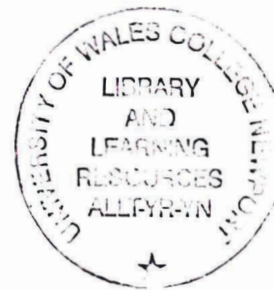


THE

COMPUTER ENGINEERING

H A N D B O O K

Edited by
VOJIN G. OKLOBDZIJA



CRC PRESS

Boca Raton London New York Washington, D.C.

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

18

Low-Power Design of Systems on Chip

18.1	Introduction	18-1
18.2	Power Reduction from High to Low Level	18-2
	Design Techniques for Low Power • Some Basic Rules • CAD Tools	
18.3	Large Power Reduction at High Level.....	18-3
	RF Devices • Low-Power Software • Processors, Instructions Sets, and Random Logic • Processor Types • Low-Power Memories • The Energy-Flexibility Gap	
18.4	Low-Power and Reliability Issues in SoCs	18-7
	Low-Power SoC Architectures • Low-Power Design and Testability Issue	
18.5	Low-Power Microcontroller Cores	18-8
	CoolRISC Microcontroller Architecture • IP “Soft” Cores • Latch-Based Designs • Gated Clock with Latch-Based Designs • Results	
18.6	Low-Power Memories.....	18-13
	Cache Memories for Microcontrollers • Electrical Design of Low-Power ROM Memories • Electrical Design of Low-Power SRAM Memories	
18.7	Low-Power Standard Cell Libraries.....	18-16

Christian Piguet

*CSEM: Centre Suisse d'Electronique
et de Microtechnique SA and
LAP-EPFL*

18.1 Introduction

For innovative portable and wireless devices, systems on chip (SoCs) containing several processors, memories, and specialized modules are obviously required. Performance and also low power are main issues in the design of such SoCs. In deep submicron technologies, SoCs contain several millions of transistors and have to work at lower and lower supply voltages to avoid too high power consumption. Consequently, digital libraries as well as ROM and SRAM memories have to be designed to work at very low supply voltages and to be very robust while considering wire delays, signal input slopes, noise, and crosstalk effects.

Are these low-power SoCs only constructed with low-power processors, memories, and logic blocks? If the latter are unavoidable, many other issues are quite important for low-power SoCs, such as the way to synchronize the communications between processors as well as test procedures, online testing, software design and development tools. This chapter is a general framework for the design of low-power SoCs, starting from the system level to the architecture level, assuming that the SoC is mainly based on the reuse of low-power processors, memories, and standard cell libraries.

18.2 Power Reduction from High to Low Level

Design methodologies at different abstraction levels such as systems, architectures, logic design, basic cells as well as layout, have to take into account the power consumption. The main goals of such design methods are the V_{dd} reduction, the activity reduction as well as the capacitance reduction [1–7]. One has to ask the following question: What are the results of several years of research, applications, industrial designs in low power?

Two ways to consider this:

- What are the new or effective design techniques to reduce power?
- What is the status of the CAD tools regarding low power, as it is well known that such tools are required today to layout several millions of transistors on a single chip?

Design Techniques for Low Power

Future SoCs will contain several different processor cores on a single chip. It results in parallel architectures, which are known to be less power hungry than fully sequential architectures based on a single processor [8]. The design of such architectures has to start with very high-level models in languages such as System C, SDL, or MATLAB. The very difficult task is then to translate such very high-level models in application software in C and in RTL languages (VHDL, Verilog) to be able to implement the system on several processors. One could think that many tasks running on many processors require a multitask but centralized operating system (OS), but regarding low power, it would be better to have tiny OS (2 K or 4 K instructions) for each processor [9], assuming that each processor executes several tasks. Obviously, this solution is easier as each processor is different even if performances could be reduced due to the inactivity of a processor that has nothing to do at a given time frame.

One has to note that most of the power can be saved at the highest levels. At the system level, partition, activity, number of steps, simplicity, data representation, and locality (cache or distributed memory instead of a centralized memory) have to be chosen (Fig. 18.1). These choices are strongly application dependent. Furthermore, these choices have to be performed by the SoC designer, and he has to be power conscious.

At the architecture level, many low-power techniques have been proposed (Fig. 18.1). The list could be gated clocks, pipelining, parallelization, very low V_{dd} , several V_{dd} , variable V_{dd} and V_T , activity estimation and optimization, low-power libraries, reduced swing, asynchronous, adiabatic. Some are used in industry, but some are not, such as adiabatic and asynchronous techniques. At the lowest levels, for instance a low-power library, only a moderate factor (about 2) in power reduction can be reached. At the logic and layout level, the choice of a mapping method to provide a netlist and the choice of a low-power library are crucial. At the physical level, layout optimization and technology have to be chosen.

High level	LP systems, LP software, processor types, processors versus random logic, parallel machines, high-level power estimation		
	Activity reduction	V_{dd} reduction	Capacitance reduction
Architecture	LP Codes asynchronous	Parallelization adiabatic	Simplicity
Circuit layout	Gated clock	Low V_T	Low-power library

FIGURE 18.1 Overview of low-power techniques.

Some Basic Rules

There are some basic rules that can be proposed to reduce power consumption at system and architecture levels:

- Reduction of the number N of operations to execute a given task.
- Sequencing that is too high always consumes more than the same functions executed in parallel.
- Obviously, parallel architectures provide better clock per instruction (CPI), as well as pipelined and RISC architectures.
- The lowest V_{dd} for the specific application has to be chosen.
- The goal is to design a chip that just fits to the speed requirements [10].

The main point is to think about systems, with power consumption reduction in mind. According to the mentioned basic rules, how to design a SoC that uses parallelism, at the right supply voltage, while minimizing the steps to perform a given operation or task.

The choice of a given processor or a random logic block is also very important. A processor results in a quite high sequencing while a random logic block works more in parallel for a same specific task. The processor type has to be chosen according to the work to be performed; if 16-bit data are to be used, it is not a good idea to choose a less expensive 8-bit controller and to work in double precision (high sequencing).

CAD Tools

Each specialized processor embedded in a SoC will be programmed in C and will execute after compilation its own code. Low-power software techniques have to be applied to each piece of software, including pruning, inlining, loop unrolling, and so on. For reconfigurable processor cores, retargetable compilers have to be available. The parallel execution of all these task have to be synchronized through communication links between processors and peripherals. It results that the co-simulation development tools have to deal with several pieces of software running on different processors and communicating between each other. Such a tool has to provide a high-level power estimation tool to check which are the power hungry processors, memories or peripherals as well as the power hungry software routines or loops [11]. Such a tool is far from being commercially available. Embedded low-power software emerges as a key design problem. The software content of SoC will increase as well as the cost of its development.

Generally speaking, the available CAD tools for SoC chips have been designed for robust and reliable synchronous designs. It means that even gated clocks, low V_{dd} , several V_{dd} , are not or not yet supported, and that asynchronous and adiabatic will not be supported in the near future. It is a major problem because CAD tools are far behind (10 years) the 2000 year requirements. Furthermore, little money is invested in CAD tools. It could be a stopper for some low-power methods. One can conclude that if power can be saved at a high level (factor 10 to 100 or more!) while using conventional CAD tools, it could be the way to go; however, power conscious SoC designers are required [12].

18.3 Large Power Reduction at High Level

As mentioned previously, a large part of the power can be saved at high level. Factors of 10 to 100 or more are possible; however, it means that the resulting system could be quite different, with less functionality or less programmability. The choice among various systems is strongly application dependent. One has to think about systems and low power to ask good questions of the customers and to get reasonable answers. Power estimation at high level is a very useful tool to verify the estimated total power consumption. Before starting a design for a customer, it is mandatory to think about the system and what is the goal about performances and power consumption. Several examples will be provided because this way of thinking is application dependent.

RF Devices

An FM radio can be designed with an analog FM receiver as well as with analog and digital (random logic) demodulations, but a software radio has also been proposed. Such a system converts the FM signal directly into digital with very high-speed ADCs and does the demodulation work with a microprocessor. Such a solution is interesting as the same hardware can be used for any radio, but one can be convinced that a very high-speed ADC is a very consuming block, as well as a microprocessor that has to perform the demodulation (16-bit ADC can consume 1–10 W at 2.2 GHz [13]). In [13], some examples are provided for a digital baseband processor, achieving 1500 mW if implemented with a DSP processor and only 10 mW if implemented with a direct mapped ASIC. The latter case provides a factor of 150 in power reduction.

The transmission of data from one location to another by RF link is more and more power consuming if the distance between the two points is increased. The power (although proportional to the distance at square in ideal case) is practically proportional to the distance at power 3 or even power 4 due to noise, interferences, and other problems. If three stations are inserted between the mentioned points, and assuming a power of 4, the power can be reduced by a factor 64.

Low-Power Software

Quite a large number of low-power techniques have been proposed for hardware, but relatively fewer for software. Hardware designers are today at least conscious that power reduction of SoCs is required for most applications. However, it seems that it is not the case for software people. Furthermore, a large part of the power consumption can be saved while modifying the application software.

For embedded applications, it is quite often the case that an industrial existing C code has to be used to design an application (for instance, MPEG, JPEG). The methodology consists in improving the industrial C code by

1. pruning, some parts are removed.
2. clear separation of (a) the control code, (b) the loops, and (c) the arithmetic operations.

Several techniques can be used to optimize the loops. In some applications, the application is 90% of the time running in loops. Three techniques can be used efficiently, such as loop fusion (loops executed in sequence with the same indices can be merged), loop tiling (to avoid fetching all the operands from the data cache for each loop iteration, so some data used by the previous iteration can be reused for the next iteration), and loop unrolling.

To unroll a loop is to repeat the loop body N times if there are N iterations of the loop. The code size is increased, but the number of executed instructions is reduced, as the loop counter (initialization, incrementation, and comparison) is removed.

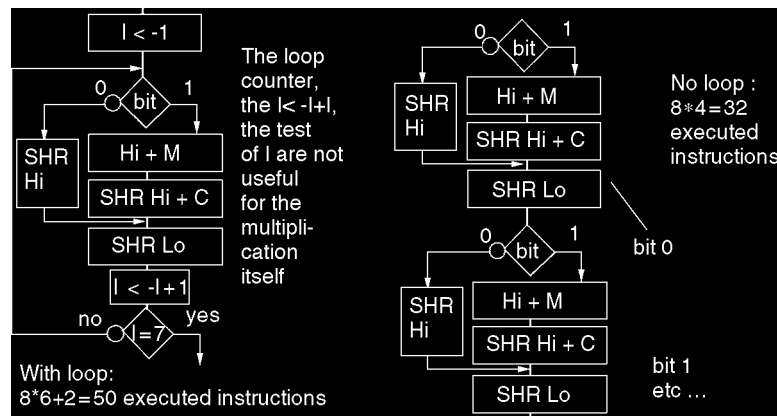
A small loop executed eight times, for instance an 8×8 multiplication, results in at least 40 executed instructions, while the loop counter has to be incremented and tested. If the loop is unrolled, the code size is larger, but the number of executed instructions is reduced to about 24 (Fig. 18.2). This example illustrates a general rule: less sequencing in the software at the price of more hardware, i.e., more instructions in the program memory. Table 18.1 also shows that a linear routine (without loops) is executed with fewer instructions than a looped routine at the price of more instructions in the program.

TABLE 18.1 Number of Instructions in the Code as well as the Number of Executed Instructions for an $N \times N$ Multiplication with a $2 \times N$ Result

Number of Instructions 8-bit Multiply Linear	CoolRISC 88 in the Code 30	CoolRISC 88 Executed 30	PIC 16C5× in the Code 35	PIC 16C5× Executed 37
8-bit multiply looped	14	56	16	71
16-bit multiply linear	127	127	240	233
16-bit multiply looped	31	170	33	333

TABLE 18.2 Multiplication with and without Hardware Multiplier

	CoolRISC 816 without Multiplier 54–62 Executed Instructions	CoolRISC 816 with Multiplier 2 Executed Instructions	Speed-Up 29
Looped 8-bit Multiply			
Looped 16-bit multiply	72–88	16	5
Floating-Point 32-bit multiply	226–308	41–53	5.7

**FIGURE 18.2** Unrolled loop multiply.

Processors, Instructions Sets, and Random Logic

A processor-based implementation results in very high sequencing. It is due to the processor architecture that is based on the reuse of the same operators, registers, and memories. For instance, only one step ($N = 1$) is necessary to up-date a hardware counter. For its software counterpart, the number of steps is much higher, while executing several instructions with many clocks in sequence. This simple example shows that the number of steps executed for the same task can be very different depending on the architecture.

The instruction set can also contain some instructions that are very useful but expensive to implement in hardware. An interesting comparison is provided by the multiply instruction that has been implemented in the CoolRISC 816 (Table 18.2). Generally, 10% of the instructions are multiplications in a given embedded code. Assume 4 K instructions, i.e., 400 instructions (10%) for multiply, resulting in 8 multiply (each multiply requires about 50 instructions), so a final code of 3.6 K instructions. This is why the CoolRISC 816 contains a hardware 8×8 multiplier.

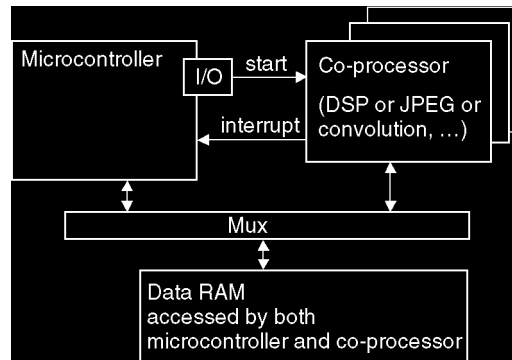
Processor Types

Several points must be fulfilled in order to save power. The first point is to adapt the data width of the processor to the required data. It results in increased sequencing to manage, for instance, 16-bit data on a 8-bit microcontroller. For a 16-bit multiply, 30 instructions are required (add-shift algorithm) on a 16-bit processor, while 127 instructions are required on a 8-bit machine (double precision). A better architecture is to have a 16×16 bit parallel-parallel multiplier with only one instruction to execute a multiplication.

Another point is to use the right processor for the right task. For control tasks, DSP processors are largely inefficient. But conversely, 8-bit microcontrollers are very inefficient for DSP tasks! For instance, to perform a JPEG compression on a 8-bit microcontroller requires about 10 millions of executed instructions for a 256×256 image (CoolRISC, 10 MHz, 10 MIPS, 1 s per image). It is quite inefficient. Factor 100 in energy reduction can be achieved with JPEG dedicated hardware. With two CSEM-designed

TABLE 18.3 Frequency and Power Consumption for a JPEG Compressor

DCT Co-processor	No. of Cycles 3.6 per Pixel	Frequency 100 MHz	Power 110 μ AMHz
Huffman co-processor	3.8 per pixel	130 MHz	20 μ AMHz
JPEG compression	3.8 per pixel	100 MHz	130 μ AMHz

**FIGURE 18.3** Microcontroller and co-processor.

co-processors working in pipeline, i.e., a DCT co-processor based on an instruction set (program memory based) and a Huffman encoder based on random logic, finite state machines, one has the following results (Table 18.3, synthesized by Synopsys in 0.25 μ m TSMC process at 2.5 V) 400 images can be compressed per second with a 13 mA power consumption. At 1.05 V, 400 images can be compressed per second with a 1 mA power consumption, resulting in quite a large number of 80,000 compressed images per watt (1000 better than a programmed-based implementation).

Figure 18.3 shows an interesting architecture to save power. For any application there is some control that is performed by a microcontroller (the best machine to perform control). But in most applications, there is also a main task to execute such as DSP tasks, convolutions, JPEG, or other tasks. The best architecture is to design a specific machine (co-processor) to execute such a task. So this task is executed by the smallest and the most energy efficient machine. Most of the time, both microcontroller and co-processors are not running in parallel.

Low-Power Memories

Memory organization is very important in systems on a chip. Generally, memories consume most of the power. So it comes immediately that memories have to be designed hierarchically. No memory technology can simultaneously maximize speed and capacity at lowest cost and power. Data for immediate use is stored in expensive registers, in cache memories, and less used data in large memories.

For each application, the choice of the memory architecture is very important. One has to think of hierarchical, parallel, interleaved, and cache memories (sometimes several levels of cache) to try to find the best trade-off. The application algorithm has to be analyzed from the data point of view, the organization of the data arrays, and how to access these structured data.

If a cache memory is used, it is possible, for instance, to minimize the number of cache miss while using adequate programming as well as a good data organization in the data memory. For instance, in inner loops of the program manipulating structured data, it is not equivalent to write (1) do i then do j or (2) do j then do i depending on how the data are located in the data memory.

Proposing a memory-centric view (as opposed to the traditional CPU-centric view) to SoC design has become quite popular. It is certainly of technological interest. It means, for instance, that the DRAM memory is integrated on the same single chip; however, it is unclear if this integration inspires any truly

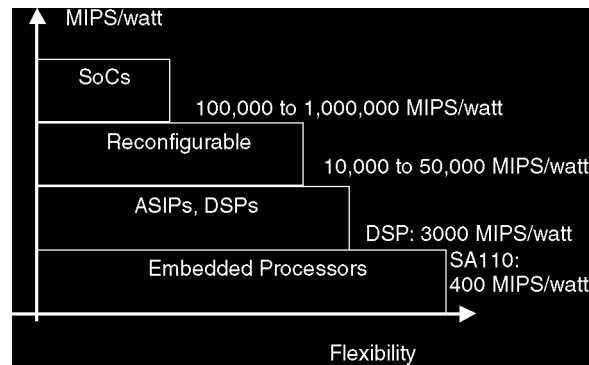


FIGURE 18.4 Energy-flexibility gap.

new architecture paradigms. We see it as more of a technological implementation issue [14]. It is, however, crucial to have most of the data on-chip, as fetching data off-chip at high frequency is a very high power consumption process.

The Energy-Flexibility Gap

Figure 18.4 shows that the flexibility [13], i.e., to use a general purpose processor or a specialized DSP processor, has a large impact on the energy required to perform a given task compared to the execution of the same given task on dedicated hardware.

18.4 Low-Power and Reliability Issues in SoCs

For SoCs, very important design problems have to be solved. They are mainly the silicon complexity (reliability, power consumption, interconnect delays), the system complexity (logic, MPU, memories, RF, FPGA, RF, MEMS), the design procedures (300–800 people for the design of a single chip, IP reuse, design levels), verification and test. The total number of transistors on a single chip could be over one billion (predicted [15] to be between 4 to 19 billions in 2014 depending in the circuit type). Some partial answers have been given to the complexity problem, such as design reuse. The SIA Roadmap [15] predicts that in 2012 reuse of processors, logic blocks, and peripherals, would reach about 90% of the embedded logic on the chip.

Low-Power SoC Architectures

Low-power SoCs will be based on low-power components, such as processor cores, memories, and libraries, that are available with good performances, i.e., 20,000–100,000 MIPS/watt [16] for some cores (using low-power techniques such as gated clocks); however, memories are the main consumers on a SoC and several techniques at the architecture and electrical levels have to be applied to reduce their power consumption (generally based on caches, DWL, bitline splitting, low swing).

The 1997 SIA Roadmap [15] recognizes that, in 2007, asynchronous design will be used in many designs. If ∂ is the distance travelled by a signal in one clock cycle, due to higher frequencies and increasing interconnect delays, a chip will contain several time zones of sizes $\partial \times \partial$. Due also to the increasing die size, this number of time zones will grow very rapidly with the new technologies, up to 10,000 zones in 2012. To synchronize 10,000 time zones is a true asynchronous problem [15]. This is why asynchronous design is strongly required for chip architectures in the future [17].

Asynchronous architectures are often presented as being capable of reducing significantly the power consumption [17]. Looking at the results from many papers, it turns out that the largest power savings are obtained with circuits or applications that present a very irregular behavior. An irregular behavior

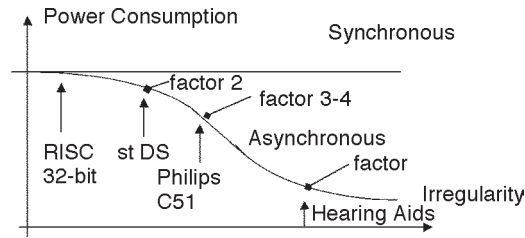


FIGURE 18.5 Comparison of the power consumption of asynchronous vs. synchronous architectures.

is, for instance, a processor that presents a tricky instruction set with multi-bytes instructions executed in a various number of cycles and phases. Or an application for which the controller has to very often stop and restart depending on the application. A very regular behavior is a 32-bit RISC core for which all instructions are always executed in one clock. Figure 18.5 illustrates this basic law [18]. Basically, SoCs will present more of an irregular behavior than a regular one.

At the architecture level, power and V_{dd} management with behavior prediction of the user will be used extensively, as well as low-power communication protocols between the various processors on a single chip. These protocols have to be kept simple and will be asynchronous due to the fact that the various cores will be clocked (if not asynchronous cores) with many different frequencies.

Low-Power Design and Testability Issue

At the low-level, low-power libraries and logic synthesis embedded with place and route are required, as well as estimation of interconnect delays with copper, low-k, and SOI; however, the main issue is V_{dd} as low as 0.6 to 0.3 V in 2014. With very deep submicron technologies and very low V_{dd} , the static power will increase significantly due to low V_t . Several techniques with double V_t , source impedance, well polarization, dynamic regulation of V_t are today under investigation and will be necessarily used in the future.

SoCs testability and debug, when the first silicon has been returned from the foundry, are important issues. Generally, the mean time to fix a bug is one week. Today, it is not possible to determine if more bugs will be present in a one-billion transistor chip, and if a one-week fix per bug is realistic or not; however, it has to be mentioned that it could be much more difficult to fix a bug in IP blocks that you have not designed. It is estimated that half of the total design effort will be devoted to verification tasks, including debugging of the embedded software [15].

18.5 Low-Power Microcontroller Cores

The most popular 8-bit microcontroller is the Intel 8051, but each instruction is executed in at least 12 clock cycles resulting in poor performances in MIPS (million of instructions per second) and MIPS/watt. MIPS performances of microcontrollers are not required to be very high. Consequently, short pipelines and low operating frequencies are allowed if, however, the number of CPI is low. Such a low CPI has been used for the CoolRISC microcontroller [19,20]. The CoolRISC 88 is an 8-bit core with eight registers and the CoolRISC 816 is an 8-bit core with 16 registers.

CoolRISC Microcontroller Architecture

The CoolRISC is a 3-stage pipelined core. The branch instruction is executed in only one clock. In that way, no load or branch delay can occur in the CoolRISC core, resulting in a strictly $CPI = 1$ (Fig. 18.6). It is not the case of other 8-bit pipelined microprocessors (PIC, Nordic μ RISC, Scenix, MCS-151, and 251). It is known that the reduction of CPI is the key to high performances. For each instruction, the first half clock is used to precharge the ROM program memory. The instruction is read and decoded in

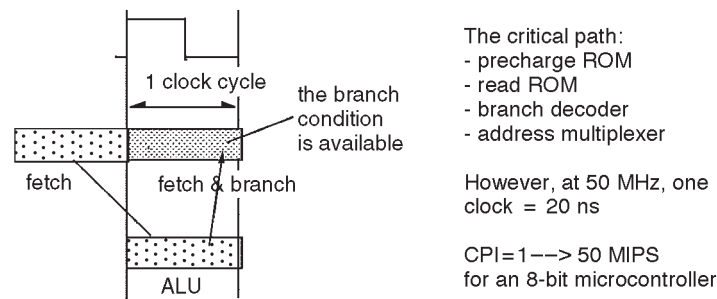


FIGURE 18.6 No branch delay.

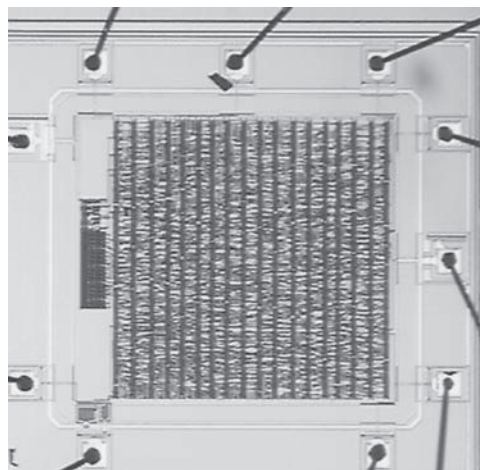


FIGURE 18.7 Microphotograph of CoolRISC 88.

the second half of the first clock. As shown in Fig. 18.6, a branch instruction is also executed during the second half of this first clock, which is long enough to perform all the necessary transfers. For a load/store instruction, only the first half of the second clock is used to store data in the RAM memory. For an arithmetic instruction, the first half of the second clock is used to read an operand in the RAM memory or in the register set, the second half of this second clock to perform the arithmetic operation, and the first half of the third clock to store the result in the register set. Figure 18.7 is a CoolRISC 88 test chip.

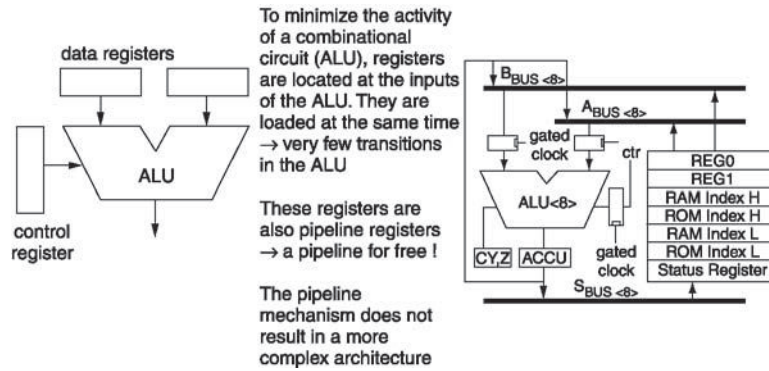
Another very important issue in the design of 8-bit microcontrollers is the power consumption. The gated clock technique has been extensively used in the design of the CoolRISC cores (Fig. 18.8). The ALU, for instance, has been designed with input and control registers that are loaded only when an ALU operation has to be executed. During the execution of another instruction (branch, load/store), these registers are not clocked thus no transitions occur in the ALU (Fig. 18.8). This reduces the power consumption. A similar mechanism is used for the instruction registers, thus in a branch, which is executed only in the first pipeline stage, no transitions occur in the second and third stages of the pipeline. It is interesting to see that gated clocks can be advantageously combined with the pipeline architecture; the input and control registers implemented to obtain a gated clocked ALU are naturally used as pipelined registers.

IP “Soft” Cores

The main issue in the design of “soft” cores [21] is reliability. In deep submicron technologies, gate delays are smaller and smaller compared to wire delays. Complex clock trees have to be designed using Synopsys to satisfy the required timing, mainly the smallest possible clock skew, and to avoid any timing violations.

TABLE 18.4 Power Consumption of the Same Core with Various Test Benches and Skew

Skew	Test Bench A	Test Bench B
10 ns	0.44 mW/MHz	0.76 mW/MHz
3 ns	0.82 mW/MHz	1.15 mW/MHz

**FIGURE 18.8** Gated-clock ALU.

Furthermore, “soft” cores have to present a low power consumption to be attractive to the possible licensees. If the clock tree is a major issue to achieve the required clock skew, its power consumption could be larger than desired. Today, most IP cores are based on a single-phase clock and are based on D-flip-flops. As shown in the following example, the power consumption is largely dependent on the required clock skew.

As an example, a DSP core synthesized with the CSEM low-power library in TSMC 0.25 μm . The test bench A contains only a few multiplication operations, while the test bench B performs a large number of MAC operations (Table 18.4). Results show that if the power is sensitive to the application program, it is also quite sensitive to the required skew: 100% of power increase from 10 ns to 3 ns skew.

The clocking scheme of IP cores is therefore a major issue. Another approach other than the conventional single-phase clock with D-flip-flops (DFF) is presented in this paper. It is based on a double-latch approach with two nonoverlapping clocks. This clocking scheme has been used for the 8-bit CoolRISC microcontroller IP core [16] as well as for other cores, such as a DSP core and other execution units [22]. The advantages as well as the disadvantages will be presented.

Latch-Based Designs

Figure 18.9 shows the double-latch concept that has been chosen for such IP cores to be more robust to the clock skew, flip-flop failures, and timing problems at very low voltage [16]. The clock skew between various ϕ_1 (respectively ϕ_2) pulses have to be shorter than half a period of CK. However, one requires two clock cycles of the master clock CK to execute a single instruction. It is why one needs, for instance, in technology TSMC 0.25 μm , 120 MHz to generate 60 MIPS (CoolRISC with CPI = 1), but the two ϕ_i clocks and clock trees are at 60 MHz. Only a very small logic block is clocked at 120 MHz to generate two 60 MHz clocks.

The design methodology using latches and two nonoverlapping clocks has many advantages over the use of DFF methodology. Due to the nonoverlapping of the clocks and the additional time barrier caused by having two latches in a loop instead of one DFF, latch-based designs support greater clock skew, before failing, than a similar DFF design (each targeting the same MIPS). This allows the synthesizer and router to use smaller clock buffers and to simplify the clock tree generation, which will reduce the power consumption of the clock tree.

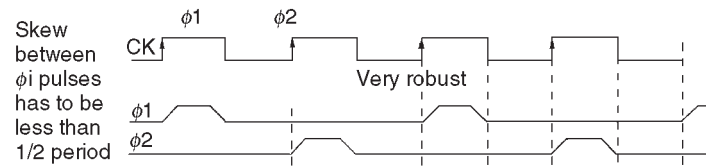


FIGURE 18.9 Double-latch clocking schemes.

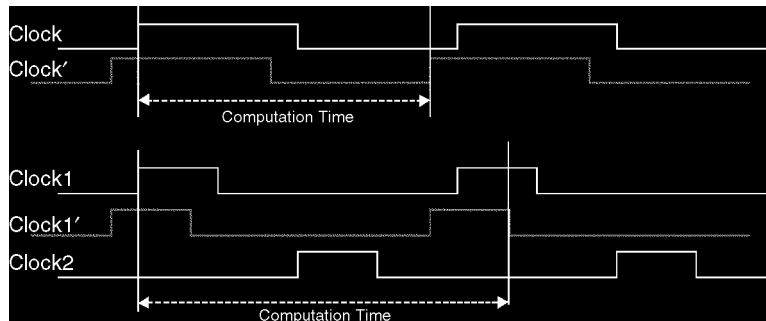


FIGURE 18.10 Time borrowing.

With latch-based designs, the clock skew becomes relevant only when its value is close to the nonoverlapping of the clocks. When working at lower frequency and thus increasing the nonoverlapping of clocks, the clock skew is never a problem. It can even be safely ignored when designing circuits at low frequency, but a shift register made with DFF can have clock skew problems at any frequency.

Furthermore, if the chip has clock skew problems at the targeted frequency after integration, one is able, with a latch-based design, to reduce the clock frequency. It results that the clock skew problem will disappear, allowing the designer to test the chip functionality and eventually to detect other bugs or to validate the design functionality. This can reduce the number of test integrations needed to validate the chip. With a DFF design, when a clock skew problem appears, you have to reroute and integrate again. This point is very important for the design of a chip in a new process not completely or badly characterized by the foundry, which is the general case as a new process and new chips in this process are designed concurrently for reducing the time to market.

Using latches for pipeline structure can also reduce power consumption when using such a scheme in conjunction with clock gating. The latch design has additional time barriers, which stop the transitions and avoid unneeded propagation of signal and thus reduce power consumption. The clock gating of each stage (latch register) of the pipeline with individual enable signals, can also reduce the number of transitions in the design compared to the equivalent DFF design, where each DFF is equal to two latches clocked and gated together.

Another advantage with a latch design is the time borrowing (Fig. 18.10). It allows a natural repartition of computation time when using pipeline structures. With DFF, each stage of logic of the pipeline should ideally use the same computation time, which is difficult to achieve, and in the end, the design will be limited by the slowest of the stage (plus a margin for the clock skew). With latches, the slowest pipeline stage can borrow time from either or both the previous and next pipeline stage. And the clock skew only reduces the time that can be borrowed. An interesting paper [23] has presented time borrowing with DFF, but such a scheme needs a complete new automatic clock tree generator that does not minimize the clock skew but uses it to borrow time between pipeline stages.

Using latches can also reduce the number of MOS of a design. For example, a microcontroller has 16×32 -bits registers, i.e., 512 DFF or 13,312 MOS (using DFF with 26 MOS). With latches, the master part of the registers can be common for all the registers, which gives 544 latches or 6528 MOS (using latches with 12 MOS). In this example, the register area is reduced by a factor of 2.

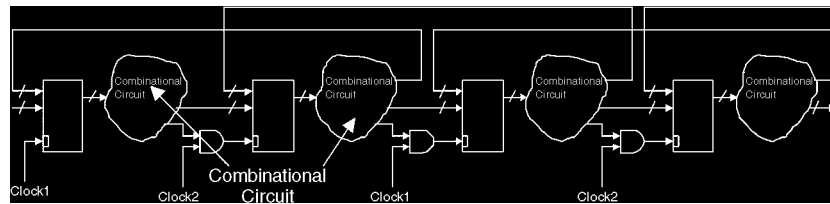


FIGURE 18.11 Latch-based clock gating.

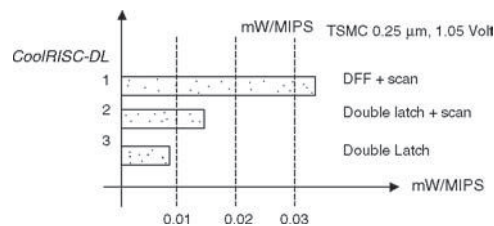


FIGURE 18.12 Power consumption comparison of “soft” CoolRISC cores.

Gated Clock with Latch-Based Designs

The latch-based design also allows a very natural and safe clock gating methodology. Figure 18.11 shows a simple and safe way of generating enable signals for clock gating. This method gives glitch-free clock signals without the adding of memory elements, as it is needed with DFF clock gating.

Synopsys handles the proposed latch-based design methodology very well. It performs the time borrowing well and appears to analyze correctly the clocks for speed optimization. So it is possible to use this design methodology with Synopsys, although there are a few points of discussion linked with the clock gating.

This clock gating methodology cannot be inserted automatically by Synopsys. The designer has to write the description of the clock gating in his VHDL code. This statement can be generalized to all designs using the above latch-based design methodology. We believe Synopsys can do automatic clock gating for pure double latch design (in which there is no combinatorial logic between the master and slave latch), but such a design causes a loss of speed over similar DFF design.

The most critical problem is to prevent the synthesizer from optimizing the clock gating AND gate with the rest of the combinatorial logic. To ensure a glitch-free clock, this AND gate has to be placed as shown in Fig. 18.11. This can be easily done manually by the designer by placing these AND gates in a separate level of hierarchy of his design or placing a “don’t touch” attribute on them.

Results

A synthesizable by Synopsys CoolRISC-DL 816 core with 16 registers has been designed according to the proposed double latch (DL) scheme (clocks Ø1 and Ø2) and provides the estimated (by Synopsys) following performances (only the core, about 20,000 transistors) in TSMC 0.25 μm :

- 2.5 V, about 60 MIPS (but 120 MHz single clock) (It is the case with the core only, if a program memory with 2 ns of access time is chosen, as the access time is included in the first pipeline stage, the achieved performance is reduced to 50 MIPS.)
- 1.05 V, about 10 $\mu\text{W}/\text{MIPS}$, about 100,000 MIPS/watt (Fig. 18.12)

The core “DFF+Scan” is a previous CoolRISC core designed with flip-flops [19,20]. The CoolRISC-DL “double latch” cores [16] with or without special scan logic provide better performances.

18.6 Low-Power Memories

As memories in SoCs are larger and larger, the ratio between power consumption of memories to the power consumption of embedded processors is significantly increased. Several solutions have been proposed at the memory architecture level, such as, for instance, cache memories, loop buffers, and hierarchical memories, i.e., to store a frequently executed piece of code in a small embedded ROM memory and large but rare executed pieces of code in a large ROM memory [19,20]. It is also possible to read the large ROM in two or four clock cycles as its read access time is too large for the chosen main frequency of the microprocessor.

Cache Memories for Microcontrollers

Cache memories are widely used for high-performance microprocessors. In SoCs, application software is stored in embedded memories, ROM, flash or EEPROM. If a conventional N -way set-associative cache is used, one has to compare the energy used for a ROM access and the energy for a SRAM cache access. While reading N tags and N blocks of the selected cache line just to select one instruction (hundreds of bits), one can see that a conventional cache consumes much more power than a ROM access.

The only way to save power is to use unconventional cache memories such as small L0 caches or buffers, which store only some instructions that are reused frequently from the cache. Such a scheme is used for some DSP processors to capture loops. Furthermore, the tags are read first and in case of a hit, only one way is accessed. So this mechanism avoids reading N blocks in parallel and saves power. Before fetching a block in the cache, one has to check if the instruction is already in the buffer in order to stop the access to the cache or to the ROM if there is a buffer hit (Fig. 18.13). Obviously, the hit rate of such a cache cannot be as high as for a N -way set associative cache of the same size.

To improve the hit rate, one has a supplementary bit (flag) per instruction generated by the compiler (or manually generated) in the main memory indicating (if activated) that this instruction has to be stored in the L0 cache. If this bit is “0”, the instruction, when fetched from L1 cache, is not stored in the L0 cache. It results in the fact that instructions not often used do not pollute the L0 cache (Fig. 18.13). Furthermore, with quite small caches (32–64 instructions), one has also to choose between cacheable instructions, which ones are the most useful to write in cache to reach the highest hit rate. Many algorithms working on program traces have been studied to maximize the number of instruction fetches per write in the cache. The results are strongly dependent on the applications, i.e., a scientific code is better than a control code (Table 18.5).

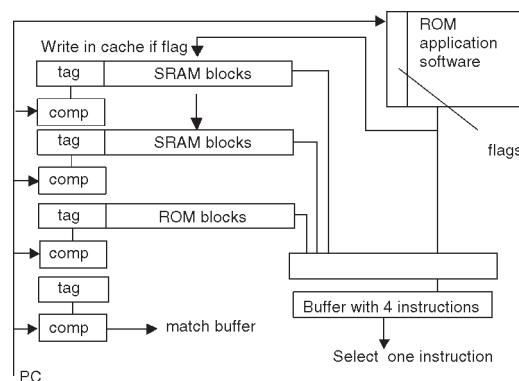


FIGURE 18.13 CoolCache for the CoolRISC microcontroller.

TABLE 18.5 CoolCache Results

Cache Type	1-way Standard Cache 64 Instructions		CoolCache 64 Instructions	
	Hit Rate	Power Gain	Hit Rate	Power Gain
Program sinus	89%	6%	89%	64%
Watch application	68%	-93%	82%	41%

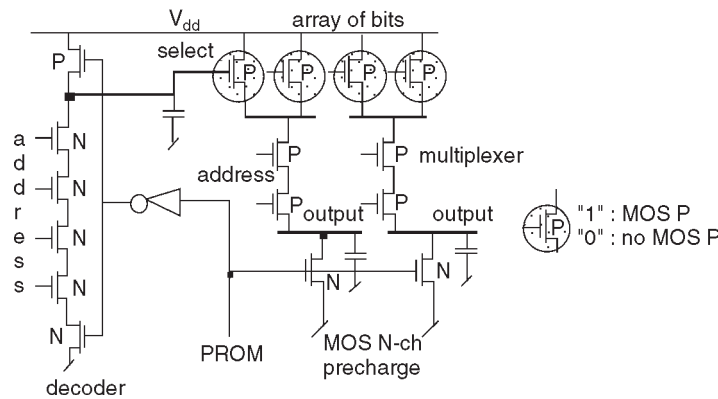


FIGURE 18.14 Old ROM memory structure.

Another idea is to search for instructions or routines that are obviously always in the cache during the program execution (or nearly always in the cache). So it is quite interesting to replace a part of the SRAM cache that is not written by a ROM structure (Fig. 18.13). In so doing, the power consumption is reduced, as a ROM access is less consuming than a RAM access. Furthermore, a ROM cell is smaller than a RAM cell, so there is no need to write something in the memory and a ROM memory is faster than a RAM. Another advantage is the fact that instructions stored in ROM are always in the ROM; it could be interesting for reactivity and real-time routines for which the execution time does not need to be dependent on the fact that this routine is already in or not in the cache.

Electrical Design of Low-Power ROM Memories

Few papers describe very low-power ROM structures. A ROM memory is an array of transistors (a transistor means a logical bit "0"). Each bit is connected to a bitline to read the ROM and controlled by a wordline.

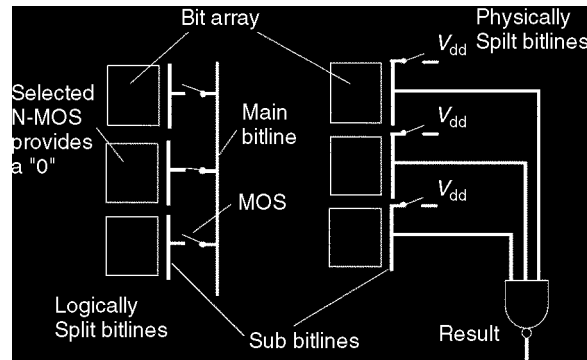
Figure 18.14 shows an old design of a ROM memory. Precharge techniques are used for both the decoder and bit array. After the address is provided, the decoder (NAND gates) is precharged to "1" and the bit array is precharged to "0" through the output multiplexer (PROM = 1). When PROM = 0, the ROM is read. If the P-MOS in the array is there, the bitline is charged to "1," otherwise it keeps its "0" value. Such a memory is very slow: NAND gates for the decoder, P-MOS for the array and long bitlines.

Existing ROM memories use a similar scheme: in a first step the bitlines are precharged to "1," and in the next step a wordline is selected. If there is a transistor (bit is "0") at the intersection of the selected wordline and a bitline, this bitline is discharged. The value of the selected word can therefore be read on the bitlines. Many techniques have been used to speed-up existing ROM memories: NOR gates, N-MOS for the array, to split the memory in several arrays, divided word lines (DWL), to divide the bit line, to precharge only the useful bitlines, partial swing with sense amplifiers. Two of these techniques are described in the following text.

Because a large part of the access time is due to the discharge of the bitline, split bitlines are generally used. Figure 18.15 presents the split bitlines principle. Each bitline is split in sub-bitlines that are

TABLE 18.6 ROM Memories Power Consumption Comparison

Company	Techno (μm)	Size	Supply (V)	Access Time	Power (μAMHz)
Ref. [24]	0.18	8K $\times$ 16	1.8	NA	110
Ref. [24]	0.18	64K $\times$ 32	1.8	NA	335
CSEM	0.25	64 Kbytes	2.5	3 ns	20

**FIGURE 18.15** Split bitlines.

connected to the main bitline (only a wire) with an additional transistor. The selection of these transistors depends on address bits. Only one of the sub-bitlines can be connected to the main bitline. Since only a small fraction of the bit array transistor drains are connected to a given sub-bitline, the capacitance to be discharged is significantly reduced. The problem is that the capacitance of the main bitline is discharged through two transistors in series. Then, in order to have a speed gain, the additional transistors (which connect the sub-bitlines to the main bitlines) must be large. The number of large transistors is the same as the number of bitlines. So, for 1000 bitlines, one has to drive 1000 transistors. It is therefore quite difficult to reduce the access time and the power consumption.

Another solution is also shown in Figure 18.15. The drains of the bit array transistors are connected directly to a sub-bitline that is a single metal line. Figure 18.15 shows three metal lines and therefore three sub-bitlines. To access the ROM, one has to precharge to “1” the three metal lines, and when a transistor of a bit array is selected, only one of the three metal lines switch to “0” (if a “0” is read). A NAND gate can detect the read value. In this solution, each “read” bitline is split only “physically” in the layout, not “logically” as in other solutions.

The second technique aims at removing the sense amplifiers. It allows the ROM memory to be capable of working at any supply voltage. Noise problems are therefore removed, as sense amplifiers are very sensitive to noise. Furthermore, it is a very simple solution, as there is no need to detect when the read operation is finished to disable the sense amplifiers; however, if sense amplifiers are removed, bitlines switch with full swings. One could think that power consumption is largely increased, but the results show that power consumption is reduced. Table 18.6 shows the results in power and speed in a 0.25 μm process.

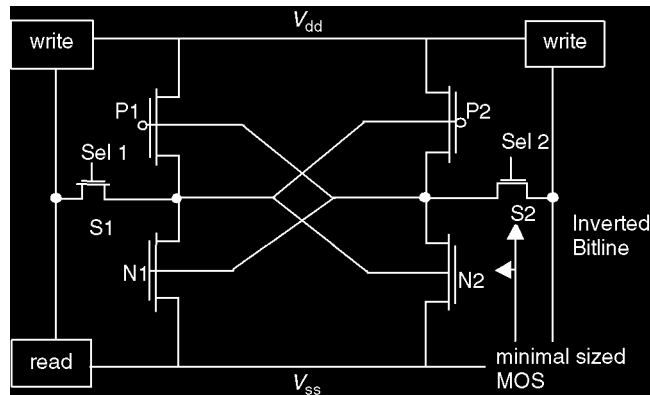
Electrical Design of Low-Power SRAM Memories

Low power and fast SRAM memories have been described in many papers [24]. Very advanced technologies have been used with double V_T . RAM cells are designed with high V_T and selection logic with low V_T transistors. Some techniques such as low swing and hierarchical sense amplifiers have been used. One can also use the fact that a RAM memory is read 85% of the time and written only 15% of the time.

Low-power RAM memories designed by CSEM also use the split bitlines described for the ROM. However, the RAM cell is based on nonsymmetrical ways to read and to write the memory. The patented

TABLE 18.7 Performances of SRAM Memories

Company	Techno (μm)	Size	Supply (V)	Access Time (ns)	Power (μAMHz)
Mirage Logic	0.25	2 K $\times$ 16	2.75	2.5	116
CSEM	0.25	4 K $\times$ 16	2.5	3	33
ST	0.25	8 K $\times$ 16	1.5	3	125
CSEM	0.25	4 K $\times$ 16	1.2	10	14
ST	0.25	8 K $\times$ 16	1.0	5.5	85
NEC	0.25	0.5 K $\times$ 16	0.9	10	16

**FIGURE 18.16** Asymmetrical SRAM cell.

idea is to write in a conventional way while using the true and inverted bitlines, but to read only through a single bitline (Fig. 18.16).

The advantages are the following:

- As it is the case in the conventional scheme, it is possible to write at low V_{dd} since both true and inverted bitlines on both sides of the cell are used.
- The use of only one bitline for reading (instead of two) decreases the power consumption.
- The read condition (to achieve a read and not to overwrite the cell) has only to be effective on one side of the cell, so some transistors can be kept minimal. It decreases the capacitance on the inverted-bitline and the power consumption when writing the RAM. Furthermore, minimal transistors result in a better ratio between cell transistors when reading the memory, resulting in a speed-up of the read mechanism.
- Due to a read process only on one side of the cell, one can use the split bitlines concept more easily (Fig. 18.15).

Table 18.7 shows some results. As mentioned, the CSEM SRAM memory achieves a full swing without any sense amplifier.

18.7 Low-Power Standard Cell Libraries

At the electrical level, digital standard cells have been designed in a robust branch-based logic style, such as hazard-free D-flip-flops [7,25]. Such libraries with 60 functions and 220 layouts have been used for industrial chips. The low-power techniques used were the branch-based logic style that reduces parasitic capacitances and a clever transistor sizing. Instead, to enlarge transistors to have more speed, parasitic capacitances were reduced by reducing the sizes of the transistors on the cell critical paths. If several years ago, power consumption reductions achieved compared to other libraries were about a factor of 3–5, it

TABLE 18.8 Delay Comparison

	Old Library Delay [ns]	μm^2	New Library Delay [ns]	μm^2
32-b multiply	16.4	907 K	12.1	999 K
fp adder	27.7	510 K	21.1	548 K
CoolRISC ALU	10.8	140 K	7.7	170 K

TABLE 18.9 Silicon Area Comparison

	Old Library Delay [ns]	μm^2	New Library Delay [ns]	μm^2
32-b multiply	17.1	868 K	17.0	830 K
fp adder	28.1	484 K	28.0	472 K
CoolRISC ALU	11.0	139 K	11.0	118 K

is today only about a factor of 2 due to a better understanding of power consumption problems of library designers.

Today, logic blocks are automatically synthesized from a VHDL description while considering a design flow using a logic synthesizer such as Synopsys. Furthermore, deep submicron technologies with large wire delays imply a better robustness, mainly for sequential cells sensitive to the clock input slope. Fully static and branch-based logic style has been found as the best; however, a new approach has been proposed that is based on a limited set of standard cells. As a result, the logic synthesizer is more efficient because it has a limited set of cells well chosen and adapted to the considered logic synthesizer. With significantly less cells than conventional libraries, the results show speed, area, and power consumption improvements for synthesized logic blocks. The number of functions for the new library has been reduced to 22 and the number of layouts to 92. Table 18.8 shows that, for a similar silicon area, delays with the new library are reduced. Table 18.9 shows that, for a similar speed, silicon area is reduced for the new library. Furthermore, as the number of layouts is drastically reduced, it takes less time to design a new library for a more advanced process.

Overall, the main issue in the design of future libraries will be the static power. For V_{dd} as low as 0.6–0.3 V in 2014, as predicted by the Roadmap [15], V_T will be reduced accordingly in very deep submicron technologies. Consequently, the static power will increase significantly due to these low V_T [3]. Several techniques with double V_T , source impedance, well polarization, dynamic regulation of V_T [4] are today under investigation and will be necessarily used in the future. This problem is crucial for portable devices that are often in standby mode in which the dynamic power is reduced to zero. It results that the static power becomes the main part of the total power. It will be a crucial point in future libraries for which more versions of the same function will be required while considering these static power problems. A same function could be realized, for instance, with low or high V_T for double V_T technologies, or several cells such as a generic cell with typical V_T , a low-power cell with high V_T , and a fast cell with low V_T .

References

1. A. P. Chandrakasan, S. Sheng, R. W. Brodersen, "Low-Power CMOS Digital Design" *IEEE J. of Solid-State Circuits*, Vol. 27, No. 4, April 1992, pp. 473–484.
2. R. F. Lyon, "Cost, Power, and Parallelism in Speech Signal Processing," *IEEE 1993 CICC*, Paper 15.1.1, San Diego, CA, USA.
3. D. Liu, C. Svensson, "Trading Speed for Low Power by Choice of Supply and Threshold Voltages," *JSSC-28*, No. 1, Jan. 1993, pp. 10–17.
4. V. von Kaenel et al. "Automatic Adjustment of Threshold & Supply Voltage Minimum Power Consumption in CMOS Digital Circuits," *1994 IEEE Symposium on Low Power Electronics*, San Diego, October 10–12, 1994, pp. 78–79.

5. J. Rabay, M. Pedram, "Low Power Design Methodologies," Kluwer Academic Publishers, Dordrecht, the Netherlands, 1996.
6. "Low-Power HF Microelectronics, a unified approach," Editor G. Machado, *IEE Circuits & Systems Series No. 8*, IEE Publishers, 1996.
7. "Low Power Design in Deep Submicron Electronics," *NATO ASI Series, Series E: Applied Sciences*, Vol. 337, Editors W. Nebel, J. Mermet, Kluwer Academic Publishers, 1997.
8. C. Piguet, "Parallelism and Low-Power," invited talk, SympA'99, Symposium Architectures de Machines, Rennes, France, June 8, 1999.
9. A. Jerraya, "Hardware/Software Codesign," Summer Course, Orebro, Sweden, August 14–16, 2000.
10. V. von Kaenel, P. Macken, M. Degrauwe, "A Voltage Reduction Technique for Battery-Operated Systems," *IEEE J. of Solid-State Circuits*, Vol. 25, No. 5, October 1990, pp. 1136–1140.
11. F. Rampogna, J.-M. Masgonty, C. Piguet, "Hardware-Software Co-simulation and Power Estimation Environment for Low-Power ASICs and SoCs," DATE'2000, *Proc. User Forum*, pp. 261–265, Paris, 27–30, March 2000.
12. D. Singh et al., "Power Conscious CAD Tools and Methodologies: A Perspective," *Proc. IEEE*, Vol. 83, No. 4, April 1994, pp. 570–594.
13. J. M. Rabay, "Managing Power Dissipation in the Generation-after-Next Wireless Systems," *FTFC'99*, June 1999, Paris, France.
14. "Billion-Transistor Architectures," *IEEE Computer*, September 1997, Issue.
15. <http://notes.sematech.org/ntrs/rdmpmem.nsf>
16. C. Arm, J.-M. Masgonty, C. Piguet, "Double-Latch Clocking Scheme for Low-Power I.P. Cores," *PATMOS 2000*, Goettingen, Germany, September 13–15, 2000.
17. M. Renaudin, "Asynchronous Circuits and Systems: a promising design alternative," MIGAS Summer School on Microelectronics for Telecommunications: Managing High Complexity and Mobility, June 28–July 4, 2000, Autrans, France.
18. C. Piguet et al., "Low-Power Digital Design and CAD Tools," invited talk, *Colloque CAO de circuits intégrés et systèmes*, Aix en Provence, May 10–12, 1999, pp. 108–127.
19. C. Piguet et al., "Low-Power Design of 8-bit Embedded CoolRISC Microcontroller Cores," *IEEE JSSC*, Vol. 32, No. 7, July 1997, pp. 1067–1078.
20. J.-M. Masgonty et al., "Low-Power Design of an Embedded Microprocessor," *ESSCIRC'96*, September 16–21, 1996, Neuchâtel, Switzerland.
21. M. Keating, P. Bricaud, "Reuse Methodology Manual," Kluwer Academic Publishers, 1999.
22. Ph. Mosch et al., "A 72 μ W, 50 MOPS, 1 V DSP for a hearing aid chip set," *ISSCC'00*, San Francisco, February 7–9, Session 14, paper 5.
23. J. G. Xi, D. Staepelaere, "Using Clock Skew as a Tool to Achieve Optimal Timing," *Integrated System Magazine*, April 1999, webmaster@isdmag.com
24. A. Turier, Ph.D. Thesis, "Etude, Conception et Caractérisation de mémoires CMOS, faible consommation, faible tension en technologies submicroniques," University of Paris VI, Pierre and Marie Curie, France, December 13, 2000.
25. C. Piguet, J. Zahnd, "Signal-Transition Graphs-Based Design of Speed-Independent CMOS Circuits," *ESSCIRC'98*, September 21–24, 1998, Den Haag, The Netherlands, pp. 432–435.

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
	CPU • 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.

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

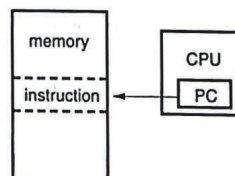


FIGURE 22.1 A CPU and memory.

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. ARM² and MIPS³ 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 C6000⁵ 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 specification⁷ 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 it 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

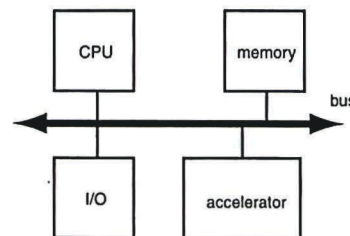


FIGURE 22.2 A bus-based, single-CPU embedded system.

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 designed 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

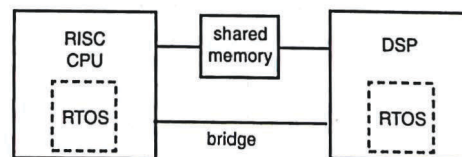


FIGURE 22.3 The TI OMAP architecture.¹⁰

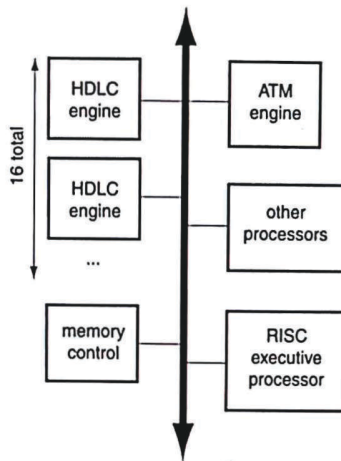


FIGURE 22.4 Block diagram of the C-Port network processor.¹¹

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. I²C 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 language¹³ is widely used to specify protocols in telecommunications systems. The Esterel language¹⁴ describes a reactive system as a network of communicating state machines.

Data-oriented languages find their greatest use in signal processing systems. Dataflow process networks¹⁵ 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

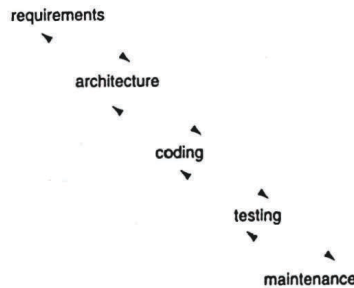


FIGURE 22.5 The waterfall model of software development.

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. Vulcan²¹ and Cosyma²² 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-simulator¹⁹ 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 development 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.