High-Performance Embedded Systems: Architectures, Algorithms, and Applications

Wayne Wolf
Princeton University

June 13, 2005

Description
High-Performance Embedded Systems focuses on the unique complexities of embedded system design. Designed as a textbook for advanced undergraduates and graduate students in CS, CE, and EE advanced embedded computing courses, it covers advanced topics in embedded computing, including multiprocessors, a comprehensive view of processors (VLIW and superscalar architectures), and power consumption. It provides advanced treatment of all the components of the system as well as in-depth coverage of networks, reconfigurable systems, hardware-software co-design, security, and program analysis. A discussion of current industry development software includes Linux and Windows CE. Examples feature the Freescale DSP with the TI C5000 and C6000 series. Real-world applications will include DVD players and cell phones.

Audience
Advanced undergraduates and graduate students in CS, CE, and EE advanced embedded computing courses

Pre-requisites
- C programming.
- Basic background in embedded computing:
  - Instruction sets and undergraduate-level computer architecture
  - I/O programming
  - Basic concepts in real-time scheduling

Outline
The outline below describes both the actual structure/content of completed first draft chapters and the proposed structure/content of chapters still in development. For unfinished chapters, keep in mind that the structure is approximate and the content subject to change.

(Reviewers of early, incomplete drafts of 5 and 6 should note that these chapters will undergo significant revision based on their previous feedback. For the purposes of this outline, I am showing them as ‘in development’.)

Changes from earlier versions of the outline are indicated in RED.

Chapter 1: Embedded Computing (1st draft completed)

Covers:
- Fundamental problems in embedded computing
- Design methodologies for embedded systems

- Models of computation
- Reliability and security
- Applications that make use of embedded computing

1.1 The Landscape of High-Performance Embedded Computing 1
1.2 Design Methodologies 4
    1.2.1 Basic Design Methodologies 5
    1.2.2 Embedded Systems Design Flows 7
    1.2.3 Standards-Based Design Methodologies 9
    1.2.4 A Methodology of Methodologies 12
    1.2.5 Joint Algorithm and Architecture Development 12
1.3 Models of Computation 13
    1.3.1 Why Study Models of Computation? 13
    1.3.2 Finite vs. Infinite State 14
    1.3.3 Parallelism and Communication 21
    1.3.4 Control Flow and Data Flow Models 25
1.4 Reliability, Safety, and Security
    1.4.1 Why Reliable Embedded Systems? 30
    1.4.2 Fundamentals of Reliable System Design 31
    1.4.3 Novel Attacks and Countermeasures 34
1.5 Example Applications 36
    1.5.1 Radio and Networking 38
    1.5.2 Multimedia 40
1.6 Summary
What We Learned 44
Further Reading 44
Questions 44
Lab Exercises 45

Chapter 2: CPUs (1st draft completed)

Covers:
- Architectural mechanisms for embedded processors
- Parallelism in embedded CPUs
- Code compression and bus encoding
- Security mechanisms
- CPU simulation

2.1 Introduction
2.2 Comparing Processors
    2.2.1 Evaluating Processors
    2.2.2 A Taxonomy of Processors
2.3 RISC Processors and Digital Signal Processors
    2.3.1 RISC Processors
    2.3.2 Digital Signal Processors
2.4 Parallel Execution Mechanisms
    2.4.1 Very Long Instruction Word Processors
    2.4.2 Superscalar Processors
    2.4.3 SIMD and Vector Processors
    2.4.4 Thread-Level Parallelism

2.4.5 Processor Resource Utilization
2.5 Variable-Performance CPU Architectures
2.5.1 Dynamic Voltage and Frequency Scaling
2.5.2 Better-Than-Worst-Case Design
2.6 Processor Memory Hierarachy
2.6.1 Memory Component Models
2.6.2 Register Files
2.6.3 Caches
2.6.4 Scratch Pad Memories
2.7 Additional CPU Mechanisms
2.7.1 Code Compression
2.7.2 Low-Power Bus Encoding
2.7.3 Security
2.8 CPU Simulation
2.8.1 Trace-Based Analysis
2.8.2 Direct Execution
2.8.2 Microarchitecture-Modeling Simulators
2.9 Automated CPU Design
2.9.1 Configurable Processors
2.9.2 Instruction Set Synthesis
2.10 Summary
What We Learned
Further Reading
Questions
Lab Exercises

Chapter 3: Programs (in development)

Proposed coverage:

- Program performance and power analysis
- Emerging programming models
- Just-in-time compilation.

3.1 Program performance evaluation: direct measurement on hardware, simulation, worst-case execution time (WCET) analysis

Direct measurement techniques
Architecture simulators, how to gather traces, how to use the simulator
How to compute WCET

Example: motion estimation for video compression
Example: Performance estimation of total MPEG-2 application

3.2 Just-in-time (JIT) code: compilation methods

3.3 Dynamically allocated data structures: usage in embedded systems; performance analysis

3.4 Program specification and synthesis *(newly added)*

Synchronous dataflow graph: specification, synthesis
Synchronous languages: specification, analysis, synthesis

3.5 Models of computation: composition of heterogeneous models *(newly added)*

3.6 Model-based program synthesis

What is a model
Model analysis
Program synthesis from a model

3.7 Program testing and verification:

Fault models for software
Black box vs. white box
Testing for real-time properties *(newly added)*

Chapter 4: Processes and Operating Systems (in development)

Proposed coverage:

- The range of scheduling mechanisms (more deeply than is typical for an undergraduate embedded course)
- Interprocess communication (in more detail)
- Structure of real-world operating systems
- Multiprogramming performance analysis

4.1 The role of processes in embedded systems *(newly added)*

4.2 Review of interprocess communication

Semaphores

Mailboxes
Other communication mechanisms
4.3 Interprocess communication problems: deadlock, critical races

Example: WinCE process model; WinCE interrupt handling
Example: Linux process model; Linux interrupt handling

4.4 Taxonomy of real-time scheduling algorithms:
Fixed vs. dynamic order
Static vs. dynamic priority
Examples: TDMA, RMS, EDF
Problems with real-time scheduling:
Priority inversion: causes, cures
Statistical scheduling models

4.5 Performance analysis: abstract models of caches for multi-process performance analysis on the CPU.

4.6 Program development: problems with reference implementations, methodologies for adapting reference implementations to embedded platforms.

Chapter 5: Hardware/Software Co-design (revised draft in development)

Proposed coverage:
- Heterogeneous architectures
- Design mechanisms for HW/SW partitioning (including performance analysis)
- FPGAs as targets for co-design

5.1 Basic co-design concepts *(newly added)*
Cost/performance enhancements via co-design
Hardware accelerators

5.2 Hardware performance analysis--high-level synthesis

5.3 Design space exploration

5.4 Platform FPGAs as targets

Example: Motion estimation for video compression.

Chapter 6: Multiprocessor Architectures (revised draft in development)

Proposed coverage:

- Generalize co-design concepts to arbitrary architectures
- Processors, memory, interconnect from a heterogeneous systems point of view.

6.1 What is a multiprocessor?

6.2 Why heterogeneous multiprocessors

Example: TI OMAP
Example: ST Nexperia

6.3 Processing element characteristics and selection---how do you choose a processor.

6.4 Interconnection networks: bus, crossbar, mesh, application-specific

6.5 Interconnection network performance model

6.6 Memory systems:

- Why partitioned memory systems
- Role of caches
- Why non-uniform memory spaces
- Memory system performance model

Example: Philips Nexperia and HDTV
Example: ARM multiprocessor

Chapter 7: Multiprocessor Software (1st draft completed)

Covers:

- Performance analysis of embedded software running on multiprocessors
- Software stacks and middleware
- Design techniques for multiprocessor software

7.1 Introduction 157
7.2 What is Different About Multiprocessor Software?
7.3 Real-Time Multiprocessor Operating Systems 159
- 7.3.1 Role of the Operating System 159
- 7.3.2 Multiprocessor Scheduling 162
- 7.3.3 Scheduling with Dynamic Tasks 176
- 7.3.4 System Modes and Scheduling 177
- 7.3.5 Quality-of-Service 178

7.4 Services and Middleware for Embedded Multiprocessors 178
7.5 Design Methods 181
- 7.5.1 Verification and Validation 182

7.5.2 Performance Analysis 182
7.6 Consumer Electronics Architectures 184
7.6.1 File Systems in Embedded Devices 184
7.6.2 High-Level Services 187
7.7 Summary 189
What We Learned 189
Further Reading 190
Questions 190
Lab Exercises 190

Chapter 8: Networks (1st draft completed)

Covers:

- General network architectures and the ISO network layers
- Automotive and aircraft networks
- Consumer electronic networks
- Sensor networks

8.1 Introduction 191
8.2 Networking Principles 192
8.2.1 Network Abstractions 192
8.2.2 Internet 194
8.3 Networks for Real-Time Control 196
8.3.1 Real-Time Vehicle Control 197
8.3.2 The CAN Bus 198
8.3.3 FlexRay 201
8.3.4 Aircraft Networks 210
8.4 Consumer Networks 212
8.4.1 Bluetooth 212
8.4.2 WiFi 214
8.4.3 Networked Consumer Devices 214
8.5 Sensor Networks 216
8.6 Summary 219
What We Learned 219
Further Reading 220
Questions 220
Lab Exercises 220

Chapter

1

Embedded Computing

- Fundamental problems in embedded computing.
- Design methodologies for embedded systems.
- Models of computation.
- Reliability and security.
- Applications that make use of embedded computing.

1.1 The Landscape of High-Performance Embedded Computing

The overarching theme of this book is that high-end embedded computing systems are measurably hard to design. Not only do they require lots of computation, but they must meet quantifiable goals: real-time performance, not just average performance; power/energy consumption; and cost. The fact that we have quantifiable goals makes the design of embedded computing systems a very different experience than the design of general-purpose computing systems, in which we cannot predict the uses to which the computer will be put.

When we try to design computer systems to meet these sorts of quantifiable goals, we quickly come to the conclusion that no one system is best for all applications. Different requirements lead us to different trade-offs between performance and power, hardware and software, etc. We must create different implementations to meet the needs of a family of applications. That solutions should be programmable enough to make the design flexible and long-lived, but not provide unnecessary flexibility that would detract from meeting the system requirements.

General-purpose computing systems separate the design of hardware and software, but in embedded computing systems we can simultaneously design the hardware and software. We often find that we can solve a problem by hardware means, software means, or a combination of the two. These solutions may have different trade-offs; the

larger design space afforded by joint hardware/software design allows us to find better solutions to design problems.

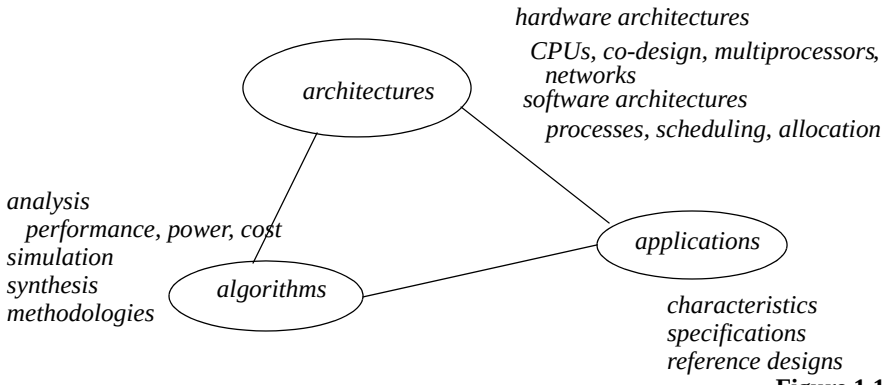


Figure 1-1 *Aspects of embedded system design.*

architectures, algorithms, applications

As illustrated in Figure 1-1, the study of embedded system design properly takes into account three aspects of the field: architectures, algorithms, and applications. Let's consider these aspects one at a time.

architectures

Because embedded system designers work with both hardware and software, they must study architectures broadly, including hardware, software, and the relationships between the two. Hardware architecture problems may range from special-purpose hardware units as created by hardware/software co-design, microarchitectures for processors, multiprocessors, or networks of distributed processors. Software architectures determine how we can take advantage of parallelism and non-determinism to improve performance and lower cost.

algorithms

The algorithmic component of embedded system design includes a wide range of tools that help us build systems. Analysis and simulation tools are widely used to evaluate cost, performance, and power consumption. Synthesis tools create optimized implementations based on specifications.

Methodologies play an especially important role in embedded computing. Not only must we design many different types of embedded systems, but we want to be able to do so reliably and predictably. The cost of the design process itself is often a significant component of the total system cost. Methodologies, which may combine tools and manual steps, codify our knowledge on how to design systems. Methodologies help us make large and small design decisions.

applications

Understanding your application is key to getting the most out of an embedded computing system. We can use the characteristics of the application to optimize our design. This can be an advantage that lets us perform many powerful optimizations that would not be possible in a general-purpose system. But it also means that we must understand the application enough to be able to take advantage of its characteristics and avoid creating problems for system implementers.

embedded computing is multidisciplinary

Embedded computing makes use of several related disciplines. Two core disciplines are real-time computing and hardware/software co-design. The study of real-time systems predates the emergence of embedded computing as a discipline. Real-time systems takes a software-oriented view of how to design computers that complete computations in a timely fashion. The scheduling techniques developed by the real-time systems community stand at the core of the body of techniques used to design embedded systems. Hardware/software co-design emerged as a field at the dawn of the modern era of embedded computing. Co-design takes a holistic view of the hardware and software used to perform deadline-oriented computations.

Embedded computing also takes advantage of many basic disciplines in computer engineering and computer science:

- Low power design started off as primarily hardware-oriented but now encompasses both software and hardware techniques.
- Programming languages and compilers have brought embedded system designers tools such as Java and highly-optimized code generators.
- Operating systems provide not only schedulers but also file systems and other facilities that are now commonplace in high-performance embedded systems.
- Networks are used to create distributed real-time control systems for vehicles and many other applications, as well as to create Internet-enabled appliances.
- Security and reliability are an increasingly important aspect of embedded system design. VLSI components are becoming less reliable at extremely fine geometries while reliability requirements become more stringent. Security threats once restricted to general-purpose systems now loom over embedded systems as well.

this chapter

In the remainder of this chapter, we will cover several topics that will serve as recurring themes throughout the book. First, we will look at design methodologies. We will see how traditional hardware or software methodologies have merged and evolved to serve the needs of embedded system designers. Next, we will consider models of computation, which serve as guides for programming and design analysis. We will then look at two closely related topics—reliability and security—as they apply to embedded systems. We will then review the basics of some important applications of embedded computing so that we can refer to those algorithms and processes in design examples.

remainder of this book

The rest of this book will proceed roughly bottom-up from simpler components to complex systems. Chapters 2 through 4 concentrate on single processor systems:

- Chapter 2 will cover CPUs, including the range of microarchitectures available to embedded system designers, processor performance, and power consumption.
- Chapter 3 will look at programs, including languages and design and how to compile efficient executable versions of programs.
- Chapter 4 will study real-time scheduling and operating systems.

Chapters 5 through 8 concentrate on problems specific to multiprocessors:

- Chapter 5 describes methods for hardware/software co-design, which designs accelerators to complement CPUs.
- In Chapter 6 we will introduce a taxonomy of multiprocessor hardware architectures and what sorts of multiprocessor structures are useful in optimizing embedded system designs.
- In Chapter 7 we will look at software for multiprocessors.
- Chapter 8 moves from closely-coupled multiprocessors to networks; we will study both hardware and software aspects of networked embedded systems.

1.2 Design Methodologies

A design methodology is not simply an abstraction—it must be defined in terms of available tools and resources. The designers of high-performance embedded systems face many challenges. Some of those challenges include:

- The design space is large and irregular. We do not have adequate synthesis tools for many important steps in the design process. As a result, designers must rely on analysis and simulation for many design phases.

- We can't afford to simulate everything in extreme detail. Not only do simulations take time, but the cost of the server farm required to run large simulations is a significant element of overall design cost. In particular, we can't perform a cycle-accurate simulation of the entire design for the large data sets that are required to validate large applications.

- We need to be able to develop simulators quickly. Simulators must reflect the structure of application-specific designs. System architects need tools to help them construct application-specific simulators.

- Software developers for systems-on-chips need to be able to write and evaluate software before the hardware is done. They need to be able to evaluate not just functionary but performance and power as well.

System designers need tools to help them quickly and reliably build heterogeneous architectures. They need tools to help them integrate several different types of processors. They also need tools to help them build multiprocessors from networks, memories, and processing elements.

1.2.1 Basic Design Methodologies

Much of the early writings on design methodologies for computer systems cover software, but the methodologies for hardware tend to be more concrete since hardware design makes more use of synthesis and simulation tools. An ideal embedded systems methodology makes use of the best of both hardware and software traditions.

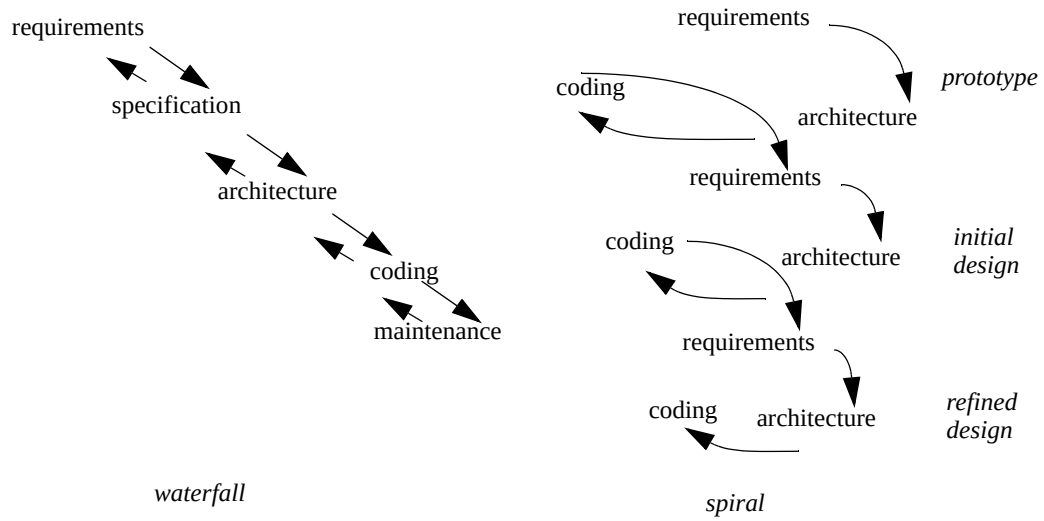


Figure 1-2

Two early models of software development.

waterfall and spiral

One of the earliest models for software development was the **waterfall model** illustrated in Figure 1-2. The waterfall model is divided into five major stages: requirements, specification, architecture, coding, and maintenance. The software is successively refined through these stages, with maintenance including software delivery and follow-on updates and fixes. Most of the information in this methodology flows from the top down—that is, from more abstract stages to more concrete stages—although some information could flow back from one stage to the preceding stage to improve the design. The general flow of design information down the levels of abstraction gives the waterfall model its name. The waterfall model was important for codifying the basic steps of software development, but researchers soon realized that the limited flow of information from detailed design back to improve the more abstract phases was both an unrealistic picture of software design practices and an undesirable feature of an ideal methodology. In practice, designers can and should use experience from design steps to go back, rethink earlier decisions, and redo some work.

The **spiral model**, also shown in Figure 1-2, was a reaction to and a refinement of the waterfall model. This model envisions software design as an iterative process in which several versions of the system, each better than the last, is created. At each phase, designers go through a requirements/architecture/coding cycle. The results of one cycle are used to guide the decisions in the next round of development. Experience from one stage should both help give a better design at the next stage and allow the design team to create that improved design more quickly.

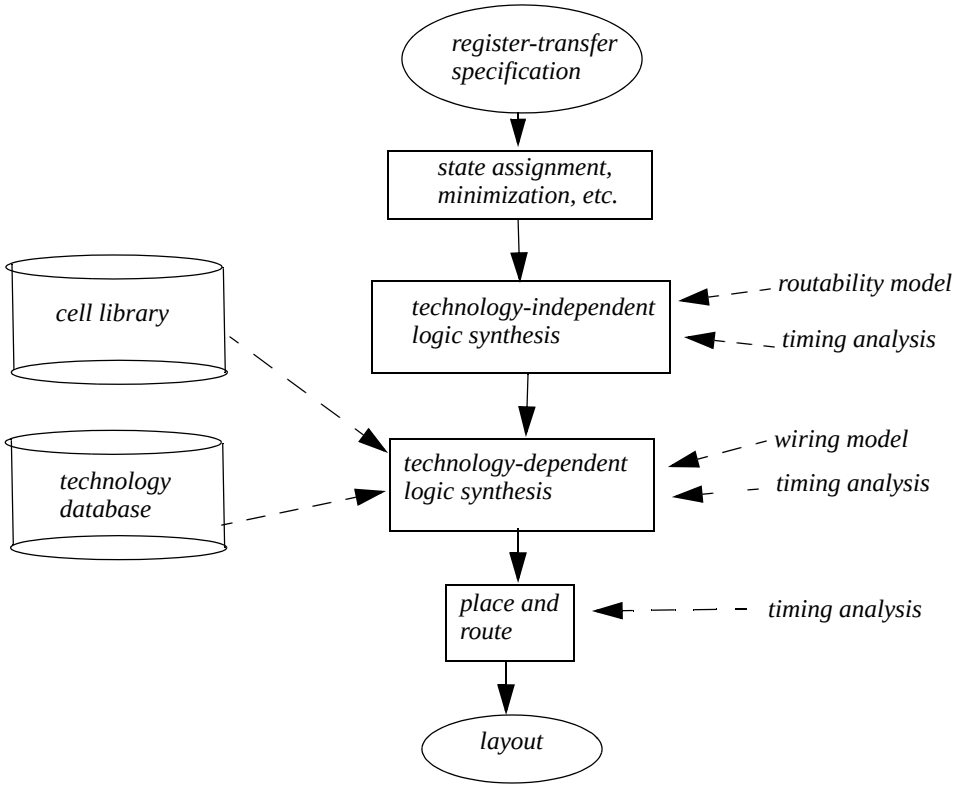


Figure 1-3

A digital synthesis design flow.

hardware design methodologies

Figure 1-3 shows a simplified version of the hardware design flows used in many VLSI designs. Modern hardware design makes extensive use of several techniques not as frequently seen in software design: search-based synthesis algorithms; and models and estimation algorithms. Hardware designs also have more quantifiable design metrics than traditional software designs. Hardware designs must meet strict cycle time requirements, power budgets, and area budgets. Although we have not shown backward design

flow from lower to higher levels of abstraction, most design flows allow such iterative design.

Modern hardware synthesis uses many types of models. In Figure 1-3, the cell library describes the cells used for logic gates and registers, both concretely in terms of layout primitives and more abstractly in terms of delay, area, etc. The technology database captures data not directly associated with cells, such as wire characteristics. These databases generally carry static data in the form of tables. Algorithms are also used to evaluate models. For example, several types of wirability models are used to estimate the properties of the wiring in the layout before that wiring is complete. Timing and power models evaluate the performance and power consumption of designs before all the details of the design are known; for example, although both timing and power depend on the exact wiring, wire length estimates can be used to help estimate timing and power before the delay is complete. Good estimators help keep design iterations local. The tools may search the design space to find a good design, but within a given level of abstraction and based up on models at that level. Good models combined with effective heuristic search can minimize the need for backtracking and throwing out design results.

1.2.2 Embedded Systems Design Flows

Embedded computing systems combine hardware and software components that must work closely together. Embedded system designers have evolved design methodologies that play into our ability to embody part of the functionality of the system in software.

co-design flows

Early researchers in hardware/software co-design emphasized the importance of concurrent design. Once the system architecture has been defined, the hardware and software components can be designed relatively separately. The goal of co-design is to make appropriate architectural decisions that allow later implementation phases to be carried out separately. Good architectural decisions, because they must satisfy hard metrics like real-time performance and power consumption, require appropriate analysis methods.

Figure 1-4 shows a generic co-design methodology. Given an executable specification, most methodologies perform some initial analysis to determine parallelism opportunities and perhaps break the specification into processes. Hardware/software partitioning chooses an architecture in which some operations are performed directly by hardware and others are performed by software running on programmable platforms. Hardware/software partitioning produces module designs that can be implemented separately. Those modules are then combined, tested for performance or power consumption, and debugged to create the final system.

platform-based design

Platform-based design is a common approach to using systems-on-chips. Platforms allow several customers to customize the same basic platform into different products. Platforms are particularly useful in standards-based markets, where some basic features must be supported but other features must be customized to differentiate products.

two-stage process

As shown in Figure 1-5, platform-based design is a two-stage process. First, the platform must be designed, based upon the overall system requirements (the standard, for example) and how the platform should be customizable. Once the platform has been designed,

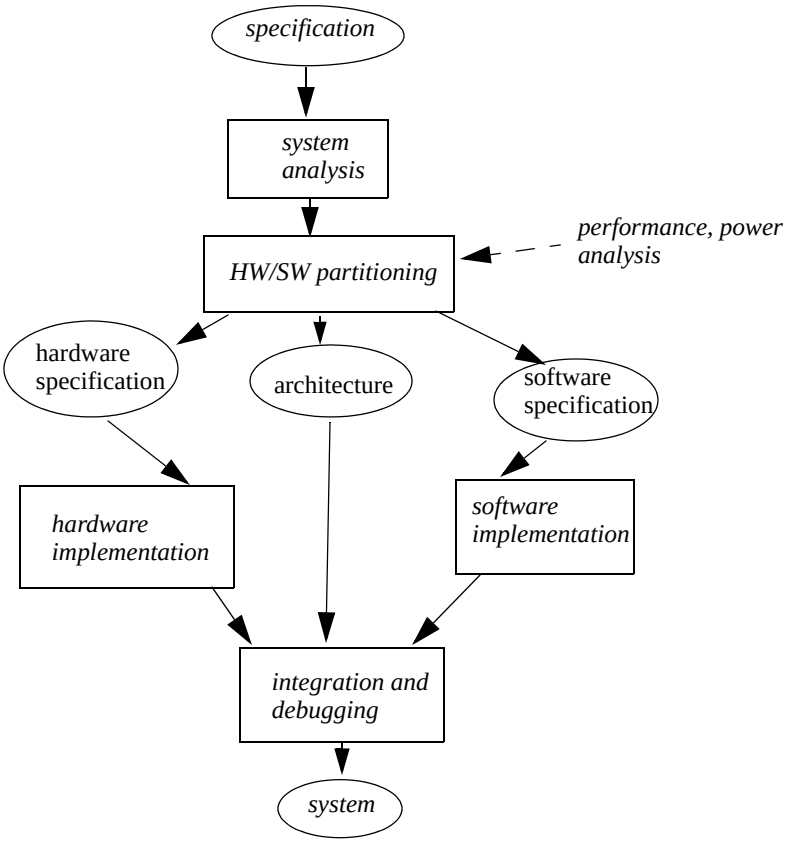


Figure 1-4

A design flow for hardware/software co-design.

it can be used to design a product. The product makes use of the platform features and adds its own features.

platform design phases

Platform design requires several design phases:

- Profiling and analysis turn system requirements and software models into more specific requirements on the platform hardware architecture.
- Design space exploration evaluates hardware alternatives.
- Architectural simulation helps evaluate and optimize the details of the architecture.

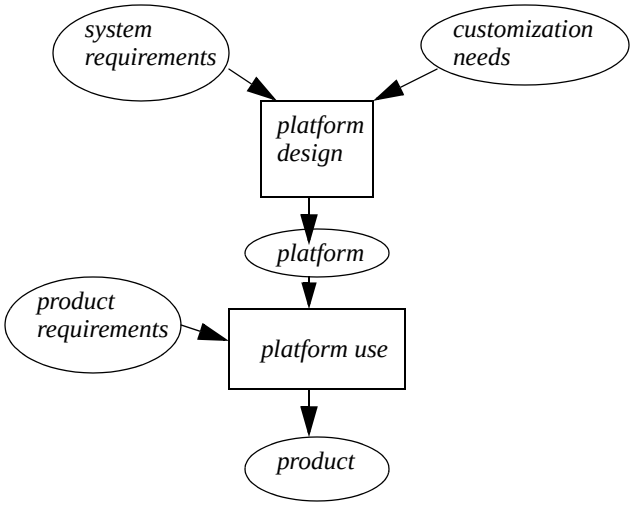


Figure 1-5

Platform-based design.

- Base software—hardware abstraction layers, operating system ports, communication, application libraries, debugging—must be developed for the platform.

programming platforms

Platform use is challenging in part because the platform requires a custom programming environment. Programmers are used to rich development environments for standard platforms. Those environments provide a number of tools—compilers, editors, debuggers, simulators—in a single graphical user interface. However, rich programming environments are typically available for uniprocessors. Multiprocessors are harder to program and heterogeneous multiprocessors are harder than homogeneous multiprocessors. The platform developers must provide tools that allow software developers to use the platform. Some of these tools come from the component CPUs but other tools must be developed from scratch. Debugging is particularly important and difficult, since debugging access is hardware-dependent. Interprocess communication is also challenging but is a critical tool for application developers.

1.2.3 Standards-Based Design Methodologies

Many high-performance embedded computing systems implement standards. Multimedia, communications, and networking all provide standards for various capabilities. One product may even implement several different standards. In this section we will consider the effects that the standards have on embedded system design methodologies [Wol04].

pros and cons of standards

On the one hand, standards enable products and in particular systems-on-chips. Standards create large markets for particular types of functions: they allow devices to interoperate and they reassure customers that the device provides the required functions. Large markets help justify any system design project, but they are particularly important in system-on-chip design. In order to cover the costs of SoC design and manufacturing, several million of the chips must be sold in many cases. Such large markets are generally created by standards.

On the other hand, the fact that the standard exists means that the chip designers have much less control over the specification of what they need to design. Standards define complex behavior that must be adhered to. As a result, some features of the architecture will be dictated by the standard.

Most standards do provide for improvements. Many standards define that certain operations must be performed, but they do not specify how they may be performed. The implementer can choose a method based upon performance, power, cost, quality, or ease of implementation. For example, video compression standards define basic parameters of motion estimation but not what motion estimation algorithm should be performed.

The intellectual property and effort required to implement a standard goes into different parts of the system than would be the case for a blank-sheet design. Algorithm design effort goes into unspecified parts of the standard and parts of the system that lie beyond the standard. For example, cell phones must adhere to communication standards but are free to design many aspects of their user interfaces.

Standards are often complex, and standards in a given field often become more complex over time. As a field evolves, practitioners learn more about how to do a better job and tend to build that knowledge into the standard. While these improvements may lead to higher quality systems, they also make the system implementation larger.

reference implementations

Standards bodies typically provide a **reference implementation**. This is an executable program that conforms to the standard. It is often written in C, but may be written in Java or some other language. The reference implementation is first used to aid the standard developers. It is then distributed to implementers of the specification. (The reference implementation may be available free of charge, but in many cases an implementer must pay a license fee to the standards body to build a system that conforms to the specification. The license fee goes primarily to patent holders whose inventions are used within the standard.) There may be several reference implementations if multiple groups experiment with the standard and release their results.

The reference implementation is something of a mixed blessing for system designers. On the one hand, the reference implementation saves the system designers a great deal of time. On the other hand, it comes with some liabilities. Of course, learning someone else's code is always time-consuming. Furthermore, the code generally cannot be used as-is. Reference implementations are typically written to run on a large workstation with infinite memory; it is generally not designed to operate in real time. The code must often be restructured in many ways: eliminating features that will not be implemented; replacing heap allocation with custom memory management; improving cache utilization; function inlining; and many other tasks.

design tasks

The implementer of a standard must perform several design tasks:

- The unspecified parts of the implementation must be designed.
- Parts of the system that are not specified by the standard (user interface, for example), must be designed.
- An initial round of platform-independent optimization must be used to improve the chosen reference implementation.
- The reference implementation and other code must be profiled and analyzed.
- The hardware platform must be designed based upon initial characterization.
- The system software must be further optimized to better match the platform.
- The platform itself must be further optimized based upon additional profiling.
- The platform and software must be verified for conformance to the standard as well as non-functional parameters such as performance and energy consumption.

The next example introduces the Advanced Video Coding standard.

Application Example 1-1

AVC/H.264

The latest generation of video compression standards is known by several names. It is officially part 10 of the MPEG-4 standard, known as Advanced Video Coding (AVC). However, the MPEG group joined forces with the H.26x group, so it is also known as H.264.

The MPEG family of standards is primarily oriented toward broadcast, in which the transmitter is more complex in favor of cheaper receivers. The H.26x family of standards, in contrast, has traditionally targeted videoconferencing, in which systems must both transmit and receive, giving little incentive to trade transmitter complexity for receiver complexity.

The H.264 standard provides many features that give improved picture quality and compression ratio. H.264 codecs typically generate encoded streams that are half the size of MPEG-2 encodings. For example, the H.264 standard allows multiple reference frames, so that motion estimation can use pixels from several frames to handle occlusion. This is an example of a feature that improves quality at the cost of increased receiver complexity.

The reference implementation for H.264 is over 700,000 lines of C code. This reference implementation uses only fairly simple algorithm for some unspecified parts of the

standard, such as motion estimation. However, it implements both video coding and decoding and it does so for the full range of display sizes supported by the standard, ranging from QCIF to HDTV.

1.2.4 A Methodology of Methodologies

The design of high-performance embedded systems is not described well by simple methodologies. Given that these systems implement specifications that are millions of lines long, it should not be surprising that we have to use many different types of design processes to build complex embedded systems.

Methodologies that we use in embedded system design include:

- **software performance analysis** Executable specifications must be analyzed to determine how much computing horsepower is needed and what types of operations need to be performed.
- **architectural optimization** Cycle-accurate simulation and other architectural methods can be used to optimize systems.
- **hardware/software co-design** Co-design helps us create efficient heterogeneous architectures.
- **network design** Whether in distributed embedded systems or systems-on-chips, networks must provide the necessary bandwidth at reasonable energy levels.
- **software testing** Software must be evaluated for functional correctness and performance on the target platform.
- **software tool generation** Tools to program the system must be generated from the hardware and software architectures.

1.2.5 Joint Algorithm and Architecture Development

Embedded systems architectures may be designed along with the algorithms they will execute. This is true even in standards-based systems, since standards generally allow for algorithmic enhancements. Joint algorithm/architecture development creates some special challenges for system designers.

Algorithm designers need estimates and models to help them tailor the algorithm to the architecture. Even though the architecture is not complete, the hardware architects

should be able to supply estimates of performance and power consumption. These should be useful for simulators that take models of the underlying architecture.

Algorithm designers also need to be able to develop software. This requires functional simulators that run as fast as possible. If hardware were available, algorithm designers could run code at native speeds. Functional simulators can provide adequate levels of performance for many applications even if they don't run at hardware speeds. Fast turnaround of compilation and simulation is very important to successful software development.

1.3 Models of Computation

A **model of computation** defines the basic capabilities of an abstract computer. In the early days of computer science, models of computation helped researchers understand the basic capabilities of computers. In embedded computing, models of computation help us understand how to correctly and easily program complex systems. In this section we will consider several models of computation and the relationships between them. The study of models of computation have influenced the way real embedded systems are designed; we will balance the theory in this section with mentions of how some of these theoretical techniques have influenced embedded software design.

1.3.1 Why Study Models of Computation?

expressiveness

Models of computation help us understand the expressiveness of various programming languages. Expressiveness has several different aspects. On the one hand, we can prove that some models are more expressive than others---that some styles of computing can do some things that other styles can't. But expressiveness also has implications for programming style that are at least as important for embedded system designers. Two languages that rae both formally equally expressive may be good at very different types of applications. For example, control and data are often programmed in very different ways; a language may express one only with difficulty but the other easily.

language styles

Experienced programmers can think of several types of expressiveness that can be useful when writing programs:

- **control vs. data** This is one of the most basic dichotomies in programming. Although control and data are formally equivalent. we tend to think about them very differently. Many programming languages have been developed for control-intensive applications like protocol design. Similarly, many other programming languages have been designed for data-intensive applications like signal processing.

- **sequential vs. parallel** This is another basic theme in computer programming. Many languages have been developed to make it easy to describe parallel programs in a way that is both intuitive and formally verifiable. However, programmers still feel comfortable with sequential programming when they can get away with it.

- **communication** The nature of communication between units in a program is related to the way that parallelism is described. Sequential languages communicate by passing control. Various communication mechanisms have been developed for different applications. These may describe control-oriented vs. data-oriented communication. They may also embody different methods of buffering communicated data.

The astute reader will note that we aren't concerned here about some traditional programming language issues such as modularity. While modularity and maintainability are important, they are not unique to embedded computing. Some of the other aspects of languages that we mention are more central to embedded systems that must implement several different styles of computation so that they work together smoothly.

interoperability

Expressiveness may lead us to use more than one programming language to build a system.—we call these systems **heterogeneously programmed**. When we mix programming languages, we must satisfy the extra burden of correctly designing the communication between modules of different programming languages. Within a given language, the language system often helps us verify certain basic operations and it is much easier to think about how the program works. When we mix and match multiple languages, it is much more difficult for us to convince ourselves that the programs will work together properly. Understanding the model under which each programming language works and the conditions under which they can reliably communicate is a critical step in the design of heterogeneously programmed systems.

1.3.2 Finite vs. Infinite State

finite vs. infinite state

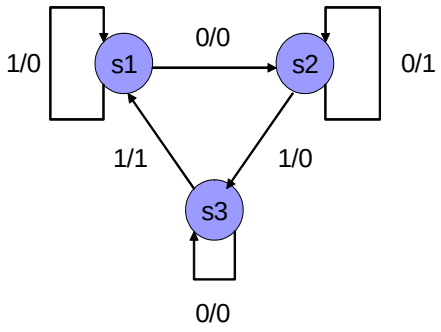
The amount of state that can be described by a model is one of the most fundamental aspects of any model of computation. Early work on computability emphasized the capabilities of finite-state vs. infinite-state machines; infinite state was generally considered to be good because it showed that the machine was more capable. However, finite-state models are much easier to verify in both theory and practice. As a result, finite-state programming models have an important place in embedded computing.

finite-state machine

Finite-state machines (**FSMs**) are well understood to both software and hardware designers. An example is shown in Figure 1-6. An FSM is typically defined as

$$M = \{I, O, S, \Delta, T\} \quad \text{(EQ 1-1)}$$

where I and O are the inputs and outputs of the machine, S is its current state, and Δ and T are the states and transitions respectively of the state transition graph. In a **Moore**



state transition graph

0	s1	s2	0
1	s1	s1	0
0	s2	s2	1
1	s2	s3	0
0	s3	s3	0
1	s3	s1	1

state transition table

Figure 1-6

A state transition graph and table for a finite-state machine.

machine, the output is a function only of *S*, while in a **Mealy machine** the output is a function of both the present state and the current input.

Although there are models for asynchronous FSMs, a key feature in the development of the finite-state machine model is the notion of synchronous operation: inputs are accepted only at certain moments. Finite-state machines view time as integer-valued, not real-valued. At each input, the FSM evaluates its state and determines its next state based upon the input received as well as the present state.

streams

In addition to the machine itself, we need to model its inputs and outputs. A **stream** is widely used as a model of terminal behavior because it describes sequential behavior—time as ordinals, not in real values. The elements of a stream are symbols in an alphabet. The alphabet may be binary, in some other base, or other types of values, but the stream itself does not impose any semantics on the alphabet. A stream is a totally ordered set of symbols $\langle s_0, s_1, \ldots \rangle$. A stream may be finite or infinite. Informally, the time at which a symbol appears in a stream is given by its ordinality in the stream. In this equation:

$$S(t) = s_t \qquad \text{(EQ 1-2)}$$

the symbol s_t is the t^{th} element of the stream *S*.

We can use streams to describe the input/output or terminal behavior of a finite-state machine. If we view the FSM as having several binary-valued inputs, the alphabet for

the input stream will be binary numbers; in some cases it is useful to think of the inputs as forming a group whose values are determined by a single symbol that defines the states of all the inputs. Similar thinking can be applied to the outputs. The behavior of the inputs is then described as one or more streams, depending on the alphabet used. Similarly, the output behavior is described as one or more streams. At time *i*, the FSM consumes a symbol on each of its input streams and produces a symbol on each of its output streams. The mapping from inputs to outputs is determined by the state transition graph and the machine's internal state. From the terminal view, the FSM is synchronous because the consumption of inputs and generation of outputs is coordinated.

verification and finite state

Although synchronous finite-state machines may be most familiar to hardware designers, synchronous behavior is a growing trend in the design of languages for embedded computing. Finite-state machines make interesting models for software because they can be more easily verified than infinite-state machines. Because an FSM has a finite number of states, we can visit all the states and exercise all the transitions in a finite amount of time. If a system has infinite state, we cannot visit all its states in finite time. Although it may seem impractical to walk through all the states of an FSM in practice, research over the past 20 years has led us to very efficient algorithms for exploring large state spaces. The **ordered Boolean decision diagram** (**OBDD**) [cite Randy Bryant here] can be used to describe combinational Boolean functions. Techniques have been developed to describe state spaces in terms of OBDDs such that properties of those state spaces can be efficiently checked in many cases. OBDDs do not take away the basic NP-completeness of combinational and state space search problems; in some cases the OBDDs can become very large and slow to evaluate. But in many cases they run very fast and even in pathological cases can be faster than competing methods.

OBDDs allow us to perform many checks that are useful tests of the correctness of practical systems:

- **product machines** It is often easier to express complex functions as systems of communicating machines. However, hidden bugs may lurk in the communication between those components. Building the product of the communicating machines is the first step in many correctness checks.

- **reachability** Many bugs manifest themselves as inabilities to reach certain states in the machine. In some cases, unreachable states may simply describe useless but unimportant behavior. In other cases, unreachable states may signal a missing feature in the system.

Non-deterministic FSMs, also known as **non-deterministic finite automata** (**NFAs**), are used to describe some types of systems.An example is shown in Figure 1-7: two transitions out of state *s1* have the same input label. One way to think about this model is that the machine non-deterministically chooses a transition such that future inputs will cause the machine to be in the proper state; another way to think about execution is that the machine follows all possible transitions simultaneously until future inputs cause it to prune some paths. It is important to remember that non-deterministic autom-

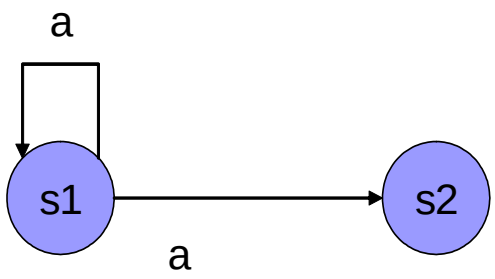


Figure 1-7

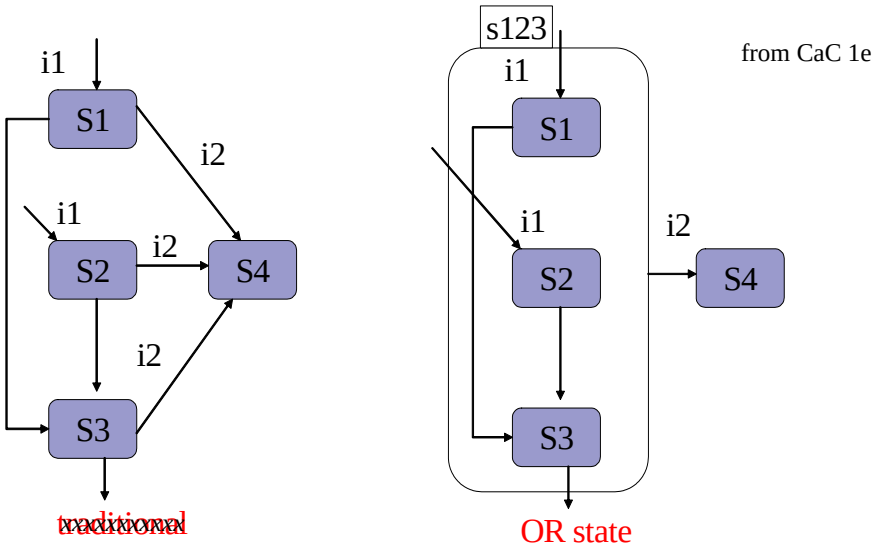
A non-deterministic FSM.

ata are not formally more expressive than deterministic FSMs. An algorithm can transform any NFA into an equivalent deterministic machine. But NFAs can be exponentially smaller than its equivalent deterministic machine. This is a simple but clear example of the stylistic aspect of expressiveness.

Statecharts

Another well-known, more stylistically expressive version of FSMs is the Statechart [CITE]. The Statechart formalism provides a hierarchy of states using two basic constructs: the AND state and the OR state. Statecharts are not formally more expressive

than standard FSMs—we can convert any Statechart to an FSM—but a Statechart may be exponentially smaller than its FSM equivalent.



change subcaption to standard to conform to text

Figure 1-8

A Statechart OR state.

Figure 1-8 illustrates the Statechart OR state. The left-hand side of the figure shows a fragment of a standard FSM. All four states have transitions to state *S4* when input *i2* appears. The OR state groups states *S1*, *S2*, and *S3* together and uses a single transition to

specify that input *i2* causes a transition from any state in the OR state (named *s123*) to state *S4*.

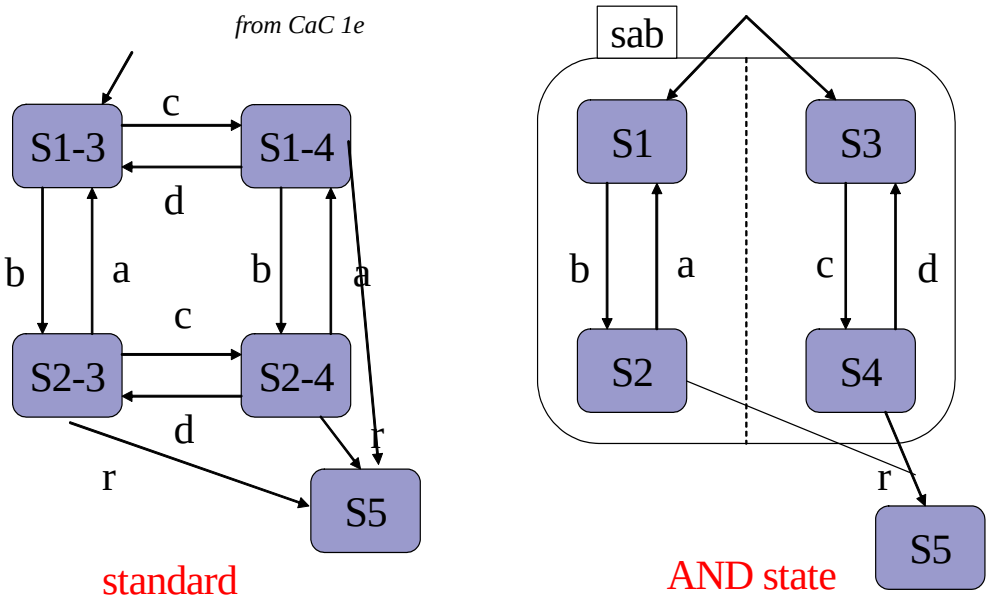


Figure 1-9

A Statechart AND state.

Figure 1-9 illustrates the Statechart AND state. In the standard FSM fragment, the states *S1-2*, *S1-4*, *S2-3*, and *S2-4* have a complex set of transitions between them. The AND state helps illustrate the structure in these transitions. AND state *sab* has two partitions: one containing *S1* and *S2*, the other containing *S3* and *S4*. In the first partition, inputs a and b cause transitions between *S1* and *S2*. Similarly, inputs *c* and *d* cause transitions between *S3* and *S4* in the other partition. When the machine enters state *sab*, it simultaneously executes both partitions. The machine can therefore be in any combination of $\{S1, S2\} \times \{S3, S4\}$.

Statecharts are an important variation in finite-state machines because they make specifications smaller and easier to understand. Statecharts and their variations have been used in many software projects. For example, the TCAS-II aviation collision avoidance system was designed using a language created by Leveson [Lev94] that included Statechart-style hierarchical states.

Turing machines

The **Turing machine** is the most well-known infinite-state model for computation. (Church developed his lambda calculus first, but the Turing machine more closely models the operation of practical computing machines.) The Turing machine itself consists of a program, a read head, and a state. The machine reads and writes a tape that has been divided into cells, each of which contains a symbol. The tape can move back and forth

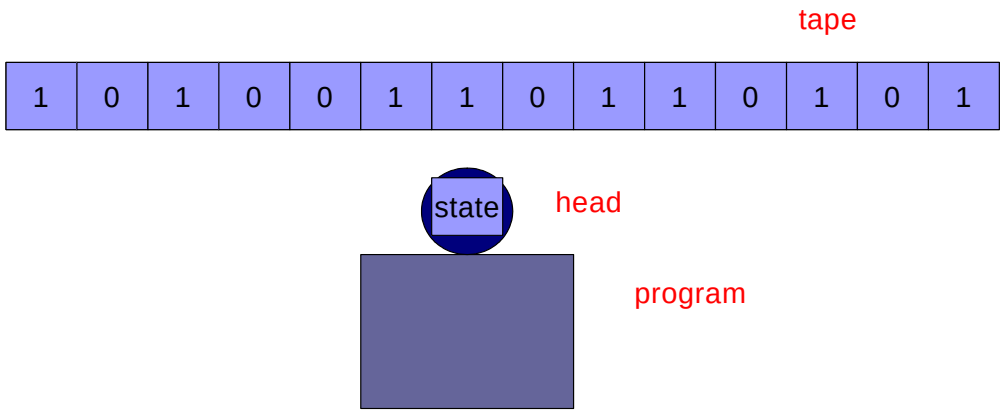


Figure 1-10

A Turing machine.

underneath the head; the head can both read and write symbols on the tape. Because the tape may be of infinite length, the Turing machine can describe infinite-state computations.

An operating cycle of a Turing machine consists of several steps:

- The machine uses the head to read the symbol in the tape cell underneath the head.
- It erases the symbol on the cell underneath the head.
- The machine consults its program to determine what to do next. Based upon the current state and the symbol that was read, the machine may write a new symbol and/or move the tape.
- The machine changes its state as described by the program.

The Turing machine is a powerful model that allows us to demonstrates the capabilities and limits of computability. However, as we noted above, finite state allows us to verify many important aspects of real programs even though the basic programming model is more limited. For example, one of the key results of theoretical computer science is the **halting problem**—the Turing model allows us to show that we cannot, in general, show that an arbitrary program will halt in a finite amount of time. The failure to ensure that programs will halt makes it impossible to verify many important problems of programs on infinite-state systems. In contrast, because we can visit all the states of a finite-state system in finite time, important properties become more tractable.

1.3.3 Parallelism and Communication

Parallelism is a fundamental concept in computer science and of great practical importance in embedded systems. Many embedded systems perform many tasks simultaneously. The real parallelism embodied in the hardware must be matched by apparent parallelism in the programs.

parallelism and architecture

We need to capture parallelism during the early stages of design so that we can use it to optimize our design. Parallel algorithms describe time as partially ordered—the exact sequence of operations is not determined up front. As we bind operations to the architecture, we move the description toward a totally ordered description (although some operations may be left partially ordered to be managed by the operating system). Different choices for ordering require different amounts of hardware resources, affecting cost and power consumption.

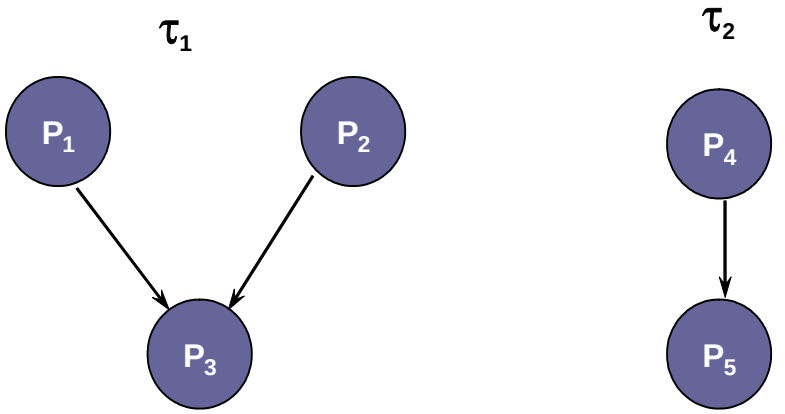


Figure 1-11

A task graph.

task graphs

A simple model of parallelism is the **task graph** as shown in Figure 1-11. The nodes in the task graph represent **processes** or **tasks** while the directed edges represent data dependencies. In the example, process *P4* must complete before *P5* can start. Task graphs model concurrency because sets of tasks that are not connected by data dependencies may operate in parallel. In the example, τ_1 and τ_2 are separate components of the graph that can run independently. Task graphs are often used to represent multi-rate systems. Unless we expose the computation within the processes, a task graph is less powerful than a Turing machine. The basic task graph cannot even describe conditional behavior. Several extended task graphs have been developed that describe conditions but even these are finite-state machines

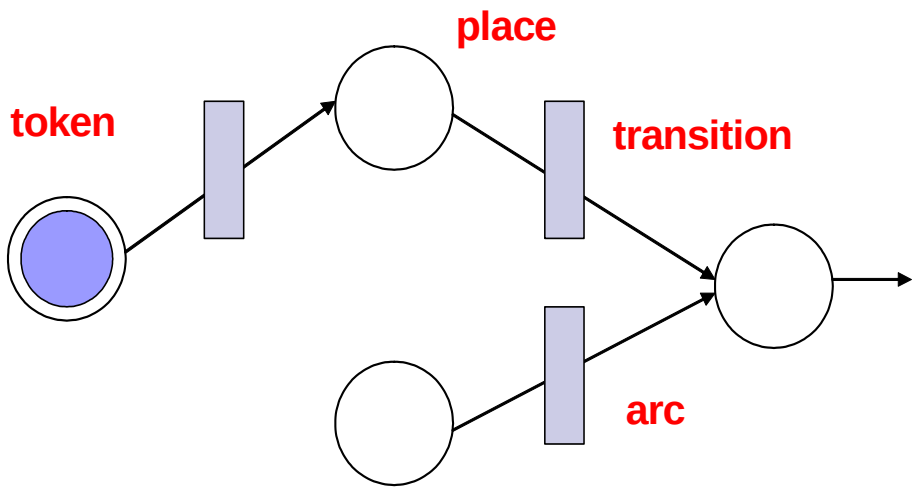


Figure 1-12

A Petri net.

Petri nets

The **Petri net** is one well-known parallel model of computation. Petri nets were originally considered to be more powerful than Turing machines, but later work showed that the two are in fact equivalent. However, Petri nets explicitly describe parallelism in a way that makes some types of programs easier to write. An example Petri net is shown in Figure 1-12. A net, which is a form of program, consists of three types of objects. places that hold state; arcs that define how state can move from place to place; and transitions that guard the arcs. The state of the executing system is defined by tokens. The tokens move around the net in accordance with firing rules. Parallelism is easy to express in a Petri net because the net can have several tokens flowing through it at once.

communication in FSMs

Petri nets have been used to study many problems in parallel programming. They are sometimes used to write parallel programs, but are not often used directly as programs. However, the notion of multiple tokens is a powerful one that serves us well in many types of programs.

Useful parallelism necessarily involves communication between the parallel components of the system. Different types of parallel models use different styles of communication. These styles can have profound implications on the efficiency of the implementation of communication. We can distinguish two basic styles of communication: **buffered** and **unbuffered**. A buffered communication assumes that memory is available to store a value if the receiving process is temporarily not ready to receive it. An unbuffered model assumes no memory in between the sender and receiver.

communication in FSMs

Even a simple model like the FSM can exhibit parallelism and communication. Figure 1-13 shows two communicating FSMs. Each machine, *M1* and *M2*, has an input

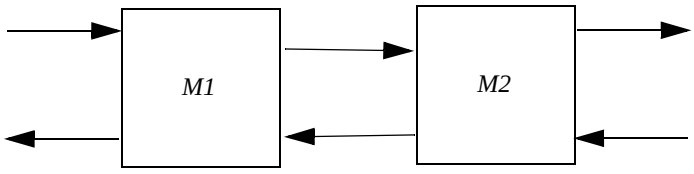


Figure 1-13

Two communicating FSMs.

from the outside world and an output to the outside world. But each has one output connected to the input of the other machine. The behavior of each machine therefore depends on the behavior of the other machine. As we noted before, the first step in analyzing the behavior of such networks of FSMs is often to form the equivalent product machine.

insert Esterel code here

Figure 1-14

An example Esterel program.

synchronous languages

Communicating FSM languages have been used for software as well as hardware. Figure 1-14 shows an example of Esterel code [cite]. As we will see in Chapter 3, each process in an Esterel program is considered as finite state machine and the behavior of the system of process is determined by building the product of the component machines. Esterel has been widely used to program avionics and other critical applications.

The communicating FSMs of Figure 1-13 communicate without buffers. A buffer would correspond to a register (in hardware) or variable (in software) in between and output on one machine and the corresponding input on the other machine. However, we can implement both synchronous and asynchronous behavior using this simple unbuffered mechanism as shown in Figure 1-15. Synchronous communication simply has one machine throw a value to the other machine. In the figure, the synchronously communicating *M1* sends val to *M2* without checking whether *M2* is ready. If the machines are designed properly, this is very efficient, but if *M1* and *M2* fall out of step then *M2* will

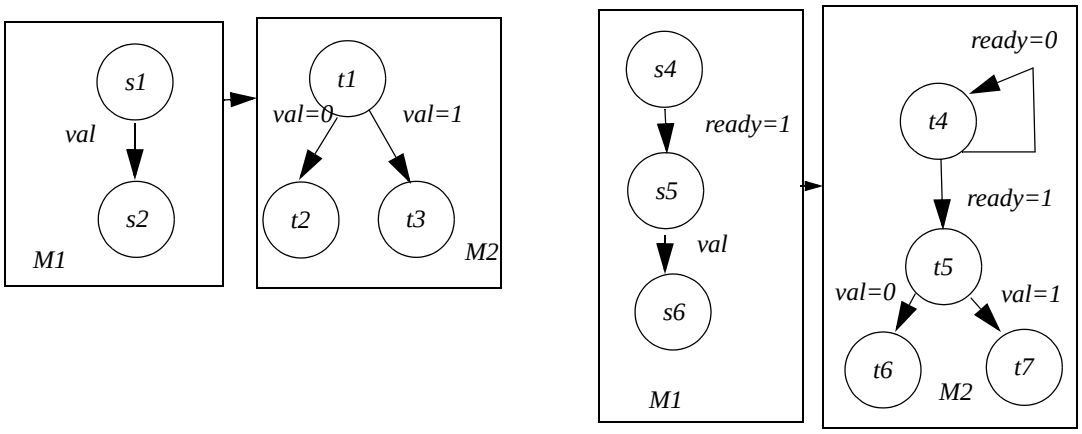


Figure 1-15

Synchronous and asynchronous communication in FSMs.

misbehave because val is either early or late. Asynchronous communication uses a handshake. On the right-hand side of the figure, the asynchronous *M1* first sends a ready signal, then a value. *M2* waits for the ready signal before looking for *val*. This requires extra states but also does not require that the machines move in lockstep.

blocking vs. unblocking

Another fundamental distinction between communication methods, **blocking** vs. **unblocking** behavior. In blocking communication, the sending process blocks, or waits until the receiving process has the data. Unblocking communication does not require the sender to wait for the receiver to receive the data. If there are no buffers between the sender and receiver, unblocking communication will drop data if the receiver is not ready. Adding a buffer allows the sender to move on even if the receiver is not ready, assuming that the buffer is not already full. An infinite-size buffer allows unlimited unblocking communication.

buffering and communication

A natural question in the case of buffered communication is the size of the buffer required. In some systems, there may be cases in which an infinite-size buffer is required to avoid losing data. In a multi-rate system in which the sender may always produce data faster than the consumer, the buffer size may grow indefinitely. However, it may be possible to show that the producer cannot keep more than some finite number of elements in the buffer even in the worst case. If we can prove the size of the buffer required, we can create a cheaper implementation. Proving that the buffer is finite also tells us that it is possible to build a system in which the buffer never overflows. As with other problems, proving buffer sizes is easier in finite-state systems.

1.3.4 Control Flow and Data Flow Models

Control and data are fundamental units of programming. Although control and data are fundamentally equivalent, we tend to think of data operations as more regular, such as arithmetic, and control as less regular and more likely to involve state.

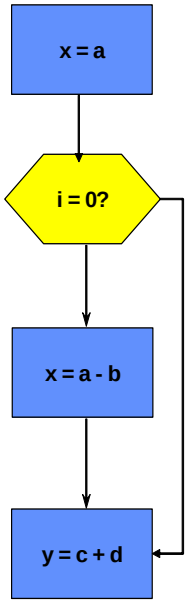


Figure 1-16
A control flow graph.

control flow graph

A basic model of control is the **control flow graph** (**CFG**) as shown in Figure 1-16. The nodes in the graph are either unconditionally executed operations (the rectangles) or conditions (the diamonds). In this case we have decorated the rectangles with operations performed at those states, but those operations are not strictly part of the control flow graph model. The control flow graph has a single thread of control, which can be thought of as a program counter moving through the program. This is a finite-state model of computation. Many compilers model a program using a **control data flow graph** (**CDFG**), which adds data flow models that we will describe in a moment to describe the operations of the unconditional nodes and the decisions in the conditional nodes.

basic data flow graphs

A basic model of data is the **data flow graph** (**DFG**), an example of which is shown in Figure 1-17. Like the task graph, the data flow graph consists of nodes and directed edges, where the directed edges represent data dependencies. The nodes in the DFG represent the data operations, such as arithmetic operations. Some edges in the DFG terminate at a node but do not start at a node; these sources provide inputs. Similarly, sinks start at a node but do not terminate at a node. (An alternative formulation is to provide three types of nodes: operator, input, and output.)

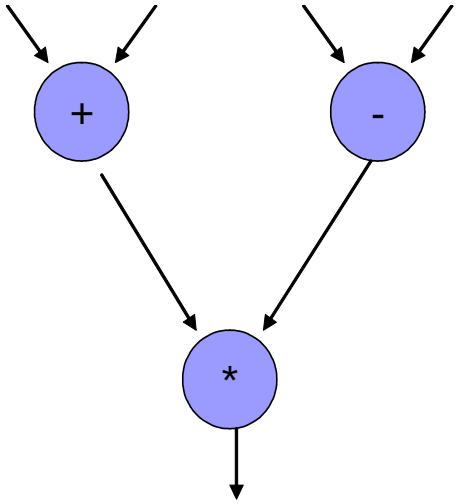


Figure 1-17

A data flow graph.

We require that DFGs be trees—they cannot have cycles. This makes the graphs easier to analyze but does limit their uses. Basic data flow graphs are commonly used in compilers.

The data flow graph is finite-state. It describes parallelism in that it defines only a partial order on the operations in the graph. Whether we execute those operations one at a time or several at once, any order of operations that satisfies the data dependencies is acceptable.

streams and firing rules

We can use streams to model the behavior of the data flow graph. Each source in the data flow graph has its own stream and each sink of the data flow graph is a stream as well. The nodes in the DFG use **firing rules** to determine their behavior. The simplest firing rule is similar to the operation of finite-state machines: firing consumes a token on each of the node's input streams and generates one token on its output; we will call this the **standard data flow firing rule**. One way to introduce conditions into the DFG is with a conditional node with n+1 terminals: data inputs d_0,d_1,... and control input k. When k=0, data input d_0 is consumed and sent to the output; when k=1, d_1 is consumed and transferred to the output, etc. In this firing rule, not all of the inputs to the node consume a token at once.

signal flow graphs

A slightly more sophisticated version of data flow is the **signal flow graph** (**SFG**) commonly used in signal processing. As shown in Figure 1-18, the signal flow graph adds a new type of node generally called a **delay** node. As signified by the Δ symbol, the delay node delays a stream by n (by default, one) time steps. Given a stream S, the result of a delay operator is $\Delta(t) = S(t-1)$. Edges in the SFG may be given weights that

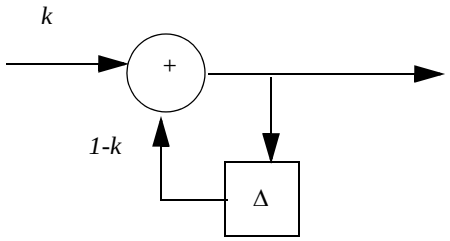


Figure 1-18

A signal flow graph.

indicate that the value given to a node is to be multiplied by the weight. We also allow signal flow graphs to have cycles. SFGs are commonly used to describe digital filters.

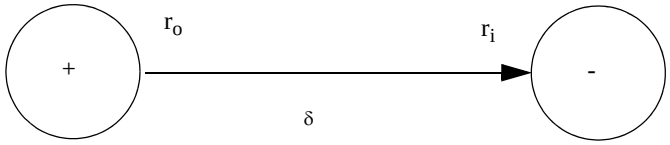


Figure 1-19
A simple synchronous data flow graph.

synchronous data flow

A more sophisticated data flow model is the **synchronous data flow** (**SDF**) model introduced by Lee and Messerschmitt. Synchronous data flow graphs allow feedback and provide methods for us to determine when a system with feedback is, in fact, legal. A simple SDF graph is shown in Figure 1-19. As with basic data flow graphs, nodes define operations and directed edges define the flow of data. The data flowing along the edges can be modeled as streams. Each edge has two labels: r_o describes the rate at which the node at the source of this edge produces tokens while r_i describes the rate at which the sink node of the edge consumes tokens. Each edge may also be labeled with a delay δ that describes the amount of time between when a token is produced at the source and when it is consumed at the edge; by convention the default delay is 0.

We can form these graphs into graphs that describe the flow of streams through systems. These graphs may have cycles. We will defer a detailed discussion of the analysis of SDF graphs to Chapter 3, but we can analyze these graphs to determine whether the systems they describe are feasible. For example, consider the SDF graph of Figure 1-20, which was originally described by Lee and Messerschmitt [CITE]. There are two paths

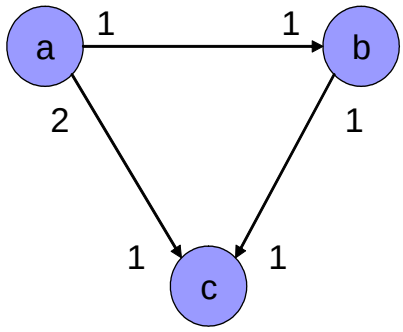


Figure 1-20

An infeasible synchronous data flow graph.

to node *c*: *a* -> *c* and *a* -> *b* -> *c*. Node *a* emits tokens at twice the rate at which c emits them. This is not inherently bad, but the path provides tokens from a to c at half that rate. As *a* result, the flow from a to be is imbalanced and the system is infeasible.

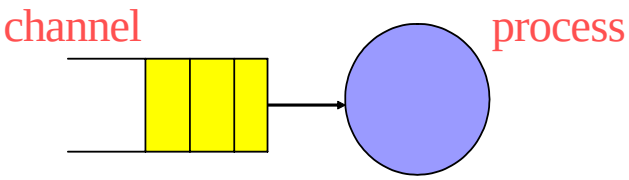


Figure 1-21

A Kahn process.

Kahn process networks

Lee and Sangiovanni-Vincentelli [CITE] identified the networks of **Kahn processes** as important models for systems of communicating processes. A Kahn process is illustrated in Figure 1-21. The process proper is connected to its inputs by infinite-size buffers. Streams model the inputs and outputs of the Kahn process during execution; the process maps its input streams to output streams. A process may have one or more inputs and one or more outputs. If *X* is a stream, then *F(X)* is the output of a Kahn process when given that stream. One important property of a Kahn process is **monotonicity**:

$$X \in X' \Rightarrow F(X) \in F(X'). \qquad \text{(EQ 1-3)}$$

A monotonic process's behavior is physical in that adding more inputs will not cause it to mysteriously generate fewer outputs.

A network of Kahn processes equates the input and output streams of processes in the network. If I is the input stream to a network and X is the set of internal streams and outputs, then the fixed point behavior of the network is

$$X = F(X, I). \quad \text{(EQ 1-4)}$$

Kahn showed that a network of monotonic processes is itself monotonic.

1.4 Reliability, Safety, and Security

In this section we will look at aspects reliable system design that are particularly important to embedded system design. The three areas in the title of this section are closely related:

- **Reliable (or dependable) system design** is concerned with making systems work even in the face of internal or external problems. Reliable system design most often assumes that problems are not caused maliciously.
- **Safety-critical system design** studies methods to make sure systems operate safely, independent of what causes the problem.
- **Security** is concerned largely with malicious attacks.

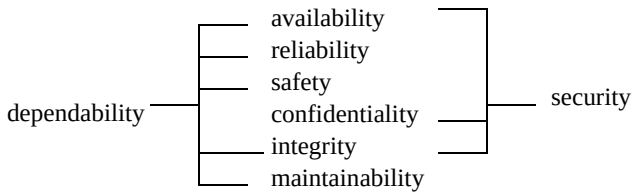


Figure 1-22

Dependability and security [Avi04].

Avizienis et al [Avi04] describe the relationship between dependability and security as shown in Figure 1-22. Dependability and security are composed of several attributes:

- **availability** for correct service
- **continuity** of correct service
- **safety** from catastrophic consequences on users and their environment

- **integrity** from improper system alterations
- **maintainability** through modifications and repairs;
- **confidentiality** of information.

Embedded systems are increasingly subject to malicious attack. But whatever the source of the problem, many embedded systems must operate properly in the presence of faults.

1.4.1 Why Reliable Embedded Systems?

applications demand reliability

Certainly many embedded systems do not need to be highly reliable. Some consumer electronics devices are so inexpensive as to be nearly disposable. Many markets do not require highly reliable embedded computers. But many embedded computers must be built to be highly reliable:

- automotive electronics;
- avionics;
- medical equipment;
- critical communications systems.

Embedded computers may also handle critical data, such as purchasing data or medical information.

The definition of reliability can vary widely with context. Certainly, computer systems that run for weeks at a time without failing are not unknown. Telephone switching systems have been designed to be down for less than 30 seconds per year.

new problems

The study of reliable digital system design goes back several decades. A variety of architectures and methodologies have been developed to allow digital systems to operate for long periods with very low failure rates. What is different between the design of these traditional reliable computers and reliable embedded systems?

First, reliable embedded computers are often distributed systems. Automotive electronics, avionics, and medical equipment are all examples of distributed embedded systems that must be highly reliable. Distributed computing can work to our advantage when designing reliable systems but distributed computers are can also be very unreliable if improperly designed.

Second, embedded computers are vulnerable to many new types of attacks. Reliable computers were traditionally servers or machines that were physically inaccessible—physical security has long been a key strategy for computer security. However, embedded computers generally operate in unprotected environments. This allows for new types of faults and attacks that require new methods of protection.

1.4.2 Fundamentals of Reliable System Design

sources of faults

Reliable systems are designed to recover from **faults**. A fault may be **permanent** or **transient**. A fault may have many sources:

- **Physical faults** come from manufacturing defects, radiation hazards, etc.
- **Design faults** are the result of improperly designed systems.
- **Operational faults** come from human error, security breaches, poorly designed human-computer interfaces, etc.

While the details of how these faults happen and how they affect the system may vary, the system's users do not really care what caused a problem, only that the system reacted properly to the problem. Whether a fault comes from a manufacturing defect or a security problem, the system must react in such a way to minimize the fault's effect on the user.

system reliability metrics

Users judge systems by how reliable they are, not by the problems that cause them to fail. Several metrics are used to quantify system reliability {Sie98].

Mean time to failure (**MTTF**) is one well-known metric. Given a set of perfectly functioning systems at time 0, MTTF is the expected time for the first system in that set to fail. Although it is defined for a large set of systems, it is also often used to characterize the reliability of a single system. The mean time to failure can be calculated by

$$[TTF = \int_{0}^{\sim} R(t)(\quad \text{(EQ 1-5)}$$

where *R(t)* is the reliability function of the system.

The **reliability function** of a system describes the probability that the system will operate correctly in the time period [0, *t*]. *R(0)* = 1 and *R(t)* monotonically decreases with time.

The hazard function *z(t)* is the failure rate of components. For a given probability function, the hazard function is defined as

$$(t) = \frac{pdf}{1 - CD}. \quad \text{(EQ 1-6)}$$

characterizing faults

Faults may be measured empirically or modeled by a probability distribution. Empirical studies are usually the basis for choosing an appropriate probability distribution. One common model for failures is the exponential distribution. In this case, the hazard function is

$$z(t) = \lambda \quad \text{(EQ 1-7)}$$

Another function used to model failures is the Weibull distribution:

$$z(t) = \alpha\lambda(\lambda t)^{\alpha-1}. \quad \text{(EQ 1-8)}$$

In this formula, α is known as the shape parameter and λ is known as the scale parameter. The Weibull distribution must normally be solved numerically.

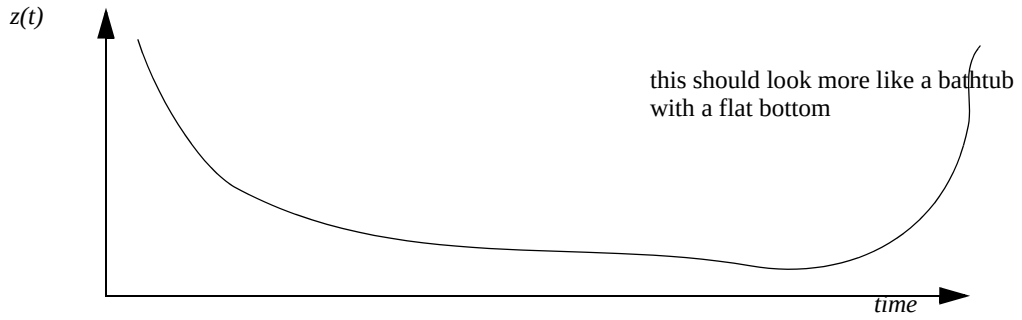


Figure 1-23

A bathtub distribution.

A distribution that is observed empirically for many hardware components is the **bathtub distribution** shown in Figure 1-23. The bathtub curve gets its name from its similarity to the cross section of a bathtub. Hardware components generally show infant mortality in which marginal components fail quickly, then a long period with few failures, followed by a period of increased failures due to long-term wear mechanisms.

actions after faults

The system can do many things after a fault. Generally several of these actions are taken in order until an action gets the system back into running condition. Actions from least to most severe include:

- **fail** All too many systems fail without trying to even detect an error.
- **detect** An error may be detected. Even if the system stops at this point, the diagnostic information provided by the detector can be useful.
- **correct** An error may be corrected. Memory errors are routinely corrected. A simple correction causes no long-term disturbance to the system.
- **recover** A recovery may take more time than a simple correction. For example, a correction may cause a noticeable pause in system operation.

- **contain** The system may take steps to ensure that a failure does not corrupt a large part of the system. This is particularly true of software or hardware failures that can, for example, cause large parts of memory to change.

- **reconfigure** One way to contain a fault is to reconfigure the system so that different parts of the system perform some operations. For example, a faulty unit may be disabled and another unit enabled to perform its work.

- **restart** Restarting the system may be the best way to wipe out the effects of an error. This is particularly true of transient errors and software errors.

- **repair** Either hardware or software components may be modified or replaced to repair the system.

reliability methods

Many techniques have been developed to make digital systems more reliable. Some are more applicable to hardware, others to software, and some may be used in both hardware and software.

error correction codes

Error correction codes were developed in the 1950's, starting with Hamming, to both detect and correct errors. These codes introduce redundant information in a way such that certain types of errors can be guaranteed to be detected or corrected. For example, a code that is single error correcting/double error detecting can both detect and correct an error in a single bit and detect, but not correct, two bit errors.

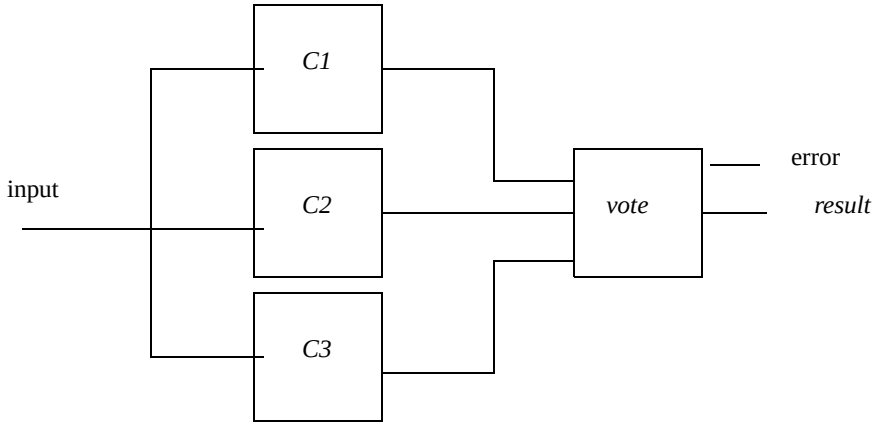


Figure 1-24

Triple modular redundancy.

voting systems

Voting schemes are often used to check at higher levels of abstraction. One well-known voting method is **triple modular redundancy**, illustrated in Figure 1-24. The computation unit C has three copies, C1, C2, and C3. All three units receive the same input. A separate unit compares the results generated by each input. If at least two results

agree, then that value is chosen as correct by the voter. If all three results differ, then no correct result can be given.

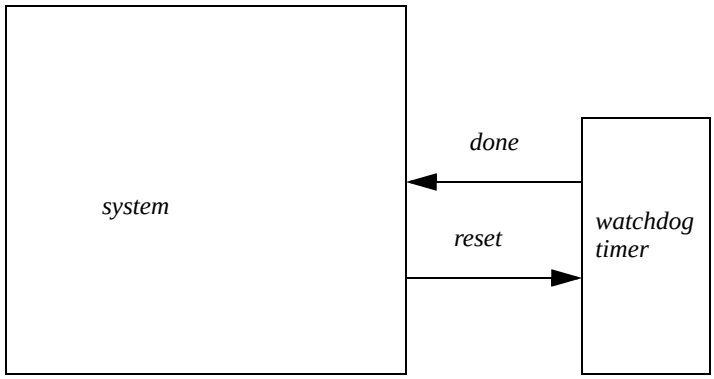


Figure 1-25

Watchdog timer.

watchdog timers

The **watchdog timer** is widely used to detect system problems. As shown in Figure 1-25, the watchdog timer is connected to a system that it watches. If the watchdog timer rolls over, it generates a done signal that should be attached to an error interrupt in the system. The system should be designed so that, when running properly, it always resets the timer before it has a chance to roll over. Thus, a **done** signal from the watchdog timer indicates that the system is somehow operating improperly. The watchdog timer can be used to guard against a wide variety of faults.

design diversity

Design diversity is a design methodology intended to reduce the chance that certain systematic errors creep into the design. When a design calls for several instances of a given type of module, different implementations of that module are used rather than using the same type of module everywhere. For example, a system with several CPUs may use several different types of CPUs rather than use the same type of CPU everywhere. In a triple modular redundant system, the components that produce results for voting may be of different implementations to decrease the chance that all embody the same design error.

1.4.3 Novel Attacks and Countermeasures

physical access

A key reason that embedded computers are more vulnerable than general-purpose computers is that many embedded computers are physically accessible to attackers. Physical security is an important technique used to secure the information in general-purpose sys-

tems—servers are physically isolated from potential attackers. When embedded computers with secure data are physically available, the attacker can gain a great deal more information about the hardware and software. This information can be used not only to attack that particular node but also helps the attacker develop ways to interfere with other nodes of that model.

Internet attacks

Some attacks on embedded systems are made much easier by Internet access. Many embedded systems today are connected to the Internet. Viruses can be downloaded or other sorts of attacks can be perpetrated over the Internet. Siewiorek et al [Sie04] argue that global volume is a key trend in reliable computing systems. They point out that hundreds of millions of networked devices are sold each year, primarily to users with little or no formal training. The combination of large numbers of devices and untrained users means that many tasks formerly performed in the privacy of a machine room must now be automated and reliably delivered to the masses and that these systems must be designed to shield against both faults and malicious attacks.

attacks on automobiles

But many devastating problems can be caused without Internet access Consider, for example, attacks on automobiles. Most modern cars uses microprocessors to control their engines and many other microprocessors are used throughout the car. The software in the engine controller could, for example, be changed to cause the car to stall under certain circumstances. This would be annoying or occasionally dangerous when performed on a single car. If a large number of cars were programmed to all stall at the same time, the resulting traffic accidents could cause significant harm. This sort of programmed accident is arguably worse if only some of the cars on the road have been programmed to stall.

Clearly, this stalling accident could be perpetrated if automobiles provided Internet access to the engine controller software. Prototype cars have demonstrated Internet access to at least part of the car's internal network. However, Internet-enabled cars are not strictly necessary. Auto enthusiasts have reprogrammed engine controllers for over 20 years to change the characteristics of their engine. A determined attacker could spread viruses through auto repair shops.

battery attack

One novel category of attack is the **battery attack.** This attack tries to disable the node by draining its battery. If a node is operated by a battery, the node's power management system can be subverted by network operations. For example, pinging a node over the Internet may be enough to cause it to operate more often than intended and drain its battery prematurely.

Battery attacks are clearly threats to battery-operated devices like cell phones and PDAs. Consider, for example, a cell phone virus that causes the phone to repeatedly make calls. Cell phone viruses have already been reported [Jap05]. But many other devices use batteries even though they also receive energy from the power grid. The battery may be used to run a real-time clock (as is done in many PCs) or to maintain other system state. A battery attack on this sort of device could cause problems that would not be noticed for quite some time.

QoS attacks

Denial-of-service attacks are well-known in general-purpose systems, but real-time embedded systems may be vulnerable to **quality-of-service** (**QoS**) attacks. If the network delivers real-time data, then small delays in delivery can cause the data to be use-

less. If that data is used for real-time control, then those small delays can cause the system to fail. We also refer to this as a **timing attack** because it changes the real-time characteristics of the system. A QoS or timing attack is powerful because its effects are not just limited to information. The dynamics of the system being controlled help to determine the response of the system. A relatively small change in timing can cause a great deal of damage if a large, heavy, fast-moving object is being controlled.

attacks on sensor networks

Wood and Stankovic [Woo02] identified a number of ways to perform denial-of-service attacks on sensor networks at different levels of the network hierarchy:

- **physical layer** jamming, tampering
- **link layer** collision, exhaustion, unfairness
- **network and routing layer** neglect and greed, horning, misdirection, black holes, authorization, probing, redundancy
- **transport layer** flooding, desynchronization

power attack

An example of an attack that is much more easily used against embedded computers than general-purpose computers is the **power attack.** Karcher showed that measurements of the power supply current of a CPU can be used to determine a great deal about the processor's internal activity. Karcher used differential power analysis to determine the bits in cryptokeys—the processor performed different operations for the 1 and 0 case, which consumed different amounts of power. This attack was originally aimed at smart cards, which draw their power from the external card reader, but it can be applied to many embedded systems.

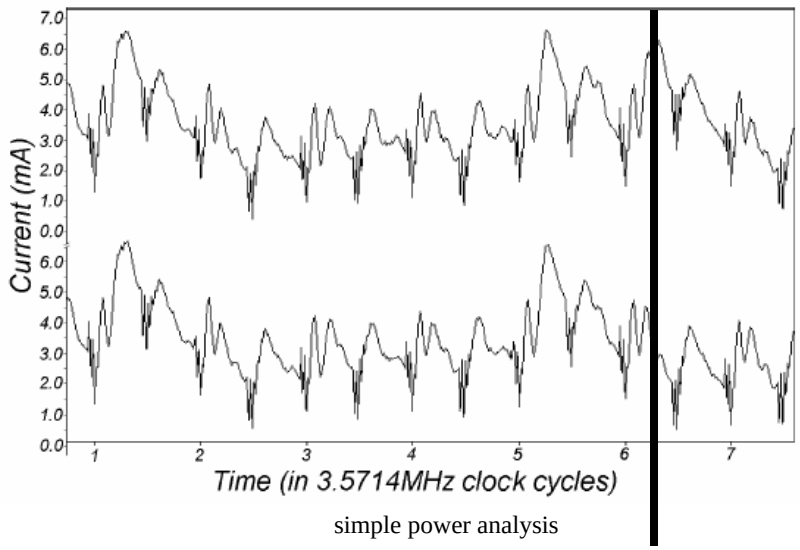
Figure 1-26 illustrates two methods of power attacks developed by Karcher. **Simple power analysis** inspects a trace manually and tries to determine the location of program actions such as branches, based upon knowledge of the power consumption of various CPU operations. Based upon program actions, the attacker then deduces bits of the key. **Differential power analysis** uses correlation to identify actions and key bits.

physical security

In some cases it may be possible to build tamper-resistant embedded systems. Making the electronic devices hard to reveal and analyze will slow down attackers. Limiting information within a chip also helps make data harder for attackers to reveal.

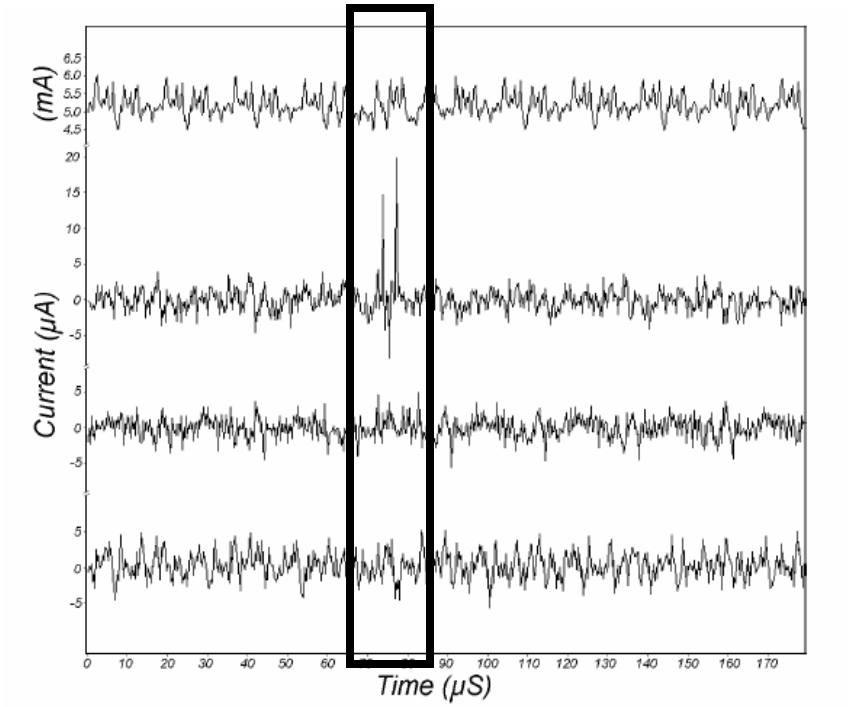
1.5 Example Applications

Some knowledge of the applications that will run on an embedded system will run is of great help to system designers. In this section we will look at some basic concepts in two common applications, communications/networking and multimedia.



simple power analysis

from Karcher check cite, copyright



reference trace

correct guess

incorrect guesses

differential power analysis

Figure 1-26

Power attacks.

1.5.1 Radio and Networking

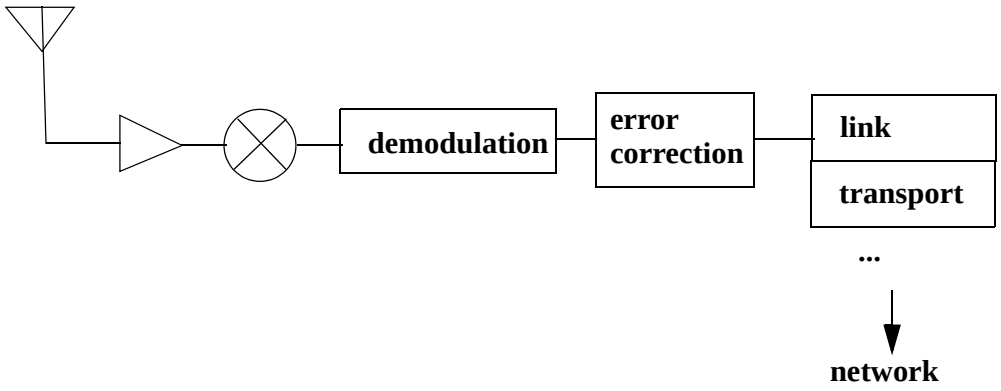


Figure 1-27

A radio and network connection.

combined wireless/network communications

Modern communications systems combine wireless and networking. As illustrated in Figure 1-27, radios carry digital information and are used to connect to networks. Those networks may be specialized, as in traditional cell phones, but increasingly radios are used as the physical layer in Internet protocol systems.

wireless

We will concentrate on receivers because they have the somewhat harder job of detecting data in the presence of noise. Radios for digital communication must perform several tasks:

- They must **demodulate** the signal down to the baseband.
- They must **detect** the baseband signal to identify bits.
- They must **correct errors** in the raw bit stream.

Demodulation requires multiplying the received signal by a signal from an oscillator and filtering the result to select the lower-frequency version of the signal. There are two alternative approaches to demodulation. **Superheterodyne receivers** take the signal through an **intermediate frequency** (**IF**) before demodulating again to the baseband. **Direct conversion** receivers do not use an intermediate frequency.

digital demodulation

Today, CPUs are used primarily for baseband operations because radio-frequency or intermediate-frequency rates are too high. The bit detection process depends somewhat on the modulation scheme, but digital communication mechanisms often rely on phase. High-data-rate systems often use multiple frequencies arranged in a constellation, as shown in Figure 1-28. The phases of the component frequencies of the signal can be

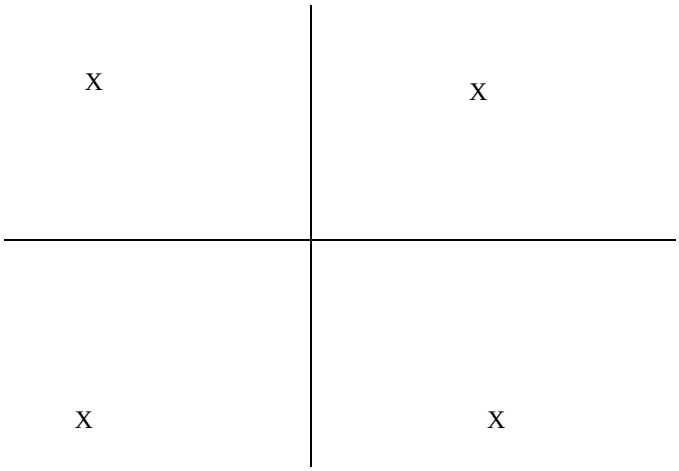


Figure 1-28

A constellation in a digital communications system.

modulated to create different symbols. The set of frequencies and phases used is known as a **constellation**.

error correction

Traditional error correction codes can be checked using combinational logic. Several more powerful codes that require iterative decoding have recently become popular. **Turbo codes** and **low-density parity check** (**LDPC**) codes both require multiple iterations to determine errors and corrections.

networking

A radio may simply act as the physical layer of a standard network stack, but many new networks are being designed that take advantage of the inherent characteristics of wireless networks. For example, traditional wired networks have only a limited number of nodes connected to a link but radios inherently broadcast; broadcast can be used to improve network control, error correction, and security. Wireless networks are generally **ad-hoc** in that the members of the network are not pre-determined and nodes may enter or leave during network operation. Ad-hoc networks require somewhat different network control than is used in fixed, wired networks.

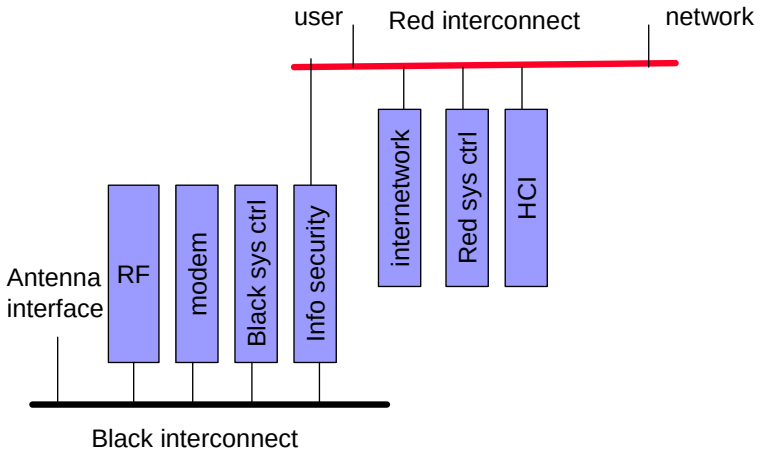
The next example describes a major effort to develop software radios for data communication.

Application Example 1-2

Joint Tactical Radio System

The Joint Tactical Radio System (JTRS) [Joi05,Ree02] is an initiative of the U. S. Department of Defense to develop next-generation communication systems based on radios that perform many functions in software. JTRS radios are designed to provide secure communication. They are also designed to be compatible with a wide range of existing radios as well as upgradeable through software.

The reference model for the hardware architecture has two major components:



The black subsystem performs low-level radio operations while the red subsystem performs higher-level network functions. The information security enforcement module that connects them helps protect the radio and the network from each other.

1.5.2 Multimedia

Today's dominant multimedia applications are based on compression: Digital television and radio broadcast, portable music, and digital cameras all rely on compression algorithms. In this section we will review some of the algorithms developed for multimedia compression.

lossy compression and perceptual coding

It is important to remember that multimedia compression methods are lossy—the decompressed signal is different from the original signal before compression. Compression algorithms make use of **perceptual coding** techniques that try to throw away data that is less perceptible to the human eye and ear. These algorithms also combine lossless compression with perceptual coding to efficiently code the signal.

JPEG-style image compression

The **JPEG** standard is widely used for image compression. The two major techniques used by JPEG are the **discrete cosine transform** (**DCT**) plus quantization, which performs perceptual coding, plus **Huffman coding** for lossless encoding.

The discrete cosine transform is a frequency transform whose coefficients describe the spatial frequency content of an image. Because it is designed to transform images, the DCT operates on a two-dimensional set of pixels, in contrast to the Fourier transform, which operates on a one-dimensional signal. However, the advantage of the DCT over other 2-D transforms is that it can be decomposed into two 1-D transforms, making it much easier to compute. The form of the DCT (CHECK THIS) is

$$v(k) = \alpha(k) \sum_{1 \le t \le N} u(t) \cos\left[\pi(2t+1)\frac{k}{2N}\right]. \qquad \text{(EQ 1-9)}$$

Many efficient algorithms have been developed to compute the DCT.

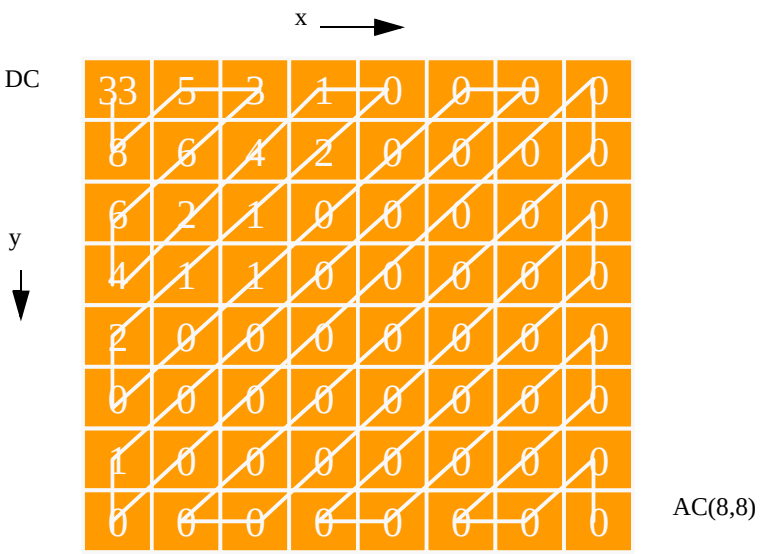


Figure 1-29
The zig-zag pattern used to transmit DCT coefficients.

JPEG performs the DCT on 8 x 8 **blocks** of pixels. The discrete cosine transform itself does not compress the image. The DCT coefficients are quantized to add loss and change the signal in such a way that lossless compression can more efficiently compress them. Low-order coefficients of the DCT correspond to large features in the 8 x 8 block and high-order coefficients correspond to fine features. Quantization concentrates on changing the higher-order coefficients to zero. This removes some fine features but provides long strings of zeroes that can be efficiently encoded to lossless compression.

Huffman coding, which is sometimes called **variable length coding**, forms the basis for the lossless compression stage. As shown in Figure 1-29, a specialized technique is used to order the quantized DCT coefficients in a way that can be easily Huffman encoded. The DCT coefficients can be arranged in an 8 x 8 matrix. The 0,0 entry at the

top left is known as the **DC coefficient** since it describes the lowest-resolution or DC component of the image. The 8,8 entry is the highest-order AC coefficient. Quantization has changed the higher-order AC coefficients to 0. If we were to traverse the matrix in row or column order, we would intersperse non-zero lower-order coefficients with higher-order coefficients that have been zeroed. By traversing the matrix in a zig-zag pattern, we move from low-order to high-order coefficients more uniformly. This creates longer strings of zeroes that can be efficiently encoded.

JPEG 2000

The **JPEG 2000** standard is compatible with JPEG but adds wavelet compression. Wavelets are a hierarchical waveform representation of the image that do not rely on blocks. Wavelets can be more computationally expensive but provide higher-quality compressed images.

video compression standards

There are two major families of video compression standards. The **MPEG** series of standards was developed primarily for broadcast applications. Broadcast system are asymmetric—putting more powerful and more expensive transmitters allows the receivers to be simpler and cheaper. The **H.26x** series is designed for symmetric applications like videoconferencing, in which both sides must encode and decode. The two groups have recently completed a joint standard, known as **Advanced Video Codec** (**AVC**) or **H.264**, that is designed to cover both types of applications.

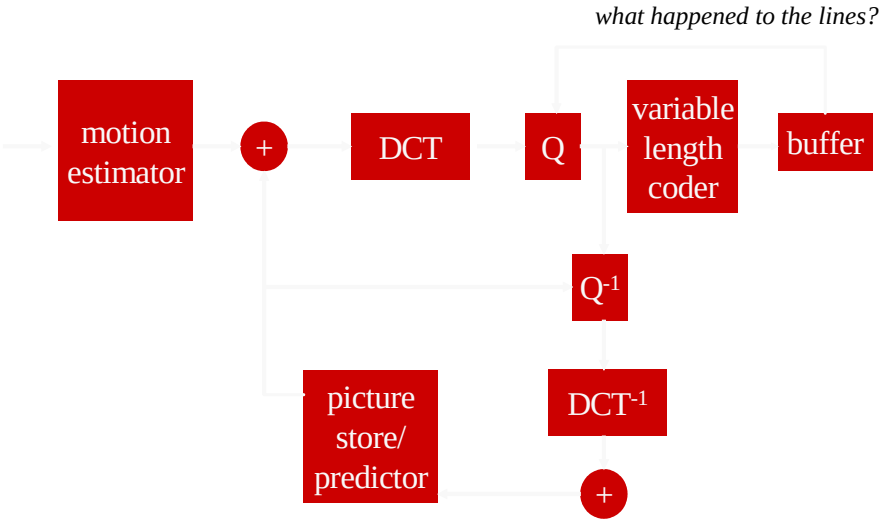


Figure 1-30

Figure 1-30 shows the block diagram of an MPEG-1 or MPEG-2 style encoder. (The MPEG-2 standard is the basis for digital television broadcast in the U. S.) The encoder makes use of the DCT and variable length coding. It adds **motion estimation** and **motion compensation** to encode the relationships between frames.

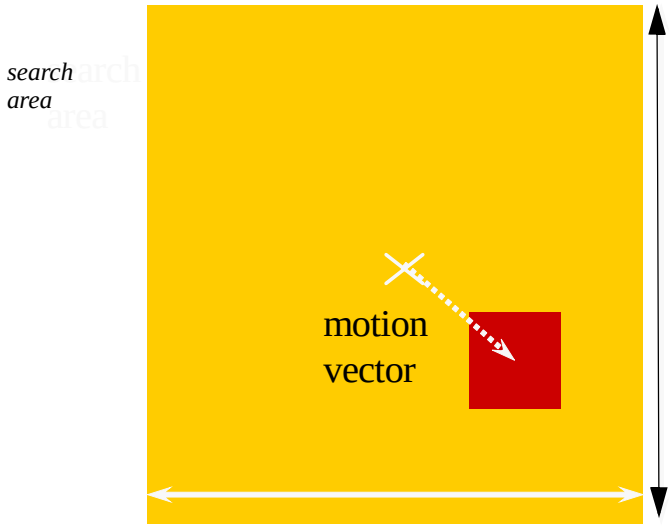


Figure 1-31

Motion estimation.

motion estimation

Motion estimation allows one frame to be encoded as translational motion from another frame. Motion estimation is performed on 16 x 16 **macroblocks**. A macroblock from one frame is selected and a search area in the **reference frame** is searched to find an identical or closely matching macroblock. Search reports an offset for the macroblock in the search area that provides the best match. That position describes a **motion vector** for the macroblock. During decompression, motion compensation copies the block to the position specified by the motion vector, thus saving the system from transmitting the entire image.

error signal

Motion estimation does not perfectly predict a frame because elements of the block may move, the search may not provide the exact match, etc. An error signal is also transmitted to correct for small imperfections in the signal. The inverse DCT and picture/store predictor in the feedback are used to generate the uncompressed version of the lossily compressed signal that would be seen by the receiver; that reconstruction is used to generate the error signal.

1.6 Summary

The designers of high-performance embedded computing systems are required to master several skill. They must be expert at system specification, not only in the informal sense but also in creating executable models. They must understand the basic architectural techniques for both hardware and software. They must be able to analyze performance and energy consumption for both hardware and software. And they must be able to make trade-offs between hardware and software at all stages in the design process.

During the remainder of this book, we will see many important design techniques that can be used in embedded system design methodologies. We will proceed largely bottom-up. We will first study hardware and software for uniprocessor systems. We will then consider hardware/software co-design. We will then move on to the multiprocessor techniques that underlie modern high-performance embedded systems.

What We Learned

- Many embedded computing systems are based upon standards. The form in which the standard is expressed affects the methodology we use to design the embedded system.
- Embedded systems are open to new types of security and reliability threats. Embedded computers that perform real-time control pose particular concerns.
- Many embedded computing systems are based upon standards. The form in which the standard is expressed affects the methodology we use to design the embedded system.
- Embedded systems are open to new types of security and reliability threats. Embedded computers that perform real-time control pose particular concerns.

Further Reading

The team of Lee and Sangiovanni-Vincentelli created the study of models of computation for embedded computing. Siewiorek and Swarz [Sie96] is the classical text on reliable computer system design. Storey [Sto96] provides a detailed description of safety-critical computers. Lee's book describes digital communication.

Questions

Q1-1 What are the essential characteristics of embedded computing systems?

Q1-2 What are the important characteristics of a software design methodology for embedded computing systems? A hardware design methodology? A complete hardware/software methodology?

Q1-3 What are the essential properties of a data flow graph?

Q1-4 What are the essential properties of a Petri net?

Lab Exercises

L1-1 Select a device of your choice and determine whether it uses embedded computers. Determine, to the extent possible, the internal hardware architecture of the device.

L1-2 How can you make an FSM and a data flow graph communicate in a reliable fashion?

L1-3 How much computation must be done to demodulate a CMDA2000 signal?

Chapter

2

CPUs

- Architectural mechanisms for embedded processors.
- Parallelism in embedded CPUs.
- Code compression and bus encoding.
- Security mechanisms.
- CPU simulation.

2.1 Introduction

CPUs are at the heart of embedded systems. Whether we use one CPU or combine several CPUs to build a multiprocessor, instruction set execution provides the combination of efficiency and generality that make embedded computing powerful.

A number of CPUs have been designed especially for embedded applications or adapted from other uses. We can also use design tools to create CPUs to match the characteristics of our application. In either case, a variety of mechanisms can be used to match the CPU characteristics to the job at hand. Some of these mechanisms are borrowed from general-purpose computing; others have been developed especially for embedded systems.

We will start with a brief introduction to the CPU design space. We will then look at the major categories of processors: RISC and DSP in Section 2.3; VLIW, superscalar, and related methods in Section 2.4. Section 2.5 considers novel variable-performance techniques such as better-than-worst-case design. In Section 2.6 we will study the design of memory hierarchies. Section 2.7 looks at additional CPU mechanisms like code compression and bus encoding. Section 2.8 surveys techniques for CPU simulation. Section 2.9 introduces some methodologies and techniques for the design of custom processors.

2.2 Comparing Processors

Choosing an CPU is one of the most important tasks faced by an embedded system designer. Fortunately, designers have a wide range of processors to choose from, allowing them to closely match the CPU to the problem requirements. They can even design their own CPU. In this section we will survey the range of processors and their evaluation before looking at CPUs in more detail.

2.2.1 Evaluating Processors

We can judge processors in several ways. Many of these are metrics. Some evaluation characteristics are harder to quantify.

performance

Performance is a key characteristic of processors. Different fields tend to use performance in different ways—for example, image processing tends to use performance to mean image quality. Computer system designers use performance to mean the rate at which programs execute.

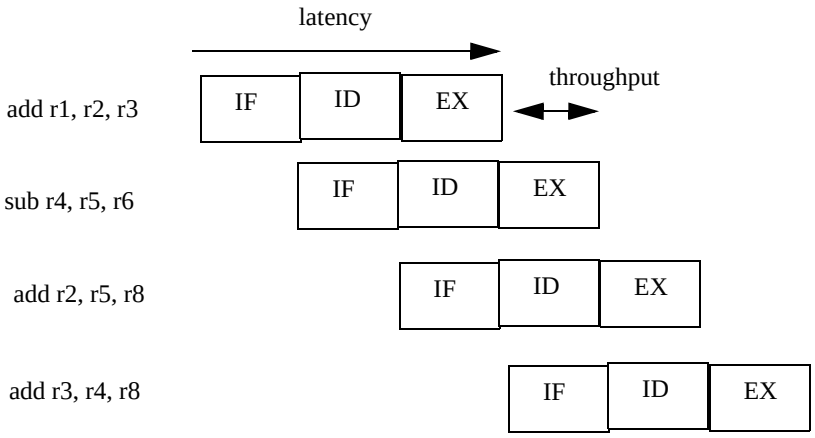


Figure 2-1

Latency and throughput in instruction execution.

We may look at computer performance more microscopically, in terms of a window of a few instructions, or macroscopically over large programs. In the microscopic view, we may consider either **latency** or **throughput**. Figure 2-1 is a simple **pipeline diagram** that shows the execution of several instructions. In the figure, latency refers to the time required to execute an instruction from start to finish, while throughput refers to the rate at which instructions are finished. Even if it takes several clock cycles to execute an instruction, the processor may still be able to finish one instruction per cycle.

At the program level, computer architects also speak of **average performance** or **peak performance**. Peak performance is often calculated assuming that instruction throughput proceeds at its maximum rate and all processor resources are fully utilized. There is no easy way to calculate average performance for most processors; it is generally measured by executing a set of benchmarks on sample data.

However, embedded system designers often talk of program performance in terms of **worst-case** (or occasionally **best-case**) **performance**. This is not simply a characteristic of the processor; it is determined for a particular program running on a given processor. As we will see in later chapters, it is generally determined by analysis because of the difficulty of determining an input set that should be used to cause the worst-case execution.

cost

Cost is another important measure of processors. In this case, we mean the purchase price of the processor. In VLSI design, cost is often measured in terms of the chip are required to implement a processor, which is closely related to chip cost.

energy/power

Energy and **power** are key characteristics of CPUs. In modern processors, energy and power consumption must be measured for a particular program and data for accurate results. Modern processors use a variety of techniques to manage energy consumption on-the-fly, meaning that simple models of energy consumption do not provide accurate results.

non-metric characteristics

There are other ways to evaluate processors that are harder to measure. **Predictability** is an important characteristic for embedded systems—when designing real-time systems we want to be able to predict execution time. Because predictability is affected by so many characteristics, ranging from the pipeline to the memory system, it is difficult to come up with a simple model for predictability.

Security is also an important characteristic of all processors, including embedded processors. Security is inherently unmeasurable since the fact that we do not know of a successful attack on a system does not mean that such an attack cannot exist.

2.2.2 A Taxonomy of Processors

We can classify processors in several dimensions. These dimensions interact somewhat but they help us to choose a processor type based upon our problem characteristics.

Flynn's categories

Flynn [Fly72] created a well-known taxonomy of processors. He classifies processors along two axes: the amount of data being processed and the number of instructions being executed. This gives several categories:

- **Single-instruction, single-data** (**SISD**). This is more commonly known today as a RISC processor. A single stream of instructions operates on a single set of data.

- **Single-instruction, multiple-data** (**SIMD**). Several processing elements each have their own data, such as registers. However, they all perform the same operations on their data in lockstep. A single program counter can be used to describe execution of all the processing elements.

- **Multiple-instruction, multiple-data** (**MIMD**). Several processing elements have their own data and their own program counters. The programs do not have to run in lockstep.

- **Multiple-instruction, single-data** (**MISD**). Few, if any commercial computers fit this category.

RISC vs. CISC

Instruction set style is one basic characteristic. The RISC/CISC divide is well-known. The origins of this dichotomy were related to performance—RISC processors were devised to make processors more easily pipelineable, increasing their throughput. However, instruction set style also has implications for code size, which can be important for cost and sometimes performance and power consumption as well (through cache utilization). CISC instruction sets tend to give smaller programs that RISC and tightly encoded instruction sets still exist on some processors that are destined for applications that need small object code.

single issue vs. multiple issue

Instruction issue width is an important aspect of processor performance. Processors that can issue more than one instruction per cycle generally execute programs faster. They do so at the cost of increased power consumption and higher cost.

static vs. dynamic scheduling

A closely related characteristic is how instructions are issued. **Static scheduling** of instructions is determined when the program is written. In contrast, **dynamic scheduling** determines what instructions are issued at run time. Dynamically scheduled instruction issue allows the processor to take data-dependent behavior into account when choosing how to issue instructions. **Superscalar** is a common technique for dynamic instruction issue. Dynamic scheduling generally requires a much more complex and costly processor than static scheduling.

vectors,. threads

Instruction issue width and scheduling mechanisms are only one way to provide parallelism. Many other mechanisms have been developed to provide new types of parallelism and concurrency. **Vector processing** uses instructions that perform on one- or two-dimensional arrays, generally performing operations common in linear algebra. **Multi-threading** is a fine-grained concurrency mechanism that allows the processor to quickly switch between several threads of execution.

2.2.3 Embedded vs. General-Purpose Processors

General-purpose processors are just that—they are designed to work well in a variety of contexts. Embedded processors must be flexible, but they can often be tuned to a particular application. As a result, some of the design precepts that are commonly followed in the design of general-purpose CPUs do not hold for embedded computers. And given the large number of embedded computers sold each year, many application areas make it worthwhile to spend the time to create a customized architecture. Not only are billions of 8-bit processors sold each year, but hundreds of millions of 32-bit processors are sold for embedded applications. Cell phones alone make the largest single application of 32-bit CPUs.

multicycle instructions

One tenet of RISC design is single-cycle instructions—an instruction spends one clock cycle in each pipeline stage. This ensures that other stages do not stall while waiting for an instruction to finish in one stage. However, the most fundamental goal of processor design is application performance. An increasing amount of evidence suggests that multicycle instructions can result in significantly higher program performance. Multicycle instructions are more useful in embedded processors than in general-purpose machines because embedded processors run a narrower mix of code, including computationally-intensive kernels. A specialized instruction can take fewer cycles than a sequence of general-purpose instructions because intermediate values do not need to be stored and function units can be tailored to the operations required. If the instruction is used frequently enough, it is worth the area and power consumption cost to add it to the instruction set.

instruction encoding

One of the consequences of the emphasis on pipelining in RISC is simplified instruction formats that are easy to decode in a single cycle. However, simple instruction formats result in increased code size. The Intel Architecture has a large number of CISC-style instructions with reduced numbers of operands and tight operation coding. Intel Architecture code is among the smallest code available when generated by a good compiler. Code size can affect performance—larger programs make less efficient use of the cache. We will discuss code compression in Section 2.7.1, which automatically generates tightly-coded instructions; several techniques have been developed to reduce the performance penalty associated with complex instruction decoding steps.

2.3 RISC Processors and Digital Signal Processors

In this section we will look at the workhorses of embedded computing, RISC and DSP. Our goal is not to exhaustively describe any particular embedded processor; that task is best left to data sheets and manuals. Instead, we will try to describe some important aspects of these processors, compare and contrast RISC and DSP approaches to CPU architecture, and consider the different emphases of general-purpose and embedded processors.

2.3.1 RISC Processors

Today, the term RISC is often used to mean single-issue processor. Term originally came from the comparison to complex instruction set (CISC) architectures. We will consider both aspects of the term.

pipeline design

A hallmark of RISC architecture is pipelining. General-purpose processors have evolved longer pipelines as clock speeds have increased. As the pipelines grow longer, the control required for their proper operation becomes ore complex. The pipelines of

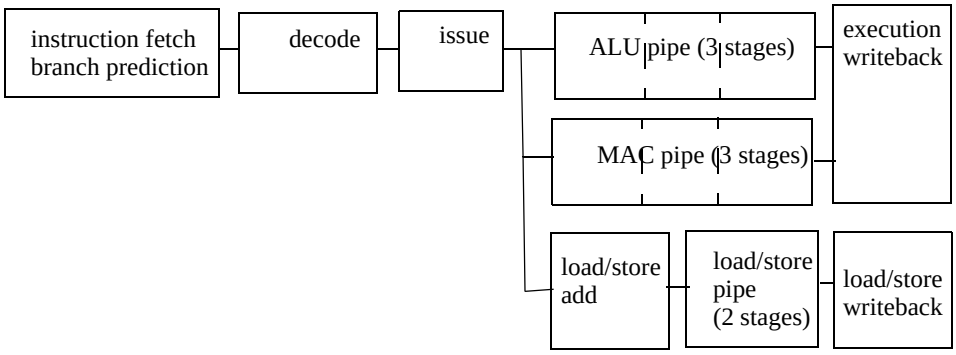


Figure 2-2

The ARM11 pipeline.

embedded processors have also grown considerably longer with more sophisticated control, as illustrated by the ARM family:

- The ARM7 uses a three-stage pipeline with fetch, decode, and execute stages. This pipeline requires only very simple control.
- The ARM9 uses a five-stage pipeline with fetch, decode, ALU, memory access, and register write stages. It does not perform branch prediction.
- The ARM11 has an eight-stage pipeline. It structure is shown in Figure 2-2. It performs dynamic branch prediction to make up for the six cycle penalty for a mispredicted branch. The pipeline has several independent completion stages; the pipeline control allows instructions to complete out-of-order.

2.3.2 Digital Signal Processors

Today, the term **digital signal processor** (DSP[1]) is often used as a marketing term. However, its original technical meaning still has some utility today. The AT&T DSP-16 was the first DSP. As illustrated in Figure 2-3, it introduced two features that define digital signal processors. First, it had an on-board multiplier and provided a multiply-accumulate instruction. At the time the DSP-16 was designed, silicon was still very expensive and the inclusion of a multiplier was a major architectural decision. The multiply-accu-

1.Unfortunately, the literature use DSP to mean both digital signal processor (a machine) and digital signal processing (a branch of mathematics).

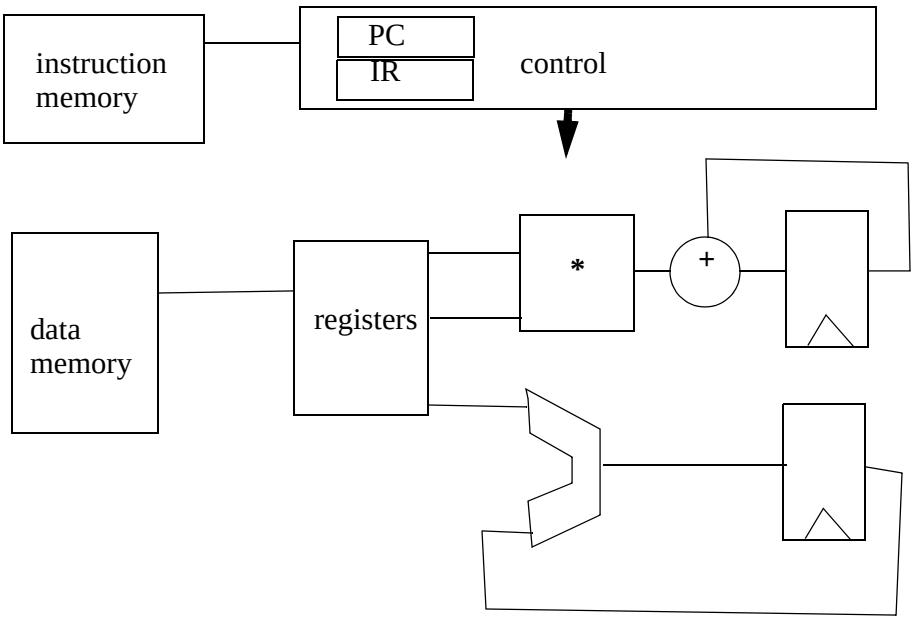


Figure 2-3

A digital signal processor with multiply-accumulate unit and Harvard architecture.

mulate instruction computes dest = src1*src2 + src3, a common operation in digital signal processing. Defining the multiply-accumulate instruction made the hardware somewhat more efficient because it eliminated a register, improved code density by combining two operations into a single instruction, and improves performance. The DSP-16 also used a Harvard architecture with separate data and instruction memories. The Harvard structure meant that data accesses could rely on consistent bandwidth from the memory, which is particularly important for sampled-data systems.

Some of the trends evident in RISC architectures also make their way into digital signal processors. For example, high-performance DSPs have very deep pipelines to support high clock rates. A major difference between modern processors used in digital signal processing vs. other applications is in the form of instructions. RISC processors generally have large, regular register files, which help simplify pipeline design as well as programming. Many DSPs, in contrast, have smaller general-purpose register files and many instructions that must use only one or a few selected registers. The accumulator is still a common feature of DSP architectures and other types of instructions may require the use of certain registers as sources or destinations for data.

The next example studies a family of high-performance DSPs.

Application Example 2-1

The Texas Instruments C5x DSP family

The C5x family [Tex01,Tex01B] is an architecture for high-performance signal processing. The C5x supports several features:

- A 40-bit arithmetic unit, which may be interpreted as 32-bit values plus 8 guard bits for improved rounding control. The ALU can also be split to perform two 16-bit operands.
- A barrel shifter performs arbitrary shifts for the ALU.
- A 17x17 multiplier and adder can perform multiply-accumulate operations.
- A comparison unit compares the high and low accumulator words to help accelerate Viterbi encoding/decoding.
- A single-cycle exponent encoder can be used for wide-dynamic-range arithmetic.
- Two dedicated address generators.

The C5x includes a variety of registers:

- Status registers include flags for arithmetic results, processor status, etc.
- Auxiliary registers are used to generate 16-bit addresses.
- A temporary register can hold a multiplicand or a shift count.
- A transition register is used for Viterbi operations.
- The stack pointer holds the top of the system stack.
- A circular buffer size register is used for circular buffers common in signal processing.
- Block-repeat registers help implement block-repeat instructions.
- Interrupt registers provide the interface to the interrupt system.

The C5x family defines a variety of addressing modes. Some of them include:

- ARn mode performs indirect addressing through the auxiliary registers.
- DP mode performs direct addressing from the DP register.
- K23 mode uses an absolute address.

- Bit instructions provide bit-mode addressing.

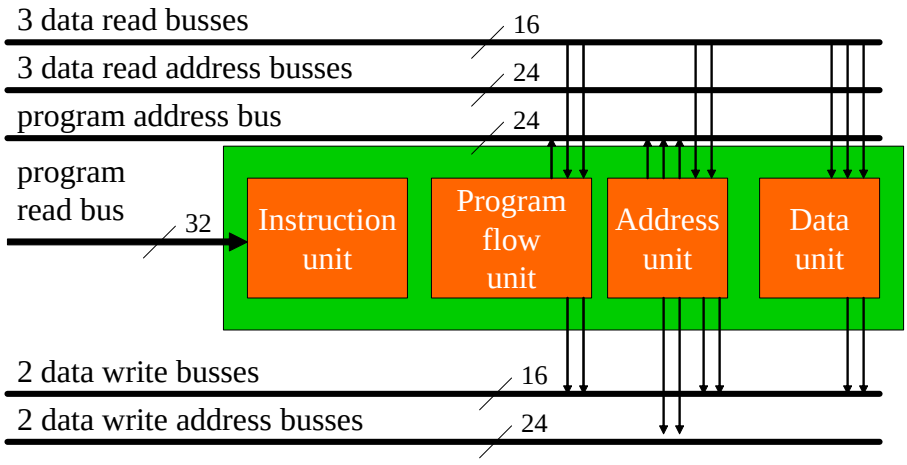
The RPT instruction provides single-instruction loops. The instruction provides a repeat count that determines the number of times the following instruction is executed. Special registers control the execution of the loop.

The C5x family includes several implementations. The C54x is a lower-performance implementation while the C55x is a higher-performance implementation.

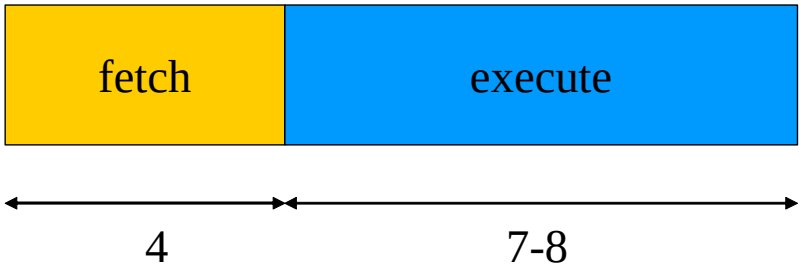
The C54x pipeline has six stages:

- Program prefetch sends the PC value on the program bus.
- Fetch loads the instruction.
- The decode stage decodes the instruction.
- The access step puts operand addresses on the busses.
- The read step gets the operand values from the bus.
- The execute step performs the operations.

The C55x microarchitecture includes three data read and two data write busses in addition to the program read bus:



The C55x pipeline is longer than that of the C54x and it has a more complex structure. It is divided into two stages:



The fetch stage takes four clock cycles; the execute stage takes seven or eight cycles.

During fetch, the prefetch 1 stage sends an address to memory, while prefetch 2 waits for the response. The fetch stage gets the instruction. Finally, the predecode stage sets up decoding.

During execution, the decode stage decodes a single instruction or instruction pair. The address stage performs address calculations. Data access stages sends data addresses to memory. The read cycle gets the data values from the bus. The execute stage performs operations and writes registers. Finally, the W and W+ stages write values to memory.

The C55x includes three computation units and fourteen operators. In general, the machine can execute two instructions per cycle. However, some combinations of operations are not legal due to resource constraints.

A **co-processor** is an execution unit that is controlled by the processor's execution unit. (In contrast, an **accelerator** is controlled by registers and is not assigned opcodes.) Co-processors are used in both RISC processors and DSPs, but DSPs show some particularly complex co-processors. Co-processors can be used to extend the instruction set to implement common signal processing operations. In some cases, the instructions provided by these co-processors can be integrated easily into other code. In other cases, the co-processor is designed to execute a particular stream of instructions and the DSP acts as a sequencer for a complex, multi-cycle operation.

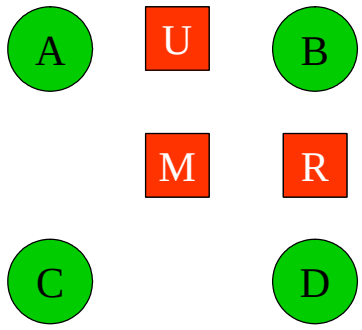
The next example looks at some co-processors for digital signal processing.

Application Example 2-2

TI C55x co-processor

The C55x provides three co-processors for use in image processing and video compression: one for pixel interpolation; one for motion estimation; and one for DCT/IDCT computation.

The pixel interpolation co-processor supports half-pixel computations that are often used in motion estimation. Given a set of four pixels A, B, C, and D, we want to compute the intermediate pixels U, M, and R:



Two instructions support this task. One loads pixels and computes:

ACy = copr(K8,AC,Lmem)

K8 is a set of control bits. The other loads pixels, computes, and stores:

ACy = copr(K8,ACx,Lmem) || Lmem=ACz

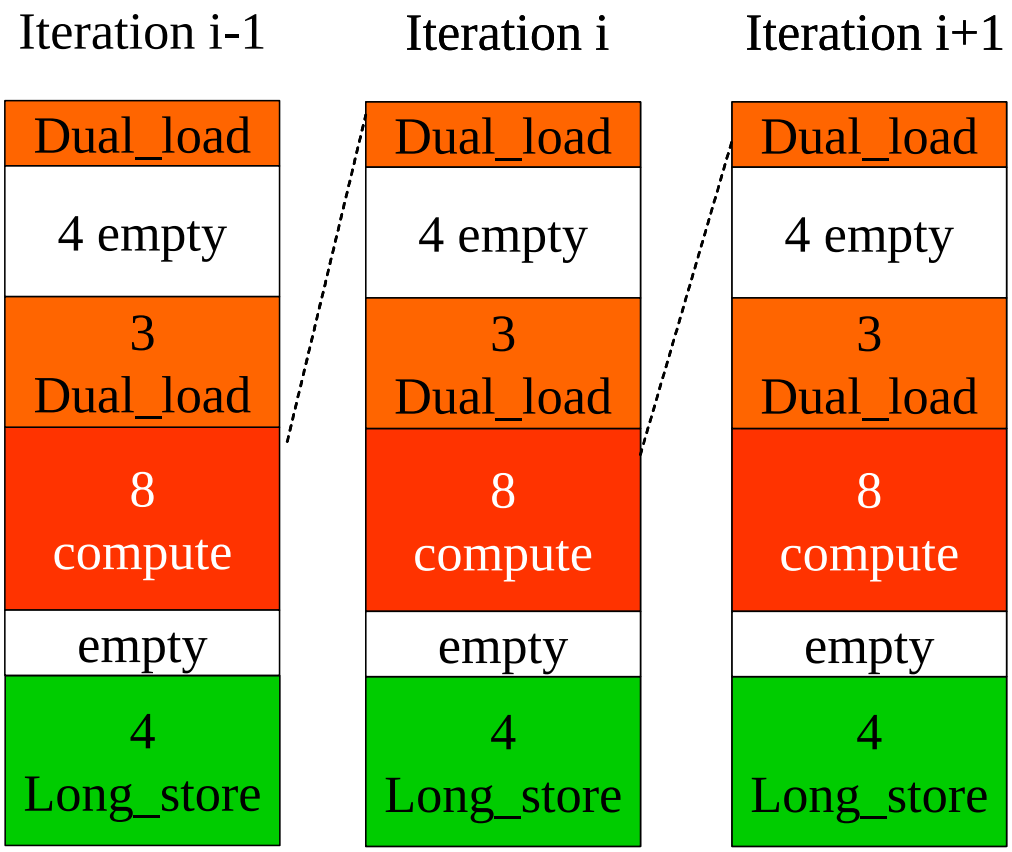
The motion estimation co-processor is built around a stylized usage pattern. It supports full search and three heuristic search algorithms: three-step, four-step, and four-step with half-pixel refinement. It can produce either one motion vector for a 16x16 macroblock or four motion vectors for four 8x8 blocks. The basic motion estimation instruction has the form:

[ACx,ACy] = copr(K8,ACx,ACy,Xmem,Ymem,Coeff)

where ACx and ACy are the accumulated sum-of-differences, K8 is a set of control bits, and Xmem and Ymem point to odd and even lines of the search window.

The DCT co-processor implements functions for one-dimensional DCT and IDCT computation. The unit is designed to support 8x8 DCT/IDCT and a particular sequence of instructions must be used to ensure that data operands are available at the required times. The co-processor provides three types of instructions: load, compute, and transfer to accumulators; compute, transfer and write to memory; and special.

Several iterations of the DCT/IDCT loop are pipelined in the co-processor when the proper sequence of instructions is used:



2.4 Parallel Execution Mechanisms

In this section we will look at various ways that processors perform operations in parallel. We will consider very long instruction word and superscalar processing, sub-word parallelism, vector processing, and thread level parallelism. We will end this section with a brief consideration of the available parallelism in some embedded applications.

2.4.1 Very Long Instruction Word Processors

Very long instruction word (VLIW) architectures were originally developed as general-purpose processors but have seen widespread use in embedded systems. VLIW architectures provide instruction-level parallelism with relatively low hardware overhead.

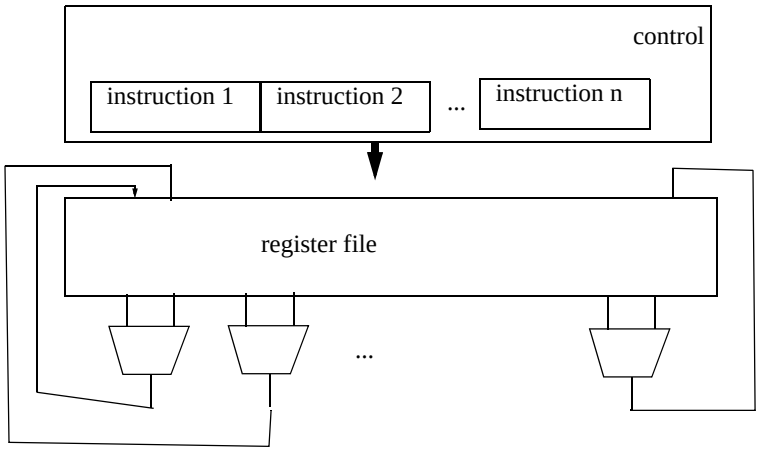


Figure 2-4

Structure of a generic VLIW processor.

VLIW basics

Figure 2-4 shows a simplified version of a VLIW processor to introduce the basic principles of the technique. The execution unit includes a pool of function units connected to a large register file. Using today's terminology for VLIW machines, the execution unit reads a **packet** of instructions—each instruction in the packet can control one of the function units in the machine. In an ideal VLIW machine, all instructions in the packet are executed simultaneously; in modern machines, it may take several cycles to retire all the instructions in the packet. Unlike a superscalar processor, the order of execution is determined by the structure of the code and how instructions are grouped into packets; the next packet will not begin execution until all the instructions in the current packet have finished.

Because the organization of instructions into packets determines the schedule of execution, VLIW machines rely on powerful compilers to identify parallelism and schedule instructions. The compiler is responsible for enforcing resource limitations and their associated scheduling policies. In compensation, the execution unit is simpler because it does not have to check for many resource interdependencies.

The ideal VLIW is relatively easy to program because of its large, uniform register file. The register file provides a communication mechanism between the function units since each function unit can read operands from and write results to any register in the register file.

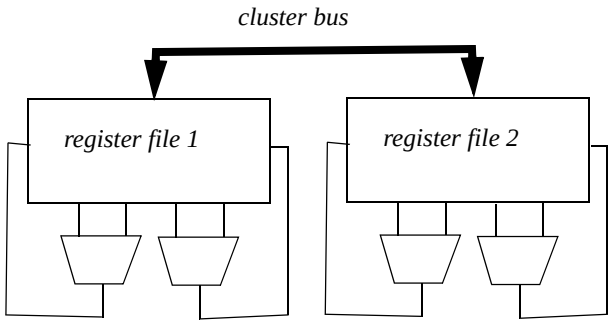


Figure 2-5

Split register files in a VLIW machine.

split register files

Unfortunately, it is difficult to build large, fast register files with many ports. As a result, many modern VLIW machines use partitioned register files as shown in Figure 2-5. In the example, the registers have been split into two register files, each of which is connected to two function units. The combination of a register file and its associated function units is sometimes called a **cluster**. A cluster bus can be used to move values between the register files. Register file-to-register file movement is performed under program control using explicit instructions. As a result, partitioned register files make the compiler's job more difficult. The compiler must partition values among the register files, determine when a value needs to be copied from one register file to another, generate the required move instructions, and adjust the schedules of the other operations to wait for the values to appear. However, the characteristics of VLIW circuits often require us to design partitioned register file architectures.

uses of VLIW

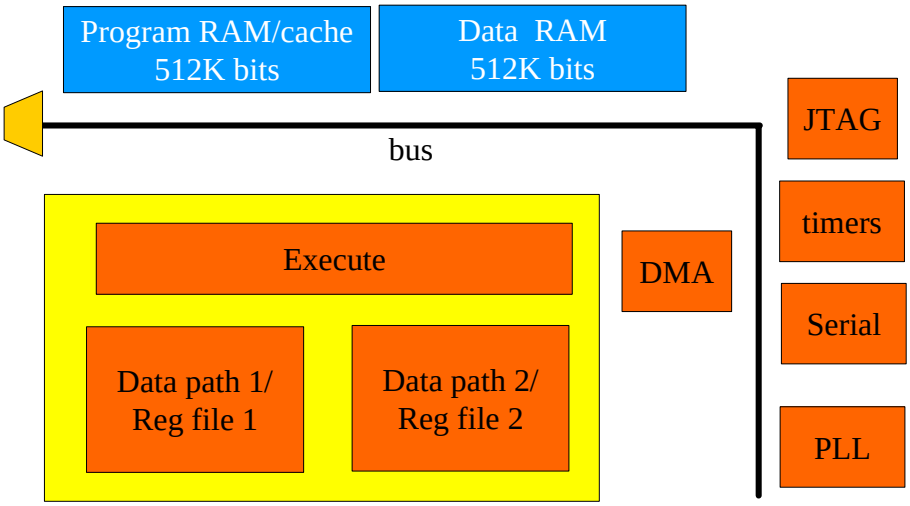
VLIW machines have been used in applications with a great deal of data parallelism. The Trimedia family of processors, for example, was designed for use in video systems. Video algorithms often perform similar operations on several pixels at time, making it relatively easy to generate parallel code. VLIW machines have also been used for signal processing and networking. Cell phone baseband systems, for example, must perform the same signal processing on many channels in parallel; the same instructions can be performed on separate data streams using VLIW architectures. Similarly, networking systems must perform the same or similar operations on several packets at the same time.

The next example describes a VLIW digital signal processor.

Application Example 2-3

Texas Instruments C6x VLIW DSP

The TI C6x is a VLIW processor designed for digital signal processing. Here is the block diagram of the C6x chip:



The chip includes on-board program and data RAM as well as standard devices and DMA. The processor core includes two clusters, each with the same configuration. Each register file holds sixteen words. Each datapath has eight function units:two load units, two store units, two data address units, and two register file cross paths.

The next example describes another VLIW machine.

Application Example 2-4

Freescale Starcore SC140 VLIW core

The Starcore architecture was jointly designed by Motorola (now Freescale Semiconductor) and Agere. The SC140 is an implementation of the Starcore architecture; the SC140 is a core that can be used in chip designs.

Like the C6x, the SC140 is organized into two clusters. But unlike the C6x, the two clusters in the SC140 perform different functions. One cluster is for data operations; it includes four data ALUs and a register file. The other cluster is for address operations; it includes two address operation units and its own register file.

The MC8126 is a chip that includes four SC140 cores along with shared memory.

2.4.2 Superscalar Processors

Superscalar processors issue more than one instruction per clock cycle. Unlike VLIW processors, they check for resource conflicts on-the-fly to determine what combinations of instructions can be issued at each step. Superscalar architectures dominate desktop and server architectures. However, relatively few embedded processors make use of superscalar techniques. Embedded computing architectures are more likely to be judged by metrics such as operations per watt rather than raw performance. Other techniques, such as VLIW or parallel processing, often provide more efficient architectures for embedded applications.

example superscalar embedded processors

The IBM PowerPC 440 is a two-issue, in-order superscalar processor. It has three pipelines: one for simple integer operations, one for multiply-accumulate operations, and one for loads and stores.

The embedded Pentium processor is a two-issue, in-order processor. It has two pipes: one for any integer operation and another for simple integer operations.

2.4.3 SIMD and Vector Processors

Many applications present data-level parallelism that lends itself to efficient computing structures. Furthermore, much of this data is relatively small, which allows us to build more parallel processing units to soak up more of that available parallelism.

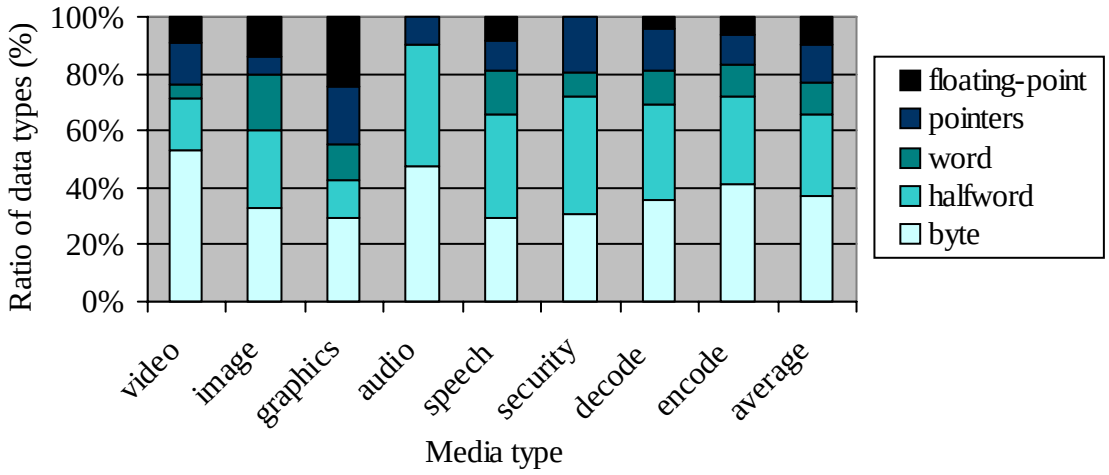


Figure 2-6

Operand sizes in MediaBench benchmarks [Fri00].

data operand sizes

A variety of studies have shown that many of the variables used in most programs have small dynamic ranges. Figure 2-6 shows the results of one such study by Fritts [Fri00]. He analyzed the data types of programs in the MediaBench benchmark suite.

The results show that 8-bit (byte) and 16-bit (halfword) operands dominate this suite of programs. If we match the function unit widths to the operand sizes, we can put more function units in the available silicon than if we simply used wide-word function units to perform all operations.

subword parallelism

One technique that exploits small operand sizes is **subword parallelism** [Lee94]. The processor's ALU can either operate in normal mode or it can be split into several smaller ALUs. An ALU can easily be split by breaking the carry chain so that bit slides operate independently. Each subword can operate on independent data; the operations are all controlled by the same opcode. Because the same instruction is performed on several data values, this technique is often referred to as a form of SIMD.

vectorization

Another technique for data parallelism is **vector processing**. Vector processors have been used in scientific computer for decades; they use specialized instructions that are designed to efficiently perform operations such as dot products on vectors of values. Vector processing does not rely on small data values, but vectors of smaller data types can perform more operations in parallel on available hardware, particularly when subword parallelism methods are used to manage datapath resources.

The next example describes a widely used vector processing architecture.

Application Example 2-5

Motorola AltiVec vector architecture

The AltiVec vector architecture [Ful98] was defined by Motorola (now Freescale Semiconductor) for the PowerPC architecture. AltiVec provides a 128-bit vector unit that can be divided into operands of several sizes: four operands of 32 bits, eight operands of 16 bits, or 16 operands of eight bits. A register file provides 32 128-bit vectors to the vector unit. The architecture defines a number of operations, including logical and arithmetic operands within an element as well as inter-element operations such as permutations.

2.4.4 Thread-Level Parallelism

Processors can also exploit thread- or task-level parallelism. It may be easier to find thread-level parallelism, particularly in embedded applications. The behavior of threads may be more predictable than instruction-level parallelism.

varieties of multithreading

Multithreading architectures must provide separate registers for each thread. But because switching between threads is stylized, the control required for multithreading is relatively straightforward. **Hardware multithreading** alternately fetches instructions from separate threads. On one cycle, it will fetch several instructions from one thread, fetching enough instructions to be able to keep the pipelines full in the absence of inter-

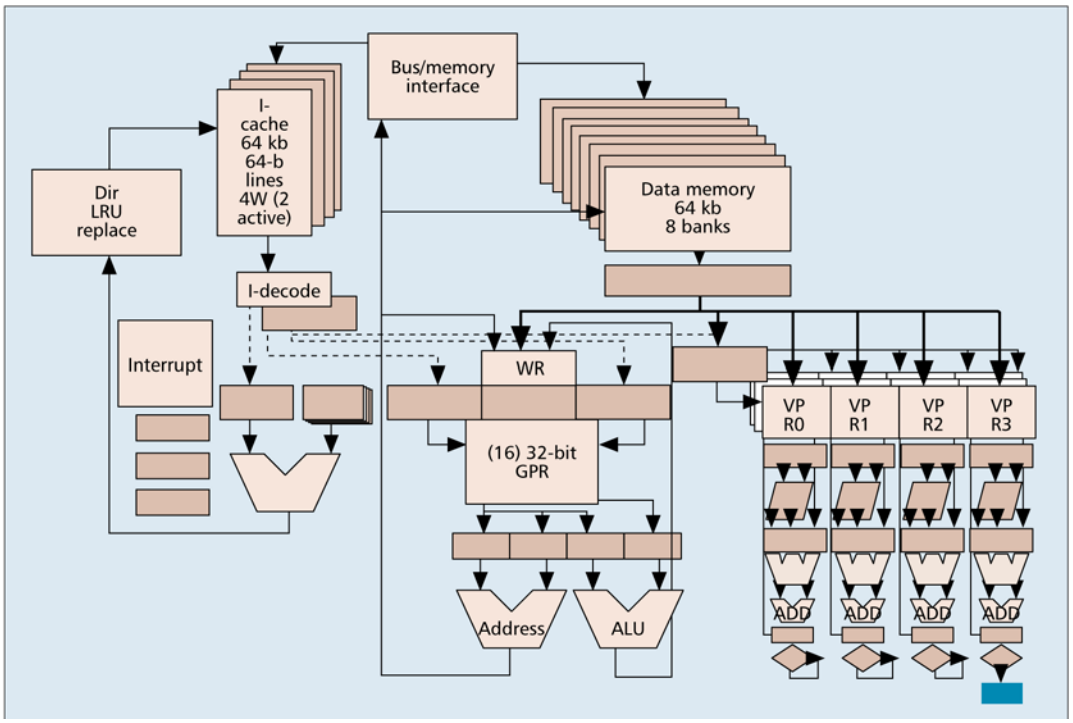
locks. On the next cycle, it would fetch instructions from another thread. **Simultaneous multithreading** (**SMT**) fetches instructions from several threads on each cycle rather than alternating between threads.

The next example describes a multithreading processor designed for cell phones.

Application Example 2-6

Sandbridge Sandstorm multi threaded CPU

The Sandstorm processor [Glo03,Glo05] is designed for mobile communications. It processes four threads:

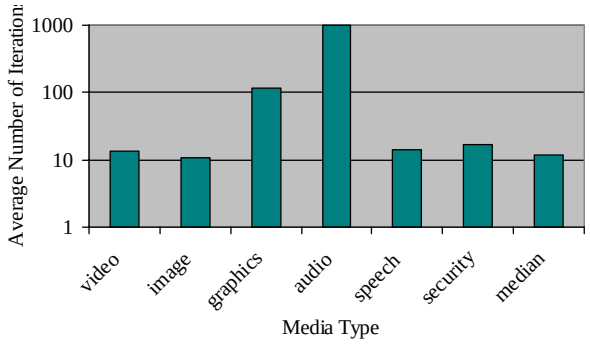


2.4.5 Processor Resource Utilization

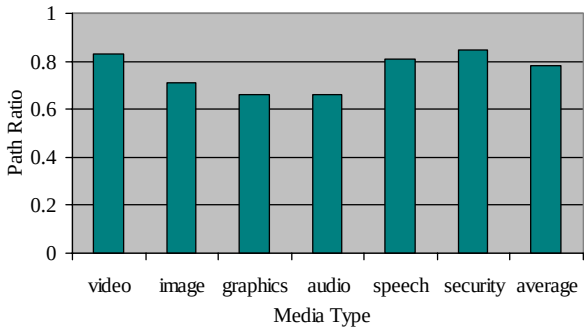
The choice of processor architecture depends in part on the characteristics of the programs to be run on the processor. In many embedded applications we can leverage our knowledge of the core algorithms to choose effective CPU architectures. However, we must be careful to understand the characteristics of those applications. As an example, many researchers assume that multimedia algorithms exhibit embarrassing levels of parallelism. Experiments show that this is not necessarily the case.

measurements on multimedia benchmarks

Tallu et al. [Tal03] evaluated the instruction-level parallelism available in multimedia applications. As shown in Figure 2-7, they evaluated several different processor configurations using SimpleScalar. They measured nine benchmark programs on the various architectures. The bar graphs show the instructions per cycle for each application; most applications exhibit fewer than four instructions per cycle.



number of iterations per loop



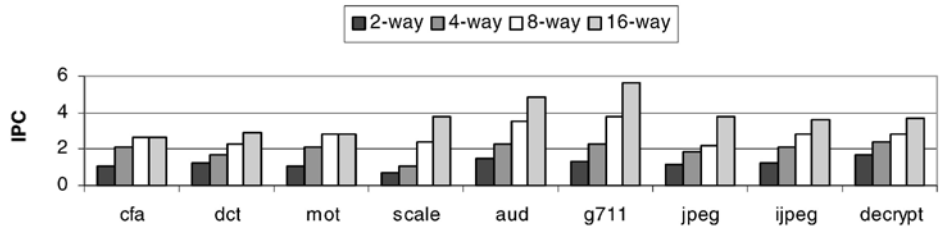
path ratio

Figure 2-8

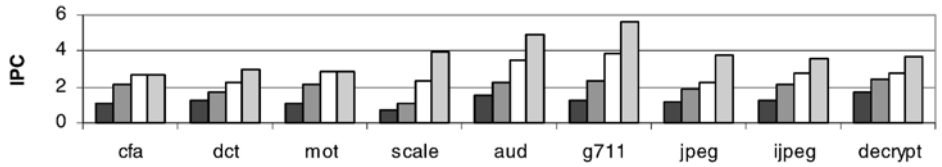
Dynamic behavior of loops in MediaBench [Fri00].

Parameters	2-way	4-way	8-way	16-way
Fetch width, Decode width, Issue width, and Commit width	2	4	8	16
RUU Size	32	64	128	256
Load Store Queue	16	32	64	128
Integer ALUs (latency/recovery = 1/1)	2	4	8	16
Integer Multipliers (latency/recovery = 3/1)	1	2	4	8
Load/Store ports (latency/recovery = 1/1)	2	4	8	16
L1 I-cache (size in KB, hit time, associativity, block size in bytes)	16, 1, 1, 32	16, 1, 1, 32	16, 1, 1, 32	32, 1, 1, 64
L1 D-cache (size in KB, hit time, associativity, block size in bytes)	16, 1, 4, 32	16, 1, 4, 32	16, 1, 4, 32	16, 1, 4, 32
L2 unified cache (size in KB, hit time, associativity, block size)	256, 6, 4, 64	256, 6, 4, 64	256, 6, 4, 64	256, 6, 4, 64
Main memory width	64 bits	128 bits	256 bits	256 bits
Main memory latency (first chunk, next chunk	65, 4	65, 4	65, 4	65,4
Branch Predictor – bimodal (size, BTB size)	2K, 2K	2K, 2K	2K, 2K	2K, 2K
SIMD ALUs	**2**	**4**	**8**	**16**
SIMD Multipliers	**1**	**2**	**4**	**8**

processor configurations



(a)



(b)

	cfa	dct	mot	scale	aud	g711	jpeg	ijpeg	decrypt
4-way	< 1 %	< 1 %	< 1 %	< 2 %	< 4 %	< 1 %	< 1 %	< 1 %	< 1 %
8-way	< 1 %	< 1 %	< 1 %	< 3 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %
16-way	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %	< 1 %

(c)

results

Figure 2-7

An evaluation of the available parallelism in multimedia applications [Tal03].

Fritts [Fri00] studied the characteristics of loops in the MediaBench suite. Figure 2-8 shows two measurements; in each case, results are shown with the benchmark programs grouped into categories based on their primary function. The first measurement shows the average number of iterations of a loop; fortunately, loops on average are executed many times. The second measurement shows **path ratio**, which is defined as

$$\frac{\text{number of loop body instructions executed}}{\text{total number of instructions in loop body}} \times 100 \qquad \text{(EQ 2-1)}$$

Path ratio measures the percentage of a loop's instructions that are actually executed. The average path ratio over all the MediaBench benchmarks was 78%, which means that 22% of the loop instructions were not executed.

multimedia algorithms

These results should not be surprising given the nature of modern embedded algorithms. Modern signal processing algorithms have moved well beyond filtering. Many algorithms use control to improve performance. The large specifications for multimedia standards will naturally result in complex programs.

implications for CPUs

To take advantage of the available parallelism in multimedia and other embedded applications, we need to match the processor architecture to the application characteristics. These experiments suggest that processor architectures must exploit parallelism at several levels of abstraction.

2.5 Variable-Performance CPU Architectures

Because so many embedded systems must meet real-time deadlines, predictable execution time is a critical feature of the components used in embedded systems. However, traditional computer architecture designs have emphasized average performance over worst-case performance, producing processors that are fast on average but whose worst-case performance is hard to bound. This often leads to conservative designs of both hardware (oversized caches, faster processors) and software (simplified coding and restricted use of instructions).

As both power consumption and reliability become even more important, new techniques have been developed that make processor behavior even more complex. Those techniques are finding their way into embedded processors even though they make designs harder to analyze. In this section we will survey two important developments, dynamic voltage and frequency scaling and better-than-worst-case design. We will explore the implications of these features and how to use them to our advantage in later chapters.

2.5.1 Dynamic Voltage and Frequency Scaling

DVFS

Dynamic voltage and frequency scaling (DVFS) [XXX] is a popular technique for controlling CPU power consumption that takes advantage of the wide operating range of CMOS digital circuits.

CMOS circuit characteristics

Unlike many other digital circuit families, CMOS circuits can operate at a wide range of voltages [Wol02]. Furthermore, CMOS circuits operate more efficiently at lower voltages. The delay of a CMOS gate is a linear function of power supply voltage:

$$t_p = 2.2RC, \quad \text{(EQ 2-2)}$$

where the effective resistance of the MOS transistor is largely a linear function of power supply voltage. The energy consumed during operation of the gate is proportional to the square of the operating voltage:

$$E = CV^2. \quad \text{(EQ 2-3)}$$

The speed-power product for CMOS (ignoring leakage) is also CV^2. Therefore, by lowering the power supply voltage, we can reduce energy consumption by V^2 while reducing performance by only V.

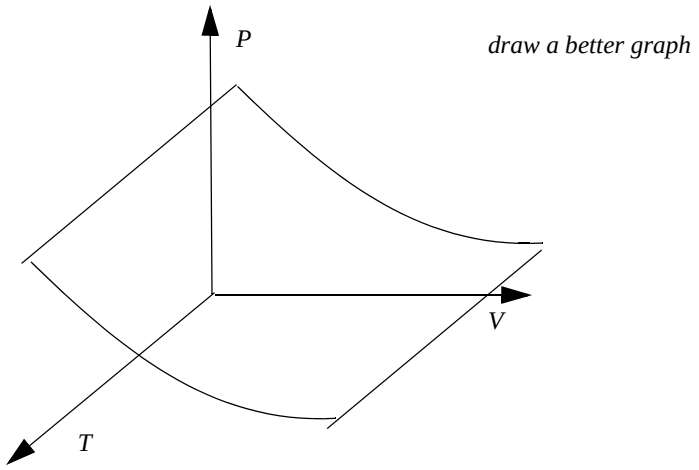


Figure 2-9

The voltage/speed/power operating space.

Because we can operate CMOS logic at many different points, a CPU can be operated across a design space. Figure 2-9 illustrates the relationship between power supply voltage (*V*), operating speed (*T*), and power (*P*).

DVFS architecture

An architecture for dynamic voltage and frequency scaling operates the CPU within this space under a control algorithm. Figure 2-10 shows the architecture of a DVFS architecture. The clock and power supply are generated by circuits that can supply a range of values; these circuits generally operate at discrete points rather than continuously-varying values. Both the clock generator and voltage generator are operated by a controller that determines when the clock frequency and voltage will change and by how much.

DVFS control strategy

A DVFS controller must operate under constraints in order to optimize a design metric. The constraints are related to clock speed and power supply voltage: not only their minimum and maximum values, but how quickly clock speed or power supply voltage

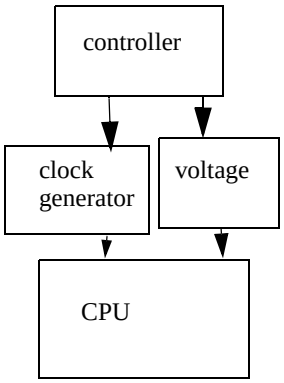


Figure 2-10

Dynamic voltage and frequency scaling (DVFS) architecture.

can be changed. The design metric may be either to maximize performance given a energy budget or to minimize energy given a performance bound.

While it is possible to encode the control algorithm in hardware, the control method is generally set at least in part by software. Registers may set the value of certain parameters. More generally, the complete control algorithm may be implemented in software.

The next example describes voltage scaling features in a modern embedded processor.

Application Example 2-7

Dynamic Voltage and Frequency Scaling in the Intel XScale

The Intel XScale [Int00] is compliant with ARM version 5TE. Operating voltage and clock frequency can be controlled by setting bits CP 14, registers 6 and 7, respectively. Software can use this programming model interface to implement a selected DVFS policy.

2.5.2 Better-Than-Worst-Case Design

Digital systems are traditionally designed as synchronous systems governed by clocks. The clock period is determined by careful analysis so that values are stored into registers properly, with the clock period extended to cover the worst-case delay. In fact, the worst-case delay is relatively rare in many circuits and the logic sits idle for some period most of the time.

Better-than-worst-case design [XXX] is an alternative design style in which logic detects and recovers from errors, allowing the circuit to run most of the time at a higher speed.

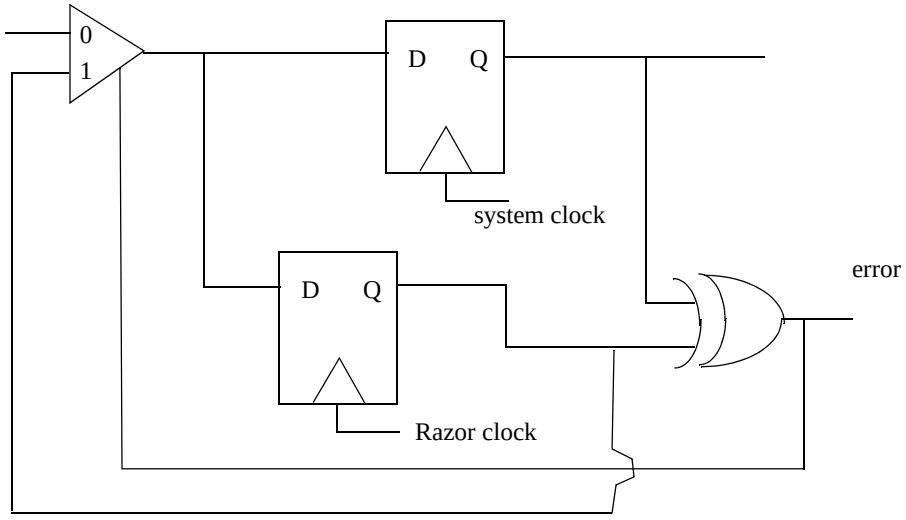


Figure 2-11

A Razor latch.

Razor microarchitecture

The Razor architecture [Ern03] is one architecture for better-than-worst-case performance. Razor uses a specialized register, shown in Figure 2-11, measures and evaluates errors. The system register holds the latched value and is clocked at the higher-than-worst-case clock rate. A separate register is clocked separately and slightly behind the system register. If the results stored in the two register are different, then an error occurred, probably due to timing. The XOR gate measures that error and causes the later value to replace the value in the system register.

The Razor microarchitecture does not cause an erroneous operation to be recalculated in the same stage. It rather forwards the operation to a later stage. This avoids having a stage with a systematic problem stall the pipeline with an indefinite number of recalculations.

2.6 Processor Memory Hierarchy

The memory hierarchy is a critical determinant of overall system performance and power consumption. In this section we will review some basic concepts in the design of memory hierarchies and how they can be exploited in the design of embedded proces-

sors. We will start by introducing a basic model of memory components that we can use to evaluate various hardware and software design strategies. We will then consider the design of register files and caches. We will end with a discussion of scratch pad memories, which have been proposed as adjuncts to caches in embedded processors.

2.6.1 Memory Component Models

In order to evaluate some memory design methods, we need models for the physical properties of memories: area, delay, and energy consumption. Because a variety of structures at different levels of the memory hierarchy are built from the same components, we can use a single model throughout the memory hierarchy and for different types of memory circuits.

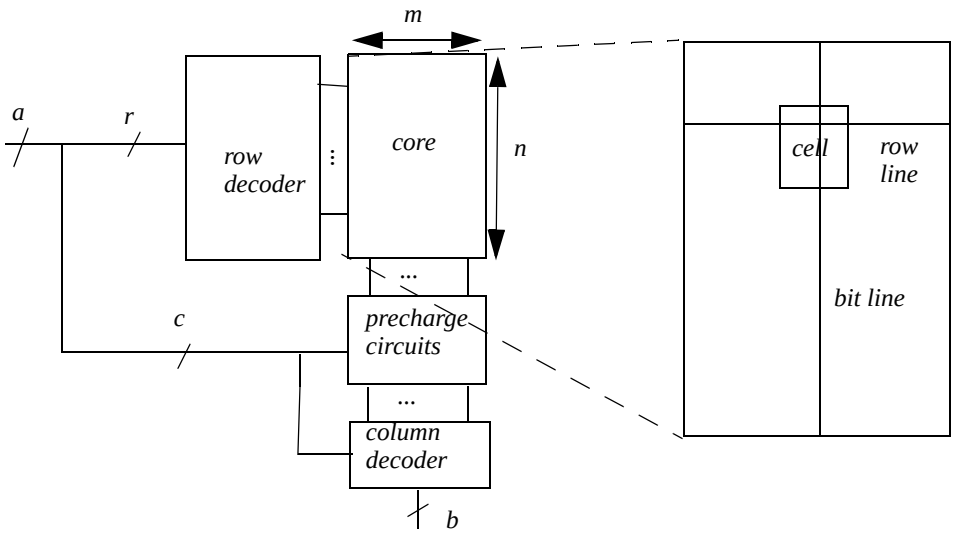


Figure 2-12

Structural model of a memory block.

memory block structure

Figure 2-12 shows a generic structural model for a two-dimensional memory block. This model does not depend on the details of the memory circuit and so applies to various types of dynamic RAM, static RAM, and read-only memory. The basic unit of storage is the **memory cell**. Cells are arranged in a two-dimensional array. This memory model describes the relationships between the cells and their associated access circuitry.

Within the memory core, cells are connected to row and bit lines that provide a two-dimensional addressing structure. The row line selects a one-dimensional row of cells, which then can be accessed (read or written) via their bit lines. When a row is selected, all the cells in that row are active. In general, there may be more than one bit line, since many memory circuits use both the true and complement forms of the bit.

The row decoder circuitry is a demultiplexer that drives one of the *n* row lines in the core by decoding the *r* bits of row address. A column decoder selects a *b*-bit wide subset of the bit lines based upon the *c* bits of column address. Some memories also require precharge circuits to control the bit lines.

area model

The area model of the memory block has components for the elements of the block model:

$$A = A_r + A_x + A_p + A_c \quad \text{(EQ 2-4)}$$

The row decoder area is

$$A_r = a_r n, \quad \text{(EQ 2-5)}$$

where a_r is the area of a one-bit slice of the row decoder.

The core area is

$$A_x = a_x mn, \quad \text{(EQ 2-6)}$$

where a_x is the area of a one-bit core cell, including its share of the row and bit lines.

The precharge circuit area is

$$A_p = a_p n, \quad \text{(EQ 2-7)}$$

where a_p is the area of a one-bit slice of the precharge circuit.

The column decoder area is

$$A_c = a_c n, \quad \text{(EQ 2-8)}$$

where a_c is the area of a one-bit slice of the column decoder.

delay model

The delay model of the memory block follows the flow of information in a memory access. Some of its elements are independent of *m* and *n* while others depend on the length of the row or column lines in the cell:

$$\Delta = \Delta_{setup} + \Delta_r + \Delta_x + \Delta_{bit} + \Delta_c . \quad \text{(EQ 2-9)}$$

Δ_{setup} is the time required for the precharge circuitry. It is generally independent of the number of columns but may depend on the number of rows due to the time required to precharge the bit line. Δ is the row decoder time, including the row line propagation time. The delay through the decoding logic generally depends upon the value of *m* but the dependence may vary due to the type of decoding circuit used. Δ_x is the reaction time of the core cell itself. Δ_{bit} is the time required for the values to propagate through the bit line. Δ_c is the delay through the column decoder, which once again may depend on the value of *n*.

energy model

The energy model must include both static and dynamic components. The dynamic component follows the structure of the block to determine the total energy consumption for a memory access:

$$E_D = E_r + E_x + E_p + E_c, \quad \text{(EQ 2-10)}$$

given the energy consumptions of the row decoder, core, precharge circuits, and column decoder. The core energy depends on the values of *m* and *n* due to the row and bit lines. The decoder circuitry energy also depends on *m* and *n*, though the details of that relationships depend on the circuits used.

The static component E_S models the standby energy consumption of the memory. The details vary for different types of memories but the static component can be significant.

The total energy consumption is

$$E = E_D + E_S. \quad \text{(EQ 2-11)}$$

multi-ported memories

This model describes a single-port memory in which a single read or write can be performed at any given time. A **multi-port memory** accepts multiple addresses/data for simultaneous accesses. Some aspects of the memory block model extend easily to multi-port memories. However, delay for a multi-port memory is a non-linear function of the number of ports. The exact relationship depends on the detail of the core circuit design, but the memory cell core circuits introduce non-linear delay as ports are added to the cell.

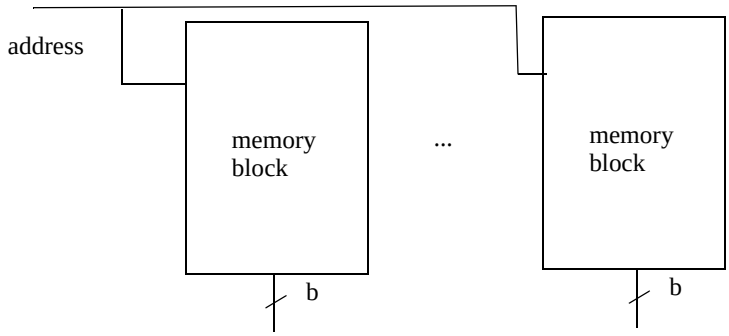


Figure 2-13

A memory array built from memory blocks.

busses

We may also want to model the bus that connects the memory to the remainder of the system. Busses present large capacitive loads that introduce significant delay and energy penalties.

memory arrays

Larger memory structures can be built from memory blocks. Figure 2-13 shows a simple wide memory in which several blocks are accessed in parallel from the same address lines. A set-associative cache could be constructed from this array, for example, by a multiplexer that selects the data from the block that corresponds to the appropriate set. Parallel memories may be built by feeding separate addresses to different memory blocks.

2.6.2 Register Files

The register file is the first stage of the memory hierarchy. Although the size of the register file is fixed when the CPU is pre-designed, if we design our own CPUs then we can select the number of registers based upon the application requirements. Register file size is a key parameter in CPU design that affects code performance and energy consumption as well as the area of the CPU.

sweet spot in register file design

Register files that are either too large or too small relative to the application's needs incur extra costs. If the register file is too small, the program must **spill** values to main memory: the value is written to main memory and later read back from main memory. Spills cost both time and energy because main memory accesses are slower and more energy-intensive than register file accesses. If the register file is too large, then it consumes static energy as well as taking extra chip area that could be used for other purposes.

register file parameters

The most important parameters in register file design are number of words and number of ports. Word width affects register file area and energy consumption but is not closely coupled to other design decisions. The number of words more directly determines area, energy, and performance. The number of ports is important because, as noted before, delay is a non-linear function of the number of ports. This non-linear dependency is the key reason that many VLIW machines use partitioned register files.

Wehmeyer et al. [Weh01] studied the effects of varying register file size on program dynamic behavior. They compiled a number of benchmark programs and used profiling tools to analyze the program's behavior. Figure 2-14 shows performance and energy consumption as a function of register file size. In both cases, overly small register files result in non-linear penalties whereas large register files present little benefit.

2.6.3 Caches

Cache design has received a lot of attention in general-purpose computer design. Most of those lessons apply to embedded computers as well, but because we may design

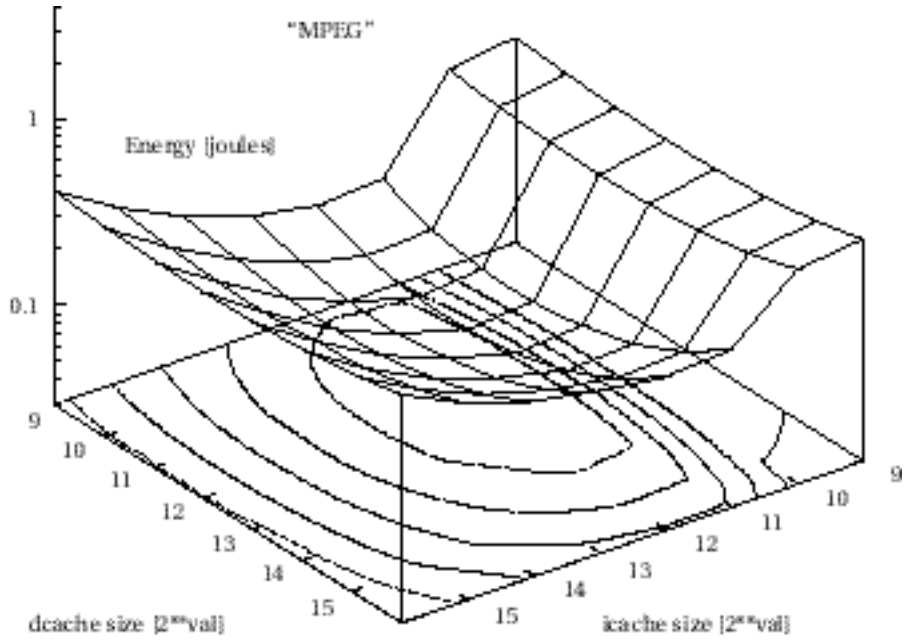


Figure 2-15 *Energy consumption versus instruction/data cache size for a benchmark program [Li98].*

the CPU to meet the needs of a particular set of applications, we can pay extra attention to the relationship between the cache configuration and the programs that will use it.

sweet spot in cache design

As with register files, caches have a sweet spot that is neither too small nor too large. Li and Henkel [Li98] measured the influence of caches on energy consumption in detail. Figure 2-15 shows the energy consumption of a CPU running an MPEG encoder. Energy consumption has a global minimum: too-small caches result in excessive main memory accesses; too-large caches consume excess static power.

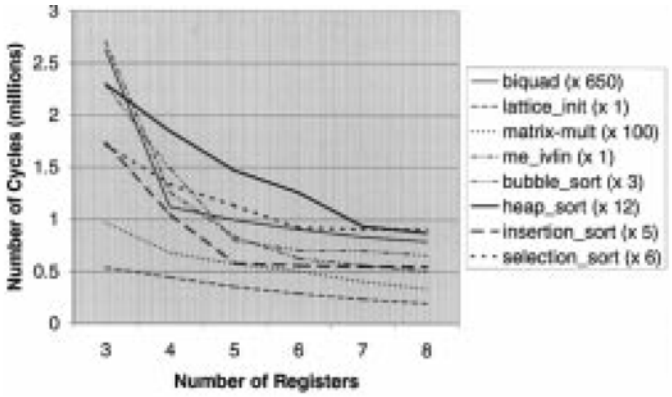
cache parameters and behavior

The most basic cache parameter is total cache size. Larger caches can hold more data or instructions at the cost of increased area and static power consumption. Given a fixed number of bits in the cache, we can vary both the set associativity and the line size. Splitting a cache into more sets allows us to independently reference more locations that map onto similar cache locations at the cost of mapping more memory addresses into a given cache line. Longer cache lines provide more prefetching bandwidth, which is useful in some algorithms but not others.

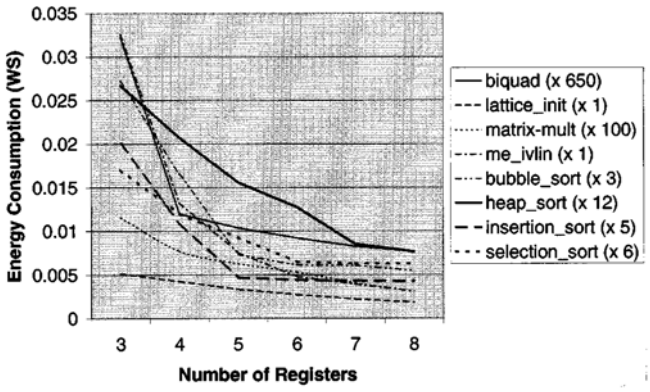
cache parameter selection

effects of cache parameters: line size, associativity, etc.

Panda et al. [Pan99] developed an algorithm to explore the memory hierarchy design space and to allocate program variables within the memory hierarchy. They allocated frequently-used scalar variables to the register file. They used the classification of Wolfe and Lam [Wol91] to analyze the behavior of arrays:



performance vs. number of registers.



energy consumption vs. number of registers

Figure 2-14

Performance and energy consumption as a function of register file size [Weh01].

- **self-temporal** reuse means that the same array element is accessed in different loop iterations;
- **self-spatial** reuse means that the same cache line is accessed in different loop iterations;
- **group-temporal** reuse means that different parts of the program access the same array element;

- **group-spatial** reuse means that different parts of the program access the same cache line.

This classification treats temporal reuse (the same data element) as a special case of spatial reuse (the same cache line). They divide memory references into equivalence classes, with each class containing a set of references with self-spatial and group-spatial reuse. The equivalence classes allow them to estimate the number of cache misses required by those references. They assume that spatial locality can result in reuse if the number of memory references in the loop is less than the cache size. Group-spatial locality is possible when a row fits into a cache and the other data elements used in the loop are smaller than the cache size. Two sets of accesses are compatible if their index expressions differ by a constant.

Gordon-Ross et al. {Gor04] developed a method to optimize multi-level cache hierarchies. They adjusted cache size, then line size, then associativity. They found that the configuration of the first-level cache affects the required configuration for the second-level cache—different first-level configurations cause different elements to miss the first-level cache, causing different behavior in the second-level cache. To take this effect into account, the alternately chose cache size for each level, then line size for each level, and finally associativity for each level.

configurable caches

Several groups, such as Balasubramonian et al [Bal03], have proposed **configurable caches** whose configuration can be changed at run time. Additional multiplexers and other logic allow a pool of memory cells to be used in several different cache configurations. Registers hold the configuration values that control the configuration logic. The cache has a configuration mode in which the cache parameters can be set; the cache acts normally in operation mode between configurations. The configuration logic incurs an area penalty as well as static and dynamic power consumption penalties. The configuration logic also increases the delay through the cache. However, it allows the cache configuration to be adjusted for different parts of the program in fairly small increments of time.

2.6.4 Scratch Pad Memories

A cache is designed to move a relatively small amount of memory close to the processor. Caches use hardwired algorithms to manage the cache contents—hardware determines when values are added or removed from the cache. Software-oriented schemes are an alternative way to manage close-in memory.

scratch pads

As shown in Figure 2-16, a **scratch pad** memory [Pan00] is located parallel to the cache. However, the scratch pad does not include hardware to manage its contents. The CPU can address the scratch pad to read and write it directly. The scratch pad appears in a fixed part of the processor's address space, such as the lower range of addresses. The size of the scratch pad is chosen to fit on-chip and provide a high-speed memory. Because the scratch pad is a part of memory, its access time is predictable, unlike accesses to a cache. Predictability is the key attribute of a scratch pad.

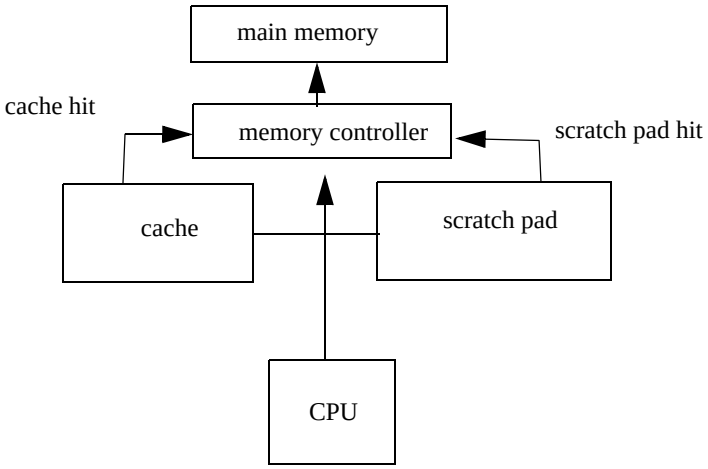


Figure 2-16

A scratch pad memory in a system.

Because the scratch pad is part of the main memory space, standard read and write instructions can be used to manage the scratch pad. Management requires determining what data is in the scratch pad and when it is removed from the cache. Software can manage the cache using a combination of compile-time and run-time decision making.

scratch pad management

Panda et al. developed methods for managing scratch pad contents by allocating variables to the scratch pad. They determined that static variables were best managed statically, with allocation decisions made at compile time. They chose to map all scalars to the scratch pad in order to avoid creating cache conflicts between arrays and scalars.

Arrays may be mapped either into the cache or the scratch pad. If two arrays have non-intersecting lifetimes, then their relative allocation is unimportant. Arrays with intersecting lifetimes may conflict; which are mapped into the cache versus the scratch pad demands some analysis. Panda et al. define several metrics to analyze conflicts:

- *VAC(u)*, the **variable access count** of variable *u*, counts the number of times that *u* is accessed during its lifetime.
- *IAC(u)*, the i**nterference access count** of variable *u*, counts the number of times other variables ($v \neq u$) are accessed during the lifetime of *u*.
- *IF(u)*, the **interference factor** of *u*, is defined as

$$IF(u) = VAC(u) + IAC(u). \quad \text{(EQ 2-12)}$$

Variables with a high *IF* value are likely to be interfered with in the cache and are good candidates for promotion to the scratch pad.

They then use these metrics to define a loop-oriented conflict metric, known as the **loop conflict factor** or *LCF*:

$$LCF(u) = \sum_{1 \le i \le p} \left[k(u) + \left(\sum_{v} k(v) \right) \right]. \quad \text{(EQ 2-13)}$$

In this formula, the outer summation is over all p loops in which u is accessed and the inner summation is over all *variables* $v \neq u$ accessed in the i^{th} loop.

The **total conflict factor** TCF for an array u is

$$TCF(u) = LCF(u) + IF(u). \quad \text{(EQ 2-14)}$$

- data partitioning formulation

scratch pad allocation algorithms

These metrics are used by algorithms to allocate variables between the scratch pad and main memory/cache. We can formulate the problem as follows:

- Given a set of arrays, each with a TCF value and a size $\{A_1, TCF(A_1), S_1\}, \ldots, \{A_n, TCF(A_n), S_n\}$
- and an SRAM of size S,
- find an optimal subset of arrays Q such that $S \geq \sum_{i \in Q} S_i$ and $\sum_{i \in Q} TCF(i)$ is maximized.

This problem is a generalized knapsack problem in that several arrays with non-overlapping lifetimes can intersect in the scratch pad. Panda et al.'s algorithm starts by clustering together arrays that could share scratch pad space. It then uses an approximation algorithm that first sorts the items by value per weight as given by access density *AD*:

$$AD(c) = \frac{\sum_{v \in c} TCF(v)}{max\{size(v), v \in c\}}. \quad \text{(EQ 2-15)}$$

Arrays are then greedily selected for allocation to the scratch pad, starting with the array with the highest *AD* value, until the scratch pad is full.

Figure 2-17 shows the allocation algorithm of Panda et al. Scalar variables are allocated to the scratch pad and arrays that are too large to fit into the scratch pad are allocated to main memory. A compatibility graph is created to determine arrays with compatible lifetimes. Cliques are then allocated, but the algorithm may allocate either a full clique or a proper subset of the clique to the scratch pad. Clique analysis takes

```
Algorithm MemoryAssign
Input: Application program P with Register-allocated variables marked;
SRAM_Size: Size of Scratch-Pad SRAM
Output: Assignment of arrays to SRAM/DRAM
    AvSpace = SRAM_Size   -- Available SRAM space
    Let U = {array  u|u  is an array in P}
    -- U is the set of all behavioral arrays in program P
    Let W = φ   -- W is the set of arrays assigned to DRAM
    for all variables v
        if v is a scalar variable or constant
            Assign v to SRAM
            AvSpace = AvSpace − size(v)
        else
            if size(v) > SRAM_Size
                W = W ∪ {v}   -- Assign v to DRAM
            end if
        end if
    end for
    Generate compatibility graph G from life-times of remaining arrays
    U = U − W   -- U is the set of all arrays <SRAM size
    while (U ≠ φ)
        for each array u ∈ U
            Find largest clique c(u) in G such that u ∈ c(u) and
            size(v) ≤ size(u) ∀v ∈ c(u)
            Compute access density AD(u) = (Σ_{v∈c(u)} TCF(v)) / size(u)
        end for
        Assign clique c(i) to SRAM, where AD(i) = max {AD(u)|u ∈ U}
        -- Assign cluster with highest access density to SRAM
        AvSpace = AvSpace − size(c)   -- size(c) =size of largest array in c
        X = {v ∈ U|size(v) > AvSpace}
        -- X = set of arrays in U larger than AvSpace
        W = W ∪ X -- Arrays in X are mapped to DRAM
        U = U − {v|(v ∈ c)} − X
        -- Remove from U arrays assigned to SRAM and arrays in X
    end while
    Assign arrays in W to DRAM
end Algorithm
```

[Pan00] fig. 5

Figure 2-17

An algorithm for scratch pad allocation [Pan00].

$O(n^3)$ time and the algorithm can iterate *n* times, so the overall complexity of the algorithm is $O(n^4)$.

multitasking and scratch pads

When several programs execute on the CPU, all the programs compete for the scratch pad, much as they compete for the cache. However, because the scratch pad is managed in software, the allocation algorithm must take multitasking into account. Panda et al. propose dividing the scratch pad into segments and assigning each task its own segment. This approach reduces run-time overhead for scratch pad management but

result in under-utilization of part of the scratch pad. When the programs are prioritized, they weight the *TCF* by the task priority, with higher-priority tasks given more weight. (Note that this is the inverse of the convention in real-time systems, in which the highest priority task is given a priority of 1.)

[Pan00] fig. 9

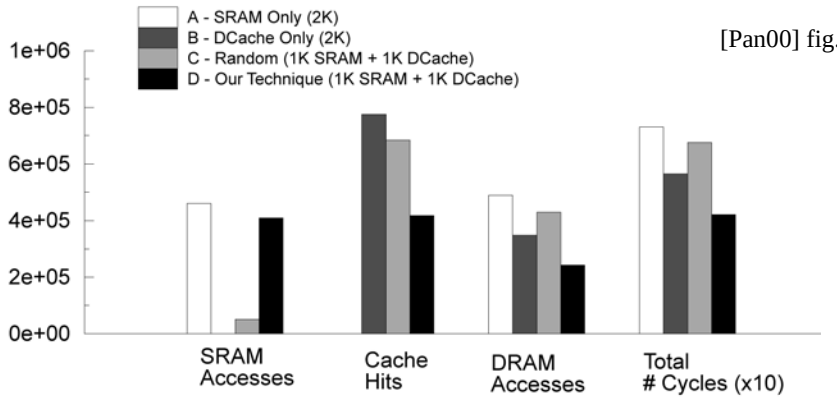


Figure 2-18

Performance comparison of scratch pad allocation [Pan00].

scratch pad evaluation

Figure 2-18 shows the performance of scratch pad allocation algorithms [Pan00] on one benchmark. The experiment compared using on-chip memory only for SRAM, only data cache, random allocation into scratch pad that occupies half the available SRAM, and Panda et al.'s allocation algorithm into a scratch pad that occupies half the available SRAM.

2.7 Additional CPU Mechanisms

This section covers other topics in the design of embedded processors. We will start with a discussion of code compression, an architectural technique that uses compression algorithms to design custom instruction encodings. We will then describe methods to encode bus traffic to reduce power consumption of address and data busses. We will end with a survey of security-related mechanisms in processors.

2.7.1 Code Compression

Code compression is one way to reduce object code size. Compressed instruction sets are not designed by people, but rather by algorithms. We can design an instruction set for a particular program, or we can use algorithms to design a program based on more

general program characteristics. Surprisingly, code compression can improve performance and energy consumption as well.

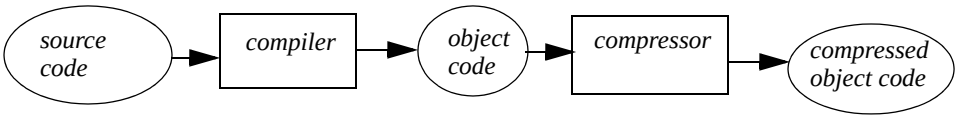


Figure 2-19

How to generate a compressed program.

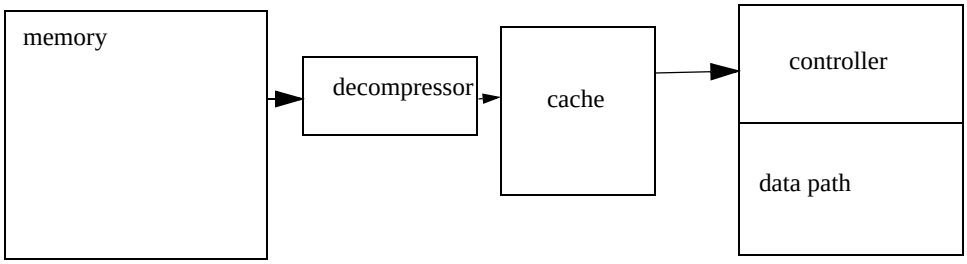


Figure 2-20

The Wolfe/Chanin architecture for executing compressed code.

executing compressed code

Wolfe and Chanin [Wol92] proposed code compression and developed the first method for executing compressed code. Relatively small modifications to both the compilation process and the processor allow the machine to execute code that has been compressed by lossless compression algorithms. Figure 2-19 shows their compilation processor. The compiler itself is not modified. The object code (or perhaps assembly code in text form) is fed into a compression program that uses lossless compression to generate a new, compressed object file that is loaded into the processor's memory. The compression program modifies the instruction but leaves data intact. Because the compiler does not need to be modified, compressed code generation is relatively easy to implement. Wolfe and Chanin used Huffman's algorithm [Huf52] to compress code.

microarchitecture with code decompression

Figure 2-20 shows the structure of a CPU modified to execute compressed code using the Wolfe/Chanin architecture. A decompression unit is added between the main memory and the cache. The decompressor intercepts instruction reads (but not data reads) from the memory and decompresses instructions as they go into the cache. The decompressor generates instructions in the CPU's native instruction set. The processor execution unit itself does not need to be modified because it does not see compressed instructions. The relatively small changes to the hardware make this scheme easy to implement with existing processors.

compressed code blocks

As illustrated in Figure 2-22, hand-designed instruction sets generally use a relatively small number of distinct instruction sizes and typically divide instructions on word

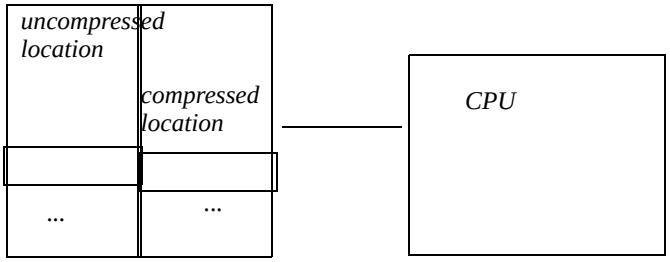


Figure 2-21

Branch tables for branch target mapping.

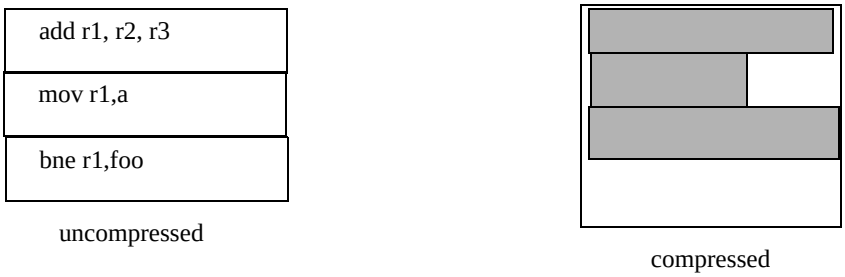


Figure 2-22

Compressed vs. uncompressed code.

or byte boundaries. Compressed instructions, in comparison, can be of arbitrary length. Compressed instructions are generally generated in blocks. The compressed instructions are packed bit-by-bit into blocks but the blocks start on more natural boundaries, such as bytes or words. This leaves empty space in the compressed program that is overhead for the compression process.

The block structure affects execution. The decompression engine decompresses code a block at a time. This means that several instructions become available in short order, although it generally takes several clock cycles to finish decompressing a block. Blocks effectively lengthen prefetch time.

Block structure also affects the compression process and the choice of a compression algorithm. Lossless compression algorithms generally work best on long blocks of data. However, longer blocks impede efficient execution because programs are not executed sequentially from beginning to end. If the entire program were a single block, we would decompress the entire block before execution, which would nullify the advantages of compression. If blocks are too short, the code will not be sufficiently compressed to be worthwhile.

Wolfe/Chanin evaluation

Figure 2-24 shows Wolfe and Chanin's comparison of several compression methods. They compressed several benchmark programs in four different ways: using the Unix *compress* utility; using standard Huffman encoding on 32-byte blocks of instructions,

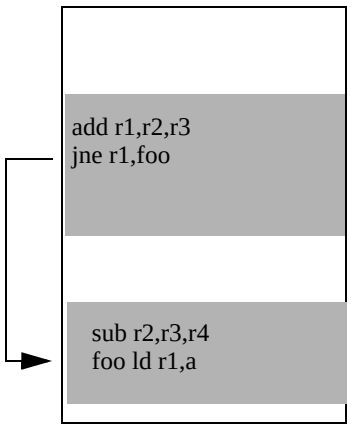


Figure 2-23

Branches and blocks in compressed code.

using a Huffman code designed specifically for that program; using a bounded Huffman code that ensures that no byte is coded in a symbol longer than 16 bits, once again with a separate code for each program; and with a single bounded Huffman code computed from several test programs and used for all the benchmarks.

Wolfe and Chanin also evaluated the performance of their architecture on the benchmarks using three different memory models: programs stored in EPROM with 100 ns memory access time; program stored in burst-mode EPROM with 3 cycles for the first access and 1 cycle for subsequent sequential accesses; and static-column DRAM with 4 cycles for the first access and 1 cycle for subsequent sequential accesses, based on 70 ns access time. They found that system performance improved when compressed code was run from slow memories and that system performance slowed down about 10% when executed from fast memory.

branches in compressed code

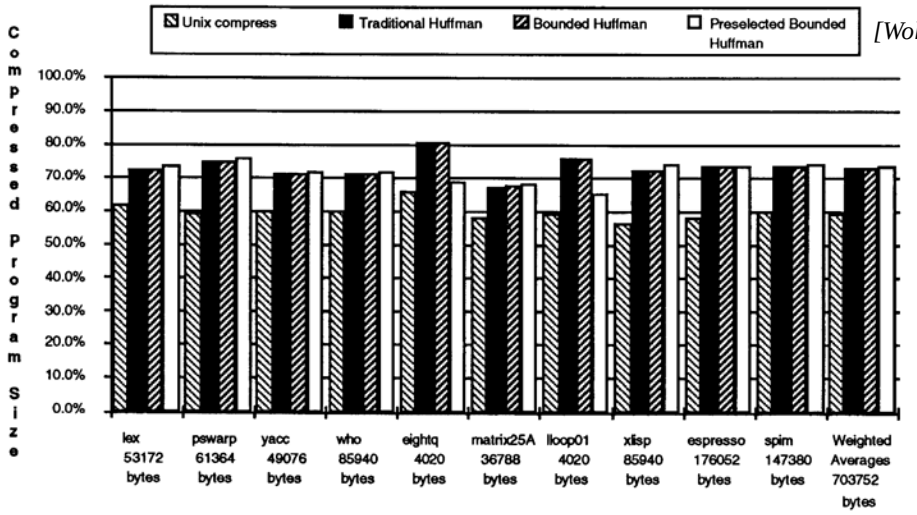
If a branch is taken in the middle of a large block, we may not use some of the instructions in the block, wasting the time and energy required to decompress those instructions. As illustrated in Figure 2-23, branches and branch targets may be at arbitrary points in blocks. The ideal size of a block is related to the distances between branches and branch targets.

branch tables

The locations of branch targets in the uncompressed code must be adjusted in the compressed code because the absolute locations of all the instructions move as a result of compression. Most instruction accesses are sequential, but branches may go to arbitrary locations given by labels. However, the location of the branch has moved in the compressed program. Wolfe and Chanin proposed that **branch tables** be used to map compressed locations to uncompressed locations during execution. The branch table would be generated by the compression program and included in the compressed object code. It would be loaded into the branch table at the start of execution (or after a context switch) and used by the CPU every time an absolute branch location needed to be translated.

branch patching

An alternative to branch tables, proposed by Bird and Mudge [Bir96], is **branch patching**. This method first compresses the code, doing so in a way that branch instructions can still be modified. After the locations of all the instructions are known, the compression system modifies the compressed code. It changes all of the branch instructions to include the address of the compressed branch target, rather than the uncompressed location. This method uses a slightly less efficient encoding for branches because the address must be modifiable, but it also eliminates the branch table. The branch table introduces several types of overhead: it is large, with one entry for each branch target; it consumes a great deal of energy; and accessing the branch table slows down execution of branches. The branch patching scheme is generally considered to be the preferred method.



[Wol92] Fig 5

Figure 2-24

Wolfe and Chanin's comparison of code compression efficiency [Wol92]

code compression metrics

We can judge code compression systems by several metrics. The first is code size We generally measure code size by **compression ratio**:

$$K = \frac{\text{compressed code size}}{\text{uncompressed code size}}. \qquad \text{(EQ 2-16)}$$

Compression ratio is measured independent of execution; in Figure 2-19, we would compare the size of the uncompressed object code to the size of the compressed object code. For this result to be meaningful, we must measure the entire object code. This includes data. It also includes artifacts in the compressed code itself, such as empty space and branch tables.

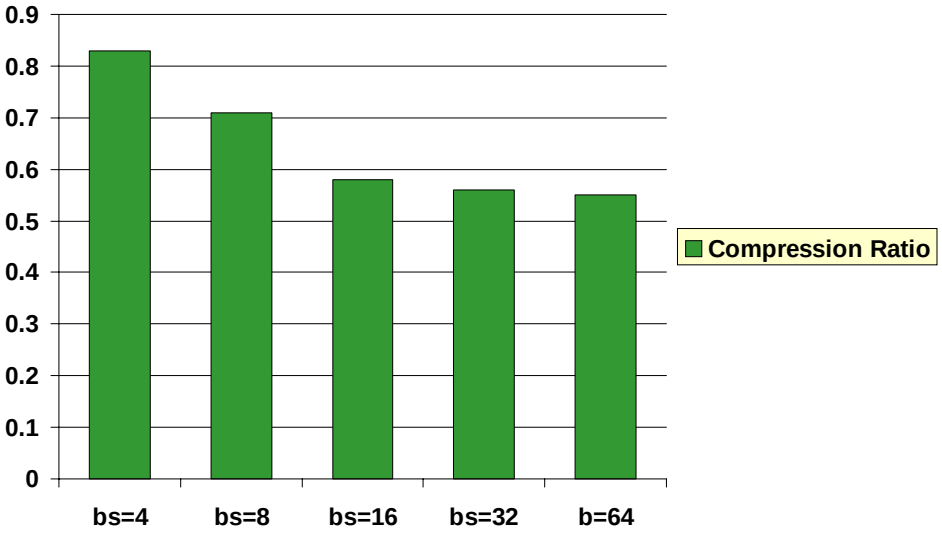


Figure 2-25

Compression ratio vs. block size (in bytes) for one compression algorithm [Lek xx].

block size vs. compression ratio

The choice of block size is a major decision in the design of a code compression algorithm. Lekatsas et al. measured compression ratio as a function of block size. These results show that very small blocks do not compress well, but that compression ratio levels off for even moderately-sized blocks.

We can also judge code compression systems by performance and power consumption.

The next example describes the major implementation of code compression in a commercial processor.

Application Example 2-8

IBM CodePack

The IBM CodePack architecture was implemented on some models of PowerPC [Kem98]. CodePack uses Huffman compression on each 16-bit half of a 32-bit instruction; instructions are divided into blocks of 64 bytes. CodePack achieved compression ratios of 55% to 65%. A branch table is used to translate addresses. The K bit to the TLB indicates whether a page in memory holds compressed instructions.

The next example describes a hand-designed, compact instruction set.

Application Example 2-9

ARM Thumb Instruction Set

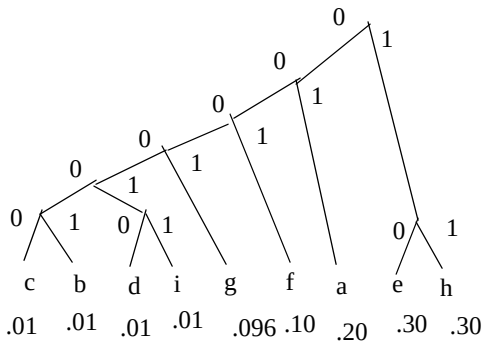
The ARM Thumb instruction set is an extension to the basic ARM instruction set; any implementation that recognizes Thumb instructions must also be able to interpret standard ARM instructions. Thumb instructions are 16 bits long.

data compression algorithms and code compression

Many data compression algorithms were originally developed to compress text. Code decompression during execution imposes very different constraints: high performance, small buffers, low energy consumption. Let's look at several data compression algorithms that have been proposed for code compression and see how they map onto the constraints of decompression during execution.

a	0.20
b	0.01
c	0.01
d	0.01
e	0.30
f	0.10
g	0.096
h	0.30
i	0.01

symbols and probabilities



coding tree

Figure 2-26

Huffman coding.

Huffman coding

Let's first review Huffman's algorithm, the first modern code compression algorithm. Huffman's method requires an alphabet of symbols and the probabilities of occurrence of those symbols. As shown in Figure 2-26, a **coding tree** is built based on those probabilities. Initially, we build a set of subtrees, each having only one leaf node for a symbol. The score of a subtree is the sum of the probabilities of all its leaf nodes. We repeatedly choose the two lowest-score subtrees and combine them into a new subtree, with the lower-probability subtree taking the 0 branch and the higher-probability subtree

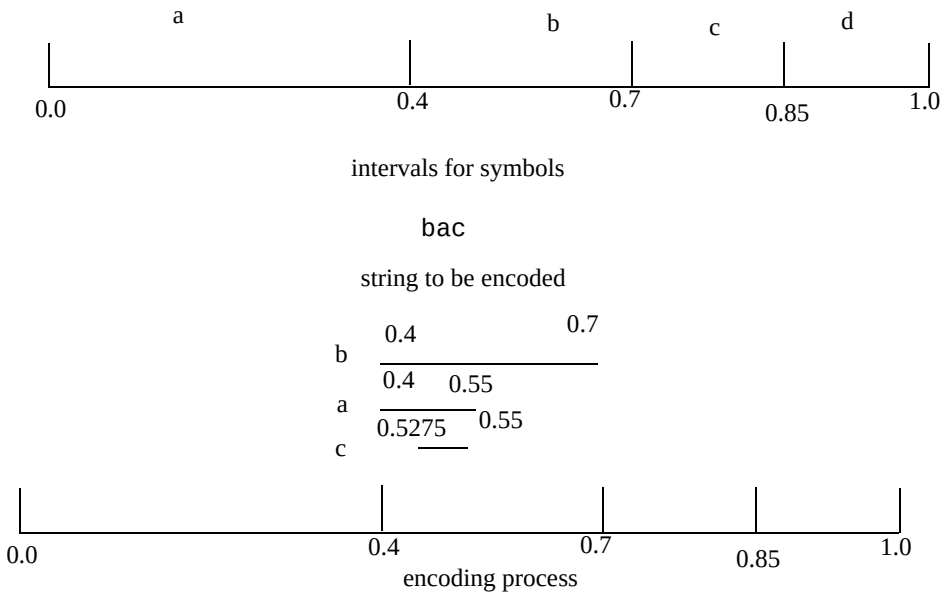


Figure 2-27

Arithmetic coding.

taking the 1 branch. We continue combining subtrees until we have formed a single large tree. The code for a symbol can be found by following the path from the root to the appropriate leaf node, noting the encoding bit at each decision point.

arithmetic coding

Arithmetic coding was proposed by Whitten, Neal, and Cleary [Whi87] as a generalization of Huffman coding. Huffman coding can make only discrete divisions of the coding space. Arithmetic coding, in contrast, uses real numbers to divide codes into arbitrarily small segments; this is particularly useful for sets of symbols with similar probabilities. As shown in Figure 2-27, the real number line [0,1] can be divided into segments corresponding to the probabilities of the symbols. For example, the symbol *a* occupies the interval [0,0.5). Any real number in a symbol's interval can be used to represent that symbol. Arithmetic coding selects values within those ranges so as to encode a string of symbols with a single real number.

A string of symbols is encoded using this algorithm:

```
low = 0; high = 1; i=0;
while (i < strlen(string)) {
    range = high - low;
    high = low + range*high_range(string[i]);
    low = low + range*low_range(string[i]);
}
```

An example is shown at the bottom of Figure 2-27. The range is repeatedly narrowed to represent the symbols and their sequence in the string.

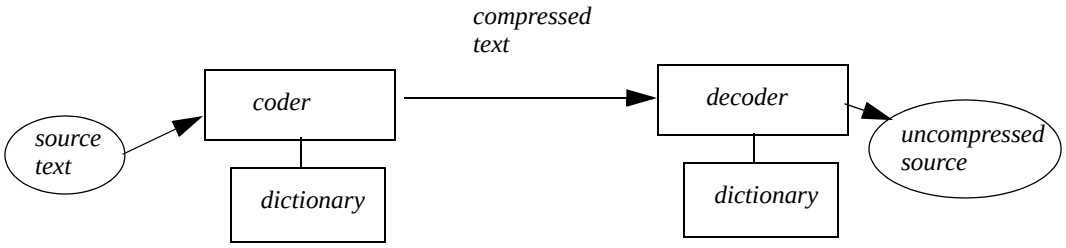


Figure 2-28

Lempel-Ziv coding.

Figure 2-29

An example of Lempel-Ziv coding.

Lempel-Ziv coding

Lempel-Ziv coding [Ziv77] builds a dictionary for coding. It does so in a way that it does not have to send the dictionary along with the compressed text. As shown in Figure 2-28, the transmitter uses a buffer to recognize repeated strings in the input text; the receiver keeps its own buffer to record repeated strings as they are received so that they may be reused in later steps of the decoding process. The Lempel-Ziv coding process is illustrated in Figure 2-29. The compressor scans the text from first to last character. If the current string, including the current character, is in the buffer, no text is sent. If the current string is not in the buffer, it is added to the buffer, the string is sent.

Lempel-Ziv-Welch coding

The Lempel-Ziv-Welch (LZW) algorithm [Wel84] uses a fixed-size buffer for the Lempel-Ziv algorithm. LZW coding was originally designed for disk drive compression, in which the buffer is a small RAM; it is also used for image encoding in the GIF format.

Markov models

Markov models are well-known statistical models. We use Markov models in data compression to model the conditional probabilities of symbols—for example, the probability that *z* follows *a*, as compared to the probability of *w* following *a*. As shown in

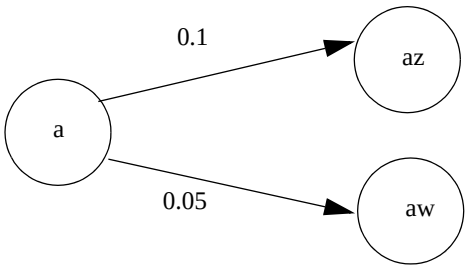


Figure 2-30

A Markov model for conditional character probabilities.

Figure 2-30, each possible state in the sequence is modeled by a state in the Markov model. A transition shows possible moves between states, with each transition labeled by its probability. In the example, states model the sequences *az* and *aw.*

arithmetic coding and Markov models

Lekatsas and Wolf [Lek98,Lek99] combined arithmetic coding and Markov models. Arithmetic coding provides more efficient codes than Huffman coding, but requires more careful coding. Markov models allow coding to take advantage of the relationships between strings of symbols. We will consider each of these factors.

Example Interval Machine for N=8			
State	P(MPS)	LPS	MPS
[0,8)	7/8	000, [0,8)	-, [1,8)
	6/8	00, [0,8)	-, [2,8)
	4/8	0, [0,8)	1, [0,8)
[1,8)	6/7	001, [0,8)	-, [2,8)
	5/7	0f, [0,8)	-, [3,8)
	4/7	0, [2,8)	1, [0,8)
[2,8)	5/6	010, [0,8)	-, [3,8)
	4/6	01, [0,8)	1, [0,8)
[3,8)	4/5	011, [0,8)	1, [0,8)
	3/5	ff, [0,8)	1, [2,8)

[Lek99] Fig 4.

Figure 2-31

An example of table-based arithmetic decoding [Lek99].

Arithmetic coding is formulated in terms of real numbers; a straightforward implementation would require floating-point arithmetic. A floating-point arithmetic unit is too

slow, too large, and too energy-intensive to be used in the instruction decode path. Howard and Vitter [How92] developed a table-based algorithm for arithmetic compression that requires only fixed-point arithmetic. An example is shown in Figure 2-31. The table encodes the segments into which the number line has been divided by the code.

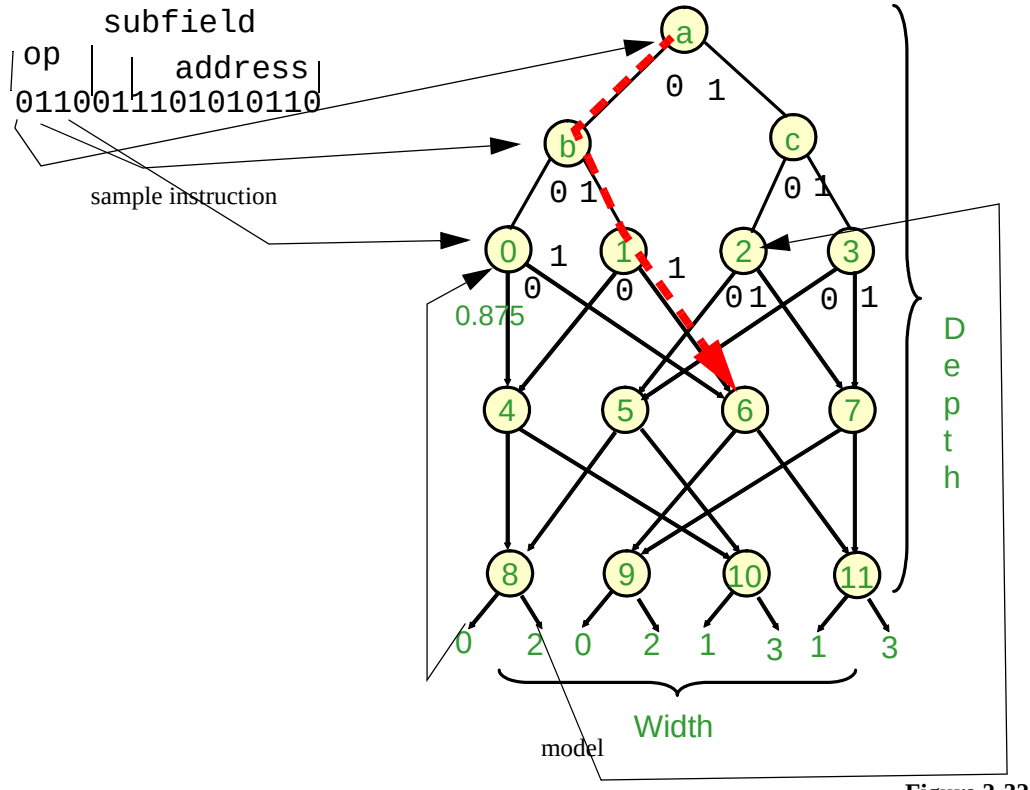
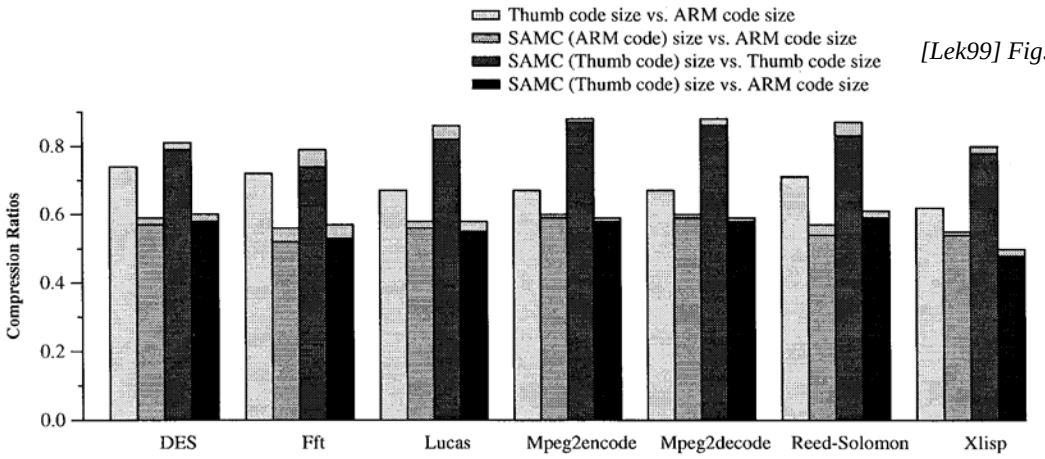


Figure 2-32

A Markov model of an instruction.

The Markov model describes the relationship between the bits in an instruction. As shown in Figure 2-32, each state has two transitions, one for a zero bit and one for a one bit. Any particular instruction defines a trajectory through the Markov model. Each state is marked with the probability of the most probable bit and whether that bit is 0 or 1. The largest possible model for a block of *b* bits would have 2^b states, which is too large. We can limit the size of the model in two ways. We can limit the width by wrapping around transitions so that a state may have more than one incoming transition. We can limit the depth in a similar way by cutting off the bottom of the model and wrapping around transitions to terminate at existing states. The depth of the model should divide the instruction size evenly or be a multiple of the instruction size so that the root state always falls on the start of an instruction.



[Lek99] Fig. 13

Figure 2-33

SAMC vs. ARM and Thumb [Lek99].

Lekatsas and Wolf compared SAMC to both ARM and Thumb. As shown in Figure 2-33, SAMC created smaller programs than Thumb—compressed ARM programs were smaller than Thumb and compressed Thumb programs were also smaller.

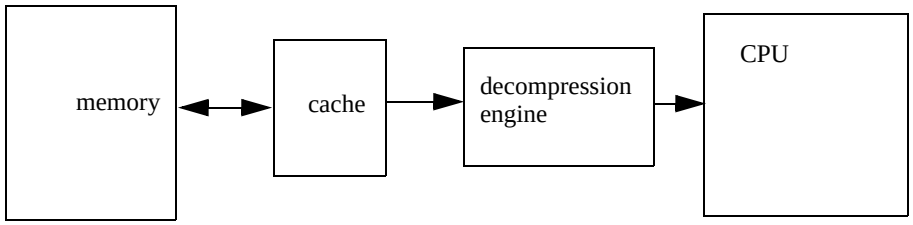


Figure 2-34

A post-cache decompression architecture.

pre-cache vs. post-cache decompression

The Wolfe/Chanin architecture compresses blocks as they come into the cache. This means that each block need be decompressed only once but it also means that the cache is filled with uncompressed instructions. Lekatsas et al [Lek00] proposed **post-cache decompression** in which instructions are decompressed between the cache and CPU, as shown in Figure 2-34. This architecture requires instructions in a block to be decompressed many times if they are repeatedly executed, such as in a loop, but it also leaves compressed instructions in the cache. The post-cache architecture effectively makes the cache larger because the instructions take up less room. Architects can either use a smaller cache to achieve the same performance or achieve a higher cache hit rate for a given cache size. This gives the architect trade-offs between area, performance, and energy consumption (since the cache consumes a large amount of energy). Surprisingly,

the post-cache architecture can be considerably faster and consume less energy even when the overhead of repeated decompressions is taken into account.

2.7.2 Low-Power Bus Encoding

The busses that connect the CPU to the caches and main memory account for a significant fraction of all the energy consumed by the CPU. These busses are both wide and long, giving them large capacitance to switch. These busses are also frequently used, inducing many switching events on their large capacitance.

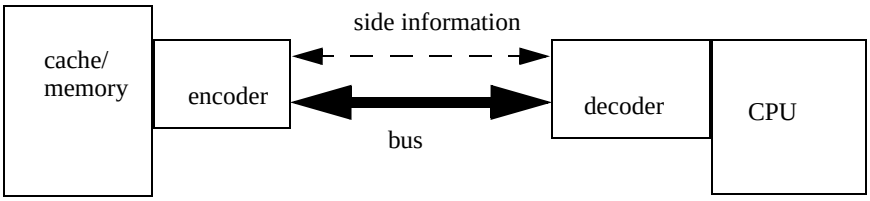


Figure 2-35

Microarchitecture of an encoded bus.

bus encoding

A number of **bus encoding** systems have been developed to reduce bus energy consumption. As shown in Figure 2-35, information to be placed on the bus is first encoded at the transmitting end and then decoded at the receiving end. The memory and CPU do not know that the bus data is being encoded. Bus encoding schemes must be invertible—we must be able to losslessly recover the data at the receiving end. Some schemes require **side information**, usually a small number of bits to help decode. Other schemes do not require side information to be transmitted alongside the bus.

metrics

The most important metric for a bus encoding scheme is energy savings. Because the energy itself depends on the physical and electrical design of the bus, we usually measure energy savings using **toggle counts**. Because bus energy is proportional to the number of transitions on each line in the bus, we can use toggle count as a relative energy measure. Toggle counts measure toggles between successive bits on a given bus signal. Because crosstalk also reduces power consumption, some schemes also look at the values of physically adjacent bus signals. We are also interested in the time, energy, and area overhead of the encoder and decoder. All these metrics must include the toggle counts and other costs of any side information that is used for bus encoding.

bus-invert coding

Stan and Burleson proposed **bus-invert coding** [Sta95] to reduce the energy consumption of busses. This scheme takes advantage of the correlation between successive values on the bus. A word on the bus may be transmitted either in its original form or inverted form to reduce the number of transitions on the bus.

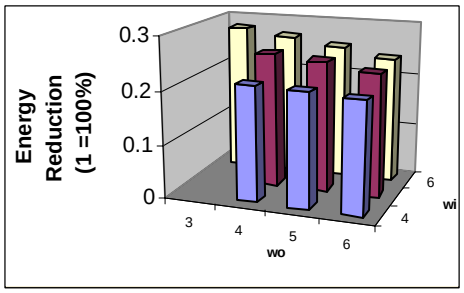


Figure 2-36

Evaluation of dictionary-based code compression [Lv03].

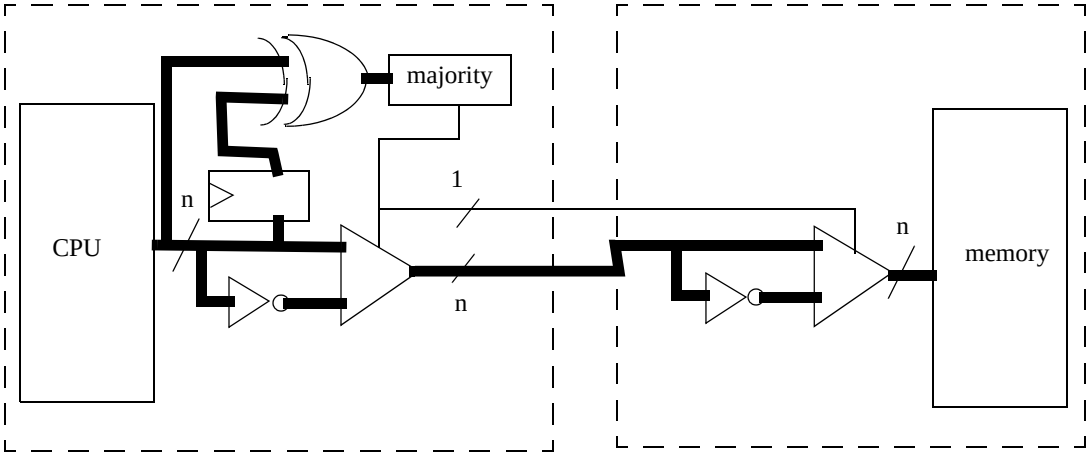
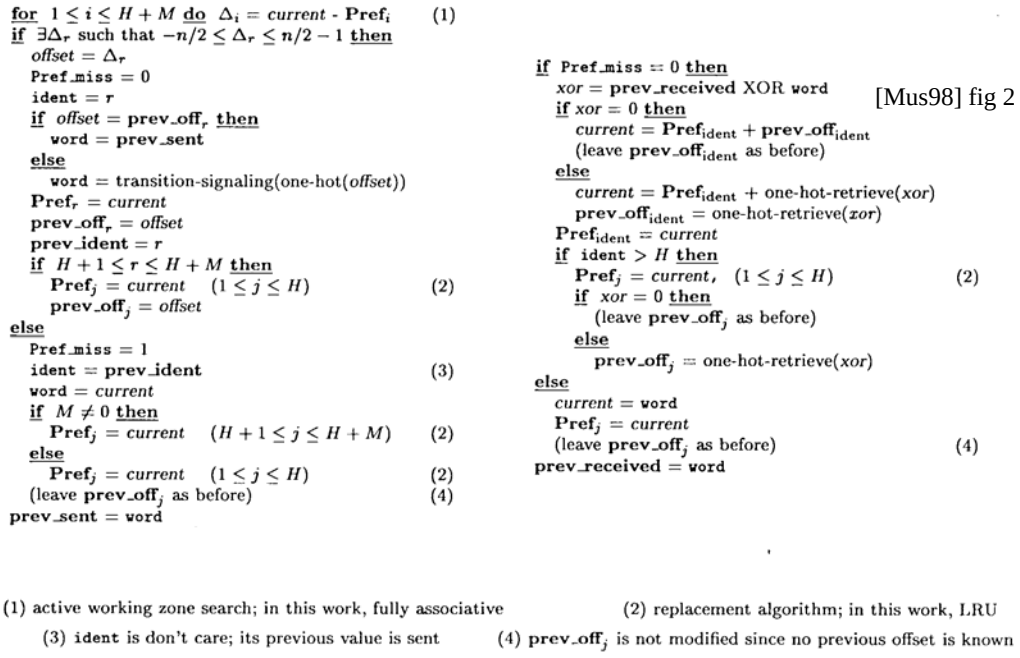


Figure 2-37

The bus-invert coding architecture.

As shown in Figure 2-37, the transmitter side of the bus (for example, the CPU) stores the previous value of the bus in a register so that it can compare the current bus value to the previous value. It then computes the number of bit-by-bit transitions using the function majority$(XOR(B^t, B^{t-1}))$, where B^t is the bus value at time *t*. If more than half the bits of the bus change value from time *t* to time *t*-1, then the inverted form of the bus value is transmitted, otherwise the original form of the bus value is transmitted. One extra line of side information is used to tell the receiver whether it needs to re-invert the value.

Stan and Burleson also proposed breaking the bus into fields and applying bus-invert coding to each field separately. This approach works well when data is naturally divided into sections with correlated behavior.



[Mus98] fig 2

Figure 2-38

Working zone encoding and decoding algorithms [Mus98].

working-zone encoding

Musoll et al. [Mus98] developed **working-zone encoding** for address busses. Their method is based on the observation that a large majority of the execution time of programs is spent within small ranges of addresses, such as during the execution of loops. They divide program addresses into sets known as **working zones**. When an address on the bus falls into a working zone, the offset from the base of the working zone is sent in a one-hot code. Addresses that do not fall into working zones are sent in their entirety.

address bus encoding

Benini et al [Ben98] developed a method for address bus encoding. They cluster address bits that are correlated, create efficient codes for those clusters, then use combinational logic to encode and decode those clustered signals.

They compute correlations of **transition variables**:

$$\eta_i^{(t)} = [x_i^{(t)} \cdot (x_i^{(t-1)})'] - [(x_i^{(t)})' x_i^{(t-1)}], \qquad \text{(EQ 2-17)}$$

where $\eta_i^{(t)}$ is 1 if bit i makes a positive transition, -1 if it makes a negative transition, and 0 if it does not change. They compute correlation coefficients of this function for the entire address stream of the program.

In order to make sure that the encoding and decoding logic is not too large, we must control the sizes of the clusters chosen. Benini et al. use a greedy algorithm to create

clusters of signals with controlled maximum sizes. They use logic synthesis techniques to design efficient encoders and decoders for the clusters.

[Ben98] table 1

benchmark	*address length*	*number of instructions*	*Benini et al. bus transitions*	*Benini et al. savings*	*working zone bus transitions*	*working zone savings*
dashboard	*84918*	*619690*	*443115*	*28.4%*	*452605*	*26.9%*
DCT	*13769*	*48917*	*31472*	*35.6%*	*36258*	*25.8%*
FFT	*23441*	*138526*	*85653*	*38.1%*	*99814*	*27.9%*
matrix multiplication	*22156*	*105947*	*60654*	*42.7%*	*72881*	*31.2%*
vector-vector multiplication	*19417*	*133272*	*46838*	*64.8%*	*85473*	*35.8%*

Figure 2-39

Experimental evaluation of address encoding using the method of Benini et al. [Ben98].

Figure 2-39 shows the results of experiments by Benini et al comparing their method to working zone encoding.

[Lv03] fig 6

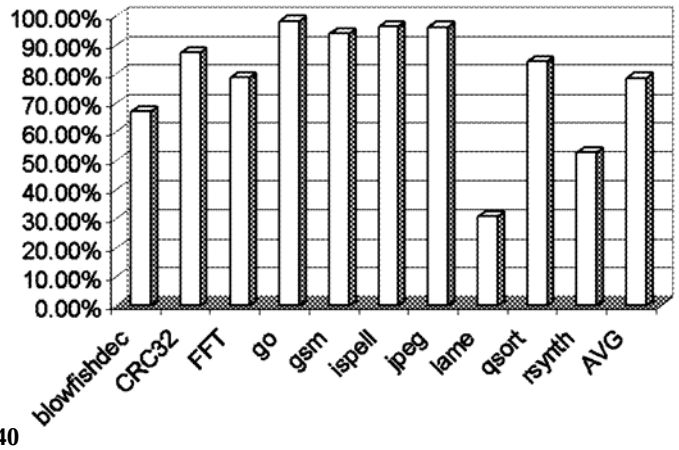


Figure 2-40

Frequency of the ten most frequent patterns in a set of benchmarks [Lv03].

dictionary-based bus encoding

Lv et al. [Lv03] developed a bus encoding method that uses dictionaries similar to those used in Lempel-Ziv encoding. This method is designed to consider both correlations between successive values and correlations between adjacent bits. They use a sim-

ple model of energy consumption for a pair of lines: the function $ENS(V^t,V^{t-1})$ 0 if both lines stay the same, 1 if one of the two lines changes, and 2 if both lines change. They then model energy in terms of transition (*ET*) and interwire (*EI*) effects:

$$ET(k) = C_L V_{DD}^2 \sum_{0 \le i \le N-1} ENS(V_i((k-1), V_i(k))), \quad \text{(EQ 2-18)}$$

$$EI(k) = C_L V_{DD}^2 \sum_{0 \le i \le N-2} ENS(V_i((((k-1), V_{i+1}(k)), V_i(k)), V_{i+1}(k))) \quad \text{(EQ 2-19)}$$

$$EN(k) = ET(k) + EI(k). \quad \text{(EQ 2-20)}$$

EN(k) gives the total energy on the k^{th} bus transaction.

Dictionary encoding makes sense for busses because many values are repeated on busses. Figure 2-40 shows the frequency of the ten most common patterns in a set of benchmarks. A very small number of patterns clearly accounts for a majority of the bus traffic in these programs. This means that a small dictionary can be used to encode many of the values that appear on the bus.

[Lv03] fig 10

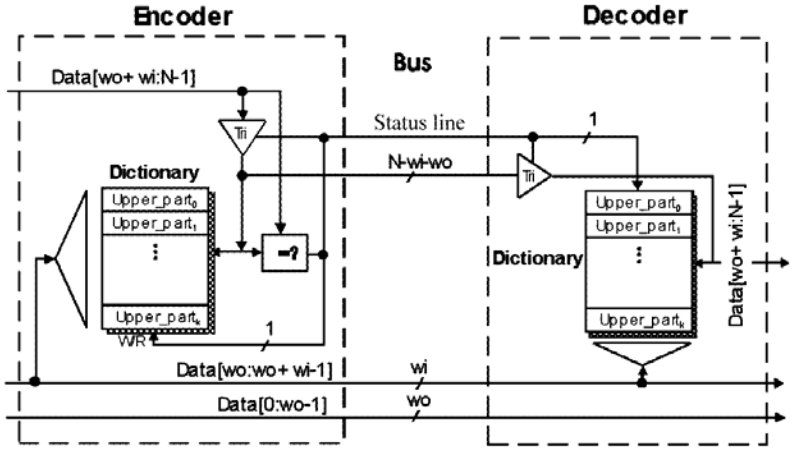


Figure 2-41

Architecture of the dictionary-based bus encoder of Lv et al. [Lv03].

Figure 2-41 shows the architecture of the dictionary-based encoding scheme. Both the encoder and decoder have small dictionaries built from static RAM. Not all of the word is used to form the dictionary entry; only the upper bits of the word are used to match. The remaining bits of the word are sent unencoded. They divide the bus into three

sections: the upper part of width *N-wi-w0*, the index part of width *wi*, and the bypassed part of width *wo*. If the upper part of the word to be sent is in the dictionary, the transmitter sends the index and the bypassed part. When the upper part is not used, those bits are put into a high-impedance state to save energy. Side information tells when a match is found. Some results are summarized in Figure 2-36; Lv et al. found that this dictionary-based saved about 25% of bus energy on data values and 36% on address values at a cost of two additional bus lines for side information and about 4400 gates.

2.7.3 Security

Security, in the context of CPUs, can mean two different things. It can mean features in the CPU designed for use in cryptography and related security operations. It can also mean protection features against attacks. We will consider both aspects of CPU security in this section.

cryptography and CPUs

Cryptographic operations are used in key-based cryptosystems. Cryptographic operations, such as key manipulation, are part of protocols that are used to authenticate data, users, and transactions.

Cryptographic arithmetic requires very long word operations and a variety of bit- and field-oriented operations. A variety of instruction set extensions have been proposed to support cryptography. Co-processors may also be used to implement these operations.

varieties of attacks

Embedded processors must also guard against attacks. Attacks used against desktop and server systems—Trojan horses, viruses, etc.—can be used against embedded systems. Because users and adversaries have physical access to the embedded processor, new types of attacks are also possible. **Side channel attacks** use information leaked from the processor to determine what the processor is doing.

smart cards

Smart cards are an excellent example of a widely-used embedded system with stringent security requirements. Smart cards are used to carry highly-sensitive data such as credit and banking information or personal medical data. Tens of millions of smart cards are in use today. Because the cards are held by individuals, they vulnerable to a variety of physical attacks, either by third parties or by the card holder.

Figure 2-42 shows a smart card chip, which includes a 32-bit microcontroller, RAM, ROM, flash memory, and I/O devices. This architecture is sometimes called the **self-programmable one-chip microcomputer** (**SPOM**) architecture. The electrically programmable memory allows the processor to change its own permanent program store.

self-reprogramming architecture

SPOM architectures allow the processor to modify either data or code, including the code that is being executed. Such changes must be done carefully to be sure that the memory is changed correctly and CPU execution is not compromised. Figure 2-43 shows an architecture for self-reprogramming [Ugo83]. The memory is divided into two sections: both may be EPROM or one may be ROM, in which case the ROM section of the memory cannot be changed. The address and data are divided among the two programs. Registers are used to hold the addresses during the memory operation as well as to hold the data to be read or written. One address register holds the address to be read/written while the other holds the address of a program segment that controls the memory

Tual MPSOC slide 3

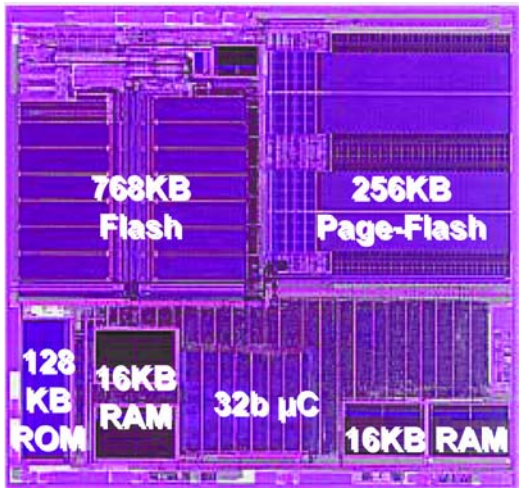


Figure 2-42
Photomicrograph of a smart card chip (courtesy Axalto).

operation. Control signals from the CPU determine the timing of the memory operation. Segmenting the memory access into two parts allows arbitrary locations to be modified without interfering with the execution of the code that governs the memory modification. Because of the registers and control logic, even the locations that control the memory operation can be overwritten without causing operational problems.

power attacks

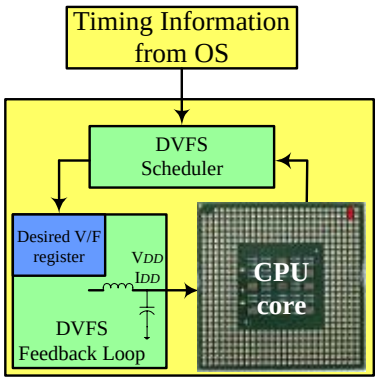
Side channel attacks, as mentioned above, use information emitted from the processor to determine the processor's internal state. Electronic systems typically emit electromagnetic energy that can be used to infer some of the circuit's activity. Similarly, the dynamic behavior of the power supply current can be used to infer internal CPU state. Kocher et al. [Koc99] showed that, using a technique they call **differential power analysis**, measurements of the power supply current into a smart card could be used to identify the cryptosystem key stored in the smart card.

countermeasures

Countermeasures have been developed for power attacks. Yang et al [Yan05] used dynamic voltage and frequency scaling to mask operations in processors as shown in Figure 2-43. They showed that proper design of a DVFS schedule can make it substantially harder for attackers to determine internal states from the processor power consumption. Figure 2-44 compares traces without dynamic voltage and frequency scaling (a and c) and traces with DVFS-based protection (b and d).

2.8 CPU Simulation

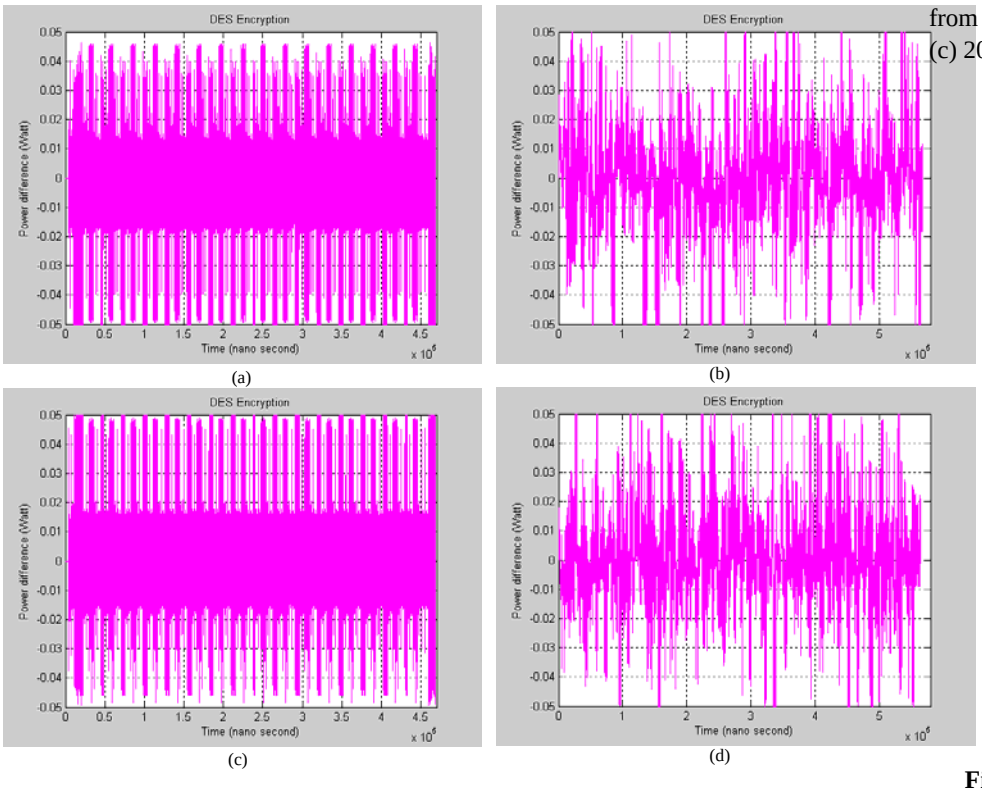
CPU simulators are essential to computer system design. CPU architects use simulators to evaluate processor designs before they are built. System designers use simulators for a



from [Yan05]
(c) 2005 ACM

Figure 2-43

Protecting against power attacks with dynamic voltage and frequency scaling.



from [Yan05]
(c) 2005 ACM

Figure 2-44

Traces without and with DVFS protection.

variety of purposes: analyzing the performance and energy consumption of programs; simulating multiprocessors before and after they are built; and system debugging.

The term "CPU simulator" is generally used broadly to mean any method of analyzing the behavior of programs on processors. We can classify CPU simulation methods along several lines:

- **performance *vs*. energy** Simulating the energy consumption of a processor requires accurate simulation of its internal behavior. Some types of performance-oriented simulation, in contrast, can perform a less detailed simulation and still provide reasonably accurate results.

- **temporal accuracy** By simulating more details of the processor, we can obtain more accurate timings. More accurate simulators take more time to execute.

- **trace *vs*. execution** Some simulators analyze a trace taken from a program that executes on the processor. Analyze the program execution directly.

- **simulation *vs*. direct execution** Some execution-based systems directly execute on the processor being analyzed while others use simulation programs.

This section surveys techniques for CPU simulation.

2.8.1 Trace-Based Analysis

Trace-based analysis systems do not directly operate on a program. Instead, they use a record of the program's execution, called a **trace**, to determine characteristics of the processor.

tracing and analysis

As shown in Figure 2-45, trace-based analysis gathers information from a running program. The trace is then analyzed after execution of the program being analyzed. The post-execution analysis is limited by the data gathered in the trace during program execution.

The trace can be generated in several different ways. The program can be instrumented with additional code that writes trace information to memory or a file. The instrumentation is generally added during compilation or by editing the object code. The trace data can also be taken by a process that interrupts the program and samples its program counter (PC). These two techniques are not mutually exclusive.

Profiling information can be taken on a variety of types of program information and at varying levels of granularity:

- **control flow** Control flow is useful in itself and a great deal of other information can be derived from control flow. The program's branching behavior can generally be captured by recording branches; behavior within the branches can be inferred. Some systems may record function calls but not branches within a function.

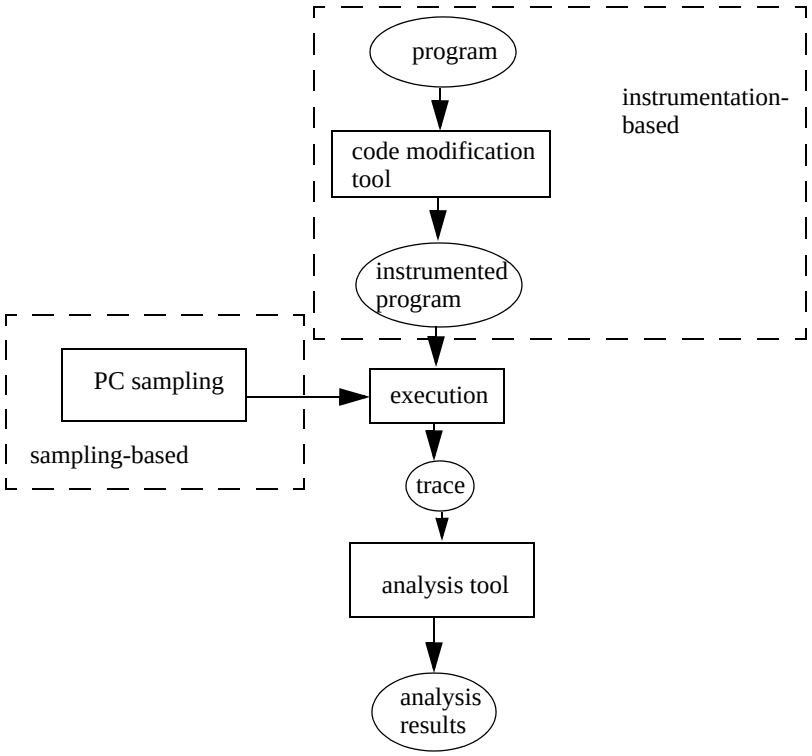


Figure 2-45

The trace-based analysis process.

- **memory accesses** Memory accesses tell us about cache behavior. The Instruction cache behavior can be inferred from control flow. Data memory accesses are usually recorded by instrumentation code that surrounds each memory access.

prof

An early and well-known trace-based analysis tool is the Unix *prof* command and its GNU cousin *gprof*. [Fen98] *gprof* uses a combination of instrumentation and PC sampling to generate the trace. The trace data can generate call-graph (procedure-level), basic-block-level, and line-by-line data.

dinero

A very different type of trace-based analysis tool is the well-known *Dinero* tool [Edl03]. *Dinero* is a cache simulator. It does not analyze the timing of a program's execution, rather it only looks at the history of references to memory. The reference memory history is captured by instrumentation within the program. After execution, the user analyzes the program behavior in the memory hierarchy using the *Dinero* tools. The user design a cache hierarchy in the form of a tree and set the parameters for the caches for analysis.

trace sampling

Traces can be sampled rather than recorded in full [Lah88]. A useful execution may require a great deal of data. Consider, for example, a video encoder that must process

several frames to exhibit a reasonable range of behavior. The trace may be sampled by taking data for a certain number of instructions and then not recording information for another sequence of instructions. It is often necessary to warm up the cache before taking samples. The usual challenges of sampled data apply: an adequate length of sample must be taken at each point and samples must be taken frequently enough to catch important behavior.

2.8.2 Direct Execution

emulating architectures

Direct execution-style simulation makes use of the host CPU used for simulation in order to help compute the state of the target machine. Direct execution is used primarily for functional and cache simulation, not for detailed timing.

The various registers of the computer comprise its state; we need to simulate those registers in the target machine that are not defined in the host machine but we can use the host machine's native state where appropriate. A compiler generates code for the simulation by adding instructions to compute the target state that needs to be simulated. Because much of the simulation runs as native code on the host machine, direct execution can be very fast.

2.8.3 Microarchitecture-Modeling Simulators

modeling detail and accuracy

We can provide more detailed performance and power measurements by building a simulator that models the internal microarchitecture of the computer. Directly simulating the logic would provide even more accurate results but would be much too slow, preventing us from running the long traces that we need to judge system performance. Logic simulation also requires us to have the logic design, which is not generally available from the CPU supplier. But in many cases we can build a functional model of the microarchitecture from publicly available information.

Microarchitecture models may vary in the level of detail they capture about the microarchitecture. **Instruction schedulers** model basic resource availability but may not be cycle-accurate. **Cycle timers**, in contrast, model the architecture in more detail in order to provide cycle-by-cycle information about execution. Accuracy generally comes at the cost of somewhat slower simulation.

modeling for simulation

A typical model for a 3-stage pipelined machine is shown in Figure 2-46. This model is not a register-transfer model in that it does not include the register file or busses as first-class elements. Those elements are instead subsumed into the models of the pipeline stages. The model captures the main units and paths that contribute to data and control flow within the microarchitecture.

simulator design

The simulation program consists of modules that correspond to the units in the microarchitectural model. Because we want the simulator to run fast, these simulators are typically written in a sequential language such as C, not in a simulation language like

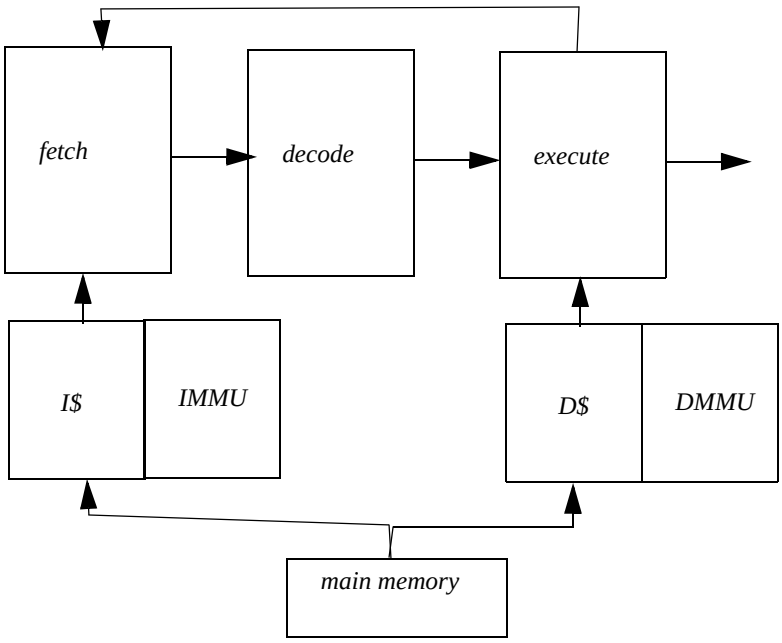


Figure 2-46

A microarchitectural model for simulation.

Verilog or VHDL. Simulation languages have mechanisms to ensure that modules are evaluated in the proper order when the simulation state changes; when we write simulators in sequential languages, we must design the control flow in the program to ensure that all the implications of a given state change are properly evaluated.

SimpleScalar

SimpleScalar [Sim05] is a well-known toolkit for simulator design. Simplescalar provides modules that model typical components of CPUs as well as tools for data collection. These tools can be put together in various ways, modified, or added to in order to create a custom simulator. A machine description file describes the microarchitecture and is used to generate parts of the simulation engine as well as programming tools like disassemblers.

power simulation

Power simulators take cycle-accurate microarchitecture simulators one step further in detail. Determining the energy/power consumption of a CPU generally requires even more accurate modeling than performance simulation. For example, a cycle-accurate timing simulator may not directly model the bus. But the bus is a major consumer of energy in a microprocessor, so a power simulator needs to model the bus, as well as register files and other major structural components. In general, a power simulator most model all significant sources of capacitance in the processor since dynamic power consumption is directly related to capacitance. However, power simulators must trade-off accuracy for simulation performance just like other cycle-accurate simulators.

Wattch and SimplePower

The two best-known power simulators are Wattch [Bro00] and SimplePower [Ye00]. Both are built on top of SimpleScalar and add capacitance models for the major units in the microarchitecture.

2.9 Automated CPU Design

System designers have long used custom processors to run applications at higher speeds. Custom processors were popular in the 1970's and 1980's thanks to bit-slice CPU components. Chips for data paths and controllers, such as the AMD 2910 series, could be combined and microprogrammed to implement a wide range of instruction sets. Today, custom integrated circuits and FPGAs offer complete freedom to designers who are willing to put the effort into creating a CPU architecture for their application. Custom CPUs are often known as **application-specific instruction processors** (**ASIPs**) or **configurable processors**.

why automate CPU design

Custom CPU design is an area that cries out for methodological and tool support. System designers need help determining what sorts of modifications to the CPU are fruitful. They also need help implementing those modifications. Today, system designers have a wide range of tools available to help them design their own processors.

axes of customization

We can customize processors in many different ways:

- **Instruction sets** can be adapted to the application.
  - — New instructions can provide compound sets of existing operations, such as multiply-accumulate.
  - — Instructions can supply new operations, such as primitives for Viterbi encoding or block motion estimation.
  - — Instructions that operate on non-standard operand sizes can be added to avoid masking and reduce energy consumption.
  - — Instructions not important to the application can be removed.
- **Pipelines** can be specialized to take into account the characteristics of function units used in new instructions, implement specialized branch prediction schemes, etc.
- **Memory hierarchy** can be modified by adding and removing cache levels, choosing a cache configuration, or choosing the banking scheme in a partitioned memory system.
- **Busses and peripherals** can be selected and optimized to meet bandwidth and I/O requirements.

software tools

ASIPs require customized versions of the tool chains that software developers have come to rely upon. Compilers, assemblers, linkers, debuggers, simulators, and IDEs

(integrated development environments) must all be modified to match the CPU characteristics.

tool support

Tools to support customized CPU design come in two major varieties. **Configuration tools** take the microarchitecture as a specification—instruction set, pipeline, memory hierarchy, etc.—as a specification and create the logic design of the CPU (usually as register-transfer Verilog or VHDL) along with the compiler and other tools for the CPU. **Architecture optimization tools** help the designer select a particular instruction set and microarchitecture based upon application characteristics.

early work

The MIMOLA system [Mar84] is an early example of both architecture optimization and configuration. MIMOLA analyzed application programs to determine opportunities for new instructions. It then generated the structure of the CPU hardware and generated code for the application program for which the CPU was designed.

in this section

We will defer our discussion of compilers for custom CPUs until the next chapter. In this section we will concentrate on architecture optimization and configuration.

2.9.1 Configurable Processors

CPU configuration spans a wide range of approaches. Relatively simple generator tools can create simple adjustments to CPUs. Complex synthesis systems can implement a large design space of microarchitectures from relatively simple specifications.

Figure 2-47 shows a typical design flow for CPU configuration. The system designer may specify instruction set extensions as well as other system parameters like cache configuration. The system may also accept designs for function units that will be plugged into the processor. Although it is possible to synthesize the microarchitecture from scratch, the CPU's internal structure is often built around a processor model that defines some basic characteristics of the generated processor. The configuration process includes several steps, including allocation of the datapath and control, memory system design, and I/O and bus design. Configuration results in both the CPU logic design and software tools (compiler, assembler, etc.) for the processor. Configurable processors are generally created in register-transfer form and used as soft IP. Standard register-transfer synthesis tools can be used to create a set of masks or FPGA configuration for the processor.

Several processor configuration systems have been created over the years by both academic and industrial teams. ASIP Meister [Kob03] is the follow-on system to PEAS. It generates Harvard architecture machines based upon estimations of area, delay, and power consumption during architecture design and micro-operation specification. The ARC family of configurable cores (http://www.arc.com) include the 600 series with a 5-stage pipeline and the 700 series with a 7-stage pipeline.

The next example describes a commercial processor configuration system.

Application Example 2-10

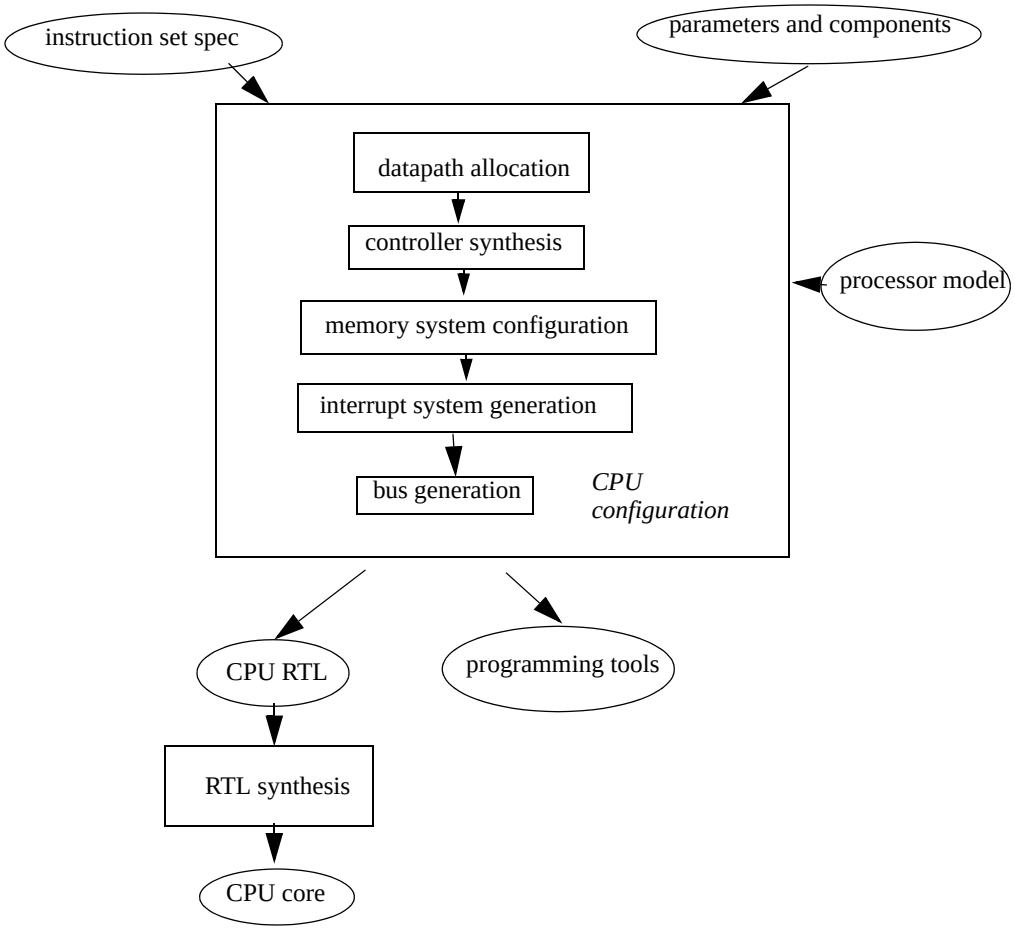


Figure 2-47

The CPU configuration process.

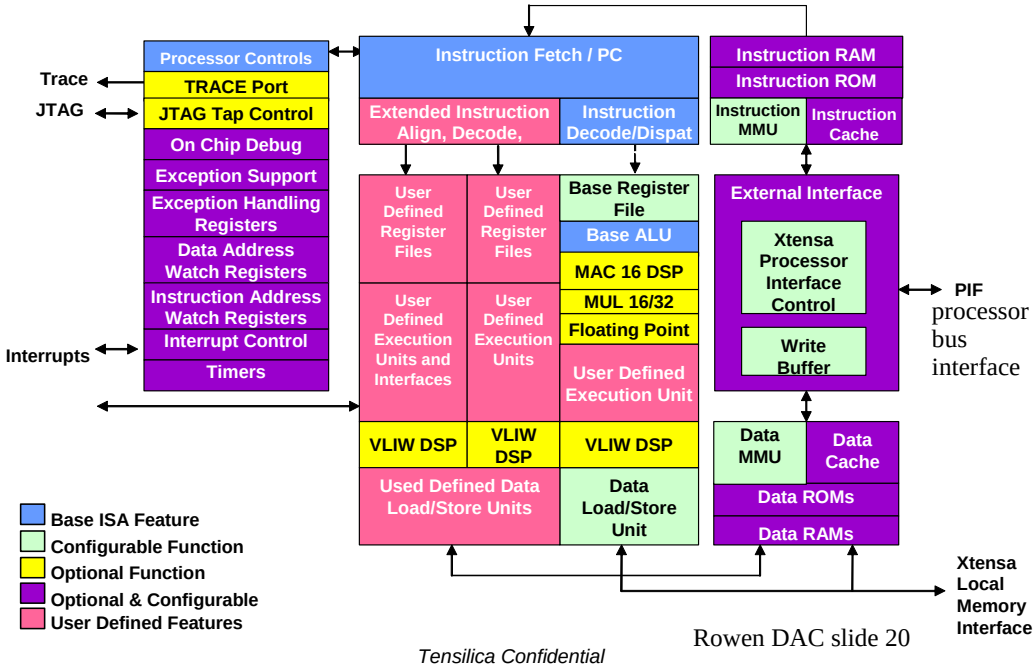
The Tensilica Xtensa Configurable Processor

The Tensilica Xtensa(TM) configurable processor is designed to allow a wide range of CPUs to be designed from a very simple specification. An Xtensa core can be customized in many ways:

- The instruction set can be augmented with basic ALU-style operations, wide instructions, DSP-style instructions, or co-processors.
- The configuration of the caches can be controlled, memory protection and translation can be configured, DMA access can be added, and addresses can be mapped into special-purpose memories.
- The CPU bus width, protocol, system registers, and scan chain can be optimized.

— Interrupts, exceptions, remote debugging features, and standard I/O devices such as timers can be added.

This figure [Row05] illustrates the range of features in the CPU that can be customized:



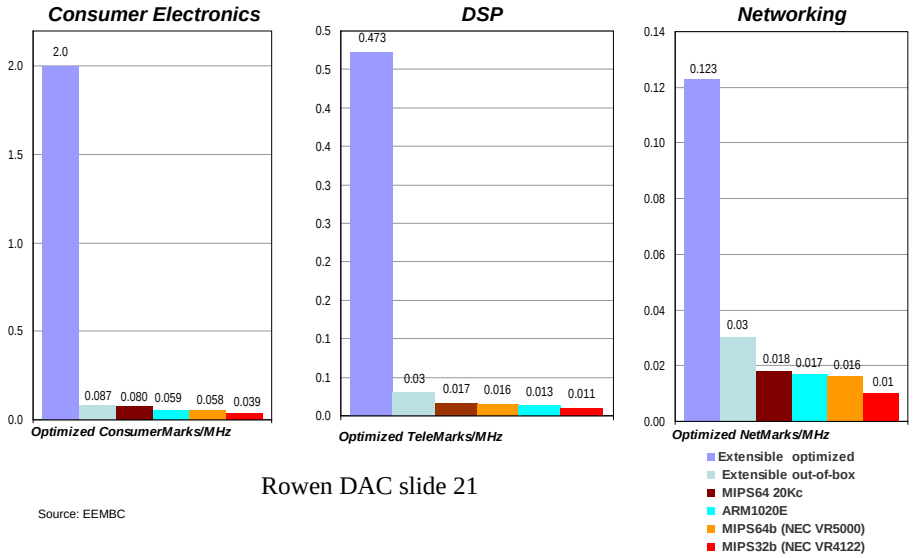
Instructions are specified using the TIE language. TIE allows the designer to declare an instruction using state declarations, instruction encodings and formats, and operation descriptions. For example, consider this simple TIE instruction specification (after Rowen):

```
Regfile LR 16 128 l
Operation add128
    { out LR sr, in LR ss, in LR st } {}
    { assign sr = st + ss; }
```

The Regfile declaration defines a large register file named LR with 16 entries, each 128 bits wide. The add128 instruction description starts with a declaration of the arguments to the instruction; each of the arguments is declared to be in the LR register file. It then defines the instruction's operation, which adds two elements of the LR register file and assigns it to a third register in LR.

New instructions may be used in programs with intrinsic calls that map onto instructions. For example, the code out[i] = add128(a[i], b[i]) makes use of the new instruction. Optimizing compilers may also map onto the new instructions.

EEMBC compared several processors on benchmarks for consumer electronics, digital signal processing, and networking. These results [Row05] show that custom, configurable processors can provide much higher performance than standard processors:



Rowen DAC slide 21

Source: EEMBC

In order to evaluate the utility of configuration, Tensilica created customized processors for four different benchmark programs:

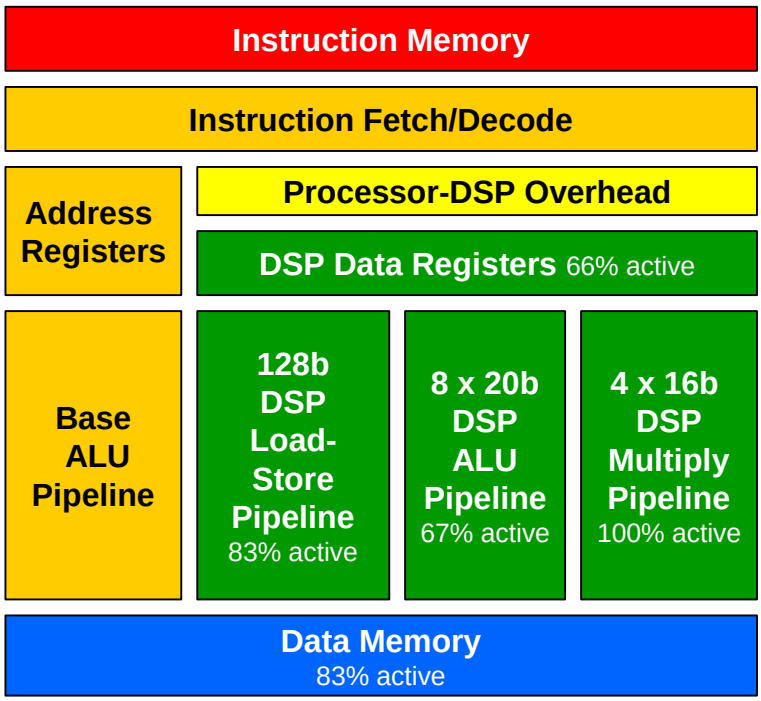
- DotProd: dot product of two 2048-element vectors.
- AES: Advanced Encryption Standard.
- Viterbi: Viterbi trellis decoder.
- FFT: 256-point fast Fourier transform.

A different CPU was designed for each benchmark. The CPUs were implemented, measured, and compared to a baseline Xtensa processor without extensions. The performance, power, and energy consumption of the processors show [Row05] that configuring customized processors can provide large energy savings:

Config	Metric	DotProd	AES	Viterbi	FFT
Reference Processor	Area (mm^2)	0.9	0.4	0.5	0.4
	Cycles (K)	12	283	280	326
	Power (mW/MHz)	0.3	0.2	0.2	0.2
	Energy (μJ)	3.3	61.1	65.7	56.6
Optimized Processor	Area (mm^2)	1.3	0.8	0.6	0.6
	Cycles (K)	5.9	2.8	7.6	13.8
	Power (mW/MHz)	0.3	0.3	0.3	0.2
	Energy (μJ)	1.6	0.7	2.0	2.5
Energy Improvement		2	82	33	22

Rowen DAC slide 17

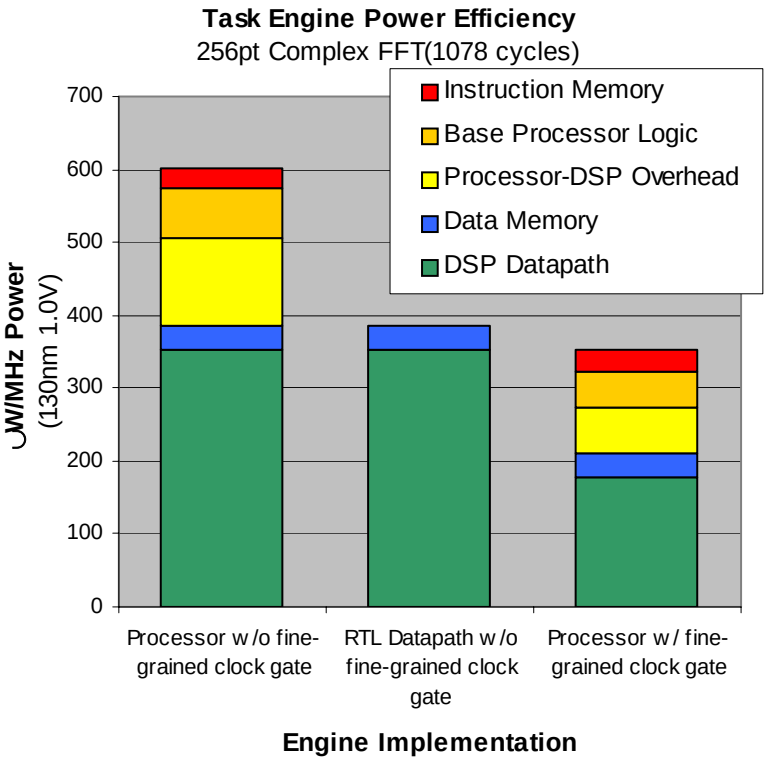
The design of a processor for a 256-point FFT computation illustrates how different types of customizations contribute to processor efficiency. Here is the architecture for the processor [Row05]:



Rowen DAC slide 27

When we analyze the energy consumption of the subsystems in the processor, we find [Row05] that fine-grained clock gating contributed substantially to energy efficiency, followed by a reduction in processor-DSP overhead:

Rowen DAC slide 27

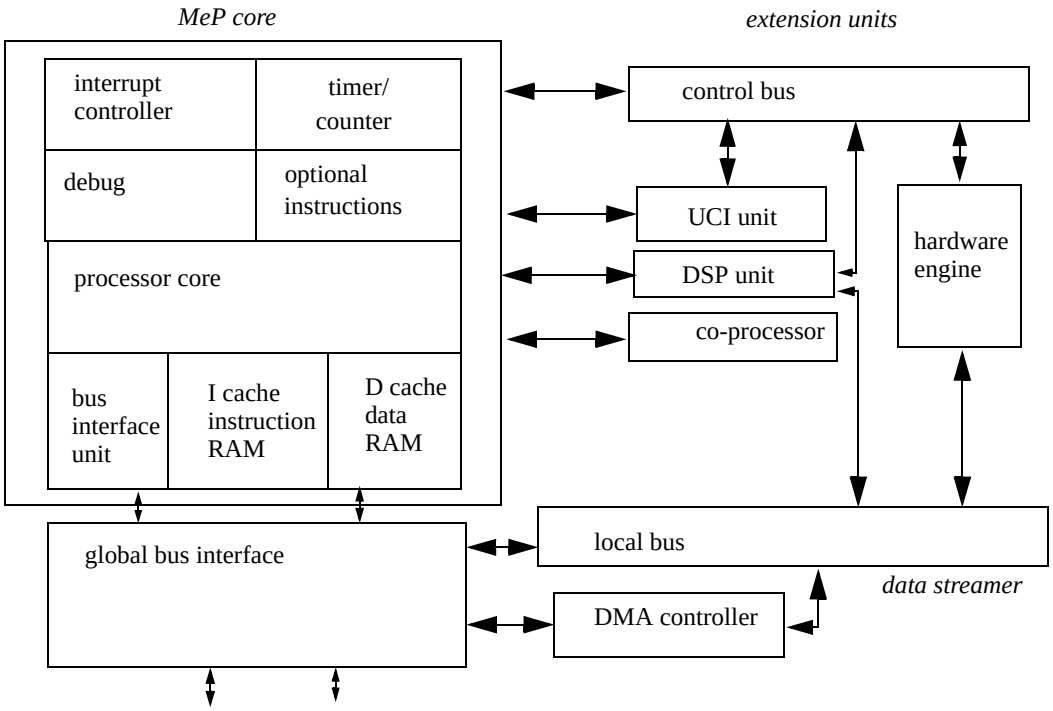


The next example describes a configurable processor designed for media processing applications.

Application Example 2-11

The Toshiba MeP Core

The MeP module [Tos05] is optimized for media processing and streaming applications. A MeP module can contain a MeP core, extension units, a data streamer, and a global bus interface unit:



The MeP core is a 32-bit RISC processor. In addition to typical RISC features, the core can be augmented with optional instructions.

Extension units can be used for further enhancements. The user-custom instruction (UCI) unit adds single-cycle instructions while the DSP unit provides multi-cycle instructions. The co-processor unit can be used to implement VLIW or other complex extensions.

The data streamer provides DMA-controlled access to memory for algorithms that require regular memory access patterns. The MeP architecture uses a hierarchy of busses to feed data to the various execution units.

CPU modeling

Let's now look in more detail at models for CPU microarchitectures and how they are used to generate CPU configurations.

LISA

The LISA system [Hof01] generates ASIPs described in the LISA language. The language mixes structural and behavioral elements to capture the processor microarchitecture.

Figure 2-48 shows example descriptions in the LISA language. The memory model is an extended version of the traditional programming model; in addition to the CPU reg-

```
RESOURCE {
    PROGRAM_COUNTER int PC;
    REGISTER signed int R[0..7];
    DATA_MEMORY signed int RAM[0..255];
    PROGRAM_MEMORY unsigned int ROM[0..255];
    PIPELINE ppu_pipe = {FI; ID; EX; WB};
    PIPELINE_REGISTER IN ppu_pipe {
        bit[6] Opcode; short operandA; short operandB;
    };
}
```

memory model

```
RESOURCE {
    REGISTER unsigned int R([0..7])6;
    DATA_MEMORY signed int RAM([0..15]);
};

OPERATION NEG_RM {
    BEHAVIOR USES (IN R[] OUT RAM[];) {
        RAM[address] = (-1) * R[index];
    }
}
```

resource model

```
OPERATION COMPARE_IMM {
    DECLARE { LABEL index; GROUP src1, dest = {register}; }
    CODING {0b10011 index =0bx[5] src1 dest }
    SYNTAX  { "CMP" src1 ~"," index ~"," dest }
    SEMANTICS { CMP (dest,src1,index) }
}

OPERATION register {
    DECLARE { LABEL index; }
    CODING { index = 0bx[4] }
    EXPRESSION { R[index] }
}
```

instruction set model

```
OPERATION ADD {
    DECLARE { GROUP src1, src2, dest = {register}; }
    CODING { 0b10010 src1 src2 dest }
    BEHAVIOR { dest = src1 + src2; saturate(&dest); }
};
```

behavioral model

Figure 2-48

Sample LISA modeling code [Hof01].

isters, it also specifies other memories in the system. The resource model describes hardware resources as well as constraints on the usage of those resources. The USES clause inside an OPERATION specifies what resources are used by that operation. The instruction set model describes the assembly syntax, instruction coding, and the function of instructions. The behavioral model is used to generate the simulator; it relates hardware structures to the operations they perform. Timing information comes from several parts of the model: the PIPELINE declaration in the resource section gives the structure of the pipeline; the IN keyword as part of an OPERATION statement assigns operations to pipeline stages; the ACTIVATION keyword in the OPERATION section launches other operations performed during the instruction. In addition, an ENTITY statement allows operations to be grouped together into functional units, such as an ALU made from several arithmetic and logical operators.

LISA hardware generation

LISA generates VHDL for the processor as a hierarchy of entities. The memory, register, and pipeline are the top-level entities. Each pipeline stage is an entity used as a component of the pipeline, while stage components like ALUs are described as entities. Groupings of operations into functional units are implemented as VHDL entities.

LISA generates VHDL for only some of the processor, leaving some entities to be implemented by hand. Some of the processor components must be carefully coded to ensure that register-transfer and physical synthesis will provide acceptable power and timing. LISA generates HDL code for the top-level entities (pipeline/registers/memory), the instruction decoder, and the pipeline decoder.

PEAS/ASIP Meister

PEAS-III [Ito00,Sas01] synthesizes a processor based upon five types of description from the designer:

- Architectural parameters for number of pipeline stages, number of branch delay slots, etc.
- Declaration of function units to be used to implement micro-operations.
- Instruction format definitions.
- Definitions of interrupt conditions and timing.
- Descriptions of instructions and interrupts in terms of micro-operations.

PEAS pipeline structure

Figure 2-49 shows the model used by PEAS-III for a single pipeline stage. The datapath portion of a stage can include one or more function units that implement operations; a function unit may take one or more clock cycles to complete. Each stage has its own controller that determines how the function units are used and when data moves forward. The datapath and controller both have registers that present their results to the next stage.

A pipeline stage controller may be in either the *valid* or *invalid* state. A stage may become invalid because of interrupts or other disruptions to the input of the instruction flow. A stage may also become invalid due to a multi-cycle operation, a branch, or other disruptions during the middle of instruction operation.

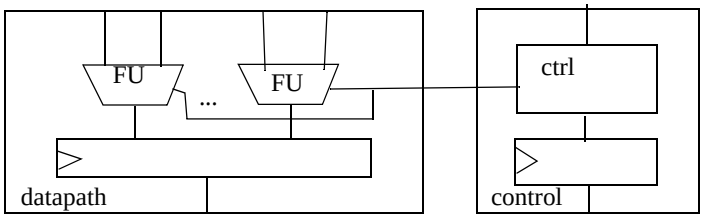


Figure 2-49

PEAS-III model of a pipeline stage.

A pipeline model for the entire processor is built by concatenating several pipeline stages. The stages may also connect to other resources on both the datapath or controller side. Datapath stages may be connected to memory or caches. Controller stages may be connected to an interrupt controller that coordinates activity during exceptions.

PEAS hardware synthesis

PEAS-III generates two VHDL models, one for simulation and another for synthesis. The datapath is generated in three phases. First, the structures required for each instruction are generated independently: the function unit resources, the ports on the resources, and the connections between those resources. The resource sets for the instructions are then merged. Finally, multiplexers and pipeline registers are added to control access to resources. After the datapath stages are synthesized, the controllers can be generated. The controllers are synthesized in three stages: the control signals required for the datapath multiplexers and registers are generated; the interlocks for multi-cycle operations are then generated; the branch control logic is then synthesized. The interrupt controller is synthesized based upon the specifications for the allowed interrupts.

2.9.2 Instruction Set Synthesis

Instruction set synthesis designs the instruction set to be implemented by the microarchitecture. This topic has not received as much attention as one might think. Many researchers in the 1970's studied instruction set design for high-level languages. That work, however, tended to take the language as the starting point, not particular programs. Instruction set synthesis requires, on the one hand, designers who are willing to create instructions that occupy a fairly small set of the static program size. This approach is justified when that small static code set is executed many times to create a large static trace. Instruction set synthesis also requires the ability to automatically generate a CPU implementation, which was not practical in the 1970's. CPU implementation requires practical logic synthesis at a minimum as well as the CPU microarchitecture synthesis tools that we have studied earlier in this section.

instruction set design space

An experiment by Sun et al. [Sun04] demonstrates the size and complexity of the instruction set design space. They studied a *BYTESWAP()* program that swaps the order

[Sun04] Fig. 2

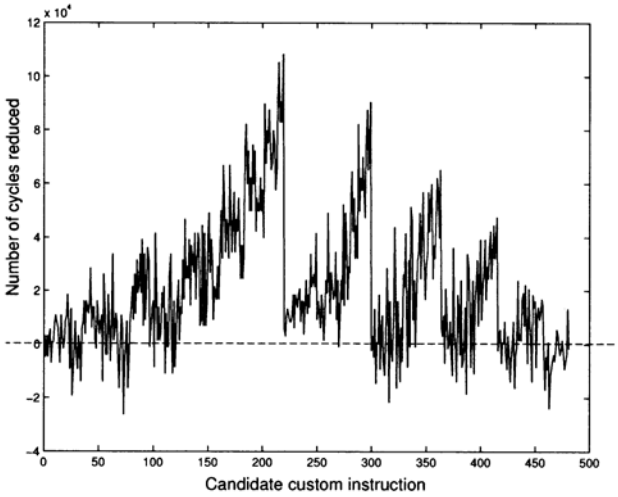


Figure 2-50

The instruction set design space for a small program [Sun04]

of bytes in a word. They generated all possible instructions for this program—they found 482 possible instructions.Figure 2-50 shows the execution time for the program with each possible instruction; the instructions are ordered arbitrarily across the x axis. Even in this simple program, different instructions result in very different performance results.

instruction set metrics

Holmer and Despain [Hol91] formulated instruction set synthesis as an optimization problem, which requires selecting an optimization function to guide the optimization process. They observed that, when designing instruction sets manually, computer architects often apply a **1% rule**—an instruction that provides less than a 1% improvement in performance over the benchmark set is not a good candidate for inclusion in the instruction set. They proposed this performance-oriented objective function:

$$100\ln C + I, \qquad \text{(EQ 2-21)}$$

where C is the number of cycles used to execute the benchmark set and I is the total number of instructions in the instruction set. The logarithm is the infinitesimal form of $\Delta C / C$ and the I term provides some benefit for adding a few high-benefit instructions over many low-benefit instructions. They also proposed an objective function one that incorporates code size:

$$100\ln C + 20\ln S + I, \qquad \text{(EQ 2-22)}$$

where S is the static number of instructions. This form imposes a 5% rule for code size improvements.

instruction formation

Holmer and Despain identified candidate instructions using methods similar to the microcode compaction algorithms used to schedule microoperations. They compiled a benchmark program into a set of primitive microoperations. They then use a branch-and-bound algorithm to combine microoperations into candidate instructions. Combinations of microoperations are then grouped into instructions.

instruction set search algorithms

Huang and Despain [Hua95] also used an *n*% rule as a criterion for instruction selection. They proposed the use of simulated annealing to search the instruction set design space. Given a set of microoperations that can be implemented in the datapath, they use move operators to generate combinations of microoperations. A move may displace a microoperation to a different time step, exchange two microoperations, insert a time step, or delete a time step. A move must be evaluated not only for performance but also whether they violate design constraints, such as resource utilization.

template generation

Kastner et al [Kas02] use clustering to generate instruction templates and cover the program. Covering is necessary to ensure that the entire program can be implemented with instructions. Clustering finds subgraphs that occur frequently in the program graph and replaces those subgraphs with supernodes that correspond to new instructions.

[Ata03] Fig 1

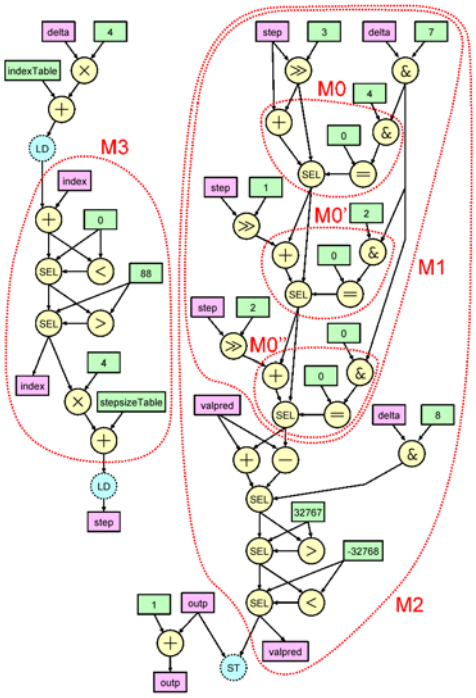


Figure 2-51

Candidate instructions in the adpcmdecode benchmark [Ata03].

Atasu et al. [Ata03] developed algorithms to find complex instructions. Figure 2-51 shows an operator graph from a section of the *adpcmdecode* benchmark. Although the

[Bis05] Fig 1

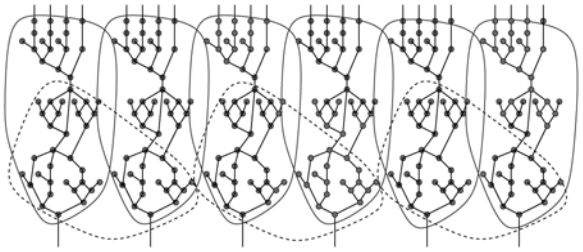


Figure 2-52

Instruction template size vs. utilization [Bis05].

M2 graph is large, the operators within it are fairly small; the entire M2 subgraph implements a 16x3-bit multiplication, which is a good candidate for encapsulation in an instruction. Atasu et al. also argue that combining several disjoint graphs, such as M2 and M3, into a single instruction is advantageous. Disjoint operations can be performed in parallel and so offer significant speedups. They also argue that multi-output operations are important candidates for specialized instructions.

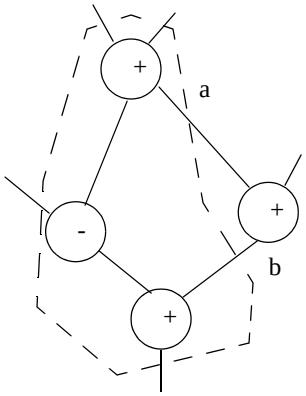


Figure 2-53

A non-convex operator graph.

A large operator graph must be convex to be mapped into an instruction. The graph identified by the dotted line in Figure 2-53 is not convex: input *b* depends upon output *a*. In this case, the instruction would have to stall and wait for *b* to be produced before it could finish.

Atasu et al. find large subgraphs in the operator graph that maximize the speedup provided by the instruction. By covering the graph with existing instructions, we can count the number of cycles required to execute the graph without the new instruction.

We can estimate the number of clock cycles required to execute the new instruction by fast logic synthesis that provides the critical path length, which we can then compare to the available cycle time. They use a branch-and-bound algorithm to identify cuts in the operator graph that define new instruction subgraphs.

Biswas et al [Bis05] use a version of the Kernighan-Lin partitioning algorithm to find instructions. They point out that finding the maximum-size instruction does not always result in the best result. In the example of Figure 2-52, the largest template, shown with the dotted line, can be used only three times in the computation, but the smaller graph shown with the solid line can be used six times.

instructions in combination

Sun et al. [Sun04] developed an instruction set synthesis system that used the Tensilica Xtensa system to implement their choices. Their system generates TIE code that can be used to synthesize processors. They generate instructions from programs by combining microinstructions. They synthesize the register-transfer hardware for each candidate instruction, then synthesize logic and layout for that hardware to evaluate its performance and area. They select a subset of all possible instructions for further evaluation based upon their speedup and area potential. Based upon this set of candidate instructions, they select a combination of instructions used to augment the processor instruction set. They use a branch-and-bound algorithm to identify a combination of instructions that minimizes area while satisfying the performance goal.

large instructions

Pozzi and Ienne developed algorithms to extract large operations as instructions. Larger combinations of microoperations provide greater speedups for many signal processing algorithms. Because large blocks may require many memory accesses, they developed algorithms that generate multi-cycle operations from large data flow graphs [Poz05]. They identify mappings that require more memory ports than are available in the register file and add pipelining registers and sequencing to perform the operations across multiple cycles.

The next example describes a recent industrial instruction set synthesis system.

Application Example 2-12

The Tensilica Xpres Compiler

The Xpres compiler [Ten04] designs instruction sets from benchmark programs. It creates TIE code and processor configurations that provide the optimizations selected from the benchmarks. Xpres looks for several types of optimized instructions:

- **Operator fusion** creates new instructions out of combinations of primitive microoperations.
- Vector/SIMD operations perform the same operation on subwords that are 2, 4, or 8 wide.

- Flix operations combine independent operations into a single instruction.
- Specialized operations may limit the source or destination registers or other operands. These specializations provide a tighter encoding for the operation that can be used to pack several operations into a single instruction.

The Xpres compiler searches the design space to identify instructions to be added to the architecture. It also allows the user to guide the search process.

---

limited-precision arithmetic

A related problem is the design of limited-precision arithmetic units for digital signal processing. Floating-point arithmetic provides high accuracy across a wide dynamic range but at a considerable cost in area, power, and performance. In many cases, if the range of possible values can be determined, finite-precision arithmetic units can be used. Mahlke et al. [Mah01] extended the PICO system to synthesize variable-bitwidth architectures. They used rules, designer input, and loop analysis to determine the required bit-

[Mah01] fig 11

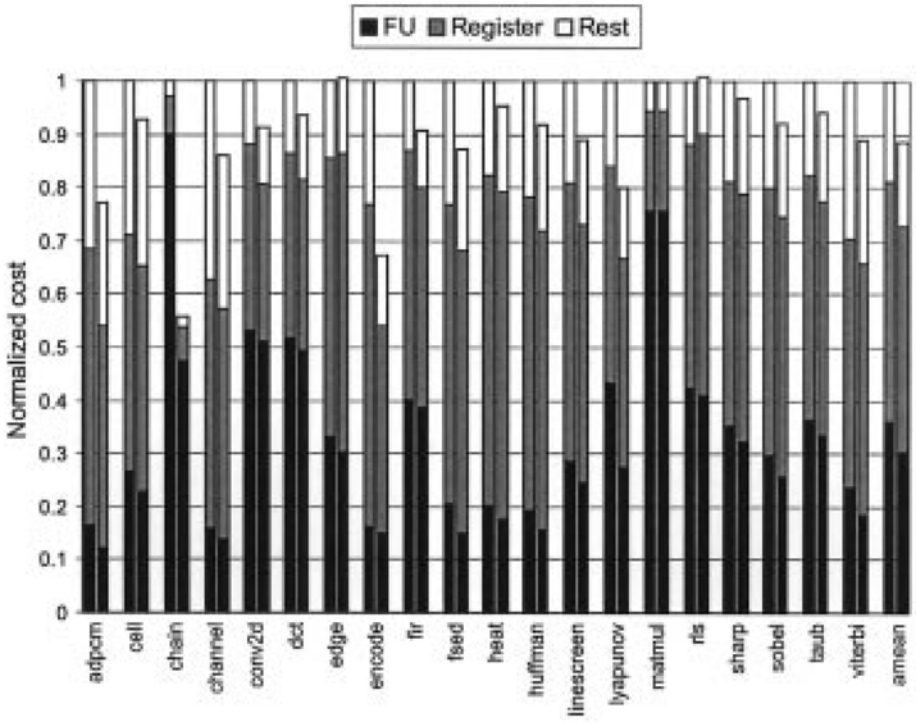


Figure 2-54

Cost of bitwidth clustering for multi-function units [Mah01].

width of variables. They used def-use analysis to analyze the propagation of bitwidths. They clustered operations together to find a small number of distinct bitwidths to implement the required accuracy with a small number of distinct units. They found that bitwidth clustering was particularly effective when operations could be mapped onto multifunction units. The results of their synthesis experiments for a number of benchmarks are shown in Figure 2-54. The right bar shows hardware cost for bitwidth analysis alone, while the left bar shows hardware cost after bitwidth analysis and clustering. Each bar divides hardware cost into registers, function units, and other logic.

The traditional way to determine the dynamic range of an algorithm is by simulation, which requires careful design of the input data set as well as long run times. Fang et al. [Fan03] used affine arithmetic to analyze the numerical characteristics of algorithms. Affine arithmetic models the range of a variable as a linear equation. Terms in the affine model can describe the correlation between the ranges of other variables; accurate analysis of correlations allows the dynamic range of variables to be tightly bounded.

2.10 Summary

CPUs are at the heart of embedded computing. CPUs may be selected from a catalog for use or they may be custom-designed for the task at hand. A variety of architectural techniques are available to optimize CPUs for performance, power consumption, and cost; these techniques can be combined in a number of ways. Processors may be designed by hand; a variety of analysis and optimization techniques have been developed to help designers customize processors.

What We Learned

- RISC and DSP approaches can be used in embedded CPUs. The design trade-offs for embedded processors lead to some different conclusions than are typical for general-purpose processors.
- A variety of parallel execution methods can be used; they must be matched to the available parallelism.
- Encoding algorithms can be used to efficiently represent signals and instructions in processors.
- Embedded processors are prone to many attacks that are not realistic in desktop or server systems.
- CPU simulation is an important tool for both processor design and software optimization. Several techniques, varying in accuracy and speed, can be used to simulate performance and energy consumption.

Further Reading

Conte [Con92] describes both CPU simulation and its uses in computer design. The SOS Research Web site (http://www.sosresearch.org/caale/caalesimulators.html) provides extensive information about CPU simulators. The chapter by Rotenberg and Anantararaman [Rot04] provides an excellent introduction to embedded CPU architectures. Witold Kinsner has created a good on-line introduction to smart cards at http://www.ee.umanitoba.ca/~kinsner/whatsnew/tutorials/tu1999/smcards.html. This chapter has made reference to United States Patents; all U.S. Patents are available online at http://www.uspto.gov.

Questions

Q2-1 Draw a pipeline diagram for a multiply instruction followed by an addition. Compare to the pipeline diagram for a multiply-accumulate instruction that requires two clock cycles to execute. Use pipelines of length:

a. three stages;

b. five stages;

c. eight stages.

Q2-2 Compare and contrast performing a matrix multiplication using subword parallel instructions and vector instructions. How do the code fragments for the two approaches differ? How do these differences affect performance?

Q2-3 Build a model for a two-way set-associative cache. Show the block-level organization of the cache. Create area, delay, and energy models for the cache based upon the formulas for the block-level model.

Q2-4 Evaluate cache configurations for several block motion estimation algorithms. The motion estimation search uses a 16 x 16 macroblock and a search area of 25 x 25. Each pixel is eight bits wide. Consider full search, three-step, and four-step search. The cache size is fixed at 4096 bytes. Evaluate direct-mapped, 2-way, and 4-way set associative caches at three different line widths: four bytes, eight bytes, and 16 bytes. Compute the cache miss rate for each case.

Q2-5 Draw a pipeline diagram for a processor that uses a code decompression engine. Not counting decode, the pipeline includes four stages. Assume that the code decompression engine requires four cycles for decoding. Show the execution of an addition followed by a branch.

Q2-6 Evaluate the energy savings of bus-invert coding on a data bus for the address sequences to the array a[10][10], where a starts at address 100s:

a. row-major sequential accesses;

b. column-major sequential access to a;

c. diagonal accesses such as those used for JPEG DCT encoding (0,0 -> 1,0 -> 0,1 -> 2,0 -> 1,1 -> 0,2 -> ...).

Q2-7 Identify possible instructions in matrix multiplication.

Q2-8 Identify possible instructions in the fast Fourier transform.

Lab Exercises

L2-1 Develop a SimpleScalar model for a DSP with a Harvard architecture and multiply-accumulate instruction.

L2-2 Use your SimpleScalar model to compare performance of a matrix multiplication routine with and without the multiply-accumulate instruction.

L2-3 Use simulation tools to analyze the effects of register file size on performance.

L2-4 Develop a SimpleScalar model for a code decompression engine. Evaluate CPU performance as the decompression time varies from one to 10 cycles.

Chapter

8

Networks

- General network architectures and the ISO network layers.
- Automotive and aircraft networks.
- Consumer electronics networks.
- Sensor networks.

8.1 Introduction

In this chapter we study **networks** that can be used to build **distributed embedded systems.** In a distributed embedded system, several **processing elements** (either microprocessors or ASICs) are connected by a network that allows them to communicate. The application is distributed over the processing elements, and some of the work is done at each node in the network.

There are several reasons to build network-based embedded systems. When the processing tasks are physically distributed, it may be necessary to put some of the computing power near where the events occur. Consider, for example, an automobile: The short time delays required for tasks such as engine control generally mean that at least parts of the task are done physically close to the engine. Data reduction is another important reason for distributed processing. It may be possible to perform some initial signal processing on captured data to reduce its volume—for example, detecting a certain type of event in a sampled data stream. Reducing the data on a separate processor may significantly reduce the load on the processor that makes use of that data. Modularity is another motivation for network-based design. For instance, when a large system is assembled out of existing components, those components may use a network port as a clean interface that does not interfere with the internal operation of the component in ways that using the microprocessor bus would. A distributed system can also be easier to debug—the microprocessors in one part of the network can be used to probe components in another part of the network. Finally, in some cases, networks are used to build fault tolerance into sys-

tems. Distributed embedded system design is another example of hardware/software co-design, since we must design the network topology as well as the software running on the network nodes.

Of course, the microprocessor bus is a simple type of network. However, we use the term *network* to mean an interconnection scheme that does not provide shared memory communication. In the next section, we develop the basic principles of hardware and software architectures for networks. Section 8.3 looks at automotive and aircraft data networks. Section 8.4 considers consumer electronics networks. Section 8.5 focuses on sensor networks.

8.2 Networking Principles

In this section we will consider network abstractions, then move onto the structure of the Internet stack.

8.2.1 Network Abstractions

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. In order to help understand (and design) networks, the International Standards Organization (ISO) has developed a seven-layer model for networks known as Open Systems Interconnection (OSI) models [Sta97A]. Understanding the OSI layers will help us to understand the details of real networks.

The seven layers of the **OSI model,** shown in Figure 8-1, are intended to cover a broad spectrum of networks and their uses. Some networks may not need the services of one or more layers because the higher layers may be totally missing or an intermediate layer may not be necessary. However, any data network should fit into the OSI model. The OSI layers from lowest to highest level of abstraction are described below.

- **Physical:** The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires), electrical properties, 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.
- **Network:** 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.

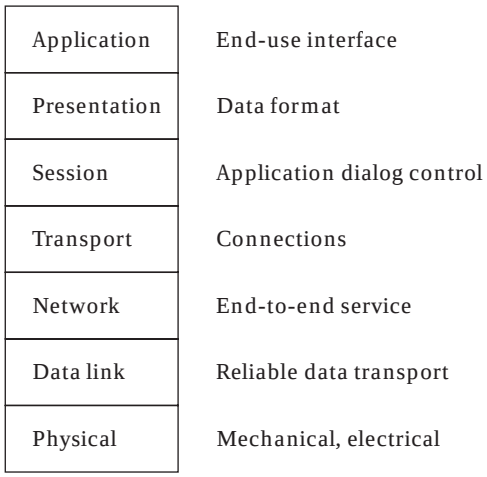


Figure 8-1 *The OSI model layers.*

- **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 end-user programs.

Although it may seem that embedded systems would be too simple to require use of the OSI model, the model is in fact quite useful. Even relatively simple embedded networks provide physical, data link, and network services. An increasing number of embedded systems provide Internet service that requires implementing the full range of functions in the OSI model.

8.2.2 Internet

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.

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 an Ethernet. In general, an Internet packet will travel 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 Figure 8-2. IP works at the network 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. IP creates packets for routing to the destination, which are then sent to the *data link* and *physical* 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 since it is not running applications, it does not need to go to higher levels of the OSI model. In general, a packet may go through several routers to get to its destination. At the destination, the IP layer provides data to the transport layer and ultimately 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.

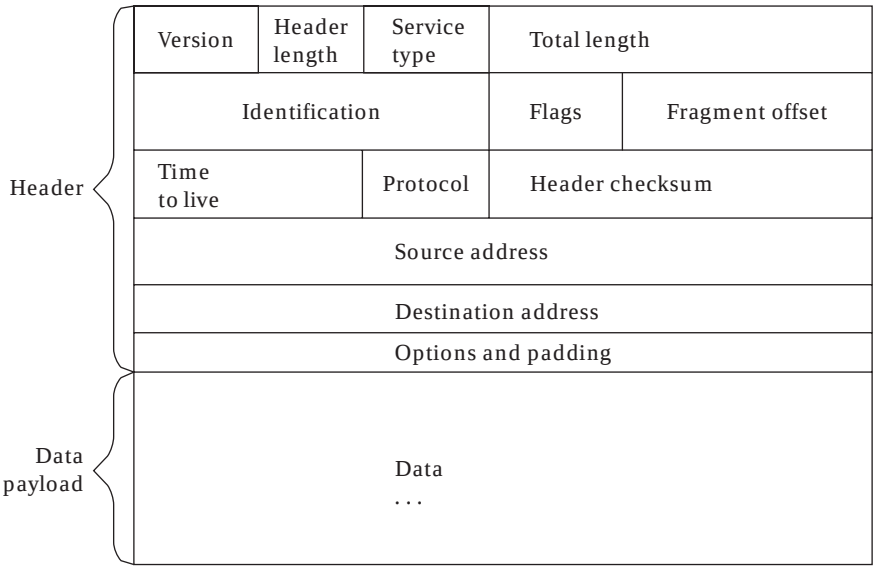


Figure 8-3 *IP packet structure.*

The basic format of an IP packet is shown in Figure 8-3. The header and data payload are both of variable 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.xx.xx.xx. The names by which users and applications typically refer to Internet nodes, such as foo.baz.com, are translated into IP

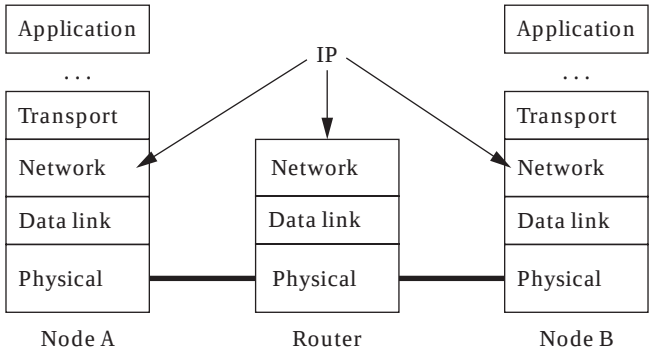


Figure 8-2 *Protocol utilization in Internet communication.*

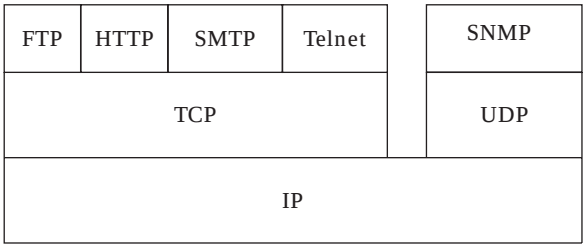


Figure 8-4 *The Internet service stack.*

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.** Since routes for data may change quickly with subsequent packets being routed along very different paths with different delays, 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. Since the performance of the Internet may depend on worldwide usage patterns, its real-time performance is inherently harder to predict.

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.

Figure 8-4 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 E-mail, and Telnet for virtual terminals. A separate transport protocol, **User Datagram Protocol (UDP),** is used as the basis for the network management services provided by the **Simple Network Management Protocol (SNMP).**

8.3 Networks for Real-Time Control

Real-time control is one of the major applications of embedded computing. Machines like automobiles and airplanes require control systems that are physically distributed around the vehicle. Networks have been designed specifically to meet the needs of real-time distributed control for automobile electronics and avionics.

We will start with an analysis of the characteristics of automotive and aeronautical electronic control systems. We will then describe the CAN bus, an early and popular bus designed for real-time automotive electronics. We will then describe a newer automotive bus known as FlexRay. We will conclude with a discussion of avionics networks.

8.3.1 Real-Time Vehicle Control

safety-critical systems

The basic fact that drives the design of control systems for vehicles is that they are safety-critical systems. Errors of any kind—component failure, design flaws, etc.—can injure or kill people. Not only must these systems be carefully verified, but they must be architected to guarantee certain properties.

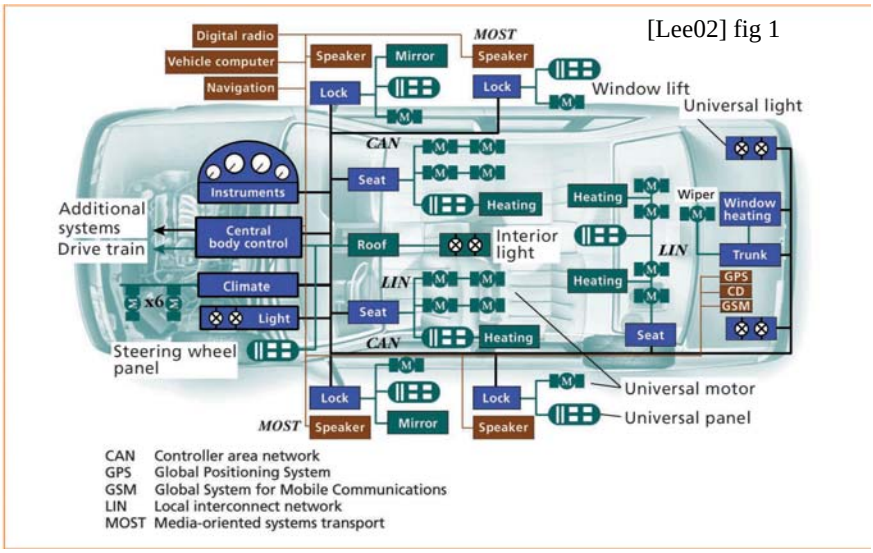


Figure 8-5

Electronic devices in modern automobiles [Lee02].

microprocessors and automobiles

As shown in Figure 8-5, modern automobiles use a number of electronic devices [Lee02] Today's low-end cars often include 40 microprocessors while high-end cars can contain 100 microprocessors. These devices are generally organized into several networks. The critical control systems, such as engine and brake control, may be on one network while non-critical functions, such as entertainment devices, may be on a separate network.

harnesses vs. networks

Until the advent of digital electronics, care generally used point-to-point wiring organized into **harnesses**, which are bundles of wires. Connecting devices into a shared network saves a great deal of weight—15 kilograms or more [Lee02]. Networks require somewhat more complicated devices that include network access hardware and software, but that overhead is relatively small and swanking over time thanks to Moore's Law.

specialized automotive networks

But why not use general-purpose networks like Ethernet? We can find reasons to build specialized automotive networks at several levels of abstraction in the network stack. One reason is electrical—automotive networks require reliable signaling under vary harsh environments. The ignition systems of automobile engines generate huge amounts of electromagnetic interference that can render many networks useless. Automobiles must also operate under wide temperature ranges and survive large doses of moisture.

Real-time control also requires guaranteed behavior from the network. Many communications networks do not provide hard real-time requirements. Communication systems are also more tolerant of latency than are control systems. While data or voice communications may be useful when the network introduces transmission delays of hundreds of milliseconds or even seconds, long latencies can easily cause disastrous oscillations in real-time control systems. Automotive networks must also operate within limited power budgets that may not apply to communications networks.

X-by-wire

Control systems have traditionally relied on mechanics or hydraulics to implement feedback and reaction. Microprocessors allow us to use hardware and software not just to sense and actuate but to implement the control laws. In general, the controller may not be physically close to the device being controlled: the controller may operate several different devices or it may be physically shielded from dangerous operating areas. Electronic control of critical functions was first performed in aircraft where the technique was known as **fly-by-wire**. Control operations that are performed over the network are called **X-by-wire** where X may be brake, steer, etc.

non-control uses

Powerful embedded devices—television systems, navigation systems, Internet access, etc.—are being introduced into cars. These devices do not perform real-time control but they can eat up large amounts of bandwidth and require real-time service for streaming data. Since we can only expect the amount of data being transmitted within a car to increase, automotive networks must be designed to be future-proof and handle workloads that are even more challenging than what we see today.

avionics

Aviation electronics systems developed in parallel to automotive electronics but are now starting to converge. Avionics must be certified for use in aircraft by governmental authorities (the Federal Aviation Administration in the United States), which means that devices for aircraft are often designed specifically for aviation use. The fact that aviation systems are certified has made it easier to use electronics for critical operations like the operation of flight control surfaces (ailerons, rudder, elevator). Airplane cockpits are also highly automated. Some commercial airplanes already provide Internet access to passengers; we expect to see such services become in cars over the next decade.

8.3.2 The CAN Bus

The **CAN bus** uses bit-serial transmission. CAN can run at rates of 1 Mb/second over a twisted pair connection of 40 meters. An optical link can also be used. The bus protocol supports multiple masters on the bus. Many of the details of the CAN and I^2C buses are similar, but there are also significant differences.

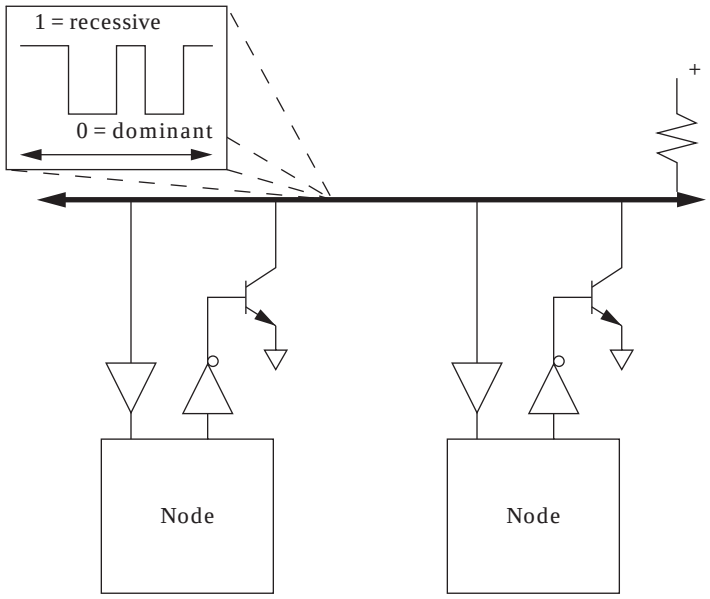


Figure 8-6 *Physical and electrical organization of a CAN bus.*

physical layer

As shown in Figure 8-6, each node in the CAN bus has its own electrical drivers and receivers that connect the node to the bus in wired-AND fashion. In CAN terminology, a logical 1 on the bus is called **recessive** and a logical 0 is **dominant.** 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 (making 0 dominant over 1). When all nodes are transmitting 1s, the bus is said to be in the recessive state; when a node transmits a 0, 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 to 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 themselves against later transitions in each frame.

data frame

The format of a CAN data frame is shown in Figure 8-7. A data frame starts with a 1 and ends with a string of seven zeroes. (There are at least three bit fields between data frames.) The first field in the packet contains the packet's destination address and 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 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 64 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 iden-

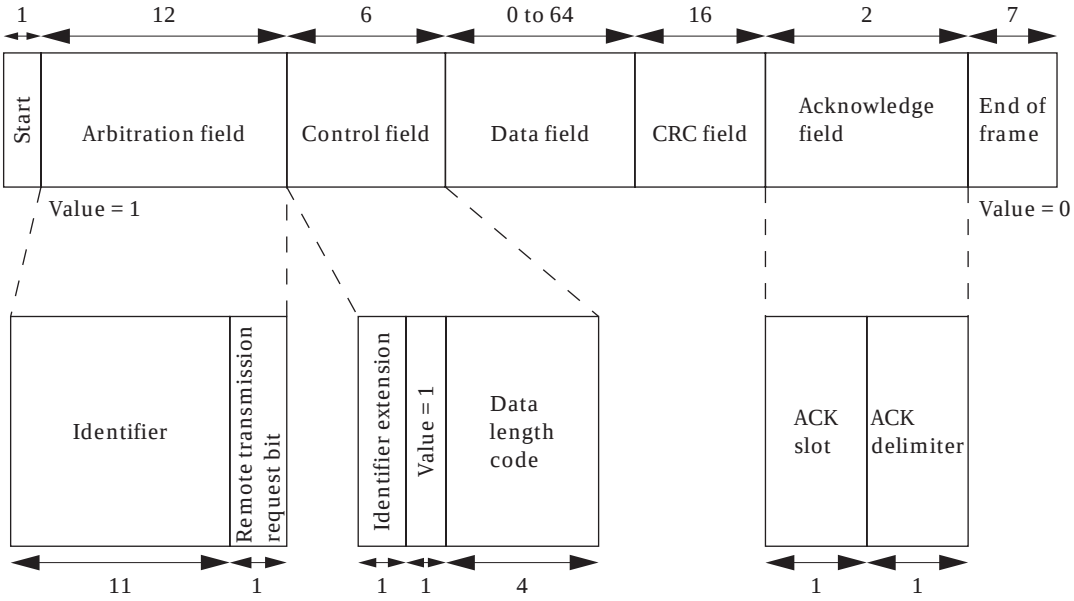


Figure 8-7 *The CAN data frame format.*

tifier signal whether the frame was correctly received: The sender puts a recessive bit (1) in the ACK slot of the acknowledge field; if the receiver detected 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 that 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). (As seen in Section 8.3.3, Ethernet uses CSMA without AMP.) This method is similar to the I^2C bus's arbitration method; like I^2C, CAN encourages a data-push programming style. Network nodes transmit synchronously, so they all start sending their identifier fields at the same time. When a node hears a dominant bit in the identifier when it 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 0 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 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.

error handling

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

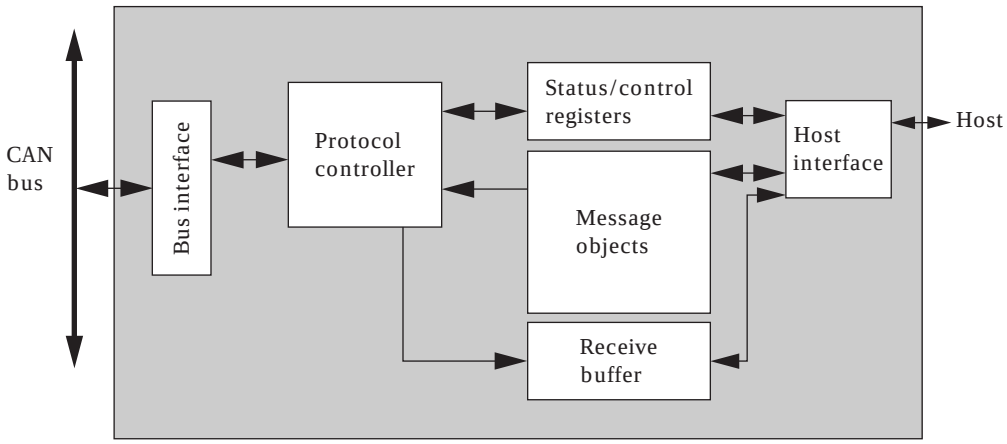


Figure 8-8 *Architecture of a CAN controller.*

consists of an error flag field followed by an error delimiter field of 8 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 for a data frame, it should retransmit the data frame until the data is acknowledged. This action corresponds to the data link layer in the OSI model.

Figure 8-8 shows the basic architecture of a typical CAN controller. The controller implements the physical and data link layers; since CAN is a bus, it does not need network layer services to establish end-to-end connections. The protocol control block is responsible for determining when to send messages, when a message must be resent due to arbitration losses, and when a message should be received.

8.3.3 FlexRay

FlexRay (http://www.flexray.com) is a second-generation standard for automotive networks. It is designed to provide higher bandwidth as well as more abstract services than are provided by CAN.

block diagram

Figure 8-9 shows the block diagram of a generic FlexRay system. The host runs applications. It talks to both communication controllers, which provide higher-level functions, and the low-level bus drivers.

bus guardians

A node that watches the operation of a network and takes action when it sees erroneous behavior is known as a **bus guardian** (whether or not the network is actually a bus). FlexRay uses bus guardians to check for errors on active stars.

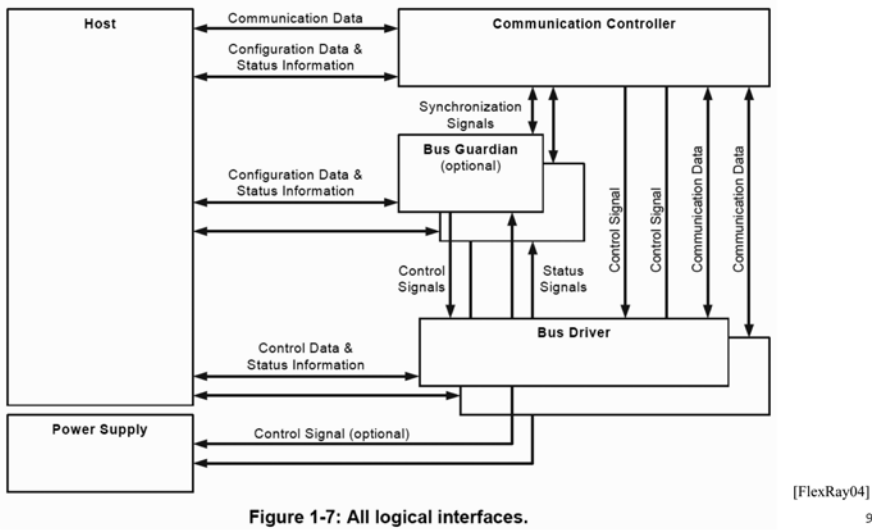


Figure 8-9

FlexRay block diagram[Flex04]

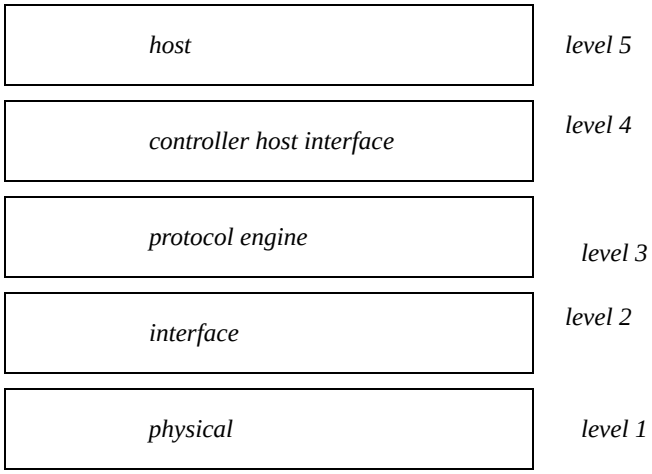


Figure 8-10

Levels of abstraction in FlexRay.

FlexRay timing

Because FlexRay is designed for real-time control, it provides network scheduling phases that guarantee real-time performance. This mode is known as the **static phase** because the scheduling of frames is chosen statically. It also provides a mode, known as **dynamic phase**, for non-time-critical and aperiodic data that will not interfere with the static mode. The transmissions in the static phase have guaranteed bandwidth and the dynamic phase messages cannot interfere with the static phase. This method creates a **temporal firewall** between time-sensitive and non-time-sensitive transmissions.

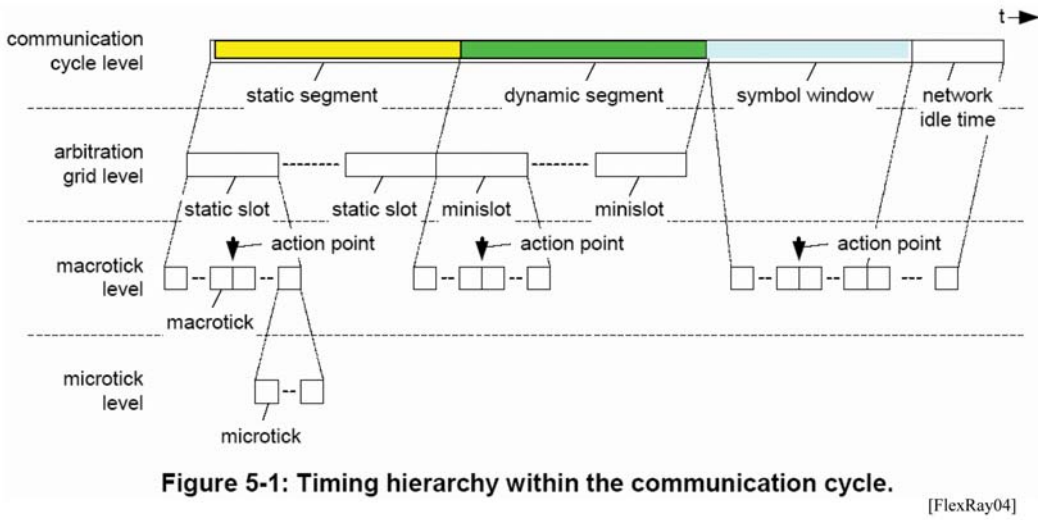


Figure 5-1: Timing hierarchy within the communication cycle.

[FlexRay04]

Figure 8-11

FlexRay timing [Flex04].

Figure 8-11 illustrates the hierarchy of timing structures used by FlexRay. Larger timing phases are built up from smaller timing elements. Starting from the lowest level of the hierarchy:

- A **microtick** is derived from the node's own internal clock or timer, not from the global FlexRay clock.

- A **macrotick**, in contrast, is derived from a cluster-wide synchronized clock. A macrotick always includes an integral number of microticks but different macroticks may contain different numbers of microticks to correct for differences between the nodes' local clocks. The boundaries between some macroticks are designated as **action points**, which form the boundaries for static and dynamic segments.

- The **arbitration grid** determines the boundaries between messages within a static or dynamic segment. An arbitration algorithm determines what nodes will be allowed to transmit in the slots determined by the action points.

- A **communication cycle** includes four basic elements: A static segment, a dynamic segment; a symbol window, and network idle time. A symbol window is a single unarbitrated time slot for application use. The idle time allows for timing corrections and housekeeping functions at the nodes.

FlexRay network stack

As shown in Figure 8-10, FlexRay is organized around five levels of abstraction. The topology level defines the structure of connections. The interface level defines the physical connections. The protocol engine defines frame formats and communication modes and services such as messages and synchronization. The controller host interface provides information on status, configuration, messages, and control for the host layer.

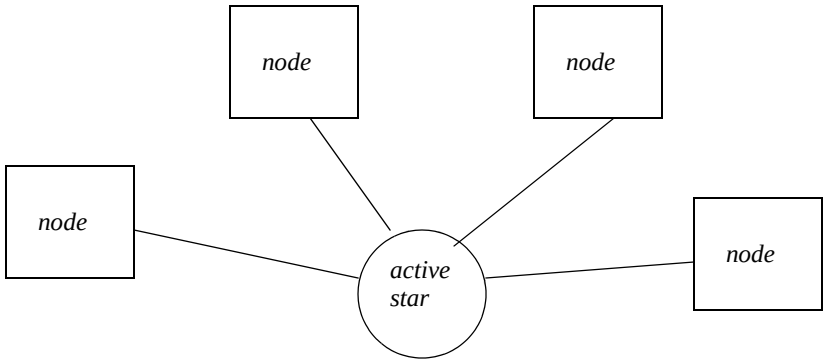


Figure 8-12

Active star networks.

active stars

As shown in Figure 8-12, FlexRay is not organized around a bus. It instead uses a star topology known as an **active star** because the router node is active. The maximum delay between two nodes in an active star is 250 ns. As a result, the active star does not store the complete message before forwarding it to the destination.

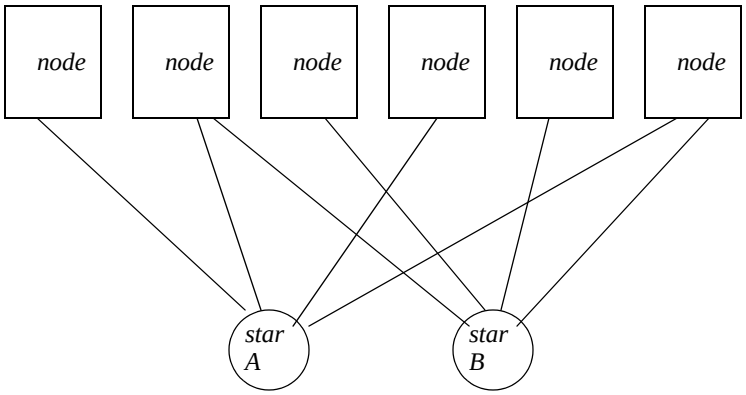


Figure 8-13

Redundant active stars.

redundant active stars

A node may be connected to more than one star to provide redundant connections in case one active star fails. In Figure 8-13, some of the nodes are connected to both stars A and B while other nodes are connected to only one of the stars.

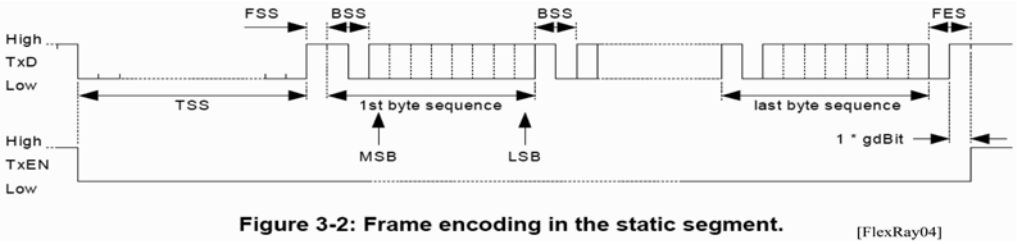


Figure 8-14

FlexRay frame encoding [Flex04].

physical layer

FlexRay transmits bits over links using differential non-return-to-zero (NRZ) coding as shown in Figure 8-15.A low-power idle phase operates at zero volts. The idle phase transmits a mid-range voltage and bits are modulated around that value. The links transmit at 10 Mbits/sec independent of the length of the link. FlexRay does not arbitrate on bits so link length is not limited by arbitration contention.

Figure 8-14 shows the encoding of static frames, the basic frame type. Data is sent as bytes. TSS stands for **transmission start sequence**, which is low for 5 to 15 bits. FSS

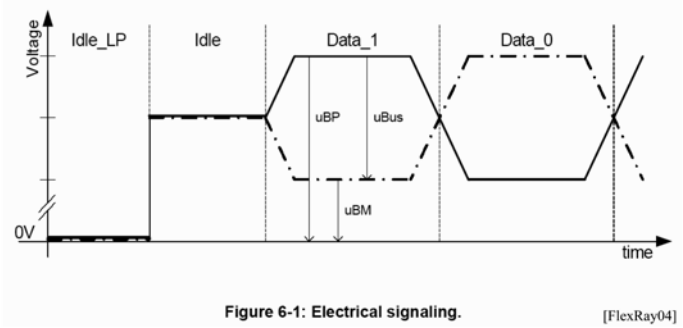


Figure 8-15

FlexRay data transmission.

starts for **frame start sequence**, which is one high bit. BSS stands for **byte start sequence**. FES stands for **frame end sequence**, which is a LO followed by a HI. Dynamic frames, which we will describe more fully in a moment, add a **dynamic trailing sequence field**.

frame fields

Figure 8-16 shows the format of a FlexRay frame:

- The **frame ID** identifies the frame's slot. Its value is in the range 0..2047.
- The **payload length** field gives the number of 16 bit words in the payload section. All messages in the static section of a communication cycle must use the same length payload.
- The **header CRC** provides error correction.
- The **cycle count** enumerates the protocol cycles. This information is used within the protocol engines to guide clock synchronization.
- The **data field** provides from 0 to 254 bytes long. As mentioned before, all packets in the static segment must provide the same length of data payload.
- The **trailer CRC** field provides additional error correction.

static segments

The static segment is the basic timing structure for time-critical messages. Figure 8-17 shows the organization of a FlexRay static segment. The static segment is scheduled using a time-division multiple access discipline—this allows the system to ensure that messages get a guaranteed amount of bandwidth. The TDMA discipline divides the static segment into slots of fixed and equal length. All the slots are used in every segment in the same order. The number of slots in the static segment is configurable in the range 0..1023.

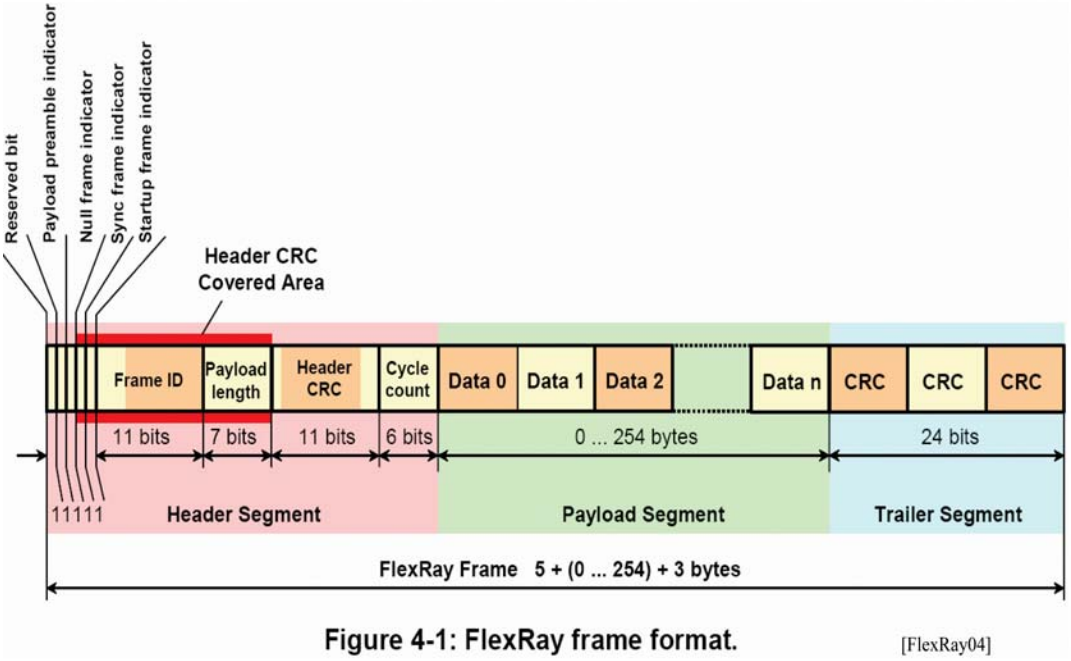


Figure 8-16

Format of a FlexRay frame [Flex04]

The static segment is split across two channels. Synchronization frames are provided on both channels. Messages may be sent on either one or both channels; less critical nodes may be designed to connect to only one channel. The slots are occupied by messages with ascending frame ID numbers. The slot numbers are not used for timing but are instead used by software to identify messages. (The message ID in the payload area can also be used to identify messages.)

dynamic segments

The dynamic segments provide bandwidth for asynchronous, unpredictable communication. The slots in the dynamic segment are arbitrated using a deterministic mechanism. Figure 8-18 shows the organization of a dynamic segment. The dynamic segment has two channels, each of which can have its own message queue.

Figure 8-19 illustrates the timing of a dynamic segment. Messages can be sent at minislot boundaries. If no message is sent for a minislot, it elapses as a short idle message. If a message is sent, it occupies a longer interval than a minislot. As a result, transmitters must watch for messages to know whether each minislot was occupied.

The frame ID is used to number slots. The first dynamic frame's number is one higher than the last static segment's number. Messages are sent in order of frame ID, with the lowest number first. The frame ID number acts as the message priority. Each frame ID number can send only one message per dynamic segment. If there are too many

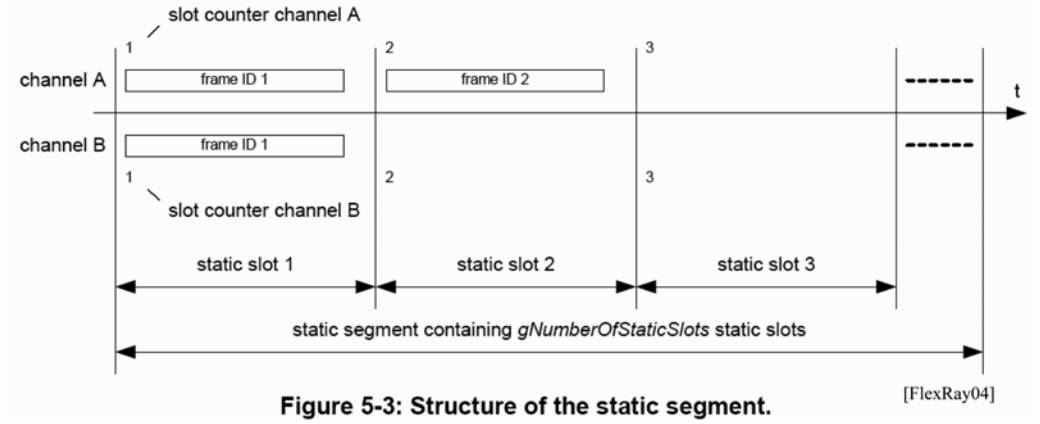


Figure 8-17

A flexRay static segment.

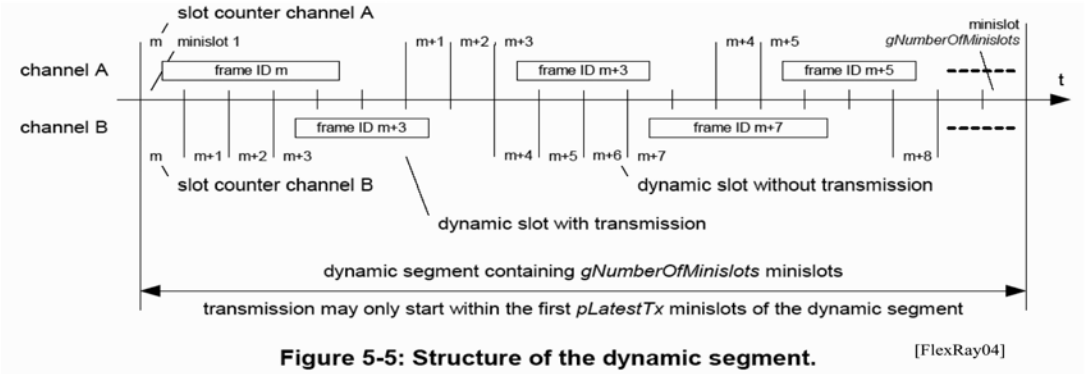


Figure 8-18

Structure of a FlexRay dynamic segment [Flex04].

messages in the queue to be sent in a single dynamic segment, those messages are carried over to the next dynamic segment.

system startup

A network with complex timing like FlexRay must be started properly. FlexRay starts with a wakeup procedure that turns on the nodes. It then performs a **coldstart** that initiates the TDMA process. At least two nodes in the system must be designated as capable of performing a coldstart. The network sends a wakeup pattern on one channel to alert the nodes to wake up. The wakeup procedure has been designed to easily detect collisions between nodes that may wake up and try to transmit simultaneously.

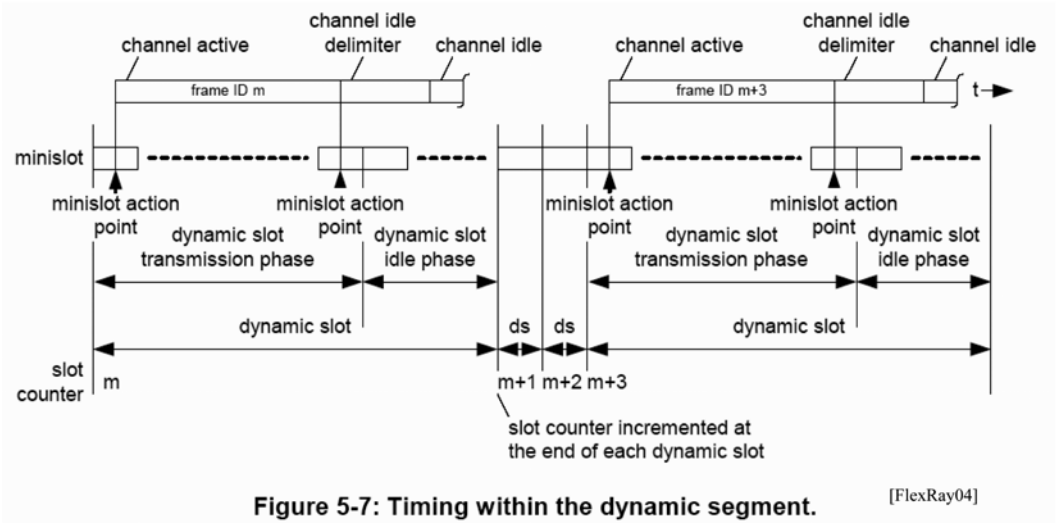


Figure 8-19

FlexRay dynamic segment timing [Flex04].

timekeeping

A TDMA network like FlexRay needs a global time source to synchronize messages. The global time is synthesized by the **clock synchronization process** (**CSP**) from the nodes' clocks using distributed timekeeping algorithms. The global time is used to determine the boundaries of macroticks, which are the basic system timekeeping unit. Macroticks are managed by the **macrotick generation process** (**MTG**), which applies the clock and any updates provided by the CSP.

Figure 8-20 illustrates the FlexRay clock synchronization process. The CSP periodically measures clocks, then applies a correction.

bus guardians

The bus guardians' role is to prevent nodes from transmitting outside their schedules. FlexRay does not require a system to have a bus guardian but they should be included in any safety-critical system. The bus guardian sends an enable signal to every node in the system that it guards. It can stop the node from transmitting by removing the enable signal. The bus guardian uses its own clock to watch the bus operation. If it sees a message coming at the wrong time, it removes the enable signal from the node that is sending the message.

controller host interface

The controller host interface (CHI) provides services to the host. Some services are required while others are optional. These services are provided in hardware. Services include status (macroticks, etc.), control (interrupt service, startup, etc.), message data (buffering, etc.), and configuration (node and cluster).

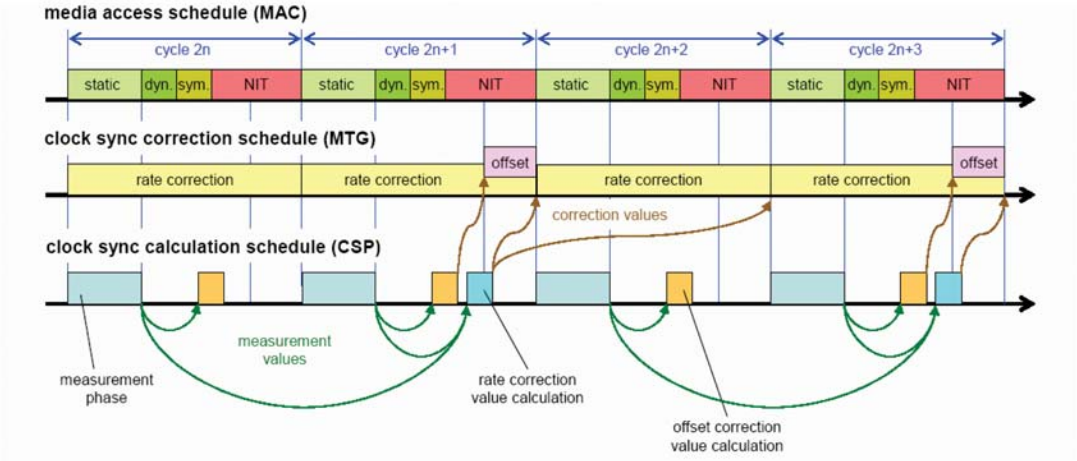


Figure 8-3: Timing relationship between clock synchronization, media access schedule, and the execution of clock synchronization functions.

[FlexRay04]

Figure 8-20

The FlexRay clock synchronization procedure [Flex04].

8.3.4 Aircraft Networks

avionics

Aircraft design is similar in some respects to automobile design, but with more stringent requirements. Aircraft are more sensitive to weight than are cars. Aircraft have more complex controls because they must be flown in three dimensions. And most aspects of aircraft design, operation and maintenance are regulated.

Aircraft electronics may be divided into roughly three categories: **instrumentation**, **navigation/communication**, and **control**. Aircraft instruments, such as the altimeter or artificial horizon, use mechanical, pneumatic, or hydraulic methods to sense aircraft characteristics. The primary role of electronics is to sense these systems and display the results or send them onto other systems. Navigation and communication relies on radios. Aircraft use several different types of radios because regulations mandate that certain activities be performed with different types of radios. Communication may be by voice or data. Navigation makes use of several techniques; moving maps that integrate navigation data onto a map display are common even in general aviation aircraft. Digital electronics can be used to both control the radios, such as set frequencies, and to present the output of these radios. Control systems operate the engines and flight control surfaces (aileron, elevator, rudder).

aircraft network categories

Because of these varied uses, modern commercial aircraft use several different types of networks:

- **cockpit networks** These networks must perform hard real-time tasks for instrumentation and control. They generally operate in TDMA mode.
- **airframe networks** These networks control non-critical devices. They may use non-guaranteed modes such as Ethernet to improve average performance and limit weight.
- **passenger networks** Some airplanes now offer Internet service to passengers through wired or wireless connections. Internet traffic is routed through a satellite link. These networks make use of existing standards and are separated from the aircraft's operating networks by firewalls.

aircraft network standards

A number of standards for aircraft data networks have been developed. Several of these standards have been developed by Aircraft Radio, Inc., (ARINC), which was chartered by the U. S. Congress to coordinate radio communications for U. S. airlines. Standards include: ARINC 429; ARINC 629; CDSB; ARINC 573 for flight data recorders; and ARINC 708 for weather radar data.

The next example looks at ARINC 429.

Application Example 8-1

ARINC 429

ARINC 492 was developed in the 1970s for emerging digital avionics communications. Because this standard was defined fairly early in the history of data communications, the standard concentrates on the physical and data link layers.

A key design problem in aircraft avionics is lightning. Lightning does not often strike aircraft, but it can happen and a lightning strike must not disable the airplane's electronics. The entire aircraft is designed to handle lightning strikes, but the network must be designed to carefully protect against lightning.

ARINC 429 uses differential signaling. A logic 1 signal is +10V while a logic 0 is -10V. Bits are coded as return to zero, with the null state at 0V. The low-speed variant of ARINC 429 operates at 13 kbits/sec. A high-speed variation of the standard operates at 100 kbits/sec.

ARINC 429 is a single-source network. A network may have several data sinks but only one source of data. If an installation requires multiple transmitters, a separate 429 bus must be used for each transmitter.

Communication in 429 is asynchronous. Data packets must have four null time slots between them to allow the beginning of packets to be recognized. Each data word has 32 bits. The first 8 bits of a word are used as a **label**. The ARINC 429 standard defines a

number of labels for various uses, such as frequency, heading, acknowledgment, etc. The word has a 18 bit data payload.

8.4 Consumer Networks

Consumer electronics devices may be connected into networks to make them easier to use and to provide access to audio and video data across the home. This section will look at some networks used in consumer devices, then consider the challenges of integrating networks into consumer devices.

8.4.1 Bluetooth

personal area networks

Bluetooth is a **personal area network**—designed to connect devices in close proximity to a person. The Bluetooth radio operates in the 2.5 GHz spectrum. Its wireless links typically operate within 2 meters, although advanced antennas can extend that range to 30 meters. A Bluetooth network can have one master and up to 7 active slaves; more slaves can be parked for a total of 255. Although its low-level communication mechanisms do require master-slave synchronization, the higher levels of Bluetooth protocols generally operate as a peer-to-peer network, without masters or slaves.

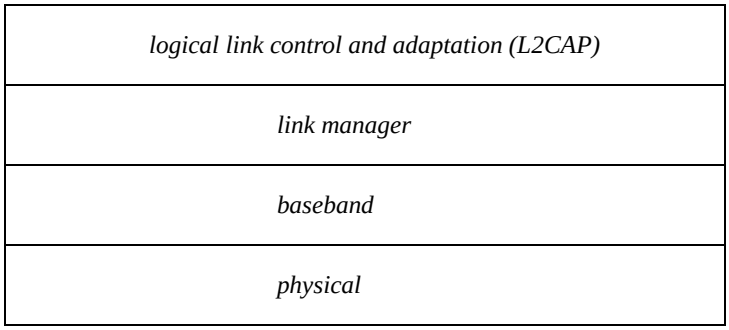


Figure 8-21

Bluetooth transport protocols.

transport group protocols

Figure 8-21 shows the **transport group protocols**, which belong to layers 1 and 2 of the OSI model:

- The physical layer provides basic radio functions.
- The **baseband** layer defines master/slave relationships and frequency hopping.

- The **link manager** provides mechanisms for negotiating link properties such as bandwidth and quality-of-service.

- The **logical link control and adaptation protocol** (**L2CAP**) hides baseband layer operations like frequency hopping.

physical layer The Bluetooth radio transmits using frequency-hopping spread spectrum, which allows several radios to operate in the same frequency band without interference. The band is divided into 79 1 MHz-wide channels; the Bluetooth radio hops between these frequencies at a rate of 1,600 hops/per second. The radio's transmission power may also be controlled.

baseband layer The baseband layer chooses frequency hops according to a pseudo-random sequence that is agreed upon by the radios; it also controls the radio signal strength to ensure that the receiver can properly decode the spread spectrum signal. The baseband layer also provides medium access control, determining packet types and processing. It controls the radio power. It provides a real-time clock. It also provides basic security algorithms.

link manager The link manager builds upon the baseband layer to provide several functions. It schedules transmissions, choosing which data packets to send next. Transmission scheduling takes quality-of-service contracts into account. It manages overall power consumption. The link manager also manages security and encrypts transmissions as specified.

L2CAP layer The L2CAP layer serves as an interface between general-purpose protocols to the lower levels of the Bluetooth stack. It primarily provides asynchronous transmissions. It also allows higher layers to exchange QoS information.

middleware group protocols The **middleware group protocols** provide several widely-used protocols. It provides a serial port abstraction for general-purpose communication. It provides protocols to interoperate with IrDA infrared networks. It provides a service discovery protocol. And because Bluetooth is widely used for telephone headsets, it provides a telephony control protocol.

RFCOMM The Bluetooth serial interface is known as RFCOMM. It multiplexes several logical serial communications onto the radio channel. Its signals are compatible with the traditional RS-232 serial standard. It provides several enhancements, including remote status and configuration, specialized status and configuration and connection establishment and termination. RFCOMM can emulate a serial port within a single device to provide seamless data service whether the ends of the link are on the same processor or not.

service discovery protocol The service discovery protocol allows a Bluetooth client device to determine whether a server device in the network provides a particular service. Services are defined by **service records**, which consists of a set of <ID,value> attributes. All service records include a few basic attributes such as class and protocol stack information. A service may define its own attributes, such as capabilities. To discover a service, the client asks a server for a type of service. The server then responds with a service record.

8.4.2 WiFi

The WIFi (TM) family of standards (http://grouper.ieee.org/groups/802/11, http://www.wi-fi.org) provide wireless data communication for computers and other devices. WiFi is a family of standards known as 802.11 from the IEEE 802 committee. The original 802.11 spec was approved in 1997. An improved version of the spec, known as 802.11b, was presented in 1999. This standard used improved encoding methods to increase the standard's bandwidth. Later standards include 802.11a, which provided substantially wider bandwidth, and 802.11g, which extended 802.11b. Figure 8-22 compares the properties of these networks.

	bandwidth	*band*
802.11b	*11 Mbps*	*2.4 GHz*
802.11a	*54 Mbps*	*5 GHz*
802.11g	*802.11g*	*2.4 GHz*

Figure 8-22

802.11 specifications.

Full-duplex communication requires two radios, one for each direction. Some devices use only one radio, which means that a device cannot transmit and receive simultaneously.

8.4.3 Networked Consumer Devices

Networked consumer devices have been proposed for a variety of functions, but particularly for home entertainment. These systems have not yet entirely fulfilled their potential. A brief survey helps us understand the challenges in building such systems.

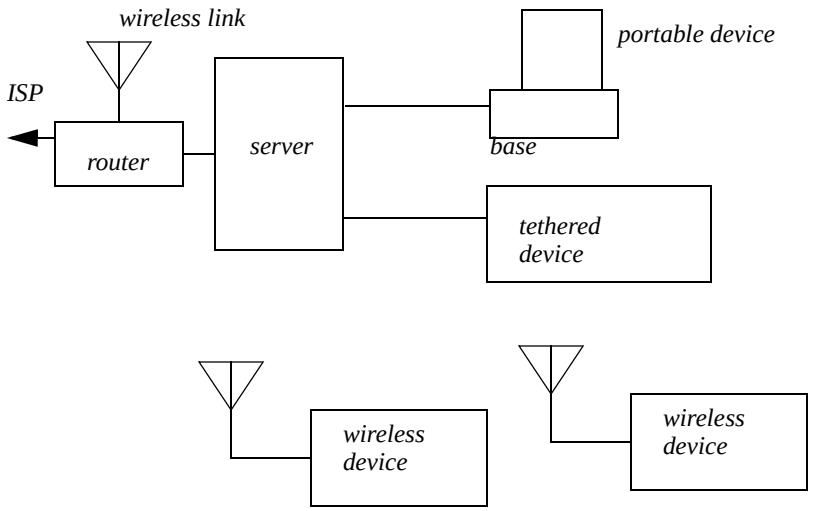


Figure 8-23

Networked home entertainment devices.

network organization

Figure 8-23 shows a typical organization of an entertainment-oriented home network:

- The PC acts as a server for file storage of music, images, movies, etc. Today's disk drives are large enough to hold substantial amounts of music or images.
- Some devices may be permanently attached to the PC. For example, the USB port may be used to send audio to an audio receiver for high-quality amplification.
- Mobile devices, such as portable music players, may dock with the PC through a base. The PC is used to manage the device.
- Other devices may connect over wireless links. They may connect to the server or to each other. For example, digital video recorders may have their own storage and may stream video to other devices.

Several companies have proposed home server devices for audio and video. Built from commodity hardware, these servers provide specialized interfaces for detailing with media. They may also include capture subsystems, such as DVD drives for reading moves from DVDs.

configuration

A key challenge in the design of home entertainment networks is configurability. Consumers do not want to spend time configuring their components for operation on their network; in many cases, the devices will not have keyboards so configuration would be very difficult. Many levels of the network hierarchy must be configured.

Clearly, physical and link-level parameters must be configured. But another important aspect of configuration is service discovery and configuration. Each device added to the network must be able to determine what other devices it can work with, what services it can get from other devices, and what services it needs to provide to the rest of the network. Service discovery protocols help devices adapt themselves to the network.

software architecture

Java has been used to develop middleware for consumer networks. Java can be efficiently executed on a number of different platforms. This not only simplifies software development but also allows devices to trade Java code to provide services.

8.5 Sensor Networks

Sensor networks are distributed systems designed to capture and process data. They typically use radio links to transmit data between themselves and to servers. Sensor networks can be used to monitor buildings, equipment, and people.

ad hoc computing

A key aspect of the design of sensor networks is the use of **ad hoc networks**. Sensor networks can be deployed in a variety of configurations and nodes may be added or removed at any time. As a result, both the network and the applications running on the sensor nodes must be designed to dynamically determine their configuration and take the necessary steps to operate under that network configuration.

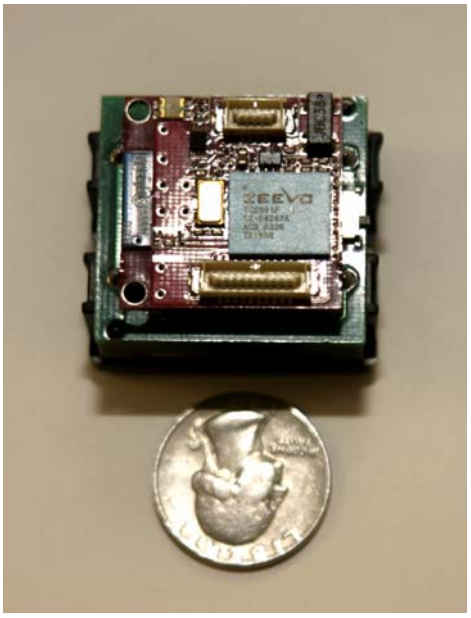
For example, when data is transmitted to a server, the nodes do not know in advance the path that data should take to arrive at the server. The nodes must provide **multi-hop routing** services to transmit data from node to node in order to arrive to the network. This problem is challenging because not all nodes are within radio range and it may take network effort and computation to determine the topology of the network.

The next two examples describe a sensor network node and its operating system.

Application Example 8-2

The Intel Mote sensor node

The Intel Mote (iMote) is a second-generation sensor network node:



draft photo

The imote uses Bluetooth as its communication link. The Zeevo chip visible in the photograph implements the Bluetooth link. That Bluetooth module includes an ARM7TDMI that is used for Bluetooth baseband functions as well as other sensor network functions. The mote includes an integrated radio antenna; an external antenna can be attached for longer range communication.

The mote provides several I/O modes so that sensing devices can be attached to the mote. I/O devices can be built onto daughter cards that stack on top of the basic mote circuit boards.

A mote may be connected to a PC using a USB slave connection. The PC can be used to monitor and control the mote network. The PC also serves as a host for code development.

Application Example 8-3

TinyOS and nesC

TinyOS (http://www.tinyos.net) is an operating system for sensor networks. It is designed to support networks and devices on a small platform, using only about 200 bytes of memory.

TinyOS code is written in a new language known as nesC. This language supports the TinyOS concurrency model based on tasks and hardware event handler. The nesC compiler detects data races at compile time. A nesC program includes one set of functions known as **events**. The program may also include functions called **commands** to help implement the program, but another component uses the events to call the program. A set of components can be assembled into a system using interface connections known as **wiring**.

TinyOS executes only one program using two threads: one containing tasks and another containing **hardware event handlers**. The tasks are scheduled by TinyOS; tasks are run to completion and do not preempt each other. Hardware event handlers are initiated by hardware interrupts. They may preempt tasks or other handlers and they run to completion.

The sensor node radio is one of the devices in the system. TinyOS provides code for packet-based communication, including multi-hop communication.

The next application describes an application of sensor networks.

Application Example 8-4

ZebraNet

ZebraNet [Jua02] is designed to record the movements of zebras in the wild. Each zebra wears a collar that includes a GPS positioning system, a network radio, a processor, and a solar cell for power. The processors periodically read the GPS position and store it in on-board memory. The collar reads position every three minutes, along with whether the zebra is in sun or shade. For three minutes every hour, the collar takes detailed readings to determine the zebra's speed. This generates about 6 kB of data per zebra per day.

Experiments show that computation is much less expensive than radio transmissions:

Table 1:

operation	current @ 3.6V
idle	< 1 mA
GPS position sampling and CPU/ storage	177 mA
base discovery only	432 mA

Table 1:

operation	current @ 3.6V
transmit data to base	1622 mA

Thus conservation of radio energy is critical.The data from the zebras is read only intermittently when biologists travel to the field. They do not want to leave behind a permanent base station, which would be hard to maintain. Instead, they bring with them a node that reads data from the network.

Because the zebras move over a wide range, not all of the zebras will be within range of the base station and it is impossible to predict which (if any) of the zebras will be in range. As a result, the ZebraNet nodes must replicate data across the network. The nodes transmit copies of their position data to each other as zebras come within range of each other. When a zebra comes within range of a base station, the base station reads all of that zebra's data, including data it has gathered from other zebras.

The ZebraNet group experimented with two data transfer protocols. One protocol—flooding—sent all data to all other available nodes. The other, history-based protocol chose one peer to send data to based upon which peer had the best past history of delivering data to the base. Simulations showed that flooding was delivered the most data for short-range radios but history-based delivered the most data for long-range radio. However, flooding consumed much more energy than history-based routing.

8.6 Summary

We often need or want to build an embedded system out of a network of connected processors rather than a single CPU with I/O devices. The uses of distributed embedded systems vary greatly, ranging from the real-time networks in an automobile to Internet-enabled information appliances. There are a great many networks that we can choose from to build embedded systems based on constraints of cost, overall throughput, and real-time behavior.

What We Learned

- The OSI layer model breaks down the structure of a network into seven layers.
- A large number of networks, many with very different characteristics, are used in embedded systems.

- Real-time networks use TDMA to guarantee delivery times.

Further Reading

Kopetz [Kop97] provides a thorough introduction to the design of distributed embedded systems. Stallings [Sta97A] provides a good introduction to data networking. Helfrick [Hel04] discusses avionics, including aircraft networks. For those interested in the principles of instrument flight, Dogan [Dog99] provides an excellent introduction.

Questions

Q8-1 Describe a FlexRay bus at the following OSI-compliant levels of detail:

a. physical.

b. data link.

c. network.

d. transport.

Q8-2 How do the requirements of aircraft and automobile networks differ? How are they similar?

Q8-3 How could you use a broadcast network like WiFi for hard real-time control? What assumptions would you have to make? What control mechanisms would need to be added to the network?

Q8-4 Plot computation vs. communication energy in a wireless network.

a. Determine the computation and communication required for a node to receive two 16-bit integers and multiply them together.

a. Plot total system energy as a function of computation vs. communication energy.

Q8-5

Lab Exercises

L8-1 Build an experimental setup that lets you monitor messages on an embedded network.

L8-2 Measure energy for a single instruction vs. transmission of a single packet for a sensor network node.