overlapping search areas, so we will want to avoid reloading pixels we already have.

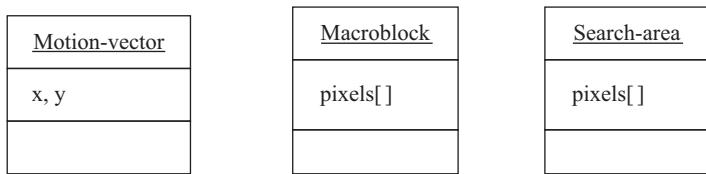
One relatively low-resolution standard video format, common intermediate format (CIF), has a frame size of  $352 \times 288$ , which gives an array of  $22 \times 18$  macroblocks. If we want to encode video, we will have to perform motion estimation on every macroblock of most frames (some frames are sent without using motion compensation).

We will build the system using an FPGA connected to the PCIe bus of a personal computer. We clearly need a high-bandwidth connection, such as the PCIe between the accelerator and the CPU. We can use the accelerator to experiment with video processing, among other things. Here are the requirements for the system:

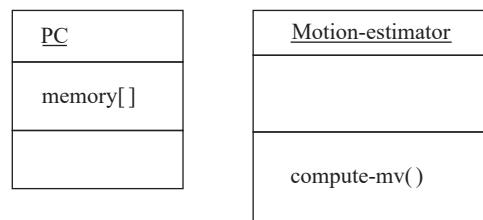
Name	Block motion estimator
Purpose	Perform block motion estimation within a PC system
Inputs	Macroblocks and search areas
Outputs	Motion vectors
Functions	Compute motion vectors using full search algorithms
Performance	As fast as we can get
Manufacturing cost	\$100
Power	Powered by PC power supply
Physical size and weight	Packaged as PCIe card for PC

### 10.5.3 Specification

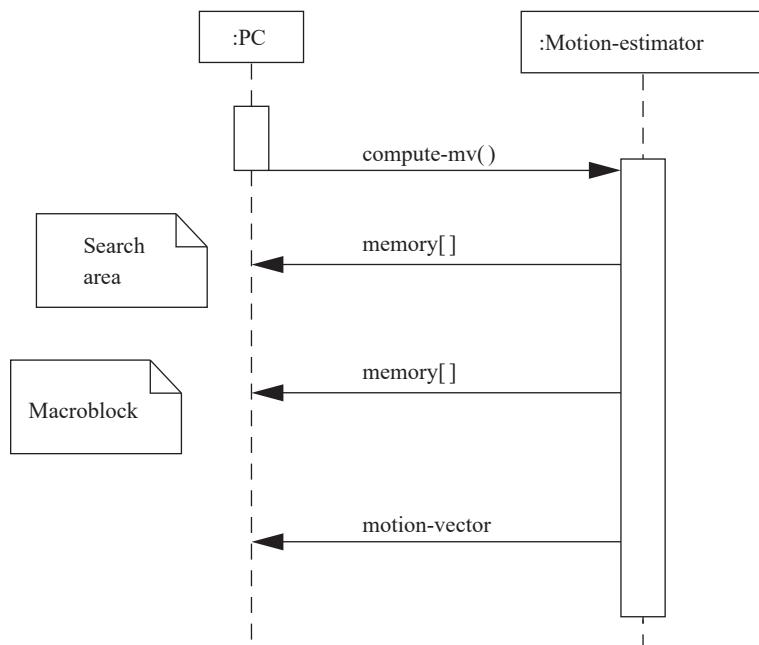
The specifications for the system are relatively straightforward because the algorithm is simple. Fig. 10.20 defines some classes that describe basic data types in the system: the motion vector, the macroblock, and the search area. These definitions are straightforward. Because the behavior is simple, we need to define only two classes to describe it: the accelerator itself and the PC. These classes are shown in Fig. 10.21. The PC makes its memory accessible to the accelerator. The accelerator provides a behavior `compute-mv()` that performs the block motion estimation algorithm. Fig. 10.22 shows a sequence diagram that describes the operation of `compute-mv()`.

**FIGURE 10.20**

Classes describing basic data types in the video accelerator.

**FIGURE 10.21**

Basic classes for the video accelerator.

**FIGURE 10.22**

Sequence diagram for the video accelerator.

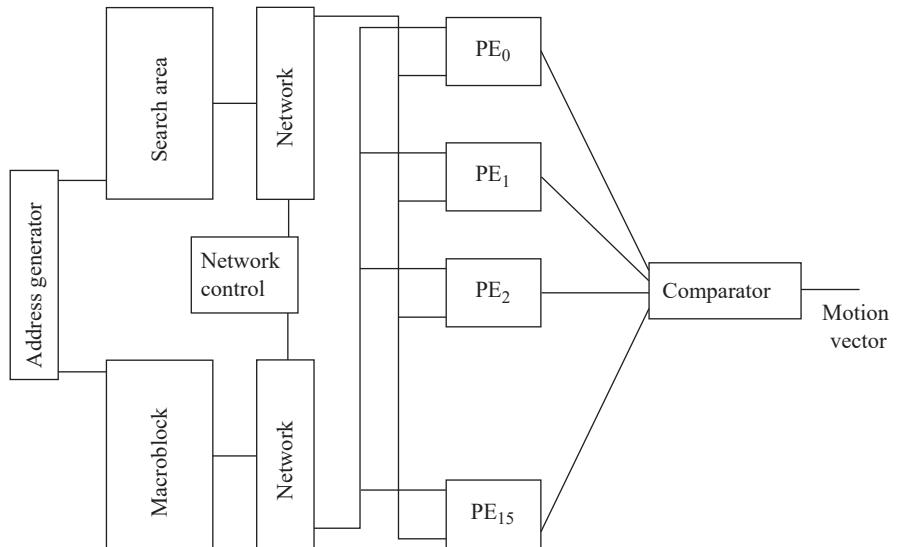
After initiating the behavior, the accelerator reads the search area and macroblock from the PC; after computing the motion vector, it returns it to the PC.

#### 10.5.4 Architecture

The accelerator will be implemented in an FPGA on a card connected to a computer's PCIe slot. Such accelerators can be purchased or can be designed from scratch. If you design such a card from scratch, you must decide early on whether the card will be used only for this video accelerator or whether it should be made general enough to support other applications as well.

The architecture for the accelerator requires some thought because of the large amount of data required by the algorithm. The macroblock has  $16 \times 16 = 256$  pixels; the search area has  $(8 + 8 + 1 + 8 + 8)2^2 = 1089$  pixels. The FPGA may not have enough memory to hold 1089 8-bit values. We have to use a memory external to the FPGA, but on the accelerator board to hold the pixels.

There are many possible architectures for motion estimators. One is shown in Fig. 10.23. The machine has two memories: one for the macroblock and another for the search memories. It has 16 PEs that perform the difference calculation on a pair of pixels; the comparator sums them up and selects the best value to find the motion vector. This architecture can be used to implement algorithms other than a full search by changing address generation and control. Depending on the number of



**FIGURE 10.23**

An architecture for the motion estimation accelerator [Dut96].

<b>t</b>	<b>M</b>	<b>S</b>	<b>S9</b>	<b>PE<sub>0</sub></b>	<b>PE<sub>1</sub></b>	<b>PE<sub>2</sub></b>
0	M(0,0)	S(0,0)		M(0,0) – S(0,0)		
1	M(0,1)	S(0,1)		M(0,1) – S(0,1)	M(0,0) – S(0,1)	
2	M(0,2)	S(0,2)		M(0,2) – S(0,2)	M(0,1) – S(0,2)	M(0,0) – S(0,2)
3	M(0,3)	S(0,3)		M(0,3) – S(0,3)	M(0,2) – S(0,3)	M(0,1) – S(0,3)
4	M(0,4)	S(0,4)		M(0,4) – S(0,4)	M(0,3) – S(0,4)	M(0,2) – S(0,4)
5	M(0,5)	S(0,5)		M(0,5) – S(0,5)	M(0,4) – S(0,5)	M(0,3) – S(0,5)
6	M(0,6)	S(0,6)		M(0,6) – S(0,6)	M(0,5) – S(0,6)	M(0,4) – S(0,6)
7	M(0,7)	S(0,7)		M(0,7) – S(0,7)	M(0,6) – S(0,7)	M(0,5) – S(0,7)
8	M(0,8)	S(0,8)		M(0,8) – S(0,8)	M(0,7) – S(0,8)	M(0,6) – S(0,8)
9	M(0,9)	S(0,9)		M(0,9) – S(0,9)	M(0,8) – S(0,9)	M(0,7) – S(0,9)
10	M(0,10)	S(0,10)		M(0,10) – S(0,10)	M(0,9) – S(0,10)	M(0,8) – S(0,10)
11	M(0,11)	S(0,11)		M(0,11) – S(0,11)	M(0,10) – S(0,11)	M(0,9) – S(0,11)
12	M(0,12)	S(0,12)		M(0,12) – S(0,12)	M(0,11) – S(0,12)	M(0,10) – S(0,12)
13	M(0,13)	S(0,13)		M(0,13) – S(0,13)	M(0,12) – S(0,13)	M(0,11) – S(0,13)
14	M(0,14)	S(0,14)		M(0,14) – S(0,14)	M(0,13) – S(0,14)	M(0,12) – S(0,14)
15	M(0,15)	S(0,15)		M(0,15) – S(0,15)	M(0,14) – S(0,15)	M(0,13) – S(0,15)
16	M(1,0)	S(1,0)	S(0,16)	M(1,0) – S(1,0)	M(0,15) – S(0,16)	M(0,14) – S(0,16)
17	M(1,1)	S(1,1)	S(0,17)	M(1,1) – S(1,1)	M(1,0) – S(1,1)	M(0,15) – S(0,17)

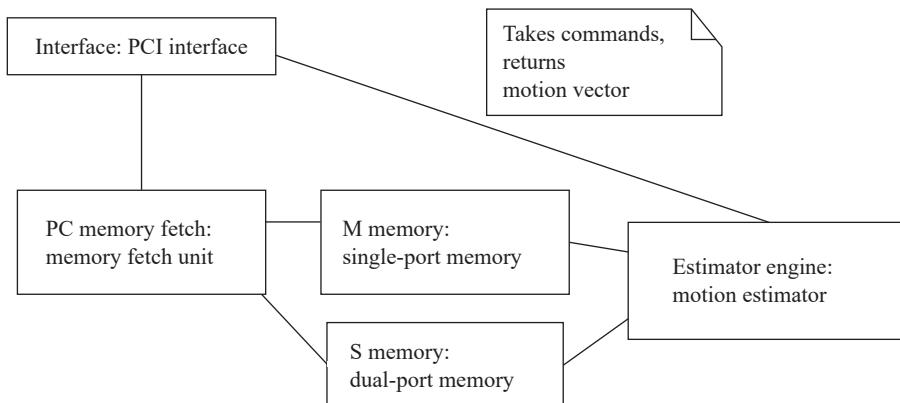
**FIGURE 10.24**

A schedule of pixel fetches for a full search [Yan89].

motion estimation algorithms that you want to execute on the machine, the networks connecting the memories to the PEs may also be simplified.

Fig. 10.24 shows how we can schedule the transfer of pixels from the memories to the processing elements to efficiently compute a full search of this architecture. The schedule fetches one pixel from the macroblock memory and (in steady state) two pixels from the search area memory per clock cycle. The pixels are distributed to the PEs in a regular pattern, as shown by the schedule. This schedule computes 16 correlations between the macroblock and the search area simultaneously. The computations for each correlation are distributed among the processing elements; the comparator handles collecting the results, finding the best match value, and remembering the corresponding motion vector.

Based on our understanding of efficient architectures for accelerating motion estimation, we can derive a more detailed definition of the architecture in a unified modeling language, which is shown in Fig. 10.25. The system includes the two memories for pixels, one a single-port memory and the other a dual-port memory. A bus

**FIGURE 10.25**

Object diagram for the video accelerator.

interface module handles communicating with the PCIe bus and the rest of the system. The estimation engine reads pixels from the *M* and *S* memories, takes commands from the bus interface, and returns the motion vector to the bus interface.

### 10.5.5 Component design

If we want to use a standard FPGA accelerator board to implement the accelerator, we must first make sure that it provides the proper memory required for *M* and *S*. Once we have verified that the accelerator board has the required structure, we can concentrate on designing FPGA logic. Designing an FPGA is, for the most part, a straightforward exercise in logic design. Because the logic for the accelerator is very regular, we can improve the FPGA's clock rate by properly placing the logic in the FPGA to reduce wire lengths.

If we are designing our own accelerator board, we must design both the video accelerator design proper and the interface to the PCIe bus. We can create and exercise the video accelerator architecture in a hardware description language like VHDL or Verilog and simulate its operation. Designing the PCIe interface requires somewhat different techniques because we may not have a simulation model for a PCIe bus. We may want to verify the operation of the basic PCIe interface before we finish implementing the video accelerator logic.

The host PC will probably deal with the accelerator as an I/O device. The accelerator board will have its own driver that is responsible for talking to the board. Because most of the data transfers are performed directly by the board using DMA, the driver can be relatively simple.

### 10.5.6 System testing

Testing video algorithms requires a large amount of data. Luckily, the data represent images and video, which are plentiful. Because we are designing only a motion estimation accelerator and not a complete video compressor, it is probably easiest to use images, not video, for test data. You can use standard video tools to extract a few frames from a digitized video and store them in JPEG format. An open source for JPEG encoders and decoders is available. These programs can be modified to read JPEG images and put out pixels in the format required by your accelerator. With a little more cleverness, the resulting motion vector can be written back onto the image for a visual confirmation of the result. If you want to be adventurous and try motion estimation on video, open-source MPEG encoders and decoders are also available.

---

## 10.6 Summary

Multiprocessors provide both absolute performance and efficiency. They do, however, introduce new levels of system complexity. Programming multiprocessors requires both new programming models and development methodologies. Multiprocessors are often heterogeneous, so that different parts of an application can be mapped to specialized PEs. A programmable PE may be specialized by, for example, adding new instructions. Accelerators are PEs designed to perform specific tasks. When adding accelerators to the system, we must be sure that the system can send data to and receive data from the rest of the system at the required rates.

---

## What we learned

- Multiprocessors can help improve real-time performance and energy consumption.
- Shared memory and message passing systems are different organizations of multiprocessors.
- MPSOCs are single-chip multiprocessors with low-latency communication while distributed systems are physically larger and generally have longer-latency communication systems.
- Shared memory multiprocessors are often used in single-chip signal processing and control systems.
- Performance analysis of an accelerated system is challenging. We must consider the performance of several implementations of an algorithm (CPU, accelerator) as well as communication costs for various configurations.
- We must partition the behavior, schedule operations in time, and allocate operations to PEs to design the system.

## Further reading

Kopetz [Kop97] provides a thorough introduction to the design of distributed embedded systems. Staunstrup and Wolf's edited volume [Sta97] surveys hardware/software co-design, including techniques for accelerated systems, such as those described in this chapter. Gupta and De Micheli [Gup93] and Ernst et al. [Ern93] describe early techniques for cosynthesis of accelerated systems. Callahan et al. [Cal00] describe an on-chip reconfigurable coprocessor connected to a CPU. The book *DVD Demystified* [Tay06] provides a thorough introduction to the DVD.

---

## Questions

**Q10-1** Describe an I<sup>2</sup>C bus at the following OSI-compliant levels of detail:

- a. physical
- b. data link
- c. network
- d. transport

**Q10-2** You are designing an embedded system using an Intel Atom as a host. Does it make sense to add an accelerator to implement the function  $z = ax + by + c$ ? Explain.

**Q10-3** You are designing an embedded system using an embedded processor with no floating-point support as host. Does it make sense to add an accelerator to implement the floating-point function  $S = A \sin(2\pi f + \phi)$ ? Explain.

**Q10-4** You are designing an embedded system using a high-performance embedded processor with floating point as host. Does it make sense to add an accelerator to implement the floating-point function  $S = A \sin(2\pi f + \phi)$ ? Explain.

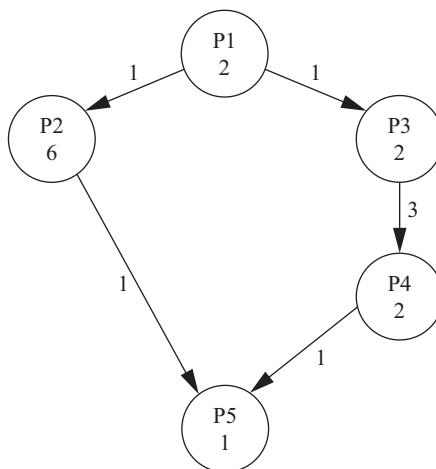
**Q10-5** You are designing an accelerated system that performs the following function as its main task:

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        f[i][j] = (pix[i][j - 1] + pix[i - 1][j] + pix[i][j] +
                    pix[i + 1][j] +
                    pix[i][j + 1]) / (5 * MAXVAL);
```

Assume that the accelerator has the entire `pix` and `f` arrays in its internal memory during the entire computation—`pix` is read into the accelerator before the operations begin and `f` is written out after all computations have been completed.

- a. Show a system schedule for the host, accelerator, and bus, assuming that the accelerator is inactive during all data transfers. All data are sent to the accelerator before it starts, and data are read from the accelerator after the computations are finished.
- b. Show a system schedule for the host, accelerator, and bus, assuming that the accelerator has enough memory for two `pix` and `f` arrays and that the host can transfer data for one set of computations while another set is being performed.

**Q10-6** Find the longest path through the graph below, using the computation times on the nodes and the communication times on the edges.



**Q10-7** Write pseudocode for an algorithm to determine the longest path through a system execution graph. The longest path is to be measured from one designated entry point to one exit point. Each node in the graph is labeled with a number, giving the execution time of the process represented by that node.

## Lab exercises

- L10-1** Determine how much logic in an FPGA must be devoted to a PCIe bus interface and how much would be left for an accelerator core.
- L10-2** Develop a debugging scheme for an accelerator. Determine how you would easily enter data into the accelerator and easily observe its behavior. You will need to verify the system thoroughly, starting with basic communication and going through algorithmic verification.

- L10-3** Develop a generic streaming interface for an accelerator. The interface should allow streaming data to be read by the accelerator from the host's memory. It should also allow streaming data to be written from the accelerator back to the memory. The interface should include a host-side mechanism for filling and draining streaming data buffers.

# Glossary

## A

- A/D converter** see *analog/digital converter*.
- absolute address** An address of an exact location in memory (Section 2.5).
- AC0-AC3** The four accumulators available in the C55x (Section 2.5).
- accumulator** A register used as both the source and destination for arithmetic operations, as in accumulating a sum (Section 2.5).
- ack** Short for *acknowledge*, a signal used in handshaking protocols (Section 4.3).
- ACPI** Advanced Configuration and Power Interface, an industry standard for power management interfaces (Section 4.9).
- act** See *activity diagram*.
- activation record** A data structure that describes the information required by a currently active procedure call (Section 2.3).
- active class** A UML class that can create its own thread of control (Section 6.4).
- active RFID** An RFID tag that transmits both on its own and in response to a request (Section 8.2).
- activity diagram** A description of a block using a combination of object and data flow notation (Section 7.4).
- ADAS** See *advanced driver assistance systems*.
- ADC** see *analog/digital converter*.
- advanced driver assistance systems** Automotive systems designed to help drivers with driving functions while requiring driver engagement (Section 9.2).
- analog/digital converter** A device that converts an analog signal into digital form (Section 4.5).
- application layer** In the OSI model, the end-user interface (Section 8.4).
- ASIC** Application-specific integrated circuit.
- aspect ratio** In a memory component, the ratio of the number of addressable units to the number of bits read per request (Section 4.8).
- assembler** A program that creates object code from a symbolic description of instructions (Section 5.4).
- atomic operation** An operation that cannot be interrupted (Section 6.5).
- attestation** A security operation designed to demonstrate the integrity of a sender (Section 4.10).
- attribute** alternate name for a field in a database (Section 8.5).
- auto-indexing** Automatically incrementing or decrementing a value before or after using it (Section 2.3).
- availability** Probability of a system's correct operation over time (Section 7.6).
- average-case execution time** A typical execution time for typical inputs (Section 5.6).

**B**

**bank** A block of memory in a memory system or cache (Section 4.4).

**base-plus-offset addressing** Calculating the address by adding a base address to an offset; the offset is usually contained in a register (Section 2.3).

**basis paths** A set of execution paths that cover the possible execution paths (Section 5.10).

**Bayer pattern** An arrangement of colors in a color filter array with two greens, one red, and one blue in a  $2 \times 2$  pattern (Section 5.12).

**Bayer pattern interpolation** See *demosaicing*.

**bdd** See *block definition diagram*.

**best-case execution time** The shortest execution time for any possible set of inputs (Section 5.6).

**best-effort routing** The Internet routing methodology, which does not guarantee completion (Section 8.4).

**big-endian** A data format in which the low-order byte is stored in the highest bits of the word (Section 2.2).

**black-box testing** Testing a program without knowledge of its implementation (Section 5.10).

**block definition diagram** A diagram that defines the types of blocks in a system.

**block motion estimation** A video compression algorithm that estimates one frame by another by analyzing the motion of blocks in the frames (Section 10.5).

**block repeat** In the C55x, a set of instructions that are executed several times in a row (Section 2.5).

**Bluetooth** A wireless network often used for IoT applications (Section 8.4).

**Bluetooth low energy** A variation of Bluetooth designed for low-energy operation (Section 8.4).

**boot-block flash** A type of flash memory that protects some of its contents (Section 4.4).

**bottom-up design** Using information from lower levels of abstraction to modify the design at higher levels of abstraction (Section 1.3).

**branch table** A multiway branching mechanism that uses a value to index a table of branch targets.

**branch target** The destination address of a branch (Section 2.3).

**branch testing** A technique to generate a set of tests for conditionals (Section 5.10).

**breakpoint** A stopping point for system execution (Section 4.5).

**bridge** A logic unit that acts as an interface between two buses (Section 4.3).

**bundle** A collection of logically related signals.

**burst transfer** A bus transfer that transfers several contiguous locations without separate addresses for each (Section 4.3).

**bus bandwidth** Bits per unit time that can be transmitted over a bus (Section 4.8).

**bus** Generally, a shared connection. CPUs use buses to connect to external devices and memory (Section 4.3).

**bus grant** The granting of ownership of the bus to a device (Section 4.3).

**bus master** Current owner of the bus (Section 4.3).

**bus request** A request to obtain ownership of the bus (Section 4.3).

**busy-wait I/O** Servicing an I/O device by executing instructions that test the device's state (Section 3.2).

## C

**cache** A small memory that holds copies of certain main memory locations for fast access (Section 3.5).

**cache hit** A memory reference to a location currently held in the cache (Section 3.5).

**cache miss** A memory reference to a location not currently in the cache (Section 3.5).

**cache miss penalty** The extra time incurred for a memory reference that is a cache miss (Section 3.6).

**CAN bus** A serial bus for networked embedded systems, originally designed for automobiles (Section 9.4).

**Capability Maturity Model** A method developed at the Software Engineering Institute of Carnegie Mellon University for assessing the quality of software development processes (Section 7.6).

**capacity miss** A cache miss that occurs because the program's working set is too large for the cache (Section 3.5).

**CAS** See *column address select*.

**CDFG** See *control/data flow graph*.

**central processing unit** The part of the computer system responsible for executing instructions fetched from memory (Section 2.2).

**certification** A legal process for determining the safety of a system (Section 9.2).

**certification authority** A system resource that manages digital certificates (Section 5.11).

**changing** In logic timing analysis, a signal whose value changes at a particular moment in time (Section 4.3).

**channel** A separate data transfer path in a memory system (Section 4.4).

**chrominance** A signal related to color (Section 5.13).

**circular buffer** An array used to hold a window of a data stream (Section 5.2).

**circular buffer start address register** In the C55x, a register used to define the start of a circular buffer.

**CISC** Complex instruction set computer; typically uses a number of instruction formats of varying length and provides complex operations in some instructions (Section 2.2).

**class** A type description in an object-oriented language (Section 1.3).

**class diagram** A UML diagram that defines classes and shows derivation relationships among them (Section 1.3).

**clear-box testing** Generating tests for a program with knowledge of its structure (Section 5.10).

**CMM** See *Capability Maturity Model*.

**CMOS** Complementary metal oxide semiconductor; the dominant VLSI technology today.

- code motion** A technique for moving operations in a program without affecting its behavior (Section 5.7).
- code signing** The act of combining a signing certificate with a digital signature of code to create a verifiable code module (Section 5.11).
- co-kernel** A specialized kernel for real-time processes (Section 6.8).
- cold miss** See *compulsory miss*.
- collaboration diagram** A UML diagram that shows communication among classes without the use of a timeline (Section 1.3). See also *sequence diagram*.
- color filter array** An array of color filters over an image sensor, one for each pixel (Section 5.13).
- color space** A mathematical representation of a set of colors (Section 5.13).
- color space conversion** Conversion of color from one representation to another, such as from RGB to YCrCb (Section 5.13).
- color temperature** A measure of the color of light (Section 5.13).
- column address select** A DRAM signal that indicates the column part of the address is being presented to the memory (Section 4.4).
- compare-and-swap** An instruction that swaps a register and memory location and performs a comparison. The operation is performed atomically (Section 6.5).
- component bandwidth** The bits per unit time transferred by a component (Section 4.7).
- compulsory miss** A cache miss that occurs the first time a location is used (Section 3.5).
- computational kernel** A small portion of an algorithm that performs a long function (Section 10.4).
- computing platform** A hardware system used for embedded computing (Section 4.1).
- concurrent engineering** Simultaneous design of several different system components.
- conflict graph** A graph that represents incompatibilities between entities; used in register allocation (Section 5.5).
- conflict miss** A cache miss caused by two locations in use mapping to the same cache location (Section 3.5).
- control/data flow graph** A graph that models both the data and control operations in a program (Section 5.3).
- controllability** The ability to set a value in system state during testing.
- co-processor** An optional unit added to a CPU responsible for executing some of the CPU's instructions (Section 3.4).
- Cortex** A family of ARM processors for compute-intensive applications (Section 2.3).
- counter** A device that counts asynchronous external events (Section 4.5).
- CPSR** Current program status register in the ARM processor (Section 2.3).
- CPU** See *central processing unit*.
- CRC card** A technique for capturing design information (Section 7.3).
- critical instant** In RMA, the worst-case combination of process (Section 6.5).
- critical section** A section of code that must be executed without interference (Section 6.5).

**critical timing race** Two operations are in a *critical timing race* if the result of the operations depends on the order in which they finish (Section 6.5).

**cross compiler** A compiler that runs on one architecture but generates code for a different one (Section 4.5).

**cryptographic hash function** A function designed to create a *message digest* that represents the message in shortened form (Section 4.10).

**cryptography** The mathematical and computational study of secure communication (Section 4.10).

**cycle-accurate simulator** A CPU simulation that is accurate to the clock-cycle level (Section 5.6).

**cyclomatic complexity** A measure of the control complexity of a program (Section 5.10).

## D

**DAC** see *digital-to-analog converter*.

**data dependency** A constraint on the order of execution of statements in a program based on data calculations and assignments (Section 2.2).

**data flow graph** A graph that models data operations without conditionals (Section 5.3).

**data flow testing** A technique for generating tests by examining the data flow representation of a program (Section 5.10).

**data link layer** In the OSI model, the layer responsible for reliable data transport (Section 8.4).

**database** A structured collection of data (Section 8.5).

**DaVinci** A media-oriented heterogeneous multiprocessor (Section 10.4).

**DCT block** In JPEG, a block of two-dimensional DCT coefficients.

**DCT** See *discrete cosine transform*.

**dead code elimination** Eliminating code that can never be executed (Section 5.5).

**deadline** The time at which a process must finish (Section 6.3).

**decision node** A node in a CDFG that models a conditional (Section 5.3).

**def-use analysis** Analyzing the relationships between reads and writes of variables in a program (Section 5.10).

**delayed branch** A branch instruction that always executes one or more instructions after the branch, independent of whether the branch is taken (Section 3.6).

**demosaicing** The process of interpolating the colors not captured at a given pixel by a color filter array (Section 5.13).

**dense instruction set** An instruction set designed to provide compact code (Section 5.9).

**dependability** The length of time for which a system can operate without defects (Section 7.6).

**dequeue** To remove something from a queue (Section 5.2).

**design flow** A series of steps used to implement a system (Section 7.2).

**design methodology** A method of proceeding through levels of abstraction to complete a design (Section 7.2).

**design process** See *design methodology*.

- Detailed Resource Modeling Profile** A MARTE profile for software and hardware resource modeling (Section 7.4).
- digital signal processor** A microprocessor whose architecture is optimized for digital signal processing applications (Sections 2.1, 2.5).
- digital signature** An authentication showing that a message comes from a particular sender (Section 4.10).
- digital/analog converter** see *digital-to-analog converter*.
- digital-to-analog converter** A circuit that translates a digital value to an analog signal value (Section 4.5).
- DIMM** Dual inline memory module, a small, printed circuit board containing RAM chips on both sides (Section 4.4).
- direct memory access** A bus transfer performed by a device without executing instructions on the CPU (Section 4.3).
- direct-mapped cache** A cache with a single set (Section 3.5).
- discrete cosine transform** An image processing transform from the pixel domain to the spatial frequency domain.
- distributed embedded system** An embedded system built around a network or one in which communication between processing elements is explicit (Section 10.3).
- DMA controller** A logic unit designed to execute DMA transfers (Section 4.3).
- DMA** See *direct memory access*.
- DNS** See *domain name service*.
- domain architecture** Automobile network architecture in which operations are functionally organized (Section 9.3).
- domain name service** An Internet service that translates names to Internet addresses (Section 8.4).
- domain-specific modeling language** A design language that captures both functional and non-functional characteristics of a system in a given design domain (Section 7.4).
- DOS FAT file system** A file system compatible with the MS-DOS file system.
- DOS** Generically, a disc-based operating system. Often used as shorthand for MS-DOS.
- downsampling** A filtering operation that reduces the sampling rate of a signal.
- DPOF** Digital print order format, a standard for generating information used to control printing of images (Section 5.13).
- DRAM** See *dynamic random access memory*.
- DRM** See *Detailed Resource Modeling Profile*.
- DSML** See *domain-specific modeling language*.
- DSP** See *digital signal processor*.
- dual-kernel** An operating system architecture that uses two kernels, one for real-time operation and another for non-real-time computation (Section 6.9).
- DVFS** See *dynamic voltage and frequency management*.

**dynamic power management** A power management technique that monitors CPU activity (Section 3.7).

**dynamic random access memory** A memory that relies on stored charge (Section 4.4).

**dynamic voltage and frequency management** A power management technique in which power supply voltage and clock frequency are adjusted based on required processing speeds (Section 3.7)

**dynamically linked library** A code library linked into a program at the start of execution (Section 5.4).

## E

**earliest deadline first** A variable priority scheduling scheme (Section 6.5).

**EDF** See *earliest deadline first*.

**EDO RAM** Extended-data-out RAM, a memory that provides looser constraints on data timing (Section 4.4).

**EEPROM** Electrically erasable programmable random access memory (Section 2.4).

**effective address** An address that can be used to fetch a memory location.

**effective address calculation** The process of calculating an effective address.

**embedded computer system** A computer used to implement functionality of something other than a general-purpose computer (Section 1.2).

**energy** Ability to do work.

**enq** Short for *enquiry*, a signal used in handshaking protocols (Section 4.3).

**enqueue** To add something to a queue (Section 5.2).

**entropy coding** A form of lossless data compression, such as Huffman coding.

**entry point** A label in an assembly language module that can be referred to by other program modules (Section 5.4).

**error injection** Evaluating test coverage by inserting errors into a program and using packaged tests to try to find those errors (Section 5.10).

**evaluation board** A printed circuit board designed to emulate a typical platform (Section 4.6).

**exception** Any unusual condition in the CPU that is recognized during execution (Section 3.3).

**executable binary** An object program that is ready for execution (Section 5.4).

**execute packet** In the C64x, a set of instructions that execute together (Section 2.6).

**EXIF** Exchangeable image file format, a file format that combines image, audio, and other information (Section 5.13).

**expression simplification** Rewriting an arithmetic expression (Section 5.5).

**external reference** A reference in an assembly language program to another module's entry point (Section 5.4).

## F

**fast return** In the C55x, a procedure return that uses some registers rather than the stack to store certain values (Section 2.5).

- federated architecture** An architecture for networked embedded systems that is constructed from several networks, each corresponding to an operational subsystem (Section 9.3).
- fetch packet** In the C64x, a set of instructions that are fetched together (Section 2.6).
- FFT** See *fast Fourier transform*.
- field** A column of a table in a database (Section 8.5)
- field-programmable gate array** An integrated circuit that can be programmed by the user; provides multilevel logic.
- file register** In the PIC architecture, a location in a general purpose register file (Section 2.4).
- fingerprinting** Identifying a piece of software by analyzing its binary code (Section 7.6).
- finite impulse response filter** A type of digital filter in which the output does not depend on the previous outputs (Section 5.2).
- FIR filter** See *finite impulse response filter*.
- first-level cache** The cache closest to the CPU (Section 3.5).
- flash file system** A file system specially designed for flash memory storage (Section 4.7).
- flash memory** An electrically erasable programmable read-only memory (Section 4.4).
- FlexRay** A network designed for real-time systems (Section 9.4).
- four-cycle handshake** A handshaking protocol that goes through four states (Section 4.3).
- FPGA** See *field-programmable gate array*.
- FPM DRAM** See *fast page mode DRAM*.
- frame pointer** Points to the end of a procedure stack frame (Section 5.5).
- function** In a programming language, a procedure that can return a value to the caller (Section 2.3).
- functional requirements** Requirements describing the logical behavior of the system (Section 7.3).

## G

- Generic Quantitative Analysis Modeling Profile** A MARTE profile for schedulability and performance analysis.
- glue logic** Interface logic or other logic having no particular structure.
- glueless interface** An interface between components that requires no glue logic.
- GPIO** General-purpose input/output; a term for uncommitted pins that can be used as inputs or outputs.
- GQAM** See *Generic Quantitative Analysis Modeling Profile*.

## H

- HAL** See *hardware abstraction layer*.
- handshake** A protocol designed to confirm the arrival of data (Section 4.3).
- hardware abstraction layer** Low-level software that provides drivers and support for basic elements of a hardware platform (Section 4.2).

- hardware platform** A hardware system used as a component in a larger system (Section 4.2).
- hardware/software co-design** The simultaneous design of hardware and software components to meet system requirements (Section 10.4).
- Harvard architecture** A computer architecture that provides separate memories for instructions and data (Section 2.2).
- heterogeneous multiprocessor** A multiprocessor with several different types of processing elements.
- High-Level Applications Modeling Profile** A MARTE profile for quantitative and qualitative features of a system.
- histogram** In signal processing, the number of samples over a given interval in each of several ranges (Section 5.13).
- hit rate** The probability of a memory access being a cache hit (Section 3.5).
- HLAM** See *High-Level Applications Modeling Profile*.
- host system** Any system used as an interface to another system (Section 4.6).
- HRM** Hardware resource modeling in MARTE (Section 7.4).
- Huffman coding** A method of data compression (Section 3.9).

## I

- I/O** Input/output (Section 3.2).
- I<sup>2</sup>C bus** A serial bus for distributed embedded systems (Section 10.4).
- ibd** See *internal block diagram*.
- IEEE 1394** A high-speed serial network for peripherals, also known as Firewire.
- IIR filter** See *infinite impulse response filter*.
- immediate operand** An operand embedded in an instruction rather than fetched from another location (Section 2.3).
- induction variable elimination** A loop optimization technique that eliminates references to variables derived from the loop control variable (Section 5.7).
- infinite impulse response filter** A type of digital filter in which the output depends on previous output values.
- initiation time** The time at which a process becomes ready to execute (Section 6.3).
- instruction set** The definitions of the operations performed by a CPU (Section 2.1).
- instruction-level simulator** A CPU simulator that is accurate to the level of the programming model but not to timing (Section 5.6).
- intellectual property** An intangible form of property ownership, such as software or hardware designs (Section 4.6).
- internal block diagram** A block diagram that shows connections between blocks (Section 7.4).
- Internet** A worldwide network based on the Internet protocol (Section 8.4).
- Internet appliance** An information system that makes use of the Internet.
- Internet protocol** A packet-based protocol (Section 8.4).

**Internet-enabled embedded system** Any embedded system that includes an Internet interface.

**Internet-of-Things** A network of devices; a soft real-time networked embedded system (Chapter 8).

**interpreter** A program that executes a given program by analyzing a high-level description of the program at execution time.

**interprocess communication** A mechanism for communication between processes (Section 6.6).

**interrupt** A mechanism that allows a device to request a service from the CPU (Section 3.2).

**interrupt handler** A routine called upon an interrupt to service the interrupting device (Section 3.2).

**interrupt latency** The time from the assertion of an interrupt to its service (Section 6.7).

**interrupt priority** Priorities used to determine which of several interrupts gets attention first (Section 3.2).

**interrupt service handler** Software that performs the minimal operations required to respond to a device interrupt (Section 6.7).

**interrupt service routine** Software that handles an interrupt request (Section 6.7).

**interrupt vector** Information used to select which segment of the program should be used to handle an interrupt request (Section 3.2).

**IP** See *Internet protocol*; see also *intellectual property*.

**ISH** See *interrupt service handler*.

**ISO 9000** A series of international standards for quality process management (Section 7.6).

**ISR** See *interrupt service routine*.

## J

**Jazelle** A set of ARM instruction set extensions for direct execution of Java bytecodes (Section 2.3).

**JFIF** JPEG file interchange format; a data format for representing JPEG data (Section 5.13).

**JIT compiler** A just-in-time compiler; compiles program sections on demand during execution.

**JPEG** 1. A widely used image compression standard; 2. Joint Photographic Experts Group.

## L

**L1 cache** See *first-level cache*.

**L2 cache** See *second-level cache*.

**label** In an assembly language, a symbolic name for a memory location (Section 2.2).

**layer diagram** A diagram showing the relationships between software components. A layer can call the layer(s) immediately below it (Section 4.2).

**lightweight process** A process that shares its memory spaces with other processes.

**LIN** Local Interconnect Network, a local area network designed for automotive electronics (Section 9.4).

**line replaceable unit** In avionics, an electronic unit that corresponds to a functional unit, such as a flight instrument.

**linker** A program that combines multiple object program units, resolving references between them (Section 5.4).

**Linux** A well-known open-source version of Unix (Section 6.8).

**little-endian** A data format in which the low-order byte is stored in the lowest bits of the word (Section 2.2).

**load map** A description of where object modules should be placed in memory (Section 5.4).

**loader** A program that loads a given program into memory for execution (Section 5.4).

**load-store architecture** An architecture in which only load and store operations can be used to access data and ALU, and other instructions cannot directly access memory (Section 2.3).

**logic analyzer** A machine that captures multiple channels of digital signals to produce a timing diagram view of execution (Section 4.6).

**longest path** The path through a weighted graph that gives the largest total sum of weights.

**loop nest** A set of loops, one inside the other (Section 5.7).

**loop unrolling** Rewriting a loop so that several instances of the loop body are included in a single iteration of the modified loop (Section 5.5).

**LRU** See *line replaceable unit*.

**luminance** A signal related to brightness (Section 5.13).

## M

**MARTE** A design language based on UML for model-based design of real-time embedded computing systems (Section 7.4).

**masking** Causing lower-priority interrupts to be held to service higher-priority interrupts (Section 3.2).

**memory controller** A logic unit designed as an interface between DRAM and other logic (Section 4.4).

**memory management unit** A unit responsible for translating logical addresses into physical addresses (Section 3.5).

**memory mapping** Translating addresses from logical to physical form (Section 3.5).

**memory-mapped I/O** Performing I/O by reading and writing memory locations corresponding to device registers (Section 3.2).

**message delay** The delay required to send a message on a network with no interference.

**message digest** The result of applying a cryptographic hash function to a message (Section 4.10).

**message passing** A style of interprocess communication (Section 6.6).

**methodology** An overall design process (Section 1.2).

**microcontroller** A microprocessor that includes memory and I/O devices, often including timers, on a single chip (Section 1.2).

**miss rate** The probability that a memory access will be a cache miss (Section 3.5).

**MMU** See *memory management unit*.

**model-based design** A design methodology that simultaneously considers the embedded computing system and physical plant (Section 7.4).

**MOST** Media Oriented Systems Transport; a local area network designed for automotive electronics (Section 9.4).

**motion vector** A vector describing the displacement between two units of an image (Section 10.5).

**MP3** An audio compression standard (Section 4.9).

**MPCore** An ARM multiprocessor.

**multihop network** A network in which messages may go through an intermediate PE when traveling from source to destinations.

**multiprocessor** A computer system that includes more than one processing element (Section 10.2).

**multirate** Operations having different deadlines, causing the operations to be performed at different rates (Sections 1.2, 6.3).

## N

**NEON** A set of ARM instruction set extensions for SIMD operations (Section 2.3).

**network** A system for communicating between components (Section 8.4).

**network availability delay** The delay incurred while waiting for the network to become available.

**network layer** In the OSI model, the layer that provides end-to-end networking services (Section 8.4).

**NFP** See *Non-Functional Properties Modeling Module*

**NMI** See *nonmaskable interrupt*.

**nonblocking communication** Interprocess communication that allows the sender to continue execution after sending a message (Section 6.6).

**Non-Functional Properties Modeling Module** A MARTE module for non-functional properties of a system (Section 7.4).

**nonfunctional requirements** Requirements that do not describe the logical behavior of a system; examples include size, weight, and power consumption (Sections 1.3, 7.3).

**nonmaskable interrupt** An interrupt that must always be handled, independent of other system activity (Section 3.2).

**normal form** A rule that helps ensure irredundancy to improve data management in a database (Section 8.5.1)

## O

**object** A program unit that includes both internal data and methods that provide the interface to the data (Section 1.3).

**object code** A program in binary form (Section 5.4).

**object oriented** Any use of objects and classes in design; can be applied at many different levels of abstraction (Section 1.3).

**observability** The ability to determine a portion of system state during testing.

**operating system** A program responsible for scheduling the CPU and controlling access to devices (Section 6.1).

**origin** The starting address of an assembly language module.

**OSI model** A model for levels of abstraction in networks (Section 8.4).

**overhead** In operating systems, the CPU time required for the operating system to switch contexts (Section 6.3).

## P

**P0** Traditional name for the procedure that takes a semaphore (Section 6.5).

**packet** 1. In VLIW architectures, a set of instructions that form a unit of execution (Section 2.2);  
2. In networks, a unit of transmission (Section 8.4).

**page fault** A reference to a memory page not currently in physical memory (Section 3.5).

**paged addressing** Division of memory into equal-sized pages (Section 3.5).

**PAM** See *Performance Analysis Modeling Profile*.

**par** See *parametric diagram*.

**parametric diagram** A SysML diagram for non-functional parametric-oriented requirements (Section 7.4).

**partitioning** Dividing a functional description into processes that can execute in parallel or modules that can be separately implemented.

**passive RFID** An RFID tag that only transmits upon request, or one with no internal battery source (Section 8.2).

**PC** 1. In computer architecture, see *program counter*; 2. Personal computer.

**PC sampling** Generating a program trace by periodically sampling the PC during execution.

**PCIe** PCI express, a high-performance bus for PCs and other applications.

**PC-relative addressing** An addressing mode that adds a value to the current PC (Section 2.3).

**PE** See *processing element*.

**peek** A high-level language routine that reads an arbitrary memory location (Section 3.2).

**Performance Analysis Modeling Profile** A MARTE profile for best-effort and soft real-time systems.

**performance** The speed at which operations occur (Section 1.3).

**period** In real-time scheduling, a periodic interval of execution (Section 6.3).

**physical layer** In the OSI model, the layer that defines electrical and mechanical properties (Section 8.4.1).

**pipeline** A logic structure that allows several operations of the same type to be performed simultaneously on multiple values, with each value having a different part of the operation performed at any one time (Section 3.6).

**pixel** A sample in an image (Section 5.13).

**platform** Hardware and associated software designed to serve as the basis for a number of different systems to be implemented.

**PLC** See *program location counter*.

**poke** A high-level language routine that writes an arbitrary location (Section 3.2).

**polling** Testing one or more devices to determine whether they are ready (Section 3.2).

**POSIX** A standardized version of Unix (Section 6.8).

**post-indexing** An addressing mode in which an index is added to the base address after the fetch (Section 2.3).

**power** Energy per unit time (Section 3.7).

**power management policy** A scheme for making power management decisions.

**power state machine** A finite-state machine model for the behavior of a component under power management (Section 3.7).

**power-down mode** A mode invoked in a CPU that causes it to reduce its power consumption (Section 3.7).

**preemptive multitasking** A scheme for sharing the CPU in which the operating system can interrupt the execution of processes (Section 6.4).

**presentation layer** In the OSI model, the layer responsible for data formats (Section 8.4.1).

**primary key** A unique identifier for a record in a database (Section 8.5.1)

**priority inheritance** An algorithm used to prevent priority inversion in which a process temporarily takes on the priority of a shared resource. (Section 6.5)

**priority inversion** A situation in which a lower-priority process prevents a higher-priority process from executing (Section 6.5).

**priority-driven scheduling** Any scheduling technique that uses priorities of processes to determine the running process (Section 6.5).

**procedure** A programming language construct that allows a single piece of code to be called at multiple points in the program (Section 2.3). Generally, it is a synonym for *subroutine*; see also *function*.

**procedure call stack** A stack of records for currently active processes (Section 2.3).

**procedure linkage** A convention for passing parameters and other actions required to call a procedure (Section 2.3).

**process** A unique execution of a program (Section 6.2).

**processing element** A component that performs a computation under the coordination of the system (Section 10.2).

**producer/consumer** A set of functions or processes in which one writes data to be read by the other (Section 5.2).

**profiling** A procedure for counting the relative execution times of different parts of a program (Section 5.6).

**program counter** A common name for the register that holds the address of the currently executing instruction (Section 2.2).

**program location counter** A variable used by an assembler to assign memory addresses to instructions and data in the assembled program (Section 5.4).

**programming model** The CPU registers visible to the programmer (Section 2.2).

**pseudo-op** An assembly language statement that does not generate code or data (Sections 2.2, 5.4).

**public-key cryptography** A cryptographic method that relies on a key that is partially available to others (Section 4.10).

## Q

**quality assurance** A process for ensuring that systems are designed and built to high quality standards (Section 7.6).

**quantization** Assignment of a continuous sample value to a discrete value.

**quantization matrix** In JPEG, a set of values used to guide quantization (Section 5.13).

**query** A request for data from a database (Section 8.5)

**queue** A data structure that provides first-in first-out access to data (Section 5.2).

## R

**race condition** A set of processes whose results depend on the order in which they execute (Section 6.5).

**race-to-dark** A power management policy used in systems with high leakage current in which programs run as fast as possible to allow the processor to shut down (Section 3.7).

**RAM** See *random-access memory*.

**random testing** Testing a program using randomly generated inputs (Section 5.10).

**random-access memory** A memory that can be addressed in arbitrary order (Section 4.4).

**RAS** See *row address select*.

**raster scan (or order) display** A display that writes pixels by rows and columns.

**rate** Inverse of period (Section 6.3).

**rate-monotonic scheduling** A fixed-priority scheduling scheme (Section 6.5).

**RDBMS** See *relational database management system*.

**reactive system** A system designed to react to external events.

**read-only memory** A memory with fixed contents (Section 4.4).

**real time** A system that must perform operations by a certain time (Section 1.2).

**real-time operating system** An operating system designed to be able to satisfy real-time constraints (Section 6.1).

**record** A row in a table in a database (Section 8.5).

**reentrancy** The ability of a program to be executed multiple times using the same memory image without error (Section 5.4).

**refresh** Restoring the values kept in a DRAM (Section 4.4).

**register allocation** Assigning variables to registers (Section 5.5).

**register** Generally, an electronic component that holds a state. In the context of computer programming, storage internal to the CPU that is part of the programming model (Section 2.2).

**register-indirect addressing** Fetching from a first memory location to find the address of the memory location containing the operand (Section 2.3).

**regression testing** Testing hardware or software by applying previously used tests (Section 5.10).

**relational database management system** A database organized on a relational model (Section 8.5)

**relative address** An address measured relative to some other location, such as the start of an object module (Section 5.4).

**reliability** Assurance with which a system is certain perform its intended function.

**repeat** In instruction sets, an instruction that allows another instruction or set of instructions to be repeated in order to create low-overhead loops (Section 2.5).

**replay attack** An attack in which the non-faulty output of a device is recorded and then played back while the device is operated improperly (Section 7.6).

**req** See *requirement diagram*.

**requirement** A description of a desired characteristic of a system (Section 7.3).

**requirement diagram** A SysML diagram used to capture functional requirements (Section 7.4).

**reservation table** A hardware technique for scheduling instructions (Section 5.5).

**response time** The time span between the initial request for a process and its completion (Section 6.3).

**RFID** Radio frequency identification (Section 8.2).

**RISC** Reduced instruction set computer (Section 2.2).

**RMA** Rate-monotonic analysis; another term for *rate-monotonic scheduling*.

**rollover** Reading multiple keys when two keys are pressed at once.

**ROM** See *read-only memory*.

**root-of-trust** A trusted environment for the execution of certain pre-verified code (Section 3.8).

**row address select** A DRAM signal that indicates the row part of the address is being presented (Section 4.4).

**RTOS** See *real-time operating system*.

## S

**SAE International driving automation levels** Levels of driving automation defined by SAE International (Section 9.2).

**safety** Release of energy in ways that does not cause harm (Section 1.2).

**saturation arithmetic** An arithmetic system that provides a result at the maximum/minimum value on overflow/underflow.

**scheduling** Determining the time at which an operation will occur (Section 105.4).

**scheduling overhead** Execution time required to make a scheduling decision (Sections 6.2, 6.3).

**scheduling policy** A methodology for making scheduling decisions (Section 6.3).

**schema** A data organization design for a database (Section 8.5).

**schemaless** database A database that does not have a schema (Section 8.5).

**SDE** See *software development environment*.

**SDRAM** See *synchronous DRAM*.

**second-level cache** A cache after the first-level cache but before main memory (Section 3.5).

**secret-key cryptography** A cryptographic method that relies on a key that is not generally known (Section 4.10).

**security** A system's ability to prevent malicious attacks (Section 1.2).

**segmented addressing** Dividing memory into large, unequal-sized segments (Section 3.5).

**semaphore** A mechanism for coordinating communicating processes (Section 6.5).

**sequence diagram** A UML diagram type that shows how objects communicate over time using a timeline (Section 1.3). See also *collaboration diagram*.

**session layer** In the OSI model, the layer responsible for application dialog control (Section 8.4).

**set-associative cache** A cache with multiple sets (Section 3.5).

**set-top box** A system used for cable or satellite television reception.

**shared memory** A communication style that allows multiple processes to access the same memory locations (Section 6.6).

**shared resource** A resource, such as an I/O device or a memory location, used by more than one process (Section 6.5).

**sharpening** In image processing, a filtering process that produces edges that appear to be sharper (Section 5.13).

**signal** 1. A Unix interprocess communication method (Section 6.6); 2. A UML stereotype for communication (Section 6.6).

**SIMM** Single inline memory module, a small, printed circuit board containing RAM chips on one side (Section 4.4).

**single-assignment form** A program that writes to each variable once at most (Section 5.3).

**single-hop network** A network in which messages can travel from one PE to any other PE without going through a third PE.

**slow return** In the C55x, a procedure return that uses the stack to restore return address and loop context, compared with a fast return that uses registers (Section 2.5).

**smart card** A portable identification card architect

**smart home** An IoT-enabled home that may provide a variety of services (Section 8.6).

**software development environment** A set of tools for developing software, often including an editor, compiler, linker, and debugger (Section 4.6).

**software interrupt** See *trap*.

**software pipelining** A technique for scheduling instructions in loops.

**software platform** Software used as a component in a larger system (Section 4.3).

**spatial frequency** A frequency representation of a modulated visual intensity (Section 5.13).

**special function register** In the PIC architecture, registers for I/O and other special operations (Section 2.4).

**specification** A complete set of requirements for a system (Section 7.3).

**speedup** Ratio of system performance before and after a design modification (Section 10.4).

**spill** Writing a register value to main memory so that the register can be used for another purpose (Section 5.5).

**spiral model** A design methodology in which the design iterates through specification, design, and test at increasingly detailed levels of abstraction.

**SRAM** See *static random-access memory*.

**SRM** Software resource modeling in MARTE (Section 7.4).

**stack pointer** Points to the top of a procedure call stack (Section 5.5).

**state machine** Generally, a machine that goes through a sequence of states over time; may be implemented in software (Sections 1.3, 5.2).

**state mode** A logic analyzer mode that provides reduced timing resolution in return for longer time spans (Section 4.5).

**static power management** A power management technique that does not consider the current CPU behavior (Section 3.7).

**static random-access memory** A RAM that consumes power to continuously maintain its stored values (Section 2.4).

**static scheduling** A scheduling policy in which process priorities are fixed (Section 6.5).

**streaming data** A sequence of data values that is received periodically, such as for digital signal processing.

**strength reduction** Replacing an operation with another equivalent operation that is less expensive (Section 5.7).

**structured query language** A language used to design queries for database systems (Section 8.5).

**subroutine** An assembly/machine language version of a *procedure* (Section 2.3).

**successive refinement** A design methodology in which the design goes through the levels of abstraction several times, adding detail in each refinement phase (Section 7.2).

**superscalar** An execution method that can perform several different instructions simultaneously using dynamically scheduled instructions (Section 2.2).

**supervisor mode** A CPU execution mode with unlimited privileges (Section 3.3). See also *user mode*.

**symbol table** Generally, a table relating symbols in a program to their meaning; in an assembler, a table giving the locations specified by labels (Section 5.4).

**synchronous DRAM** A memory that uses a clock (Section 4.4).

**SysML** A UML-based design language for broad-spectrum systems engineering (Section 7.4).

**system-on-silicon** A single-chip system that includes computation, memory, and I/O.

## T

**tag** The part of a cache block that provides the address bits from which the cache entry came (Section 3.5).

**target system** A system being debugged with the aid of a host (Section 4.6).

**task graph** A graph that shows processes and data dependencies among them (Section 6.2).

**TCP** See *transmission control protocol*.

**testbench** A setup used to test a design; may be implemented in software to test other software (Section 4.6).

**testbench program** A program running on a host used to interface to a debugger that runs on an embedded processor (Section 4.6).

**thread** See *lightweight process*.

**thumbnail** A small version of an image (Section 5.13).

**TIFF** Tagged Image File Format, an image file format (Section 5.13).

**Time Modeling Module** A MARTE module for chronometric and sequential time.

**Time** See *Time Modeling Module*.

**timer** A device that measures time from a clock input.

**timewheel** A time-sorted queue used to manage the processing of events over time (Section 8.5)

**timing mode** A logic analyzer mode that provides increased timing resolution (Section 4.6).

**TLB** See *translation lookaside buffer*.

**top-down design** Designing from higher levels of abstraction to lower levels of abstraction (Section 1.3).

**trace** A record of the execution path of a program (Section 5.6).

**trace-driven analysis** Analyzing a trace of a program's execution (Section 5.6).

**translation lookaside buffer** A cache used to speed up virtual-to-physical address translation (Section 3.5).

**Transmission control protocol** A connection-oriented protocol built upon the IP (Section 8.4).

**transport layer** In the OSI model, the layer responsible for connections (Section 8.4).

**trap** An instruction that causes the CPU to execute a predetermined handler (Section 3.3).

**trusted execution environment** An execution environment allowed additional privileges based on its protection (Section 3.8).

**TrustZone** A set of ARM instruction set extensions for security operations (Section 2.3).

**tuple** An alternate name for a record in a database (Section 8.5)

## U

**UART** Universal Asynchronous Receiver/Transmitter, a serial I/O device.

**UML** See *Unified Modeling Language*.

**unified cache** A cache that holds both instructions and data (Section 3.5).

**Unified Modeling Language** A widely used graphical language that can be used to describe designs at many levels of abstraction (Section 1.3).

**upsampling** A filtering operation that increases the sampling rate of a signal.

**usage scenario** A description of how a system will be used (Section 7.3).

**USB** Universal Serial Bus, a high-performance serial bus for PCs and other systems.

**use case** A description of the operation of a system by external actors (Section 1.3).

**user mode** A CPU execution mode with limited privileges (Section 3.3). See also *supervisor mode*.

**utilization** Generally, the fractional or percentage time that we can effectively use a resource; the term is most often applied to how processes make use of a CPU (Section 6.3).

## V

**V()** Traditional name for the procedure that releases a semaphore (Section 6.5).

**very long instruction word** A style of computer architecture in which multiple instructions are statically scheduled. Compare to *superscalar* (Section 2.2).

**virtual addressing** Translating an address from a logical to a physical location (Section 3.5).

**VLIW** See *very long instruction word*.

**VLSI** Acronym for *very large-scale integration*; generally, any modern integrated circuit fabrication process.

**von Neumann architecture** A computer architecture that stores instructions and data in the same memory (Section 2.2).

## W

**wait state** A state in a bus transaction that waits for the response of a memory or device (Section 4.3).

**watchdog timer** A timer that resets the system when the system fails to periodically reset the timer (Section 4.6).

**waterfall model** A design methodology in which the design proceeds from higher to lower levels of abstraction (Section 7.2).

**way** A bank in a cache (Section 3.5).

**white-box testing** See *clear-box testing*.

**word** The basic unit of memory access in a computer (Section 2.2).

**working set** The set of memory locations used during a chosen interval of a program's execution (Section 3.5).

**worst-case execution time** The longest execution time for any possible set of inputs (Section 5.6).

**write-back** Writing to main memory only when a line is removed from the cache (Section 3.5).

**write-through** Writing to main memory for every write into the cache (Section 3.5).

## Z

**ZigBee** A wireless network often used in IoT applications (Section 8.4).

**zig-zag pattern** In JPEG, the order in which DCT coefficients are read from the matrix. The zig-zag pattern starts at the upper left and moves in diagonals to the lower right (Section 5.13).

**zone architecture** An automobile network architecture in which gateways in different areas of the car perform operations for a variety of different functions (Section 9.3).

# References

- [ACM18] Association for Computing Machinery, “Fathers of the deep learning revolution receive ACM A. M. Turing Award,” 2018, 2018 Turing Award ([acm.org](http://acm.org))
- [ACP13] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation, *Advanced Configuration and Power Interface Specification*, Revision 5.0 Errata A, November 13, 2013.
- [Ado92] Adobe Developers Association, TIFF, Revision 6.0, June 3, 1992. Available at <http://partners.adobe.com/public/developer/tiff/index.html>.
- [Aho06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, second edition. Reading, MA: Addison-Wesley, 2006.
- [Aki96] Olu Akiwumu-Assani and Marnix Vlot, “Multimedia terminal architecture,” *Philips Journal of Research* 50(1/2) (1996): 169–184.
- [Ald73] Robin Alder, Mark Baker, and Howard D. Marshall, “The logic analyzer: A new instrument for observing logic signals,” *Hewlett-Packard Journal* 25(2) October (1973): 2–16.
- [ARM00] ARM Limited, *Integrator/LM-XCV400+ Logic Module*, ARM DUI 0130A, February 2000. Available at [www.arm.com](http://www.arm.com).
- [ARM02] ARM Limited, *ARM PrimeCell Vectored Interrupt Controller (PLI92) Technical Reference Manual*, ARM DDI 0273A, 2002. Available at [www.arm.com](http://www.arm.com).
- [ARM08] ARM Limited, ARM11 MPCore Processor Technical Reference Manual, revision r2p0, 2008. Available at [www.arm.com](http://www.arm.com).
- [ARM09] ARM Limited, ARM Security Technology: Building a Secure System using TrustZone Technology, 2009. Available at [www.arm.com](http://www.arm.com).
- [ARM11] ARM Limited, Cortex-R5 Processor Technical Reference Manual, revision r1p2, 2011. Available at [www.arm.com](http://www.arm.com).
- [ARM13] ARM Limited, Global Platform based Trusted Execution Environment and Trust-Zone® Ready, Rob Coombs/ATC-314, October 31, 2013.
- [ARM13B] ARM Limited, Arm Architecture Reference Manual, Armv8, for Armv8-A architecture profile, ARM DDI 0487G.a ID011921, 2013-2021.
- [ARM16] ARM Limited, *Memory Protection Unit (MPU)*, Version 1.0, 8 July 2016.
- [ARM17] ARM Limited, ARMv8-A Power management, version 1.0, ARM 100960\_0100\_en, 2017.
- [Arm21] Arm Limited, *Arm Architecture Reference Manual, ARMv8, for Armv8-A architecture profile*, ARM DDI 0487G.a (ID011921), 2021.
- [ARM96] ARM Limited, ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition, ARM DDI 0406C.d, 1996-1998, 200, 2004-2012, 2014, 2018.
- [ARM99A] ARM Limited, *AMBA(TM) Specification (Rev 2.0)*, 1999. Available at [www.arm.com](http://www.arm.com).
- [ARM99B] ARM Limited, *ARM7TDMI-S Technical Reference Manual*, 1999. Available at [www.arm.com](http://www.arm.com).
- [Asa98] Mutsuhiko Asada and Pong Mang Yan, “Strengthening software quality assurance,” *Hewlett-Packard Journal* 49(2) May (1998): 89–97.
- [Aud93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, “Applying new scheduling theory to priority pre-emptive scheduling,” *Software Engineering Journal*, Volume 8, Issue 5, September 1993, pp. 284–292.

- [Aus04] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Wayne Wolf, “Mobile supercomputers,” *IEEE Computer* 37(5) May (2004): 81–83.
- [Ban93] Uptal Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*. Boston: Kluwer Academic Publishers, 1993.
- [Ban94] Uptal Banerjee, *Loop Parallelization*. Boston: Kluwer Academic Publishers, 1994.
- [Ban95] Amir Ban, “Flash file system,” U. S. Patent 5,404,485, April 4, 1995.
- [Bar07] Richard Barry, “The Free RTOS Project” <http://www.freertos.org>.
- [Bay76] Bryce E. Bayer, “Color imaging array,” U. S. Patent 3,971,065, July 20, 1976.
- [Bei11] <http://beagleboard.org>. February 14, 2012.
- [Bei13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers, “The SIMON and SPECK families of lightweight block ciphers,” National Security Agency, 9800 Savage Road, Fort Meade MD 20755, USA, June 19, 2013.
- [Bei84] Boris Beizer, *Software System Testing and Quality Assurance*. New York: Van Nostrand Reinhold, 1984.
- [Bei90] Boris Beizer, *Software Testing Techniques*, second edition. New York: Van Nostrand Reinhold, 1990.
- [Ben00] L. Benini, A. Bogliolo, and G. De Micheli, “A survey of design techniques for system-level dynamic power management,” *IEEE Transactions on VLSI Systems* 8(3) June (2000): 299–316.
- [Bod95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su, “Myrinet—a gigabit-per-second local-area network,” *IEEE Micro* February (1995): 29–36.
- [Boe84] Barry W. Boehm, “Verifying and validating software requirements and design specifications,” *IEEE Software* 1(1) January (1984): 75–88.
- [Boe87] Barry W. Boehm, “A spiral model of software development and enhancement,” in *Software Engineering Project Management*, 1987, pp. 128–142. Reprinted in Richard H. Thayer and Merlin Dorfman, eds., *System and Software Requirements Engineering*, Los Alamitos, CA: IEEE Computer Society Press, 1990.
- [Boe92] Robert A. Boeller, Samuel A. Stodder, John F. Meyer, and Victor T. Escobedo, “A large-format thermal inkjet drafting plotter,” *Hewlett-Packard Journal* 43(6) December (1992): 6–15.
- [Boo91] Grady Booch, *Object-Oriented Design*. Redwood City, CA: Benjamin/Cummings, 1991.
- [Boo99] Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.
- [Bos07] Robert Bosch GMBH, *Automotive Electrics Automotive Electronics*, fifth edition. Cambridge, MA: Bentley Publishers, 2007.
- [Bra94] K. Brandenburg, G. Stoll, F. Dehery, J. D. Johnston, D. Kerkhof, and E. F. Schroder, “ISO-MPEG-1 audio: A generic standard for coding of high-quality digital audio,” *Journal of the Audio Engineering Society* 42(10) October (1994): 780–792.
- [Cai03] L. Cai and D. Gajski, “Transaction level modeling: an overview,” First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721), 2003, pp. 19–24, <https://doi.org/10.1109/CODESS.2003.1275250>.
- [Cal00] Timothy J. Callahan, John R. Hauser, and John Wawrzynek, “The Garp architecture and C compiler,” *IEEE Computer* 33(4) April (2000): 62–69.

- [Car20] William Carter, editor, *OCP Terminology Guidelines for Inclusion and Openness*, Revision B, December 20, 2020, Open Compute Project.
- [Cat98] Francky Catthoor, Sven Wuytack, Eddy De Greef, Florin Balasa, Lode Nachtergael, and Arnout Vandecappelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Norwell, MA: Kluwer Academic Publishers, 1998.
- [CCI92] CCITT, *Terminal Equipment and Protocols for Telematic Services, Information Technology—Digital Compression and Coding of Continuous-Tone Still Images—Requirements and Guidelines*, Recommendation T.81, September 1992.
- [Cha15] Kenneth Chang, “LightSail, a private spacecraft, goes unexpectedly quiet,” New York Times, June 5, 2015, [http://www.nytimes.com/2015/06/06/science/space/lightsail-solar-sail-bill-nye-glitch.html?\\_r=0](http://www.nytimes.com/2015/06/06/science/space/lightsail-solar-sail-bill-nye-glitch.html?_r=0), accessed August 24, 2015.
- [Cha92] [Cha92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen, “Low-power CMOS digital design,” *IEEE Journal of Solid-State Circuits* 27(4) April (1992): 473–484.
- [Che07] Brian Chess and Jacob West, *Secure Programming With Static Analysis*, Upper Saddle River NJ: Addison-Wesley, 2007.
- [Che11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno. USENIX Security, August 10–12, 2011.
- [Chi15] Richard Chirgwin, “Airbus warns of software bug in A400M transport planes,” *The Register*, 20 May 2015, <http://www.theregister.co.uk/2015/05/20/airbus.warns.of.a400m.software.bug/>.
- [Chi94] M. Chioldo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Ventellini, “Hardware/software co-design of embedded systems,” *IEEE Micro* 14(4) August (1994): 26–36.
- [Cho85] Tsun S. Chow, *Tutorial: Software Quality Assurance: A Practical Approach*. Silver Spring, MD: IEEE Computer Society Press, 1985.
- [CIP10] Camera and Imaging Products Association Standardization Committee, Design rule for Camera File system: DCF Version 2.0 (Edition 2010), April 26, 2010.
- [Cod70] E. F. Codd, “A relational model of data for large shared data banks,” *Communications of the ACM*, 13(6), June 1970, pp. 377–387.
- [Coe14] David Coelho, private communication, August 2, 2014.
- [Coh81] Danny Cohen, “On holy wars and a plea for peace,” *Computer* 14(10) October (1981): 48–54.
- [Cok11] Coker, George & Guttman, Joshua & Loscocco, Peter & Herzog, Amy & Millen, Jonathan & O’Hanlon, Brian & Ramsdell, John & Segall, Ariel & Sheehy, Justin & Sniffen, Brian. (2011). Principles of remote attestation. *Int. J. Inf. Sec.* 10. 63–81. <https://doi.org/10.1007/s10207-011-0124-7>.
- [Col97] Robert R. Collins, “In-circuit emulation,” *Dr. Dobb’s Journal*, September (1997): 111–113.
- [Cop04] M. Coppola, S. Curaba, M. D. Grammatikakis, G. Maruccia and F. Papariello, “OCCN: a network-on-chip modeling and simulation framework,” Proceedings Design, Automation and Test in Europe Conference and Exhibition, 2004, pp. 174–179 Vol.3, <https://doi.org/10.1109/DATE.2004.1269226>.
- [Cra97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko, “Compiling Java just in time,” *IEEE Micro*, 17(3) May/June (1997): 36–43.

- [Cup01] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge, “High performance DRAMs in workstation environments,” *IEEE Transactions on Computers* 50(11) November (2001): 1133–1153.
- [Cus91] Michael A. Cusumano, *Japan’s Software Factories*. New York: Oxford University Press, 1991. Available at <http://drdobbs.com>.
- [Cyp20] Cypress Semiconductor Corporation, *PSoC 6 MCU: CY8C62x8, CY8C62xA Data-sheet*, revised October 9, 2020.
- [Dac05] Dacfey Dzung, Martin Naedele, Thomas P. von Hoff, and Mario Crevatin, “Security for industrial communication systems,” *Proceedings of the IEEE* 93(6) June (2005): 1152–1177.
- [Dah00] Tom Dahlin, “Reach out and touch: Designing a resistive touch screen,” *Circuit Cellar*, 114, January (2000): 20–25.
- [Dav90] Alan M. Davis, *Software Requirements: Analysis and Specification*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- [DiG90] Joseph Di Giacomo, *Digital Bus Handbook*. New York: McGraw-Hill, 1990.
- [Dom05] Jean-Dominique Decotignie, “Ethernet-based real-time and industrial communications,” *Proceedings of the IEEE* 93(6) June (2005): 1102–1117.
- [Dou98] Bruce Powel Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Reading, MA: Addison-Wesley Longman, 1998.
- [Dou99] Bruce Powel Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Reading, MA: Addison-Wesley Longman, 1999.
- [DPO00] DPOF committee, DPOF Version 1.10, July 17, 2000. Available at [http://panasonic.jp/dc/dpof\\_110/](http://panasonic.jp/dc/dpof_110/).
- [Dun13] Michael Dunn, “Toyota’s killer firmware, bad designs and its consequences,” *EDN Network*, October 28, 2013, <http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware-Bad-design-and-its-consequences>.
- [Dut96] Santanu Dutta and Wayne Wolf, “A flexible parallel architecture adapted to block-matching motion-estimation algorithms,” *IEEE Transactions on Circuits and Systems for Video Technology* 6(1) February (1996): 74–86.
- [Dwo15] Morris J. Dworkin, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,” Federal Information Processing Standards (NIST FIPS) — 202, August 4, 2015.
- [Ear97] Richard W. Earnshaw, Lee D. Smith, and Kevin Welton, “Challenges in cross-development,” *IEEE Micro*, July/August (1997): 28–36.
- [ECM13] ECMA International, *The JSON Data Interchange Format*, Standard ECMA-404, 1<sup>st</sup> edition, October 2013.
- [End75] Albert Endres, “An analysis of errors and their causes in system programs,” *IEEE Transactions on Software Engineering* June (1975): 140–149.
- [Ern93] Rolf Ernst, Joerg Henkel, and Thomas Benner, “Hardware-software cosynthesis for microcontrollers,” *IEEE Design and Test of Computers* 10(4) December (1993): 64–75.
- [FAA98] Federal Aviation Administration, *Aeronautical Information Manual*. Washington, DC: Government Printing Office, 1998.
- [Fag76] M. E. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Systems Journal* 15(3) (1976): 219–248.
- [Fal10B] Nicholas Falliere, “Stuxnet introduces the first known rootkit for industrial control systems,” Symantec Official Blog, August 6, 2010, <http://www.symantec.com/connect/blogs/stuxnet-introduces-first-known-rootkit-scada-devices>.

- [Fall11] Nicholas Falliere, Liam O Murchu, and Eric Chien, *W32.Stuxnet Dossier*, version 1.4 February 2011, available at <http://www.symantec.com>.
- [Far08] Shahin Farahani, *Zigbee Wireless Network and Transceivers*, Burlington MA: Newnes, 2008.
- [Fel05] Max Felser, “Real-time ethernet—industry perspective,” *Proceedings of the IEEE* 93(6) June (2005): 1118–1129.
- [Fra19] Dustin Franklin, “Jetson Nano Brings AI Computing to Everyone,” <https://developer.nvidia.com/blog/jetson-nano-ai-computing/>, March 18, 2019.
- [Fra88] Phyllis G. Frankl and Elaine J. Weyuker, “An applicable family of data flow testing criteria,” *IEEE Transactions on Software Engineering* 14(10) October (1988): 1483–1498.
- [Fre11] Freescale Semiconductor, *MPC5602D Microcontroller Reference Manual*, document number MPC5602DRM, rev. 4, 5 May 2011. Available at <http://www.freescale.com>.
- [Fre11B] Freescale Semiconductor, *MPC5676R Product Brief*, document number MPC5676RPB, rev. 2, October 2011. Available at <http://www.freescale.com>.
- [Fru21] Andrei Frumusanu, “The Apple A15 SoC Performance Review: Faster & More Efficient,” Anandtech, October 4, 2021, available at <https://www.anandtech.com>.
- [Fur96] Steve Furber, *ARM System Architecture*. Harlow, England: Addison-Wesley, 1996.
- [Gal92] Bill Gallmeister, “Understanding POSIX.4 and POSIX.4a,” in *Proceedings of the Embedded Systems Conference*, 1992, <https://www.embedded.com/understanding-posix-4-and-posix-4a/#:~:text=POSIX.4%20%28Realtime%20Extensions%20for%20Portable%20Operating%20Systems%2C%20Draft,6%2C%201992%29%20is%20the%20%E2%80%9Cthreads%20extension%E2%80%9D%20to%201003.1>.
- [Gal95] Bill O. Gallmeister, *Posix.4: Programming for the Real World*. Sebastopol, CA: O’Reilly and Associates, 1995.
- [Gar] Dr. Sanjay Garg, “Fundamentals of Aircraft Turbine Engine Control,” NASA Glenn Research Center, undated, [https://www.grc.nasa.gov/WWW/cdtb/aboutus/Fundamentals\\_of\\_Engine\\_Control.pdf](https://www.grc.nasa.gov/WWW/cdtb/aboutus/Fundamentals_of_Engine_Control.pdf).
- [Gar81] John R. Garman, “The ‘bug’ heard ’round the world,” *Software Engineering Notes* 6(5) October (1981): 3–10.
- [Gho97] Somnath Ghosh, Margaret Martonosi, and Sharad Malik, “Cache miss equations: An analytical representation of cache misses.” In *Proceedings of the 11th ACM International Conference in Supercomputing*. ACM Press: New York 1997.
- [Gra03] Mark G. Graff and Kenneth R. van Wyk, *Secure Coding: Principles & Practices*, Sebastopol CA: O’Reilly & Associates, 2003.
- [Gre15] Andy Greenberg, “Hackers remotely kill a Jeep on the highway—with me in it,” [wired.com](http://wired.com), July 21, 2015. Accessed August 24, 2015.
- [GS114] GS1, *EPC Tag Data Standard, version 1.9, Ratified, Nov-2014*.
- [Gup93] Rajesh K. Gupta and Giovanni De Micheli, “Hardware-software cosynthesis for digital systems,” *IEEE Design and Test of Computers* 10(3) September (1993): 29–40.
- [Hal11] Christopher Hallinan, *Embedded Linux Primer: A Practical Real-World Approach*, second edition. Boston: Prentice Hall, 2011.
- [Ham92] Eric Hamilton, *JPEG File Interchange Format*, version 1.02, September 1, 1992.
- [Har03] Michael González Harbour, “Real-Time POSIX: An Overview,” November 2003.
- [Har87] D. Harel, “Statecharts: A visual formalism for complex systems,” *Science of Computer Programming* 8 (1987): 231–274.
- [Has96] Barry G. Haskell, Atul Puri, and Arun N. Netravali, *Digital Video: An Introduction to MPEG-2*. Springer: New York, 1997.

- [Hat88] Derek J. Hatley and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*. New York: Dorset House, 1988.
- [Hel04] Albert Helfrick, *Principles of Avionics*, third edition. Avionics Communications Inc., 2004.
- [Hen06] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, fourth edition. San Francisco: Morgan Kaufmann, 2006.
- [Hen94] J. Henkel, R. Ernst, U. Holtmann, and T. Benner, "Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis." In *Proceedings, ICCAD-94*. Los Alamitos, CA: IEEE Computer Society Press, 1994, pp. 96–100.
- [Hey13] Robin Heydon, *Bluetooth Low Energy: The Developer's Handbook*, Upper Saddle River NJ: Prentice Hall, 2013.
- [Hor96] Joseph R. Horgan and Aditya P. Mathur, "Software testing and reliability," Chapter 13. In *Handbook of Software Reliability Engineering*, ed. Michael R. Lyu, 531–566. Los Alamitos, CA: IEEE Computer Society Press/McGraw-Hill, 1996.
- [How82] W. E. Howden, "Weak mutation testing and the completeness of test cases," *IEEE Transactions on Software Engineering* SE-8(4) July (1982) 371–379.
- [Hsu94] T. Richard Hsueh, Thomas F. Houghton, Joseph F. Maranzano, and Gerald P. Pasterнак, "Software production: From art/craft to engineering," *AT&T Technical Journal* January/February (1994): 59–68.
- [Huf52] David A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE* (40) September (1952): 1098–1101.
- [IEE06] IEEE Computer Society, *IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, IEEE Std 802.15.4-2006, New York: IEEE, 8 September 2006.
- [IEE12] "IEEE Standard for Standard SystemC Language Reference Manual," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005), pp. 1–638, 9 Jan. 2012, <https://doi.org/10.1109/IEEESTD.2012.6134619>.
- [IEE97] IEEE Computer Society, *Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 111: Wireless LAN Medium Access Control and Physical Layer (PHY) specifications*, IEEE Std 802.11-1997, New York: IEEE, 26 June 1997.
- [Inf08] Infineon, *XC2200 Derivatives: 16/32-Bit Single-Chip Microcontroller with 32-Bit Performance, Volume 1 (of 2): System Units*, User's Manual, V2.1, August 2008.
- [Inf12] Infineon, *Infineon SC 2000 Family 16/32-bit µC, Scalable and Highly Integrated 12/32-bit Microcontrollers for Automotive Applications*, February 2012.
- [Int03] Intel, *Intel Advanced+ Boot Block Flash Memory (C3)*, 290645-017, October 2003.
- [Int82] Intel, *Microprocessor and Peripheral Handbook*. Intel, Santa Clara, CA 1982.
- [Int89] Intel, *80960KB Hardware Designer's Reference Manual*. 1989. ISBN 1-55512-100-4.
- [Int91] Intel, *i960 KA/KB Microprocessor Programmer's Reference Manual*. 1991. ISBN 1-55512-137-3.
- [Int96] Intel, Microsoft, and Toshiba, *Advanced Configuration and Power Interface Specification*, 1996. Available at <http://www.teleport.com/~acpi>.
- [Int99] Intel, *Intel StrongARM SA-1100 Microprocessor Technical Reference Manual*, March 1999. Available at <http://www.intel.com>.

- [ISO10] ISO/IEC, *ISO/EC 18033-3:2010: Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers*, ISO/IEC, December 15, 2010.
- [ISO13] International Standards Organization, *ISO/IEC TS 17961:2013, Information technology — Programming languages, their environments and system software interfaces — C secure coding rules*, November 15, 2015. Available at <http://www.iso.org>.
- [ISO18A] International Standards Organization, *ISO 26262-1:2018, Road vehicles – Functional safety – Part 1: Vocabulary*, 2018. Available at <http://www.iso.org>.
- [ISO18B] International Standards Organization, *ISO 26262-2:2018, Road vehicles – Functional safety – Part 2: Management of functional safety*, 2018. Available at <http://www.iso.org>.
- [ISO18C] International Standards Organization, *ISO 26262-9:2018, Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*, 2011. Available at <http://www.iso.org>.
- [ISO94] International Standards Organization, ISO/IEC 10918-1, *Information Technology—Digital Compression and Coding of Continuous-Tone Still Images*, 1994. Available at <http://www.iso.org>.
- [Jac03] Bruce Jacob, “A case for studying DRAM issues at the system level,” *IEEE Micro* 23(4) July-August (2003): 44–56.
- [Jag95] Dave Jaggar, ed., *Advanced RISC Machines Architectural Reference Manual*. London: Prentice Hall, 1995.
- [Jen95] Michael G. Jenner, *Software Quality Management and ISO 9001: How to Make Them Work for You*. New York: John Wiley and Sons, 1995.
- [Jon78] T. C. Jones, “Measuring programming quality and productivity,” *IBM Systems Journal* 17(1) (1978): 39–63.
- [Kar03] G. Karsai, J. Sztipanovits, A. Ledeczi and T. Bapty, “Model-integrated development of embedded software,” in Proceedings of the IEEE, vol. 91, no. 1, pp. 145–164, Jan. 2003, <https://doi.org/10.1109/JPROC.2002.805824>.
- [Kar06] Holger Karl and Andreas Willig, *Protocols and Architectures for Wireless Sensor Networks*. New York: John Wiley and Sons, 2006.
- [Kas79] J. M. Kasson, “The ROLM Computerized Branch Exchange: An advanced digital PBX,” *IEEE Computer* June (1979): 24–31.
- [Kem98] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach, “A decompression core for PowerPC,” *IBM Journal of Research and Development* 42(6) November (1998): 807–812.
- [Ker88] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, second edition. New York: Prentice Hall, 1988.
- [Kla19] Jochen Klaus-Wagenbrenner, “Zonal EE Architecture: Towards a Fully Automotive Ethernet-Based Vehicle Infrastructure,” Visteon, September 24, 2019.
- [Klo15] Irene Klotz, “Pluto probe glitch traced to software timing flaw,” [discovery.com](http://news.discovery.com/space/pluto-probe-glitch-traced-to-software-timing-flaw-150607.htm), July 6, 2015, <http://news.discovery.com/space/pluto-probe-glitch-traced-to-software-timing-flaw-150607.htm>, accessed August 24, 2015.
- [Kog81] Peter M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.
- [Koh78] Loren M. Kohnfelder, *Towards a Practical Public-key Cryptosystem*, Bachelor of Science thesis, Massachusetts Institute of Technology, May 1978.
- [Koo10] Philip Koopman, *Better Embedded System Software*. Pittsburgh: Drumnadrochit Press, 2010.

- [Koo14] Prof. Phil Koopman, “A case study of Toyota unintended acceleration and software safety,” presentation slides, September 18, 2014, [http://users.ece.cmu.edu/~koopman/pubs/koopman14\\_toyota\\_ua\\_slides.pdf](http://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf).
- [Kop03] Hermann Kopetz and Gunther Bauer, “The time-triggered architecture,” *Proceedings of the IEEE*, 91(1), January 2003, pp. 112–126.
- [Kop97] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston: Kluwer Academic Publishers, 1997.
- [Kos10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage. IEEE Symposium on Security and Privacy, Oakland, CA, May 16–19, 2010.
- [Lev86] Nancy G. Leveson, “Software safety: Why, what, and how,” *Computing Surveys* 18(2) June (1986): 125–163.
- [Lev93] Nancy G. Leveson and Clark S. Turner, “An investigation of the Therac-25 accidents,” *IEEE Computer* July (1993): 18–41.
- [Lev94] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese, “Requirements specification for process-control systems,” *IEEE Transactions on Software Engineering* 20(9) September (1994): 684–707.
- [Li97A] Yanbing Li and Wayne Wolf, “Scheduling and allocation of multirate real-time embedded systems.” In *Proceedings, ED&TC '97*. Los Alamitos, CA: IEEE Computer Society Press, 1997, pp. 134–139.
- [Li97B] Yanbing Li and Wayne Wolf, “A task-level hierarchical memory model for system synthesis of multiprocessors.” In *Proceedings, 34th Design Automation Conference*. ACM Press: New York 1997, pp. 153–156.
- [Li97C] Yanbing Li, Miodrag Potkonjak, and Wayne Wolf, “Real-time operating systems for embedded computing.” In *Proceedings, ICCD '97*. Los Alamitos, CA: IEEE Computer Society Press, 1997.
- [Li97D] Yau-Tsun Steven Li and Sharad Malik, “Performance analysis of embedded software using implicit path enumeration,” *IEEE Transactions on CAD/ICAS* 16(12) December (1997): 1477–1487.
- [Li98] Yanbing Li and Joerg Henkel, “A framework for estimating and minimizing energy dissipation of embedded HW/SW systems.” In *Proceedings, DAC '98*. New York: ACM Press, 1998, pp. 188–193.
- [Li99] Yanbing Li and Wayne Wolf, “A task-level hierarchical memory model for system synthesis of multiprocessors,” *IEEE Transactions on CAD* 18(10) October (1999): 1405–1417.
- [Lin04] Chang Hong Lin, Tiehan Lv, Wayne Wolf, and I. Burak Ozer, “A peer-to-peer architecture for distributed real-time gesture recognition.” In *Proceedings, International Conference on Multimedia and Exhibition*, IEEE, Piscataway NJ, 2004, vol. 1, pp. 27–30.
- [Liu00] Jane W. S. Liu, *Real-Time Systems*. Prentice Hall, Upper Saddle River NJ, 2000.
- [Liu12] Yuan Liu, Dean K. Frederick, Jonathan A. DeCastro, Jonathan S. Litt and William W. Chan, *User’s Guide for the Commercial Modular Aero-Propulsion System Simulation (C-MAPSS)*, Version 2, NASA/TM-2012-217432, March 2012.
- [Liu17] S. Liu, J. Tang, Z. Zhang and J. Gaudiot, “Computer Architectures for Autonomous Driving,” in *Computer*, vol. 50, no. 8, pp. 18–25, 2017, <https://doi.org/10.1109/MC.2017.3001256>.
- [Liu73] C. L. Liu and James W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM* 20(1) January (1973): 46–61.

- [Los97] Pete Loshin, *TCP/IP Clearly Explained*, second edition. New York: Academic Press, 1997.
- [Lu82] David Jun Lu, “Watchdog processors and structural integrity checking,” *IEEE Transactions on Computers*, C-31(7), July 1982, pp. 681–685.
- [Lyu96] Michael R. Lyu, ed., *Handbook of Software Reliability Engineering*. Los Alamitos, CA: IEEE Computer Society Press/McGraw-Hill, 1996.
- [Mad97] J. Madsen, J. Grode, P. V. Knudsen, M. E. Peterson, and A. Haxthausen, “LYCOS: The Lungby Co-Synthesis System,” *Design Automation for Embedded Systems* 2(2) March (1997): 165–195.
- [Mah88] Aamer Mahmood and E. J. McCluskey, “Concurrent error detection using watchdog processors—a survey,” *IEEE Transactions on Computers*, 37(2), February 1988, pp. 160–174.
- [Mal96] Sharad Malik, Wayne Wolf, Andrew Wolfe, Yao-Tsun Steven Li, and Ti-Yen Yen, “Performance analysis of embedded systems.” In *Hardware-Software Co-Design*, eds. G. De Micheli and M. Sami. Boston: Kluwer Academic Publishers, 1996.
- [Man99] William H. Mangione-Smith, “Technical challenges for designing personal digital assistants,” *Design Automation for Embedded Systems* 4(1) January (1999): 23–40.
- [Mar78] John Marley, “Evolving microprocessors which better meet the needs of automotive electronics,” *Proceedings of the IEEE* 66(2) February (1978): 142–150.
- [McC76] T. J. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering* 2 (1976): 308–320.
- [McD13] Geoff McDonald, Liam O Murchu, Stephen Doherty, and Eric Chien, *Stuxnet 0.5: The Missing Link*, version 1.0, February 25, 2013, available at [www.symantec.com](http://www.symantec.com).
- [McD98] Charles E. McDowell, Bruce R. Montague, Michael R. Allen, Elizabeth A. Baldwin, and Marcelo E. Montoreano, “Javacam: Trimming Java down to size,” *IEEE Internet Computing* May/June (1998): 53–59.
- [Meb92] Alfred Holt Mebane IV, James R. Schmedake, Iue-Shuenn Chen, and Anne P. Kadonaga, “Electronic and firmware design of the HP DesignJet Drafting Plotter,” *Hewlett-Packard Journal* 43(6) December (1992): 16–23.
- [Met97] Hufeza Metha, Robert Michael Owens, Mary Jane Irwin, Rita Chen, and Debashree Ghosh, “Techniques for low energy software.” In *Proceedings, 1997 International Symposium on Low Power Electronics and Design*. New York: ACM Press, 1997, pp. 72–75.
- [Mic00] Micron Technology, Inc., “512 Mb Synchronous SDRAM,” available at <http://www.micron.com/products/dram/sdram>.
- [Mic00] Microsoft Corporation, Microsoft Extensible Firmware Initiative FAT32 File System Specification, version 1.03, December 6, 2000.
- [Mic07] Microchip Technology Inc., *PICmicro™ Mid-Range MCU Family Reference Manual*, December 1997. Available at <http://www.microchip.com>.
- [Mic09] Microchip Technology Inc., *PIC16F882/883/884/886/887 Data Sheet*, 2009. Available at <http://www.microchip.com>.
- [Mic17] Microsoft, *Introduction to Code Signing*, August 15, 2017, available at <https://docs.microsoft.com>.
- [Mic97B] Microchip Technology Inc., *PWM, A Software Solution for the PIC16CXXX*, 1997. Available at <http://www.microchip.com>.
- [Mil01] Brent A. Miller and Chatschik Bisdikian, *Bluetooth Revealed*, Upper Saddle River NJ: Prentice Hall PTR, 2001.
- [Min95] Mindshare, Inc., Tom Shanley and Don Anderson, *PCI System Architecture*, third edition. Reading, MA: Addison-Wesley, 1995.

- [MIS08] The Motor Industry Software Reliability Association, *MISRA C++: 2008, Guidelines for the use of the C++ language in critical systems*, Warwickshire UK: MIRA Limited, June 2008.
- [MIS12] The Motor Industry Software Reliability Association, *MISRA C: 2012, Guidelines for the use of the C language in critical systems*, Warwickshire UK: MIRA Limited, March 2013.
- [Mor07] Michael J. Morgan, “Boeing B-777,” Chapter 9. In *Digital Avionics Handbook, second edition: Avionics Development and Implementation*, ed. Cary R. Spitzer. Boca Raton, FL: CRC Press, 2007.
- [Muc97] Steven S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann, 1997.
- [Mye79] G. Myers, *The Art of Software Testing*. New York: John Wiley and Sons, 1979.
- [Nak05] Junichi Nakamura, ed., *Image sensors and Signal Processing for Digital Still Cameras*. CRC Press, Danvers MA, 2005.
- [NAS21A] NASA/JPL, “Mars Helicopter Flight Delayed to No Earlier than April 14,” Status Update, April 10, 2021, Mars Helicopter Flight Delayed to No Earlier than April 14 - NASA Mars
- [NAS21B] NASA/JPL, “Work Progresses Toward Ingenuity’s First Flight on Mars,” Status Update, April 12, 2021, Work Progresses Toward Ingenuity’s First Flight on Mars - NASA Mars.
- [Nat19] National Transportation Safety Board, *Collision Between Vehicle Controlled by Developmental Driving System and Pedestrian, Tempe, Arizona, March 18, 2018*, Accident Report, NTSB/HAR-19/03, PB2019-101402, Notation 59392, November 19, 2019.
- [NIS15] Information Technology Laboratory, National Institute of Standards and Technology, *Secure Hash Standard (SHS)*, FIPS PUB 180-4, August 2015.
- [NXP11] NXP Semiconductor, “Using the LPC13xx low power modes and wake-up times on the LPCXpresso,” Application note AN10973, rev. 2, January 6, 2011.
- [NXP12] NXP Semiconductor, “LPC1311/13/42/43, 32-bit ARM Cortex-Me microcontroller; up to 32 kB flash and 8 kB SRAM; USB device,” Product data sheet, rev. 6, June 6, 2012.
- [Obe99] James Oberg, “Why the Mars probe went off course,” *IEEE Spectrum* December (1999): 34–39.
- [OMG17] OMG, *OMG Systems Modeling Language<sup>TM</sup>*, version 1.5, formal/2017-05-01, May 2017.
- [OMG19] OMG, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*, Version 1.2, formal/19-04-01, April 2019.
- [Ope18] The Open Group Base Specifications Issue 7, 2018 edition, IEEE Std. 1003.1-2017 (Revision of IEEE Std. 1003.1-2008), 2018.
- [Owe15] Jeffrey J. Owens, “The design of innovation that drives tomorrow,” keynote presentation, 52<sup>nd</sup> Design Automation Conference, June 9, 2015.
- [Pag15] Pierluigi Paganini, “FBI: researcher hacked plane in-flight, causing it to ‘climb’,” Security Affairs, May 16, 2015, <http://securityaffairs.co/wordpress/36872/cyber-crime/researcher-hacked-flight.html>.
- [Pag15B] Pierluigi Paganini, “Airbus—be aware a software bug in A400M can crash the plane,” Security Affairs, May 20, 2015, <http://securityaffairs.co/wordpress/36972/security/airbus-software-bug-a400m.html>.
- [Pan99] Preeti Ranjan Panda, Nikil Dutt, and Alexandru Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Norwell, MA: Kluwer Academic Publishers, 1999.

- [Pat98] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, second edition. San Francisco: Morgan Kaufmann, 1998.
- [Phi89] “Using the 8XC751 microcontroller as an I<sup>2</sup>C bus master,” Philips Application Note AN422, September 1989, revised June 1993. In *Application Notes and Development Tools for 80C51 Microcontrollers*, Philips Semiconductors, 1995.
- [Phi92] “The I<sup>2</sup>C bus and how to use it (including specification),” January 1992. In *Application Notes and Development Tools for 80C51 Microcontrollers*, Philips Semiconductors, 1995.
- [Phi96] Philips Semiconductors, *I<sup>2</sup>S bus specification*, February 1986, revised June 5, 1996.
- [Pil05] Dan Pilone with Neil Pitman, *UML 2.0 In A Nutshell*. Sebastopol, CA: O’Reilly Media, 2005.
- [Pre97] Roger S. Pressman, *Software Engineering: A Practitioner’s Approach*. New York: McGraw-Hill, 1997.
- [Qua07] Gang Quan and Xiaobo Sharon Hu, “Static DVFS scheduling,” Chapter 10 in Joerg Henkel and Sri Parameswaran, eds., *Designing Embedded Processors: A Low Power Perspective*, Berlin: Springer, 2007.
- [Qua15] Qualcomm, *QCA4004*, San Diego: Qualcomm, 2015.
- [Rat96] Kamlesh Rath and James W. Wendorf, “Set-top box control software: A key component in digital video,” *Philips Journal of Research* 50(1/2) (1996): 185–199.
- [Rho97] David L. Rhodes and Wayne Wolf, “Allocation and data arrival design of hard real-time systems.” In *Proceedings, ICCD ’97*. Los Alamitos, CA: IEEE Computer Society Press, 1997.
- [Rol15] Rolls-Royce, *The Jet Engine*, fifth edition, Wiley, 2015.
- [RTC11] Radio Technical Commission for Aeronautics, **DO-178C Software Considerations in Airborne Systems and Equipment Certification**, Committee: SC-205, 12/13/2011.
- [Rum91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- [SAE18] SAE International, *Surface Vehicle Recommended Practice, (R) Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, J3016, revised June 2018.
- [Sar16] Roberto Saracco, “Guess what requires 150 million lines of code...,” *IEEE Future Directions*, January 13, 2016, Guess what requires 150 million lines of code.... – IEEE Future Directions
- [Sas91] Steven J. Sasson and Robert G. Hills, “Electronic still camera utilizing image compression and digital storage,” U. S. Patent 5,016,107, May 14, 1991.
- [Sch94] Charles H. Schmauch, *ISO 9000 for Software Developers*. Milwaukee: ASQC Quality Press, 1994.
- [Sch96] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, second edition, New York: John Wiley & Sons, 1996.
- [Sea14] Robert C. Seacord, *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*, second edition, Upper Saddle River NJ: Pearson, 2014.
- [Seg21] Simon Segars, “Armv9: The Future of Specialized Compute,” March 30, 2021, Armv9: The Future of Specialized Compute - Arm Blueprint
- [Seg97] Simon Segars, “ARM7TDMI power consumption,” *IEEE Micro* July/August (1997): 12–19.
- [SEI99] Software Engineering Institute, “Capability Maturity Model (SW-CMM) for Software,” 1999. Available at [www.sei.cmu.edu/cmm/cmm.html](http://www.sei.cmu.edu/cmm/cmm.html).

- [Sel94] Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-Time Object-Oriented Modeling*. New York: John Wiley and Sons, 1994.
- [Ser18] Dimitrios Serpanos and Marilyn Wolf, *Internet-of-Things (IoT) Systems*, Kluwer, 2018.
- [Sha89] Alan C. Shaw, “Reasoning about time in higher-level language software,” *IEEE Transactions on Software Engineering* 15 July (1989): 875–889.
- [Shl92] Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*. New York: Yourdon Press Computing Series, 1992.
- [Sie98] Daniel P. Siewiorek and Robert S. Swartz, *Reliable Computer Systems: Design and Evaluation*, third edition, A. K. Peters/CRC Press, 1998.
- [Slo04] Andrew N. Sloss, Dominic Symes, and Chris Wright, *ARM System Developer’s Guide: Designing and Optimizing System Software*. San Francisco: Morgan Kaufman, 2004.
- [Spa99] Peter Spasov, *Microcontroller Technology: The 68HC11*, third edition. Upper Saddle River, NJ: Prentice Hall, 1999.
- [Spi07] Cary R. Spitzer, ed., *Digital Avionics Handbook, second edition: Avionics Development and Implementation*. Boca Raton, FL: CRC Press, 2007.
- [Sri94] Amitabh Srivastava and Alan Eustace, “ATOM: A system for building customized program analysis tools,” Digital Equipment Corp., WRL Research Report 94/2, March 1994. Available at [www.research.digital.com](http://www.research.digital.com).
- [Sta97A] William Stallings, *Data and Computer Communication*, fifth edition. Upper Saddle River, NJ: Prentice Hall, 1997.
- [Sta97B] J. Staunstrup and W. Wolf, eds., *Hardware/Software Co-Design: Principles and Practice*. Boston: Kluwer Academic Publishers, 1997.
- [Sto95] Thomas M. Stout and Theodore J. Williams, “Pioneering work in the field of computer process control,” *IEEE Annals of the History of Computing* 17(1) (1995): 6–18.
- [Str97] Bjarne Stroustrup, *The C++ Programming Language*, third edition. Reading, MA: Addison-Wesley Professional, 1997.
- [Tay06] Jim Taylor, *DVD Demystified*, third edition. New York: McGraw Hill, 2006.
- [Tex00] Texas Instruments, *TMS320VC5510/5510A Fixed-Point Digital Signal Processors Data Manual*, document SPRS076N, June 2000, revised July 2006.
- [Tex00B] Texas Instruments, *TMS320C55x DSP Functional Overview*, SPRU312, June 2000.
- [Tex01] Texas Instruments, *TMS320C55x DSP Programmer’s Guide*, Preliminary Draft, document SPRU376A, August 2001.
- [Tex02] Texas Instruments, *TMS320C55x DSP Mnemonic Instruction Set Reference Guide*, document SPRU374G, October 2002.
- [Tex04] Texas Instruments, *TMS320C55x DSP CPU Reference Guide*, document SPRU371F, February 2004.
- [Tex04B] Texas Instruments, *TMS320VC5510 DSP Instruction Cache Reference Guide*, SPRU576D, June 2004.
- [Tex10] Texas Instruments, *TMS320C64x+ DSP CPU and Instruction Set Reference Guide*, SPRU732J, July 2010.
- [Tex11] Texas Instruments, *TMS320DM816x DaVinci Digital Media Processors Technical Reference Manual*, SPRUGX8, March 1, 2011.
- [Tex11B] Texas Instruments, *TMS320DM816x DaVinci Digital Media Processors*, SPRS614, March 1, 2011.
- [Tex11c] Texas Instruments, *Automotive Central Body Controller*, <http://focus.ti.com/docs/solution/folders/print/490.html> [3/30/2011 9:20:04 PM], March 30, 2011.

- [Tex14] Texas Instruments, *TM4C Microcontrollers*, 2014, SPMT285d.
- [Tex19] Texas Instruments, *AFE7422 Dual-channel, RF-sampling AFE with 14-bit, 9-GSPS DACs and 14-bit, 3-GSPS ADCs*, SLAES9A, October 2018, revised January 2019.
- [Tex20A] Texas Instruments, *ADC12xJ1600-Q1 Quad/Dual/Single Channel, 1.6 -GSPS, 12-bit, Analog-to-Digital Converter (ADC) with JESD204C Interface*, SBAS960A, February 2020, revised August 2020.
- [Tex20B] Texas Instruments, *DACx1001 20-Bit, 18-Bit, and 16-bit, Low-Noise, Ultra-Low Harmonic Distortion, Fast-Settling, High-Voltage Output, Digital-to-Analog Converters (DACs)*, SLASEL0B, October 2019, revised June 2020.
- [Tex21] Texas Instruments, *ADS126x 32-bit, Precision, 38-kSPS, Analog-to-Digital Converter (ADC) with Programmable Gain Amplifier (PGA) and Voltage Reference*, SBAS6611C, February 2015, revised May 2021.
- [Tha90] Richard H. Thayer and Merlin Dorfman, eds., *System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1990.
- [Tho15] Mark Thompson and Ivana Kottasova, “Volkswagen scandal widens,” CNN Money, September 22, 2015, <http://money.cnn.com/2015/09/22/news/vw-recall-diesel/index.html>.
- [Tiw94] Vivek Tiwari, Sharad Malik, and Andrew Wolfe, “Power analysis of embedded software: A first step toward software power minimization,” *IEEE Transactions on VLSI Systems* 2(4) December (1994): 437–445.
- [Toy] Toyota Motor Sales, “Engine Controls Part #2 – ECU Process and Output Functions,” date unknown, downloaded from <facultyfiles.deanza.edu/gems/waltonjohn/Toyotaignition.pdf>.
- [Tru98] T. E. Truman, T. Pering, R. Doering, and R. W. Brodersen, “The InfoPad multimedia terminal: A portable device for wireless information access,” *IEEE Transactions on Computers* 47(10) October (1998): 1073–1087.
- [Ugo86] Michel Ugon, “Single-chip microprocessor with on-board modifiable memory,” U. S. Patent 4,382,279, May 3, 1983.
- [van97] Albert van der Werf, Font Brüls, Richard Kleinhorst, Erwin Waterlander, Matt Verstraeler, and Thomas Friedrich, “I.McIC: A single-chip MPEG2 video encoder for storage,” In *ISSCC '97 Digest of Technical Papers*. Castine, ME: John W. Wuorinen, 1997, pp. 254–255.
- [Vem20] Arun T. Vemuri, “Processing the advantages of zone architecture in automotive,” Texas Instruments, December 11, 2020, <https://e2e.ti.com>.
- [Vos89] L. D. Vos and M. Stegherr, “Parameterizable VLSI architectures for the full-search block-matching algorithm,” *IEEE Transactions on Circuits and Systems* 36(10) October (1989): 1309–1316.
- [Wal07] Randy Walter and Chris Watkins, “Genesis Platform,” Chapter 12. In *Digital Avionics Handbook, second edition: Avionics Development and Implementation*, ed. Cary R. Spitzer. Boca Raton, FL: CRC Press, 2007.
- [Wal97] Dave Walsh, “Reducing system cost with software modems,” *IEEE Micro* July/August (1997): 37–55.
- [Wat17] Dr. Conal Watterson, *Controller Area Network (CAN) Implementation Guide*, Application Note AN-1123, Analog Devices, 2017.
- [Wat96] Arthur H. Watson and Thomas J. McCabe, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, September 1996.
- [Wei91] Mark Weiser, “The computer for the 21<sup>st</sup> century,” *Scientific American*, 265(3), September 1991, pp. 94–104. Reprinted in *ACM SIGMOBILE Mobile Computing and Communications Review — Special Issue Dedicated to Mark Weiser*, 3(3), July 1999, pp. 3–11.

- [Whi80] L. J. White and E. I. Cohen, “A domain strategy for program testing,” *IEEE Transactions on Software Engineering* 14(6) June (1980): 868–874.
- [Wol08] Wayne Wolf, *Modern VLSI Design: IP-Based System Design*, fourth edition. Upper Saddle River, NJ: Prentice Hall, 1998.
- [Wol08B] Wayne Wolf, Ahmed A. Jerraya, and Grant Martin, “Multiprocessor System-on-Chip (MPSoC) Technology,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(10) October (2008): 1701–1713.
- [Wol15] Marilyn Wolf, Mihaela van der Schaar, Honggab Kim, and Jie Xu, “Caring analytics for adults with special needs,” *IEEE Design & Test*.
- [Wol18] M. Wolf and D. Serpanos, “Safety and Security in Cyber-Physical Systems and Internet-of-Things Systems,” in *Proceedings of the IEEE*, vol. 106, no. 1, pp. 9–20, Jan. 2018. <https://doi.org/10.1109/JPROC.2017.2781198>.
- [Wol92] Wayne Wolf, “Expert opinion: In search of simpler software integration,” *IEEE Spectrum*, 29(1) January (1992): 31.
- [Wol96] Wayne Wolf, Andrew Wolfe, Steve Chinatti, Ravi Koshy, Gary Slater, and Spencer Sun, “Lessons from the design of a PC-based private branch exchange,” *Design Automation for Embedded Systems* 1(4) (1996): 297–314.
- [Wol97] Wayne Wolf, “Hardware/software co-design for multimedia.” In *Advanced Signal Processing: Algorithms, Architectures, and Implementations VII*, Society of Photo-Optical Instrumentation Engineers, Bellingham WA, 1997.
- [Wu11] Xin Wu, Prabhuram Gopalan, and Greg Lara, *Xilinx 28 nm Next Generation FPGA Overview*, WP312 (v1.1), March 26, 2011.
- [Xil11] Xilinx, *ZYNQ-7000 EPP Product Brief*, 2011.
- [Yaf11] YAFFS, <http://yaffs.net>, accessed February 14, 2012.
- [Yag08] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum, *Building Embedded Linux Systems*, second edition. Sebastopol, CA: O’Reilly, 2008.
- [Yan89] Kun-Min Yang, Ming-Ting Sun, and Lancelot Wu, “A family of VLSI designs for the motion compensation block-matching algorithm,” *IEEE Transactions on Circuits and Systems* 36(10) October (1989): 1317–1325.
- [Yen98] Ti-Yen Yen and Wayne Wolf, “Performance analysis of distributed embedded systems,” *IEEE Transactions on Parallel and Distributed Systems* 9(11) November (1998): 1125–1136.
- [Zax12] David Zax, “Many cars have a hundred million lines of code,” *MIT Technology Review*, December 3, 2012.

# Index

'Note: Page number followed by "f" indicate figure and "t" indicate table.'

## A

- ABS. *See* Antilock braking system (ABS)  
Absolute addresses, 84, 236, 242  
Accelerators, 458–460, 459f  
  execution time, 462, 462f  
  performance analysis, 460–465, 461f  
Accumulator, 80  
  architecture, 80  
  C55x DSP, 80  
ACPI. *See* Advanced configuration and power interface (ACPI)  
Activation record, 73  
Active rangefinding approach, 297  
Activity diagram (act), 395  
Address, 54, 61, 166, 236, 469  
  bus, 167–168  
  data byte, 33  
  Internet, 417  
  modes, 69  
  range, for PIC16F microprocessor, 77  
  spaces, in TMS320C55x, 84f  
  translation, 125–129  
  transmissions, 469  
  types, 84  
Ad hoc networks, 419  
  services, 419–420  
Admission control, 420–421  
Advanced configuration and power interface (ACPI), 200  
Aircraft electronics, 439  
Alarm clock, system design  
  component design and testing, 209  
  requirements, 202–204, 203f  
  specification, 204–207, 204f–207f  
  system architecture, 208–209, 209f  
  system integration and testing, 210  
Allocate computations, 465  
Allocating processes, on distributed embedded systems, 465–467  
AMBA high-performance bus (AHB), 176  
AMBA peripherals bus (APB), 176  
Analog physical objects, in train control System, 37f  
Antilock braking system (ABS), 3  
Aperiodic process, 327  
Application interface, 471–472  
Application layer, 416  
Arbitration, 444–445  
Architecture design, 18–20, 381  
  system analysis and, 398–400  
AR indirect addressing, 85  
Arithmetic expression, 246–248  
Arm, 98, 100, 176, 239–240  
  address translation, 125–129  
  AMBA bus system, elements of, 177f  
  assembly language, example, 57–58  
  coprocessors, 118  
  data processing instruction, format of, 58f  
  evaluation module, 183, 186f  
  execution time of for loop, 133–134  
  load-store instructions and pseudo-operations, 65–66, 66f  
  memory-mapped I/O, 100  
  processor, 60  
    condition codes, 70f  
    data operations, 61–69, 62f  
    flow of control, 78–79  
    instructions, 64f  
    memory organization, 60–61, 61f  
    models, 76  
    procedure calls in, 73  
    programming model, 62, 62f  
ARM7 pipeline, 131  
ARM Cortex-A8, 183  
ARM Procedure Call Standard (APCS), 74–75  
Array padding, 272–273  
Assemblers, 236–240  
Assembly languages, 219, 336  
  for C55x, 131  
  features, 57  
  program, 333  
Assembly source code, 333  
Asynchronous connectionless (ACL) packets, 424  
Asynchronous input, 324  
Asynchronous interrupts, 116  
Attribute Protocol Layer, 424  
Auto-indexing, 69  
Automated driving system (ADS), 450–451  
Automatic stability control system, 3  
Automobile, 3  
  engine controllers, 325–326  
Automotive engine control, 325–326  
Automotive Ethernet, 447  
Average-case execution time, 262  
Avionics, 439

## B

- Backup flight control system (BFS), 329  
Bandwidth, 476  
  and memory access times, 199

- Bandwidth (*Continued*)
  - as performance, 195
  - bus, 165–166, 476
  - component, 265–266
- Base class, 24–25
- Baseline packet, 33
- Base-plus-offset addressing, 69
- Basic block, 230–231, 231f
- Basic service set (BSS), 427
- Baud rate, 293
- Bayer pattern, 297–298, 298f
  - interpolation, 365
- Beacon transmissions, 421, 422f
- BeagleBoard, 183, 184f
  - intellectual property, 187
- Best-case execution time, 262
- BFS. *See* Backup flight control system (BFS)
- Big-endian mode, 56
- Bit manipulations, 152
- 8-Bit microcontroller, 2, 358–359
- Black-and-white display, 24–25
- Black-box testing, 288–289
- Block definition diagram (bdd), 395
- Blocking, 461
- Block motion estimation, 472–473, 474f
- Block motion search parameters, 475f
- Block repeat registers, 82
- Blocks, cache, 120
- Bluetooth, 423–425, 447
- Bluetooth low energy (BLE), 423–425, 425f
- BMW 850i, microprocessors used in, 3
- Bottom-up design, 11–12
- Branches, 59, 248–251
  - penalty, 132–133
  - testing, 284–285
- Breakpoint, 190
- Bugs, 20–21, 289–290, 389–390, 390f
- Burst access, 177–178
- Bus, 165–176
  - arbitration, 445, 471
  - bandwidth, 165–166
  - bridges, 175–176
  - burst transfers, 176
  - components of, 167
  - configurations, 175–176, 175f–177f
  - direct memory access (DMA), 172–174, 173f–175f
  - disconnected transfers, 171
  - grant, 173
  - interface module, 479–480
  - organization and protocol, 165–172, 165f–172f
  - reads and writes, 167
  - request, 171
- signals, bundle of, 165
- state diagrams for read transaction, 172f
- timing diagram, 167–168, 168f–169f
- transaction, 469
  - on inter-integrated circuit bus, 470f
- Bus-based system, performance bottlenecks in, 194–195
- Busy-wait I/O, 101–102
- Byte, 33
  - format, 470
  - organizations, within ARM word, 61f
- C**
- Cache, 119–125, 266
  - effect, 266
  - controllers, 119, 119f
  - direct-mapped *vs.* set-associative, 120
  - hit, 120
  - memory system, 120
  - microprocessor architectures, 119
  - miss, 119
  - optimizations, 252–260
  - organization, 120
  - set-associative, 121–122, 122f
  - two-level system, 120, 120f
- Call event, 27
- CAD. *See* Computer-aided design (CAD)
- CAN controller, 446, 446f
- CAN data frame format, 444–445, 445f
- Capability maturity model (CMM), 402
- Car subsystems, interactions, 438–439
- Carrier sense multiple access with collision avoidance (CSMA-CA), 426
- CAS. *See* Collision avoidance system (CAS)
- CDFG. *See* Control/data flow graph (CDFG)
- CDP indirect addressing, 85–86
- Certification authority, 290
- Certification process, 439
- Chrominance, 297
- Circular buffers, 81–82, 222–223, 222f
- CISCs. *See* Complex instruction set computers (CISCs)
- Classes, responsibilities and collaborators (CRC) cards, 385, 386f
- Clear-box testing, 281–288
  - basis paths, 283, 284f
  - branch testing, 284–285
  - control/data flow graph (CDFG), 230–231
  - cyclomatic complexity, 284, 369f
  - data flow testing, 286
  - def-use pair, 286–287
  - domain testing, 285–286, 286f
  - execution path, 282

- incidence matrix, 283
- loops, testing, 287
- CMM. *See* Capability maturity model (CMM)
- CMOS. *See* Complementary metal oxide semiconductor (CMOS)
- Code generator, 244
- Code modules, 241
- Code motion, 270, 271f
- Coding, bug, 389–390
- Coefficient data pointer registers, 81
- Coefficient indirect addressing, 86
- Coefficient matrix, in zig-zag pattern, 301, 301f
- Collision avoidance system (CAS), 392–394
- Color
  - channels, 300
  - filter array, 297–298, 298f
  - spaces, 299–300
  - temperature, 298–299
- Common intermediate format, 476
- Common object file format (COFF), 240
- Communications, 198
- Compare instruction, 87
- Complementary metal oxide semiconductor (CMOS), 137
- Complex instruction set computers (CISCs), 55
- Compilation, 188, 235
  - methods, 244–252
    - arithmetic expression, 246–248
    - control flow diagram, 246–248
    - data structures, 251
    - linkage mechanism, 244
    - stack pointer (sp), 244–245
    - two-dimensional arrays, 251
  - process, 243–244
- Compiler
  - optimizations, 252–260
    - dead code elimination, 254
    - function inlining, 252
    - graph coloring, 255, 256f
    - instruction selection, 258–259
    - loop distribution, 253–254
    - loop fusion, 253–254
    - loop unrolling, 253
    - operator scheduling, 256
    - outlining, 252–253
    - register allocation, 254
    - reservation table, instruction scheduling, 258, 258f
    - template matching, code generation by, 259–260, 259f
- Component bandwidth, 198
- Component design, testing and, 20, 209, 295–296, 371
- Component suppliers, 448
- Compression algorithm, 323
- Computational kernels, 458
- Computer-aided design (CAD), 383
- Computing platforms, 161–165
  - choosing, 184–187
  - designing with, 183–193
  - hardware components, 162–164, 162f
  - software components, 164–165, 164f
- Console class, 35–36
- Context, 103
- Context switching mechanism, 336
- Contrast detection, 297
- Control algorithm, 325–326
- Control channel, 447
- Control dependencies, 59
- Control flow, 81
  - diagram, 248, 248f
- Control flow-oriented testing, 284
- Control/data flow graph (CDFG), 220, 230, 233–234, 234f–235f
- Controller class, 36, 305
- Control stall, 132–133
- Coprocessors, 118, 459
- Core processor modules (CPMs), 443
- Cortex, 76
- C programming language
  - assignments, in ARM instructions, 68
  - coding guidelines, 89–90
  - functions, 73
- CPU
  - accelerator, 458, 459f
  - bus. *See* Bus
  - cache performance, 136
  - direct memory access (DMA) bus transaction on, 172
  - performance, 130–136
  - pipeline
    - ARM7, 131, 131f
    - PIC16F, 133–134
    - RISC machines, 132
    - stalls, 132
  - power consumption
    - energy vs. power, 136
    - power state machine, 138
    - static vs. dynamic power management, 137–138
    - utilization, 330
- Cross-compiler, 188
- Current program status register (CPSR), 63
- C55x
  - interrupts, 115
  - pipeline, 131
- C55x DSPs, 80
  - and memory organization, 81–82
  - memory map, 82
- Cyber-physical considerations, 454
- Cyber-physical system, 7
- Cycle-accurate simulator, 268

Cyclic redundancy check (CRC), 444–445  
 Cyclomatic complexity, 284  
 Cypress PSoC 6, system organization of, 163–164

## D

### Data

access system, 54–55  
 buffers, 145–146, 279  
 dependencies, 59, 330f  
 frames, 444  
     CAN, 444–445, 445f  
 instructions, 59, 469  
 link layer, 415, 469  
 operations, 61–69, 62f  
 in ARM processor, 62  
 in PIC16F microprocessor, 77  
 pointers, 82  
 pointer registers, auxiliary and coefficient, 81  
 ready signal, 166  
 registers, 98  
 space, 82  
 stall, 132  
 stream style, 222  
 structures, 251  
     queues, 230  
 transmission, 447  
 types, 89  
 video accelerator, 476–478, 477f

### Data compressor

program design  
     non-object-oriented implementation, 151  
     object-oriented (OO) design in C++, 146  
 requirements and algorithm, 142–144, 142f  
 specification, 144–146, 145f–147f  
 testing, 152–153, 153f

### Data-dependent program paths, 263

### Data flow

graphs, 230–231, 231f–232f  
 nodes, 233  
 testing, 286

### DCC. *See* Digital Command Control (DCC)

### DCT. *See* Discrete cosine transform (DCT)

### Dead code, 254

### Deadline, 4, 193, 326–327, 327f

### Debugging, 21, 188, 472

- challenges, 192–193
- techniques, 189–192

### Decision nodes, 233

### Definition-use analysis, 286

### Delayed branch, 133

### Delay slots, 92–93

Demosaicing, 297–298  
 Dense instruction sets, 280  
 Dequeueing, 228–229  
 Derived class, 24–25  
 Design  
     flows, 383  
     methodologies  
         example of, 381–382  
         product metrics, 382  
         process, 381–382  
         review format, 403  
 Design rule for Camera File (DCF) standard, 302–303  
 Desktop processors, power requirements of, 454–455, 455f  
 Detailed resource modelling, 397  
 Detector class, 36–38  
 Device driver, 103  
 Diamond-shaped nodes, 233  
 Digital Command Control (DCC), 32–33  
 Digital filters, 229  
 Digital media processor, 457–458  
 Digital still camera (DSC), 296–306  
     architecture of, 305–306, 305f–307f  
     component design and testing, 306  
     file formats for, 302  
     image compression, 299, 299f  
     imaging algorithms, 297  
     integration and testing, 306  
     operating modes, 303, 304f  
     operation and requirements, 296–301  
     specification, 301–303  
 Digital system, 5  
 Direct-mapped cache, 120  
 Direct memory access (DMA), 172–174  
     controller, 172, 173f  
     request, cyclic scheduling of, 175f  
 Directed acyclic graph (DAG), 329–330  
 Discrete cosine transform (DCT)  
     coefficients, 300  
 Display class, 22–23  
 Distributed system  
     system-on-chip *vs.*, 456–457  
 DLLs. *See* Dynamically linked libraries (DLLs)  
 DMA. *See* Direct memory access (DMA)  
 Documents, DCC, 32  
 Dominant, 444  
 Domain Name Server (DNS), 417  
 Dominant, 444–445  
 Domain-specific modeling language (DSML), 391  
 DOS file systems, 194  
 Double in-line memory modules (DIMMs), 178  
 DRAM. *See* Dynamic RAM (DRAM)  
 DSC. *See* Digital still camera (DSC)

- Dual AR indirect addressing, 85  
 Dual-kernel approach, 366  
 Dynamic power management mechanism, 137–138  
 Dynamic programming, 259–260  
 Dynamic RAM (DRAM), 177  
     synchronous, 178, 179f  
 Dynamically linked libraries (DLLs), 241–242  
 Dynamic voltage and frequency scaling (DVFS), 139
- E**  
 Earliest deadline first (EDF), 345–348  
 ECU. *See* Engine control unit (ECU)  
 Electrical interface to I<sup>2</sup>C bus, 467–468, 468f  
 Electronic control units (ECUs), 440  
 Embedded computing  
     multiprocessor system-on-chip (MPSoC), 456–457  
     systems, 1, 161, 322, 411–412, 454  
         challenges in design, 8–9  
         characteristics of applications, 4–5  
         design process, 381  
         multitrate, 323, 323f  
         performance of, 9–10  
 Embedded multiprocessor, 454  
     accelerator performance analysis, 460–465  
     accelerators, 458–460  
     algorithm and requirements, 474–476  
     architecture, 478–480  
     categories, 456–457  
     component design, 480  
     debugging, 472  
     defined, 453–455  
     heterogeneous shared memory multiprocessors, 457–458  
     MPSoCs, 457–472  
     overview, 453  
     scheduling and allocation, 465–467  
     shared memory multiprocessors, 457–472  
     specification, 476–478  
     system integration, 467–472  
     system testing, 481  
     video compression, 472–474  
 Embedded programs, components for  
     circular buffers and stream-oriented programming, 222–227  
     queues and producer/consumer systems, 228–230  
     state machines, 220–222  
 Embedded system  
     based on computing platform, 161–165  
     design process, 10–29, 11f  
         architecture design, 18–20, 18f–19f  
         behavioral description, 26–29, 27f–29f  
         formalisms, 21–22  
         hardware and software components, 20  
     specification, 17–18  
     structural description, 22–26, 23f–26f  
     system integration, 20–21  
     requirements, 12–17, 14f  
     software layer diagram for, 164–165, 164f  
 Encode, 145  
     behavior, 146  
 Energy consumption, 276–278, 276f  
 Energy optimization, 278–279  
 Engine control unit (ECU)  
     component design and testing, 371  
     specification, 367–368, 368f  
     system  
         architecture, 368–371, 369f–370f  
         integration and testing, 371  
         theory of operation and requirements, 366–367, 367f  
 Engine controller, 325–326, 366–367, 438  
 Enhanced modular I/O subsystem (eMIOS), 370  
 Enqueueing, 228–229  
 Entry point, 240, 241f  
 Environmental sensors, 412  
 Error correction data byte, 33  
 Error delimiter field, 446  
 Error handling, 446  
 Error injection, 289–290  
 Ethernet, 183, 188  
 Evaluation board, 183  
 Exceptions, 117  
 Exchangeable Image File Format (EXIF), 302, 303f  
 Executable binary file, 235–236  
 Execute packet, 92  
 Execution path, 260–262, 265–266  
 Execution time, 256, 258–260  
     accelerator, 461, 462f  
 Extended service set (ESS) network, 427  
 External reference, 240, 241f
- F**  
 Fast Fourier transform (FFT), 187  
 Fast interrupt requests (FIQs), 114  
 Fast return, 89  
 Federated network, 443  
 Fetch packets, 92  
 Field-programmable gate arrays (FPGAs), 5, 460, 476, 478  
     designing of, 480  
 File allocation table (FAT), 194  
 File systems, embedded, 194  
 File Transport Protocol (FTP), 418–419  
 Finite impulse response (FIR) filter, 222  
 Finite-state machine, 220  
 First-level cache, 124

Flash file systems, 194  
 Flash memory, 178–179, 194  
 FlexRay network, 447  
 Floating-point operations, 261  
 Flow of control, 69–75, 248, 248f  
 Flush, 145  
 Foreground program, 208  
 Formatter class, 36, 39  
 Four-cycle handshake, 166, 166f  
 FPGAs. *See* Field-programmable gate arrays (FPGAs)  
 Fragmentation, 126–127  
 Frame, 195  
 Frame pointer (FP), 67–68, 244–245  
 FreeRTOS.org, 335–336, 336f, 359  
 Frequency-shift keying (FSK), 292  
 Full authority digital electronic control (FADEC), 210  
 Full-function device (FFD), 426  
 Functional requirements, 50, 385  
 Functional tests  
     code coverage of, 289f  
     evaluating, 289–290

**G**

Generalization relationship, UML, 25  
 General-purpose computer, 2  
 Generic Attribute Profile Layer (GATT), 425  
 Generic quantitative analysis modelling, 397–398  
 Genesis Platform, 443  
 Global Positioning System (GPS), 4  
 Graph  
     coloring, 255, 256f  
     matrix representation of, 284f  
     theory, 283

**H**

Hacking  
     car and airplane, 448–449  
 Hardware  
     block diagram, 18  
     codesign, 458  
 Hardware abstraction layer (HAL), 164–165  
 Harvard architectures, 77  
 HD video coprocessor subsystem, 457–458  
 HD video processing subsystem (HDVPPS), 457–458  
 Heterogeneous shared memory multiprocessors, 457–458  
 Hierarchical design flows, 384  
 High-level application modelling, 397  
 High-level programming language, 335, 459–460  
 High-performance processors, 5  
 Hit rate, 120

Host, 458  
     system, 188, 188f  
 HTTP, 418–419  
 Huffman coding, 142–144, 301

**I**

Image sensors, 297–298  
 Immediate operands, 63–64  
 Implanted devices, 412  
 Incidence matrix, 283  
 In-circuit emulator (ICE), 190–191  
 Indirect addressing, 67f  
 Induction variable, 270  
 Initiation interval, 327–328  
 Initiation time, 327, 327f  
 Input and output (I/O) devices, 181–183  
 Input and output (I/O) programming  
     busy-wait devices, 101–102  
     devices, 98–100, 98f  
     interrupts  
         ARM, 114–115  
         basics, 103–109, 103f  
         C55x, 115  
         overhead, 113–114  
         PIC16F, 116  
         power-down, 111  
         priorities and vectors, 109–113, 110f  
         subroutines, 109  
         primitives, 100–101  
         prioritized interrupts, 110, 110f  
 Input symbols, 142–144  
 Instruction data byte, 33  
 Instruction execution, 91–92  
 Instruction scheduling, reservation table for, 258, 258f  
 Instruction-level simulator, 268  
 Intellectual property (IP), 187  
 Interconnection network. *See* Networks  
 Interface  
     advanced configuration and power interface (ACPI), 200  
     CPU, 459–460  
     electrical, 467–468  
     I<sup>2</sup>C, 471–472  
     motor, 36  
     PCIe, 476  
     user, 4, 13, 205f, 208  
 Inter-integrated circuit (I<sup>2</sup>C) bus, 467  
 Internal block diagram (ibd), 395  
 Internal consistency of requirements, 15  
 Internet-of-Things (IoT) systems  
     applications, 411–413  
     architectures

- control system, 414, 415f
  - distributed sensors, 414
  - edge and cloud components, 413
  - monitoring system, 414, 414f
  - use case for, 413, 413f
  - databases
    - data representation, 428–429, 429f
    - joins, 430
    - normal forms, 430
    - queries, 430
    - relational databases, 428
    - schemaless databases, 431
  - networks
    - ad hoc networks. *See* Ad hoc networks
    - BLE, 424, 425f
    - bluetooth, 423
    - edge networks, 419
    - gateway, 419, 419f
    - IEEE 802.15.4 standard, 425–427
    - IP, 416–419, 417f–418f
    - LoRa, 428
    - OSI model, 414–416, 416f
    - QoS, 420–421
    - radio energy consumption analysis, 422, 422f
    - routing packets, 420, 421f
    - synchronization, 421, 422f
    - topology, 420, 420f
    - Wi-Fi, 427–428
    - ZigBee, 425–427
    - smart home, 432–434, 433f
      - light activation, 434, 434f
      - resident's activity analysis, 432–433, 433f
      - UML object diagram, 434, 434f
    - timewheels, 431–432, 431f
  - Internet Protocol (IP), 416–419, 417f–418f
  - Internet service stack, 418–419, 418f
  - Internetworking, 417
  - Interprocess communication mechanisms, 356–361
    - mailboxes, 360–361
    - message passing, 358–359, 358f
    - shared memory communication, 357–358, 357f
    - signals, 359–360
  - Interrupt service routine (ISR), 362
  - Interrupts, 332–333
    - acknowledge signal, 103
    - ARM, 114–115
    - basics, 103–109
    - buffers, 105–108
    - C55x, 115
    - debugging code, 108–109
    - handler, 103, 208–209
    - mechanism, 103, 103f
    - overhead, 113–114
    - PIC16F, 116
    - power-down, 111
    - priorities and vectors, 109–113, 110f
    - request, 103
    - vectors, 113f
  - I/O instructions, 100
  - I/O programming. *See* Input and output (I/O) programming
  - ISR. *See* Interrupt service routine (ISR)
- J**
- Jazelle instruction, 76
  - Jet engine controller
    - component design, 212
    - operation and requirements, 210, 210f
    - specifications, 211, 211f
    - system architecture, 211–212, 211f–212f
    - system integration and testing, 212
  - Jitter, 328
  - Joint Photographic Experts Group (JPEG) images, compression process for, 299
  - Jump instruction, 234
- L**
- Latency, 361–362
  - L1 cache, 124
  - L2 cache, 124
  - Leakage, 137
  - Light-emitting diodes (LEDs), 190
  - Linkage mechanism, 244
  - Linker, 235–236
  - Linking process, 240–241
  - Links, 25–26
  - Linux, 362–363
  - Little-endian mode, 56
  - Loader, 235–236
  - Load map file, 240–241
  - Load-store architecture, 62
  - Local Interconnect Network (LIN), 441–442
  - Logical link control (LLC), 415
  - Logic analyzer, 191, 192f, 268
  - Longest path, 463–464
  - Loop fusion, 253–254
  - Loop tiling, 273–274
  - Loop-back testing, 296
  - Loops
    - distribution, 253–254
    - nest, 272
    - optimizations, 272–274
    - testing, 287
    - unrolling, 253

Lossless compression methods, 299  
 Lossy compression algorithm, 299  
 Luminance, 297

**M**

Machine-independent optimizations, 244  
 Macroblocks, 472–473  
 Maintenance, 448  
 Manufacturing cost, 4, 12  
 Masking, 111  
 Measurement-driven performance analysis, 267–270  
 Medium access control (MAC), 415  
 Memory  
     access times, 199  
     aspect ratio, 199, 199f  
     channels and banks in, 180, 180f  
     controllers, 179–180, 180f  
     organization, 179–181  
 Memory mapping, 81, 125  
 Memory-mapped I/O, 100  
 Memory-mapped registers, 81  
 Memory protection unit (MPU), 129–130  
 Memory system mechanisms  
     Cache. *See* Cache  
     MMUs, 118–119  
 Memory system performance, 120  
 Mesh network, 420  
 Message passing, 357, 456  
 Methodological techniques, 289  
 Microcontroller, 56–57, 163, 358–359  
 Microprocessor, 1, 261  
     architectures, 280  
     cyber-physical system, 7  
     embedding computers, 2–3  
     in-circuit emulator, 190–191  
     system bus configurations, 175–176  
 Miss rate, 120  
 Mock-up, 13  
 Model train controller, 29–43, 30f  
     conceptual specification, 33–36  
     DCC, 32–33  
     detailed specification, 36–42  
     requirements, 31  
 Motion-based coding, 472–473  
 Motor interface, 36, 38, 38f  
 Move instruction, 86  
 Moving map, block diagram for, 15–17  
 MPEG-2 encoder, 472, 473f  
 MPSoC. *See* Multiprocessor system-on-chip (MPSoC)  
 Multiple inheritance, UML, 25

Multiple processes, 322  
     timing requirements on, 326–330  
 Multiple tasks, 322–324  
 Multiple timers, 333  
 Multiply-accumulate (MAC) instructions, 64–65  
 Multiply instructions, 87  
 Multiprocessor, 6, 453–454  
 Multiprocessor system-on-chip (MPSoC), 456–457  
     shared memory multiprocessors, 457–472  
 Multirate behavior, 4  
 Multirate communication, 330  
 Multitasking system, 10  
 Multithreaded system, 461, 461f  
 Multirate systems, 324–334

**N**

NEON instructions, 76  
 Nested loop, 271  
 Network, 453–454  
 Networked control systems, 399, 437  
     car and airplanes, 440–443  
     network devices, 440–442  
 Network Information Base (NIB), 426  
 New-symbol-table, 145  
 Nodes, 224–225, 231, 233  
 Nonblocking, 461  
 Nonfunctional properties modelling, 397  
 Nonfunctional requirements, 12–13, 385  
 Nonmaskable interrupt (NMI), 111  
 Nonrecurring engineering (NRE) costs, 13  
 Nonrepeatable instructions, 88  
 NWK layer data entity, 426  
 NWK layer management entity, 426

**O**

Object-oriented design, 21  
     processes and, 339  
 Object-oriented modeling language, 21  
 Object-oriented programming, 22  
 Object-oriented specification, 21–22  
 Objects  
     code, 235–236  
     design, 242  
     file format, 240  
     UML notation, 22  
 One-dimensional array, 251, 251f  
 One-time programmable (OTP) memory, 178–179  
 Open source platforms, 183  
 Open System Interconnection (OSI) models, 415–416, 416f

- Operating system (OS), 187, 321  
 performance evaluating  
     interrupt latency, 361  
     ISH, 363  
     ISR, 363  
     POSIX, 365  
     RTOS, 371  
     process and scheduling states, 331–332
- Optimization techniques, 243
- ORG statements, 237
- Output symbols, 142–144
- Oxygen sensor (OX), 325–326, 368
- P**
- Packets, 33
- Page fault, 125–126
- Page mode access, 199
- Panel class, 38
- Parametric diagram (par), 395–397
- Passengers, 448
- Passers-by, 448
- PC. *See* Program counter (PC)
- PC stack, 78
- Peek function, 101
- Performance measures, 262
- Performance optimization strategies, 274–275
- Periodic processes, 327–328
- Peripheral page pointers, 82
- Personal computers (PCs), 6
- Phase detection, 297
- Physical performance measurement, 268
- PIC16F882 microcontroller, system organization of, 163–164
- PICmicro midrange family  
     data operations, 77–78  
     flow of control, 78–79  
     processor and memory organization, 77
- Piconets, 423
- Picture-taking process, 303  
     sequence diagram for, 305
- Pipeline, 258  
     stalls, 132
- Platform-level performance analysis, 194–199, 195f
- Platforms, 6
- Polled processes, 355
- Poke function, 101
- Polling, 101
- POSIX, 365  
     Linux, 366  
     real-time scheduling in, 365–366  
     threads, 365
- Power consumption, 4, 6–7, 138, 259–260, 454–455, 455f  
     Power-down mode, 137–138  
     Power state machine, 138  
     Power supply voltage, 30  
     Preexisting systems, 390–391  
     Priority inversion, 352–353  
     Priority-based scheduling, 340–356  
         earliest-deadline-first (EDF), 345–349, 348f  
         events and sporadic tasks, 355–356, 356f  
         low power, 353  
         modeling assumptions, 353–355  
         priority inversion, 352–353  
         rate-monotonic scheduling (RMS), 341–345, 349  
         round-robin, 340  
         shared resources, mutexes, and semaphores, 349–352, 350f–351f
- Procedure call stack, 73
- Procedure linkage, 73–74, 244
- Process priorities, 335–336
- Process states, 331–332
- Processing element (PE), 453–454
- Processor interrupt latency, 362
- Producer/consumer systems, 228–230
- Product metrics, 382
- Profiling, 267–268
- Program counter (PC), 54, 79, 233
- Program execution time, 262
- Program generation, compilation, 235–236, 236f
- Program-level power management, 139–140
- Program location counter (PLC), 237
- Programmability of microprocessors, 6
- Programming model, 56
- Program performance, 260  
     analysis of, 262–267
- Program size, analysis and optimization, 279–280
- Program trace, 267
- Program-level energy, 275–279
- Program-level performance analysis  
     energy consumption, 275, 276f  
     energy optimization, 278  
     execution time, 276–278, 277f  
     measurement-driven performance analysis, 267–270  
     memory effects, 276  
     performance measures, types of, 262  
     program performance, analysis of, 262–267
- Programs, 9  
     controlling and observing, 281  
     embedded, 220–230  
     models of, 230–234  
     performance measures on, 262
- Prototypes, 390–391
- Prototyping languages, 390–391
- Pulser class, 38

**Q**

Qualcomm QCA4004 Low-power Wi-Fi, 428  
 Quality assurance techniques, 401–403  
 Quality-of-service (QoS), 420–421  
 Quantization matrix, 300  
 Queues, 228–230, 359

**R**

Radio-frequency identification (RFID), 412  
 Random tests, 288  
 Random values, 288  
 Rate-monotonic analysis (RMA), 341  
 Rate-monotonic scheduling (RMS), 341–345  
 Reachability analysis, 254  
 Real-time code, timing error in, 192  
 Real-time computing, 9–10  
 Real-time operating systems (RTOSs), 322  
     example of, POSIX, 365–366  
     preemptive operating system, 334–339  
         basic concepts, 335–336, 335f  
         object-oriented design, 339, 339f–340f  
         processes and context, 336–339, 336f  
 Real time performance, 6, 195, 417–418, 454  
 Receiver class, 36, 39  
 Recessive, 444  
 Reduced-function devices (RFDs), 426  
 Reduced instruction set computers (RISCs), 55  
 Reentrant, 242  
 Register, 54  
     control interrupts, 82  
     C55x DSPs, 82  
 Register allocation, 254  
 Register indirect addressing, 65–66  
 Register transfer design, 400  
 Regression tests, 288–289  
 Relational database management system (RDBMS), 428  
 Relational databases, 428  
 Relative addresses, 236  
 Relative code, 240  
 Relocatable code, 240  
 Remote frame, 445–446  
 Requirement diagram (req), 395–397  
 Requirements analysis, 385–391  
 Requirements form, 13–14, 14f  
 Reservation table, instruction scheduling, 258  
 RISCs. *See* Reduced instruction set computers (RISCs)  
 RMA. *See* Rate-monotonic analysis (RMA)  
 Root of trust, 140  
 Round nodes, 231  
 Round-robin scheduling, 340

Row major, 251

RTOSs. *See* Real-time operating systems (RTOSs)

**S**

Saved program status register (SPSR), 114  
 Scaffolding code, 153  
 SCCHED\_OTHER, 365–366  
 SCCHED\_RR, 365–366  
 Schedule operations, 465  
 Scheduling overhead, 332, 454, 455f  
 Scheduling policy, 332, 341  
 Scheduling processes, 465–467  
 Scheduling states, 331  
 Schemaless databases, 431  
 SDRAM. *See* Synchronous dynamic RAM (SDRAM)  
 Second-level cache, 124  
 Security, 201–202  
     attestation, 202  
     cryptography, 201  
         digital signature, 201–202  
         hash function, 201  
         public-key cryptography, 201  
         secret key cryptography, 201  
 Semaphores, 349–352  
 Send-command method, 39  
 Sequence diagram, 167, 305, 307f, 335f–336f, 362, 476–478, 477f  
 Serial clock line (SCL), 467  
 Serial data line (SDL), 467  
 Set-associative cache, 121–122, 122f  
 Shared memory multiprocessors, 457–472  
     accelerator performance analysis, 460–465  
     accelerators, 458–460  
     debugging, 472  
     heterogeneous shared memory multiprocessors, 457–458  
     scheduling and allocation, 465–467  
     system integration, 467–472  
 Shared memory systems, 456  
 Sharpening algorithms, 299  
 Signal flow graph, 224–225, 224f  
 Signal processing algorithms, 289  
 Signal, in UML, 27  
 Simple mail transfer protocol, 418–419  
 Simple network management protocol, 418–419  
 Single in-line memory modules (SIMMs), 178  
 Single-assignment form, 230–231, 231f  
 Single-chip  
     platform, 163  
     CPUs, 49  
 Single-instruction multiple-data (SIMD), 76  
 Single-issue processor, 56

- Single-repeat registers, 82  
 Single-threaded system, 461, 461f  
 Slow return, 89  
 Smart cards, 141  
 Smartphones as platforms, 15–17  
 Society of Automotive Engineers (SAE), 371  
 Software  
     block diagram, 18  
     codesign, 458  
     designing components, 20  
     interrupt, 109–110  
     pipelining, 258  
     physics of, 6–7  
     scaffolding, 267  
 Software engineers, 382  
 Software modem  
     architecture of, 294–295, 295f  
     component design and testing, 295–296  
     design of, 292–296  
     integration and testing, 296  
     operation and requirements, 292–294, 292f–293f  
     specification, 294f  
 Software performance optimization  
     cache optimizations, 272–274  
     loop optimizations, 270–272, 271f  
     strategies, 274–275  
 Software testing, techniques, 401–402  
 Source code, 230, 236–237  
 SP. *See* Stack pointer (SP)  
 Space shuttle software error, 329  
 Special function registers, 77–78  
 Specification languages, 390–391  
 Stack pointer (SP), 63, 81, 244–245. *See also* Frame pointer (FP)  
 Standard data flow graph, 232f  
 Star network, 420  
 State machines, 26–27, 220–222  
 State mode, 191  
 State chart, 392–394  
 Static power management, 137–138  
 Static scheduling policy, 341  
 Status register, 81, 98  
 Stereotypes, in UML, 26  
 Stream-oriented programming, 222–227  
 Strength reduction, 270  
 Structural description, 22–26  
 Subroutines, 88, 109, 279–280, 324  
 Superscalar processor, 56  
 Supervisor mode, 116–117  
 Symbol table, 145, 237  
 Synchronous and asynchronous communication, 420–421  
 Synchronous connection-oriented (SCO) packets, 423  
 Synchronous dynamic RAM (SDRAM), 178, 179f  
 System design techniques  
     analysis and architecture design, 398–400  
     design patterns, 398–400  
     transaction-level modelling, 400  
 Dependability, safety, and security, 400–408  
 Design reviews, 403–404  
 Quality assurance techniques, 401–403  
 Safety-oriented methodologies, 404–408  
 Security, 408  
 Design flows, 383  
 Design methodologies, 381–384  
     embedded computing, 383–384, 384f  
 Modeling, 391–398  
     model-based design, 391–394  
     UML dialects, 394–398, 395f–399f  
 Requirements analysis, 385–391  
     capture, 385–388, 386f  
     specifications, 389  
         characteristics, 389  
         validation, 389–391, 390f  
 System integration, 20–21, 467–472  
     DSC, 306  
     testing and, 210, 212, 296, 306, 371  
 System-on-chip, 456–457  
 System requirements  
     *vs.* specifications, 12  
     validating, 13  
 System speedup, 463, 464f, 465  
 System testing, DSC, 306  
 Systems modeling language (SysML), 395

## T

- Table attribute, 145  
 Tagged Image File Format (TIFF), 302  
 Target system, 188, 188f  
 Task, 21  
     graph, 322  
     set, 330  
 Temporary registers, 82  
 TCAS. *See* Traffic alert and collision avoidance system (TCAS)  
 Template matching, code generation by, 259–260, 259f  
 Testbench program, 188  
 Test generation programs, 259f, 402  
 Therac-25 medical imaging system, 404–406  
 Threads, 323  
 Threat models, 448  
 Throttle resolver angle (TRA), 211  
 Throttle settings, 325–326  
 Thumbnail, 302  
 TI C64x, 91–93  
 Time modelling, 397

Time-out event, 27  
 Time quantum, 335  
 Timer, 181, 261, 333  
 Timesharing system, 242  
 Time-triggered architecture, 446–447  
 Timing mode, 191  
 TMR0 overflow interrupt enable bit, 116  
 Top down design, 11–12  
 Traffic alert and collision avoidance system (TCAS), 392–394  
 Train controller commands, 34f  
     refining, 42–43, 44f  
 Transition registers, 82  
 Translation lookaside buffer (TLB), 129  
 Transmission control protocol (TCP), 418  
 Transmitter class, 36  
 Traps, 117–118  
 Tree network, 420  
 TrustZone, 76, 140–141  
 Two-dimensional arrays, 251, 252f  
 Type certification, 439

**U**

Unified cache, 124  
 Unified Modeling Language (UML), 339, 339f, 479–480  
     active objects, 339  
     collaboration diagram, 34–35, 34f  
     multiple inheritance in, 25, 25f  
     object in, 22, 23f  
     sequence diagram, 29  
     sequence, of DMA transfer, 174f  
     state diagram, of bus bridge operation, 176f  
     state and transition in, 27f  
 Universal Asynchronous Receiver/Transmitter (UART), 98–100, 357–358  
 Universally unique identifiers (UUIDs), 424  
 Universal Serial Bus (USB), 163  
     port, 183  
 Unrolled schedules, 341–343  
 Usage scenarios, 387, 391  
 User Datagram Protocol (UDP), 418–419  
 User interface, 4, 13, 188, 192, 204, 205f, 208  
 User mode, 116

**V**

Value nodes, 231  
 Variable data rates, 323–324  
 Vectors interrupt, 242  
 Vehicles  
     cyber-physical systems, 437–439  
     avionics, 439  
     car subsystems, 437–438, 438f  
     interactions, 438–439  
     driver assistance and autonomy, 439–440  
 Vehicular networks, 443–447  
     architectures, 442–443, 442f  
     CAN bus, 443–446, 444f–446f  
 Video accelerator  
     algorithm and requirements, 474–476, 475f  
     architecture, 478–480, 478f–480f  
     component design, 480  
     compression, 472–474, 473f–474f  
     specification for, 476–478, 477f  
     system testing, 481  
 Virtual addressing, 125  
 Virtual memory, 129  
 Very-long instruction word (VLIW) processor, 56  
     embedded computing, 60  
     packet, 59  
     processor, 58–60  
     superscalar, 58–59  
 V model, 384, 384f  
 Von Neumann machine, 54

**W**

Wait states, 170, 170f, 196–197  
 Wall-mountable camera, 412  
 Waterfall model, 384  
 Wearable devices, 412  
 While loop, 101–102, 208, 233, 235f  
 Word length, 55–56  
 Working set, 119  
 Worst-case execution time, 262  
     analysis, 267  
 Write-back policy, 120–121  
 Write-through scheme, 120–121