THE COMPUTER ENGINEERING HANDBOOK

Edited by
VOJIN G. OKLOBDZIJA

CRC PRESS

Boca Raton London New York Washington, D.C.

MTK 1012

Cover photos from Molecular Expressions Website (www.microscopy.fsu.edu), National High Magnetic Field Laboratory, Optical Microscopy Division, The Florida State University, Tallahassee, FL. ©1995–2001 Michael W. Davidson and The Florida State University. With permission.

Library of Congress Cataloging-in-Publication Data

The computer engineering handbook / Vojin G. Oklobdzija, editor-in-chief.
p. cm.--(Electrical engineering handbook series)
Includes bibliographical references and index.
ISBN 0-8493-0885-2 (alk. paper)
1. Computer engineering. 2. Electronic digital computers. I. Oklobdzija, Vojin G. II. Series.

TK7885 .C645 2001
004—dc21 2001043891

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the authors and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

Neither this book nor any part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, microfilming, and recording, or by any information storage or retrieval system, without prior permission in writing from the publisher.

All rights reserved. Authorization to photocopy items for internal or personal use, or the personal or internal use of specific clients, may be granted by CRC Press LLC, provided that $1.50 per page photocopied is paid directly to Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923 USA The fee code for users of the Transactional Reporting Service is ISBN 0-8493-0885-2/02/$0.00+$1.50. The fee is subject to change without notice. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

The consent of CRC Press LLC does not extend to copying for general distribution, for promotion, for creating new works, or for resale. Specific permission must be obtained in writing from CRC Press LLC for such copying.

Direct all inquiries to CRC Press LLC, 2000 N.W. Corporate Blvd., Boca Raton, Florida 33431.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation, without intent to infringe.

Visit the CRC Press Web site at www.crcpress.com

© 2002 by CRC Press LLC

No claim to original U.S. Government works
International Standard Book Number 0-8493-0885-2
Library of Congress Card Number 2001043891
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

22

Embedded Systems-on-Chips

22.1 Introduction 22-1
22.2 Requirements on Embedded SoCs 22-2
22.3 Embedded SoC Components 22-2
CPUs • Interconnect • Memory • Software Components
22.4 Embedded System Architectures 22-5
22.5 Embedded SoC Design Methodologies 22-7
Specifications • Design Flows • Platform-Based Design • Software Performance Analysis and Optimization • Energy/Power Analysis and Optimization
22.6 Summary 22-11

Wayne Wolf
Princeton University

22.1 Introduction

Advances in VLSI technology now allow us to build systems-on-chips (SoCs), also known as systems-on-silicon (SoS). SoCs are complex at all levels of abstraction; they contain hundreds of millions of transistors; they also provide sophisticated functionality, unlike earlier generations of commodity memory parts. As a result, SoCs present a major productivity challenge.

One solution to the SoC productivity problem is to use embedded computers.[1] An embedded computer is a programmable processor that is a component in a larger system that is not a general-purpose computer. Embedded computers help tame design complexity by separating (at least to some degree) hardware and software design concerns. A processor can be used as a pre-designed component—known as intellectual property (IP)—that operates at a known speed and power consumption. The software required to implement the desired functionality can be designed somewhat separately.

In exchange for separating hardware and software design, some elements traditionally found in hardware design must be transferred to software design. Software designers have traditionally concentrated on functionality while hardware designers have worried about critical delay paths, power consumption, and area. Embedded software designers must worry about real-time deadlines, power consumption, and program and data size. As a result, embedded SoC design disciplines require a blending of hardware and software skills.

This chapter considers the characteristics of SoCs built from embedded processors. The next section surveys the types of requirements that are generally demanded from embedded SoCs. Section 22.3 surveys the characteristics of components used to build embedded systems. Section 22.4 introduces the types of architectures used in embedded systems. Section 22.5 reviews design methodologies for embedded SoCs.

0-8493-0885-2/02/$0.00+$1.50

© 2002 by CRC Press LLC

22.2 Requirements on Embedded SoCs

A digital system typically uses embedded processors to meet a combination of performance, complexity, and possibly design time goals. If the system's behavior is very regular and easy to specify as hardware, it may not be necessary to use embedded software. An embedded processor becomes more attractive when the behavior is too complex to be easily captured in hardwired logic.

Using embedded processors may reduce design time by allowing the design to be separated into distinct software and hardware units. In many cases, the CPU will be predesigned; even if the CPU and associated hardware is being designed for the project, many aspects of the hardware design can be performed separately from the software design. (Experience with embedded system designs does show, however, that the hardware and software designs are intertwined and that embedded software is prone to some of the same scheduling problems as mainframe software projects.)

But even if embedded processors seem attractive by reducing much of the design to "just programming," it must be remembered that embedded software design is much more challenging than typical applications programming for workstations or PCs. Embedded software must be designed to meet not just functional requirements—the software's input and output behavior—but also stringent nonfunctional requirements. Those nonfunctional requirements include:

- *Performance*—Although all programmers are interested in speed of execution, performance is measured much more precisely in the typical embedded system. Many embedded systems must meet real-time deadlines. The deadline is measured between two points in the software: if the program completely executes from the starting point to the end point by the deadline, the system malfunctions.
- *Energy/power*—Traditional programmers don't worry about power or energy consumption. However, energy and power are important to most embedded systems. Energy consumption is of course important in battery-operated systems, but the heat generated as a result of power consumption is increasingly important to wall-powered systems.
- *Size*—The amount of memory required by the embedded software determines the amount of memory required by the embedded system. Memory is often one of the major cost components of an embedded system.

Embedded software design resembles hardware design in its emphasis on nonfunctional requirements such as performance and power. The challenge in embedded SoC design is to take advantage of the best aspects of both hardware and software components to quickly build a cost-effective system.

22.3 Embedded SoC Components

CPUs

As shown in Fig. 22.1, a CPU is a programmable instruction set processor. Instructions are kept in a separate memory—a program counter (PC) that points to the current instruction. This definition does not consider reconfigurable logic to be a programmable computer, because it does not have a separate instruction memory and a PC. Reconfigurable logic can be used to implement sequential machines, and

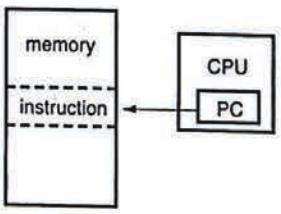

FIGURE 22.1 A CPU and memory.

so a CPU could be built in reconfigurable logic. But the separation of CPU logic and memory is an important abstraction for program design.

An embedded processor is judged by several characteristics:

- *Performance*—The overall speed of execution may be important in some systems, but in many cases we particularly care about the CPU's performance on critical sections of code.
- *Energy and power*—Processors provide different mechanisms to manage power consumption.
- *Area*—The area of the processor contributes to the total implementation cost of the SoC. The area of the memory required to store the program also contributes to implementation cost.

These characteristics are judged relative to the embedded software they are expected to run. A processor may exhibit very different performance or energy consumption on different applications.

RISC processors are commonly used in embedded computing. ARM[2] and MIPS[3] processors are examples of RISC processors that are widely used in embedded systems. A RISC CPU uses a pipeline to increase CPU performance. Many RISC instructions take the same amount of time to execute, simplifying performance analysis. However, many RISC architectures do have exceptions to this rule. An example is the multiple-register feature of the ARM processor: an instruction can load or store a set of registers, for which the instruction takes one cycle per instruction.

Most CPUs used in PCs and workstations today are superscalar processors. A superscalar processor builds on RISC techniques by adding logic that examines the instruction stream and determines, based on what CPU resources are needed, when several instructions can be executed in parallel. Superscalar scheduling logic adds quite a bit of area to the CPU in order to check all the possible conflicts between combinations of instructions; the size of a superscalar scheduler grows as n^2, where n is the number of instructions that are under consideration for scheduling. Many embedded systems, and in particular SoCs, do not use superscalar processors and instead stick with RISC processors. Embedded system designers tend to use other techniques, such as instruction-set optimization caches, to improve performance. Because SoC designers are concerned with overall system performance, not just CPU performance, and because they have a better idea of the types of software run on their hardware, they can tackle performance problems in a variety of ways that may use the available silicon area more cost-effectively.

Some embedded processors are known as digital signal processors (DSPs). The term DSP was originally used to mean one of two things: either a CPU with a Harvard architecture that provided separate memories for programs and data; or a CPU with a multiply-accumulate unit to efficiently implement digital filtering operations. Today, the meaning of the term has blurred somewhat. For instance, version 9 of the ARM architecture is a Harvard architecture to better support digital signal processing. Modern usage applies the term DSP to almost any processor that can be used to efficiently implement signal processing algorithms.

The application-specific integrated processor (ASIP)[4] is one approach to improving the performance of RISC processors for embedded application. An ASIP's instruction set is designed to match the requirements of the application software it will run. On the one hand, special-purpose function units and instructions to control them may be added to speed up certain operations. On the other hand, function units, registers, and busses may be eliminated to reduce the CPU's cost if they do not provide enough benefit for the application at hand. The ASIP may be designed manually or automatically based on profiling information. One advantage of generating the ASIP automatically is that the same information can be used to generate the processor's programming environment: a compiler, assembler, and debugger are necessary to make the ASIP useful building blocks.

Another increasingly popular architecture for embedded computing is very long instruction word (VLIW). A VLIW machine can execute several instructions simultaneously but, unlike a superscalar processor, relies on the compiler to schedule parallel instructions at compilation time. A pure VLIW machine uses slots in the long, fixed-length instruction word to control the CPU's function units, with NOPs used to indicate slots that cannot be used for useful work by the compiler. Modern VLIW machines, such as the TI C6000[5] and the Motorola/Agere StarCore,[6] group single-operation instructions into

execution packets; the packet's length can vary depending on the number of instructions that the compiler was able to schedule for simultaneous operation. VLIW machines provide instruction-level parallelism with a much smaller CPU than is possible in a superscalar system; however, the compiler must be able to extract parallelism at compilation time to be able to use the CPU's resources. Signal processing applications often have parallel operations that can be exploited at compilation time. For example, a parallel set of filter banks runs the same code on different data; the operations for each channel can be scheduled together in the VLIW instruction group.

Interconnect

Embedded SoCs may connect several CPUs, on-chip memories, and devices on a single chip. High-performance interconnect systems are required to meet the system's performance demands. The interconnection systems must also comply with standards so that existing components may be connected to them.

Busses are still the dominant interconnection scheme for embedded SoCs. Although richer interconnection schemes could be used on-chip, where they are not limited by pinout as in board-level systems, many existing architectures are still memory-limited and not interconnect-limited. However, future generations of embedded SoCs may need more sophisticated interconnection schemes.

A bus provides a protocol for communication between components. It also defines a memory space and the uses of various addresses in that memory space, for example, the address range assigned to a device connected to the bus. Busses for SoCs may be designed for high-performance or low-cost operation. A high-performance bus uses a combination of techniques—advanced circuits, additional bus lines, efficient protocols—to maximize transaction performance. One common protocol used for efficient transfers is the block transfer, in which a range of locations is transferred based on a single address, eliminating the need to transfer all the addresses on the bus. Some recent busses allow split transactions—the data request and data transfer are performed on separate bus cycles, allowing other bus operations to be performed while the original request is serviced. A low-cost bus design provides modest performance that may not be acceptable for instruction fetching or other time-critical operations. A low-cost bus is designed to require little hardware in the bus itself and to impose a small hardware and software overhead on the devices connecting to the bus. A system may contain more than one bus; a bridge can be used to connect one bus to another.

The ARM AMBA bus specification[7] is an example of a bus specification for SoCs. The AMBA spec actually includes two busses: the high-performance AMBA high-performance bus (AHB) and the low-cost AMBA peripherals bus (APB). The Virtual Sockets Interface committee has defined another standard for interconnecting components on SoCs.

Memory

One of the great advantages of SoC technology is that memory can be placed on the same chip as the system components that use the memory. On-chip memory both increases performance and reduces power consumption because on-chip connections present less reactive load than do pins and traces between chips; however, an SoC may still need to use separate chips for off-chip memory.

Although on-chip embedded memory has many advantages, it still is not as good as commodity memory. A commodity SRAM or DRAM's manufacturing process has been carefully tuned to the requirements of that component. In contrast, an on-chip memory's manufacturing needs must be balanced against the requirements of the logic circuits on the chip. The transistors, interconnections, and storage nodes of on-chip memories all have somewhat different needs than logic transistors.

Embedded DRAMs suffer the most because they need quite different manufacturing processes than do logic circuits. The processing steps required to build the storage capacitors for the DRAM cell are not good for small-geometry transistors. As a result, embedded DRAM technologies often compromise both the memory cells and the logic transistors, with neither being as good as they would be in separate,

optimized processes. Although embedded DRAM has been the subject of research for many years, its limitations have kept its from becoming a widely used technology at the time of this writing.

SRAM circuits' characteristics are closer to those of logic circuits and so can be built on SoCs with less penalty. SRAM consumes more power and requires more chip area than does DRAM, but SRAM does not need refreshing, which noticeably simplifies the system architecture.

Software Components

Software elements are also components of embedded systems. Just as pre-designed hardware components are used to both reduce design time and to provide predictions of the characteristics of parts of the system, software components can also be used to speed up software implementation time and to provide useful data on the characteristics of systems.

CPU vendors often supply software libraries for their processors. These libraries generally supply code for two types of operations. First, they provide drivers for input and output operations. Second, they provide efficient implementations of commonly-used algorithms. For example, libraries for DSPs generally include code for digital filtering, fast Fourier transforms, and other common signal processing algorithms. Code libraries are important because compilers are still not as adept as expert human programmers at creating code that is both fast and small.

The real-time operating system (RTOS) is the second major category of software component. Many applications perform several different types of operations, often with their own performance requirements. As a result, the software is split into processes that run independently under the control of an RTOS. The RTOS schedules the processes to meet performance goals and efficiently utilize the CPU and other hardware resources. The RTOS may also provide utilities, such as interprocess communication, networking, or debugging. An RTOS's scheduling policy is necessarily very different from that used in workstations and mainframes, because the RTOS must meet real-time deadlines. A priority-driven scheduling algorithm such as rate-monotonic scheduling (RMS)[9] is often used by the RTOS to schedule activity in the system.

22.4 Embedded System Architectures

The hardware architecture of an embedded SoC is generally tuned to the requirements of the application. Different domains, such as automotive, image processing, and networking all have very different characteristics. In order to make best use of the available silicon area, the system architecture is chosen to match the computational and communication requirements of the application. As a result, a much wider range of hardware architectures is found for embedded systems as compared with traditional computer systems.

Figure 22.2 shows one common configuration, a bus-based uniprocessor architecture for an embedded system. This architecture has one CPU, which greatly simplifies the software architecture. In addition to I/O devices, the architecture may include several devices known as accelerators designed to speed up

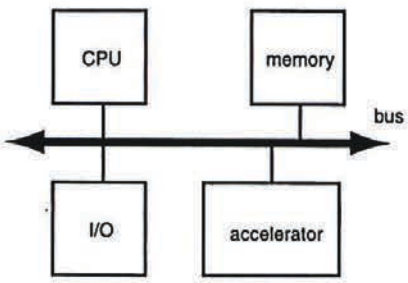

FIGURE 22.2 A bus-based, single-CPU embedded system.

computations. (Though some authors refer to these units as co-processors, we prefer to reserve that term for units that are dispatched by the CPU's execution unit.) For example, a video operation's inner loops may be implemented in an application-specific IC (ASIC) so that the operation can be performed more quickly than would be possible on the CPU. An accelerator can achieve performance gains through several mechanisms: by implementing some functions in special hardware that takes fewer cycles than is required on the CPU, by reducing the time required for control operations that would require instructions on the CPU, and by using additional registers and custom data flow within the accelerator to more efficiently implement the available communication. The single-CPU/bus architecture is commonly used in applications that do not have extensive real-time characteristics and ones that need to run a wider variety of software. For example, many PDAs use this type of architecture. A single-CPU system simplifies software design and debugging since all the work is assumed to happen on one processing element. The single CPU system is also relatively inexpensive.

In general, however, a high-performance embedded system requires a heterogeneous multiprocessor—a multiprocessor that uses more than one type of processing element and/or a specialized communication topology. Scientific parallel processors generally use a regular architecture to simplify programming. Embedded systems use heterogeneous architectures for several reasons:

- *Cost*—A regular architecture may be much larger and more expensive than a heterogeneous architecture, which freed from the constraint of regularity, can remove resources from parts of the architecture where they are not needed and add them to parts where they are needed.
- *Real-time performance*—Scientific processors are desgined for overall performance but not to meet deadlines. Embedded systems must often put processing power near the I/O that requires real-time responsiveness; this is particularly true if the processing must be performed at a high rate. Even if a high-rate, real-time operation requires relatively little computation on each iteration, the high interrupt rate may make it difficult to perform other processing tasks on the same processing element.

Many embedded systems use heterogeneous multiprocessors. One example comes from telephony. A telephone must perform both control- and data-intensive operations: both the network protocol and the user interface require control-oriented code; the signal processing operations require data-oriented code. The Texas Instruments OMAP architecture, shown in Fig. 22.3, is designed for telephony: the RISC processor handles general-purpose and control-oriented code while the DSP handles signal processing. Shared memory allows processes on the two CPUs to communicate, as does a bridge. Each CPU has its own RTOS that coordinates processes on the CPU and also mediates communication with the other CPU.

The C-Port network processor,[11] whose hardware architecture is shown in Fig. 22.4, provides an example of a heterogeneous multiprocessor in a different domain. The multiprocessor is a high-speed bus. The RISC executive processor is C programmable and provides overall control, initialization, etc. Each of the 16 HDLC processors is also C programmable. Other interfaces for higher-speed networks are not general-purpose computers and can be programmed only with register settings.

Another category of heterogeneous parallel embedded systems is the networked embedded system. Automobiles are a prime example of this type of system: the typical high-end car includes over a hundred microprocessors ranging from 4-bit microcontrollers to high-performance 32-bit processors. Networks help to distribute high-rate processing to specialized processing elements, as in the HP DesignJet, but

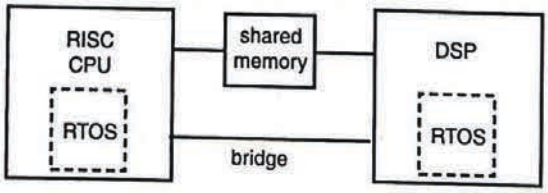

FIGURE 22.3 The TI OMAP architecture.[10]

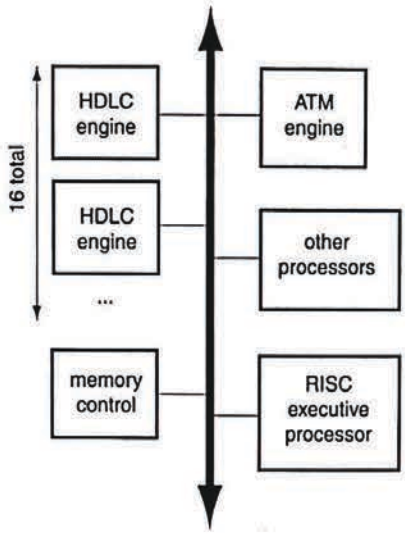

FIGURE 22.4 Block diagram of the C-Port network processor.[11]

they are most useful when the processing elements must be physically distributed. When the processing elements are sufficiently far apart, busses designed for lumped microprocessor systems do not work well. The network is generally used for data transfer between the processing elements, with each processing element maintaining its own program memory as well as a local data memory. The processing elements communicate data and control information as required by the application. I^2C and CAN are two widely-used networks for distributed systems.

22.5 Embedded SoC Design Methodologies

Specifications

As described in section 22.2, embedded computers are typically used to build systems with complex functionality. Therefore, capturing a functional description of the system is an important part of the design process. A variety of specification languages have been developed. Many of these languages were developed for software systems, but several languages have been developed over the past decade with embedded systems in mind.

Specification languages are generally designed to capture particular styles of design. Many languages have been created to describe control-oriented systems. An early example was Statecharts,[12] which introduced hierarchical states that provided a structured description of state machines. The SDL language[13] is widely used to specify protocols in telecommunications systems. The Esterel language[14] describes a reactive system as a network of communicating state machines.

Data-oriented languages find their greatest use in signal processing systems. Dataflow process networks[15] are one example of a specification language for signal processing. Object-oriented specification and design have become very popular in software design. Object-oriented techniques mix control and data orientation. Objects tend to reflect natural assemblages of data; the data values of an object define its state and the states of the objects define the state of the system. Messages providing communication and control. The real-time object-oriented Methodology (ROOM)[16] is an example of an object-oriented methodology created for embedded system design.

In practice, many systems are specified in the C programming language. Many practical systems combine control and data operations, making it difficult to use one language that is specialized for any type of description. Algorithm designers generally want to prototype their algorithms and verify them through experimentation; as a result, an executable program generally exists as the golden standard with

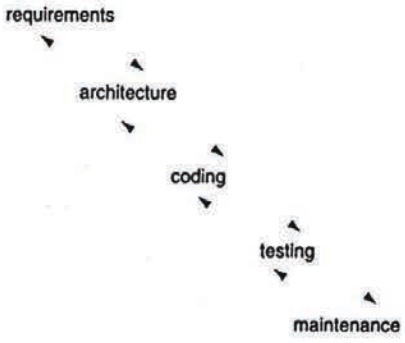

FIGURE 22.5 The waterfall model of software development.

which the implementation must conform. This is especially true when the product's capabilities are defined by standards committees, which typically generate one or more reference implementations, usually in C. Once a working piece of code exists in C, there is little incentive to rewrite it in a different specification language; however, the C specification is generally a long way from an implementation. Algorithmic designs are usually written for uniprocessors and ignore many aspects of I/O, whereas embedded systems must perform real-time I/O and often distribute tasks among several processing elements. Algorithm designers often do not optimize their code for any particular platform, and their code is certainly not optimized for any particular embedded platform. As a result, a C language specification often requires substantial re-engineering before it can be used in an embedded system.

Design Flows

In order to better understand modern design methodologies for embedded SoCs, we can start with traditional software engineering methodologies. The waterfall model, one of the first models of software design, is shown in Fig. 22.5. The waterfall model is a top-down model with only local feedback. Other software design models, such as the spiral model, try to capture more bottom-up feedback from implementation to system design; however, software design methodologies are designed primarily to implement functionality and to create a maintainable design. Embedded SoCs must, as mentioned in section 22.2, satisfy performance and power goals as well. As a result, embedded system design methodologies must be more complex.

The design of the architecture of an embedded SoC is particularly important because the architecture defines the capabilities that will limit both the hardware and software implementations. The architecture must of course be cost effective, but it must also provide the features necessary to do the job. Because the architecture is custom designed for the application, it is quite possible to miss architectural features that are necessary to efficiently implement the system. Retrofitting those features back into the architecture may be difficult or even impossible if the hardware and software design efforts do not keep in sync.

Important decisions about the hardware architecture include:

- How many processing elements are needed?
- What processing elements should be programmable and which ones should be hardwired?
- How much communication bandwidth is needed in the system and where is it needed?
- How much memory is needed and where should it go in the system?
- What types of components will be used for processors, communication, and memory?

The design of the software architecture is just as important and goes hand-in-hand with the hardware architecture design. Important decisions about the software architecture include:

- How should the functionality be split into processes?
- How are input and output performed?

- How should processes be allocated to the various processing elements in the hardware architecture?
- When should processes be scheduled?

In practice, information required to make these decisions comes from several sources. One important source is previous designs. Though technology and requirements both change over time, similar designs can provide valuable lessons on how to (and not to) design the next system. Another important source is implementation. Some implementation information can come from pre-designed hardware or software components, which is one reason why intellectual-property-based design is so important. Implementation can also come from early design efforts.

A variety of CAD algorithms have been developed to explore the embedded system design space and to help automate system architectures. Vulcan[21] and Cosyma[22] were early hardware/software partitioning systems that implemented a design using a CPU and one or more custom ASICs. Other algorithms target more general architectures.[23,24]

Once the system architecture has been defined, the hardware and software must be implemented. Hardware implementation challenges include:

- finding efficient protocols to connect together existing hardware blocks,
- memory design,
- clock rate optimization,
- power optimization.

Software implementation challenges include:

- meeting performance deadlines,
- minimizing power consumption,
- minimizing memory requirements.

The design must be verified throughout the design process. Once the design progresses to hardware and software implementation, simulation becomes challenging because the various components operate at very different levels of abstraction. Hardware units are modeled at the clock-cycle level. Software components must often be run at the instruction level or in some cases at even higher levels of abstraction. A hardware/software co-simulator[19] is designed to coordinate simulations that run at different time scales. The co-simulator coordinates multiple simulators—hardware simulators, instruction-level simulators, behavioral software processes—and keeps track of the time in each simulation. The co-simulator ensures that communications between the simulators happen at the right time for each simulator.

Design verification must include performance, power, and size as well as functionality. Although these sorts of checks are common in hardware design, they are relatively new to software design. Performance and power verification of software may require cache simulation. Some recent work has developed higher-level power models for CPUs.

Platform-Based Design

One response to the conflicting demands of SoC design has been the devlopment of platform-based design methodologies. On the one hand, SoCs are becoming very complex. On the other hand, they must be designed very quickly to meet the electronics industry's short product lifecycles.

Platform-based design tries to tackle this problem by dividing the design process into two phases. In the first phase, a platform is designed. The platform defines the hardware and software architectures for the system. The degree to which the architecture can be changed depends on the needs of the marketplace. In some cases, the system may be customizable only by reprogramming. In other cases, it may be possible to add or delete hardware components to provide specialized I/O, additional processing capabilities, etc. In the second phase, the platform is specialized into a product. Because much of the initial design work was done in the first phase, the product can be developed relatively quickly based on the platform.

Platform-based design is particularly well suited to products derived from standards. On the one hand, all products must meet the minimum requirements of the standard. On the other hand, standards committees generally leave room for different implementations to distinguish themselves: added features, lower power, etc. Designers will generally want to modify their design to add features that differentiate their product in the marketplace.

Platform-based design also allows designers to incorporate design experience into products. Each product derived from the platform will teach something: how to better design part of the system, unexpected needs of customers, etc. The platform can be updated as new products are developed from it so that successive designs will be easier.

Platforms are usually designed within a technology generation. A new VLSI technology generally changes enough design decisions that platforms must be rethought for each new generation of technology. Therefore, the platform itself must be designed quickly and each product based on the platform must be completed quickly in order to gain effective use of the platform design effort in the 18-month lifecycle of a manufacturing technology.

Software Performance Analysis and Optimization

Although methods for hardware performance analysis and optimization are well-known, software techniques for optimizing performance have been developed only recently to meet the demands of embedded design methodologies.

The performance of an embedded system is influenced by several factors at different levels of abstraction. The first is the performance of the CPU pipeline itself. RISC design techniques tend to provide uniform execution times for instructions, but software performance is not always simple to predict. Register forwarding, which is used to enhance pipeline performance, also makes execution time less predictable. Branch prediction causes similar problems.

Superscalar processors, because they schedule instructions at execution time based upon execution data, provide much less predictable performance than do either RISC or VLIW processors. This is one reason why superscalar processors are not frequently used in real-time embedded systems.

The memory system is often an even greater source of uncertainty in embedded systems. CPUs use caches to improve average memory response time, but the effect of the cache on a particular piece of software requires complex analysis. In pathological cases, the cache can add uncertainty to execution times without actually improving the performance of critical software components. Cache simulation is often used to analyze the behavior of a program in a cache. Analysis must take into account both instructions and data. Unlike in workstation CPUs, in which the cache configuration is chosen by the CPU architect based on benchmarks, the designer of an embedded SoC can choose the configurations of caches to match the characteristics of the embedded software. Embedded system designers can choose between hardware and software optimizations to meet performance goals.

Analyzing the performance of a program requires determining both the execution path and the execution time of instructions along that path.[18] Both are challenging problems.[20] The execution path of a program clearly depends on input data values. To ensure that the program meets a deadline, the worst-case execution path must be determined. The execution time of instructions along the path depend on several factors: data values, interactions between instructions, and cache state.

Energy/Power Analysis and Optimization

Many embedded systems must also meet energy and power goals as well as performance goals. The specification may impose several types of power requirements: peak power consumption, average power consumption, energy consumption for a given operation.

To a first-order, high-performance design is low-power design. Efficient implementations that run faster also tend to reduce power consumption, but trade-offs between performance and power in embedded system design. For example, the power consumption of a cache depends on both its size and the

memory system activity.[25] If the cache is too small, too many references require expensive main memory accesses. If the cache is too large, it burns too much static power. Many applications exhibit a sweet spot at which the cache is large enough to provide most of the available performance benefit while not burning too much static power. Techniques have been developed to estimate hardware/software power consumption.[26]

System-level approaches can also help reduce power consumption.[27] Components can be selectively turned off to save energy; however, because turning a component on again may consume both time and energy, the decision to turn it off must be made carefully. Statistical methods based on Markov models can be used to create effective system-level power management methodologies.

22.6 Summary

Embedded computers promise to solve a critical design bottleneck for SoCs. Because we can design CPUs relatively independently of the programs they run and reuse those CPUs design across many chips, embedded computers help to close the designer productivity gap. Embedded processors, on the other hand, require that many design techniques traditionally reserved for hardware—deadline-driven performance, power minimization, size—must now be applied to software as well. Design methodologies for embedded SoCs must carefully design system architectures that will allow hardware and software components to work together to meet performance, power, and cost goals while implementing complex functionality.

References

1. Wayne Wolf, *Computers as Components: Principles of Embedded Computer System Design*, San Francisco: Morgan Kaufman, 2000.
2. http://www.arm.com.
3. http://www.mips.com.
4. G. Goossens, J. van Praet, D. Lanneer, W. Geurts, A. Kifli, C. Liem, and P. G. Paulin, "Embedded software in real-time signal processing systems: design technologies," *Proceedings of the IEEE*, 85(3), March 1997, pp. 436–453.
5. http://www.ti.com.
6. http://www.lucent.com/micro/starcore/motover.htm.
7. ARM Limited, *AMBA(TM) Specification (Rev 2.0)*, ARM Limited, 1999.
8. http://www.vsi.com.
9. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, 20(1), 1973, pp. 46–61.
10. http://www.ti.com/sc/docs/apps/wireless/omap/overview.htm.
11. http://www.cportcorp.com/products/digital.htm.
12. D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, 8, 1987, pp. 231–274.
13. Anders Rockstrom and Roberto Saracco, "SDL—CCITT specification and description language," *IEEE Transactions on Communication*, 30(6), June 1982, pp. 1310–1318.
14. Albert Benveniste and Gerard Berry, "The synchronous approach to reactive real-time systems," *Proceedings of the IEEE*, 79(9), September 1991, pp. 1270–1282.
15. E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, 83(5), May 1995, pp. 773–801.
16. Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-Time Object-Oriented Modeling*, New York: John Wiley and Sons, 1994.
17. Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Todd, *Surviving the SOC Revolution: A Guide to Platform-Based Design*, Kluwer Academic Publishers, 1999.
18. Chang Yun Park and Alan C. Shaw, "Experiments with a program timing tool based on source-level timing scheme," *IEEE Computer*, 24(5), May 1991, pp. 48–57.

19. David Becker, Raj K. Singh, and Stephen G. Tell, "An engineering environment for hardware/software co-simulation," in *Proceedings, 29th Design Automation Conference,* IEEE Computer Society Press, 1992, pp. 129–134.
20. Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe, "Performance estimation of embedded software with instruction cache modeling," in *Proceedings, ICCAD-95,* IEEE Computer Society Press, 1995, pp. 380–387.
21. Rajesh K. Gupta and Giovanni De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Design & Test,* 10(3), September 1993, pp. 29–41.
22. R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design & Test,* 10(4), December 1993, pp. 64–75.
23. Wayne Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Transactions on VLSI Systems,* 5(2), June 1997, pp. 218–229.
24. Asawaree Kalavade and Edward A. Lee, "The extended partitioning problem: Hardware/software mapping, scheduling, and implementation-bin selection," *Design Automation for Embedded Systems,* 2(2), March 1997, pp. 125–163.
25. Yanbing Li and Joerg Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proceedings, 35th Design Automation Conference,* ACM Press, 1998, pp. 188–194.
26. W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano, "Power estimation of embedded systems: a hardware/software codesign approach," *IEEE Transactions on VLSI Systems,* 6(2), June 1998, pp. 266–275.
27. L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Transactions on VLSI Systems,* 8(3), June 2000, pp. 299–316.