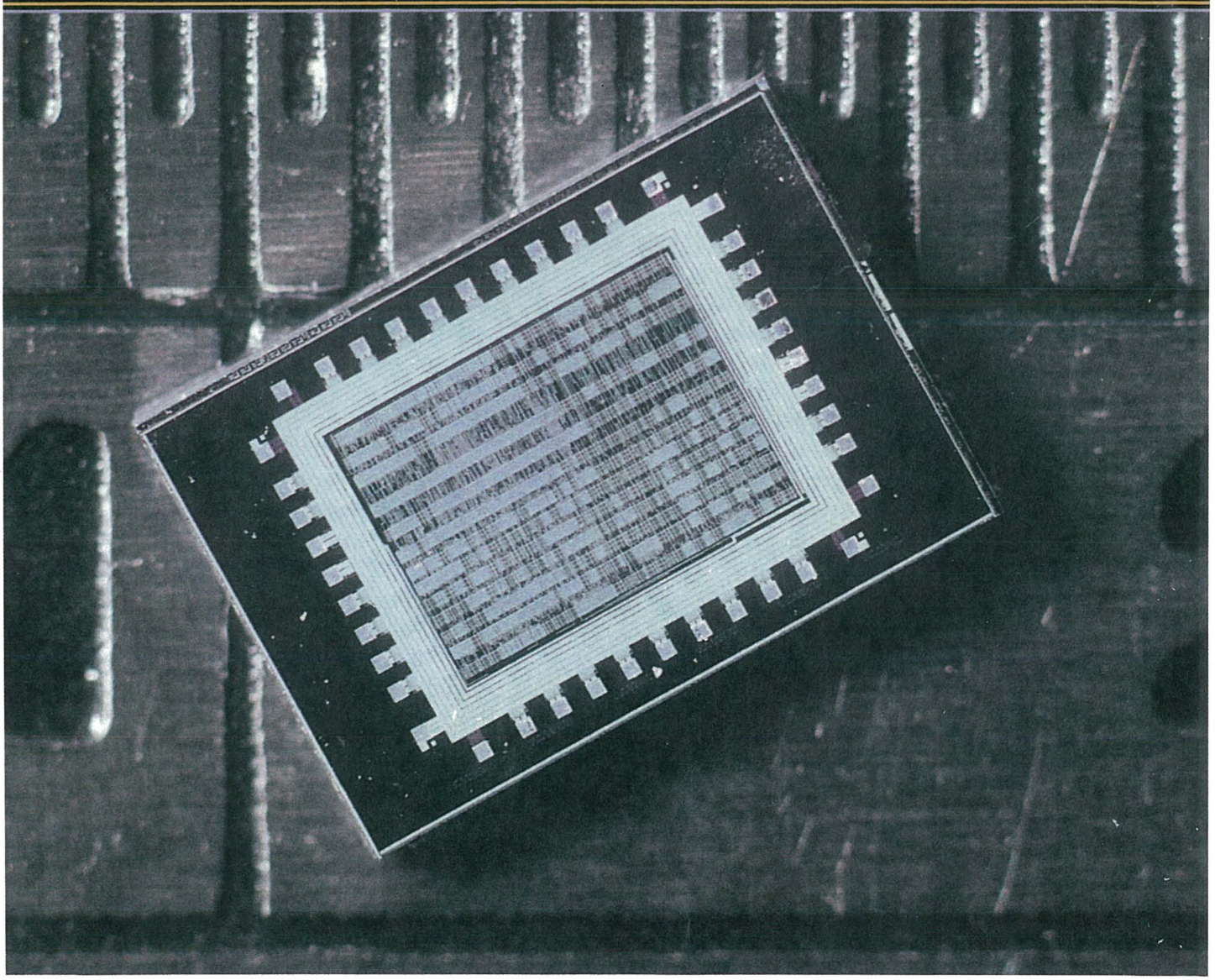
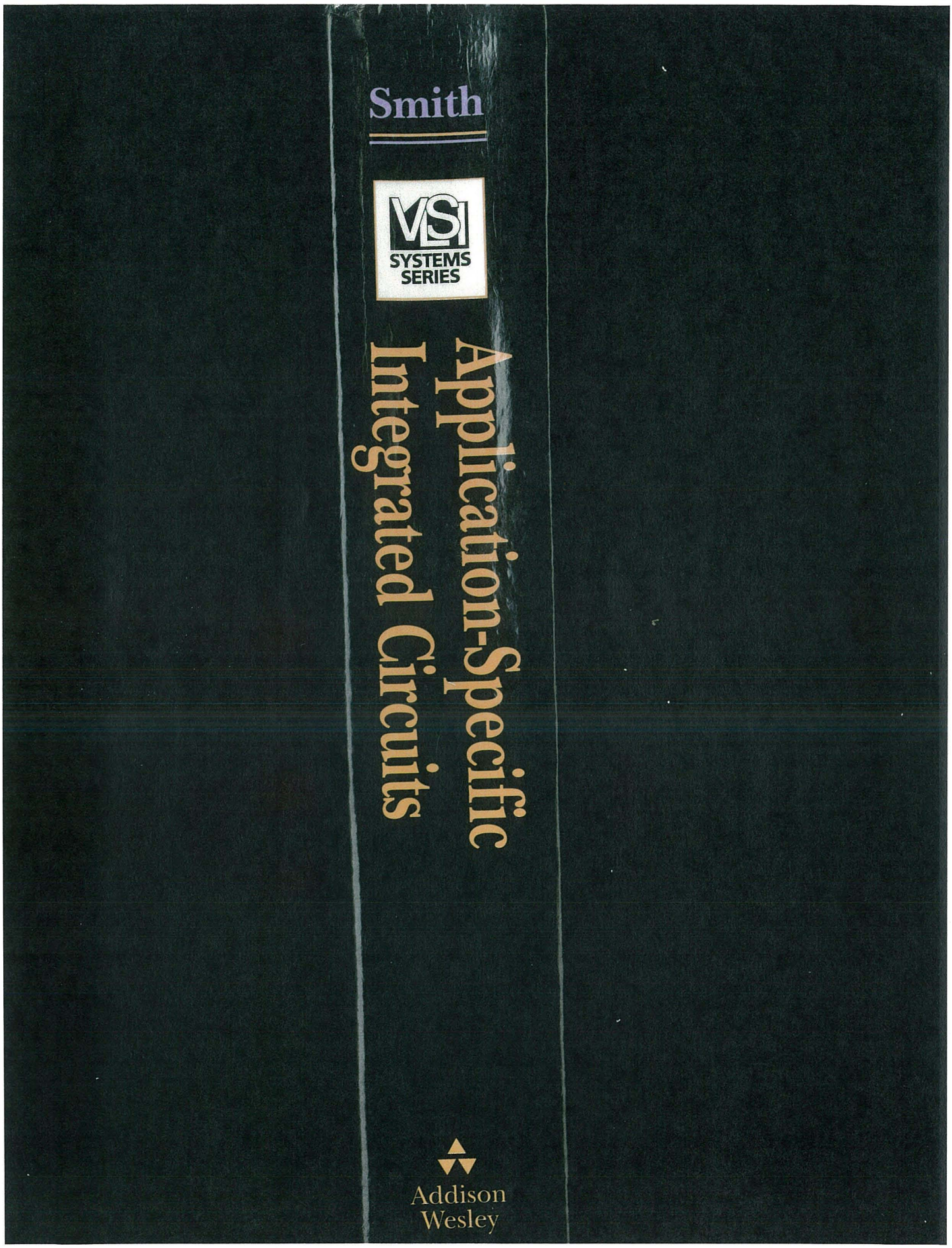
Application-Specific Integrated Circuits

Michael John Sebastian Smith




Smith
VLSI
SYSTEMS
SERIES
Application-Specific
Integrated Circuits
Addison
Wesley

Brinkmann

Application-Specific Integrated Circuits

Michael John Sebastian Smith

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sidney • Tokyo • Singapore • Mexico City

This book is in the Addison-Wesley VLSI Systems Series
Lynn Conway and Charles Seitz, *Consulting Editors*

Sponsoring Editor	Peter Gordon
Associate Editor	Helen Goldstein
Senior Production Supervisor	Juliet Silveri
Copyeditor/Proofreader	Cynthia Benn
Cover Design Supervisor	Simone Payment
Marketing Manager	Tracy Russ
Manufacturing Manager	Roy Logan

Library of Congress Cataloging-in-Publication Data

Smith, Michael J. S. (Michael John Sebastian)
Application-specific integrated circuits / Michael J.S. Smith.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-50022-1
1. Application-specific integrated circuits. I. Title.
TK7874.6.S63 1997
621.39'5--dc20 93-32538
CIP

ISBN 0-201-50022-1
Text printed on recycled paper
8 9 10 11 12 13 14—CRW—0403020100
8th printing, May 2000

PREFACE

In 1988 I began to teach full-custom VLSI design. In 1990 I started teaching ASIC design instead, because my students found it easier to get jobs in this field. I wrote a proposal to The National Science Foundation (NSF) to use electronic distribution of teaching material. Dick Lyon helped me with preparing the first few CD-ROMs at Apple, but Chuck Seitz, Lynn Conway, and others explained to me that I was facing a problem that Carver Mead and Lynn had experienced in trying to get the concept of multichip wafers adopted. It was not until the publication of the Mead–Conway text that people accepted this new idea. It was suggested that I must generate interest using a conventional format before people would use my material in a new one (CD-ROM or the Internet). In 1992 I stopped writing papers and began writing this book—a result of my experiments in computer-based education. I have nearly finished this book twice. The first time was a copy of my notes. The second time was just before the second edition of Weste and Eshragian was published—a hard act to follow. In order to finish in 1997 I had to stop updating and including new ideas and material and now this book consists of three parts: Chapters 1–8 are an introduction to ASICs, 9–14 cover ASIC logical design, and 15–17 cover the physical design of ASICs.

The book is intended for a wide audience. It may be used in an undergraduate or graduate course. It is also intended for those in industry who are involved with ASICs. Another function of this book is an "ASIC Encyclopedia," and therefore I have kept the background material needed to a minimum. The book makes extensive use of industrial tools and examples. The examples in Chapters 2 and 3 use tools and libraries from MicroSim (PSpice), Meta Software (HSPICE), Compass Design Automation (standard-cell and gate-array libraries), and Tanner Research (L-Edit). The programmable ASIC design examples in Chapter 4–8 use tools from Compass, Synopsys, Actel, Altera, and Xilinx. The examples in Chapter 9 (covering low-level design entry) used tools from Exemplar, MINC, AMD, UC Berkeley, Compass, Capilano, Mentor Graphics Corporation, and Cadence Design Systems. The VHDL examples in Chapter 10 (VHDL) were checked using QuickVHDL from Mentor, V-System Plus from Model Technology, and Scout from Compass. The Verilog examples in Chapter 11 were checked using Verilog-XL from Cadence, V-System Plus, and VeriWell from Wellspring Solutions. The logic synthesis examples in

Chapter 12 were checked with the ASIC Synthesizer product family from Compass and tools from Mentor, Synopsys, and UC Berkeley. The simulation examples in Chapter 13 were checked with QuickVHDL, V-System/Plus, PSpice, Verilog-XL, DesignWorks from Capilano Computing, CompassSim, QSim, MixSim, and HSPICE. The test examples in Chapter 14 were checked using test software from Compass, Cadence, Mentor, Synopsys and Capilano's DesignWorks. The physical design examples in Chapters 15–17 were generated and tested using Preview, Gate Ensemble, and Cell Ensemble (Cadence) as well as ChipPlanner, ChipCompiler, and PathFinder (Compass). All these tools are installed at the University of Hawaii.

I wrote the text using FrameMaker. This allows me to project the text and figures using an LCD screen and an overhead projector. I used a succession of Apple Macintosh computers: a PowerBook 145, a 520, and lastly a 3400 with 144MB of RAM, which made it possible for me to create updates to the index in just under one minute. Equations are "live" in FrameMaker. Thus,

$$\text{book thickness} = \#\text{pages} \times 0.0015 \text{ in./page} \approx (1000)\ (1.5 \times 10^{-3}) = 1.5 \text{ in.}$$

can be updated in a lecture and the new result displayed. The circuit layouts are color EPS files with enhanced B&W PICT previews created using L-Edit from Tanner Research. All of the Verilog and VHDL code examples, compiler and simulation input/output, and the layout CIF that were used in the final version are included as conditional (hidden) text in the FrameMaker document, which is approximately 200MB and just over 6,000 pages (my original source material spans fourteen 560MB optical disks). Software can operate on the hidden text, allowing, for example, a choice of simulators to run the HDL code live in class. I converted draft versions of the VHDL and Verilog LRMs and related standards to FrameMaker and built hypertext links to my text, but copyright problems will have to be solved before this type of material may be published. I drew all the figures using FreeHand. They are "layered" allowing complex drawings to be built-up slowly or animated by turning layers on or off. This is difficult to utilize in book form, but can be done live in the classroom.

A course based on FPGAs can use Chapter 1 and Chapters 4–8. A course using commercial semicustom ASIC design tools may use Chapters 1–2 or Chapters 1–3 and then skip to Chapter 9 if you use schematic entry, Chapter 10 (if you use VHDL), or Chapter 11 (if you use Verilog) together with Chapter 12. All classes can use Chapters 13 and 14. FPGA-based classes may skim Chapters 15–17, but classes in semicustom design should cover these chapters. The chapter dependencies—Y (X) means Chapter Y depends on X—are approximately: 1, 2(1), 3(2), 4(2), 5(4), 6(5), 7(6), 8(7), 9(2), 10(2), 11(2), 12(10 or 11), 13(2), 14(13), 15(2), 16(15), 17(16).

I used the following references to help me with the orthography of complex terms, style, and punctuation while writing: *Merriam-Webster's Collegiate Dictionary,* 10th edition, 1996, Springfield, MA: Merriam-Webster, ISBN 0-87779-709-9, PE1628.M36; *The Chicago Manual of Style,* 14th edition, Chicago: University of

Chicago Press, 1993, ISBN 0-226-10389-7, Z253.U69; and *Merriam-Webster's Standard American Style Manual,* 1985, Springfield, MA: Merriam-Webster, ISBN 0-87779-133-3, PN147.W36. A particularly helpful book on technical writing is *BUGS in Writing* by Lyn Dupré, 1995, Reading, MA: Addison-Wesley, ISBN 0-201-60019-6, PE1408.D85 (Lyn's book grew from her unpublished work, Style SomeX, which I used).

The bibliography at the end of each chapter provides alternative sources if you cannot find what you are looking for. I have included the International Standard Book Number[1] (ISBN) and Library of Congress (LOC) Call Number for books, and the International Standard Serial Number[2] (ISSN) for journals (see the LOC information system, LOCIS, at `http://www.loc.gov`). I did not include references to material that I could not find myself (except where I have noted in the case of new or as yet unpublished books). The electronic references given in this text have (a last) access date of 4/19/97 and omit enclosing <> if the reference does not include spaces.

I receive a tremendous level of support and cooperation from industry in my work. I thank the following for help with this project: Cynthia Benn and Lyn Dupré for editing; Helen Goldstein, Peter Gordon, Susan London-Payne, Tracy Russ, and Juliet Silveri, all at Addison-Wesley; Matt Bowditch and Kim Arney at Argosy; Richard Lyon, Don North, William Rivard, Glen Stone, the managers of the Newton group, and many others at Apple Computer who provided financial support; Apple for providing support in the form of software and computers; Bill Becker, Fern Forcier, Donna Isidro, Mike Kliment, Paul McLellan, Tom Schaefer, Al Stein, Rich Talburt, Bill Walker, and others at Compass Design Automation and VLSI Technology for providing the opportunity for me to work on this book over many years and allowing me to test material inside these companies and on lecture tours they sponsored; Chuck Seitz at Caltech; Joseph Cavallaro, Bernie Chern, Jerry Dillion, Mike Foster, and Paul Hulina at the NSF; the NSF for financial support with a Presidential Young Investigator Award; Jim Rowson and Doug Fairbairn; Constantine Anagnostopolous, Pin Tschang and members of the ASIC design groups at Kodak for financial support; the disk-drive design group at Digital Equipment Corp. (Massachusetts), Hewlett-Packard, and Sun Microsystems for financial support; Ms. MOSIS and all of the staff at MOSIS who each have helped me at one point or another by providing silicon, technical support, and documentation; Bob Brodersen, Roger Howe, Randy Katz, and Ed Lee of UC Berkeley for help while I was visiting UCB; James Plummer of Stanford, for providing me with access to the Terman Engineering Library as a visiting scholar, as well as Abbas El Gamal and Paul Losleben, also at Stanford, for help on several occasions; Don Bouldin at University of Tennessee; Krzysztof Kozminski at MCNC for providing Uncle lay-

[1] A code that uniquely identifies a book, the tenth and last digit is a check digit.

[2] This number uniquely identifies a serial (a magazine, a journal, and so on). It is a seven-digit number with an eighth check digit (which may be the roman numeral X, the value ten).

out software; Gershom Kedem at Duke University for the public domain tools his group has written; Sue Drouin, José De Castro, and others at Mentor Graphics Corporation in Oregon for providing documentation and tools; Vahan Kasardjhan, Gail Grego, Michele Warthen, Steve Gardner, and others at the University Program at Cadence Design Systems in San Jose who helped with tools, documentation, and support; Karen Dorrington and the Cadence group in Massachusetts; Andy Haines, Tom Koppin, Sherri Mieth, Velma Miller, Robert Nalesnik, Mike Sarpa, Telle Whitney, and others at Actel for software, hardware, parts, and documentation; Peter Alfke, Leslie Baxter, Brad Fawcett, Chris Kingsley, Karlton Lau, Rick Mitchell, Scott Nance, and Richard Ravel at Xilinx for support, parts, software, and documentation; Greg Hedmann at NorComp for data on FPGAs; Anna Acevedo, Suzanne Bailey, Antje MacNaughton, Richard Terrell, and Altera for providing software, hardware programmers, parts, and documentation; the documentation group and executive management at LSI Logic for tools, libraries, and documentation; Toshiba, NEC, AT&T/NCR, Lucent, and Hitachi (for documentation); NEC for their visiting scholar program at UH; Fred Furtek, Oscar Naval, and Claire Pinkham at Concurrent Logic, Randy Fish at Crosspoint, and Gary Banta at Plus Logic—all for documentation; Paul Titchener and others at Comdisco (now part of Cadence Design Systems) for providing design tools; John Tanner and his staff at Tanner Research for providing their tools and documentation; Mahendra Jain and Nanci Magoun, who let me debug early prototypes at the IDEA conference organized by ASIC Technology and News; Exemplar for providing documentation on its tools; MINC for providing a copy of its FPGA software and documentation; Claudia Traver and Synopsys for tools and documentation; Mentor Graphics Corporation for providing its complete range of software; Alain Hanover and others at ViewLogic for providing tools; Mary Shepherd and Jerry Walker at IEEE for help with permissions; Meta Software for providing HSPICE; Chris Dewhurst and colleagues at Capilano Computing for its design tools; Greg Seltzer (Model Technology) and Charley Rowley for providing V-System Plus with online documentation prototypes; Farallon and Telebit for the software and hardware I used for early experiments with telelectures. Many research students at the University of Hawaii helped me throughout this project including: Chin Huang, Clem Portmann, Christeen Gray, Karlton Lau, Jon Otaguro, Moe Lwin, Troy Stockstad, Ron Jorgenson, Derwin Mattos, William Rivard, Wendy Ching, Anil Aggarwal, Sudhakar Jilla, Linda Xu, Angshuman Saha, Harish Pareek, Claude van Ham, Wen Huang, Kumar Vadhri, Yan Zhong, Yatin Acharya, and Barana Ranaweera. Each of the classes that used early versions of this text at the University of Hawaii at Manoa have also contributed by finding errors. The remaining errors are mine.

Links to figures, software, code, problem solutions, and other resources for this book may be found at:

`http://www.awl.com/cp/authors/smithm/asics/asics.html`.

Michael John Sebastian Smith
Palo Alto and Honolulu, 1997

CONTENTS

INTRODUCTION TO ASICs

1

An **ASIC** (pronounced "a-sick"; bold typeface defines a new term) is an **application-specific integrated circuit**—at least that is what the acronym stands for. Before we answer the question of what *that* means we first look at the evolution of the silicon chip or **integrated circuit** (**IC**).

Figure 1.1(a) shows an IC package (this is a pin-grid array, or PGA, shown upside down; the pins will go through holes in a printed-circuit board). People often call the package a chip, but, as you can see in Figure 1.1(b), the silicon chip itself (more properly called a **die**) is mounted in the cavity under the sealed lid. A PGA package is usually made from a ceramic material, but plastic packages are also common.

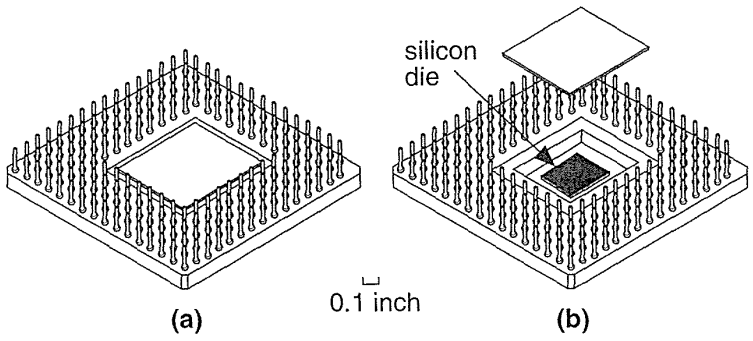


FIGURE 1.1 An integrated circuit (IC). (a) A pin-grid array (PGA) package. (b) The silicon die or chip is under the package lid.

The physical size of a silicon die varies from a few millimeters on a side to over 1 inch on a side, but instead we often measure the size of an IC by the number of logic gates or the number of transistors that the IC contains. As a unit of measure a **gate equivalent** corresponds to a two-input NAND gate (a circuit that performs the logic function, $F = \overline{A \cdot B}$). Often we just use the term *gates* instead of gate equivalents when we are measuring chip size—not to be confused with the gate terminal of a transistor. For example, a 100 k-gate IC contains the equivalent of 100,000 two-input NAND gates.

The semiconductor industry has evolved from the first ICs of the early 1970s and matured rapidly since then. Early **small-scale integration** (**SSI**) ICs contained a few (1 to 10) logic gates—NAND gates, NOR gates, and so on—amounting to a few tens of transistors. The era of **medium-scale integration** (**MSI**) increased the range of integrated logic available to counters and similar, larger scale, logic functions. The era of **large-scale integration** (**LSI**) packed even larger logic functions, such as the first microprocessors, into a single chip. The era of **very large-scale integration** (**VLSI**) now offers 64-bit microprocessors, complete with cache memory and floating-point arithmetic units—well over a million transistors—on a single piece of silicon. As CMOS process technology improves, transistors continue to get smaller and ICs hold more and more transistors. Some people (especially in Japan) use the term **ultralarge scale integration** (**ULSI**), but most people stop at the term VLSI; otherwise we have to start inventing new words.

The earliest ICs used **bipolar technology** and the majority of logic ICs used either **transistor–transistor logic** (**TTL**) or emitter-coupled logic (ECL). Although invented before the bipolar transistor, the **metal-oxide-silicon** (**MOS**) transistor was initially difficult to manufacture because of problems with the oxide interface. As these problems were gradually solved, metal-gate *n*-channel MOS (**nMOS** or **NMOS**) technology developed in the 1970s. At that time MOS technology required fewer masking steps, was denser, and consumed less power than equivalent bipolar ICs. This meant that, for a given performance, an MOS IC was cheaper than a bipolar IC and led to investment and growth of the MOS IC market.

By the early 1980s the aluminum gates of the transistors were replaced by polysilicon gates, but the name MOS remained. The introduction of polysilicon as a gate material was a major improvement in CMOS technology, making it easier to make two types of transistors, *n*-channel MOS and *p*-channel MOS transistors, on the same IC—a **complementary MOS** (**CMOS**, never cMOS) technology. The principal advantage of CMOS over NMOS is lower power consumption. Another advantage of a polysilicon gate was a simplification of the fabrication process, allowing devices to be scaled down in size.

There are four CMOS transistors in a two-input NAND gate (and a two-input NOR gate too), so to convert between gates and transistors, you multiply the number of gates by 4 to obtain the number of transistors. We can also measure an IC process by the smallest resolvable **feature size** (roughly half the length of the smallest transistor) imprinted on the IC. Transistor dimensions are measured in microns (a

micron, 1 μm, is a millionth of a meter). Thus we talk about a 0.2 μm IC or say an IC is built in (or with) a 0.2 μm process, meaning that the smallest transistors are approximtely 0.2 μm in length. We give a special label, λ or **lambda**, to this smallest resolvable feature size. Since lambda is roughly equal to half of the smallest transistor length, $\lambda \approx 0.1$ μm in a 0.2 μm process. Many of the drawings in this book use a scale marked with lambda for the same reason we place a scale on a map.

A modern submicron CMOS process is now just as complicated as a submicron bipolar or BiCMOS (a combination of bipolar and CMOS) process. However, CMOS ICs have established a dominant position, are manufactured in much greater volume than any other technology, and therefore, because of the economy of scale, the cost of CMOS ICs is less than a bipolar or BiCMOS IC for the same function. Bipolar and BiCMOS ICs are still used for special needs. For example, bipolar technology is generally capable of handling higher voltages than CMOS. This makes bipolar and BiCMOS ICs useful in power electronics, cars, telephone circuits, and so on.

Some digital logic ICs and their analog counterparts (analog/digital converters, for example) are **standard parts**, or standard ICs. You can select standard ICs from catalogs and data books and buy them from distributors. Systems manufacturers and designers can use the same standard part in a variety of different **microelectronic systems** (systems that use microelectronics or ICs).

With the advent of VLSI in the 1980s engineers began to realize the advantages of designing an IC that was customized or tailored to a particular system or application rather than using standard ICs alone. Microelectronic system design then becomes a matter of defining the functions that you can implement using standard ICs and then implementing the remaining logic functions (sometimes called **glue logic**) with one or more **custom ICs**. As VLSI became possible you could build a system from a smaller number of components by combining many standard ICs into a few custom ICs. Building a microelectronic system with fewer ICs allows you to reduce cost and improve reliability.

Of course, there are many situations in which it is not appropriate to use a custom IC for each and every part of an microelectronic system. If you need a large amount of memory, for example, it is still best to use standard memory ICs, either **dynamic random-access memory** (**DRAM** or dRAM), or **static RAM** (**SRAM** or sRAM), in conjunction with custom ICs.

One of the first conferences to be devoted to this rapidly emerging segment of the IC industry was the *IEEE Custom Integrated Circuits Conference* (CICC), and the proceedings of this annual conference form a useful reference to the development of custom ICs. As different types of custom ICs began to evolve for different types of applications, these new ICs gave rise to a new term: application-specific IC, or ASIC. Now we have the *IEEE International ASIC Conference*, which tracks advances in ASICs separately from other types of custom ICs. Although the exact definition of an ASIC is difficult, we shall look at some examples to help clarify what people in the IC industry understand by the term.

Examples of ICs that are *not* ASICs include standard parts such as: memory chips sold as a commodity item—ROMs, DRAM, and SRAM; microprocessors; TTL or TTL-equivalent ICs at SSI, MSI, and LSI levels.

Examples of ICs that *are* ASICs include: a chip for a toy bear that talks; a chip for a satellite; a chip designed to handle the interface between memory and a microprocessor for a workstation CPU; and a chip containing a microprocessor as a cell together with other logic.

As a general rule, if you can find it in a data book, then it is probably not an ASIC, but there are some exceptions. For example, two ICs that might or might not be considered ASICs are a controller chip for a PC and a chip for a modem. Both of these examples are specific to an application (shades of an ASIC) but are sold to many different system vendors (shades of a standard part). ASICs such as these are sometimes called **application-specific standard products** (**ASSPs**).

Trying to decide which members of the huge IC family are application-specific is tricky—after all, every IC has an application. For example, people do not usually consider an application-specific microprocessor to be an ASIC. I shall describe how to design an ASIC that may include large cells such as microprocessors, but I shall not describe the design of the microprocessors themselves. Defining an ASIC by looking at the application can be confusing, so we shall look at a different way to categorize the IC family. The easiest way to recognize people is by their faces and physical characteristics: tall, short, thin. The easiest characteristics of ASICs to understand are physical ones too, and we shall look at these next. It is important to understand these differences because they affect such factors as the price of an ASIC and the way you design an ASIC.

1.1 Types of ASICs

ICs are made on a thin (a few hundred microns thick), circular silicon **wafer**, with each wafer holding hundreds of die (sometimes people use dies or dice for the plural of die). The transistors and wiring are made from many layers (usually between 10 and 15 distinct layers) built on top of one another. Each successive **mask layer** has a pattern that is defined using a **mask** similar to a glass photographic slide. The first half-dozen or so layers define the transistors. The last half-dozen or so layers define the metal wires between the transistors (the **interconnect**).

A **full-custom IC** includes some (possibly all) logic cells that are customized and all mask layers that are customized. A microprocessor is an example of a full-custom IC—designers spend many hours squeezing the most out of every last square micron of microprocessor chip space by hand. Customizing all of the IC features in this way allows designers to include analog circuits, optimized memory cells, or mechanical structures on an IC, for example. Full-custom ICs are the most expen-

sive to manufacture and to design. The **manufacturing lead time** (the time it takes just to make an IC—not including design time) is typically eight weeks for a full-custom IC. These specialized full-custom ICs are often intended for a specific application, so we might call some of them full-custom ASICs.

We shall discuss full-custom ASICs briefly next, but the members of the IC family that we are more interested in are **semicustom ASICs**, for which all of the logic cells are predesigned and some (possibly all) of the mask layers are customized. Using predesigned cells from a **cell library** makes our lives as designers much, much easier. There are two types of semicustom ASICs that we shall cover: standard-cell–based ASICs and gate-array–based ASICs. Following this we shall describe the **programmable ASICs**, for which all of the logic cells are predesigned and none of the mask layers are customized. There are two types of programmable ASICs: the programmable logic device and, the newest member of the ASIC family, the field-programmable gate array.

1.1.1 Full-Custom ASICs

In a **full-custom ASIC** an engineer designs some or all of the logic cells, circuits, or layout specifically for one ASIC. This means the designer abandons the approach of using pretested and precharacterized cells for all or part of that design. It makes sense to take this approach only if there are no suitable existing cell libraries available that can be used for the entire design. This might be because existing cell libraries are not fast enough, or the logic cells are not small enough or consume too much power. You may need to use full-custom design if the ASIC technology is new or so specialized that there are no existing cell libraries or because the ASIC is so specialized that some circuits must be custom designed. Fewer and fewer full-custom ICs are being designed because of the problems with these special parts of the ASIC. There is one growing member of this family, though, the mixed analog/digital ASIC, which we shall discuss next.

Bipolar technology has historically been used for precision analog functions. There are some fundamental reasons for this. In all integrated circuits the matching of component characteristics between chips is very poor, while the matching of characteristics between components on the same chip is excellent. Suppose we have transistors T1, T2, and T3 on an analog/digital ASIC. The three transistors are all the same size and are constructed in an identical fashion. Transistors T1 and T2 are located adjacent to each other and have the same orientation. Transistor T3 is the same size as T1 and T2 but is located on the other side of the chip from T1 and T2 and has a different orientation. ICs are made in batches called wafer lots. A **wafer lot** is a group of silicon wafers that are all processed together. Usually there are between 5 and 30 wafers in a lot. Each wafer can contain tens or hundreds of chips depending on the size of the IC and the wafer.

If we were to make measurements of the characteristics of transistors T1, T2, and T3 we would find the following:

- Transistors T1 will have virtually identical characteristics to T2 on the same IC. We say that the transistors **match** well or the **tracking** between devices is excellent.
- Transistor T3 will match transistors T1 and T2 on the same IC very well, but not as closely as T1 matches T2 on the same IC.
- Transistor T1, T2, and T3 will match fairly well with transistors T1, T2, and T3 on a different IC on the same wafer. The matching will depend on how far apart the two ICs are on the wafer.
- Transistors on ICs from different wafers in the same wafer lot will not match very well.
- Transistors on ICs from different wafer lots will match very poorly.

For many analog designs the close matching of transistors is crucial to circuit operation. For these circuit designs pairs of transistors are used, located adjacent to each other. Device physics dictates that a pair of bipolar transistors will always match more precisely than CMOS transistors of a comparable size. Bipolar technology has historically been more widely used for full-custom analog design because of its improved precision. Despite its poorer analog properties, the use of CMOS technology for analog functions is increasing. There are two reasons for this. The first reason is that CMOS is now by far the most widely available IC technology. Many more CMOS ASICs and CMOS standard products are now being manufactured than bipolar ICs. The second reason is that increased levels of integration require mixing analog and digital functions on the same IC: this has forced designers to find ways to use CMOS technology to implement analog functions. Circuit designers, using clever new techniques, have been very successful in finding new ways to design analog CMOS circuits that can approach the accuracy of bipolar analog designs.

1.1.2 Standard-Cell–Based ASICs

A **cell-based ASIC** (cell-based IC, or **CBIC**—a common term in Japan, pronounced "sea-bick") uses predesigned logic cells (AND gates, OR gates, multiplexers, and flip-flops, for example) known as **standard cells**. We could apply the term CBIC to any IC that uses cells, but it is generally accepted that a cell-based ASIC or CBIC means a standard-cell–based ASIC.

The standard-cell areas (also called flexible blocks) in a CBIC are built of rows of standard cells—like a wall built of bricks. The standard-cell areas may be used in combination with larger predesigned cells, perhaps microcontrollers or even microprocessors, known as **megacells**. Megacells are also called megafunctions, full-custom blocks, system-level macros (SLMs), fixed blocks, cores, or Functional Standard Blocks (FSBs).

The ASIC designer defines only the placement of the standard cells and the interconnect in a CBIC. However, the standard cells can be placed anywhere on the silicon; this means that all the mask layers of a CBIC are customized and are unique to a particular customer. The advantage of CBICs is that designers save time, money, and reduce risk by using a predesigned, pretested, and precharacterized **standard-cell library**. In addition each standard cell can be optimized individually. During the design of the cell library each and every transistor in every standard cell can be chosen to maximize speed or minimize area, for example. The disadvantages are the time or expense of designing or buying the standard-cell library and the time needed to fabricate all layers of the ASIC for each new design.

Figure 1.2 shows a CBIC (looking down on the die shown in Figure 1.1b, for example). The important features of this type of ASIC are as follows:

- All mask layers are customized—transistors and interconnect.
- Custom blocks can be embedded.
- Manufacturing lead time is about eight weeks.

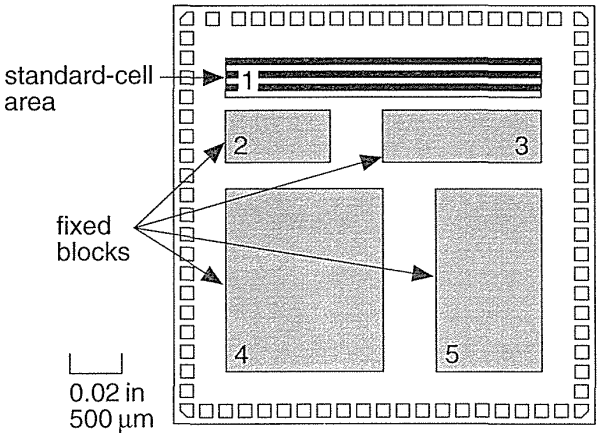


FIGURE 1.2 A cell-based ASIC (CBIC) die with a single standard-cell area (a flexible block) together with four fixed blocks. The flexible block contains rows of standard cells. This is what you might see through a low-powered microscope looking down on the die of Figure 1.1(b). The small squares around the edge of the die are bonding pads that are connected to the pins of the ASIC package.

Each standard cell in the library is constructed using full-custom design methods, but you can use these predesigned and precharacterized circuits without having to do any full-custom design yourself. This design style gives you the same performance and flexibility advantages of a full-custom ASIC but reduces design time and reduces risk.

Standard cells are designed to fit together like bricks in a wall. Figure 1.3 shows an example of a simple standard cell (it is simple in the sense it is not maximized for density—but ideal for showing you its internal construction). Power and ground buses (VDD and GND or VSS) run horizontally on metal lines inside the cells.

Standard-cell design allows the automation of the process of assembling an ASIC. Groups of standard cells fit horizontally together to form rows. The rows stack vertically to form flexible rectangular blocks (which you can reshape during

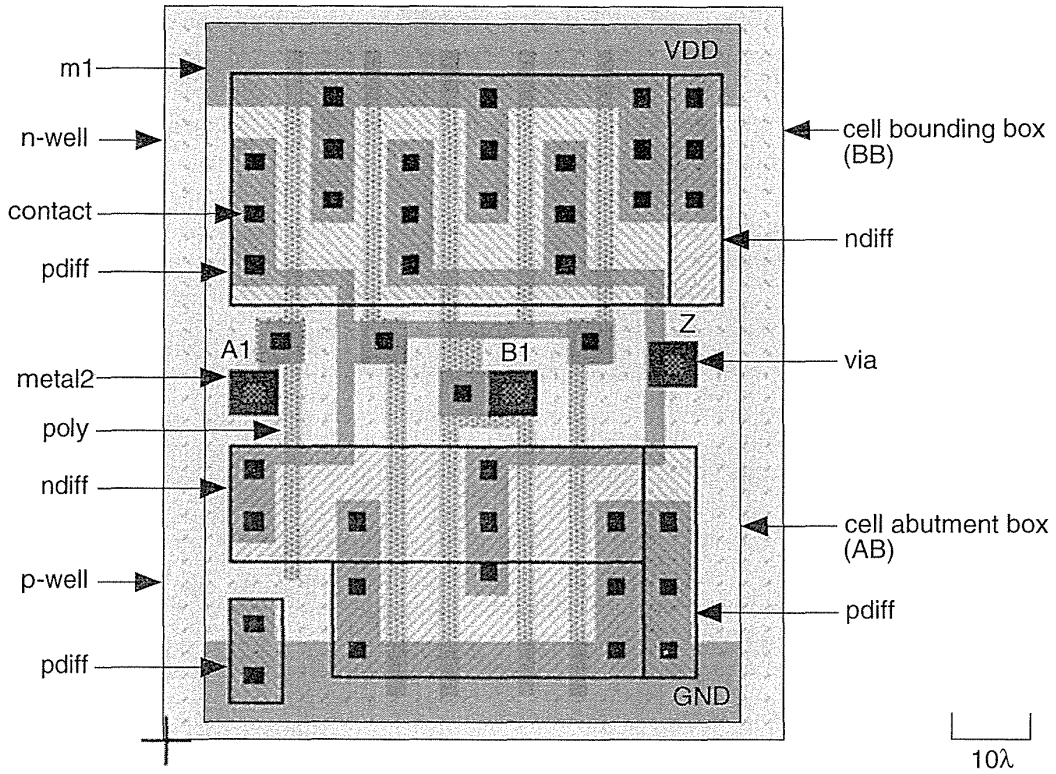


FIGURE 1.3 Looking down on the layout of a standard cell. This cell would be approximately 25 microns wide on an ASIC with λ (lambda) = 0.25 microns (a micron is 10^{-6} m). Standard cells are stacked like bricks in a wall; the abutment box (AB) defines the "edges" of the brick. The difference between the bounding box (BB) and the AB is the area of overlap between the bricks. Power supplies (labeled VDD and GND) run horizontally inside a standard cell on a metal layer that lies above the transistor layers. Each different shaded and labeled pattern represents a different layer. This standard cell has center connectors (the three squares, labeled A1, B1, and Z) that allow the cell to connect to others. The layout was drawn using ROSE, a symbolic layout editor developed by Rockwell and Compass, and then imported into Tanner Research's L-Edit.

design). You may then connect a **flexible block** built from several rows of standard cells to other standard-cell blocks or other full-custom logic blocks. For example, you might want to include a custom interface to a standard, predesigned microcontroller together with some memory. The microcontroller block may be a fixed-size megacell, you might generate the memory using a memory compiler, and the custom logic and memory controller will be built from flexible standard-cell blocks, shaped to fit in the empty spaces on the chip.

Both cell-based and gate-array ASICs use predefined cells, but there is a difference—we can change the transistor sizes in a standard cell to optimize speed and performance, but the device sizes in a gate array are fixed. This results in a trade-off in performance and area in a gate array at the silicon level. The trade-off between area and performance is made at the library level for a standard-cell ASIC.

Modern CMOS ASICs use two, three, or more levels (or layers) of metal for interconnect. This allows wires to cross over different layers in the same way that we use copper traces on different layers on a printed-circuit board. In a two-level metal CMOS technology, connections to the standard-cell inputs and outputs are usually made using the second level of metal (**metal2**, the upper level of metal) at the tops and bottoms of the cells. In a three-level metal technology, connections may be internal to the logic cell (as they are in Figure 1.3). This allows for more sophisticated routing programs to take advantage of the extra metal layer to route interconnect over the top of the logic cells. We shall cover the details of routing ASICs in Chapter 17.

A connection that needs to cross over a row of standard cells uses a feedthrough. The term **feedthrough** can refer either to the piece of metal that is used to pass a signal through a cell or to a space in a cell waiting to be used as a feedthrough—very confusing. Figure 1.4 shows two feedthroughs: one in cell A.14 and one in cell A.23.

In both two-level and three-level metal technology, the power buses (VDD and GND) inside the standard cells normally use the lowest (closest to the transistors) layer of metal (**metal1**). The width of each row of standard cells is adjusted so that they may be aligned using **spacer cells**. The power buses, or rails, are then connected to additional vertical power rails using **row-end cells** at the aligned ends of each standard-cell block. If the rows of standard cells are long, then vertical power rails can also be run in metal2 through the cell rows using special **power cells** that just connect to VDD and GND. Usually the designer manually controls the number and width of the vertical power rails connected to the standard-cell blocks during physical design. A diagram of the power distribution scheme for a CBIC is shown in Figure 1.4.

All the mask layers of a CBIC are customized. This allows megacells (SRAM, a SCSI controller, or an MPEG decoder, for example) to be placed on the same IC with standard cells. Megacells are usually supplied by an ASIC or library company complete with behavioral models and some way to test them (a test strategy). ASIC library companies also supply compilers to generate flexible DRAM, SRAM, and ROM blocks. Since all mask layers on a standard-cell design are customized, memory design is more efficient and denser than for gate arrays.

For logic that operates on multiple signals across a data bus—a **datapath** (**DP**)—the use of standard cells may not be the most efficient ASIC design style. Some ASIC library companies provide a **datapath compiler** that automatically generates **datapath logic**. A **datapath library** typically contains cells such as adders, subtracters, multipliers, and simple **arithmetic and logical units** (**ALUs**). The con-

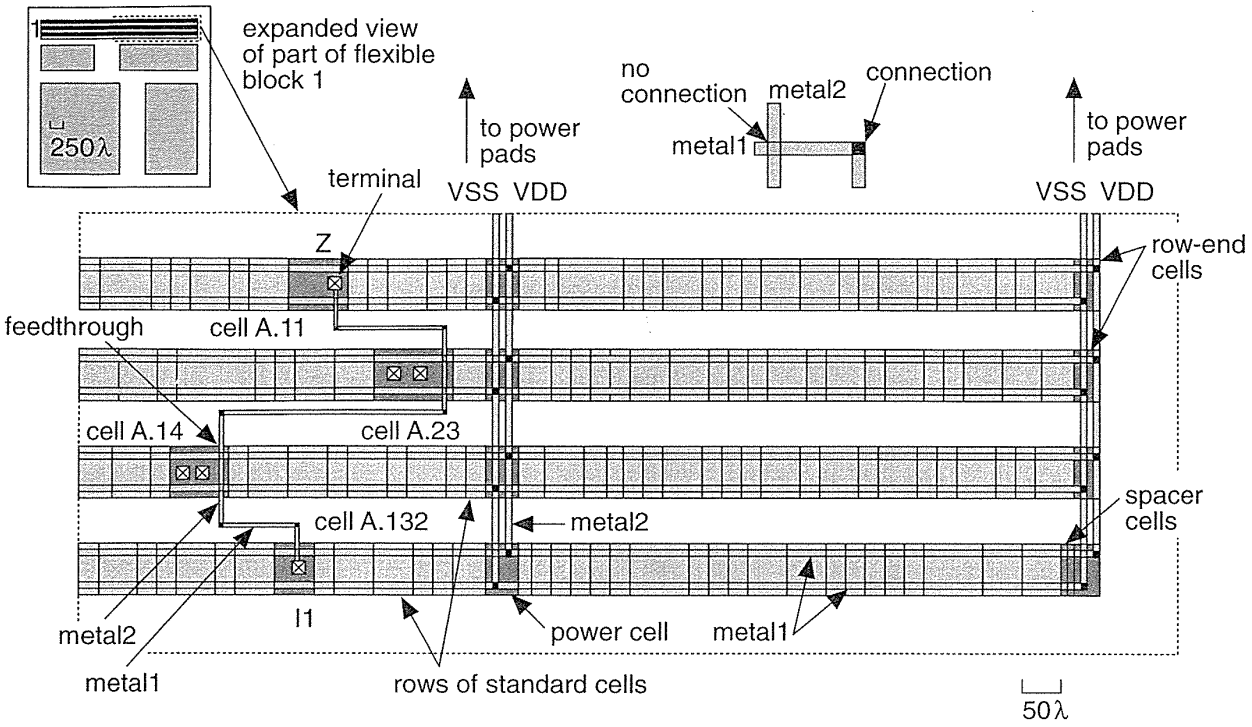


FIGURE 1.4 Routing the CBIC (cell-based IC) shown in Figure 1.2. The use of regularly shaped standard cells, such as the one in Figure 1.3, from a library allows ASICs like this to be designed automatically. This ASIC uses two separate layers of metal interconnect (metal1 and metal2) running at right angles to each other (like traces on a printed-circuit board). Interconnections between logic cells uses spaces (called channels) between the rows of cells. ASICs may have three (or more) layers of metal allowing the cell rows to touch with the interconnect running over the top of the cells.

nectors of datapath library cells are **pitch-matched** to each other so that they fit together. Connecting datapath cells to form a datapath usually, but not always, results in faster and denser layout than using standard cells or a gate array.

Standard-cell and gate-array libraries may contain hundreds of different logic cells, including combinational functions (NAND, NOR, AND, OR gates) with multiple inputs, as well as latches and flip-flops with different combinations of reset, preset and clocking options. The ASIC library company provides designers with a data book in paper or electronic form with all of the functional descriptions and timing information for each library element.

1.1.3 Gate-Array–Based ASICs

In a **gate array** (sometimes abbreviated to GA) or gate-array–based ASIC the transistors are predefined on the silicon wafer. The predefined pattern of transistors on a gate array is the **base array**, and the smallest element that is replicated to make the base array (like an M. C. Escher drawing, or tiles on a floor) is the **base cell** (sometimes called a **primitive cell**). Only the top few layers of metal, which define the interconnect between transistors, are defined by the designer using custom masks. To distinguish this type of gate array from other types of gate array, it is often called a **masked gate array** (**MGA**). The designer chooses from a gate-array library of predesigned and precharacterized logic cells. The logic cells in a gate-array library are often called **macros**. The reason for this is that the base-cell layout is the same for each logic cell, and only the interconnect (inside cells and between cells) is customized, so that there is a similarity between gate-array macros and a software macro. Inside IBM, gate-array macros are known as **books** (so that books are part of a library), but unfortunately this descriptive term is not very widely used outside IBM.

We can complete the diffusion steps that form the transistors and then stockpile wafers (sometimes we call a gate array a **prediffused array** for this reason). Since only the metal interconnections are unique to an MGA, we can use the stockpiled wafers for different customers as needed. Using wafers prefabricated up to the metallization steps reduces the time needed to make an MGA, the **turnaround time**, to a few days or at most a couple of weeks. The costs for all the initial fabrication steps for an MGA are shared for each customer and this reduces the cost of an MGA compared to a full-custom or standard-cell ASIC design.

There are the following different types of MGA or gate-array–based ASICs:

- Channeled gate arrays.
- Channelless gate arrays.
- Structured gate arrays.

The hyphenation of these terms when they are used as adjectives explains their construction. For example, in the term "channeled gate-array architecture," the *gate array* is *channeled*, as will be explained. There are two common ways of arranging (or arraying) the transistors on a MGA: in a channeled gate array we leave space between the rows of transistors for wiring; the routing on a channelless gate array uses rows of unused transistors. The channeled gate array was the first to be developed, but the channelless gate-array architecture is now more widely used. A structured (or embedded) gate array can be either channeled or channelless but it includes (or embeds) a custom block.

1.1.4 Channeled Gate Array

Figure 1.5 shows a **channeled gate array**. The important features of this type of MGA are:

- Only the interconnect is customized.
- The interconnect uses predefined spaces between rows of base cells.
- Manufacturing lead time is between two days and two weeks.

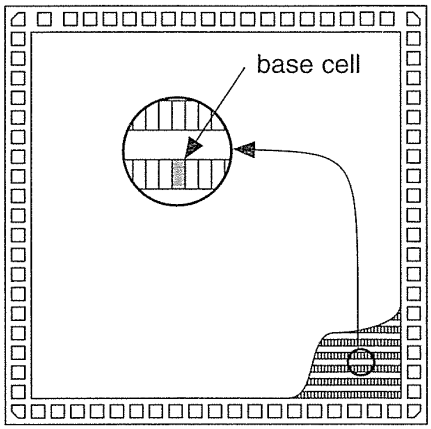


FIGURE 1.5 A channeled gate-array die. The spaces between rows of the base cells are set aside for interconnect.

A channeled gate array is similar to a CBIC—both use rows of cells separated by channels used for interconnect. One difference is that the space for interconnect between rows of cells are fixed in height in a channeled gate array, whereas the space between rows of cells may be adjusted in a CBIC.

1.1.5 Channelless Gate Array

Figure 1.6 shows a **channelless gate array** (also known as a **channel-free gate array**, **sea-of-gates array**, or **SOG** array). The important features of this type of MGA are as follows:

- Only some (the top few) mask layers are customized—the interconnect.
- Manufacturing lead time is between two days and two weeks.

The key difference between a channelless gate array and channeled gate array is that there are no predefined areas set aside for routing between cells on a channelless gate array. Instead we route over the top of the gate-array devices. We can do this because we customize the contact layer that defines the connections between metal1, the first layer of metal, and the transistors. When we use an area of transistors for routing in a channelless array, we do not make any contacts to the devices lying underneath; we simply leave the transistors unused.

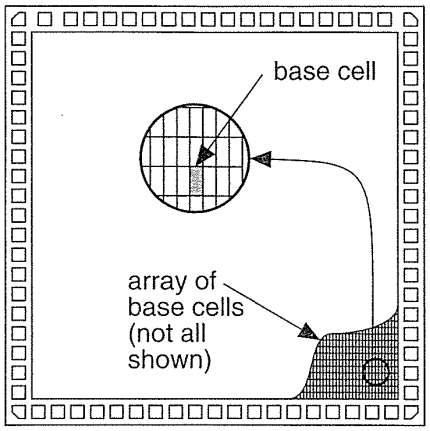


FIGURE 1.6 A channelless gate-array or sea-of-gates (SOG) array die. The core area of the die is completely filled with an array of base cells (the base array).

The logic density—the amount of logic that can be implemented in a given silicon area—is higher for channelless gate arrays than for channeled gate arrays. This is usually attributed to the difference in structure between the two types of array. In fact, the difference occurs because the contact mask is customized in a channelless gate array, but is not usually customized in a channeled gate array. This leads to denser cells in the channelless architectures. Customizing the contact layer in a channelless gate array allows us to increase the density of gate-array cells because we can route over the top of unused contact sites.

1.1.6 Structured Gate Array

An **embedded gate array** or **structured gate array** (also known as **masterslice** or **masterimage**) combines some of the features of CBICs and MGAs. One of the disadvantages of the MGA is the fixed gate-array base cell. This makes the implementation of memory, for example, difficult and inefficient. In an embedded gate array we set aside some of the IC area and dedicate it to a specific function. This embedded area either can contain a different base cell that is more suitable for building memory cells, or it can contain a complete circuit block, such as a microcontroller.

Figure 1.7 shows an embedded gate array. The important features of this type of MGA are the following:

- Only the interconnect is customized.
- Custom blocks (the same for each design) can be embedded.
- Manufacturing lead time is between two days and two weeks.

An embedded gate array gives the improved area efficiency and increased performance of a CBIC but with the lower cost and faster turnaround of an MGA. One disadvantage of an embedded gate array is that the embedded function is fixed. For example, if an embedded gate array contains an area set aside for a 32 k-bit memory,

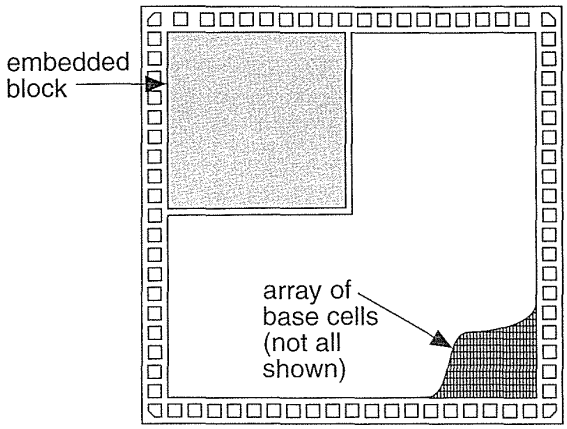


FIGURE 1.7 A structured or embedded gate-array die showing an embedded block in the upper left corner (a static random-access memory, for example). The rest of the die is filled with an array of base cells.

but we only need a 16 k-bit memory, then we may have to waste half of the embedded memory function. However, this may still be more efficient and cheaper than implementing a 32 k-bit memory using macros on a SOG array.

ASIC vendors may offer several embedded gate array structures containing different memory types and sizes as well as a variety of embedded functions. ASIC companies wishing to offer a wide range of embedded functions must ensure that enough customers use each different embedded gate array to give the cost advantages over a custom gate array or CBIC (the Sun Microsystems SPARCstation 1 described in Section 1.3 made use of LSI Logic embedded gate arrays—and the 10K and 100K series of embedded gate arrays were two of LSI Logic's most successful products).

1.1.7 Programmable Logic Devices

Programmable logic devices (**PLDs**) are standard ICs that are available in standard configurations from a catalog of parts and are sold in very high volume to many different customers. However, PLDs may be configured or programmed to create a part customized to a specific application, and so they also belong to the family of ASICs. PLDs use different technologies to allow programming of the device. Figure 1.8 shows a PLD and the following important features that all PLDs have in common:

- No customized mask layers or logic cells
- Fast design turnaround
- A single large block of programmable interconnect
- A matrix of logic macrocells that usually consist of programmable array logic followed by a flip-flop or latch

The simplest type of programmable IC is a **read-only memory** (**ROM**). The most common types of ROM use a metal fuse that can be blown permanently (a **programmable ROM** or **PROM**). An **electrically programmable ROM**, or

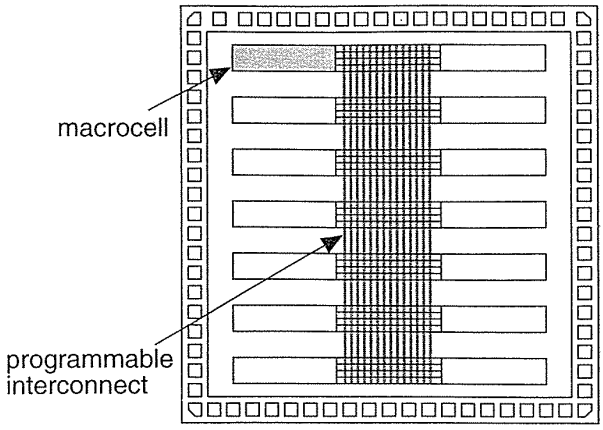


FIGURE 1.8 A programmable logic device (PLD) die. The macrocells typically consist of programmable array logic followed by a flip-flop or latch. The macrocells are connected using a large programmable interconnect block.

EPROM, uses programmable MOS transistors whose characteristics are altered by applying a high voltage. You can erase an EPROM either by using another high voltage (an **electrically erasable PROM**, or **EEPROM**) or by exposing the device to ultraviolet light (**UV-erasable PROM**, or **UVPROM**).

There is another type of ROM that can be placed on any ASIC—a **mask-programmable ROM** (mask-programmed ROM or masked ROM). A masked ROM is a regular array of transistors permanently programmed using custom mask patterns. An embedded masked ROM is thus a large, specialized, logic cell.

The same programmable technologies used to make ROMs can be applied to more flexible logic structures. By using the programmable devices in a large array of AND gates and an array of OR gates, we create a family of flexible and programmable logic devices called **logic arrays**. The company Monolithic Memories (bought by AMD) was the first to produce **Programmable Array Logic** (PAL®, a registered trademark of AMD) devices that you can use, for example, as transition decoders for state machines. A PAL can also include registers (flip-flops) to store the current state information so that you can use a PAL to make a complete state machine.

Just as we have a mask-programmable ROM, we could place a logic array as a cell on a custom ASIC. This type of logic array is called a **programmable logic array** (PLA). There is a difference between a PAL and a PLA: a PLA has a programmable AND logic array, or **AND plane**, followed by a programmable OR logic array, or **OR plane**; a PAL has a programmable AND plane and, in contrast to a PLA, a fixed OR plane.

Depending on how the PLD is programmed, we can have an **erasable PLD** (EPLD), or **mask-programmed PLD** (sometimes called a masked PLD but usually just PLD). The first PALs, PLAs, and PLDs were based on bipolar technology and used programmable fuses or links. CMOS PLDs usually employ floating-gate transistors (see Section 4.3, "EPROM and EEPROM Technology").

1.1.8 Field-Programmable Gate Arrays

A step above the PLD in complexity is the **field-programmable gate array** (**FPGA**). There is very little difference between an FPGA and a PLD—an FPGA is usually just larger and more complex than a PLD. In fact, some companies that manufacture programmable ASICs call their products FPGAs and some call them **complex PLDs**. FPGAs are the newest member of the ASIC family and are rapidly growing in importance, replacing TTL in microelectronic systems. Even though an FPGA is a type of gate array, we do not consider the term gate-array–based ASICs to include FPGAs. This may change as FPGAs and MGAs start to look more alike.

Figure 1.9 illustrates the essential characteristics of an FPGA:

- None of the mask layers are customized.
- A method for programming the basic logic cells and the interconnect.
- The core is a regular array of programmable basic logic cells that can implement combinational as well as sequential logic (flip-flops).
- A matrix of programmable interconnect surrounds the basic logic cells.
- Programmable I/O cells surround the core.
- Design turnaround is a few hours.

We shall examine these features in detail in Chapters 4–8.

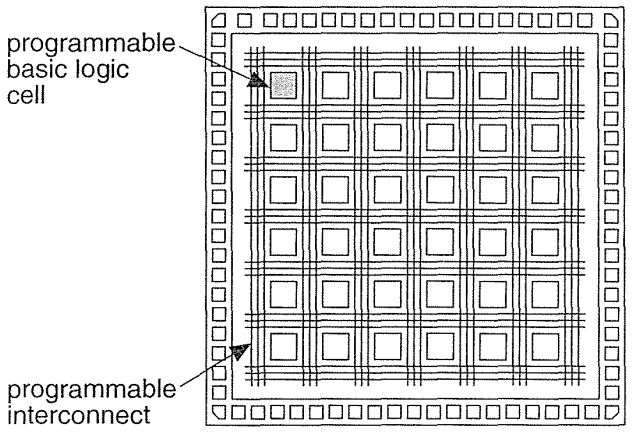


FIGURE 1.9 A field-programmable gate array (FPGA) die. All FPGAs contain a regular structure of programmable basic logic cells surrounded by programmable interconnect. The exact type, size, and number of the programmable basic logic cells varies tremendously.

1.2 Design Flow

Figure 1.10 shows the sequence of steps to design an ASIC; we call this a **design flow**. The steps are listed below (numbered to correspond to the labels in Figure 1.10) with a brief description of the function of each step.

1. *Design entry.* Enter the design into an ASIC design system, either using a **hardware description language** (**HDL**) or *schematic entry.*

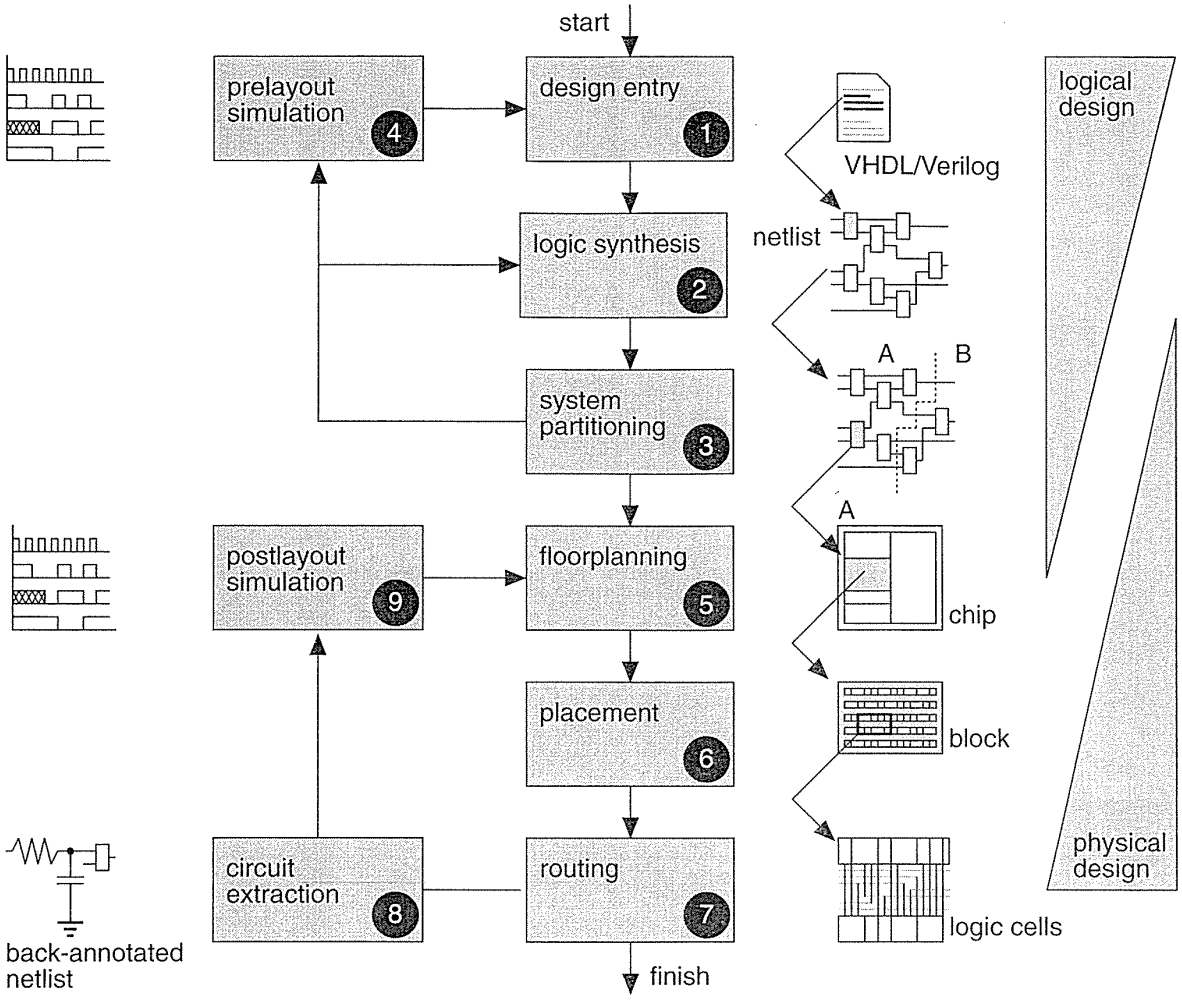


FIGURE 1.10 ASIC design flow.

2. *Logic synthesis.* Use an HDL (VHDL or Verilog) and a logic synthesis tool to produce a **netlist**—a description of the logic cells and their connections.
3. *System partitioning.* Divide a large system into ASIC-sized pieces.
4. *Prelayout simulation.* Check to see if the design functions correctly.
5. *Floorplanning.* Arrange the blocks of the netlist on the chip.
6. *Placement.* Decide the locations of cells in a block.
7. *Routing.* Make the connections between cells and blocks.
8. *Extraction.* Determine the resistance and capacitance of the interconnect.

9. *Postlayout simulation.* Check to see the design still works with the added loads of the interconnect.

Steps 1–4 are part of **logical design**, and steps 5–9 are part of **physical design**. There is some overlap. For example, system partitioning might be considered as either logical or physical design. To put it another way, when we are performing system partitioning we have to consider both logical and physical factors. Chapters 9–14 of this book is largely about logical design and Chapters 15–17 largely about physical design.

1.3 Case Study

Sun Microsystems released the SPARCstation 1 in April 1989. It is now an old design but a very important example because it was one of the first workstations to make extensive use of ASICs to achieve the following:

- Better performance at lower cost
- Compact size, reduced power, and quiet operation
- Reduced number of parts, easier assembly, and improved reliability

The SPARCstation 1 contains about 50 ICs on the system motherboard—excluding the DRAM used for the system memory (standard parts). The SPARCstation 1 designers partitioned the system into the nine ASICs shown in Table 1.1 and wrote specifications for each ASIC—this took about three months.[1] LSI Logic and Fujitsu designed the SPARC **integer unit** (IU) and **floating-point unit** (**FPU**) to these specifications. The clock ASIC is a fairly straightforward design and, of the six remaining ASICs, the video controller/data buffer, the RAM controller, and the **direct memory access** (**DMA**) controller are defined by the 32-bit **system bus** (**SBus**) and the other ASICs that they connect to. The rest of the system is partitioned into three more ASICs: the **cache controller**, **memory-management unit** (MMU), and the data buffer. These three ASICs, with the IU and FPU, have the most critical timing paths and determine the system partitioning. The design of ASICs 3–8 in Table 1.1 took five Sun engineers six months after the specifications were complete. During the design process, the Sun engineers simulated the entire SPARCstation 1—including execution of the Sun operating system (SunOS).

[1]Some information in Section 1.3 and Section 15.3 is from the SPARCstation 10 Architecture Guide—May 1992, p. 2 and pp. 27–28 and from two publicity brochures (known as "sparkle sheets"). The first is "Concept to System: How Sun Microsystems Created SPARCstation 1 Using LSI Logic's ASIC System Technology," A. Bechtolsheim, T. Westberg, M. Insley, and J. Ludemann of Sun Microsystems; J-H. Huang and D. Boyle of LSI Logic. This is an LSI Logic publication. The second paper is "SPARCstation 1: Beyond the 3M Horizon," A. Bechtolsheim and E. Frank, a Sun Microsystems publication. I did not include these as references since they are impossible to obtain now, but I would like to give credit to Andy Bechtolsheim and the Sun Microsystems and LSI Logic engineers.

TABLE 1.1 The ASICs in the Sun Microsystems SPARCstation 1.

	SPARCstation 1 ASIC	Gates (k-gates)
1	SPARC integer unit (IU)	20
2	SPARC floating-point unit (FPU)	50
3	Cache controller	9
4	Memory-management unit (MMU)	5
5	Data buffer	3
6	Direct memory access (DMA) controller	9
7	Video controller/data buffer	4
8	RAM controller	1
9	Clock generator	1

Table 1.2 shows the software tools used to design the SPARCstation 1, many of which are now obsolete. The important point to notice, though, is that there is a lot more to microelectronic system design than designing the ASICs—less than one-third of the tools listed in Table 1.2 were ASIC design tools.

TABLE 1.2 The CAD tools used in the design of the Sun Microsystems SPARCstation 1.

Design level	Function	Tool[1]
ASIC design	ASIC physical design	LSI Logic
	ASIC logic synthesis	Internal tools and UC Berkeley tools
	ASIC simulation	LSI Logic
Board design	Schematic capture	Valid Logic
	PCB layout	Valid Logic Allegro
	Timing verification	Quad Design Motive and internal tools
Mechanical design	Case and enclosure	Autocad
	Thermal analysis	Pacific Numerix
	Structural analysis	Cosmos
Management	Scheduling	Suntrac
	Documentation	Interleaf and FrameMaker

[1]Names are trademarks of their respective companies.

The SPARCstation 1 cost about $9000 in 1989 or, since it has an execution rate of approximately 12 million instructions per second (MIPS), $750/MIPS. Using ASIC technology reduces the motherboard to about the size of a piece of paper—8.5 inches by 11 inches—with a power consumption of about 12 W. The SPARCstation 1 "pizza box" is 16 inches across and 3 inches high—smaller than a typical IBM-compatible personal computer in 1989. This speed, power, and size performance is (there are still SPARCstation 1s in use) made possible by using ASICs. We shall return to the SPARCstation 1, to look more closely at the partitioning step, in Section 15.3, "System Partitioning."

1.4 Economics of ASICs

In this section we shall discuss the economics of using ASICs in a product and compare the most popular types of ASICs: an FPGA, an MGA, and a CBIC. To make an economic comparison between these alternatives, we consider the ASIC itself as a product and examine the components of product cost: fixed costs and variable costs. Making cost comparisons is dangerous—costs change rapidly and the semiconductor industry is notorious for keeping its costs, prices, and pricing strategy closely guarded secrets. The figures in the following sections are approximate and used to illustrate the different components of cost.

1.4.1 Comparison Between ASIC Technologies

The most obvious economic factor in making a choice between the different ASIC types is the **part cost**. Part costs vary enormously—you can pay anywhere from a few dollars to several hundreds of dollars for an ASIC. In general, however, FPGAs are more expensive per gate than MGAs, which are, in turn, more expensive than CBICs. For example, a 0.5 μm, 20 k-gate array might cost 0.01–0.02 cents/gate (for more than 10,000 parts) or $2–$4 per part, but an equivalent FPGA might be $20. The price per gate for an FPGA to implement the same function is typically 2–5 times the cost of an MGA or CBIC.

Given that an FPGA is more expensive than an MGA, which is more expensive than a CBIC, when and why does it make sense to choose a more expensive part? Is the increased flexibility of an FPGA worth the extra cost per part? Given that an MGA or CBIC is specially tailored for each customer, there are extra hidden costs associated with this step that we should consider. To make a true comparison between the different ASIC technologies, we shall quantify some of these costs.

1.4.2 Product Cost

The total cost of any product can be separated into **fixed costs** and **variable costs**:

$$\text{total product cost} = \text{fixed product cost} + \text{variable product cost} \times \text{products sold}. \quad (1.1)$$

Fixed costs are independent of **sales volume**—the number of products sold. However, the fixed costs amortized per product sold (fixed costs divided by products sold) decrease as sales volume increases. Variable costs include the cost of the parts used in the product, assembly costs, and other manufacturing costs.

Let us look more closely at the parts in a product. If we want to buy ASICs to assemble our product, the total part cost is

$$\text{total part cost} = \text{fixed part cost} + \text{variable cost per part} \times \text{volume of parts}. \quad (1.2)$$

Our fixed cost when we use an FPGA is low—we just have to buy the software and any programming equipment. The fixed part costs for an MGA or CBIC are higher and include the costs of the masks, simulation, and test program development. We shall discuss these extra costs in more detail in Sections 1.4.3 and 1.4.4. Figure 1.11 shows a **break-even graph** that compares the total part cost for an FPGA, MGA, and a CBIC with the following assumptions:

- FPGA fixed cost is $21,800, part cost is $39.
- MGA fixed cost is $86,000, part cost is $10.
- CBIC fixed cost is $146,000, part cost is $8.

At low volumes, the MGA and the CBIC are more expensive because of their higher fixed costs. The total part costs of two alternative types of ASIC are equal at the **break-even volume**. In Figure 1.11 the break-even volume for the FPGA and the MGA is about 2000 parts. The break-even volume between the FPGA and the CBIC is about 4000 parts. The break-even volume between the MGA and the CBIC is higher—at about 20,000 parts.

We shall describe how to calculate the fixed part costs next. Following that we shall discuss how we came up with cost per part of $39, $10, and $8 for the FPGA, MGA, and CBIC.

1.4.3 ASIC Fixed Costs

Figure 1.12 shows a spreadsheet, "Fixed Costs," that calculates the fixed part costs associated with ASIC design.

The **training cost** includes the cost of the time to learn any new **electronic design automation** (**EDA**) system. For example, a new FPGA design system might require a few days to learn; a new gate-array or cell-based design system might require taking a course. Figure 1.12 assumes that the cost of an engineer (including overhead, benefits, infrastructure, and so on) is between $100,000 and $200,000 per year or $2000 to $4000 per week (in the United States in 1990s dollars).

Next we consider the **hardware and software cost** for ASIC design. Figure 1.12 shows some typical figures, but you can spend anywhere from $1000 to $1 million (and more) on ASIC design software and the necessary infrastructure.

We try to measure **productivity** of an ASIC designer in gates (or transistors) per day. This is like trying to predict how long it takes to dig a hole, and the number of gates per day an engineer averages varies wildly. ASIC design productivity must

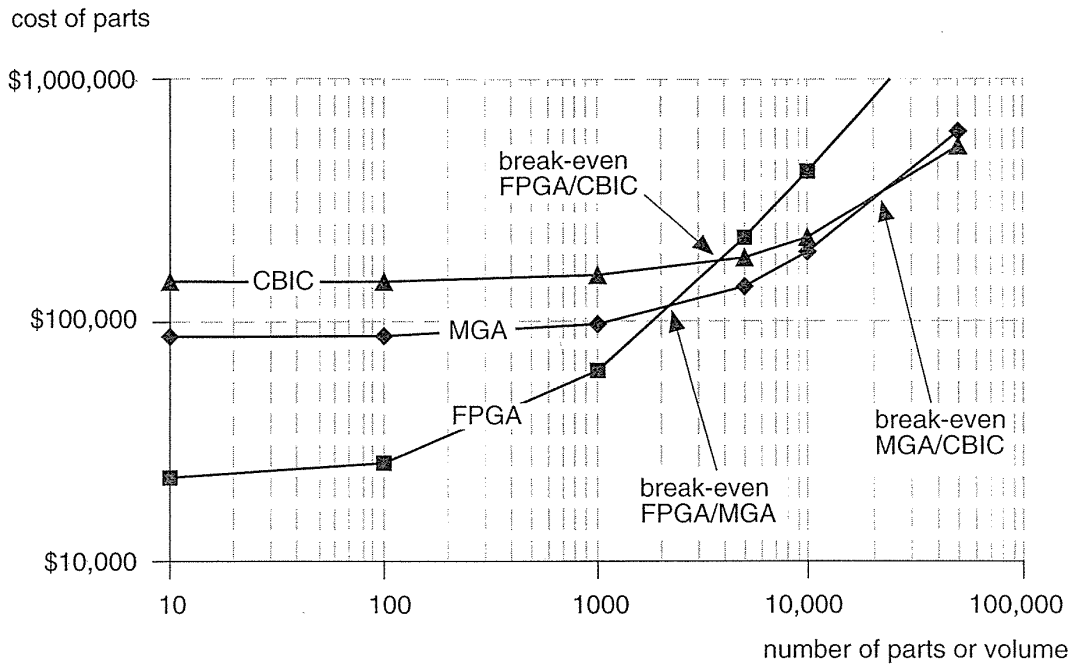


FIGURE 1.11 A break-even analysis for an FPGA, a masked gate array (MGA) and a custom cell-based ASIC (CBIC). The break-even volume between two technologies is the point at which the total cost of parts are equal. These numbers are very approximate.

increase as ASIC sizes increase and will depend on experience, design tools, and the ASIC complexity. If we are using similar design methods, design productivity ought to be independent of the type of ASIC, but FPGA design software is usually available as a complete bundle on a PC. This means that it is often easier to learn and use than semicustom ASIC design tools.

Every ASIC has to pass a **production test** to make sure that it works. With modern test tools the generation of any test circuits on each ASIC that are needed for production testing can be automatic, but it still involves a cost for **design for test**. An FPGA is tested by the manufacturer before it is sold to you and before you program it. You are still paying for testing an FPGA, but it is a hidden cost folded into the part cost of the FPGA. You do have to pay for any **programming costs** for an FPGA, but we can include these in the hardware and software cost.

The **nonrecurring-engineering** (**NRE**) charge includes the cost of work done by the ASIC vendor and the cost of the masks. The production test uses sets of test inputs called **test vectors**, often many thousands of them. Most ASIC vendors require simulation to generate test vectors and test programs for production testing, and will charge for a **test-program development cost**. The number of masks required by an ASIC during fabrication can range from three or four (for a gate array) to 15 or more (for a CBIC). Total mask costs can range from $5000 to

	FPGA		MGA		CBIC	
Training:	$800		$2,000		$2,000	
Days		2		5		5
Cost/day		$400		$400		$400
Hardware	$10,000		$10,000		$10,000	
Software	$1,000		$20,000		$40,000	
Design:	$8,000		$20,000		$20,000	
Size (gates)		10,000		10,000		10,000
Gates/day		500		200		200
Days		20		50		50
Cost/day		$400		$400		$400
Design for test:			$2,000		$2,000	
Days				5		5
Cost/day				$400		$400
NRE:			$30,000		$70,000	
Masks				$10,000		$50,000
Simulation				$10,000		$10,000
Test program				$10,000		$10,000
Second source:	$2,000		$2,000		$2,000	
Days		5		5		5
Cost/day		$400		$400		$400
Total fixed costs	$21,800		$86,000		$146,000	

FIGURE 1.12 A spreadsheet, "Fixed Costs," for a field-programmable gate array (FPGA), a masked gate array (MGA), and a cell-based ASIC (CBIC). These costs can vary wildly.

$50,000 or more. The total NRE charge can range from $10,000 to $300,000 or more and will vary with volume and the size of the ASIC. If you commit to high volumes (above 100,000 parts), the vendor may waive the NRE charge. The NRE charge may also include the costs of software tools, design verification, and prototype samples.

If your design does not work the first time, you have to complete a further design **pass** (**turn** or **spin**) that requires additional NRE charges. Normally you sign a contract (sign off a design) with an ASIC vendor that guarantees first-pass success—this means that if you designed your ASIC according to rules specified by the vendor, then the vendor guarantees that the silicon will perform according to the simulation or you get your money back. This is why the difference between semicustom and full-custom design styles is so important—the ASIC vendor will not (and cannot) guarantee your design will work if you use any full-custom design techniques.

Nowadays it is almost routine to have an ASIC work on the first pass. However, if your design does fail, it is little consolation to have a second pass for free if your company goes bankrupt in the meantime. Figure 1.13 shows a **profit model** that represents the **profit flow** during the **product lifetime**. Using this model, we can estimate the lost profit due to any delay.

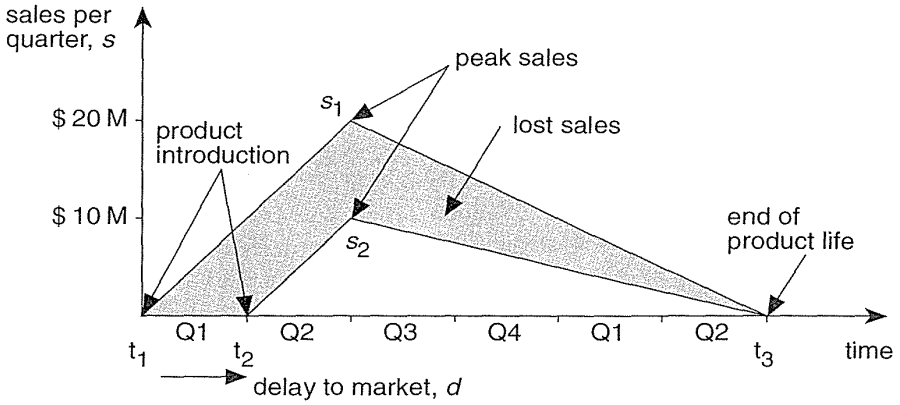


FIGURE 1.13 A profit model. If a product is introduced on time, the total sales are $60 million (the area of the higher triangle). With a three-month (one fiscal quarter) delay the sales decline to $25 million. The difference is shown as the shaded area between the two triangles and amounts to a lost revenue of $35 million.

Suppose we have the following situation:

- The product lifetime is 18 months (6 fiscal quarters).
- The product sales increase (linearly) at $10 million per quarter independently of when the product is introduced (we suppose this is because we can increase production and sales only at a fixed rate).
- The product reaches its peak sales at a point in time that is independent of when we introduce a product (because of external market factors that we cannot control).
- The product declines in sales (linearly) to the end of its life—a point in time that is also independent of when we introduce the product (again due to external market forces).

The simple profit and revenue model of Figure 1.13 shows us that we would lose $35 million in sales in this situation due to a 3-month delay. Despite the obvious problems with such a simple model (how can we introduce the same product twice to compare the performance?), it is widely used in marketing. In the electronics industry product lifetimes continue to shrink. In the PC industry it is not unusual to have a product lifetime of 18 months or less. This means that it is critical to achieve a rapid design time (or high **product velocity**) with no delays.

The last fixed cost shown in Figure 1.12 corresponds to an "insurance policy." When a company buys an ASIC part, it needs to be assured that it will always have a back-up source, or **second source**, in case something happens to its first or primary source. Established FPGA companies have a second source that produces equivalent parts. With a custom ASIC you may have to do some redesign to transfer your ASIC to the second source. However, for all ASIC types, switching production to a second source will involve some cost. Figure 1.12 assumes a second-source cost of $2000 for all types of ASIC (the amount may be substantially more than this).

1.4.4 ASIC Variable Costs

Figure 1.14 shows a spreadsheet, "Variable Costs," that calculates some example part costs. This spreadsheet uses the terms and parameters defined below the figure.

	FPGA	MGA	CBIC	Units
Wafer size	6	6	6	inches
Wafer cost	1,400	1,300	1,500	$
Design	10,000	10,000	10,000	gates
Density	10,000	20,000	25,000	gates/sq.cm
Utilization	60	85	100	%
Die size	1.67	0.59	0.40	sq.cm
Die/wafer	88	248	365	
Defect density	1.10	0.90	1.00	defects/sq.cm
Yield	65	72	80	%
Die cost	25	7	5	$
Profit margin	60	45	50	%
Price/gate	0.39	0.10	0.08	cents
Part cost	$39	$10	$8	

FIGURE 1.14 A spreadsheet, "Variable Costs," to calculate the part cost (that is the variable cost for a product using ASICs) for different ASIC technologies.

- The **wafer size** increases every few years. From 1985 to 1990, 4-inch to 6-inch diameter wafers were common; equipment using 6-inch to 8-inch wafers was introduced between 1990 and 1995; the next step is the 300 cm or 12-inch wafer. The 12-inch wafer will probably take us to 2005.
- The **wafer cost** depends on the equipment costs, process costs, and overhead in the fabrication line. A typical wafer cost is between $1000 and $5000, with $2000 being average; the cost declines slightly during the life of a process and increases only slightly from one process generation to the next.

- **Moore's Law** (after Gordon Moore of Intel) models the observation that the number of transistors on a chip roughly doubles every 18 months. Not all designs follow this law, but a "large" ASIC design seems to grow by a factor of 10 every 5 years (close to Moore's Law). In 1990 a large ASIC design size was 10 k-gate, in 1995 a large design was about 100 k-gate, in 2000 it will be 1 M-gate, in 2005 it will be 10 M-gate.
- The **gate density** is the number of gate equivalents per unit area (remember: a gate equivalent, or gate, corresponds to a two-input NAND gate).
- The **gate utilization** is the percentage of gates that are on a die that we can use (on a gate array we waste some gate space for interconnect).
- The **die size** is determined by the design size (in gates), the gate density, and the utilization of the die.
- The number of **die per wafer** depends on the die size and the wafer size (we have to pack rectangular or square die, together with some test chips, on to a circular wafer so some space is wasted).
- The **defect density** is a measure of the quality of the fabrication process. The smaller the defect density the less likely there is to be a flaw on any one die. A single defect on a die is almost always fatal for that die. Defect density usually increases with the number of steps in a process. A defect density of less than 1 cm^{-2} is typical and required for a submicron CMOS process.
- The **yield** of a process is the key to a profitable ASIC company. The yield is the fraction of die on a wafer that are good (expressed as a percentage). Yield depends on the complexity and maturity of a process. A process may start out with a yield of close to zero for complex chips, which then climbs to above 50 percent within the first few months of production. Within a year the yield has to be brought to around 80 percent for the average complexity ASIC for the process to be profitable. Yields of 90 percent or more are not uncommon.
- The **die cost** is determined by wafer cost, number of die per wafer, and the yield. Of these parameters, the most variable and the most critical to control is the yield.
- The **profit margin** (what you sell a product for, less what it costs you to make it, divided by the cost) is determined by the ASIC company's fixed and variable costs. ASIC vendors that make and sell custom ASICs have huge fixed and variable costs associated with building and running fabrication facilities (a fabrication plant is a **fab**). FPGA companies are typically **fabless**—they do not own a fab—they must pass on the costs of the chip manufacture (plus the profit margin of the chip manufacturer) and the development cost of the FPGA structure in the FPGA part cost. The profitability of any company in the ASIC business varies greatly.

- The **price per gate** (usually measured in cents per gate) is determined by die costs and design size. It varies with design size and declines over time.
- The **part cost** is determined by all of the preceding factors. As such it will vary widely with time, process, yield, economic climate, ASIC size and complexity, and many other factors.

As an estimate you can assume that the price per gate for any process technology falls at about 20% per year during its life (the average life of a CMOS process is 2–4 years, and can vary widely). Beyond the life of a process, prices can increase as demand falls and the fabrication equipment becomes harder to maintain. Figure 1.15 shows the price per gate for the different ASICs and process technologies using the following assumptions:

- For any new process technology the price per gate decreases by 40% in the first year, 30% in the second year, and then remains constant.
- A new process technology is introduced approximately every 2 years, with feature size decreasing by a factor of two every 5 years as follows: 2 μm in 1985, 1.5 μm in 1987, 1 μm in 1989, 0.8–0.6 μm in 1991–1993, 0.5–0.35 μm in 1996–1997, 0.25–0.18 μm in 1998–2000.
- CBICs and MGAs are introduced at approximately the same time and price.
- The price of a new process technology is initially 10% above the process that it replaces.
- FPGAs are introduced one year after CBICs that use the same process technology.
- The initial FPGA price (per gate) is 10 percent higher than the initial price for CBICs or MGAs using the same process technology.

From Figure 1.15 you can see that the successive introduction of new process technologies every 2 years drives the price per gate down at a rate close to 30 percent per year. The cost figures that we have used in this section are very approximate and can vary widely (this means they may be off by a factor of 2 but probably are correct within a factor of 10). ASIC companies do use spreadsheet models like these to calculate their costs.

Having decided if, and then which, ASIC technology is appropriate, you need to choose the appropriate cell library. Next we shall discuss the issues surrounding ASIC cell libraries: the different types, their sources, and their contents.

1.5 ASIC Cell Libraries

The cell library is the key part of ASIC design. For a programmable ASIC the FPGA company supplies you with a library of logic cells in the form of a **design kit**, you normally do not have a choice, and the cost is usually a few thousand dollars. For MGAs and CBICs you have three choices: the **ASIC vendor** (the company that will

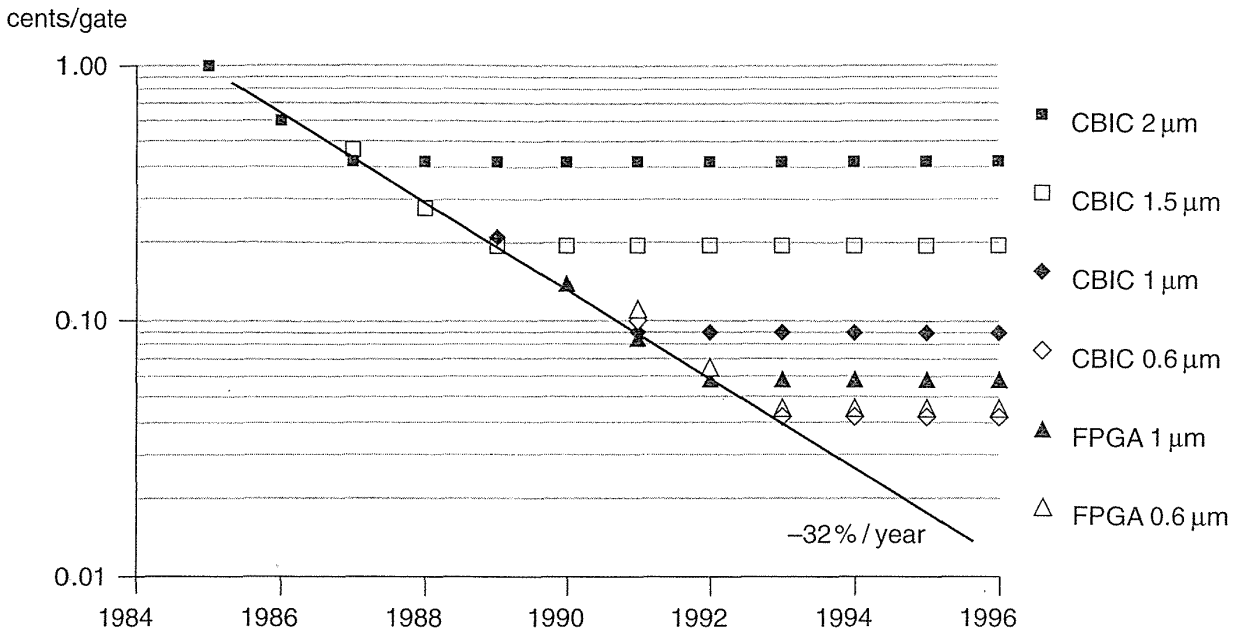


FIGURE 1.15 Example price per gate figures.

build your ASIC) will supply a cell library, or you can buy a cell library from a third-party **library vendor**, or you can build your own cell library.

The first choice, using an **ASIC-vendor library**, requires you to use a set of design tools approved by the ASIC vendor to enter and simulate your design. You have to buy the tools, and the cost of the cell library is folded into the NRE. Some ASIC vendors (especially for MGAs) supply tools that they have developed in-house. For some reason the more common model in Japan is to use tools supplied by the ASIC vendor, but in the United States, Europe, and elsewhere designers want to choose their own tools. Perhaps this has to do with the relationship between customer and supplier being a lot closer in Japan than it is elsewhere.

An ASIC vendor library is normally a **phantom library**—the cells are empty boxes, or **phantoms**, but contain enough information for layout (for example, you would only see the bounding box or abutment box in a phantom version of the cell in Figure 1.3). After you complete layout you **hand off** a netlist to the ASIC vendor, who fills in the empty boxes (**phantom instantiation**) before manufacturing your chip.

The second and third choices require you to make a **buy-or-build decision**. If you complete an ASIC design using a cell library that you bought, you also own the masks (the **tooling**) that are used to manufacture your ASIC. This is called **customer-owned tooling** (**COT**, pronounced "see-oh-tee"). A library vendor normally develops a cell library using information about a process supplied by an **ASIC**

foundry. An ASIC foundry (in contrast to an ASIC vendor) only provides manufacturing, with no design help. If the cell library meets the foundry specifications, we call this a **qualified cell library**. These cell libraries are normally expensive (possibly several hundred thousand dollars), but if a library is qualified at several foundries this allows you to shop around for the most attractive terms. This means that buying an expensive library can be cheaper in the long run than the other solutions for high-volume production.

The third choice is to develop a cell library in-house. Many large computer and electronics companies make this choice. Most of the cell libraries designed today are still developed in-house despite the fact that the process of **library development** is complex and very expensive.

However created, each cell in an ASIC cell library must contain the following:

- A physical layout
- A behavioral model
- A Verilog/VHDL model
- A detailed timing model
- A test strategy
- A circuit schematic
- A cell icon
- A wire-load model
- A routing model

For MGA and CBIC cell libraries we need to complete cell design and **cell layout** and shall discuss this in Chapter 2. The ASIC designer may not actually see the layout if it is hidden inside a phantom, but the layout will be needed eventually. In a programmable ASIC the cell layout is part of the programmable ASIC design (see Chapter 4).

The ASIC designer needs a high-level, **behavioral model** for each cell because simulation at the detailed timing level takes too long for a complete ASIC design. For a NAND gate a behavioral model is simple. A multiport RAM model can be very complex. We shall discuss behavioral models when we describe Verilog and VHDL in Chapter 10 and Chapter 11. The designer may require Verilog and VHDL models in addition to the models for a particular logic simulator.

ASIC designers also need a detailed **timing model** for each cell to determine the performance of the critical pieces of an ASIC. It is too difficult, too time-consuming, and too expensive to build every cell in silicon and measure the cell delays. Instead library engineers simulate the delay of each cell, a process known as **characterization**. Characterizing a standard-cell or gate-array library involves **circuit extraction** from the full-custom cell layout for each cell. The extracted schematic includes all the parasitic resistance and capacitance elements. Then library engineers perform a simulation of each cell including the parasitic elements to determine the switching delays. The simulation models for the transistors are derived from measurements on special

chips included on a wafer called **process control monitors** (**PCMs**) or **drop-ins**. Library engineers then use the results of the circuit simulation to generate detailed timing models for logic simulation. We shall cover timing models in Chapter 13.

All ASICs need to be production tested (programmable ASICs may be tested by the manufacturer before they are customized, but they still need to be tested). Simple cells in small or medium-size blocks can be tested using automated techniques, but large blocks such as RAM or multipliers need a planned strategy. We shall discuss test in Chapter 14.

The **cell schematic** (a netlist description) describes each cell so that the cell designer can perform simulation for complex cells. You may not need the detailed cell schematic for all cells, but you need enough information to compare what you think is on the silicon (the schematic) with what is actually on the silicon (the layout)—this is a **layout versus schematic** (**LVS**) check.

If the ASIC designer uses schematic entry, each cell needs a **cell icon** together with connector and naming information that can be used by design tools from different vendors. We shall cover ASIC design using schematic entry in Chapter 9. One of the advantages of using **logic synthesis** (Chapter 12) rather than schematic design entry is eliminating the problems with icons, connectors, and cell names. Logic synthesis also makes moving an ASIC between different cell libraries, or **retargeting**, much easier.

In order to estimate the parasitic capacitance of wires before we actually complete any routing, we need a statistical estimate of the capacitance for a net in a given size circuit block. This usually takes the form of a look-up table known as a **wire-load model**. We also need a **routing model** for each cell. Large cells are too complex for the physical design or layout tools to handle directly and we need a simpler representation—a **phantom**—of the physical layout that still contains all the necessary information. The phantom may include information that tells the automated routing tool where it can and cannot place wires over the cell, as well as the location and types of the connections to the cell.

1.6 Summary

In this chapter we have looked at the difference between full-custom ASICs, semi-custom ASICs, and programmable ASICs. Table 1.3 summarizes their different features. ASICs use a library of predesigned and precharacterized logic cells. In fact, we could define an ASIC as a design style that uses a cell library rather than in terms of what an ASIC is or what an ASIC does.

You can think of ICs like pizza. A full-custom pizza is built from scratch. You can customize all the layers of a CBIC pizza, but from a predefined selection, and it takes a while to cook. An MGA pizza uses precooked crusts with fixed sizes and you choose only from a few different standard types on a menu. This makes MGA pizza

TABLE 1.3 Types of ASIC.

ASIC type	Family member	Custom mask layers	Custom logic cells
Full-custom	Analog/digital	All	Some
Semicustom	Cell-based (CBIC)	All	None
	Masked gate array (MGA)	Some	None
Programmable	Field-programmable gate array (FPGA)	None	None
	Programmable logic device (PLD)	None	None

a little faster to cook and a little cheaper. An FPGA is rather like a frozen pizza—you buy it at the supermarket in a limited selection of sizes and types, but you can put it in the microwave at home and it will be ready in a few minutes.

In each chapter we shall indicate the key concepts. In this chapter they are

- The difference between full-custom and semicustom ASICs
- The difference between standard-cell, gate-array, and programmable ASICs
- The ASIC design flow
- Design economics including part cost, NRE, and breakeven volume
- The contents and use of an ASIC cell library

Next, in Chapter 2, we shall take a closer look at the semicustom ASICs that were introduced in this chapter.

1.7 Problems

1.1 (Break-even volumes, 60 min.) You need a spreadsheet program (such as Microsoft Excel) for this problem.

a. Build a spreadsheet, "Break-even Analysis," to generate Figure 1.11.

b. Derive equations for the break-even volumes (there are three: FPGA/MGA, FPGA/CBIC, and MGA/CBIC) and calculate their values.

c. Increase the FPGA part cost by $10 and use your spreadsheet to produce the new break-even graph. *Hint:* (For users of Excel-like spreadsheets) use the XY scatter plot option. Use the first column for the x-axis data.

d. Find the new break-even volumes (change the volume until the cost becomes the same for two technologies).

e. Program your spreadsheet to automatically find the break-even volumes. Now graph the break-even volume (for a choice between FPGA and CBIC) for

values of FPGA part costs ranging from \$10–\$50 and CBIC costs ranging from \$2–\$10 (do not change the fixed costs from Figure 1.12).

f. Calculate the sensitivity of the break-even volumes to changes in the part costs and fixed costs. There are three break-even volumes and each of these is sensitive to two part costs and two fixed costs. Express your answers in two ways: in equation form and as numbers (for the values in Section 1.4.2 and Figure 1.11).

g. The costs in Figure 1.11 are not unrealistic. What can you say from your answers if you are a defense contractor, primarily selling products in volumes of less than 1000 parts? What if you are a PC board vendor selling between 10,000 and 100,000 parts?

1.2 (Design productivity, 10 min.) Given the figures for the SPARCstation 1 ASICs described in Section 1.3 what was the productivity measured in transistors/day? and measured in gates/day? Compare your answers with the figures for productivity in Section 1.4.3 and explain any differences. How accurate do you think productivity estimates are?

1.3 (ASIC package size, 30 min.) Assuming, for this problem, a gate density of 1.0 gate/mil^2 (see Section 15.4, "Estimating ASIC Size," for a detailed explanation of this figure), the maximum number of gates you can put in a package is determined by the maximum die size for each of the packages shown in Table 1.4. The maximum die size is determined by the package cavity size; these are **package-limited** ASICs. Calculate the maximum number of I/O pads that can be placed on a die for each package if the pad spacing is: (i) 5 mil, and (ii) 10 mil. Compare your answers with the maximum numbers of pins (or leads) on each package and comment. Now calculate the minimum number of gates that you can put in each package determined by the minimum die size.

1.4 (ASIC vendor costs, 30 min.) There is a well-known saying in the ASIC business: "We lose money on every part—but we make it up in volume." This has a serious side. Suppose Sumo Silicon currently has two customers: Mr. Big, who currently buys 10,000 parts per week, and Ms. Smart, who currently buys 4800 parts per week. A new customer, Ms. Teeny (who is growing fast), wants to buy 1200 parts per week. Sumo's costs are

$$\text{wafer cost} = \$500 + (\$250{,}000/W),$$

where W is the number of wafer starts per week. Assume each wafer carries 200 chips (parts), all parts are identical, and the yield is

$$\text{yield} = 70 + 0.2 \times (W - 80)\% \tag{1.3}$$

Currently Sumo has a profit margin of 35 percent. Sumo is currently running at 100 wafer starts per week for Mr. Big and Ms. Smart. Sumo thinks they can get 50 cents more out of Mr. Big for his chips, but Ms. Smart won't pay any more. We

TABLE 1.4 Die size limits for ASIC packages.

Package[1]	Number of pins or leads	Maximum die size[2] (mil^2)	Minimum die size[3] (mil^2)
PLCC	44	320 × 320	94 × 94
PLCC	68	420 × 420	154 × 154
PLCC	84	395 × 395	171 × 171
PQFP	100	338 × 338	124 × 124
PQFP	144	350 × 350	266 × 266
PQFP	160	429 × 429	248 × 248
PQFP	208	501 × 501	427 × 427
CPGA	68	480 × 480	200 × 200
CPGA	84	370 × 370	200 × 200
CPGA	120	480 × 480	175 × 175
CPGA	144	470 × 470	250 × 250
CPGA	223	590 × 590	290 × 290
CPGA	299	590 × 590	470 × 470
PPGA	64	230 × 230	120 × 120
PPGA	84	380 × 380	150 × 150
PPGA	100	395 × 395	150 × 150
PPGA	120	395 × 395	190 × 190
PPGA	144	660 × 655	230 × 230
PPGA	180	540 × 540	330 × 330
PPGA	208	500 × 500	395 × 395

[1]PLCC = plastic leaded chip carrier, PQFP = plastic quad flat pack, CPGA = ceramic pin-grid array, PPGA = plastic pin-grid array.
[2]Maximum die size is not standard and varies between manufacturers.
[3]Minimum die size is an estimate based on bond length restrictions.

can calculate how much Sumo can afford to lose per chip if they want Ms. Teeny's business really badly.

a. What is Sumo's current yield?

b. How many good parts is Sumo currently producing per week? (*Hint:* Is this enough to supply Mr. Big and Ms. Smart?)

c. Calculate how many extra wafer starts per week we need to supply Ms. Teeny (the yield will change—what is the new yield?). Think when you give this answer.

d. What is Sumo's increase in costs to supply Ms. Teeny?

e. Multiply your answer to part d by 1.35 (to account for Sumo's profit). This is the increase in revenue we need to cover our increased costs to supply Ms. Teeny.

f. Now suppose we charge Mr. Big 50 cents more per part. How much extra revenue does that generate?

g. How much does Ms. Teeny's extra business reduce the wafer cost?

h. How much can Sumo Silicon afford to lose on each of Ms. Teeny's parts, cover its costs, and still make a 35 percent profit?

1.5 (Silicon, 20 min.) How much does a 6-inch silicon wafer weigh? a 12-inch wafer? How much does a carrier (called a boat) that holds twenty 12-inch wafers weigh? What implications does this have for manufacturing?

a. How many die that are 1-inch on a side does a 12-inch wafer hold? If each die is worth $100, how much is a 20-wafer boat worth? If a factory is processing 10 of these boats in different furnaces when the power is interrupted and those wafers have to be scrapped, how much money is lost?

b. The size of silicon factories (fabs or foundries) is measured in wafer starts per week. If a factory is capable of 5000 12-inch wafer starts per week, with an average die of 500 mil on a side that sells for $20 and 90 percent yield, what is the value in dollars/year of the factory production? What fraction of the current gross national (or domestic) product (GNP/GDP) of your country is that? If the yield suddenly drops from 90 percent to 40 percent (a yield bust) how much revenue is the company losing per day? If the company has a cash reserve of $100 million and this revenue loss drops "straight to the bottom line," how long does it take for the company to go out of business?

c. TSMC produced 2 million 6-inch wafers in 1996, how many 500 mil die is that? TSMC's $500 million Camas fab in Washington is scheduled to produce 30,000 8-inch wafers per month by the year 2000 using a 0.35 μm process. If a 1 Mb SRAM yields 1500 good die per 8-inch wafer and there are 1700 gross die per wafer, what is the yield? What is the die size? If the SRAM cell size is 7 μm^2, what fraction of the die is used by the cells? What is TSMC's cost per bit for SRAM if the wafer cost is $2000? If a 16Mb DRAM on the same fab line uses a 16 mm^2 die, what is the cost per bit for DRAM assuming the same yield?

1.6 (Simulation time, 30 min.) "...The system-level simulation used approximately 4000 lines of SPARC assembly language...each simulation clock was simulated in three real time seconds" (Sun Technology article).

a. With a 20 MHz clock how much slower is simulated time than real time?

b. How long would it take to simulate all 4000 lines of test code? (Assume one line of assembly code per cycle—a good approximation compared to the others we are making.)

The article continues: "the entire system was simulated, running actual code, including several milliseconds of SunOS execution. Four days after power-up, SPARCstation 1 booted SunOS and announced: `'hello world'`."

c. How long would it take to simulate 5 ms of code?

d. Find out how long it takes to boot a UNIX workstation in real time. How many clock cycles is this?

e. The machine is not executing boot code all this time; you have to wait for disk drives to spin-up, file systems checks to complete, and so on. Make some estimates as to how much code is required to boot an operating system (OS) and how many clock cycles this would take to execute.

The number of clock cycles you need to simulate to boot a system is somewhere between your answers to parts d and e.

f. From your answers make an estimate of how long it takes to simulate booting the OS. Does this seem reasonable?

g. Could the engineers have simulated a complete boot sequence?

h. Do you think the engineers expected the system to boot on first silicon, given the complexity of the system and how long they would have to wait to simulate a complete boot sequence? Explain.

1.7 (Price per gate, 5 min.) Given the assumptions of Section 1.4.4 on the price per gate of different ASIC technologies, what has to change for the price per gate for an FPGA to be less than that for an MGA or CBIC—if all three use the same process?

1.8 (Pentiums, 20 min.) Read the online tour of the Pentium Pro at `http://www.intel.com` (adapted from a paper presented at the 1995 International Solid-State Circuits Conference). This is not an ASIC design; notice the section on full-custom circuit design. Notice also the comments on the use of `'assert'` statements in the HDL code that described the circuits. Find out the approximate cost of the Intel Pentium (3.3 million transistors) and Pentium Pro (5.5 million transistors) microprocessors.

a. Assuming there a four transistors per gate equivalent, what is the price per gate?

b. Find out the cost of a 1 Mb, 4 Mb, 8 Mb, or 16 Mb DRAM. Assuming one transistor per memory bit, what is the price per gate of DRAM?

c. Considering that both have roughly the same die size, are just as complex to design and to manufacture, why is there such a huge difference in price per gate between microprocessors and DRAM?

1.9 (Inverse embedded arrays, 10 min.) A relatively new cousin of the embedded gate array, the **inverse-embedded gate array**, is a cell-based ASIC that contains an embedded gate-array megacell. List the features as well as the advantages and disadvantages of this type of ASIC in the same way as for the other members of the ASIC family in Section 1.1.

1.10 (0.5-gate design, 60 min.) It is a good idea to complete a 0.5-gate ASIC design (an inverter connected between an input pad and an output pad) in the first week (day) of class. Capture the commands in a report that shows all the steps taken to create your chip starting from an empty directory—`halfgate`.

1.11 (Filenames, 30 min.) Start a list of filename extensions used in ASIC design. Table 1.5 shows an example. Expand this list as you use more tools.

TABLE 1.5 CAD tool filename extensions.

Extension	Description	From	To
`.ini`	Viewlogic startup file, library search paths, etc.	Viewlogic/Viewdraw	Internal tools use other Viewlogic tools
`.wir`	Schematic file		

1.8 Bibliography

The Addison-Wesley VLSI Design Series covers all aspects of VLSI design. Mead and Conway [1980] is an introduction to VLSI design. Glasser and Dobberpuhl [1985] deal primarily with NMOS technology, but their book is still a valuable circuit design reference. Bakoglu's book [1990] concentrates on system interconnect issues. Both editions of Weste and Eshraghian [1993] describe full-custom VLSI design.

Other books on CMOS design include books by Kang and Leblebici [1996], Wolf [1994], Price [1994], Hurst [1992], and Shoji [1988]. Alvarez [1993] covers BiCMOS, but concentrates more on technology than design. Embabi, Bellaouar, and Elmasry [1993] also cover BiCMOS design from a similar perspective. Elmasry's book [1994] contains a collection of papers on BiCMOS design. Einspruch and Hilbert [1991]; Huber and Rosneck [1991]; and Veendrick [1992] are introductions to ASIC design for nontechnical readers. Long and Butner [1990] cover gallium arsenide (GaAs) IC design. Most books on CMOS and ASIC design are classified in the TK7874 section of the Library of Congress catalog (T is for technology).

Several journals and magazines publish articles on ASICs and ASIC design. The *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (ISSN 1063-8210, TK7874.I3273, 1993–) is dedicated to VLSI design. The *IEEE Custom Integrated Circuits Conference* (ISSN 0886-5930, TK7874.C865, 1979–) and the *IEEE International ASIC Conference* (TK7874.6.I34a, 1988–1991; TK7874.6.I35,

ISSN 1063-0988, 1991–) both cover the design and use of ASICs. *EE Times* (ISSN 0192-1541, `http://techweb.cmp.com/eet`) is a newsletter that includes a wide-ranging coverage of system design, ASICs, and ASIC design. *Integrated System Design* (ISSN 1080-2797), formerly *ASIC & EDA*) is a monthly publication that includes ASIC design topics. *High Performance Systems* (ISSN 0887-9664), formerly *VLSI Design* (ISSN 0279-2834), deals with system design including the use of ASICs. *EDN* (ISSN 0012-7515, `http://www.ednmag.com`) has broader coverage of the electronics industry, including articles on VLSI and systems design. *Computer Design* (ISSN 0010-4566) is targeted at systems-level design but includes coverage of ASICs (for example, a special issue in August 1996 was devoted to ASIC design).

The Electronic Industries Association (EIA) has produced a standard, JESD12-1B, "Terms and definitions for gate arrays and cell-based digital integrated circuits," to define terms and definitions.

University Video Communication (`http://www.uvc.com`) produces several videotapes on computer science and engineering topics including ASIC design. Maly's book [1987] is a picture book containing drawings and cross-sections of devices, and shows how a transistor is fabricated.

It is difficult to obtain detailed technical information from ASIC companies and vendors apart from the glossy brochures (**sparkle sheets**). It used to be possible to obtain data books on cell libraries (now these are large and difficult to produce, and are often only available in electronic form) as well as design guidelines and handbooks. Fortunately there are now many resources available on the World Wide Web, which are, of course, constantly changing. EDAC (Electronic Design Automation Companies) has a Web page (`http://www.edac.org`) with links to most of the EDA companies. The Electrical Engineering page on the World Wide Web (E2W3) (`http://www.e2w3.com`) contains links to many ASIC related areas, including distributors, ASIC companies, and semiconductor companies. SEMATECH (Semiconductor Manufacturing Technology) is a nonprofit consortium of U.S. semiconductor companies and has a Web page (`http://www.sematech.org`) that includes links to major semiconductor manufacturers. The MIT Semiconductor Subway (`http://www-mtl.mit.edu`) is more oriented toward devices, processes, and materials but contains links to other VLSI industrial and academic areas. There is a list of EDA companies at `http://www.yahoo.com` under `Business_and_Economy` in `Companies/Computers/Software/Graphics/CAD/IC_Design`.

The MOS Implementation Service (MOSIS), located at the Information Sciences Institute (ISI) at the University of Southern California (USC), is a "silicon broker" for universities in the United States and also provides commercial access to fabrication facilities (`http://www.isi.edu`). Professor Don Bouldin maintains The Microelectronic Systems Newsletter, formerly the MOSIS Users Group (MUG) Newsletter, at `http://www-ece.engr.utk.edu/ece`.

NASA (`http://nppp.jpl.nasa.gov/dmg/jpl/loc/asic`) has an extensive online ASIC guide, developed by the Office of Safety and Mission Assurance, that covers ASIC management, vendor evaluation, design, and part acceptance.

1.9 References

Alvarez, A. R. (Ed.). 1993. *BiCMOS Technology and Applications.* Norwell, MA: Kluwer. ISBN 0-7923-9384-8. TK7871.99.M44.

Bakoglu, H. B. 1990. *Circuits, Interconnections, and Packaging for VLSI.* Reading, MA: Addison-Wesley, 527 p. ISBN 0-86341-165-7. TK7874.B345. Based on a Stanford Ph.D. thesis and contains chapters on: devices and interconnections, packaging, transmission lines, cross talk, clocking of high-speed systems, system level performance.

Einspruch N. G., and J. L. Hilbert (Eds.). 1991. *Application Specific Integrated Circuit (ASIC) Technology.* San Diego, CA: Academic Press. ISBN 0122341236. TK7874.V56 vol. 23. Includes: "Introduction to ASIC technology," Hilbert; "Market dynamics of the ASIC revolution," Collett; "Marketing ASICs," Chakraverty; "Design and architecture of ASIC products," Hickman et al.; "Model and library development," Lubhan; "Computer-aided design tools and systems," Rowson; "ASIC manufacturing," Montalbo; "Test and testability of ASICs," Rosqvist; "Electronic packaging for ASICs," Herrell and Prokop; "Application and selection of ASICs," Mitchell; "Designing with ASICs," Wilkerson; "Quality and reliability," Young.

Elmasry, M. I. 1994. *BiCMOS Integrated Circuit Design: with Analog, Digital, and Smart Power Applications.* New York: IEEE Press, ISBN 0780304306. TK7871.99.M44.B53.

Embabi, S. H. K., A. Bellaouar, and M. I. Elmasry. 1993. *Digital BiCMOS Integrated Circuit Design.* Norwell: MA: Kluwer, 398 p. ISBN 0-7923-9276-0. TK7874.E52.

Glasser, L. A., and D. W. Dobberpuhl. 1985. *The Design and Analysis of VLSI Circuits.* Reading, MA: Addison-Wesley, 473 p. ISBN 0-201-12580-3. TK7874.G573. Detailed analysis of circuits, but largely nMOS.

Huber, J. P., and M. W. Rosneck. 1991. *Successful ASIC Design the First Time Through.* New York: Van Nostrand Reinhold, 200 p. ISBN 0-442-00312-9. TK7874.H83.

Hurst, S. L. 1992. *Custom VLSI Microelectronics.* Englewood Cliffs, NJ: Prentice-Hall, 466 p. ISBN 0-13-194416-9. TK7874.H883.

Kang, S-M, and Y. Leblebici. 1996. *CMOS Digital Integrated Circuits: Analysis and Design.* New York: McGraw-Hill, 614 p. ISBN 0070380465.

Long, S. I., and S. E. Butner. 1990. *Gallium Arsenide Digital Integrated Circuit Design.* New York: McGraw-Hill, 486 p. ISBN 0-07-038687-0. TK7874.L66.

Maly, W. 1987. *Atlas of IC Technologies: An Introduction to VLSI Processes.* Menlo Park, CA: Benjamin-Cummings, 340 p. ISBN 0-8053-6850-7. TK7874.M254. Cross-sectional drawings showing construction of nMOS and CMOS processes.

Mead, C. A., and L. A. Conway. 1980. *Introduction to VLSI Systems.* Reading, MA: Addison-Wesley, 396 p. ISBN 0-201-04358-0. TK7874.M37.

Price, T. E. 1994. *Introduction to VLSI Technology.* Englewood Cliffs, NJ: Prentice-Hall, 280 p. ISBN 0-13-500422-5. TK7874.P736.

Shoji, M. 1988. *CMOS Digital Circuit Technology.* Englewood Cliffs, NJ: Prentice-Hall, 434 p. ISBN 0131388436. TK7871.99.M44. See also Shoji, M., *High Speed Digital Circuits,* Reading, MA: Addison-Wesley, 1996, 360 p., ISBN 0-201-63483-X, TK7874.65.S56

Weste, N. H. E., and K. Eshraghian. 1993. *Principles of CMOS VLSI Design: A Systems Perspective.* 2nd ed. Reading, MA: Addison-Wesley, 713 p. ISBN 0-201-53376-6. TK7874.W46. Concentrates on full-custom design.

Wolf, W. H. 1994. *Modern VLSI Design: A Systems Approach.* Englewood Cliffs, NJ: Prentice-Hall, 468 p. ISBN 0-13-588377-6. TK7874.65.W65.

Veendrick, H. J. M. 1992. *MOS ICs from Basics to ASICs.* New York: VCH, ISBN 1-56081197-8. TK7874.V397.

CMOS LOGIC

2

A **CMOS transistor** (or device) has four terminals: **gate**, **source**, **drain**, and a fourth terminal that we shall ignore until the next section. A CMOS transistor is a switch. The switch must be conducting or *on* to allow current to flow between the source and drain terminals (using open and closed for switches is confusing—for the same reason we say a tap is *on* and not that it is *closed*). The transistor source and drain terminals are equivalent as far as digital signals are concerned—we do not worry about labeling an electrical switch with two terminals.

- V_{AB} is the potential difference, or voltage, between nodes A and B in a circuit; V_{AB} is positive if node A is more positive than node B.
- Italics denote variables; constants are set in roman (upright) type. Uppercase letters denote DC, large-signal, or steady-state voltages.
- For TTL the positive power supply is called VCC (V_{CC} or V_{CC}). The 'C' denotes that the supply is connected indirectly to the collectors of the *npn* bipolar transistors (a bipolar transistor has a collector, base, and emitter—corresponding roughly to the drain, gate, and source of an MOS transistor).
- Following the example of TTL we used VDD (V_{DD} or V_{DD}) to denote the positive supply in an NMOS chip where the devices are all *n*-channel transistors and the drains of these devices are connected indirectly to the positive supply. The supply nomenclature for NMOS chips has stuck for CMOS.

- VDD is the name of the power supply node or net; V_{DD} represents the value (uppercase since V_{DD} is a DC quantity). Since V_{DD} is a variable, it is italic (words and multiletter abbreviations use roman—thus it is V_{DD}, but V_{drain}).
- Logic designers often call the CMOS negative supply VSS or V_{SS} even if it is actually ground or GND. I shall use VSS for the node and V_{SS} for the value.
- CMOS uses **positive logic**—VDD is logic '1' and VSS is logic '0'.

We turn a transistor on or off using the gate terminal. There are two kinds of CMOS transistors: *n*-channel transistors and *p*-channel transistors. An *n*-channel transistor requires a logic '1' (from now on I'll just say a '1') on the gate to make the switch conducting (to turn the transistor *on*). A *p*-channel transistor requires a logic '0' (again from now on, I'll just say a '0') on the gate to make the switch conducting (to turn the transistor *on*). The *p*-channel transistor symbol has a bubble on its gate to remind us that the gate has to be a '1' to turn the transistor *off*. All this is shown in Figure 2.1(a) and (b).

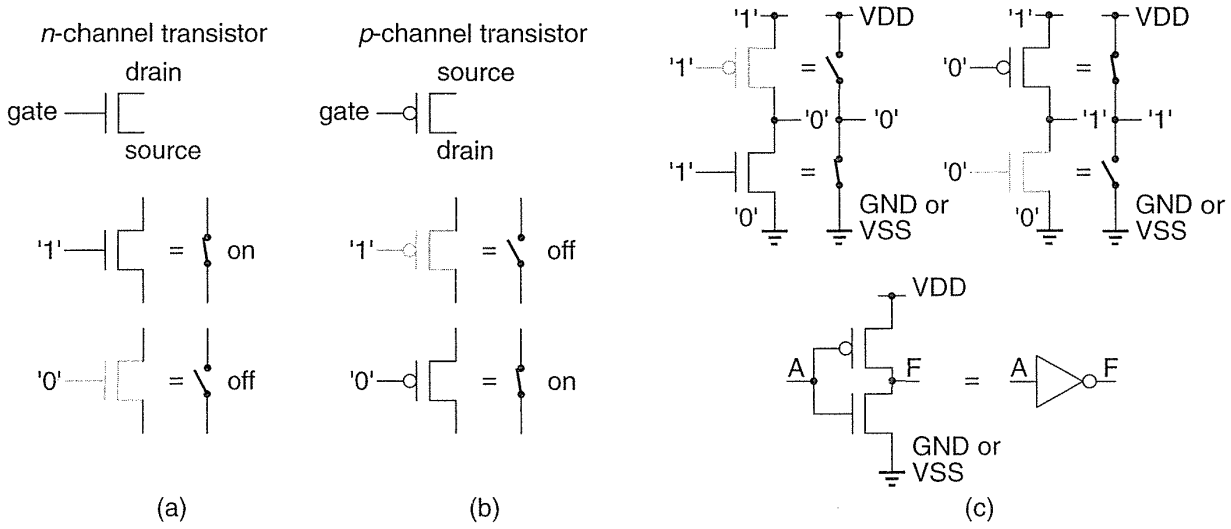


FIGURE 2.1 CMOS transistors as switches. (a) An *n*-channel transistor. (b) A *p*-channel transistor. (c) A CMOS inverter and its symbol (an *equilateral* triangle and a *circle*).

If we connect an *n*-channel transistor in series with a *p*-channel transistor, as shown in Figure 2.1(c), we form an **inverter**. With four transistors we can form a two-input **NAND gate** (Figure 2.2a). We can also make a two-input **NOR gate** (Figure 2.2b). Logic designers normally use the terms NAND gate and logic gate (or just gate), but I shall try to use the terms NAND **cell** and **logic cell** rather than NAND gate or logic gate in this chapter to avoid any possible confusion with the gate terminal of a transistor.

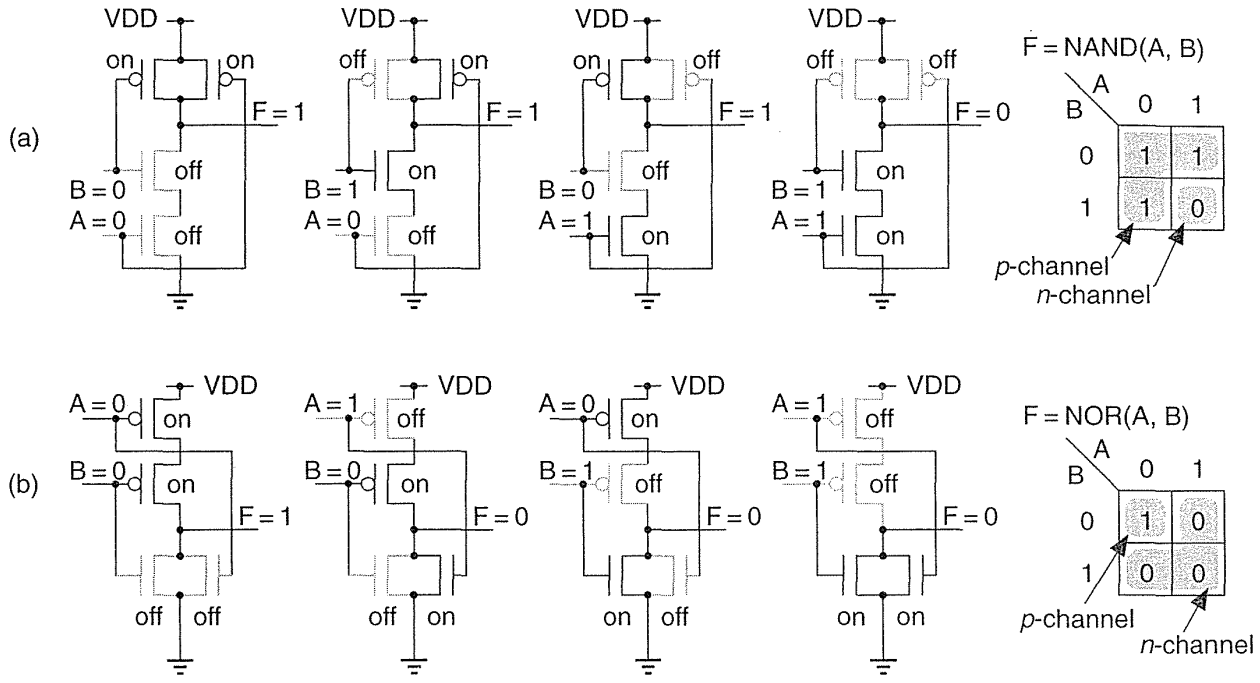


FIGURE 2.2 CMOS logic. (a) A two-input NAND logic cell. (b) A two-input NOR logic cell. The *n*-channel and *p*-channel transistor switches implement the '1's and '0's of a Karnaugh map.

2.1 CMOS Transistors

Figure 2.3 illustrates how electrons and holes abandon their dopant atoms leaving a **depletion region** around a transistor's source and drain. The region between source and drain is normally nonconducting. To make an *n*-channel transistor conducting, we must apply a positive voltage V_{GS} (the gate voltage with respect to the source) that is greater than the *n*-channel transistor **threshold voltage**, V_{tn} (a typical value is 0.5 V and, as far as we are presently concerned, is a constant). This establishes a thin (≈50 Å) conducting channel of electrons under the gate. MOS transistors can carry a very small current (the **subthreshold current**—a few microamperes or less) with $V_{GS} < V_{tn}$, but we shall ignore this. A transistor can be conducting ($V_{GS} > V_{tn}$) without any current flowing. To make current flow in an *n*-channel transistor we must also apply a positive voltage, V_{DS}, to the drain with respect to the source. Figure 2.3 shows these connections and the connection to the fourth terminal of an MOS transistor—the **bulk** (**well**, **tub**, or **substrate**) terminal. For an *n*-channel transistor we must connect the bulk to the most negative potential, GND or VSS, to reverse bias the bulk-to-drain and bulk-to-source *pn*-diodes. The arrow in the four-terminal *n*-channel transistor symbol in Figure 2.3 reflects the polarity of these *pn*-diodes.

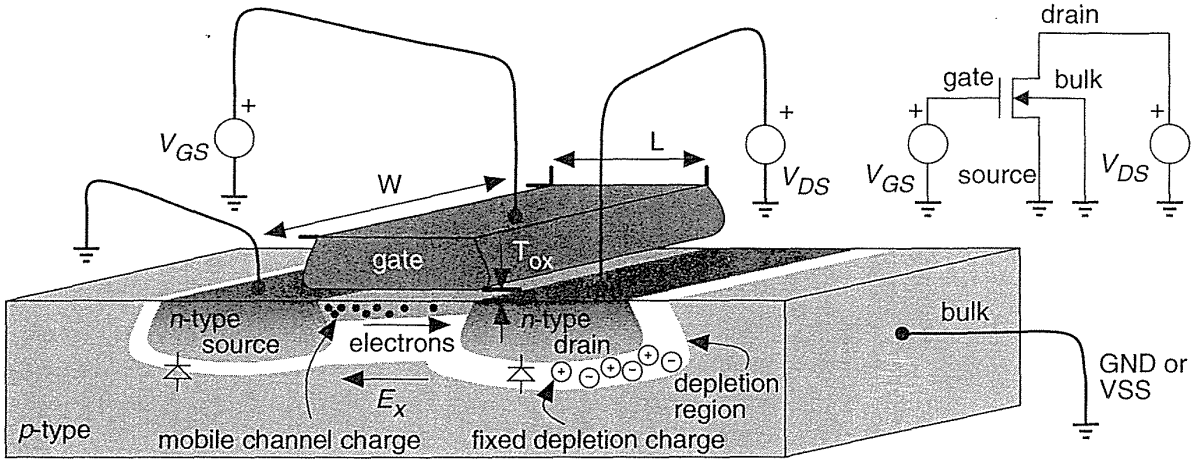


FIGURE 2.3 An *n*-channel MOS transistor. The gate-oxide thickness, T_{OX}, is approximately 100 angstroms (0.01 μm). A typical transistor length, $L = 2\lambda$. The bulk may be either the substrate or a well. The diodes represent *pn*-junctions that must be reverse-biased.

The current flowing in the transistor is

current (amperes) = charge (coulombs) per unit time (second). (2.1)

We can express the current in terms of the total charge in the channel, Q (imagine taking a picture and counting the number of electrons in the channel at that instant). If t_f (for **time of flight**—sometimes called the **transit time**) is the time that it takes an electron to cross between source and drain, the drain-to-source current, I_{DSn}, is

$$I_{DSn} = \frac{Q}{t_f}. \tag{2.2}$$

We need to find Q and t_f. The velocity of the electrons $\mathbf{v}$ (a vector) is given by the equation that forms the basis of Ohm's law:

$$\mathbf{v} = -\mu_n \mathbf{E}, \tag{2.3}$$

where μ_n is the **electron mobility** (μ_p is the **hole mobility**) and $\mathbf{E}$ is the electric field (with units Vm^{-1}).

Typical **carrier mobility** values are $\mu_n = 500\text{–}1000\ cm^2V^{-1}s^{-1}$ and $\mu_p = 100\text{–}400\ cm^2V^{-1}s^{-1}$. Equation 2.3 is a vector equation, but we shall ignore the vertical electric field and concentrate on the horizontal electric field, E_x, that moves the electrons between source and drain. The horizontal component of the electric

field is $E_x = -V_{DS}/\mathrm{L}$, directed from the drain to the source, where L is the channel length (see Figure 2.3). The electrons travel a distance L with horizontal velocity $v_x = -\mu_n E_x$, so that

$$t_f = \frac{\mathrm{L}}{v_x} = \frac{\mathrm{L}^2}{\mu_n V_{DS}}. \tag{2.4}$$

Next we find the channel charge, Q. The channel and the gate form the plates of a capacitor, separated by an insulator—the gate oxide. We know that the charge on a linear capacitor, C, is $Q = CV$. Our lower plate, the channel, is not a linear conductor. Charge only appears on the lower plate when the voltage between the gate and the channel, V_{GC}, exceeds the n-channel threshold voltage. For our nonlinear capacitor we need to modify the equation for a linear capacitor to the following:

$$Q = \mathrm{C}(V_{GC} - \mathrm{V}_{tn}). \tag{2.5}$$

The lower plate of our capacitor is resistive and conducting current, so that the potential in the channel, V_{GC}, varies. In fact, $V_{GC} = V_{GS}$ at the source and $V_{GC} = V_{GS} - V_{DS}$ at the drain. What we really should do is find an expression for the channel charge as a function of channel voltage and sum (integrate) the charge all the way across the channel, from $x = 0$ (at the source) to $x = \mathrm{L}$ (at the drain). Instead we shall assume that the channel voltage, $V_{GC}(x)$, is a linear function of distance from the source and take the average value of the charge, which is thus

$$Q = \mathrm{C}\left[(V_{GS} - \mathrm{V}_{tn}) - \frac{1}{2}V_{DS}\right]. \tag{2.6}$$

The gate capacitance, C, is given by the formula for a parallel-plate capacitor with length L, width W, and plate separation equal to the gate-oxide thickness, T_{ox}. Thus the gate capacitance is

$$\mathrm{C} = \frac{\mathrm{WL}\varepsilon_{ox}}{\mathrm{T}_{ox}} = \mathrm{WLC}_{ox}, \tag{2.7}$$

where ε_{ox} is the gate-oxide dielectric permittivity. For silicon dioxide, $Si0_2$, $\varepsilon_{ox} \approx 3.45 \times 10^{-11}\,\mathrm{Fm}^{-1}$, so that, for a typical gate-oxide thickness of 100 Å (1Å = 1 angstrom = 0.1 nm), the gate capacitance per unit area, $\mathrm{C}_{ox} \approx 3\,\mathrm{fF\mu m}^{-2}$.

Now we can express the channel charge in terms of the transistor parameters,

$$Q = \mathrm{WLC}_{ox}\left[(V_{GS} - \mathrm{V}_{tn}) - \frac{1}{2}V_{DS}\right]. \tag{2.8}$$

Finally, the drain–source current is

$$I_{DSn} = \frac{Q}{t_f} = \frac{W}{L}\mu_n C_{ox}\left[(V_{GS} - V_{tn}) - \frac{1}{2}V_{DS}\right]V_{DS}$$
$$= \frac{W}{L}k_n'\left[(V_{GS} - V_{tn}) - \frac{1}{2}V_{DS}\right]V_{DS}. \tag{2.9}$$

The constant k_n' is the process transconductance parameter (or **intrinsic transconductance**):

$$k_n' = \mu_n C_{ox}. \tag{2.10}$$

We also define β_n, the **transistor gain factor** (or just **gain factor**) as

$$\beta_n = k_n'\frac{W}{L}. \tag{2.11}$$

The factor W/L (transistor width divided by length) is the transistor **shape factor.**

Equation 2.9 describes the **linear region** (or triode region) of operation. This equation is valid until $V_{DS} = V_{GS} - V_{tn}$ and then predicts that I_{DS} decreases with increasing V_{DS}, which does not make physical sense. At $V_{DS} = V_{GS} - V_{tn} = V_{DS(sat)}$ (the **saturation voltage**) there is no longer enough voltage between the gate and the drain end of the channel to support any channel charge. Clearly a small amount of charge remains or the current would go to zero, but with very little free charge the channel resistance in a small region close to the drain increases rapidly and any further increase in V_{DS} is dropped over this region. Thus for $V_{DS} > V_{GS} - V_{tn}$ (the **saturation region**, or pentode region, of operation) the drain current I_{DS} remains approximately constant at the **saturation current**, $I_{DSn(sat)}$, where

$$I_{DSn(sat)} = \frac{\beta_n}{2}(V_{GS} - V_{tn})^2; \quad V_{DS} > V_{GS} - V_{tn}. \tag{2.12}$$

Figure 2.4 shows the n-channel transistor I_{DS}–V_{DS} characteristics for a generic 0.5 μm CMOS process that we shall call **G5**. We can fit Eq. 2.12 to the long-channel transistor characteristics (W = 60 μm, L = 6 μm) in Figure 2.4(a). If $I_{DSn(sat)} = 2.5$ mA (with $V_{DS} = 3.0$ V, $V_{GS} = 3.0$ V, $V_{tn} = 0.65$ V, $T_{ox} = 100$ Å), the intrinsic transconductance is

$$k_n' = \frac{2(L/W)I_{DSn(sat)}}{(V_{GS} - V_{tn})^2} = \frac{2(6/60)(2.5\times10^{-3})}{(3.0 - 0.65)^2} = 9.05\times10^{-5}\,AV^{-2}, \tag{2.13}$$

or approximately $90\,\mu AV^{-2}$. This value of k_n', calculated in the saturation region, will be different (typically lower by a factor of 2 or more) from the value of k_n' measured in the linear region. We assumed the mobility, μ_n, and the threshold voltage, V_{tn}, are constants—neither of which is true, as we shall see in Section 2.1.2.

For the p-channel transistor in the G5 process, $I_{DSp(sat)} = -850\,\mu A$ ($V_{DS} = -3.0$ V, $V_{GS} = -3.0$ V, $V_{tp} = -0.85$ V, W = 60 μm, L = 6 μm). Then

$$k'_p = \frac{2\,(L/W)\,(-I_{DSp\,(sat)})}{(V_{GS} - V_{tp})^2} = \frac{2\,(6/60)\,(850\times10^{-6})}{(-3.0-(-0.85))^2} = 3.68\times10^{-5}\,AV^{-2}. \quad (2.14)$$

The next section explains the signs in Eq. 2.14.

2.1.1 P-Channel Transistors

The source and drain of CMOS transistors look identical; we have to know which way the current is flowing to distinguish them. The source of an n-channel transistor is lower in potential than the drain and vice versa for a p-channel transistor. In an n-channel transistor the threshold voltage, V_{tn}, is normally positive, and the terminal voltages V_{DS} and V_{GS} are also usually positive. In a p-channel transistor V_{tp} is normally negative and we have a choice: We can write everything in terms of the magnitudes of the voltages and currents or we can use negative signs in a consistent fashion.

Here are the equations for a p-channel transistor using negative signs:

$$\begin{aligned} I_{DSp} &= -k'_p \frac{W}{L}\left[(V_{GS} - V_{tp}) - \frac{1}{2}V_{DS}\right]V_{DS} \qquad V_{DS} > V_{GS} - V_{tp} \\ I_{DSp\,(sat)} &= \frac{-\beta_p}{2}(V_{GS} - V_{tp})^2 \qquad V_{DS} < V_{GS} - V_{tp} \end{aligned} \quad (2.15)$$

In these two equations V_{tp} is negative, and the terminal voltages V_{DS} and V_{GS} are also normally negative (and -3 V < -2 V, for example). The current I_{DSp} is then negative, corresponding to conventional current flowing from source to drain of a p-channel transistor (and hence the negative sign for $I_{DSp(sat)}$ in Eq. 2.14).

2.1.2 Velocity Saturation

For a deep submicron transistor, Eq. 2.12 may overestimate the drain–source current by a factor of 2 or more. There are three reasons for this error. First, the threshold voltage is not constant. Second, the actual length of the channel (the electrical or effective length, often written as L_{eff}) is less than the drawn (mask) length. The third reason is that Eq. 2.3 is not valid for high electric fields. The electrons cannot move any faster than about $v_{maxn} = 10^5\ ms^{-1}$ when the electric field is above $10^6\ Vm^{-1}$ (reached when 1 V is dropped across 1 μm); the electrons become **velocity saturated**. In this case $t_f = L_{eff}/v_{maxn}$, the drain–source saturation current is independent of the transistor length, and Eq. 2.12 becomes

$$I_{DSn\,(sat)} = W v_{maxn} C_{ox}\,(V_{GS} - V_{tn})\,;\ V_{DS} > V_{DS\,(sat)} \quad \text{(velocity saturated)}. \quad (2.16)$$

We can see this behavior for the short-channel transistor characteristics in Figure 2.4(a) and (c).

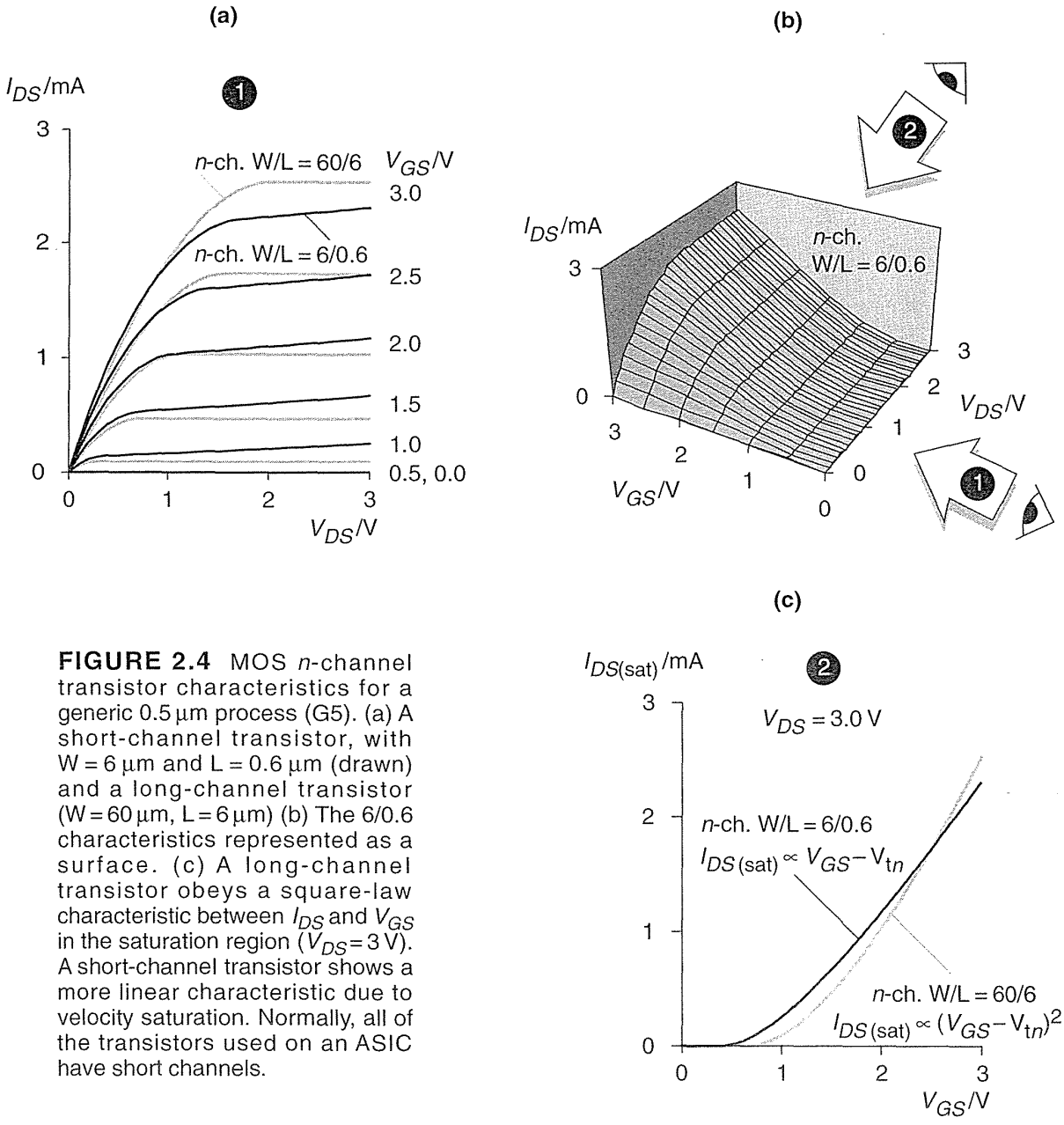


FIGURE 2.4 MOS *n*-channel transistor characteristics for a generic 0.5 μm process (G5). (a) A short-channel transistor, with W = 6 μm and L = 0.6 μm (drawn) and a long-channel transistor (W = 60 μm, L = 6 μm) (b) The 6/0.6 characteristics represented as a surface. (c) A long-channel transistor obeys a square-law characteristic between I_{DS} and V_{GS} in the saturation region (V_{DS} = 3 V). A short-channel transistor shows a more linear characteristic due to velocity saturation. Normally, all of the transistors used on an ASIC have short channels.

Transistor current is often specified per micron of gate width because of the form of Eq. 2.16. As an example, suppose $I_{DSn(sat)}/W = 300\ \mu A \mu m^{-1}$ for the *n*-channel transistors in our G5 process (with $V_{DS} = 3.0$ V, $V_{GS} = 3.0$ V, $V_{tn} = 0.65$ V, $L_{eff} = 0.5\ \mu m$ and $T_{ox} = 100$ Å). Then $E_x \approx (3 - 0.65)\ V / 0.5\ \mu m \approx 5\ V \mu m^{-1}$,

$$v_{maxn} = \frac{I_{DSn(sat)}/W}{C_{ox}(V_{GS} - V_{tn})} = \frac{(300\times10^{-6})\ (1\times10^{6})}{(3.45\times10^{-3})\ (3-0.65)} = 37{,}000\ \text{ms}^{-1}, \quad (2.17)$$

and $t_f \approx 0.5\ \mu\text{m}/37{,}000\ \text{ms}^{-1} \approx 13$ ps.

The value for v_{maxn} is lower than the $10^5\ \text{ms}^{-1}$ we expected because the carrier velocity is also lowered by **mobility degradation** due the vertical electric field—which we have ignored. This vertical field forces the carriers to keep "bumping" in to the interface between the silicon and the gate oxide, slowing them down.

2.1.3 SPICE Models

The simulation program **SPICE** (which stands for **Simulation Program with Integrated Circuit Emphasis**) is often used to characterize logic cells. Table 2.1 shows a typical set of model parameters for our G5 process. The SPICE parameter KP (given in μAV^{-2}) corresponds to k_n' (and k_p'). SPICE parameters VTO and TOX correspond to V_{tn} (and V_{tp}), and T_{ox}. SPICE parameter U0 (given in $cm^2V^{-1}s^{-1}$) corresponds to the ideal **bulk mobility** values, μ_n (and μ_p). Many of the other parameters model velocity saturation and mobility degradation (and thus the effective value of k_n' and k_p').

TABLE 2.1 SPICE parameters for a generic 0.5 μm process, G5 (0.6 μm drawn gate length). The n-channel transistor characteristics are shown in Figure 2.4.

```
.MODEL CMOSN NMOS LEVEL=3 PHI=0.7 TOX=10E-09 XJ=0.2U TPG=1 VTO=0.65 DELTA=0.7
+ LD=5E-08 KP=2E-04 UO=550 THETA=0.27 RSH=2 GAMMA=0.6 NSUB=1.4E+17 NFS=6E+11
+ VMAX=2E+05 ETA=3.7E-02 KAPPA=2.9E-02 CGDO=3.0E-10 CGSO=3.0E-10 CGBO=4.0E-10
+ CJ=5.6E-04 MJ=0.56 CJSW=5E-11 MJSW=0.52 PB=1
.MODEL CMOSP PMOS LEVEL=3 PHI=0.7 TOX=10E-09 XJ=0.2U TPG=-1 VTO=-0.92 DELTA=0.29
+ LD=3.5E-08 KP=4.9E-05 UO=135 THETA=0.18 RSH=2 GAMMA=0.47 NSUB=8.5E+16 NFS=6.5E+11
+ VMAX=2.5E+05 ETA=2.45E-02 KAPPA=7.96 CGDO=2.4E-10 CGSO=2.4E-10 CGBO=3.8E-10
+ CJ=9.3E-04 MJ=0.47 CJSW=2.9E-10 MJSW=0.505 PB=1
```

2.1.4 Logic Levels

Figure 2.5 shows how to use transistors as logic switches. The bulk connection for the *n*-channel transistor in Figure 2.5(a–b) is a *p*-well. The bulk connection for the *p*-channel transistor is an *n*-well. The remaining connections show what happens when we try and pass a logic signal between the drain and source terminals.

In Figure 2.5(a) we apply a logic '1' (or V_{DD}—I shall use these interchangeably) to the gate and a logic '0' (V_{SS}) to the source (we know it is the source since electrons must flow from this point, since V_{SS} is the lowest voltage on the chip). The application of these voltages makes the *n*-channel transistor conduct current, and electrons flow from source to drain.

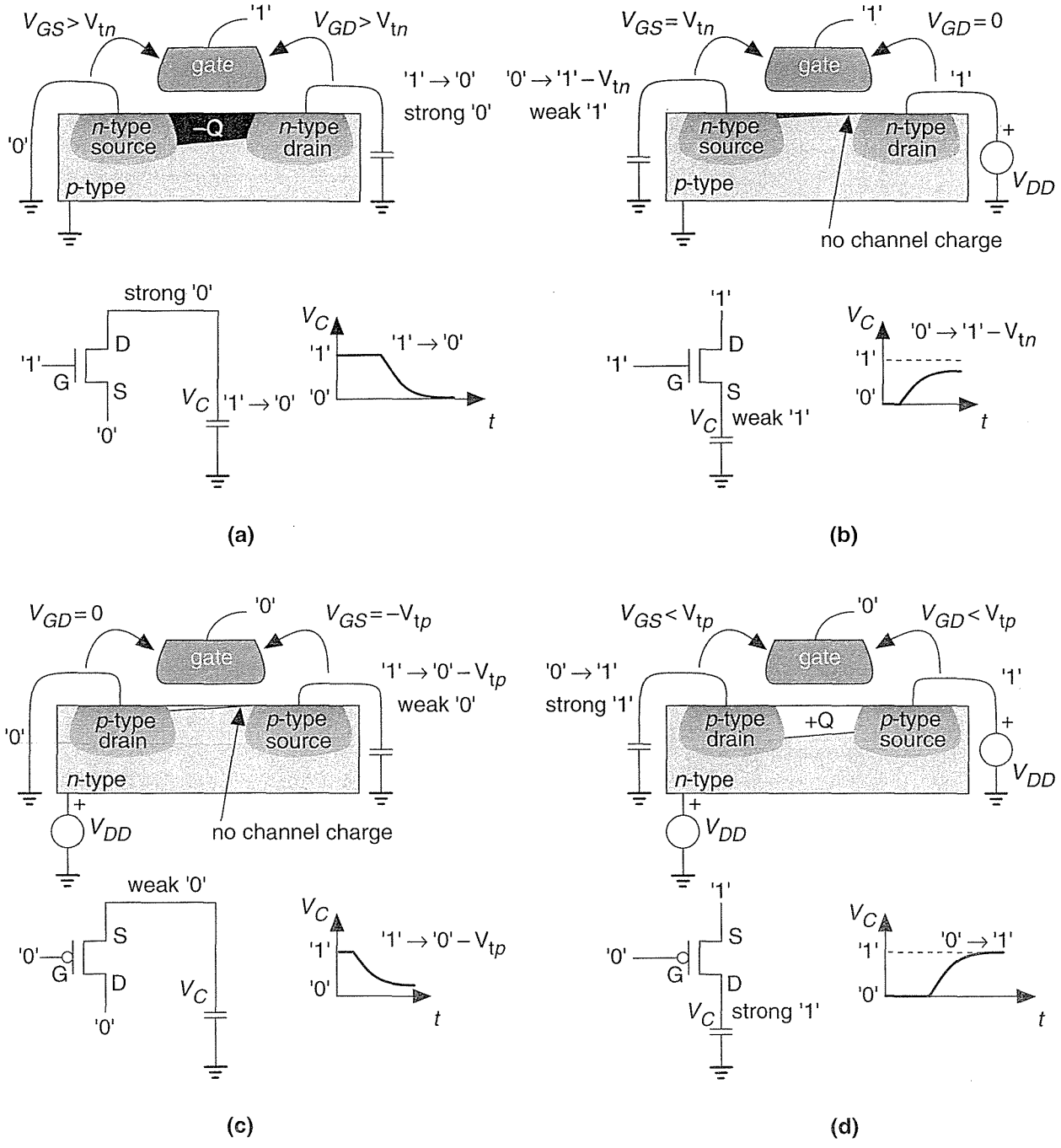


FIGURE 2.5 CMOS logic levels. (a) A strong '0'. (b) A weak '1'. (c) A weak '0'. (d) A strong '1'. (V_{tn} is positive and V_{tp} is negative.) The depth of the channels is greatly exaggerated.

Suppose the drain is initially at logic '1'; then the *n*-channel transistor will begin to discharge any capacitance that is connected to its drain (due to another logic cell, for example). This will continue until the drain terminal reaches a logic '0', and at that time V_{GD} and V_{GS} are both equal to V_{DD}, a full logic '1'. The transistor is strongly conducting now (with a large channel charge, Q, but there is no current flowing since $V_{DS} = 0\,\text{V}$). The transistor will strongly object to attempts to change its drain terminal from a logic '0'. We say that the **logic level** at the drain is a **strong** '0'.

In Figure 2.5(b) we apply a logic '1' to the drain (it must now be the drain since electrons have to flow toward a logic '1'). The situation is now quite different—the transistor is still on but V_{GS} is decreasing as the source voltage approaches its final value. In fact, the source terminal never gets to a logic '1'—the source will stop increasing in voltage when V_{GS} reaches V_{tn}. At this point the transistor is very nearly off and the source voltage creeps slowly up to $V_{DD} - V_{tn}$. Because the transistor is very nearly off, it would be easy for a logic cell connected to the source to change the potential there, since there is so little channel charge. The logic level at the source is a **weak** '1'. Figure 2.5(c–d) show the state of affairs for a *p*-channel transistor is the exact reverse or complement of the *n*-channel transistor situation.

In summary, we have the following logic levels:

- An *n*-channel transistor provides a strong '0', but a weak '1'.
- A *p*-channel transistor provides a strong '1', but a weak '0'.

Sometimes we refer to the weak versions of '0' and '1' as **degraded logic levels**. In CMOS technology we can use both types of transistor together to produce strong '0' logic levels as well as strong '1' logic levels.

2.2 The CMOS Process

Figure 2.6 outlines the steps to create an integrated circuit. The starting material is silicon, Si, refined from quartzite (with less than 1 impurity in 10^{10} silicon atoms). We draw a single-crystal silicon **boule** (or ingot) from a crucible containing a melt at approximately 1500 °C (the melting point of silicon at 1 atm. pressure is 1414 °C). This method is known as Czochralski growth. Acceptor (*p*-type) or donor (*n*-type) dopants may be introduced into the melt to alter the type of silicon grown.

The boule is sawn to form thin circular wafers (6, 8, or 12 inches in diameter, and typically 600 μm thick), and a flat is ground (the primary flat), perpendicular to the <110> crystal axis—as a "this edge down" indication. The boule is drawn so that the wafer surface is either in the (111) or (100) crystal planes. A smaller secondary flat indicates the wafer crystalline orientation and doping type. A typical submicron CMOS processes uses *p*-type (100) wafers with a resistivity of approximately 10 Ωcm—this type of wafer has two flats, 90° apart. Wafers are made by chemical companies and sold to the IC manufacturers. A blank 8-inch wafer costs about $100.

To begin IC fabrication we place a batch of wafers (a **wafer lot**) on a **boat** and grow a layer (typically a few thousand angstroms) of **silicon dioxide**, SiO_2, using a furnace. Silicon is used in the semiconductor industry not so much for the properties of silicon, but because of the physical, chemical, and electrical properties of its native oxide, SiO_2. An IC fabrication **process** contains a series of masking steps (that in turn contain other steps) to create the layers that define the transistors and metal interconnect.

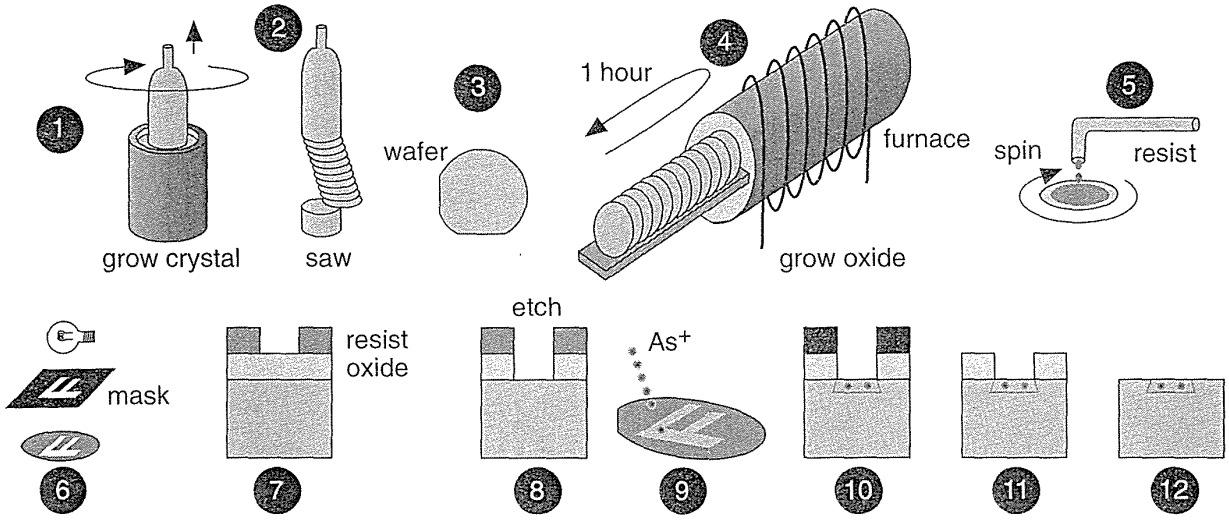


FIGURE 2.6 IC fabrication. Grow crystalline silicon (1); make a wafer (2–3); grow a silicon dioxide (oxide) layer in a furnace (4); apply liquid photoresist (resist) (5); mask exposure (6); a cross-section through a wafer showing the developed resist (7); etch the oxide layer (8); ion implantation (9–10); strip the resist (11); strip the oxide (12). Steps similar to 4–12 are repeated for each layer (typically 12–20 times for a CMOS process).

Each masking step starts by spinning a thin layer (approximately 1 μm) of liquid photoresist (**resist**) onto each wafer. The wafers are baked at about 100 °C to remove the solvent and harden the resist before being exposed to ultraviolet (UV) light (typically less than 200 nm wavelength) through a **mask**. The UV light alters the structure of the resist, allowing it to be removed by developing. The exposed oxide may then be **etched** (removed). Dry **plasma etching** etches in the vertical direction much faster than it does horizontally (an **anisotropic** etch). Wet etch techniques are usually **isotropic**. The resist functions as a mask during the etch step and transfers the desired pattern to the oxide layer.

Dopant ions are then introduced into the exposed silicon areas. Figure 2.6 illustrates the use of **ion implantation**. An **ion implanter is** a cross between a TV and a mass spectrometer and fires dopant ions into the silicon wafer. Ions can only

penetrate materials to a depth (the **range**, normally a few microns) that depends on the closely controlled **implant energy** (measured in keV—usually between 10 and 100 keV; an electron volt, 1 eV, is 1.6×10^{-19} J). By using layers of resist, oxide, and polysilicon we can prevent dopant ions from reaching the silicon surface and thus block the silicon from receiving an **implant**. We control the doping level by counting the number of ions we implant (by integrating the ion-beam current). The **implant dose** is measured in atoms/cm^2 (typical doses are from 10^{13} to 10^{15} cm^{-2}). As an alternative to ion implantation we may instead strip the resist and introduce dopants by diffusion from a gaseous source in a furnace.

Once we have completed the transistor diffusion layers we can deposit layers of other materials. Layers of polycrystalline silicon (polysilicon or **poly**), SiO_2, and silicon nitride (Si_3N_4), for example, may be deposited using **chemical vapor deposition** (**CVD**). Metal layers can be deposited using **sputtering**. All these layers are patterned using masks and similar **photolithography** steps to those shown in Figure 2.6.

TABLE 2.2 CMOS process layers.

Mask/layer name	Derivation from drawn layers	Alternative names for mask/layer	MOSIS mask label
n-well	= nwell[1]	bulk, substrate, tub, *n*-tub, moat	CWN
p-well	= pwell[1]	bulk, substrate, tub, *p*-tub, moat	CWP
active	= pdiff + ndiff	thin oxide, thinox, island, gate oxide	CAA
polysilicon	= poly	poly, gate	CPG
n-diffusion implant[2]	= grow(ndiff)	ndiff, *n*-select, nplus, n+	CSN
p-diffusion implant[2]	= grow(pdiff)	pdiff, *p*-select, pplus, p+	CSP
contact	= contact	contact cut, poly contact, diffusion contact	CCP and CCA[3]
metal1	= m1	first-level metal	CMF
metal2	= m2	second-level metal	CMS
via2	= via2	metal2/metal3 via, m2/m3 via	CVS
metal3	= m3	third-level metal	CMT
glass	= glass	passivation, overglass, pad	COG

[1]If only one well layer is drawn, the other mask may be derived from the drawn layer. For example, *p*-well(mask) = not(nwell(drawn)). A single-well process requires only one well mask.
[2]The implant masks may be derived or drawn.
[3]Largely for historical reasons the contacts to poly and contacts to active have different layer names. In the past this allowed a different sizing or process bias to be applied to each contact type when the mask was made.

Table 2.2 shows the mask layers (and their relation to the drawn layers) for a submicron, silicon-gate, three-level metal, self-aligned, CMOS **process**. A process in which the effective gate length is less than 1 μm is referred to as a **submicron process**. Gate lengths below 0.35 μm are considered in the **deep-submicron** regime.

Figure 2.7 shows the layers that we draw to define the masks for the logic cell of Figure 1.3. Potential confusion arises because we like to keep layout simple but maintain a "what you see is what you get" (WYSIWYG) approach. This means that the drawn layers do not correspond directly to the masks in all cases.

We can construct wells in a CMOS process in several ways. In an **n-well process**, the substrate is *p*-type (the wafer itself) and we use an *n*-well mask to build the *n*-well. We do not need a *p*-well mask because there are no *p*-wells in an *n*-well process—the *n*-channel transistors all sit in the substrate (the wafer)—but we often draw the *p*-well layer as though it existed. In a **p-well process** we use a *p*-well mask to make the *p*-wells and the *n*-wells are the substrate. In a **twin-tub** (or **twin-well**) process, we create individual wells for both types of transistors, and neither well is the substrate (which may be either *n*-type or *p*-type). There are even **triple-well** processes used to achieve even more control over the transistor performance. Whatever process that we use we must connect all the *n*-wells to the most positive potential on the chip, normally VDD, and all the *p*-wells to VSS; otherwise we may forward bias the bulk to source/drain *pn*-junctions. The bulk connections for CMOS transistors are not usually drawn in digital circuit schematics, but these **substrate contacts** (**well contacts** or **tub ties**) are very important. After we make the well(s), we grow a layer (approximately 1500 Å) of Si_3N_4 over the wafer. The **active** mask (CAA) leaves this nitride layer only in the active areas that will later become transistors or substrate contacts. Thus

$$\text{CAA (mask)} = \text{ndiff (drawn)} \vee \text{pdiff (drawn)}, \tag{2.18}$$

the $\vee$ symbol represents OR (union) of the two drawn layers, ndiff and pdiff. Everything outside the active areas is known as the field region, or just **field**.

Next we implant the substrate to prevent unwanted transistors from forming in the field region—this is the **field implant** or **channel-stop implant**. The nitride over the active areas acts as an implant mask and we may use another field-implant mask at this step also. Following this we grow a thick (approximately 5000 Å) layer of SiO_2, the **field oxide** (**FOX**). The FOX will not grow over the nitride areas. When we strip the nitride we are left with FOX in the areas we do *not* want to dope the silicon. Following this we deposit, dope, mask, and etch the poly gate material, CPG (mask) = poly (drawn). Next we create the doped regions that form the sources, drains, and substrate contacts using ion implantation. The poly gate functions like masking tape in these steps. One implant (using phosphorous or arsenic ions) forms the *n*-type source/drain for the *n*-channel transistors and *n*-type substrate contacts

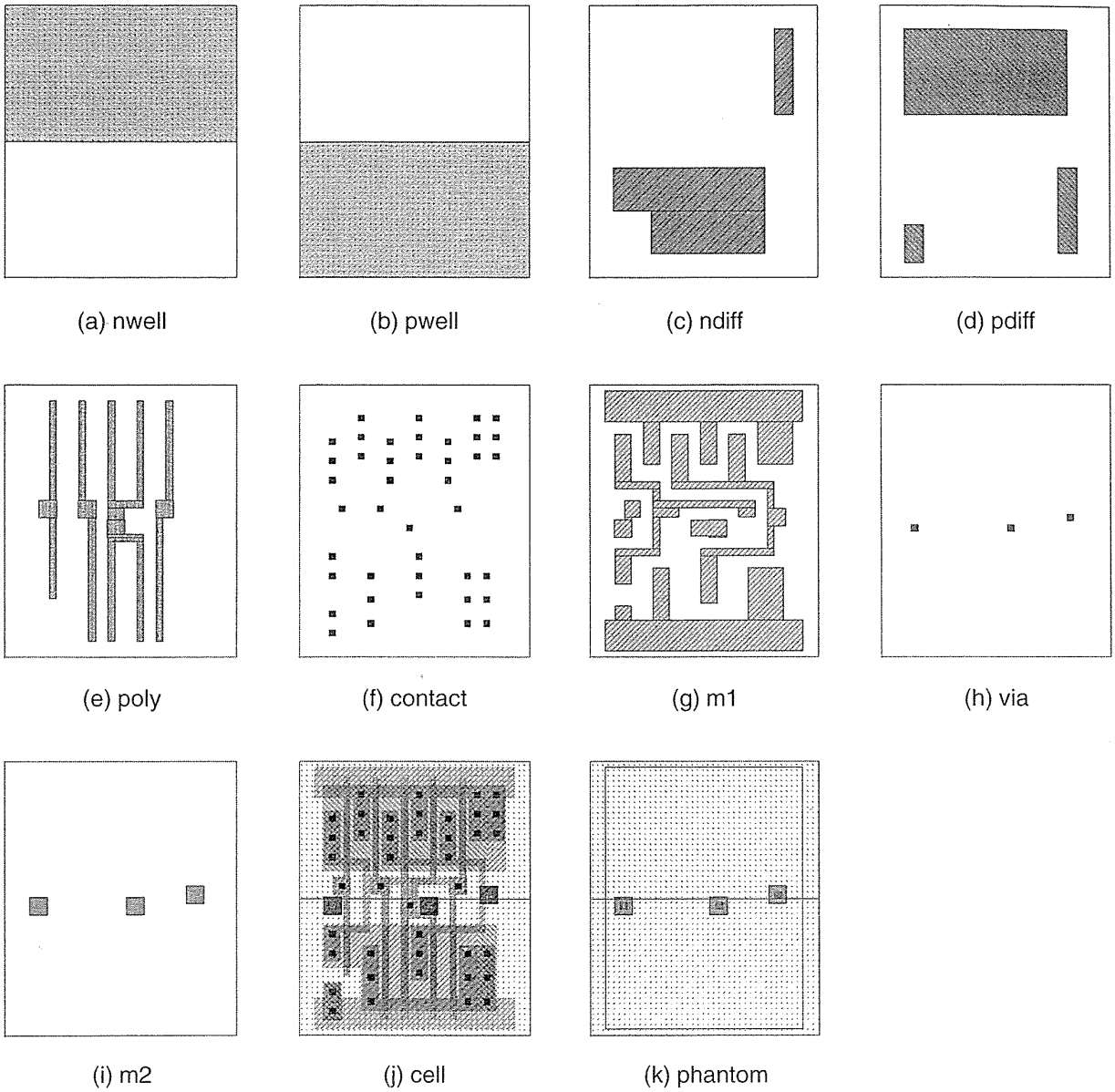


FIGURE 2.7 The standard cell shown in Figure 1.3. (a)–(i) The drawn layers that define the masks. The active mask is the union of the ndiff and pdiff drawn layers. The *n*-diffusion implant and *p*-diffusion implant masks are bloated versions of the ndiff and pdiff drawn layers. (j) The complete cell layout. (k) The phantom cell layout. Often an ASIC vendor hides the details of the internal cell construction. The phantom cell is used for layout by the customer and then "instantiated" by the ASIC vendor after layout is complete. This layout uses gray-scale stipple patterns to distinguish between layers.

(CSN). A second implant (using boron ions) forms the p-type source–drain for the p-channel transistors and p-type substrate contacts (CSP). These implants are masked as follows

$$\text{CSN (mask)} = \text{grow (ndiff (drawn))}, \tag{2.19}$$

$$\text{CSP (mask)} = \text{grow (pdiff (drawn))}, \tag{2.20}$$

where "grow" means that we expand or **bloat** the drawn ndiff and drawn pdiff layers slightly (usually by a few λ).

During implantation the **dopant** ions are blocked by the resist pattern defined by the CSN and CSP masks. The CSN mask thus prevents the n-type regions being implanted with p-type dopants (and vice versa for the CSP mask). As we shall see, the CSN and CSP masks are not intended to define the edges of the n-type and p-type regions. Instead these two masks function more like newspaper that prevents paint from spraying everywhere. The dopant ions are also blocked from reaching the silicon surface by the poly gates and this aligns the edge of the source and drain regions to the edges of the gates (we call this a **self-aligned process**). In addition, the implants are blocked by the FOX and this defines the outside edges of the source, drain, and substrate contact regions.

The only areas of the silicon surface that are doped n-type are

$$n\text{-diffusion (silicon)} = (\text{CAA (mask)} \wedge \text{CSN (mask)}) \wedge (\neg \text{CPG (mask)}); \tag{2.21}$$

where the $\wedge$ symbol represents AND (the intersection of two layers); and the $\neg$ symbol represents NOT.

Similarly, the only regions that are doped p-type are

$$p\text{-diffusion (silicon)} = (\text{CAA (mask)} \wedge \text{CSP (mask)}) \wedge (\neg \text{CPG (mask)}). \tag{2.22}$$

If the CSN and CSP masks do not overlap, it is possible to save a mask by using one implant mask (CSN or CSP) for the other type (CSP or CSN). We can do this by using a **positive resist** (the pattern of resist remaining after developing is the same as the dark areas on the mask) for one implant step and a **negative resist** (vice versa) for the other step. However, because of the poor resolution of negative resist and because of difficulties in generating the implant masks automatically from the drawn diffusions (especially when opposite diffusion types are drawn close to each other or touching), it is now common to draw both implant masks as well as the two diffusion layers.

It is important to remember that, even though poly is above diffusion, the polysilicon is deposited first and acts like masking tape. It is rather like airbrushing a stripe—you use masking tape and spray everywhere without worrying about making straight lines. The edges of the pattern will align to the edge of the tape. Here the analogy ends because the poly is left in place. Thus,

$$n\text{-diffusion (silicon)} = (\text{ndiff (drawn)}) \wedge (\neg \text{poly (drawn)}) \quad \text{and} \tag{2.23}$$

$$p\text{-diffusion (silicon)} = (\text{pdiff (drawn)}) \wedge (\neg \text{poly (drawn)}). \tag{2.24}$$

In the ASIC industry the names nplus, $n+$, and n-diffusion (as well as the p-type equivalents) are used in various ways. These names may refer to either the drawn diffusion layer (that we call ndiff), the mask (CSN), or the doped region on the silicon (the intersection of the active and implant mask that we call n-diffusion)—very confusing.

The source and drain are often formed from two separate implants. The first is a light implant close to the edge of the gate, the second a heavier implant that forms the rest of the source or drain region. The separate diffusions reduce the electric field near the drain end of the channel. Tailoring the device characteristics in this fashion is known as **drain engineering** and a process including these steps is referred to as an **LDD process**, for **lightly doped drain**; the first light implant is known as an **LDD diffusion** or LDD implant.

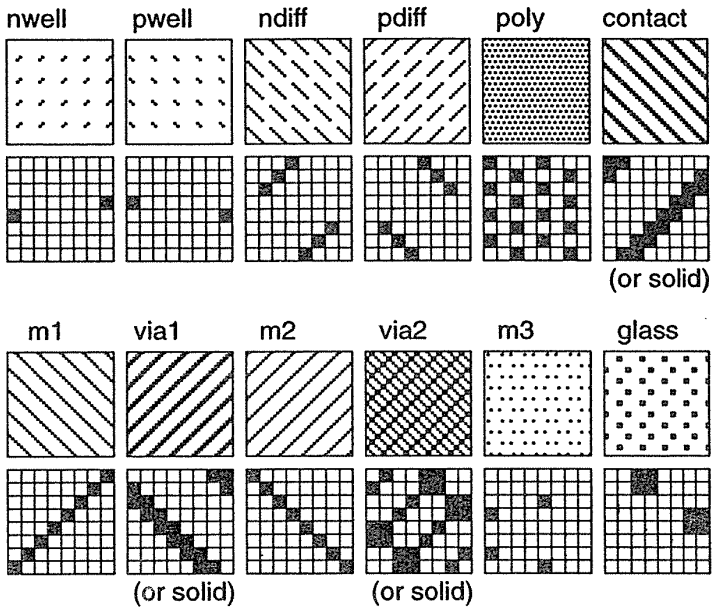


FIGURE 2.8 Drawn layers and an example set of black-and-white stipple patterns for a CMOS process. On top are the patterns as they appear in layout. Underneath are the magnified 8-by-8 pixel patterns. If we are trying to simplify layout we may use solid black or white for contact and vias. If we have contacts and vias placed on top of one another we may use stipple patterns or other means to help distinguish between them. Each stipple pattern is transparent, so that black shows through from underneath when layers are superimposed. There are no standards for these patterns.

Figure 2.8 shows a **stipple-pattern** matrix for a CMOS process. When we draw layout you can see through the layers—all the stipple patterns are OR'ed together. Figure 2.9 shows the transistor layers as they appear in layout (drawn using the patterns from Figure 2.8) and as they appear on the silicon. Figure 2.10 shows the same thing for the interconnect layers.

2.2.1 Sheet Resistance

Tables 2.3 and 2.4 show the sheet resistance for each conducting layer (in decreasing order of resistance) for two different generations of CMOS process.

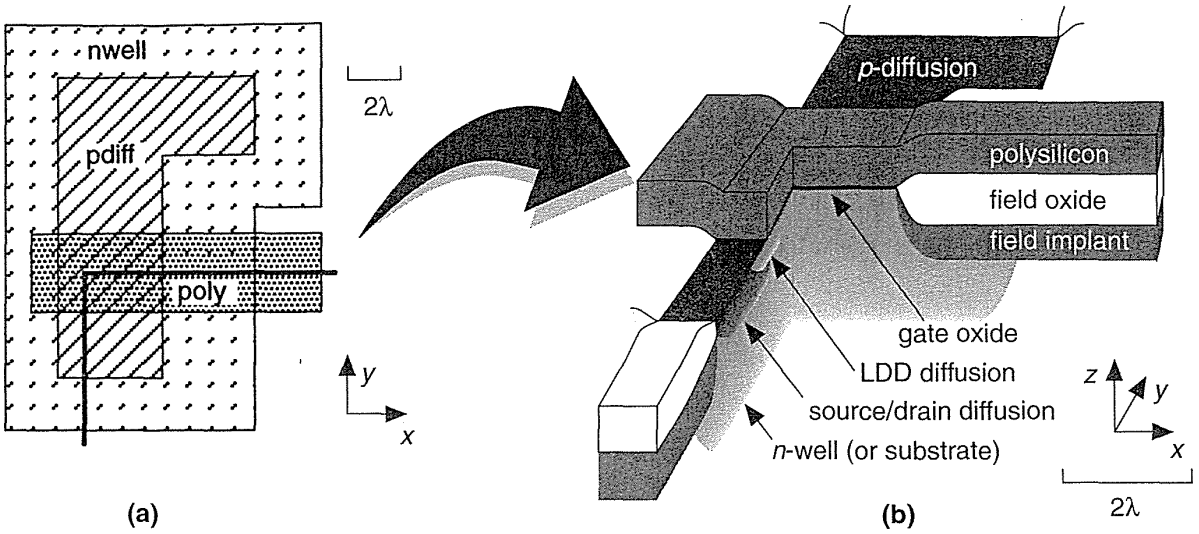
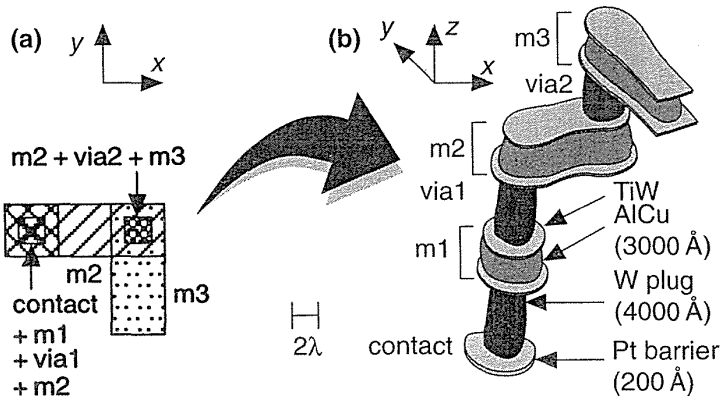


FIGURE 2.9 The transistor layers. (a) A *p*-channel transistor as drawn in layout. (b) The corresponding silicon cross section (the heavy lines in part a show the cuts). This is how a *p*-channel transistor would look just after completing the source and drain implant steps.

FIGURE 2.10 The interconnect layers. (a) Metal layers as drawn in layout. (b) The corresponding structure (as it might appear in a scanning-electron micrograph). The insulating layers between the metal layers are not shown. Contact is made to the underlying silicon through a platinum barrier layer. Each via consists of a tungsten plug. Each metal layer consists of a titanium–tungsten and aluminum–copper sandwich. Most deep submicron CMOS processes use metal structures similar to this. The scale, rounding, and irregularity of the features are realistic.



The **diffusion** layers, *n*-diffusion and *p*-diffusion, both have a high resistivity—typically from 1–100 Ω/square. We measure resistance in **Ω/square** (ohms per square) because for a fixed thickness of material it does not matter what the size of a square is—the resistance is the same. Thus the resistance of a rectangular shape of a sheet of material may be calculated from the number of squares it contains times the **sheet resistance** in Ω/square. We can use diffusion for very short connections inside a logic cell, but not for interconnect between logic cells. Poly has the next highest

TABLE 2.3 Sheet resistance (1 μm CMOS).

Layer	Sheet resistance	Units
n-well	1.15 ± 0.25	kΩ/square
poly	3.5 ± 2.0	Ω/square
n-diffusion	75 ± 20	Ω/square
p-diffusion	140 ± 40	Ω/square
m1/2	70± 6	mΩ/square
m3	30± 3	mΩ/square

TABLE 2.4 Sheet resistance (0.35 μm CMOS).

Layer	Sheet resistance	Units
n-well	1 ± 0.4	kΩ/square
poly	10 ± 4.0	Ω/square
n-diffusion	3.5 ± 2.0	Ω/square
p-diffusion	2.5 ± 1.5	Ω/square
m1/2/3	60 ± 6	mΩ/square
metal4	30 ± 3	mΩ/square

resistance to diffusion. Most submicron CMOS processes use a **silicide** material (a metallic compound of silicon) that has much lower resistivity (at several Ω/square) than the poly or diffusion layers alone. Examples are tantalum silicide, TaSi; tungsten silicide, WSi; or titanium silicide, TiSi. The **stoichiometry** of these deposited silicides varies. For example, for tungsten silicide W:Si ≈ 1:2.6.

There are two types of silicide process. In a silicide process only the gate is silicided. This reduces the poly sheet resistance, but not that of the source–drain. In a self-aligned silicide (**salicide**) process, both the gate and the source–drain regions are silicided. In some processes silicide can be used to connect adjacent poly and diffusion (we call this feature **LI**, white metal, local interconnect, metal0, or m0). LI is useful to reduce the area of ASIC RAM cells, for example.

Interconnect uses metal layers with resistivities of tens of mΩ/square, several orders of magnitude less than the other layers. There are usually several layers of metal in a CMOS ASIC process, each separated by an insulating layer. The metal layer above the poly gate layer is the first-level metal (**m1** or metal1), the next is the second-level metal (**m2** or metal2), and so on. We can make connections from m1 to diffusion using **diffusion contacts** or to the poly using **polysilicon contacts**.

After we etch the contact holes a thin **barrier metal** (typically platinum) is deposited over the silicon and poly. Next we form **contact plugs** (**via plugs** for connections between metal layers) to reduce contact resistance and the likelihood of breaks in the contacts. Tungsten is commonly used for these plugs. Following this we form the metal layers as sandwiches. The middle of the sandwich is a layer (usually from 3000 Å to 10,000 Å) of aluminum and copper. The top and bottom layers are normally titanium–tungsten (TiW, pronounced "tie-tungsten"). Submicron processes use **chemical–mechanical polishing** (**CMP**) to smooth the wafers flat before each metal deposition step to help with step coverage.

An insulating glass, often sputtered quartz (SiO_2), though other materials are also used, is deposited between metal layers to help create a smooth surface for the deposition of the metal. Design rules may refer to this insulator as an **intermetal**

oxide (IMO) whether they are in fact oxides or not, or **interlevel dielectric (ILD)**. The IMO may be a spin-on polymer; boron-doped phosphosilicate glass (BPSG); Si_3N_4; or sandwiches of these materials (oxynitrides, for example).

We make the connections between m1 and m2 using **metal vias**, **cuts**, or just **vias**. We cannot connect m2 directly to diffusion or poly; instead we must make these connections through m1 using a via. Most processes allow contacts and vias to be placed directly above each other without restriction, arrangements known as **stacked vias** and **stacked contacts**. We call a process with m1 and m2 a **two-level metal (2LM)** technology. A **3LM** process includes a third-level metal layer (**m3** or metal3), and some processes include more metal layers. In this case a connection between m1 and m2 will use an m1/m2 via, or **via1**; a connection between m2 and m3 will use an m2/m3 via, or **via2**, and so on.

The minimum spacing of interconnects, the **metal pitch**, may increase with successive metal layers. The minimum metal pitch is the minimum spacing between the centers of adjacent interconnects and is equal to the minimum metal width plus the minimum metal spacing.

Aluminum interconnect tends to break when carrying a high current density. Collisions between high-energy electrons and atoms move the metal atoms over a long period of time in a process known as **electromigration**. Copper is added to the aluminum to help reduce the problem. The other solution is to reduce the current density by using wider than minimum-width metal lines.

Tables 2.5 and 2.6 show maximum specified **contact resistance** and **via resistance** for two generations of CMOS processes. Notice that a m1 contact in either process is equal in resistance to several hundred squares of metal.

TABLE 2.5 Contact resistance (1 μm CMOS).

Contact/via type	Resistance (maximum)
m2/m3 via (via2)	5 Ω
m1/m2 via (via1)	2 Ω
m1/*p*-diffusion contact	20 Ω
m1/*n*-diffusion contact	20 Ω
m1/poly contact	20 Ω

TABLE 2.6 Contact resistance (0.35 μm CMOS).

Contact/via type	Resistance (maximum)
m2/m3 via (via2)	6 Ω
m1/m2 via (via1)	6 Ω
m1/*p*-diffusion contact	20 Ω
m1/*n*-diffusion contact	20 Ω
m1/poly contact	20 Ω

2.3 CMOS Design Rules

Figure 2.11 defines the **design rules** for a CMOS process using pictures. Arrows between objects denote a minimum spacing, and arrows showing the size of an

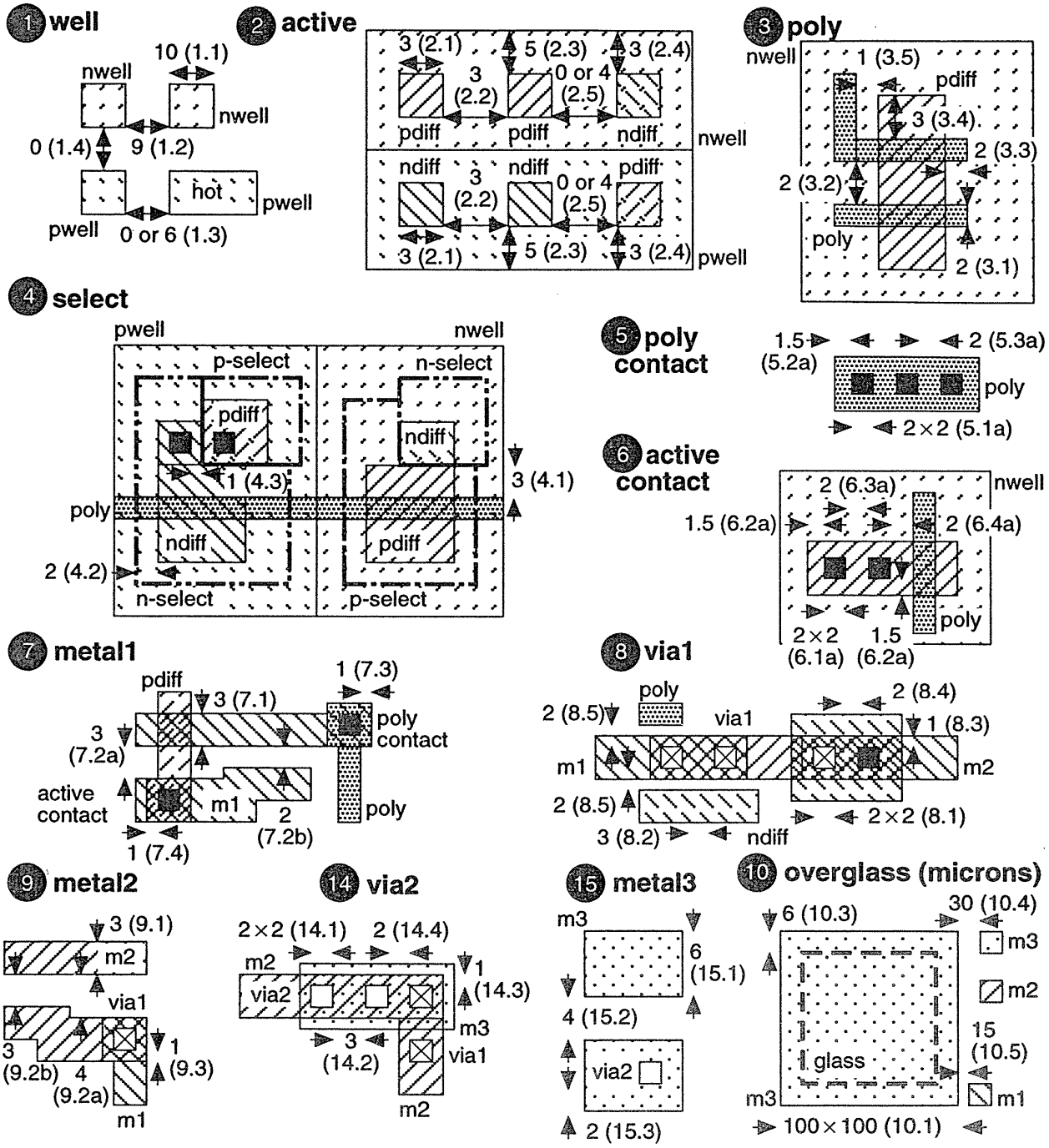


FIGURE 2.11 The MOSIS scalable CMOS design rules (rev. 7). Dimensions are in λ. Rule numbers are in parentheses (missing rule sets 11–13 are extensions to this basic process).

object denote a minimum width. Rule 3.1, for example, is the minimum width of poly (2 λ). Each of the rule numbers may have different values for different manufacturers—there are no standards for design rules. Tables 2.7–2.9 show the MOSIS scalable CMOS rules. Table 2.7 shows the layer rules for the **process front end**, which is the front end of the line (as in production line) or **FEOL**. Table 2.8 shows the rules for the process back end (**BEOL**), the metal interconnect, and Table 2.9 shows the rules for the pad layer and glass layer.

The rules in Table 2.7 and Table 2.8 are given as multiples of λ. If we use **lambda-based rules** we can move between successive process generations just by changing the value of λ. For example, we can scale 0.5 μm layouts (λ = 0.25 μm) by a factor of 0.175/0.25 for a 0.35 μm process (λ = 0.175 μm)—at least in theory. You may get an inkling of the practical problems from the fact that the values for pad dimensions and spacing in Table 2.9 are given in microns and not in λ. This is because bonding to the pads is an operation that does not scale well. Often companies have two sets of design rules: one in λ (with fractional λ rules) and the other in microns. Ideally we would like to express all of the design rules in integer multiples of λ. This was true for revisions 4–6, but not revision 7 of the MOSIS rules. In revision 7 rules 5.2a/6.2a are noninteger. The original Mead–Conway NMOS rules include a noninteger 1.5 λ rule for the implant layer.

2.4 Combinational Logic Cells

The AND-OR-INVERT (AOI) and the OR-AND-INVERT (OAI) logic cells are particularly efficient in CMOS. Figure 2.12 shows an AOI221 and an OAI321 logic cell (the logic symbols in Figure 2.12 are not standards, but are widely used). All indices (the indices are the numbers after AOI or OAI) in the logic cell name greater than 1 correspond to the inputs to the first "level" or stage—the AND gate(s) in an AOI cell, for example. An index of '1' corresponds to a direct input to the second-stage cell. We write indices in descending order; so it is AOI221 and not AOI122 (but both are equivalent cells), and AOI32 not AOI23. If we have more than one direct input to the second stage we repeat the '1'; thus an AOI211 cell performs the function Z = (A.B + C + D)'. A three-input NAND cell is an OAI111, but calling it that would be very confusing. These rules are not standard, but form a convention that we shall adopt and one that is widely used in the ASIC industry.

There are many ways to represent the logical operator, AND. I shall use the **middle dot** and write A · B (rather than AB, A.B, or A ∧ B); occasionally I may use AND(A, B). Similarly I shall write A + B as well as OR(A, B). I shall use an apostrophe like this, A', to denote the complement of A rather than $\overline{A}$ since sometimes it is difficult or inappropriate to use an overbar (*vinculum*) or diacritical mark (macron). It is possible to misinterpret AB' as $A\overline{B}$ rather than $\overline{AB}$ (but the former alternative would be A · B' in my convention). I shall be careful in these situations.

TABLE 2.7 MOSIS scalable CMOS rules version 7—the process front end.

Layer	Rule	Explanation	Value/λ
well (CWN, CWP)	1.1	minimum width	10
	1.2	minimum space (different potential, a hot well)	9
	1.3	minimum space (same potential)	0 or 6
	1.4	minimum space (different well type)	0
active (CAA)	2.1/2.2	minimum width/space	3
	2.3	source/drain active to well edge space	5
	2.4	substrate/well contact active to well edge space	3
	2.5	minimum space between active (different implant type)	0 or 4
poly (CPG)	3.1/3.2	minimum width/space	2
	3.3	minimum gate extension of active	2
	3.4	minimum active extension of poly	3
	3.5	minimum field poly to active space	1
select (CSN, CSP)	4.1	minimum select spacing to channel of transistor[1]	3
	4.2	minimum select overlap of active	2
	4.3	minimum select overlap of contact	1
	4.4	minimum select width and spacing [2]	2
poly contact (CCP)	5.1.a	exact contact size	2×2
	5.2.a	minimum poly overlap	1.5
	5.3.a	minimum contact spacing	2
active contact (CCA)	6.1.a	exact contact size	2×2
	6.2.a	minimum active overlap	1.5
	6.3.a	minimum contact spacing	2
	6.4.a	minimum space to gate of transistor	2

[1]To ensure source and drain width.
[2]Different select types may touch but not overlap.

TABLE 2.8 MOSIS scalable CMOS rules version 7—the process back end.

Layer	Rule	Explanation	Value/λ
metal1 (CMF)	7.1	minimum width	3
	7.2.a	minimum space	3
	7.2.b	minimum space (for minimum-width wires only)	2
	7.3	minimum overlap of poly contact	1
	7.4	minimum overlap of active contact	1
via1 (CVA)	8.1	exact size	2×2
	8.2	minimum via spacing	3
	8.3	minimum overlap by metal1	1
	8.4	minimum spacing to contact	2
	8.5	minimum spacing to poly or active edge	2
metal2 (CMS)	9.1	minimum width	3
	9.2.a	minimum space	4
	9.2.b	minimum space (for minimum-width wires only)	3
	9.3	minimum overlap of via1	1
via2 (CVS)	14.1	exact size	2×2
	14.2	minimum space	3
	14.3	minimum overlap by metal2	1
	14.4	minimum spacing to via1	2
metal3 (CMT)	15.1	minimum width	6
	15.2	minimum space	4
	15.3	minimum overlap of via2	2

TABLE 2.9 MOSIS scalable CMOS rules version 7—the pads and overglass (passivation).

Layer	Rule	Explanation	Value
glass (COG)	10.1	minimum bonding-pad width	100 μm×100 μm
	10.2	minimum probe-pad width	75 μm×75 μm
	10.3	pad overlap of glass opening	6 μm
	10.4	minimum pad spacing to unrelated metal2 (or metal3)	30 μm
	10.5	minimum pad spacing to unrelated metal1, poly, or active	15 μm

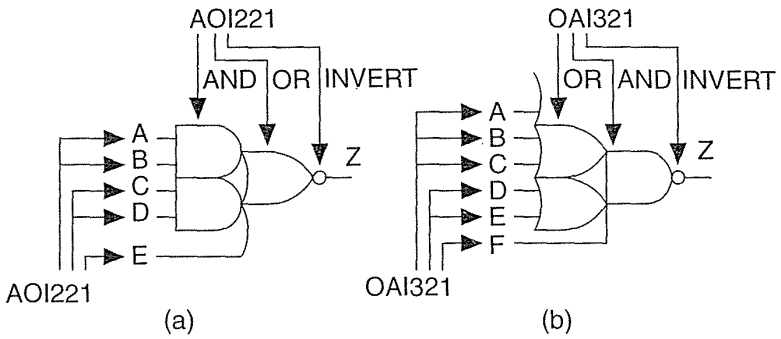


FIGURE 2.12 Naming and numbering complex CMOS combinational cells. (a) An AND-OR-INVERT cell, an AOI221. (b) An OR-AND-INVERT cell, an OAI321. Numbering is always in descending order.

We can express the function of the AOI221 cell in Figure 2.12(a) as

$$Z = (A \cdot B + C \cdot D + E)'. \tag{2.25}$$

We can also write this equation unambiguously as Z = OAI221(A, B, C, D, E), just as we might write X = NAND(I, J, K) to describe the logic function $X = (I \cdot J \cdot K)'$.

This notation is useful because, for example, if we write OAI321(P, Q, R, S, T, U) we immediately know that U (the sixth input) is the (only) direct input connected to the second stage. Sometimes we need to refer to particular inputs without listing them all. We can adopt another convention that letters of the input names change with the index position. Now we can refer to input B2 of an AOI321 cell, for example, and know which input we are talking about without writing

$$Z = \text{AOI321(A1, A2, A3, B1, B2, C)}. \tag{2.26}$$

Table 2.10 shows the **AOI family** of logic cells with three indices (with branches in the family for AOI, OAI, AO, and OA cells). There are 5 types and 14 separate members of each branch of this family. There are thus $4 \times 14 = 56$ cells of the type X*abc* where X = {OAI, AOI, OA, AO} and each of the indexes *a*, *b*, and *c* can range from 1 to 3. We form the AND-OR (AO) and OR-AND (OA) cells by adding an inverter to the output of an AOI or OAI cell.

2.4.1 Pushing Bubbles

The AOI and OAI logic cells can be built using a single stage in CMOS using series–parallel networks of transistors called **stacks**. Figure 2.13 illustrates the procedure to build the *n*-channel and *p*-channel stacks, using the AOI221 cell as an example.

Here are the steps to construct any single-stage combinational CMOS logic cell:

1. Draw a schematic icon with an inversion (bubble) on the last cell (the bubble-out schematic). Use **de Morgan's theorems**—"A NAND is an OR with inverted inputs and a NOR is an AND with inverted inputs"—to push the output bubble back to the inputs (this the dual icon or bubble-in schematic).

TABLE 2.10 The AOI family of cells with three index numbers or less.

Cell type[1]	Cells	Number of unique cells
Xa1	X21, X31	2
Xa11	X211, X311	2
Xab	X22, X33, X32	3
Xab1	X221, X331, X321	3
Xabc	X222, X333, X332, X322	4
Total		14

[1]Xabc: X={AOI, AO, OAI, OA}; a, b, c = {2, 3}; { } means "choose one."

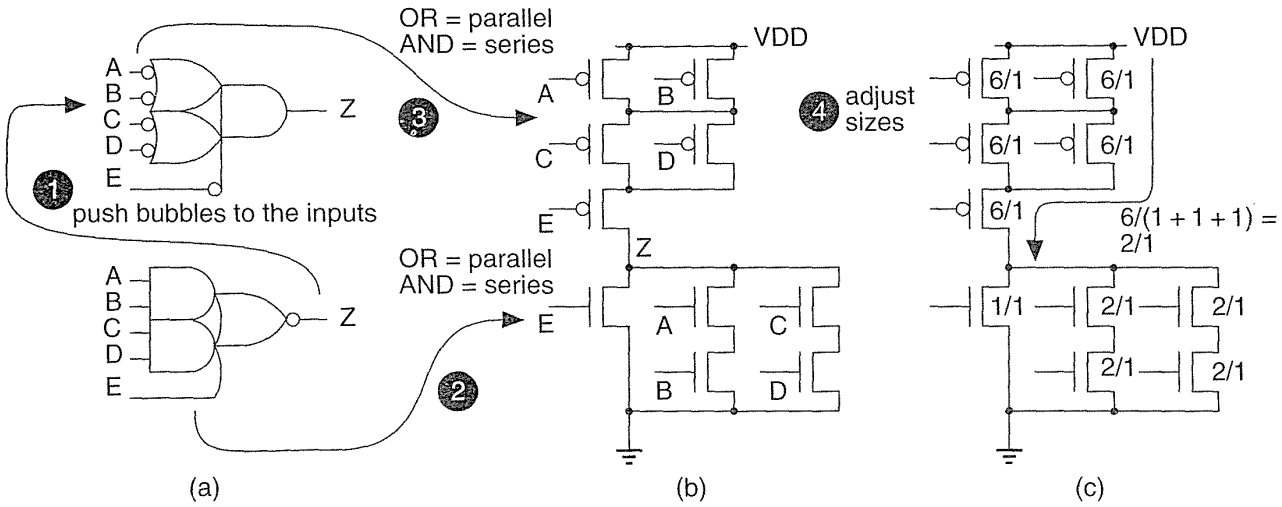


FIGURE 2.13 Constructing a CMOS logic cell—an AOI221. (a) First build the dual icon by using de Morgan's theorem to "push" inversion bubbles to the inputs. (b) Next build the *n*-channel and *p*-channel stacks from series and parallel combinations of transistors. (c) Adjust transistor sizes so that the *n*-channel and *p*-channel stacks have equal strengths.

2. Form the *n*-channel stack working from the inputs on the bubble-out schematic: OR translates to a parallel connection, AND translates to a series connection. If you have a bubble at an input, you need an inverter.
3. Form the *p*-channel stack using the bubble-in schematic (ignore the inversions at the inputs—the bubbles on the gate terminals of the *p*-channel transistors

take care of these). If you do not have a bubble at the input gate terminals, you need an inverter (these will be the same input gate terminals that had bubbles in the bubble-out schematic).

The two stacks are **network duals** (they can be derived from each other by swapping series connections for parallel, and parallel for series connections). The *n*-channel stack implements the strong '0's of the function and the *p*-channel stack provides the strong '1's. The final step is to adjust the drive strength of the logic cell by sizing the transistors.

2.4.2 Drive Strength

Normally we **ratio** the sizes of the *n*-channel and *p*-channel transistors in an inverter so that both types of transistors have the same resistance, or **drive strength**. That is, we make $\beta_n = \beta_p$. At low dopant concentrations and low electric fields μ_n is about twice μ_p. To compensate we make the shape factor, W/L, of the *p*-channel transistor in an inverter about twice that of the *n*-channel transistor (we say the logic has a ratio of 2). Since the transistor lengths are normally equal to the minimum poly width for both types of transistors, the ratio of the transistor widths is also equal to 2. With the high dopant concentrations and high electric fields in submicron transistors the difference in mobilities is less—typically between 1 and 1.5.

Logic cells in a library have a range of drive strengths. We normally call the minimum-size inverter a 1X inverter. The drive strength of a logic cell is often used as a suffix; thus a 1X inverter has a cell name such as INVX1 or INVD1. An inverter with transistors that are twice the size will be an INVX2. Drive strengths are normally scaled in a geometric ratio, so we have 1X, 2X, 4X, and (sometimes) 8X or even higher, drive-strength cells. We can size a logic cell using these basic rules:

- Any string of transistors connected between a power supply and the output in a cell with 1X drive should have the same resistance as the *n*-channel transistor in a 1X inverter.
- A transistor with shape factor W_1/L_1 has a resistance proportional to L_1/W_1 (so the larger W_1 is, the smaller the resistance).
- Two transistors in parallel with shape factors W_1/L_1 and W_2/L_2 are equivalent to a single transistor $(W_1/L_1 + W_2/L_2)/1$. For example, a 2/1 in parallel with a 3/1 is a 5/1.
- Two transistors, with shape factors W_1/L_2 and W_2/L_2, in series are equivalent to a single $1/(L_1/W_1 + L_2/W_2)$ transistor.

For example, a transistor with shape factor 3/1 (we shall call this "a 3/1") in series with another 3/1 is equivalent to a $1/((1/3) + (1/3))$ or a 3/2. We can use the following method to calculate equivalent transistor sizes:

- To add transistors in parallel, make all the lengths 1 and add the widths.
- To add transistors in series, make all the widths 1 and add the lengths.

We have to be careful to keep W and L reasonable. For example, a 3/1 in series with a 2/1 is equivalent to a 1/((1/3) + (1/2)) or 1/0.83. Since we cannot make a device 2 λ wide and 1.66 λ long, a 1/0.83 is more naturally written as 3/2.5. We like to keep both W and L as integer multiples of 0.5 (equivalent to making W and L integer multiples of λ), but W and L must be greater than 1.

In Figure 2.13(c) the transistors in the AOI221 cell are sized so that any string through the *p*-channel stack has a drive strength equivalent to a 2/1 *p*-channel transistor (we choose the worst case, if more than one transistor in parallel is conducting then the drive strength will be higher). The *n*-channel stack is sized so that it has a drive strength of a 1/1 *n*-channel transistor. The ratio in this library is thus 2.

If we were to use four drive strengths for each of the AOI family of cells shown in Table 2.10, we would have a total of 224 combinational library cells—just for the AOI family. The synthesis tools can handle this number of cells, but we may not be able to design this many cells in a reasonable amount of time. Section 3.3, "Logical Effort," will help us choose the most logically efficient cells.

2.4.3 Transmission Gates

Figure 2.14(a) and (b) shows a CMOS **transmission gate** (**TG**, TX gate, pass gate, coupler). We connect a *p*-channel transistor (to transmit a strong '1') in parallel with an *n*-channel transistor (to transmit a strong '0').

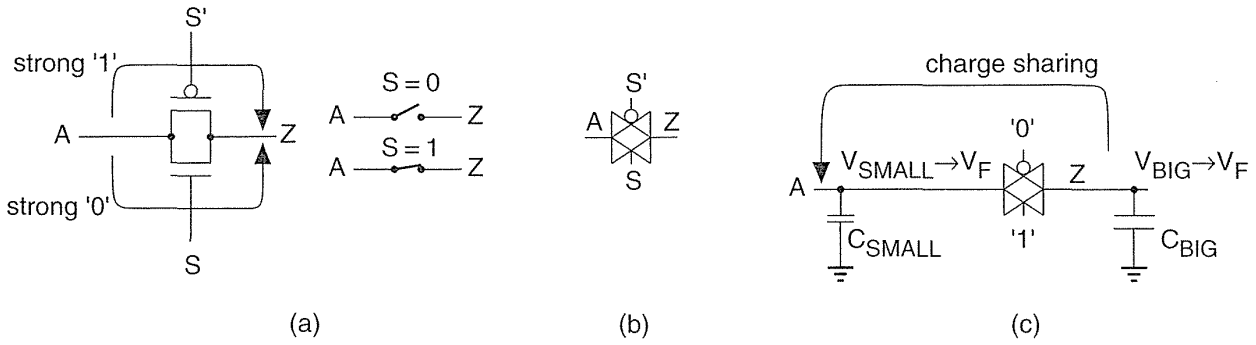


FIGURE 2.14 CMOS transmission gate (TG). (a) An *n*-channel and *p*-channel transistor in parallel form a TG. (b) A common symbol for a TG. (c) The charge-sharing problem.

We can express the function of a TG as

$$Z = TG(A, S), \tag{2.27}$$

but this is ambiguous—if we write TG(X, Y), how do we know if X is connected to the gates or sources/drains of the TG? We shall always define TG(X, Y) when we use it. It is tempting to write TG(A, S) = A · S, but what is the value of Z when S ='0' in Figure 2.14(a), since Z is then left floating? A TG is a switch, not an AND logic cell.

There is a potential problem if we use a TG as a switch connecting a node Z that has a large capacitance, C_{BIG}, to an input node A that has only a small capacitance C_{SMALL} (see Figure 2.14c). If the initial voltage at A is V_{SMALL} and the initial voltage at Z is V_{BIG}, when we close the TG (by setting S = '1') the final voltage on both nodes A and Z is

$$V_F = \frac{C_{\mathrm{BIG}} V_{\mathrm{BIG}} + C_{\mathrm{SMALL}} V_{\mathrm{SMALL}}}{C_{\mathrm{BIG}} + C_{\mathrm{SMALL}}}. \tag{2.28}$$

Imagine we want to drive a '1' onto node Z from node A. Suppose $C_{\mathrm{BIG}} = 0.2$ pF (about 10 standard loads in a 0.5 μm process) and $C_{\mathrm{SMALL}} = 0.02$ pF, $V_{\mathrm{BIG}} = 0$ V and $V_{\mathrm{SMALL}} = 5$ V; then

$$V_F = \frac{\left(0.2 \times 10^{-12}\right)0 + \left(0.02 \times 10^{-12}\right)5}{\left(0.2 \times 10^{-12}\right) + \left(0.02 \times 10^{-12}\right)} = 0.45 \text{ V}. \tag{2.29}$$

This is not what we want at all, the "big" capacitor has forced node A to a voltage close to a '0'. This type of problem is known as **charge sharing**. We should make sure that either (1) node A is strong enough to overcome the big capacitor, or (2) insulate node A from node Z by including a **buffer** (an inverter, for example) between node A and node Z. We must not use charge to drive another logic cell—only a logic cell can drive a logic cell.

If we omit one of the transistors in a TG (usually the *p*-channel transistor) we have a **pass transistor**. There is a branch of full-custom VLSI design that uses pass-transistor logic. Much of this is based on relay-based logic, since a single transistor switch looks like a relay contact. There are many problems associated with pass-transistor logic related to charge sharing, reduced noise margins, and the difficulty of predicting delays. Though pass transistors may appear in an ASIC cell inside a library, they are not used by ASIC designers.

We can use two TGs to form a **multiplexer** (or multiplex*or*—people use both orthographies) as shown in Figure 2.15(a). We often shorten multiplexer to **MUX**. The MUX function for two data inputs, A and B, with a select signal S, is

$$\mathrm{Z} = \mathrm{TG(A, S')} + \mathrm{TG(B, S)}. \tag{2.30}$$

We can write this as $\mathrm{Z} = \mathrm{A} \cdot \mathrm{S'} + \mathrm{B} \cdot \mathrm{S}$, since node Z is always connected to one or other of the inputs (and we assume both are driven). This is a two-input MUX (2-to-1 MUX or 2:1 MUX). Unfortunately, we can also write the MUX function as $\mathrm{Z} = \mathrm{A} \cdot \mathrm{S} + \mathrm{B} \cdot \mathrm{S'}$, so it is difficult to write the MUX function unambiguously as Z = MUX(X, Y, Z). For example, is the select input X, Y, or Z? We shall define the function MUX(X, Y, Z) each time we use it. We must also be careful to label a MUX if we use the symbol shown in Figure 2.15(b). Symbols for a MUX are shown in Figure 2.15(b–d). In the IEEE notation 'G' specifies an AND dependency. Thus, in Figure 2.15(c), G = '1' selects the input labeled '1'. Figure 2.15(d) uses the **common control block** symbol (the notched rectangle). Here, G1 = '1' selects the input '1',

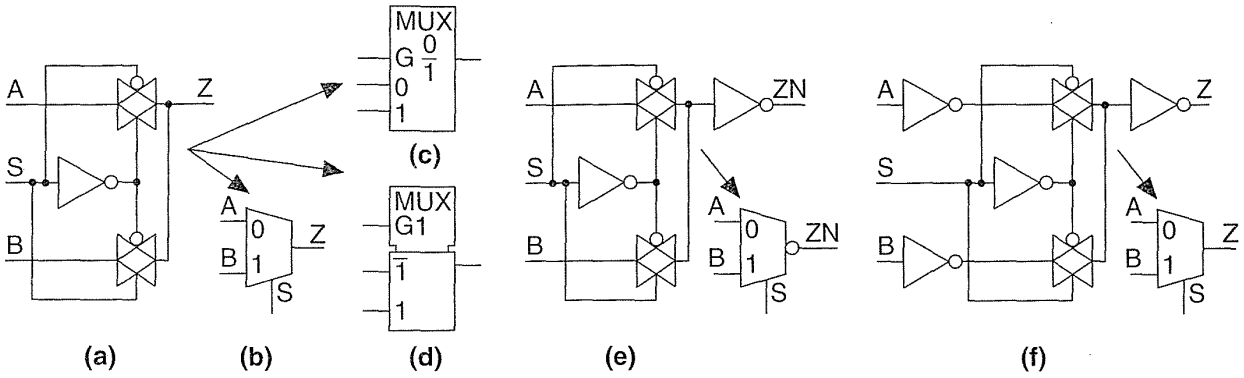


FIGURE 2.15 The CMOS multiplexer (MUX). (a) A noninverting 2:1 MUX using transmission gates without buffering. (b) A symbol for a MUX (note how the inputs are labeled). (c) An IEEE standard symbol for a MUX. (d) A nonstandard, but very common, IEEE symbol for a MUX. (e) An inverting MUX with output buffer. (f) A noninverting buffered MUX.

and G1 = '0' selects the input '$\bar{1}$'. Strictly this form of IEEE symbol should be used only for elements with more than one section controlled by common signals, but the symbol of Figure 2.15(d) is used often for a 2:1 MUX.

The MUX shown in Figure 2.15(a) works, but there is a potential charge-sharing problem if we cascade MUXes (connect them in series). Instead most ASIC libraries use MUX cells built with a more conservative approach. We could buffer the output using an inverter (Figure 2.15e), but then the MUX becomes inverting. To build a safe, noninverting MUX we can buffer the inputs and output (Figure 2.15f)—requiring 12 transistors, or 3 gate equivalents (only the gate equivalent counts are shown from now on).

Figure 2.16 shows how to use an OAI22 logic cell (and an inverter) to implement an inverting MUX. The implementation in equation form (2.5 gates) is

$$\begin{aligned} ZN &= A' \cdot S' + B' \cdot S = [(A' \cdot S')' \cdot (B' \cdot S)']' = [\,(A + S) \cdot (B + S')\,]' \\ &= \text{OAI22}[A, S, B, \text{NOT}(S)]. \end{aligned} \tag{2.31}$$

(both A' and NOT(A) represent an inverter, depending on which representation is most convenient—they are equivalent). I often use an equation to describe a cell implementation.

The following factors will determine which MUX implementation is best:

1. Do we want to minimize the delay between the select input and the output or between the data inputs and the output?
2. Do we want an inverting or noninverting MUX?

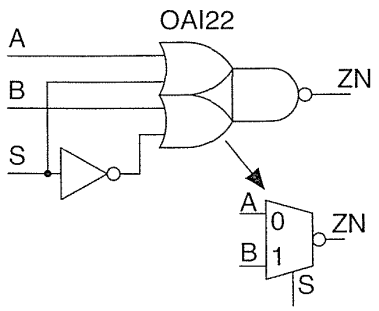


FIGURE 2.16 An inverting 2:1 MUX based on an OAI22 cell.

3. Do we object to having any logic cell inputs tied directly to the source/drain diffusions of a transmission gate? (Some companies forbid such **transmission-gate inputs**—since some simulation tools cannot handle them.)
4. Do we object to any logic cell outputs being tied to the source/drain of a transmission gate? (Some companies will not allow this because of the dangers of charge sharing.)
5. What drive strength do we require (and is size or speed more important)?

A minimum-size TG is a little slower than a minimum-size inverter, so there is not much difference between the implementations shown in Figure 2.15 and Figure 2.16, but the difference can become important for 4:1 and larger MUXes.

2.4.4 Exclusive-OR Cell

The two-input **exclusive-OR** (**XOR**, EXOR, not-equivalence, ring-OR) function is

$$A1 \oplus A2 = XOR(A1, A2) = A1 \cdot A2' + A1' \cdot A2. \tag{2.32}$$

We are now using multiletter symbols, but there should be no doubt that A1' means anything other than NOT(A1). We can implement a two-input XOR using a MUX and an inverter as follows (2 gates):

$$XOR(A1, A2) = MUX[NOT(A1), A1, A2], \tag{2.33}$$

where

$$MUX(A, B, S) = A \cdot S + B \cdot S'. \tag{2.34}$$

This implementation only buffers one input and does not buffer the MUX output. We can use inverter buffers (3.5 gates total) or an inverting MUX so that the XOR cell does not have any external connections to source/drain diffusions as follows (3 gates total):

$$XOR(A1, A2) = NOT[MUX(NOT[NOT(A1)], NOT(A1), A2)]. \tag{2.35}$$

We can also implement a two-input XOR using an AOI21 (and a NOR cell), since

$$\begin{aligned} \text{XOR}(A1, A2) &= A1 \cdot A2' + A1' \cdot A2 = [\,(A1 \cdot A2) + (A1 + A2)'\,]' \\ &= \text{AOI21}[A1, A2, \text{NOR}(A1, A2)], \end{aligned} \tag{2.36}$$

(2.5 gates). Similarly we can implement an **exclusive-NOR** (XNOR, equivalence) logic cell using an inverting MUX (and two inverters, total 3.5 gates) or an OAI21 logic cell (and a NAND cell, total 2.5 gates) as follows (using the MUX function of Eq. 2.34):

$$\begin{aligned} \text{XNOR}(A1, A2) &= A1 \cdot A2 + \text{NOT}(A1) \cdot \text{NOT}(A2) \\ &= \text{NOT}[\text{NOT}[\text{MUX}(A1, \text{NOT}\ (A1), A2]] \\ &= \text{OAI21}[A1, A2, \text{NAND}(A1, A2)] \end{aligned} \tag{2.37}$$

2.5 Sequential Logic Cells

There are two main approaches to clocking in VLSI design: **multiphase clocks** or a single clock and **synchronous design**. The second approach has the following key advantages: (1) it allows automated design, (2) it is safe, and (3) it permits vendor signoff (a guarantee that the ASIC will work as simulated). These advantages of synchronous design (especially the last one) usually outweigh every other consideration in the choice of a clocking scheme. The vast majority of ASICs use a rigid synchronous design style.

2.5.1 Latch

Figure 2.17(a) shows a sequential logic cell—a **latch**. The internal clock signals, CLKN (N for negative) and CLKP (P for positive), are generated from the system clock, CLK, by two inverters (I4 and I5) that are part of every latch cell—it is usually too dangerous to have these signals supplied externally, even though it would save space.

To emphasize the difference between a latch and flip-flop, sometimes people refer to the clock input of a latch as an **enable**. This makes sense when we look at Figure 2.17(b), which shows the operation of a latch. When the clock input is high, the latch is **transparent**—changes at the D input appear at the output Q (quite different from a flip-flop as we shall see). When the enable (clock) goes low (Figure 2.17c), inverters I2 and I3 are connected together, forming a storage loop that holds the last value on D until the enable goes high again. The storage loop will hold its state as long as power is on; we call this a **static** latch. A **sequential logic cell** is different from a combinational cell because it has this feature of storage or memory.

Notice that the output Q is unbuffered and connected directly to the output of I2 (and the input of I3), which is a storage node. In an ASIC library we are conservative and add an inverter to buffer the output, isolate the sensitive storage node, and

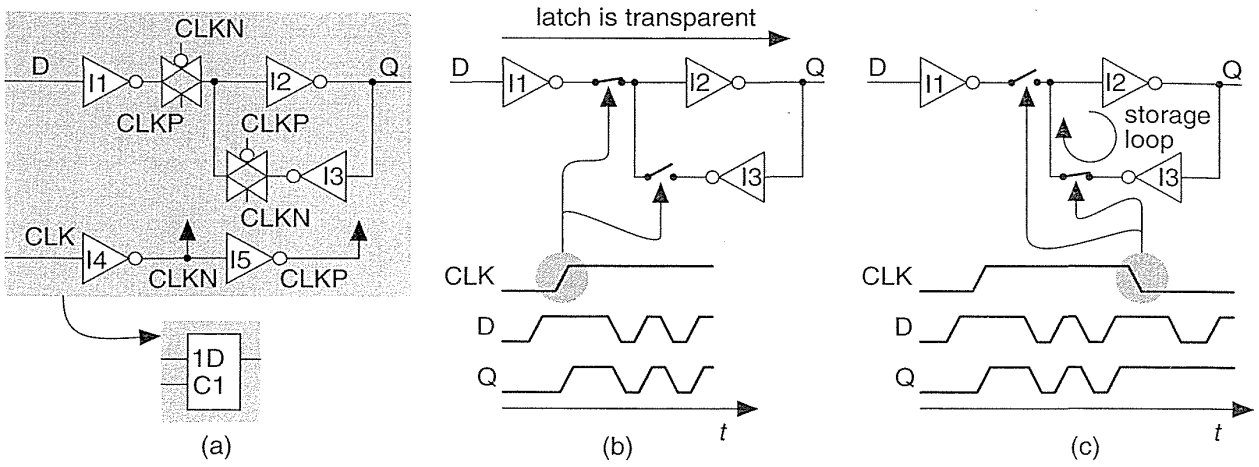


FIGURE 2.17 CMOS latch. (a) A positive-enable latch using transmission gates without output buffering, the enable (clock) signal is buffered inside the latch. (b) A positive-enable latch is transparent while the enable is high. (c) The latch stores the last value at D when the enable goes low.

thus invert the sense of Q. If we want both Q and QN we have to add two inverters to the circuit of Figure 2.17(a). This means that a latch requires seven inverters and two TGs (4.5 gates).

The latch of Figure 2.17(a) is a positive-enable D latch, active-high D latch, or transparent-high D latch (sometimes people also call this a D-type latch). A negative-enable (active-low) D latch can be built by inverting all the clock polarities in Figure 2.17(a) (swap CLKN for CLKP and vice-versa).

2.5.2 Flip-Flop

Figure 2.18(a) shows a **flip-flop** constructed from two D latches: a **master latch** (the first one) and a **slave latch**. This flip-flop contains a total of nine inverters and four TGs, or 6.5 gates. In this flip-flop design the storage node S is buffered and the clock-to-Q delay will be one inverter delay less than the clock-to-QN delay.

In Figure 2.18(b) the clock input is high, the master latch is transparent, and node M (for master) will follow the D input. Meanwhile the slave latch is disconnected from the master latch and is storing whatever the previous value of Q was. As the clock goes low (the negative edge) the slave latch is enabled and will update its state (and the output Q) to the value of node M at the negative edge of the clock. The slave latch will then keep this value of M at the output Q, despite any changes at the D input while the clock is low (Figure 2.18c). When the clock goes high again, the slave latch will store the captured value of M (and we are back where we started our explanation).

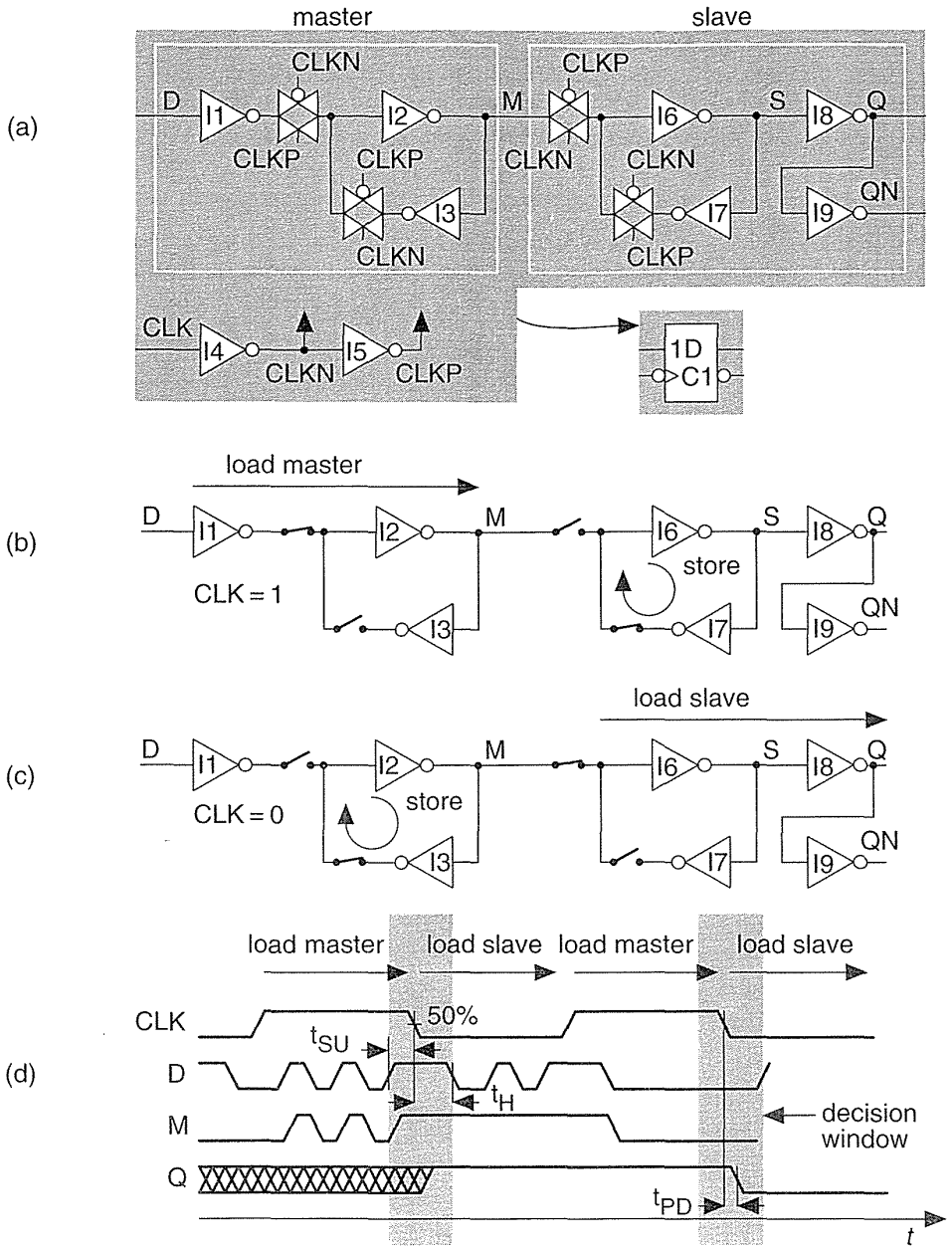


FIGURE 2.18 CMOS flip-flop. (a) This negative-edge–triggered flip-flop consists of two latches: master and slave. (b) While the clock is high, the master latch is loaded. (c) As the clock goes low, the slave latch loads the value of the master latch. (d) Waveforms illustrating the definition of the flip-flop setup time t_{SU}, hold time t_H, and propagation delay from clock to Q, t_{PD}.

The combination of the master and slave latches acts to capture or sample the D input at the negative clock edge, the **active clock edge**. This type of flip-flop is a **negative-edge–triggered flip-flop** and its behavior is quite different from a latch. The behavior is shown on the IEEE symbol by using a triangular "notch" to denote an edge-sensitive input. A bubble shows the input is sensitive to the negative edge. To build a positive-edge–triggered flip-flop we invert the polarity of all the clocks—as we did for a latch.

The waveforms in Figure 2.18(d) show the operation of the flip-flop as we have described it, and illustrate the definition of **setup time** (t_{SU}), **hold time** (t_H), and clock-to-Q propagation delay (t_{PD}). We must keep the data stable (a fixed logic '1' or '0') for a time t_{SU} prior to the active clock edge, and stable for a time t_H after the active clock edge (during the decision window shown).

In Figure 2.18(d) times are measured from the points at which the waveforms cross 50 percent of V_{DD}. We say the **trip point** is 50 percent or 0.5. Common choices are 0.5 or 0.65/0.35 (a signal has to reach $0.65V_{DD}$ to be a '1', and reach $0.35V_{DD}$ to be a '0'), or 0.1/0.9 (there is no standard way to write a trip point). Some vendors use different trip points for the input and output waveforms (especially in I/O cells).

The flip-flop in Figure 2.18(a) is a D flip-flop and is by far the most widely used type of flip-flop in ASIC design. There are other types of flip-flops—J-K, T (toggle), and S-R flip-flops—that are provided in some ASIC cell libraries mainly for compatibility with TTL design. Some people use the term **register** to mean an array (more than one) of flip-flops or latches (on a data bus, for example), but some people use register to mean a single flip-flop or a latch. This is confusing since flip-flops and latches are quite different in their behavior. When I am talking about logic cells, I use the term *register* to mean more than one flip-flop.

To add an **asynchronous set** (Q to '1') or **asynchronous reset** (Q to '0') to the flip-flop of Figure 2.18(a), we replace one inverter in both the master and slave latches with two-input NAND cells. Thus, for an active-low set, we replace I2 and I7 with two-input NAND cells, and, for an active-low reset, we replace I3 and I6. For both set and reset we replace all four inverters: I2, I3, I6, and I7. Some TTL flip-flops have **dominant reset** or **dominant set**, but this is difficult (and dangerous) to do in ASIC design. An input that forces Q to '1' is sometimes also called **preset**. The IEEE logic symbols use 'P' to denote an input with a presetting action. An input that forces Q to '0' is often also called **clear**. The IEEE symbols use 'R' to denote an input with a resetting action.

2.5.3 Clocked Inverter

Figure 2.19 shows how we can derive the structure of a **clocked inverter** from the series combination of an inverter and a TG. The arrows in Figure 2.19(b) represent the flow of current when the inverter is charging (I_R) or discharging (I_F) a load capacitance through the TG. We can break the connection between the inverter cells and use the circuit of Figure 2.19(c) without substantially affecting the operation of

the circuit. The symbol for the clocked inverter shown in Figure 2.19(d) is common, but by no means a standard.

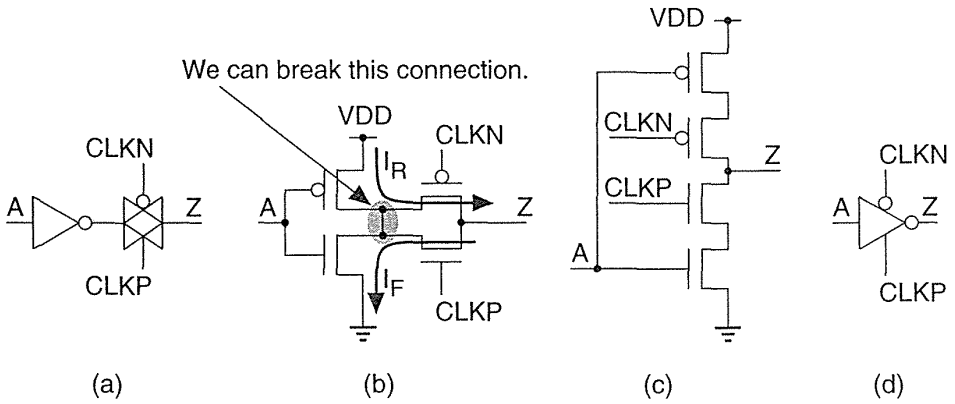


FIGURE 2.19 Clocked inverter. (a) An inverter plus transmission gate (TG). (b) The current flow in the inverter and TG allows us to break the connection between the transistors in the inverter. (c) Breaking the connection forms a clocked inverter. (d) A common symbol.

We can use the clocked inverter to replace the inverter–TG pairs in latches and flip-flops. For example, we can replace one or both of the inverters I1 and I3 (together with the TGs that follow them) in Figure 2.17(a) by clocked inverters. There is not much to choose between the different implementations in this case, except that layout may be easier for the clocked inverter versions (since there is one less connection to make).

More interesting is the flip-flop design: We can only replace inverters I1, I3, and I7 (and the TGs that follow them) in Figure 2.18(a) by clocked inverters. We cannot replace inverter I6 because it is not directly connected to a TG. We can replace the TG attached to node M with a clocked inverter, and this will invert the sense of the output Q, which thus becomes QN. Now the clock-to-Q delay will be slower than clock-to-QN, since Q (which was QN) now comes one inverter later than QN.

If we wish to build a flip-flop with a fast clock-to-QN delay it may be better to build it using clocked inverters and use inverters with TGs for a flip-flop with a fast clock-to-Q delay. In fact, since we do not always use both Q and QN outputs of a flip-flop, some libraries include Q only or QN only flip-flops that are slightly smaller than those with both polarity outputs. It is slightly easier to layout clocked inverters than an inverter plus a TG, so flip-flops in commercial libraries include a mixture of clocked-inverter and TG implementations.

2.6 Datapath Logic Cells

Suppose we wish to build an n-bit adder (that adds two n-bit numbers) and to exploit the regularity of this function in the layout. We can do so using a *datapath* structure.

The following two functions, SUM and COUT, implement the sum and carry out for a **full adder** (**FA**) with two data inputs (A, B) and a carry in, CIN:

$$\text{SUM} = \text{A} \oplus \text{B} \oplus \text{CIN} = \text{SUM(A, B, CIN)} = \text{PARITY(A, B, CIN)}, \tag{2.38}$$

$$\text{COUT} = \text{A} \cdot \text{B} + \text{A} \cdot \text{CIN} + \text{B} \cdot \text{CIN} = \text{MAJ(A, B, CIN)}. \tag{2.39}$$

The sum uses the **parity function** ('1' if there are an odd numbers of '1's in the inputs). The carry out, COUT, uses the 2-of-3 **majority function** ('1' if the majority of the inputs are '1'). We can combine these two functions in a single FA logic cell, ADD(A[i], B[i], CIN, S[i], COUT), shown in Figure 2.20(a), where

$$\text{S}[i] = \text{SUM(A}[i], \text{B}[i], \text{CIN}), \tag{2.40}$$

$$\text{COUT} = \text{MAJ(A}[i], \text{B}[i], \text{CIN}). \tag{2.41}$$

Now we can build a 4-bit **ripple-carry adder** (**RCA**) by connecting four of these ADD cells together as shown in Figure 2.20(b). The ith ADD cell is arranged with the following: two bus inputs A[i], B[i]; one bus output S[i]; an input, CIN, that is the carry in from stage $(i-1)$ below and is also passed up to the cell above as an output; and an output, COUT, that is the carry out to stage $(i+1)$ above. In the 4-bit adder shown in Figure 2.20(b) we connect the carry input, CIN[0], to VSS and use COUT[3] and COUT[2] to indicate arithmetic overflow (in Section 2.6.1 we shall see why we may need both signals). Notice that we build the ADD cell so that COUT[2] is available at the top of the datapath when we need it.

Figure 2.20(c) shows a layout of the ADD cell. The A inputs, B inputs, and S outputs all use m1 interconnect running in the horizontal direction—we call these **data** signals. Other signals can enter or exit from the top or bottom and run vertically across the datapath in m2—we call these **control** signals. We can also use m1 for control and m2 for data, but we normally do not mix these approaches in the same structure. Control signals are typically clocks and other signals common to elements. For example, in Figure 2.20(c) the carry signals, CIN and COUT, run vertically in m2 between cells. To build a 4-bit adder we stack four ADD cells creating the array structure shown in Figure 2.20(d). In this case the A and B data bus inputs enter from the left and bus S, the sum, exits at the right, but we can connect A, B, and S to either side if we want.

The layout of buswide logic that operates on data signals in this fashion is called a **datapath**. The module ADD is a **datapath cell** or **datapath element**. Just as we do for standard cells we make all the datapath cells in a library the same height so we can abut other datapath cells on either side of the adder to create a more complex datapath. When people talk about a datapath they always assume that it is oriented so that increasing the size in bits makes the datapath grow in height,

upwards in the vertical direction, and adding different datapath elements to increase the function makes the datapath grow in width, in the horizontal direction—but we can rotate and position a completed datapath in any direction we want on a chip.

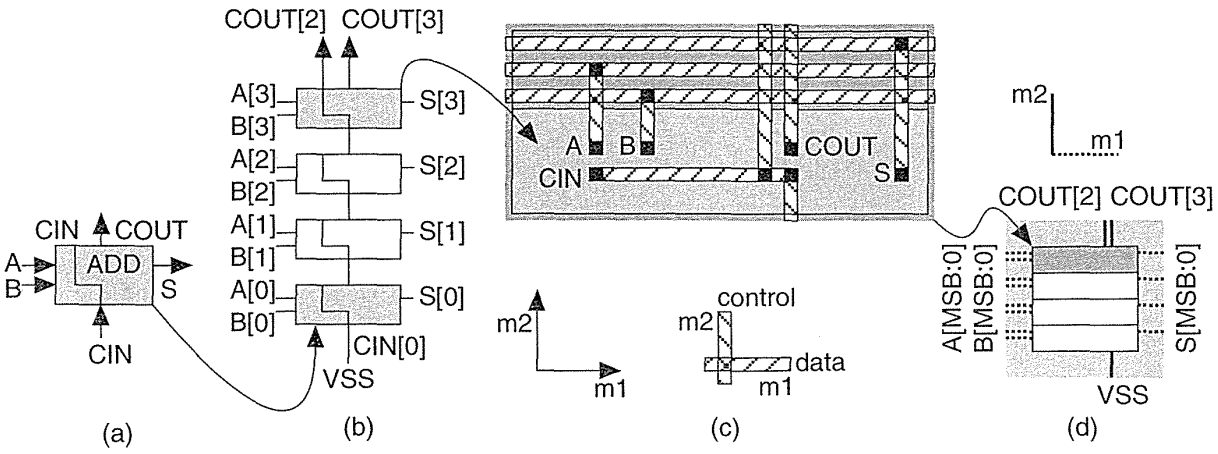


FIGURE 2.20 A datapath adder. (a) A full-adder (FA) cell with inputs (A and B), a carry in, CIN, sum output, S, and carry out, COUT. (b) A 4-bit adder. (c) The layout, using two-level metal, with data in m1 and control in m2. In this example the wiring is completed outside the cell; it is also possible to design the datapath cells to contain the wiring. Using three levels of metal, it is possible to wire over the top of the datapath cells. (d) The datapath layout.

What is the difference between using a datapath, standard cells, or gate arrays? Cells are placed together in rows on a CBIC or an MGA, but there is no generally no regularity to the arrangement of the cells within the rows—we let software arrange the cells and complete the interconnect. Datapath layout automatically takes care of most of the interconnect between the cells with the following advantages:

- Regular layout produces predictable and equal delay for each bit.
- Interconnect between cells can be built into each cell.

There are some disadvantages of using a datapath:

- The overhead (buffering and routing the control signals, for example) can make a narrow (small number of bits) datapath larger and slower than a standard-cell (or even gate-array) implementation.
- Datapath cells have to be predesigned (otherwise we are using full-custom design) for use in a wide range of datapath sizes. Datapath cell design can be harder than designing gate-array macros or standard cells.
- Software to assemble a datapath is more complex and not as widely used as software for assembling standard cells or gate arrays.

There are some newer standard-cell and gate-array tools that can take advantage of regularity in a design and position cells carefully. The problem is in finding the regularity if it is not specified. Using a datapath is one way to specify regularity to ASIC design tools.

2.6.1 Datapath Elements

Figure 2.21 shows some typical datapath symbols for an adder (people rarely use the IEEE standards in ASIC datapath libraries). I use heavy lines (they are 1.5 point wide) with a stroke to denote a data bus (that flows in the horizontal direction in a datapath), and regular lines (0.5 point) to denote the control signals (that flow vertically in a datapath). At the risk of adding confusion where there is none, this stroke to indicate a data bus has nothing to do with mixed-logic conventions. For a bus, A[31:0] denotes a 32-bit bus with A[31] as the leftmost or **most-significant bit** or **MSB**, and A[0] as the **least-significant bit** or **LSB**. Sometimes we shall use A[MSB] or A[LSB] to refer to these bits. Notice that if we have an n-bit bus and LSB = 0, then MSB = $n-1$. Also, for example, A[4] is the fifth bit on the bus (from the LSB). We use a 'Σ' or 'ADD' inside the symbol to denote an adder instead of '+', so we can attach '–' or '+/–' to the inputs for a subtracter or adder/subtracter.

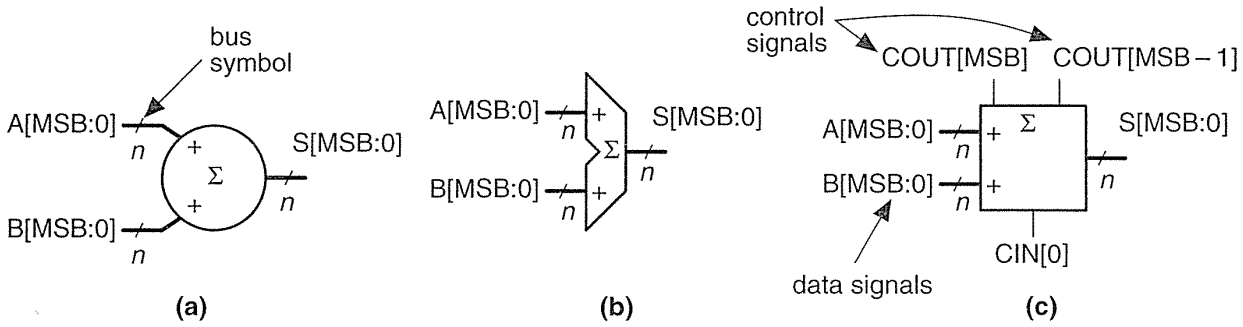


FIGURE 2.21 Symbols for a datapath adder. (a) A data bus is shown by a heavy line (1.5 point) and a bus symbol. If the bus is n-bits wide then MSB = $n-1$. (b) An alternative symbol for an adder. (c) Control signals are shown as lightweight (0.5 point) lines.

Some schematic datapath symbols include only data signals and omit the control signals—but we must not forget them. In Figure 2.21, for example, we may need to explicitly tie CIN[0] to VSS and use COUT[MSB] and COUT[MSB – 1] to detect overflow. Why might we need both of these control signals? Table 2.11 shows the process of simple arithmetic for the different binary number representations, including unsigned, signed magnitude, ones' complement, and two's complement.

TABLE 2.11 Binary arithmetic.

Operation	Binary Number Representation			
	Unsigned	Signed magnitude	Ones' complement	Two's complement
	no change	**if** positive **then** MSB = 0 **else** MSB = 1	**if** negative **then** flip bits	**if** negative **then** {flip bits; add 1}
3 =	0011	0011	0011	0011
–3 =	NA	1011	1100	1101
zero =	0000	0000 or 1000	1111 or 0000	0000
max. positive =	1111 = 15	0111 = 7	0111 = 7	0111 = 7
max. negative =	0000 = 0	1111 = –7	1000 = –7	1000 = –8
addition = S = A + B = addend + augend SG(A) = sign of A	S = A + B	**if** SG(A) = SG(B) **then** S = A + B **else** {**if** B < A **then** S = A – B **else** S = B – A}	S = A + B + COUT[MSB] COUT is carry out	S = A + B
addition result: OV = overflow, OR = out of range	OR = COUT[MSB] COUT is carry out	**if** SG(A) = SG(B) **then** OV = COUT[MSB] **else** OV = 0 (impossible)	OV = XOR(COUT[MSB], COUT[MSB–1])	OV = XOR(COUT[MSB], COUT[MSB – 1])
SG(S) = sign of S S = A + B	NA	**if** SG(A) = SG(B) **then** SG(S) = SG(A) **else** {**if** B < A **then** SG(S) = SG(A) **else** SG(S) = SG(B)}	NA	NA
subtraction = D = A – B = minuend – subtrahend	D = A – B	SG(B) = NOT(SG(B)); D = A + B	Z = –B (negate); D = A + Z	Z = –B (negate); D = A + Z
subtraction result: OV = overflow, OR = out of range	OR = BOUT[MSB] BOUT is borrow out	as in addition	as in addition	as in addition
negation: Z = –A (negate)	NA	Z = A; SG(Z) = NOT(SG(A))	Z = NOT(A)	Z = NOT(A) + 1

2.6.2 Adders

We can view addition in terms of **generate**, G[i], and **propagate**, P[i], signals.

method 1	method 2	
$G[i] = A[i] \cdot B[i]$	$G[i] = A[i] \cdot B[i]$	(2.42)
$P[i] = A[i] \oplus B[i]$	$P[i] = A[i] + B[i]$	(2.43)
$C[i] = G[i] + P[i] \cdot C[i-1]$	$C[i] = G[i] + P[i] \cdot C[i-1]$	(2.44)
$S[i] = P[i] \oplus C[i-1]$	$S[i] = A[i] \oplus B[i] \oplus C[i-1]$	(2.45)

where C[i] is the carry-out signal from stage i, equal to the carry in of stage $(i+1)$. Thus, $C[i] = COUT[i] = CIN[i+1]$. We need to be careful because C[0] might represent either the carry in or the carry out of the LSB stage. For an adder we set the carry in to the first stage (stage zero), C[–1] or CIN[0], to '0'. Some people use **delete** (D) or **kill** (K) in various ways for the complements of G[i] and P[i], but unfortunately others use C for COUT and D for CIN—so I avoid using any of these. Do not confuse the two different methods (both of which are used) in Eqs. 2.42–2.45 when forming the sum, since the propagate signal, P[i], is different for each method.

Figure 2.22(a) shows a conventional RCA. The delay of an n-bit RCA is proportional to n and is limited by the propagation of the carry signal through all of the stages. We can reduce delay by using pairs of "go-faster" bubbles to change AND and OR gates to fast two-input NAND gates as shown in Figure 2.22(a). Alternatively, we can write the equations for the carry signal in two different ways:

$$\text{either} \qquad C[i] = A[i] \cdot B[i] + P[i] \cdot C[i-1] \tag{2.46}$$

$$\text{or} \qquad C[i] = (A[i] + B[i]) \cdot (P[i]' + C[i-1]), \tag{2.47}$$

where P[i]' = NOT(P[i]). Equations 2.46 and 2.47 allow us to build the carry chain from two-input NAND gates, one per cell, using different logic in even and odd stages (Figure 2.22b):

even stages	odd stages	
$C1[i]' = P[i] \cdot C3[i-1] \cdot C4[i-1]$	$C3[i]' = P[i] \cdot C1[i-1] \cdot C2[i-1]$	(2.48)
$C2[i] = A[i] + B[i]$	$C4[i]' = A[i] \cdot B[i]$	(2.49)
$C[i] = C1[i] \cdot C2[i]$	$C[i] = C3[i]' + C4[i]'$	(2.50)

(the carry inputs to stage zero are C3[–1] = C4[–1] = '0'). We can use the RCA of Figure 2.22(b) in a datapath, with standard cells, or on a gate array.

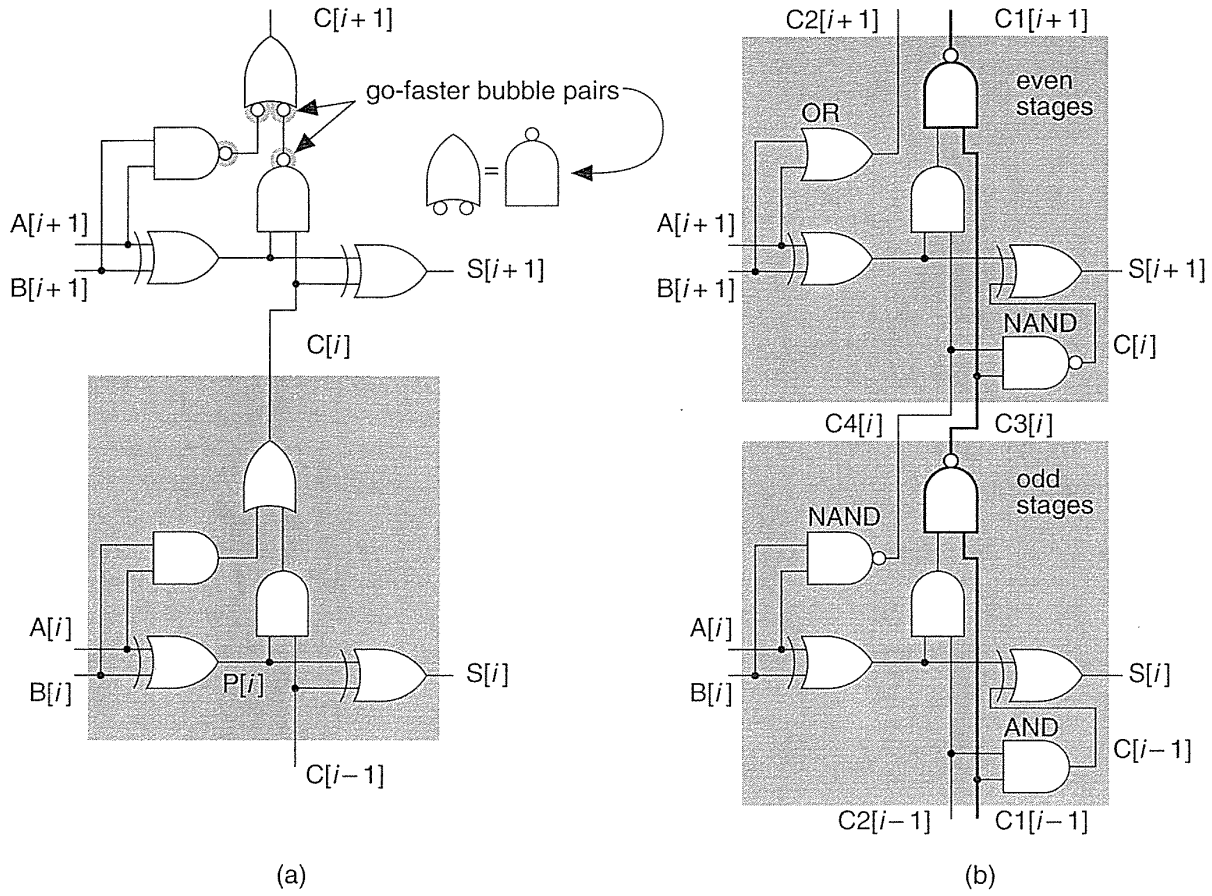


FIGURE 2.22 The ripple-carry adder (RCA). (a) A conventional RCA. The delay may be reduced slightly by adding pairs of bubbles as shown to use two-input NAND gates. (b) An alternative RCA circuit topology using different cells for odd and even stages and an extra connection between cells. The carry chain is a fast string of NAND gates (shown in bold).

Instead of propagating the carries through each stage of an RCA, Figure 2.23 shows a different approach. A **carry-save adder** (CSA) cell CSA(A1[i], A2[i], A3[i], CIN, S1[i], S2[i], COUT) has three outputs:

$$S1[i] = CIN \tag{2.51}$$

$$S2[i] = A1[i] \oplus A2[i] \oplus A3[i] = PARITY(A1[i], A2[i], A3[i]) \tag{2.52}$$

$$COUT = A1[i] \cdot A2[i] + [(A1[i] + A2[i]) \cdot A3[i]] = MAJ(A1[i], A2[i], A3[i]) \tag{2.53}$$

The inputs, A1, A2, and A3; and outputs, S1 and S2, are buses. The input, CIN, is the carry from stage $(i-1)$. The carry in, CIN, is connected directly to the output bus S1—indicated by the schematic symbol (Figure 2.23a). We connect CIN[0] to VSS. The output, COUT, is the carry out to stage $(i+1)$.

A 4-bit CSA is shown in Figure 2.23(b). The arithmetic overflow signal for ones' complement or two's complement arithmetic, OV, is XOR(COUT[MSB], COUT[MSB – 1]) as shown in Figure 2.23(c). In a CSA the carries are "saved" at each stage and shifted left onto the bus S1. There is thus no carry propagation and the delay of a CSA is constant. At the output of a CSA we still need to add the S1 bus (all the saved carries) and the S2 bus (all the sums) to get an n-bit result using a final stage that is not shown in Figure 2.23(c). We might regard the n-bit sum as being encoded in the two buses, S1 and S2, in the form of the parity and majority functions.

We can use a CSA to add multiple inputs—as an example, an adder with four 4-bit inputs is shown in Figure 2.23(d). The last stage sums two input buses using a **carry-propagate adder (CPA)**. We have used an RCA as the CPA in Figure 2.23(d) and (e), but we can use any type of adder. Notice in Figure 2.23(e) how the two CSA cells and the RCA cell abut together horizontally to form a **bit slice** (or slice) and then the slices are stacked vertically to form the datapath.

We can register the CSA stages by adding vectors of flip-flops as shown in Figure 2.23(f). This reduces the adder delay to that of the slowest adder stage, usually the CPA. By using registers between stages of combinational logic we use **pipelining** to increase the speed and pay a price of increased area (for the registers) and introduce **latency**. It takes a few clock cycles (the latency, equal to n clock cycles for an n-stage pipeline) to fill the pipeline, but once it is filled, the answers emerge every clock cycle. Ferris wheels work much the same way. When the fair opens it takes a while (latency) to fill the wheel, but once it is full the people can get on and off every few seconds. (We can also pipeline the RCA of Figure 2.20. We add i registers on the A and B inputs before ADD[i] and add $(n-i)$ registers after the output S[i], with a single register before each C[i].)

The problem with an RCA is that every stage has to wait to make its carry decision, C[i], until the previous stage has calculated C[$i-1$]. If we examine the propagate signals we can bypass this critical path. Thus, for example, to bypass the carries for bits 4–7 (stages 5–8) of an adder we can compute BYPASS = P[4].P[5].P[6].P[7] and then use a MUX as follows:

$$C[7] = (G[7] + P[7] \cdot C[6]) \cdot BYPASS' + C[3] \cdot BYPASS. \qquad (2.54)$$

Adders based on this principle are called **carry-bypass adders (CBA)** [Sato et al., 1992]. Large, custom adders employ **Manchester-carry chains** to compute the carries and the bypass operation using TGs or just pass transistors [Weste and Eshraghian, 1993, pp. 530–531]. These types of carry chains may be part of a predesigned ASIC adder cell, but are not used by ASIC designers.

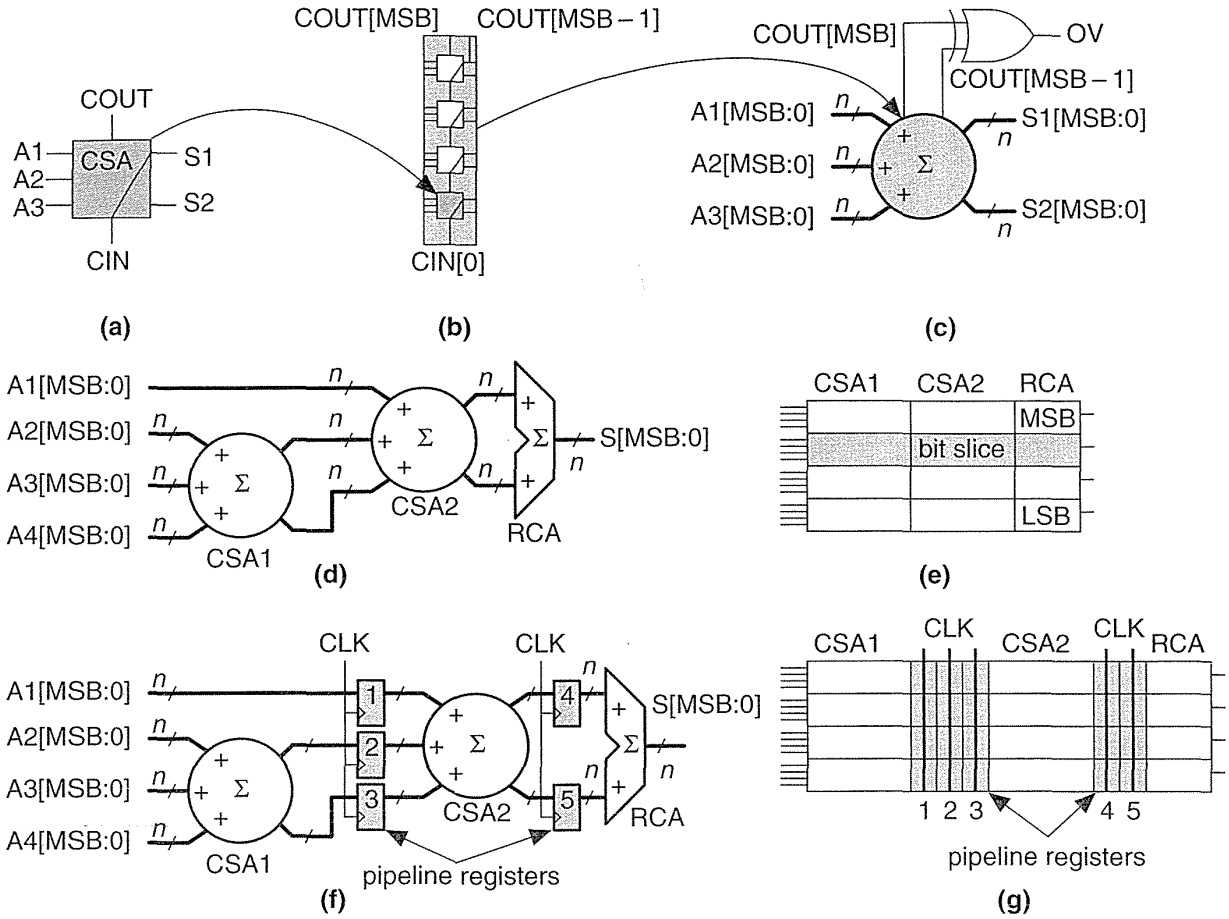


FIGURE 2.23 The carry-save adder (CSA). (a) A CSA cell. (b) A 4-bit CSA. (c) Symbol for a CSA. (d) A four-input CSA. (e) The datapath for a four-input, 4-bit adder using CSAs with a ripple-carry adder (RCA) as the final stage. (f) A pipelined adder. (g) The datapath for the pipelined version showing the pipeline registers as well as the clock control lines that use m2.

Instead of checking the propagate signals we can check the inputs. For example we can compute $\text{SKIP} = (\text{A}[i-1] \oplus \text{B}[i-1]) + (\text{A}[i] \oplus \text{B}[i])$ and then use a 2:1 MUX to select $\text{C}[i]$. Thus,

$$\text{CSKIP}[i] = (\text{G}[i] + \text{P}[i] \cdot \text{C}[i-1]) \cdot \text{SKIP}' + \text{C}[i-2] \cdot \text{SKIP}. \tag{2.55}$$

This is a **carry-skip adder** [Keutzer, Malik, and Saldanha, 1991; Lehman, 1961]. Carry-bypass and carry-skip adders may include redundant logic (since the carry is computed in two different ways—we just take the first signal to arrive). We must be careful that the redundant logic is not optimized away during logic synthesis.

If we evaluate Eq. 2.44 recursively for $i = 1$, we get the following:

$$\begin{aligned} C[1] &= G[1] + P[1] \cdot C[0] = G[1] + P[1] \cdot (G[0] + P[1] \cdot C[-1]) \\ &= G[1] + P[1] \cdot G[0]. \end{aligned} \tag{2.56}$$

This result means that we can "look ahead" by two stages and calculate the carry into the third stage (bit 2), which is C[1], using only the first-stage inputs (to calculate G[0]) and the second-stage inputs. This is a **carry-lookahead adder** (**CLA**) [MacSorley, 1961]. If we continue expanding Eq. 2.44, we find:

$$\begin{aligned} C[2] &= G[2] + P[2] \cdot G[1] + P[2] \cdot P[1] \cdot G[0], \\ C[3] &= G[3] + P[2] \cdot G[2] + P[2] \cdot P[1] \cdot G[1] + P[3] \cdot P[2] \cdot P[1] \cdot G[0]. \end{aligned} \tag{2.57}$$

As we look ahead further these equations become more complex, take longer to calculate, and the logic becomes less regular when implemented using cells with a limited number of inputs. Datapath layout must fit in a bit slice, so the physical and logical structure of each bit must be similar. In a standard cell or gate array we are not so concerned about a regular physical structure, but a regular logical structure simplifies design. The **Brent–Kung adder** reduces the delay and increases the regularity of the carry-lookahead scheme [Brent and Kung, 1982]. Figure 2.24(a) shows a regular 4-bit CLA, using the carry-lookahead generator cell (CLG) shown in Figure 2.24(b).

In a **carry-select adder** we duplicate two small adders (usually 4-bit or 8-bit adders—often CLAs) for the cases CIN = '0' and CIN = '1' and then use a MUX to select the case that we need—wasteful, but fast [Bedrij, 1962]. A carry-select adder is often used as the fast adder in a datapath library because its layout is regular.

We can use the carry-select, carry-bypass, and carry-skip architectures to split a 12-bit adder, for example, into three blocks. The delay of the adder is then partly dependent on the delays of the MUX between each block. Suppose the delay due to 1-bit in an adder block (we shall call this a bit delay) is approximately equal to the MUX delay. In this case it may be faster to make the blocks 3-, 4-, and 5-bits long instead of being equal in size. Now the delays into the final MUX are equal—3 bit-delays plus 2 MUX delays for the carry signal from bits 0–6 and 5 bit-delays for the carry from bits 7–11. Adjusting the block size reduces the delay of large adders (more than 16 bits).

We can extend the idea behind a carry-select adder as follows. Suppose we have an n-bit adder that generates two sums: One sum assumes a carry-in condition of '0', the other sum assumes a carry-in condition of '1'. We can split this n-bit adder into an i-bit adder for the i LSBs and an $(n-i)$-bit adder for the $n-i$ MSBs. Both of the smaller adders generate two conditional sums as well as true and complement carry signals. The two (true and complement) carry signals from the LSB adder are used to select between the two $(n-i+1)$-bit conditional sums from the MSB adder using $2(n-i+1)$ two-input MUXes. This is a **conditional-sum adder** (also often abbreviated to CSA) [Sklansky, 1960]. We can recursively apply this technique. For example, we can split a 16-bit adder using $i = 8$ and $n = 8$; then we can split one or both 8–bit adders again—and so on.

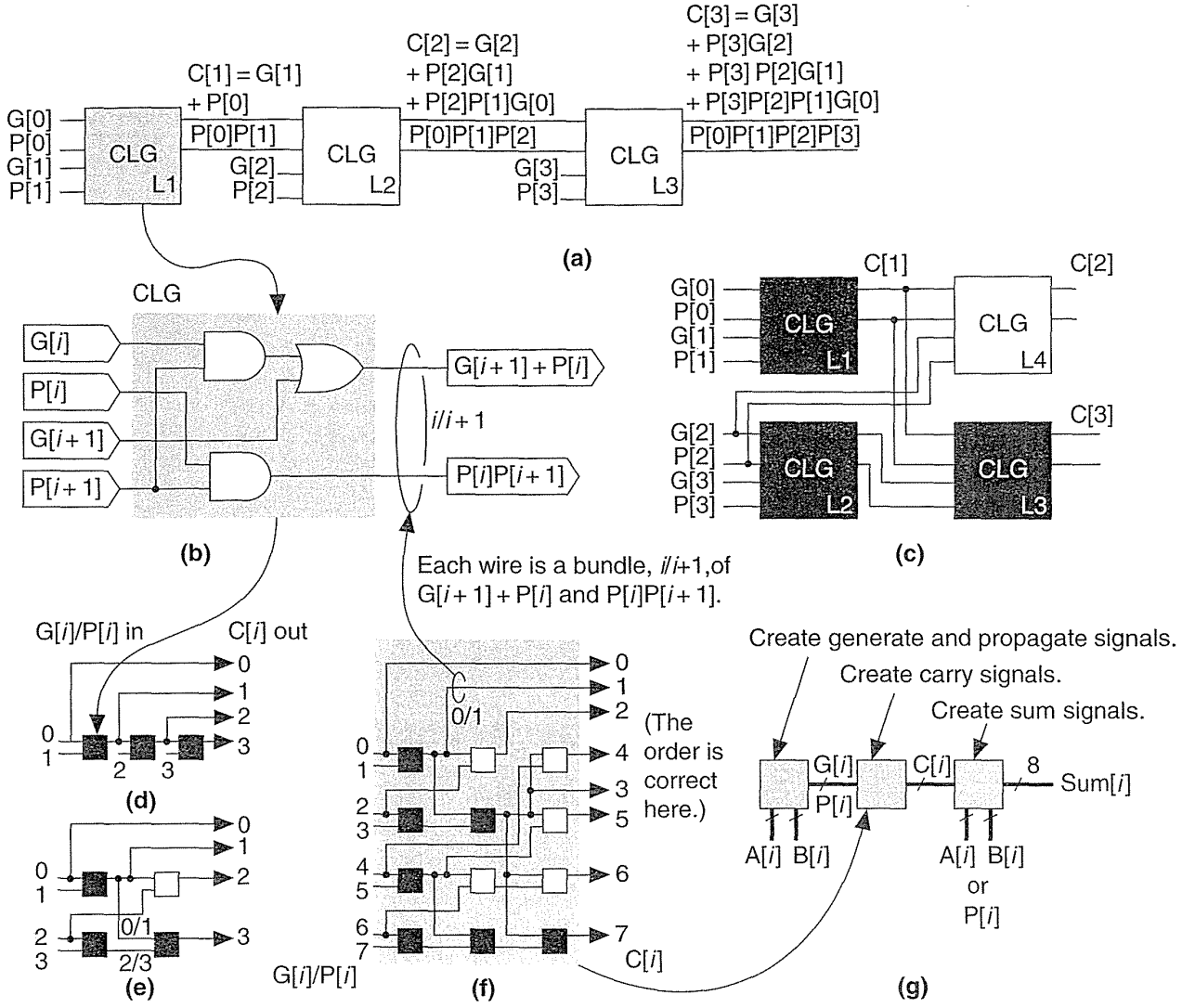
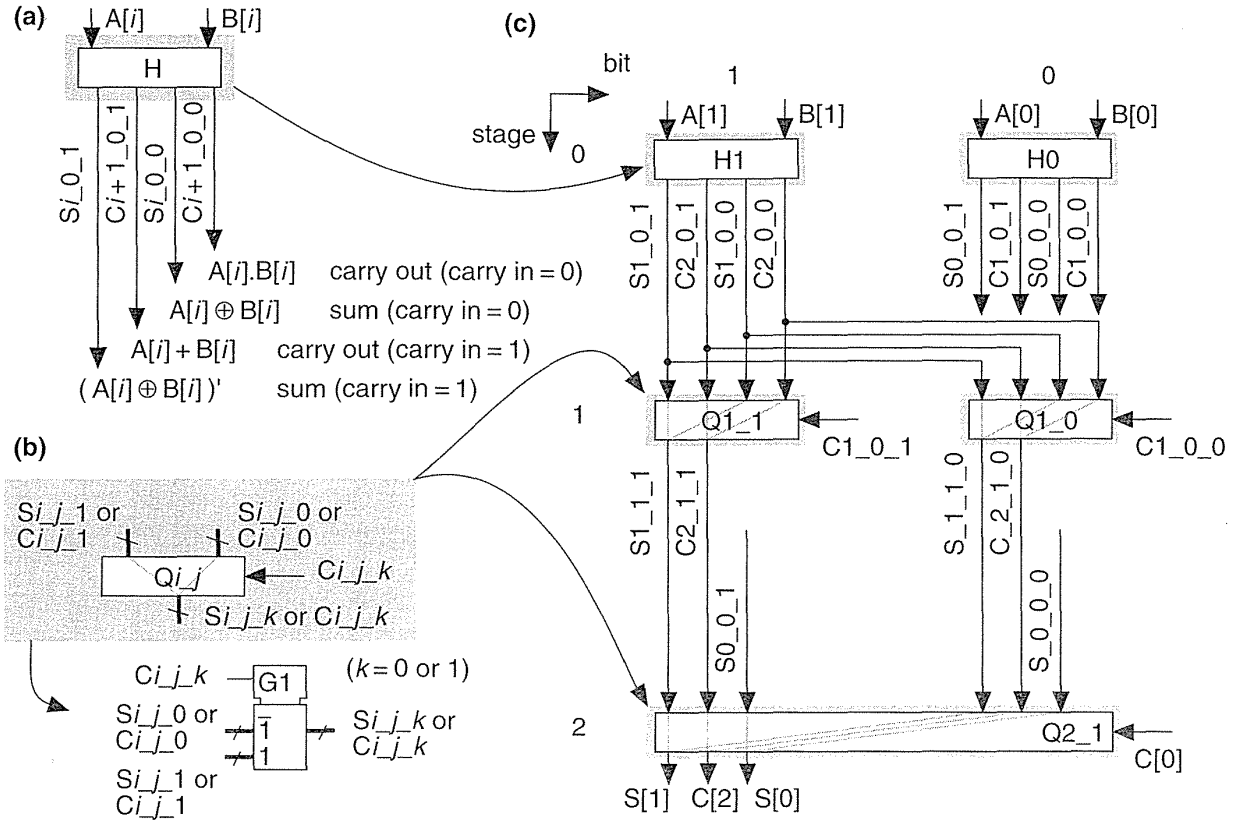


FIGURE 2.24 The Brent–Kung carry-lookahead adder (CLA). (a) Carry generation in a 4-bit CLA. (b) A cell to generate the lookahead terms, C[0]–C[3]. (c) Cells L1, L2, and L3 are rearranged into a tree that has less delay. Cell L4 is added to calculate C[2] that is lost in the translation. (d) and (e) Simplified representations of parts a and c. (f) The lookahead logic for an 8-bit adder. The inputs, 0–7, are the propagate and carry terms formed from the inputs to the adder. (g) An 8-bit Brent–Kung CLA. The outputs of the lookahead logic are the carry bits that (together with the inputs) form the sum. One advantage of this adder is that delays from the inputs to the outputs are more nearly equal than in other adders. This tends to reduce the number of unwanted and unnecessary switching events and thus reduces power dissipation.

Figure 2.25 shows the simplest form of an n-bit conditional-sum adder that uses n single-bit conditional adders, H (each with four outputs: two conditional sums, true carry, and complement carry), together with a tree of 2:1 MUXes (Qi_j). The conditional-sum adder is usually the fastest of all the adders we have discussed (it is the fastest when logic cell delay increases with the number of inputs—this is true for all ASICs except FPGAs).



Ci_j_k = carry in to the ith bit assuming the carry in to the jth bit is k (k = 0 or 1)
Si_j_k = sum at the ith bit assuming the carry in to the jth bit is k (k = 0 or 1)

FIGURE 2.25 The conditional-sum adder. (a) A 1-bit conditional adder that calculates the sum and carry out assuming the carry in is either '1' or '0'. (b) The multiplexer that selects between sums and carries. (c) A 4-bit conditional-sum adder with carry input, C[0].

2.6.3 A Simple Example

How do we make and use datapath elements? What does a design look like? We may use predesigned cells from a library or build the elements ourselves from logic cells

using a schematic or a design language. Table 2.12 shows an 8-bit conditional-sum adder intended for an FPGA. This Verilog implementation uses the same structure as Figure 2.25, but the equations are collapsed to use four or five variables. A basic logic cell in certain Xilinx FPGAs, for example, can implement two equations of the same four variables or one equation with five variables. The equations shown in Table 2.12 requires three levels of FPGA logic cells (so, for example, if each FPGA logic cell has a 5ns delay, the 8-bit conditional-sum adder delay is 15ns).

TABLE 2.12 An 8-bit conditional-sum adder (the notation is described in Figure 2.25).

```
module m8bitCSum (C0, a, b, s, C8); // Verilog conditional-sum adder for an FPGA        //1
input [7:0] C0, a, b; output [7:0] s; output C8;                                        //2
wire A7,A6,A5,A4,A3,A2,A1,A0,B7,B6,B5,B4,B3,B2,B1,B0,S8,S7,S6,S5,S4,S3,S2,S1,S0;        //3
wire C0, C2, C4_2_0, C4_2_1, S5_4_0, S5_4_1, C6, C6_4_0, C6_4_1, C8;                    //4
assign {A7,A6,A5,A4,A3,A2,A1,A0} = a; assign {B7,B6,B5,B4,B3,B2,B1,B0} = b;             //5
assign s = { S7,S6,S5,S4,S3,S2,S1,S0 };                                                 //6
assign S0 = A0^B0^C0 ; // start of level 1: & = AND, ^ = XOR, | = OR, ! = NOT          //7
assign S1 = A1^B1^(A0&B0|(A0|B0)&C0) ;                                                  //8
assign C2 = A1&B1|(A1|B1)&(A0&B0|(A0|B0)&C0) ;                                          //9
assign C4_2_0 = A3&B3|(A3|B3)&(A2&B2) ; assign  C4_2_1 = A3&B3|(A3|B3)&(A2|B2) ;       //10
assign S5_4_0 = A5^B5^(A4&B4) ; assign S5_4_1 = A5^B5^(A4|B4) ;                         //11
assign C6_4_0 = A5&B5|(A5|B5)&(A4&B4) ; assign C6_4_1 = A5&B5|(A5|B5)&(A4|B4) ;         //12
assign S2 = A2^B2^C2 ; // start of level 2                                             //13
assign S3 = A3^B3^(A2&B2|(A2|B2)&C2) ;                                                  //14
assign S4 = A4^B4^(C4_2_0|C4_2_1&C2) ;                                                  //15
assign S5 = S5_4_0& !(C4_2_0|C4_2_1&C2)|S5_4_1&(C4_2_0|C4_2_1&C2) ;                     //16
assign C6 = C6_4_0|C6_4_1&(C4_2_0|C4_2_1&C2) ;                                          //17
assign S6 = A6^B6^C6 ; // start of level 3                                             //18
assign S7 = A7^B7^(A6&B6|(A6|B6)&C6) ;                                                  //19
assign C8 = A7&B7|(A7|B7s)&(A6&B6|(A6|B6)&C6) ;                                         //20
endmodule                                                                               //21
```

Source: R. Halverson, University of Hawaii.

Figure 2.26 shows the normalized delay and area figures for a set of predesigned datapath adders. The data in Figure 2.26 is from a series of ASIC datapath cell libraries (Compass Passport) that may be synthesized together with test vectors and simulation models. We can combine the different adder techniques, but the adders then lose regularity and become less suited to a datapath implementation.

There are other adders that are not used in datapaths, but are occasionally useful in ASIC design. A **serial adder** is smaller but slower than the **parallel adders** we have described [Denyer and Renshaw, 1985]. The **carry-completion adder** is a variable delay adder and rarely used in synchronous designs [Sklansky, 1960].

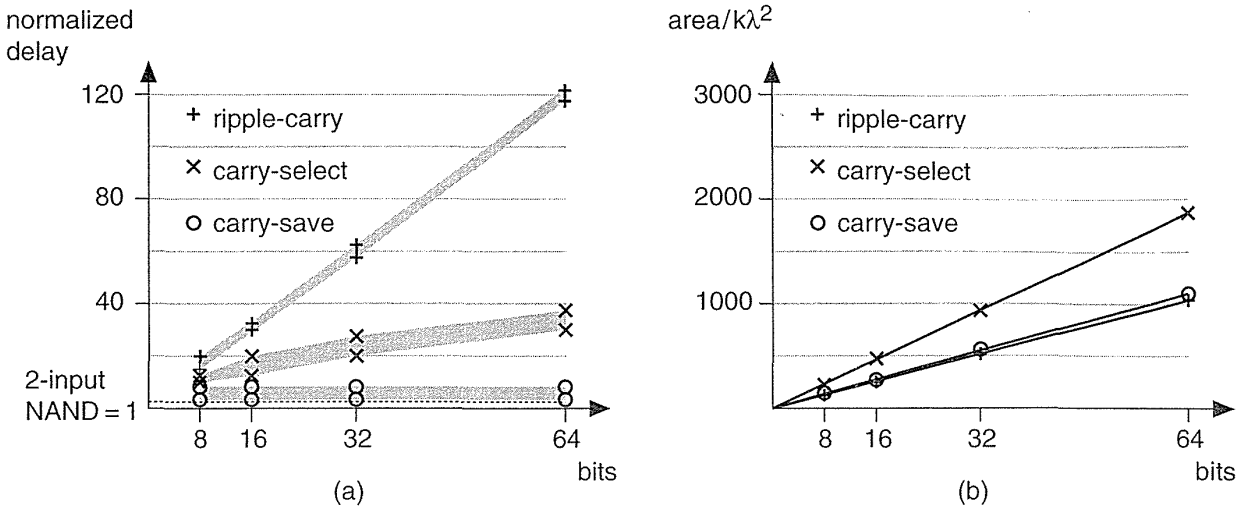


FIGURE 2.26 Datapath adders. This data is from a series of submicron datapath libraries. (a) Delay normalized to a two-input NAND logic cell delay (approximately equal to 250 ps in a 0.5 μm process). For example, a 64-bit ripple-carry adder (RCA) has a delay of approximately 30 ns in a 0.5 μm process. The spread in delay is due to variation in delays between different inputs and outputs. An n-bit RCA has a delay proportional to n. The delay of an n-bit carry-select adder is approximately proportional to $\log_2 n$. The carry-save adder delay is constant (but requires a carry-propagate adder to complete an addition). (b) In a datapath library the area of all adders are proportional to the bit size.

2.6.4 Multipliers

Figure 2.27 shows a symmetric 6-bit array **multiplier** (an n-bit multiplier multiplies two n-bit numbers; we shall use n-bit by m-bit multiplier if the lengths are different). Adders a0–f0 may be eliminated, which then eliminates adders a1–a6, leaving an asymmetric CSA array of 30 (5×6) adders (including one half adder). An n-bit array multiplier has a delay proportional to n plus the delay of the CPA (adders b6–f6 in Figure 2.27). There are two items we can attack to improve the performance of a multiplier: the number of partial products and the addition of the partial products.

Suppose we wish to multiply 15 (the **multiplicand**) by 19 (the **multiplier**) mentally. It is easier to calculate 15×20 and subtract 15. In effect we complete the multiplication as $15 \times (20 - 1)$ and we could write this as $15 \times 2\bar{1}$, with the overbar representing a minus sign. Now suppose we wish to multiply an 8-bit binary number, A, by B = 00010111 (decimal $16 + 4 + 2 + 1 = 23$). It is easier to multiply A by the canonical signed-digit vector (**CSD vector**) D = 0010$\bar{1}$001 (decimal $32 - 8 + 1 = 23$) since this requires only three add or subtract operations (and a sub-

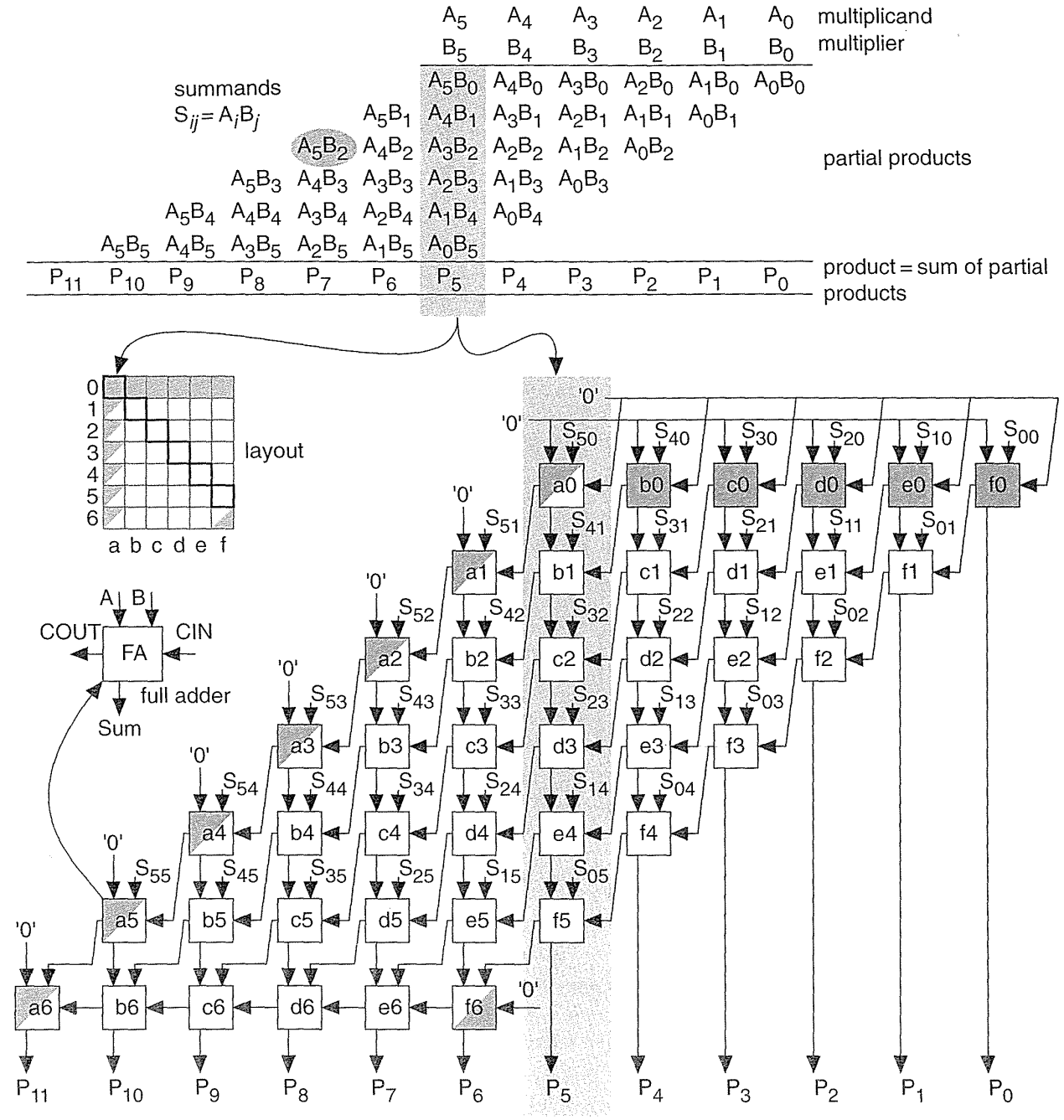


FIGURE 2.27 Multiplication. A 6-bit array multiplier using a final carry-propagate adder (full-adder cells a6–f6, a ripple-carry adder). Apart from the generation of the summands this multiplier uses the same structure as the carry-save adder of Figure 2.23(d).

traction is as easy as an addition). We say B has a **weight** of 4 and D has a weight of 3. By using D instead of B we have reduced the number of partial products by 1 (=4 – 3).

We can **recode** (or encode) any binary number, B, as a CSD vector, D, as follows (canonical means there is only one CSD vector for any number):

$$D_i = B_i + C_i - 2C_{i+1}, \tag{2.58}$$

where C_{i+1} is the carry from the sum of $B_{i+1} + B_i + C_i$ (we start with $C_0 = 0$).

As another example, if B = 011 ($B_2 = 0$, $B_1 = 1$, $B_0 = 1$; decimal 3), then, using Eq. 2.58,

$$\begin{aligned} D_0 &= B_0 + C_0 - 2C_1 = 1 + 0 - 2 = \bar{1}, \\ D_1 &= B_1 + C_1 - 2C_2 = 1 + 1 - 2 = 0, \\ D_2 &= B_2 + C_2 - 2C_3 = 0 + 1 - 0 = 1, \end{aligned} \tag{2.59}$$

so that D = $10\bar{1}$ (decimal 4 – 1 = 3). CSD vectors are useful to represent fixed coefficients in digital filters, for example.

We can recode using a **radix** other than 2. Suppose B is an $(n+1)$-digit two's complement number,

$$B = B_0 + B_1 2 + B_2 2^2 + \ldots + B_i 2^i + \ldots + B_{n-1} 2^{n-1} - B_n 2^n. \tag{2.60}$$

We can rewrite the expression for B using the following sleight-of-hand:

$$\begin{aligned} 2B - B = B &= -B_0 + (B_0 - B_1)2 + \ldots + (B_{i-1} - B_i)2^i + \ldots + B_{n-1} 2^{n-1} - B_n 2^n \\ &= (-2B_1 + B_0)2^0 + (-2B_3 + B_2 + B_1)2^2 + \ldots \\ &\quad + (-2B_i + B_{i-1} + B_{i-2})2^{i-1} + (-2B_{i+2} + B_{i+1} + B_i)2^{i+1} + \ldots \\ &\quad + (-2B_n + B_{i-1} + B_{i-2})2^{n-1}. \end{aligned} \tag{2.61}$$

This is very useful. Consider B = 101001 (decimal 9 – 32 = –23, $n = 5$),

$$\begin{aligned} B = 101001 &= (-2B_1 + B_0)2^0 + (-2B_3 + B_2 + B_1)2^2 + (-2B_5 + B_4 + B_3)2^4 \\ &= ((-2 \times 0) + 1)2^0 + ((-2 \times 1) + 0 + 0)2^2 + ((-2 \times 1) + 0 + 1)2^4. \end{aligned} \tag{2.62}$$

Equation 2.61 tells us how to encode B as a radix-4 signed digit, E = $\bar{1}\bar{2}1$ (decimal –16 – 8 + 1 = –23). To multiply by B encoded as E we only have to perform a multiplication by 2 (a shift) and three add/subtract operations.

Using Eq. 2.61 we can encode any number by taking groups of three bits at a time and calculating

$$E_j = -2B_i + B_{i-1} + B_{i-2}, \quad E_{j+1} = -2B_{i+2} + B_{i+1} + B_i, \quad \ldots, \tag{2.63}$$

where each 3-bit group overlaps by one bit. We pad B with a zero, $B_n \ldots B_1 B_0 0$, to match the first term in Eq. 2.61. If B has an odd number of bits, then we extend the sign: $B_n B_n \ldots B_1 B_0 0$. For example, B = 01011 (eleven), encodes to $E = 1\bar{1}\bar{1}$ $(16 - 4 - 1)$; and B = 101 is $E = \bar{1}1$. This is called **Booth encoding** and reduces the number of partial products by a factor of two and thus considerably reduces the area as well as increasing the speed of our multiplier [Booth, 1951].

Next we turn our attention to improving the speed of addition in the CSA array. Figure 2.28(a) shows a section of the 6-bit array multiplier from Figure 2.27. We can collapse the chain of adders a0–f5 (5 adder delays) to the **Wallace tree** consisting of adders 5.1–5.4 (4 adder delays) shown in Figure 2.28(b).

Figure 2.28(c) pictorially represents multiplication as a sort of golf course. Each link corresponds to an adder. The holes or dots are the outputs of one stage (and the inputs of the next). At each stage we have the following three choices: (1) sum three outputs using a full adder (denoted by a box enclosing three dots); (2) sum two outputs using a half adder (a box with two dots); (3) pass the outputs directly to the next stage. The two outputs of an adder are joined by a diagonal line (full adders use black dots, half adders white dots). The object of the game is to choose (1), (2), or (3) at each stage to maximize the performance of the multiplier. In **tree-based multipliers** there are two ways to do this—working forward and working backward.

In a **Wallace-tree multiplier** we work forward from the multiplier inputs, compressing the number of signals to be added at each stage [Wallace, 1960]. We can view an FA as a **3:2 compressor** or **(3, 2) counter**—it counts the number of '1's on the inputs. Thus, for example, an input of '101' (two '1's) results in an output '10' (2). A half adder is a **(2, 2) counter**. To form P_5 in Figure 2.29 we must add 6 summands (S_{05}, S_{14}, S_{23}, S_{32}, S_{41}, and S_{50}) and 4 carries from the P_4 column. We add these in stages 1–7, compressing from 6:3:2:2:3:1:1. Notice that we wait until stage 5 to add the last carry from column P_4, and this means we expand (rather than compress) the number of signals (from 2 to 3) between stages 3 and 5. The maximum delay through the CSA array of Figure 2.29 is 6 adder delays. To this we must add the delay of the 4-bit (9 inputs) CPA (stage 7). There are 26 adders (6 half adders) plus the 4 adders in the CPA.

In a **Dadda multiplier** (Figure 2.30) we work backward from the final product [Dadda, 1965]. Each stage has a maximum of 2, 3, 4, 6, 9, 13, 19, ... outputs (each successive stage is 3/2 times larger—rounded down to an integer). Thus, for example, in Figure 2.28(d) we require 3 stages (with 3 adder delays—plus the delay of a 10-bit output CPA) for a 6-bit Dadda multiplier. There are 19 adders (4 half adders) in the CSA plus the 10 adders (2 half adders) in the CPA. A Dadda multiplier is usually faster and smaller than a Wallace-tree multiplier.

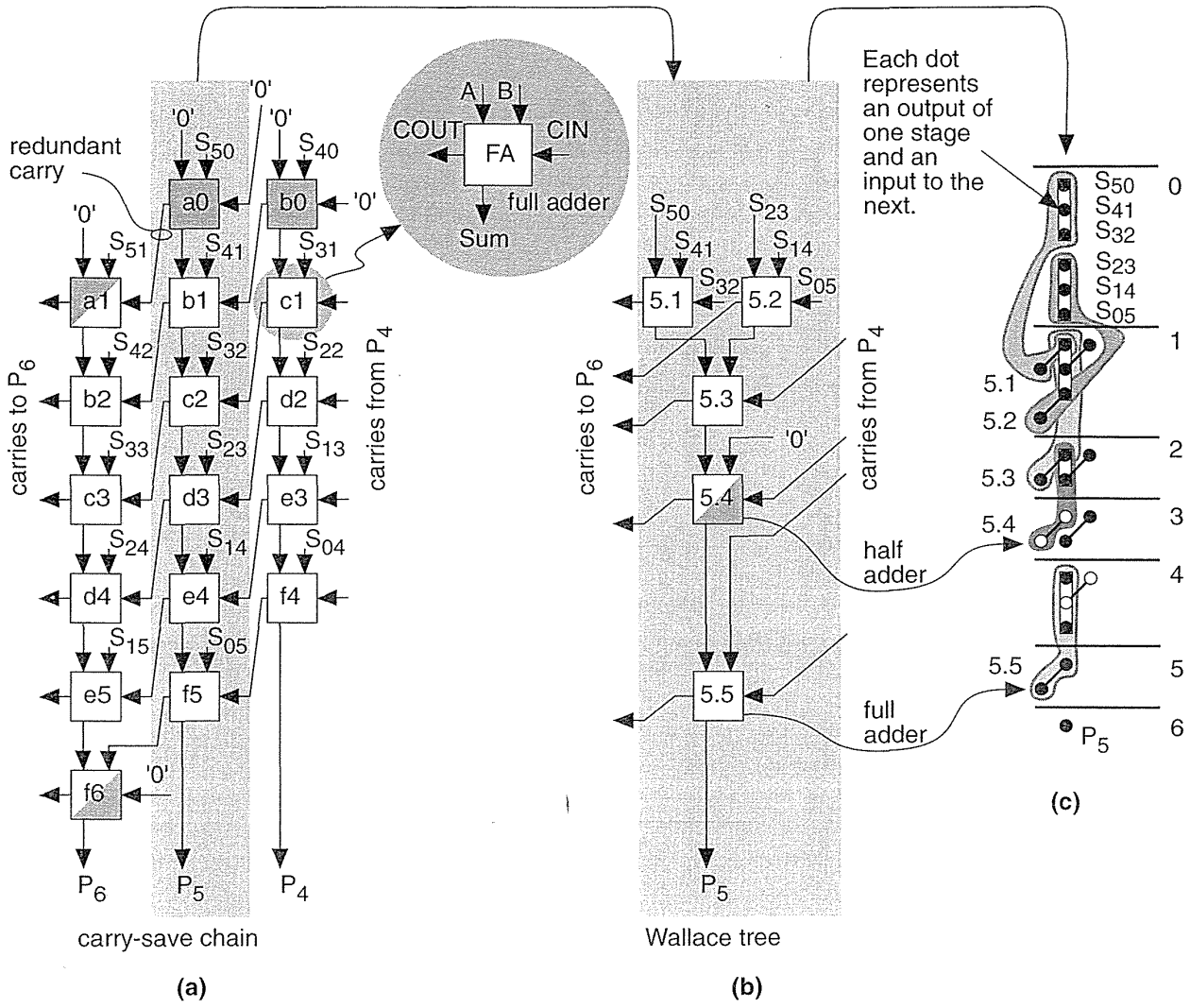


FIGURE 2.28 Tree-based multiplication. (a) The portion of Figure 2.27 that calculates the sum bit, P_5, using a chain of adders (cells a0–f5). (b) We can collapse this chain to a Wallace tree (cells 5.1–5.5). (c) The stages of multiplication.

In general, the number of stages and thus delay (in units of an FA delay—excluding the CPA) for an n-bit tree-based multiplier using (3, 2) counters is

$$\log_{1.5} n = \log_{10} n / \log_{10} 1.5 = \log_{10} n / 0.176. \tag{2.64}$$

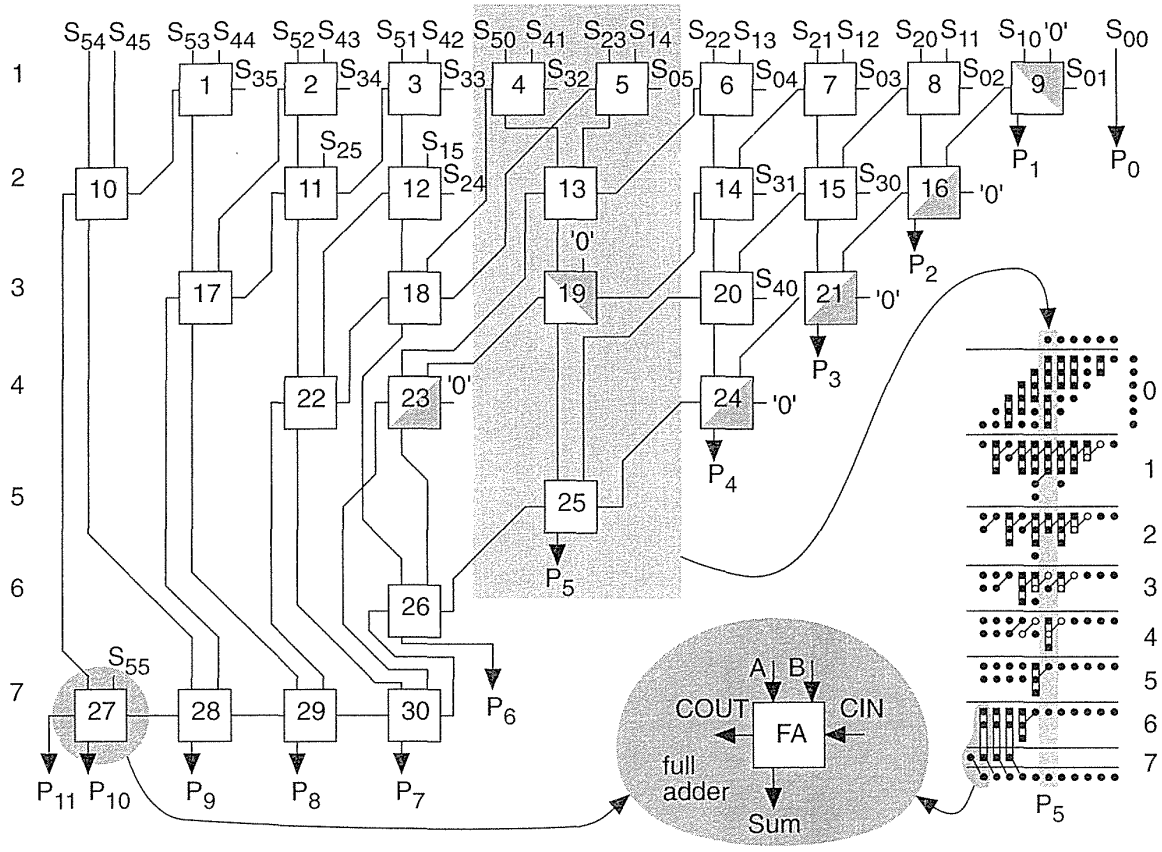


FIGURE 2.29 A 6-bit Wallace-tree multiplier. The carry-save adder (CSA) requires 26 adders (cells 1–26, six are half adders). The final carry-propagate adder (CPA) consists of 4 adder cells (27–30). The delay of the CSA is 6 adders. The delay of the CPA is 4 adders.

Figure 2.31(a) shows how the partial-product array is constructed in a conventional 4-bit multiplier. The **Ferrari–Stefanelli multiplier** (Figure 2.31b) "nests" multipliers—the 2-bit submultipliers reduce the number of partial products [Ferrari and Stefanelli, 1969].

There are several issues in deciding between parallel multiplier architectures:

1. Since it is easier to fold triangles rather than trapezoids into squares, a Wallace-tree multiplier is more suited to full-custom layout, but is slightly larger, than a Dadda multiplier—both are less regular than an array multiplier. For cell-based ASICs, a Dadda multiplier is smaller than a Wallace-tree multiplier.
2. The overall multiplier speed does depend on the size and architecture of the final CPA, but this may be optimized independently of the CSA array. This means a Dadda multiplier is always at least as fast as the Wallace-tree version.

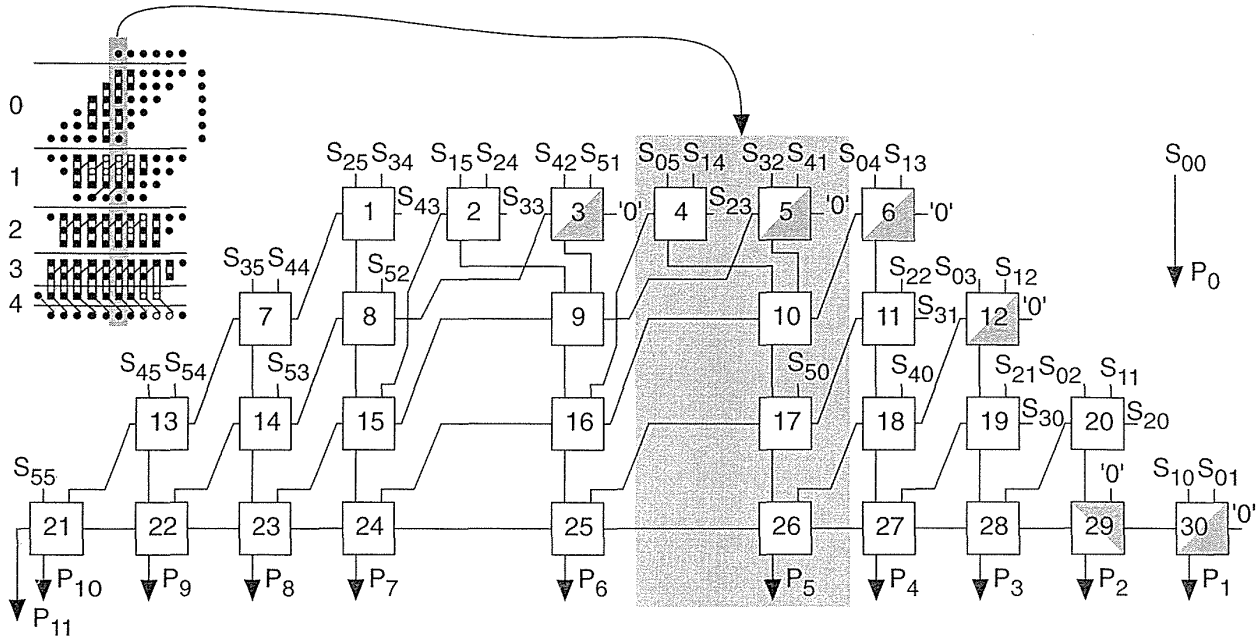


FIGURE 2.30 The 6-bit Dadda multiplier. The carry-save adder (CSA) requires 20 adders (cells 1–20, four are half adders). The carry-propagate adder (CPA, cells 21–30) is a ripple-carry adder (RCA). The CSA is smaller (20 versus 26 adders), faster (3 adder delays versus 6 adder delays), and more regular than the Wallace-tree CSA of Figure 2.29. The overall speed of this implementation is approximately the same as the Wallace-tree multiplier of Figure 2.29; however, the speed may be increased by substituting a faster CPA.

3. The low-order bits of any parallel multiplier settle first and can be added in the CPA before the remaining bits settle. This allows multiplication and the final addition to be overlapped in time.
4. Any of the parallel multiplier architectures may be pipelined. We may also use a **variably pipelined** approach that tailors the register locations to the size of the multiplier.
5. Using (4, 2), (5, 3), (7, 3), or (15, 4) counters increases the stage compression and permits the size of the stages to be tuned. Some ASIC cell libraries contain a (7, 3) counter—a **2-bit full-adder**. A (15, 4) counter is a 3-bit full adder. There is a trade-off in using these counters between the speed and size of the logic cells and the delay as well as area of the interconnect.
6. Power dissipation is reduced by the tree-based structures. The simplified carry-save logic produces fewer signal transitions and the tree structures produce fewer glitches than a chain.

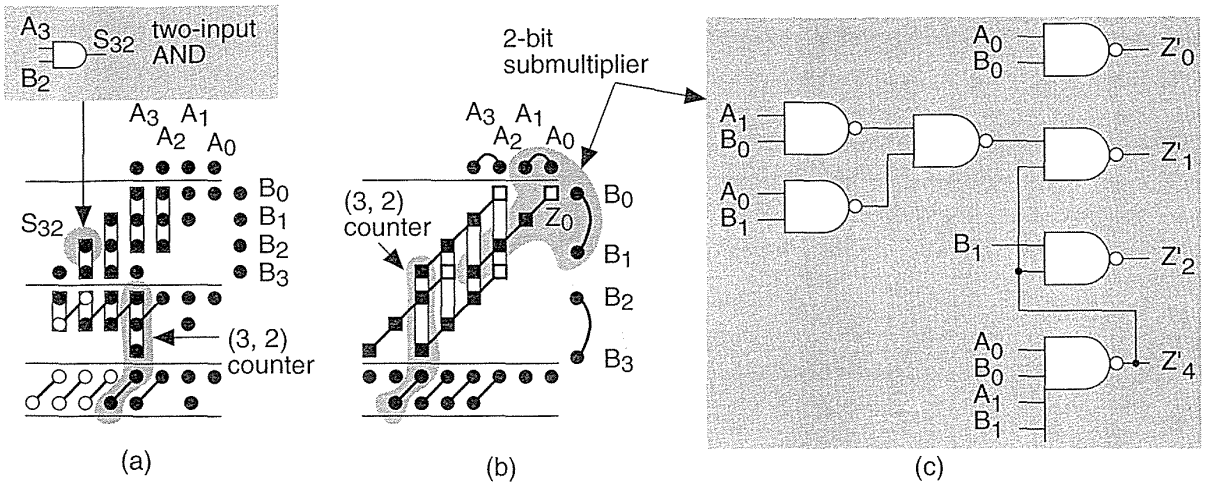


FIGURE 2.31 Ferrari–Stefanelli multiplier. (a) A conventional 4-bit array multiplier using AND gates to calculate the summands with (2, 2) and (3, 2) counters to sum the partial products. (b) A 4-bit Ferrari–Stefanelli multiplier using 2-bit submultipliers to construct the partial product array. (c) A circuit implementation for an inverting 2-bit submultiplier.

7. None of the multiplier structures we have discussed take into account the possibility of staggered arrival times for different bits of the multiplicand or the multiplier. Optimization then requires a logic-synthesis tool.

2.6.5 Other Arithmetic Systems

There are other schemes for addition and multiplication that are useful in special circumstances. Addition of numbers using **redundant binary encoding** avoids carry propagation and is thus potentially very fast. Table 2.13 shows the rules for addition using an intermediate carry and sum that are added without the need for carry. For example,

binary	decimal	redundant binary	CSD vector	
1010111	87	$10\bar{1}0\bar{1}00\bar{1}$	$10\bar{1}0\bar{1}00\bar{1}$	addend
+ 1100101	101	+ $1\bar{1}100111$	+ 01100101	augend
= 10111100	= 188	$0100\bar{1}\bar{1}10$	$\bar{1}\bar{1}00\bar{1}100$	intermediate sum
		$1\bar{1}000101$	11000000	intermediate carry
		= $1\bar{1}1000\bar{1}00$	= $10\bar{1}00\bar{1}100$	sum

TABLE 2.13 Redundant binary addition.

A[*i*]	B[*i*]	A[*i*−1]	B[*i*−1]	Intermediate sum	Intermediate carry
$\bar{1}$	$\bar{1}$	x	x	0	$\bar{1}$
$\bar{1}$	0	A[i−1]=0/1 and B[i−1]=0/1		$\bar{1}$	0
0	$\bar{1}$	A[i−1]=$\bar{1}$ or B[i−1]=$\bar{1}$		1	$\bar{1}$
$\bar{1}$	1	x	x	0	0
1	$\bar{1}$	x	x	0	0
0	0	x	x	0	0
0	1	A[i−1]=0/1 and B[i−1]=0/1		$\bar{1}$	1
1	0	A[i−1]=$\bar{1}$ or B[i−1]=$\bar{1}$		1	0
1	1	x	x	0	1

The redundant binary representation is not unique. We can represent 101 (decimal), for example, by 1100101 (binary and CSD vector) or $1\bar{1}100\bar{1}1\bar{1}$. As another example, 188 (decimal) can be represented by 10111100 (binary), $1\bar{1}1000\bar{1}00$, $10\bar{1}00\bar{1}100$, or $10\bar{1}000\bar{1}00$ (CSD vector). Redundant binary addition of binary, redundant binary, or CSD vectors does not result in a unique sum, and addition of two CSD vectors does not result in a CSD vector. Each n-bit redundant binary number requires a rather wasteful $2n$-bit binary number for storage. Thus $10\bar{1}$ is represented as 010010, for example (using sign magnitude). The other disadvantage of redundant binary arithmetic is the need to convert to and from binary representation.

Table 2.14 shows the (5, 3) **residue number system**. As an example, 11 (decimal) is represented as [1, 2] residue (5, 3) since $11R_5 = 11 \bmod 5 = 1$ and $11R_3 = 11 \bmod 3 = 2$. The size of this system is thus $3 \times 5 = 15$. We add, subtract, or multiply residue numbers using the modulus of each bit position—without any carry. Thus:

```
   4      [4, 1]        12      [2, 0]        3      [3, 0]
+  7   +  [2, 1]     −  4   −  [4, 1]     ×  4   ×  [4, 1]
= 11   =  [1, 2]     =  8   =  [3, 2]     = 12   =  [2, 0]
```

The choice of moduli determines the system size and the computing complexity. The most useful choices are relative primes (such as 3 and 5). With p prime, numbers of the form 2^p and $2^p - 1$ are particularly useful ($2^p - 1$ are **Mersenne's numbers**) [Waser and Flynn, 1982].

2.6.6 Other Datapath Operators

Figure 2.32 shows symbols for some other datapath elements. The combinational datapath cells, NAND, NOR, and so on, and sequential datapath cells (flip-flops and latches) have standard-cell equivalents and function identically. I use a bold outline

TABLE 2.14 The 5, 3 residue number system.

n	residue 5	residue 3	n	residue 5	residue 3	n	residue 5	residue 3
0	0	0	5	0	2	10	0	1
1	1	1	6	1	0	11	1	2
2	2	2	7	2	1	12	2	0
3	3	0	8	3	2	13	3	1
4	4	1	9	4	0	14	4	2

(1 point) for datapath cells instead of the regular (0.5 point) line I use for scalar symbols. We call a set of identical cells a **vector** of datapath elements in the same way that a bold symbol, **A**, represents a vector and A represents a scalar.

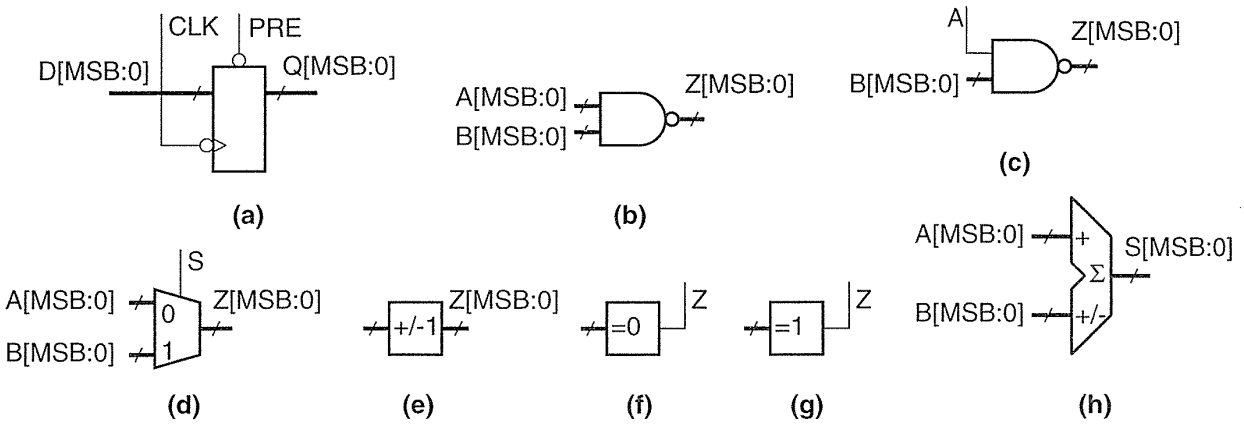


FIGURE 2.32 Symbols for datapath elements. (a) An array or vector of flip-flops (a register). (b) A two-input NAND cell with databus inputs. (c) A two-input NAND cell with a control input. (d) A buswide MUX. (e) An incrementer/decrementer. (f) An all-zeros detector. (g) An all-ones detector. (h) An adder/subtracter.

A **subtracter** is similar to an adder, except in a **full subtracter** we have a borrow-in signal, BIN; a borrow-out signal, BOUT; and a difference signal, DIFF:

$$\text{DIFF} = \text{A} \oplus \text{NOT(B)} \oplus \text{NOT(BIN)} = \text{SUM(A, NOT(B), NOT(BIN))} \quad (2.65)$$

$$\begin{aligned}\text{NOT(BOUT)} &= \text{A} \cdot \text{NOT(B)} + \text{A} \cdot \text{NOT(BIN)} + \text{NOT(B)} \cdot \text{NOT(BIN)} \\ &= \text{MAJ(NOT(A), B, NOT(BIN))} \end{aligned} \quad (2.66)$$

These equations are the same as those for the FA (Eqs. 2.38 and 2.39) except that the B input is inverted and the sense of the carry chain is inverted. To build a subtracter that calculates (A – B) we invert the entire B input bus and connect the BIN[0] input to VDD (not to VSS as we did for CIN[0] in an adder). As an example, to subtract B = '0011' from A = '1001' we calculate '1001' + '1100' + '1' = '0110'. As with an adder, the true overflow is XOR(BOUT[MSB], BOUT[MSB – 1]).

We can build a ripple-borrow subtracter (a type of borrow-propagate subtracter), a borrow-save subtracter, and a borrow-select subtracter in the same way we built these adder architectures. An **adder/subtracter** has a control signal that gates the A input with an exclusive-OR cell (forming a programmable inversion) to switch between an adder or subtracter. Some adder/subtracters gate both inputs to allow us to compute (–A – B). We must be careful to connect the input to the LSB of the carry chain (CIN[0] or BIN[0]) when changing between addition (connect to VSS) and subtraction (connect to VDD).

A **barrel shifter** rotates or shifts an input bus by a specified amount. For example if we have an eight-input barrel shifter with input '1111 0000' and we specify a shift of '0001 0000' (3, coded by bit position) the right-shifted 8-bit output is '0001 1110'. A barrel shifter may rotate left or right (or switch between the two under a separate control). A barrel shifter may also have an output width that is smaller than the input. To use a simple example, we may have an 8-bit input and a 4-bit output. This situation is equivalent to having a barrel shifter with two 4-bit inputs and a 4-bit output. Barrel shifters are used extensively in floating-point arithmetic to align (we call this **normalize** and **denormalize**) floating-point numbers (with sign, exponent, and mantissa).

A **leading-one detector** is used with a normalizing (left-shift) barrel shifter to align mantissas in floating-point numbers. The input is an n-bit bus A, the output is an n-bit bus, S, with a single '1' in the bit position corresponding to the most significant '1' in the input. Thus, for example, if the input is A = '0000 0101' the leading-one detector output is S = '0000 0100', indicating the leading one in A is in bit position 2 (bit 7 is the MSB, bit zero is the LSB). If we feed the output, S, of the leading-one detector to the shift select input of a normalizing (left-shift) barrel shifter, the shifter will normalize the input A. In our example, with an input of A = '0000 0101', and a left-shift of S = '0000 0100', the barrel shifter will shift A left by five bits and the output of the shifter is Z = '1010 0000'. Now that Z is aligned (with the MSB equal to '1') we can multiply Z with another normalized number.

The output of a **priority encoder** is the binary-encoded position of the leading one in an input. For example, with an input A = '0000 0101' the leading 1 is in bit position 3 (MSB is bit position 7) so the output of a 4-bit priority encoder would be Z = '0011' (3). In some cell libraries the encoding is reversed so that the MSB has an output code of zero, in this case Z = '0101' (5). This second, reversed, encoding scheme is useful in floating-point arithmetic. If A is a mantissa and we normalize A to '1010 0000' we have to subtract 5 from the exponent, this **exponent correction** is equal to the output of the priority encoder.

An **accumulator** is an adder/subtracter and a register. Sometimes these are combined with a multiplier to form a **multiplier–accumulator** (**MAC**). An **incrementer** adds 1 to the input bus, Z = A + 1, so we can use this function, together with a register, to negate a two's complement number for example. The implementation is Z[*i*] = XOR(A[*i*], CIN[*i*]), and COUT[*i*] = AND(A[*i*], CIN[*i*]). The carry-in control input, CIN[0], thus acts as an enable: If it is set to '0' the output is the same as the input.

The implementation of arithmetic cells is often a little more complicated than we have explained. CMOS logic is naturally inverting, so that it is faster to implement an incrementer as

$$Z[i(\text{even})] = \text{XOR}(A[i], \text{CIN}[i]) \quad \text{and} \quad \text{COUT}[i(\text{even})] = \text{NAND}(A[i], \text{CIN}[i]).$$

This inverts COUT, so that in the following stage we must invert it again. If we push an inverting bubble to the input CIN we find that:

$$Z[i(\text{odd})] = \text{XNOR}(A[i], \text{CIN}[i]) \quad \text{and} \quad \text{COUT}[i(\text{even})] = \text{NOR}(\text{NOT}(A[i]), \text{CIN}[i]).$$

In many datapath implementations all odd-bit cells operate on inverted carry signals, and thus the odd-bit and even-bit datapath elements are different. In fact, *all* the adder and subtracter datapath elements we have described may use this technique. Normally this is completely hidden from the designer in the datapath assembly and any output control signals are inverted, if necessary, by inserting buffers.

A **decrementer** subtracts 1 from the input bus, the logical implementation is Z[*i*] = XOR(A[*i*], CIN[*i*]) and COUT[*i*] = AND(NOT(A[*i*]), CIN[*i*]). The implementation may invert the odd carry signals, with CIN[0] again acting as an enable.

An **incrementer/decrementer** has a second control input that gates the input, inverting the input to the carry chain. This has the effect of selecting either the increment or decrement function.

Using the **all-zeros detectors** and **all-ones detectors**, remember that, for a 4-bit number, for example, zero in ones' complement arithmetic is '1111' or '0000', and that zero in signed magnitude arithmetic is '1000' or '0000'.

A **register file** (or scratchpad memory) is a bank of flip-flops arranged across the bus; sometimes these have the option of multiple ports (multiport register files) for read and write. Normally these register files are the densest logic and hardest to fit in a datapath. For large register files it may be more appropriate to use a multiport memory. We can add control logic to a register file to create a **first-in first-out register** (**FIFO**), or **last-in first-out register** (**LIFO**).

In Section 2.5 we saw that the standard-cell version and gate-array macro version of the sequential cells (latches and flip-flops) each contain their own clock buffers. The reason for this is that (without intelligent placement software) we do not know where a standard cell or a gate-array macro will be placed on a chip. We also have no idea of the condition of the clock signal coming into a sequential cell. The ability to place the clock buffers outside the sequential cells in a datapath gives us more flexibility and saves space. For example, we can place the clock buffers for all the clocked elements at the top of the datapath (together with the buffers for the con-

trol signals) and **river route** (in river routing the interconnect lines all flow in the same direction on the same layer) the connections to the clock lines. This saves space and allows us to guarantee the clock skew and timing. It may mean, however, that there is a fixed overhead associated with a datapath. For example, it might make no sense to build a 4-bit datapath if the clock and control buffers take up twice the space of the datapath logic. Some tools allow us to design logic using a **portable netlist**. After we complete the design we can decide whether to implement the portable netlist in a datapath, standard cells, or even a gate array, based on area, speed, or power considerations.

2.7 I/O Cells

Figure 2.33 shows a three-state bidirectional output buffer (Tri-State® is a registered trademark of National Semiconductor). When the output enable (OE) signal is high, the circuit functions as a noninverting buffer driving the value of DATAin onto the I/O pad. When OE is low, the output transistors or **drivers**, M1 and M2, are disconnected. This allows multiple drivers to be connected on a bus. It is up to the designer to make sure that a bus never has two drivers—a problem known as **contention**.

In order to prevent the problem opposite to contention—a bus floating to an intermediate voltage when there are no bus drivers—we can use a **bus keeper** or **bus-hold** cell (TI calls this Bus-Friendly logic). A bus keeper normally acts like two weak (low drive-strength) cross-coupled inverters that act as a latch to retain the last logic state on the bus, but the latch is weak enough that it may be driven easily to the opposite state. Even though bus keepers act like latches, and will simulate like latches, they should not be used as latches, since their drive strength is weak.

Transistors M1 and M2 in Figure 2.33 have to drive large off-chip loads. If we wish to change the voltage on a $C = 200\,\text{pF}$ load by 5 V in 5 ns (a **slew rate** of 1 Vns^{-1}) we will require a current in the output transistors of

$$I_{DS} = C\,(dV/dt) = (200 \times 10^{-12})\,(5/5 \times 10^{-9}) = 0.2\,\text{A} \quad \text{or} \quad 200\,\text{mA}.$$

Such large currents flowing in the output transistors must also flow in the power supply bus and can cause problems. There is always some inductance in series with the power supply, between the point at which the supply enters the ASIC package and reaches the power bus on the chip. The inductance is due to the bond wire, lead frame, and package pin. If we have a power-supply inductance of 2 nH and a current changing from zero to 1 A (32 I/O cells on a bus switching at 30 mA each) in 5 ns, we will have a voltage spike on the power supply (called **power-supply bounce**) of $L(\text{d}I/\text{d}t) = (2 \times 10^{-9})(1/(5 \times 10^{-9})) = 0.4\,\text{V}$.

We do several things to alleviate this problem: We can limit the number of **simultaneously switching outputs** (SSOs), we can limit the number of I/O drivers that can be attached to any one VDD and GND pad, and we can design the output buffer to limit the slew rate of the output (we call these slew-rate limited I/O pads).

Quiet-I/O cells also use two separate power supplies and two sets of I/O drivers: an AC supply (clean or quiet supply) with small AC drivers for the I/O circuits that start and stop the output slewing at the beginning and end of a output transition, and a DC supply (noisy or dirty supply) for the transistors that handle large currents as they slew the output.

The three-state buffer allows us to employ the same pad for input and output—**bidirectional I/O**. When we want to use the pad as an input, we set OE low and take the data from DATAin. Of course, it is not necessary to have all these features on every pad: We can build output-only or input-only pads.

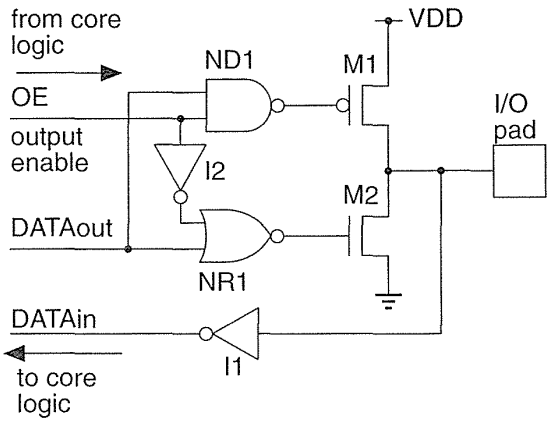


FIGURE 2.33 A three-state bidirectional output buffer. When the output enable, OE, is '1' the output section is enabled and drives the I/O pad. When OE is '0' the output buffer is placed in a high-impedance state.

We can also use many of these output cell features for input cells that have to drive large on-chip loads (a clock pad cell, for example). Some gate arrays simply turn an output buffer around to drive a grid of interconnect that supplies a clock signal internally. With a typical interconnect capacitance of $0.2\,\text{pFcm}^{-1}$, a grid of 100 cm (consisting of 10 by 10 lines running all the way across a 1 cm chip) presents a load of 20 pF to the clock buffer.

Some libraries include I/O cells that have passive pull-ups or pull-downs (resistors) instead of the transistors, M1 and M2 (the resistors are normally still constructed from transistors with long gate lengths). We can also omit one of the driver transistors, M1 or M2, to form **open-drain** outputs that require an external pull-up or pull-down. We can design the output driver to produce TTL output levels rather than CMOS logic levels. We may also add input hysteresis (using a Schmitt trigger) to the input buffer, I1 in Figure 2.33, to accept input data signals that contain glitches (from bouncing switch contacts, for example) or that are slow rising. The input buffer can also include a **level shifter** to accept TTL input levels and shift the input signal to CMOS levels.

The gate oxide in CMOS transistors is extremely thin (100 Å or less). This leaves the gate oxide of the I/O cell input transistors susceptible to breakdown from static electricity (**electrostatic discharge**, or **ESD**). ESD arises when we or machines handle the package leads (like the shock I sometimes get when I touch a

doorknob after walking across the carpet at work). Sometimes this problem is called **electrical overstress** (EOS) since most ESD-related failures are caused not by gate-oxide breakdown, but by the thermal stress (melting) that occurs when the *n*-channel transistor in an output driver overheats (melts) due to the large current that can flow in the drain diffusion connected to a pad during an ESD event.

To protect the I/O cells from ESD, the input pads are normally tied to device structures that clamp the input voltage to below the gate breakdown voltage (which can be as low as 10 V with a 100 Å gate oxide). Some I/O cells use transistors with a special **ESD implant** that increases breakdown voltage and provides protection. I/O driver transistors can also use elongated drain structures (ladder structures) and large drain-to-gate spacing to help limit current, but in a salicide process that lowers the drain resistance this is difficult. One solution is to mask the I/O cells during the salicide step. Another solution is to use *pnpn* and *npnp* diffusion structures called silicon-controlled rectifiers (SCRs) to clamp voltages and divert current to protect the I/O circuits from ESD.

There are several ways to model the capability of an I/O cell to withstand EOS. The **human-body model** (**HBM**) represents ESD by a 100 pF capacitor discharging through a 1.5 kΩ resistor (this is an International Electrotechnical Committee, IEC, specification). Typical voltages generated by the human body are in the range of 2–4 kV, and we often see an I/O pad cell rated by the voltage it can withstand using the HBM. The **machine model** (**MM**) represents an ESD event generated by automated machine handlers. Typical MM parameters use a 200 pF capacitor (typically charged to 200 V) discharged through a 25 Ω resistor, corresponding to a peak initial current of nearly 10 A. The **charge-device model** (**CDM**, also called device charge–discharge) represents the problem when an IC package is charged, in a shipping tube for example, and then grounded. If the maximum charge on a package is 3 nC (a typical measured figure) and the package capacitance to ground is 1.5 pF, we can simulate this event by charging a 1.5 pF capacitor to 2 kV and discharging it through a 1 Ω resistor.

If the diffusion structures in the I/O cells are not designed with care, it is possible to construct an SCR structure unwittingly, and instead of protecting the transistors the SCR can enter a mode where it is latched on and conducting large enough currents to destroy the chip. This failure mode is called **latch-up**. Latch-up can occur if the *pn*-diodes on a chip become forward-biased and inject minority carriers (electrons in *p*-type material, holes in *n*-type material) into the substrate. The source–substrate and drain–substrate diodes can become forward-biased due to power-supply bounce or output **undershoot** (the cell outputs fall below V_{SS}) or **overshoot** (outputs rise to greater than V_{DD}) for example. These injected minority carriers can travel fairly large distances and interact with nearby transistors causing latch-up. I/O cells normally surround the I/O transistors with **guard rings** (a continuous ring of *n*-diffusion in an *n*-well connected to VDD, and a ring of *p*-diffusion in a *p*-well connected to VSS) to collect these minority carriers. This is a problem that can also occur in the logic core and this is one reason that we normally include substrate and well connections to the power supplies in every cell.

2.8 Cell Compilers

The process of hand crafting circuits and layout for a full-custom IC is a tedious, time-consuming, and error-prone task. There are two types of automated layout assembly tools, often known as a **silicon compilers**. The first type produces a specific kind of circuit, a **RAM compiler** or **multiplier compiler**, for example. The second type of compiler is more flexible, usually providing a programming language that assembles or tiles layout from an input command file, but this is full-custom IC design.

We can build a register file from latches or flip-flops, but, at 4.5–6.5 gates (18–26 transistors) per bit, this is an expensive way to build memory. Dynamic RAM (DRAM) can use a cell with only one transistor, storing charge on a capacitor that has to be periodically refreshed as the charge leaks away. ASIC RAM is invariably static (SRAM), so we do not need to refresh the bits. When we refer to RAM in an ASIC environment we almost always mean SRAM. Most ASIC RAMs use a six-transistor cell (four transistors to form two cross-coupled inverters that form the storage loop, and two more transistors to allow us to read from and write to the cell). RAM compilers are available that produce **single-port RAM** (a single shared bus for read and write) as well as **dual-port RAMs**, and **multiport RAMs**. In a multiport RAM the compiler may or may not handle the problem of **address contention** (attempts to read and write to the same RAM address simultaneously). RAM can be **asynchronous** (the read and write cycles are triggered by control and/or address transitions asynchronous to a clock) or **synchronous** (using the system clock).

In addition to producing layout we also need a **model compiler** so that we can verify the circuit at the behavioral level, and we need a netlist from a **netlist compiler** so that we can simulate the circuit and verify that it works correctly at the structural level. Silicon compilers are thus complex pieces of software. We assume that a silicon compiler will produce working silicon even if every configuration has not been tested. This is still ASIC design, but now we are relying on the fact that the tool works correctly and therefore the compiled blocks are **correct by construction**.

2.9 Summary

The most important concepts that we covered in this chapter are the following:

- The use of transistors as switches
- The difference between a flip-flop and a latch
- The meaning of setup time and hold time
- Pipelines and latency
- The difference between datapath, standard-cell, and gate-array logic cells
- Strong and weak logic levels

- Pushing bubbles
- Ratio of logic
- Resistance per square of layers and their relative values in CMOS
- Design rules and λ

2.10 Problems

* = Difficult,** = Very difficult, *** = Extremely difficult

2.1 (Switches, 20 min.) **(a)** Draw a circuit schematic for a two-way light switch: flipping the switch at the top or bottom of the stairs reverses the state of two light bulbs, one at the top and one at the bottom of the stairs. Your schematic should show and label all the cables, switches, and bulbs. **(b)** Repeat the problem for three switches and one light in a warehouse.

2.2 (Logic, 10 min.) The queen wished to choose her successor wisely. She blindfolded and then placed a crown on each of her three children, explaining that there were three red and two blue crowns, and they must deduce the color of their own crown. With blindfolds removed the children could see the two other crowns, but not their own. After a while Anne said: "My crown is red." How did she know?

2.3 (Minus signs, 20 min.) The channel charge in an n-channel transistor is negative. **(a)** Should there not be a minus sign in Eq. 2.5 to account for this? **(b)** If so, then where in the derivation of Section 2.1 does the minus sign disappear to arrive at Eq. 2.9 for the current in an n-channel transistor? **(c)** The equations for the current in a p-channel transistor (Eq. 2.15) have the opposite sign to those for an n-channel transistor. Where in the derivation in Section 2.1 does the extra minus sign arise?

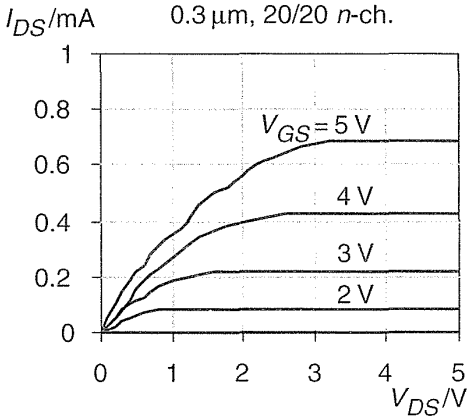


FIGURE 2.34 Transistor characteristics for a 0.3 µm process (Problem 2.4).

2.4 (Transistor curves, 20 min.) Figure 2.34 shows the measured I_{DS}–V_{DS} characteristics for a 20/20 n-channel transistor in a 0.3 μm (effective gate length) process from an ASIC foundry. Derive as much information as you can from this figure.

2.5 (Body effect, 20 min). The equations for the drain–source current (2.9, 2.12, and 2.15) do not contain V_{SB}, the source voltage with respect to the bulk, because we assumed that it was zero. This is not true for the n-channel transistor whose drain is connected to the output in a two-input NAND gate, for example. A reverse **substrate bias** (or back-gate bias; $V_{SB} > 0$ for an n-channel transistor) makes the bulk act like a second gate (the back gate) and modifies an n-channel transistor threshold voltage as follows:

$$V_{tn} = V_{t0n} + \gamma\left(\sqrt{\phi_0 + V_{SB}} - \sqrt{\phi_0}\right), \tag{2.67}$$

where V_{t0n} is measured with $V_{SB} = 0\,\text{V}$; ϕ_0 is called the surface potential; and γ (gamma) is the **body-effect coefficient** (back-gate bias coefficient),

$$\gamma = \frac{\sqrt{2q\varepsilon_{Si}N_A}}{C_{ox}}. \tag{2.68}$$

There are several alternative names and symbols for ϕ_0 ("phi," a positive quantity for an n-channel transistor, typically between 0.6–0.7 V)—you may also see ϕ_b (for bulk potential) or $2\phi_F$ (twice the Fermi potential, a negative quantity). In Eq. 2.68, $\varepsilon_{Si} = \varepsilon_0\varepsilon_r = 1.053 \times 10^{-10}\,\text{Fm}^{-1}$ is the **permittivity of silicon** (the permittivity of a vacuum $\varepsilon_0 = 8.85 \times 10^{-12}\,\text{Fm}^{-1}$ and the relative permittivity of silicon is $\varepsilon_r = 11.7$); N_A is the acceptor doping concentration in the bulk (for p-type substrate or well—N_D for the donor concentration in an n-type substrate or well); and C_{ox} is the gate capacitance per unit area given by

$$C_{ox} = \frac{\varepsilon_{ox}}{T_{ox}}. \tag{2.69}$$

a. Calculate the theoretical value of γ for $N_A = 10^{16}\,\text{cm}^{-3}$, $T_{ox} = 100\,\text{Å}$.

b. Calculate and plot V_{tn} for V_{SB} ranging from 0 V to 5 V in increments of 1 V assuming values of $\gamma = 0.5\,\text{V}^{0.5}$, $\phi_0 = 0.6\,\text{V}$, and $V_{t0n} = 0.5\,\text{V}$ obtained from transistor characteristics.

c. Fit a linear approximation to V_{tn}.

d. Recognizing $V_{SB} \leq 0\,\text{V}$, rewrite Eq. 2.67 for a p-channel device.

e. (Harder) What effect does the back-gate bias effect have on CMOS logic circuits?

Answer: (a) $0.17\,\text{V}^{0.5}$ (b) 0.50–1.3 V.

2.6 (Sizing layout, 10 min.) Stating clearly whatever assumptions you make and describing the tools and methods you use, estimate the size (in λ) of the standard cell shown in Figure 1.3. Estimate the size of each of the transistors, giving their channel lengths and widths (stating clearly which is which).

2.7 (CMOS process) (20 min.) Table 2.15 shows the major steps involved in a typical deep submicron CMOS process. There are approximately 100 major steps in the process.

a. If each major step has a yield of 0.9, what is the overall process yield?

b. If the process yield is 90% (not uncommon), what is the average yield at each major step?

c. If each of the major steps in Table 2.15 consists of an average of five other microtasks, what is the average yield of each of the 500 microtasks.

d. Suppose, for example, an operator loads and unloads a furnace five times a day as a microtask, how many days must the operator work without making a mistake to achieve this microtask yield?

e. Does this seem reasonable? What is wrong with our model?

f. (**60 min.) Draw the process cross-section showing, in particular, the poly, FOX, gate oxide, IMOs and metal layers. You may have to make some assumptions about the meanings and functions of the various steps and layers. Assume all layers are deposited on top of each other according to the thicknesses shown (do not attempt to correct for the silicon consumed during oxidation—even if you understand what this means). The abbreviations in Table 2.15 are as follows: dep. = deposition; LPCVD = low-pressure chemical vapor deposition (for growing oxide and poly); LDD = lightly doped drain (a way to improve transistor characteristics); SOG = silicon overglass (a deposited quartz to help with step coverage between metal layers).

Answer: (a) Zero. (b) 0.999. (c) 0.9998. (d) 3 years.

2.8 (Stipple patterns, 30 min.)

a. Check the stipple patterns in Figure 2.9. Using ruled paper draw 8-by-8 stipple patterns for all the combinations of layers shown.

b. Repeat part a for Figure 2.10.

2.9 (Select, 20 min.) Can you draw a design-rule correct (according to the design rules in Tables 2.7–2.9) layout with a piece of select that has a minimum width of 2λ (rule 4.4)?

2.10 (*Inverter layout, 60 min.) Using 1/4-inch ruled paper (or similar) draw a minimum-size inverter (W/L = 1 for both *p*-channel and *n*-channel transistors). Use a scale of one square to 2λ and the design rules in Table 2.7–Table 2.9. Do not use m2 or m3—only m1. Draw the nwell, pwell, ndiff, and pdiff layers, but not the implant layers or the active layer. Include connections to the input, output, VDD, and VSS in m1. There must be at least one well connection to each well (*n*-well to VDD, and *p*-well to VSS). Minimize the size of your cell BB. Draw the BB outline and write its size in λ^2 on your drawing. Use green diagonal stripes for ndiff, brown diagonal stripes for pdiff, red diagonal stripes for poly, blue diagonal stripes for m1, solid black for contact). Include a key on your drawing, and clearly label the input, output, VDD, and VSS contacts.

TABLE 2.15 CMOS process steps (Problem 2.7).[1]

	Step	Depth		Step	Depth		Step	Depth
1	substrate		32	resist strip		63	m1 mask	
2	oxide 1 dep.	500	33	WSi anneal		64	m1 etch	
3	nitride 1 dep.	1500	34	nLDD mask		65	resist strip	
4	*n*-well mask		35	nLDD implant		66	base oxide dep.	6000
5	*n*-well etch		36	resist strip		67	SOG coat1/2	3000
6	*n*-well implant		37	pLDD mask		68	SOG cure/etch	–4000
7	resist strip		38	pLDD implant		69	cap oxide dep.	4000
8	blocking oxide dep.	2000	39	resist strip		70	via1 mask	
9	nitride 1 strip		40	spacer oxide dep.	3000	71	via1 etch	–2500
10	*p*-well implant		41	WSi anneal		72	resist strip	
11	*p*-well drive		42	SD oxide dep	200	73	TiW dep.	2000
12	active oxide dep.	250	43	n+ mask		74	AlCu/TiW dep.	4000
13	nitride 2 dep.	1500	44	n+ implant		75	m2 mask	
14	active mask		45	resist strip		76	m2 etch	
15	active etch		46	ESD mask		77	resist strip	
16	resist strip		47	ESD implant		78	base oxide dep.	6000
17	field mask		48	resist strip		79	SOG coat 1/2	3000
18	field implant		49	p+ mask		80	SOG cure/etch	–4000
19	resist strip		50	p+ implant		81	cap oxide dep.	4000
20	field oxide dep.	5000	51	resist strip		82	via2 mask	
21	nitride 2 strip		52	implant anneal		83	via2 etch	–2500
22	sacrificial oxide dep.	300	53	LPCVD oxide dep.	1500	84	resist strip	
23	Vt adjust implant		54	BPSG dep./densify	4000	85	TiW dep.	2000
24	gate oxide dep.	80	55	contact mask		86	AlCu/TiW dep.	4000
25	LPCVD poly dep.	1500	56	contact etch	–2500	87	m3 mask	
26	deglaze		57	resist strip		88	m3 etch	
27	WSi dep.	1500	58	Pt dep.	200	89	resist strip	
28	LPCVD oxide dep.	750	59	Pt sinter		90	oxide dep.	4000
29	poly mask		60	Pt strip		92	nitride dep.	10,000
30	oxide etch		61	TiW dep.	2000	93	pad mask	
31	polycide etch		62	AlCu/TiW dep.	4000	94	pad etch	

[1]Depths of layers are in angstroms (negative values are etch depths). For abbreviations used, see Problem 2.7.

2.11 (*AOI221 Layout, 120 min.) Layout the AOI221 shown in Figure 2.13 with the design rules of Tables 2.7–2.9 and using Figure 1.3 as a guide. Label clearly the m1 corresponding to the inputs, output, VDD bus, and GND (VSS) bus. Remember to include substrate contacts. What is the size of your BB in λ^2?

2.12 (Resistance, 20 min.)

a. Using the values for sheet resistance shown in Table 2.3, calculate the resistance of a 200λ long (in the direction of current flow) by 3λ wide piece of each of the layers.

b. Estimate the resistance of an 8-inch, 10 Ω cm, *p*-type, <100> wafer, measured (i) from edge to edge across a diameter and (ii) from face center to the face center on the other side.

2.13 (*Layout graphics, 120 min.) Write a tutorial for capturing layout. As an example:

To capture EPSF (encapsulated PostScript format) from Tanner Research's L-Edit for documentation, Macintosh version... Create a black-and-white technology file, use Setup, Layers..., in L-Edit. The method described here does not work well for grayscale or color. Use File, Print..., Destination check button File to print from L-Edit to an EPS (encapsulated PostScript) file. After you choose Save, a dialog box appears. Select Format: EPS Enhanced Mac Preview, ASCII, Level 1 Compatible, Font Inclusion: None. Save the file. Switch to Frame. Create an Anchored Frame. Use File, Import, File... to bring up a dialog box. Check button Copy into Document, select Format: EPSF. Import the EPS file that will appear as a "page image". Grab the graphic inside the Anchored Frame and move the "page image" around. There will be a footer with text on the "page image" that you may want to hide by using the Anchored Frame edges to crop the image.

Your instructions should be precise, concise, assume nothing, and use the names of menu items, buttons and so on exactly as they appear to the user. Most of the layout figures in this book were created using L-Edit running on a Macintosh, with labels added in FrameMaker. Most of the layouts use the Compass layout editor.

2.14 (Transistor resistance, 20 min.) Calculate I_{DS} and the resistance (the DC value V_{DS}/I_{DS} as well as the AC value $\partial V_{DS}/\partial I_{DS}$ as appropriate) of long-channel transistors with the following parameters, under the specified conditions. In each case state whether the transistor is in the saturation region, linear region, or off:

(i) *n*-channel: $V_{tn} = 0.5$ V, $\beta_n = 40\ \mu AV^{-2}$:

$V_{GS} = 3.3$V: **a.** $V_{DS} = 3.3$ V **b.** $V_{DS} = 0.0$ V **c.** $V_{GS} = 0.0$ V, $V_{DS} = 3.3$ V

(ii) *p*-channel: $V_{tp} = -0.6$ V, $\beta_p = 20\ \mu AV^{-2}$:

$V_{GS} = 0.0$ V: **a.** $V_{DS} = 0.0$ V **b.** $V_{DS} = -5.0$ V **c.** $V_{GS} = -5.0$ V, $V_{DS} = -5.0$ V

2.15 (Circuit theory, 15 min.) You accidentally created the "inverter" shown in Figure 2.35 on a full-custom ASIC currently being fabricated. Will it work? Your manager wants a yes or no answer. Your group is a little more understanding: You are to make a presentation to them to explain the problems ahead. Prepare two foils as well as a one page list of alternatives and recommendations.

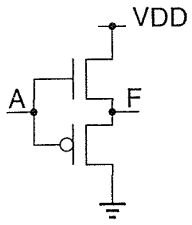


FIGURE 2.35 A CMOS "inverter" with *n*-channel and *p*-channel transistors swapped (Problem 2.15).

2.16 (Mask resolution, 10 min.) People use LaserWriters to make printed-circuit boards all the time.

a. Do you think it is possible to make an IC mask using a 600 dpi (dots per inch) LaserWriter and a transparency?

b. What would λ be?

c. (Harder) See if you can use a microscope to look at the dot and the rectangular bars (serifs) of a letter 'i' from the output of a LaserWriter on paper (most are 300 dpi or 600 dpi). Estimate λ. What is causing the problem? Why is there no rush to generate 1200 dpi LaserWriters for paper? Put a page of this textbook under the microscope: can you see the difference? What are the similar problems printing patterns on a wafer?

2.17 (Lambda, 10 min.) Estimate λ

a. for your TV screen,

b. for your computer monitor,

c. (harder) a photograph.

2.18 (Pass-transistor logic, 10 min.)

a. In Figure 2.36 suppose we set A = B = C = D = '1', what is the value of F?

b. What is the logic strength of the signal at F?

c. If $V_{DD} = 5$ V and $V_{tn} = 0.6$ V, what would the voltage at the source and drain terminals of M1, M2, and M3 be?

d. Will this circuit still work if $V_{DD} = 3$ V?

e. At what point does it stop working?

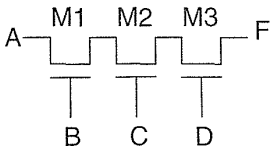


FIGURE 2.36 A pass transistor chain (Problem 2.18).

2.19 (Transistor parameters, 20 min.) Calculate the (**a**) electron and (**b**) hole mobility for the transistor parameters given in Section 2.1 if $k'_n = 80\,\mu AV^{-2}$ and $k'_p = 40\,\mu AV^{-2}$.

Answer: (a) $0.023\,m^2V^{-1}s^{-1}$.

2.20 (Quantum behavior, 10 min.) The average thermal energy of an electron is approximately kT, where $k = 1.38 \times 10^{-23}\,JK^{-1}$ is Boltzmann's constant and T is the absolute temperature in kelvin.

a. The kinetic energy of an electron is $(1/2)mv^2$, where v is due to random thermal motion, and $m = 9.11 \times 10^{-31}$ kg is the rest mass. What is v at 300 K?

b. The electron wavelength $l = h/p$, where $h = 6.62 \times 10^{-34}$ Js is the Planck constant, and $p = mv$ is the electron momentum. What is l at 25 °C?

c. Compare the thermal velocity with the saturation velocity.

d. Compare the electron wavelength with the MOS channel length and with the gate-oxide thickness in a 0.25 μm process and a 0.1 μm process.

2.21 (Gallium arsenide, 5 min.) The electron mobility in GaAs is about $8500\,cm^2V^{-1}s^{-1}$; the hole mobility is about $400\,cm^2V^{-1}s^{-1}$. If we could make complementary n-channel and p-channel GaAs transistors (the same way that we do in a CMOS process) what would the ratio of a GaAs inverter be to equalize rise and fall times? About how much faster would you expect GaAs transistors to be than silicon for the same transistor sizes?

2.22 (Margaret of Anjou, 5 min.)

a. Why is it ones' complement but two's complement?

b. Why Queen's University, Belfast but Queens' College, Cambridge?

2.23 (Logic cell equations, 5 min.) Show that Eq. 2.31, 2.36, and 2.37 are correct.

2.24 (Carry-lookahead equations, 10 min.)

a. Derive the carry-lookahead equations for $i = 8$. Write them in the same form as Eq. 2.56.

b. Derive the equations for the Brent–Kung structure for $i = 8$.

2.25 (OAI cells, 20 min.) Draw a circuit schematic, including transistor sizes, for (**a**) an OAI321 cell, (**b**) an AOI321 cell. (**c**) Which do you think will be larger?

2.26 (**Making stipple patterns) Construct a set of black-and-white, transparent, 8-by-8 stipple patterns for a CMOS process in which we draw both well layers, the active layer, poly, and both diffusion implant layers separately. Consider only the layers up to m1 (but include m1 and the contact layer). One useful tool is the Apple Macintosh Control Panel, 'General Controls,' that changes the Mac desktop pattern.

a. (60 min.) Create a set of patterns with which you can detect any errors (for example, n-well and p-well overlap, or n-implant and p-implant overlap).

b. (60 min.+) Using a layout of an inverter as an example, find a set of patterns that allows you to trace transistors and connections (a very qualitative goal).

c. (Days+) Find a set of grayscale stipple patterns that allow you to produce layouts that "look nice" in a report (much, much harder than it sounds).

2.27 (AOI and OAI cells, 10 min.). Draw the circuit schematics for an AOI22 and an OAI22 cell. Clearly label each transistor as on or off for each cell for an input vector of (A1, A2, B1, B2) = (0101).

2.28 (Flip-flops and latches, 10 min.) In no more than 20 words describe the difference between a flip-flop and a latch.

2.29 (**An old argument) Should setup and hold times appear under maximum, minimum, or typical in a data sheet? (From Peter Alfke.)

2.30 (***Setup, 20 min.) "There is no such thing as a setup and hold time, just two setup times—for a '1' and for a '0'." Comment. (From Clemenz Portmann.)

2.31 (Subtracter, 20 min.) Show that you can rewrite the equations for a full subtracter (Eqs. 2.65–2.66) to be the same as a full adder—except that A is inverted in the borrow out equation, as follows:

$$\text{DIFF} = \text{A} \oplus \text{B} \oplus \text{BIN} = \text{SUM(A, B, BIN)} \tag{2.70}$$

$$\text{BOUT} = \text{NOT(A)} \cdot \text{B} + \text{NOT(A)} \cdot \text{BIN} + \text{B} \cdot \text{BIN} = \text{MAJ(NOT(A), B, CIN)} \tag{2.71}$$

Explain very carefully why we need to connect BIN[0] to VSS. Show that for a subtracter implemented by inverting the B input of an adder and setting CIN[0] = '1', the true overflow for ones' complement or two's complement representations is XOR(CIN[MSB], CIN[MSB – 1]). Does this hold for the above subtracter?

2.32 (Complex CMOS cells) Logic synthesis has completely changed the nature of combinational logic design. Synthesis tools like to see a huge selection of cells from which to choose in order to optimize speed or area.

a. (20 min.) How many AOI*nnnn* cells are there, if the maximum value of $n = 4$?

b. (30 min.) Consider cells of the form AOI*nnnn* where *n* can be negative—indicating a set of inputs are inverted. Thus, an AOI-22 (where the hyphen '-' indicates the following input is inverted) is a NOR(NOR(A, B), AND(C, D)), for example. How many logically different cells of the AOI*xxxx* family are there if *x* can be '-2', '-1', '1', or '2' with no more than four inputs? Remember the AOI family includes OAI, AO, and OA cells as well as just AOI. List them using an extension to the notation for a cell with mixed-sign inputs: for example, an AO(1-1)1 cell is NOT(NOR(AND(A, NOT(B)), C)). *Hint:* Be very careful because some cells with negative inputs are logically equivalent to others with positive inputs.

c. (10 min.) If we include NAND and NOR cells with inverting inputs in a library, how many different cells in the NAND family are there with four or fewer inputs (the NAND family includes NOR, AND, and OR cells)?

d. (30 min.) How many cells in the AOI and NAND families are there with four inputs or less that use fewer than eight transistors? Include cells that are logically equivalent but have different physical implementations. For example, a NAND1-1 cell, requiring six transistors, is logically equivalent to an OR1-1 cell that requires eight transistors. The OR1-1 implementation may be useful because the output inverter can easily be sized to produce an OR1-1 cell with higher drive.

e. (**60 min.) How many cells are there with fewer than four inputs that do not fit into the AOI or NAND families? *Hint:* There is an inverter, a buffer, a half-adder, and the three-input majority function, for example.

f. (***) Recommend a better, user-friendly, naming system (which is also CAD tool compatible) for combinational cells.

2.33 (**Design rules, 60 min.) A typical set of deep submicron CMOS design rules is shown in Table 2.16. Design rules are often confusing and use the following "buzz-words," perhaps to prevent others from understanding them.

The **end cap** is the extension of poly gate beyond the active or diffusion.

Overlap. Normally one material is completely contained within the other, overlap is then the amount of the "surround."

Extension refers to the extension of diffusion beyond the poly gate.

Same (in a spacing rule) means the space to the same type of diffusion or implant.

Opposite refers to the space to the opposite type of diffusion or implant.

A **dogbone** is the area surrounding a contact. Often the spacing to a dogbone contact is allowed be slightly less than to an isolated line.

Field is the area outside the active regions. The field oxide (sandwiched between the diffusion layers and the poly or m1 layers) is thicker than the gate oxide and separates transistors.

Exact refers to contacts that are all the same size to simplify fabrication.

A **butting contact** consists of two adjacent diffusions of the opposite type (connected with metal). This occurs when a well contact is placed next to a source contact.

Fat metal. Some design rules use different spacing for metal lines that are wider than a certain amount.

a. Draw a copy of the MOSIS rules as shown in Figure 2.11, but using the rule numbers and values in microns and λ from Table 2.16.

b. How compatible are the two sets of rules?

2.34 (ESD, 10 min.)

a. Explain carefully why a CMOS device can withstand a 2000 V ESD event when the gate breakdown voltage is only 5–10 V, but that shorting a device pin to a 10 V supply can destroy it.

b. Explain why an electric shock from a 240 VAC supply can kill you, but an 3000 VDC shock from a static charge (walking across a nylon carpet and touching a metal doorknob) only gives you a surprise.

2.35 (*Stacks in CMOS cells, 60 min.)

a. Given a CMOS cell of the form AOIijk or OAIijk ($i, j, k > 0$) derive an equation for the height (the number of transistors in series) and the width (the number of transistor in parallel) of the n-channel and p-channel stacks.

b. Suppose we increase the number of indices to four, i.e. AOI$ijkl$. How do your equations change?

TABLE 2.16 ASIC design rules (Problem 2.33). Absolute values in microns are given for $\lambda = 0.2\,\mu m$.

Layer	Rule[1]	μm	λ	Layer	Rule	μm	λ
nwell	N.1 width	2	10	implant	I.1 width	0.6	3
	N.2 sp. (same)	1	5		I.2 sp. (same)	0.6	3
diff	D.1 width	0.5	2.5		I.3 sp. to diff (same)	0.55	2.75
	D.2 transistor width	0.6	3		I.4 sp. to butting diff	0	0
	D.3 sp. (same)	0.6	3		I.4 ov. of diff	0.25	1.25
	D.4 sp. (opposite)	0.8	4		I.5 sp. to poly on active	0.5	2.5
	D.5 p+ (nwell) to n+ (pwell)	2.4	12		I.6 sp. (opposite)	0.3	1.5
	D.6 nwell ov. of n+	0.6	3		I.7 sp. to butting implant	0	0
	D.7 nwell sp. to p+	0.6	3	contact	C.1 size (exact)	0.4	2
	D.8 extension over gate	0.6	3		C.2 sp.	0.6	3
	D.9 nwell ov. of p+	1.2	6		C.3 poly ov.	0.3	1.5
	D.10 nwell sp. to n+	1.2	6		C.4 diff ov. (2 sides/others)	0.25/0.35	1.25/1.75
poly	P.1 width	0.4	2		C.5 metal ov.	0.25	1.25
	P.2 gate	0.4	2		C.6 sp. to poly	0.3	1.5
	P.3 sp. (over active)	0.6	3		C.7 poly contact to diff	0.5	2.5
	P.4 sp. (over field)	0.5	2.5	m1	Mn.1 width	0.6/0.7/1.0	3/3.5/4
	P.5 short sp. (dogbone)	0.45	2.25	+ m2/m3	Mn.2 sp. (fat > 25 λ is 5 λ)	0.6/0.7/1.0	3/3.5/4
	P.6 end cap	0.45	2.25		Mn.3 sp. (dogbone)	0.5	2.5
	P.7 sp. to diffusion	0.2	1	v1	Vn.1 size (exact)	0.4	2
				+v2/v3	Vn.2 sp.	0.8	4
					Vn.3 metal ov.	0.25	1.25

[1]sp. = space; ov. = overlap; same = same diffusion or implant type; opposite = opposite implant or diffusion type; diff = p+ or n+; p+ = p+ diffusion; n+ = n+ diffusion; implant = p+ or n+ implant select.

c. If the stack height cannot be greater than three, which three-index AOI*ijk* and OAI*ijk* cells are illegal? Often limiting the stack height to three or four is a design rule for **radiation-hard** libraries—useful for satellites.

2.36 (Duals, 20 min.) Draw the *n*-channel stack (including device sizes, assuming a ratio of 2) that complements the *p*-channel stack shown in Figure 2.37.

2.37 (***FPGA conditional-sum adder, days+) A Xilinx application-note (M. Klein, "Conditional sum adder adds 16 bits in 33 ns," Xilinx Application Brief, Xilinx data book, 1992, p. 6-26) describes a 16-bit conditional-sum adder using 41 CLBs in three stages of addition; see also [Sklansky, 1960]. A Xilinx XC3000 or

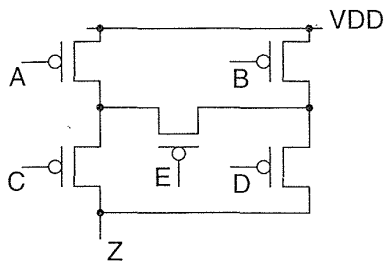


FIGURE 2.37 A *p*-channel stack using a bridge device, E (Problem 2.36).

XC4000 CLB can perform any logic function of five variables, or two functions of (the same) four variables. Can you find a solution with fewer CLBs in three stages? *Hint:* R. P. Halverson of the University of Hawaii produced a solution with 36 CLBs.

2.38 (Encoding, 10min.) Booth's algorithm was suggested by a shortcut used by operators of decimal calculating machines that required turning a handle. To multiply 5 by 23 you set the levers to 5 and turned the handle three times, change gears and turn twice more.

a. What is the equivalent of $1\bar{4}\bar{2}3\bar{4}3$?

b. How many turns do we save using the shortcut?

2.39 (CSD, 20min.)

a. Show how to convert 1010111 (decimal 87) to the CSD vector $10\bar{1}0\bar{1}00\bar{1}$.

b. Convert 1000101 to the CSD vector.

c. How do you know that $1\bar{1}100\bar{1}1\bar{1}$ (decimal 101) is not the CSD vector representation of 1100101 (decimal 101)?

2.11 Bibliography

The topics of this chapter are covered in more detail in Weste and Eshraghian [1993]. The simulator SPICE was developed at UC Berkeley and now has many commercial derivatives including Meta Software's HSPICE and Microsim's PSpice. Mead [1989] gives a description of MOS transistor operation in the subthreshold region of operation. Muller and Kamins provide an introduction to device physics [1977 and 1986]. Sze [1988]; Chang and Sze [1996]; and Campbell [1996] cover process technology in detail at an advanced level. Rabaey [1996] describes full-custom CMOS datapath circuit design, Chandrakasan and Brodersen [1995] describe low-power datapath design. Books by Brodersen [1992] and Gajski [1988] cover silicon compilers. Mukherjee [1986] covers CMOS process and fabrication issues at an introductory level. Texts on analog ASIC design include Haskard and May [1988], and Trontelj [1989]. J. Y. Chen [1990] and Uyemura [1992] provide an analysis of combinational and sequential logic design. The book by Diaz [1995] contains hard to find material on I/O cell design for ESD protection. The patent literature is the

only source for often proprietary high-speed and quiet I/O design. Wakerly [1994] and Katz [1994] are basic references for CMOS logic design (including sequential logic and binary arithmetic) though they emphasize PLDs rather than ASICs. Advanced material on computer arithmetic can be found in books by Hwang [1979]; Waser and Flynn [1982]; Cavanagh [1984]; and C. H. Chen [1992].

A large number of papers on digital arithmetic were published in the 1960s. In ASIC design we work at the architectural level and not at the transistor level and so this early work is useful. Many of these early papers appeared in the *IRE Transactions on Computers* that changed to *IRE Transactions on Electronic Computers* (ISSN 0367-7508, 1963–67) and then to the *IEEE Transactions on Computers* (ISSN 0018-9340, 1967–). A series of important papers on multipliers appeared in *Alta Frequenza* (ISSN 0002-6557, 1932–89; ISSN 1120-1908, 1989–) [Dadda, 1965; Dadda and Ferrari, 1968]. Copies of these papers may be obtained through interlibrary loans (in the United States from Texas A&M library, for example). The two volumes by Swartzlander [1990] contain reprints of some of these articles. Ranganathan [1993] contains reprints of more recent articles. Papers on CMOS logic and arithmetic may be found in the reports of the following conferences: *Proceedings of the Symposium on Computer Arithmetic* (QA76.9.C62.S95a, ISSN 1063-6889), *IEEE International Conference on Computer Design* (TK7888.4.I35a, ISSN 1063-6404), and the *IEEE International Solid-State Circuits Conference* (TK7870.I58; ISSN 0074-8587, 1960-68; ISSN 0193-6530, 1969–). Papers on arithmetic and algorithms that are more theoretical in nature can be found in the *Journal of the Association of Computing Machinery.* Online ACM journal articles can be found at `http://www.acm.org`.

2.12 References

Page numbers in brackets after a reference indicate its location in the chapter body.

Bedrij, O. 1962. "Carry select adder." *IRE Transactions on Electronic Computers,* vol. 11, pp. 340–346. Original reference to carry-select adder. See also [Weste, 1993] p. 532. [p. 83]

Booth, A. 1951. "A signed binary multiplication technique." *Quarterly Journal of Mechanics and Applied Mathematics,* vol. 4, pt. 2, pp. 236–240. Original reference for the Booth-encoded multiplier. See also Swartzlander [1990] and Weste [1993, pp. 547–554]. [p. 90]

Brent, R., and H. T. Kung. 1982. "A regular layout for parallel adders." *IEEE Transactions on Computers,* vol. 31, no. 3, pp. 260–264. Describes a regular carry-lookahead adder. [p. 83]

Brodersen, R. (Ed.). 1992. *Anatomy of a Silicon Compiler.* Boston: Kluwer, 362 p. ISBN 0-7923-9249-3. TK7874.A59.

Campbell, S. 1996. *The Science and Engineering of Microelectronic Fabrication.* New York: Oxford University Press, 536 p. ISBN 0-19-510508-7. TK7871.85.C25. [p. 113]

Cavanagh, J. J. F. 1984. *Digital Computer Arithmetic Design and Implementation.* New York: McGraw-Hill, 468 p. QA76.9.C62.C38. ISBN 0070102821.

Chandrakasan A. P., and R. Brodersen. 1995. *Low Power Digital CMOS Design.* Boston: Kluwer, 424 p. ISBN 0-7923-9576-X. TK7871.99.M44C43.

Chang, C. Y., and S. M. Sze. 1996. *ULSI Technology.* New York: McGraw-Hill, 726 p. ISBN 0070630623.

Chen, C. H. (Ed.). 1992. *Computer Engineering Handbook.* New York: McGraw-Hill. ISBN 0-07-010924-9. TK7888.3.C652. Chapter 4, "Computer arithmetic," by E. E. Swartzlander, pp. 20, contains descriptions of adder, multiplier, and divider architectures.

Chen, J. Y. 1990. *CMOS Devices and Technology for VLSI.* Englewood Cliffs, NJ: Prentice-Hall, 348 p. ISBN 0-13-138082-6. TK7874.C523.

Dadda, L. 1965. "Some schemes for parallel multipliers." *Alta Frequenza,* vol. 34, pp. 349–356. The original reference to the Dadda multiplier. This paper contains some errors in the diagrams for the multipliers; some remain in the reprint in Swartzlander [1990, vol. 1]. See also sequel papers: L. Dadda and D. Ferrari, "Digital multipliers: a unified approach," *Alta Frequenza,* vol. 37, pp. 1079–1086, 1968; and L. Dadda, "On parallel digital multipliers," *Alta Frequenza,* vol. 45, pp. 574–580, 1976. [p. 90]

Denyer, P. B., and D. Renshaw. 1985. *VLSI Signal Processing: A Bit-Serial Approach.* Reading, MA: Addison-Wesley, 312 p. ISBN 0201144042. TK7874.D46. See also P. B. Denyer and S. G. Smith, *Serial-Data Computation.* Boston: Kluwer, 1988, 239 p. ISBN 089838253X. TK7874.S623. [p. 86]

Diaz, C. H., et al. 1995. *Modeling of Electrical Overstress in Integrated Circuits.* Norwell, MA: Kluwer Academic, 148 p. ISBN 0-7923-9505-0. TK7874.D498. Includes 101 references. Introduction to ESD problems and models.

Ferrari, D., and R. Stefanelli. 1969. "Some new schemes for parallel multipliers." *Alta Frequenza,* vol. 38, pp. 843–852. The original reference for the Ferrari–Stefanelli multiplier. Describes the use of 2-bit and 3-bit submultipliers to generate the product array. Contains tables showing the number of stages and delay for different configurations. [p. 92]

Gajski, D. D. (Ed.). 1988. *Silicon Compilation.* Reading, MA: Addison-Wesley, 450 p. ISBN 0-201-109915-2. TK7874.S52.

Goldberg, D. 1990. "Computer arithmetic." In D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach.* San Mateo, CA: Morgan Kaufmann, 2nd ed., 1995. QA76.9.A73. P377. ISBN 1-55860-329-8. See also the first edition of this book (1990).

Haskard, M. R., and I. C. May. 1988. *Analog VLSI Design: nMOS and CMOS.* Englewood Cliffs, NJ: Prentice-Hall, 243 p. ISBN 0-13-032640-2. TK7874.H392.

Hwang, K. 1979. *Computer Arithmetic: Principles, Architecture, and Design.* New York: Wiley, 423 p. ISBN 0471034967. TK7888.3.H9.

Katz, R. H., 1994. *Contemporary Logic Design.* Reading, MA: Addison-Wesley, 699 p. ISBN 0-8053-2703-7.

Keutzer, K., S. Malik, and A. Saldanha. 1991. "Is redundancy necessary to reduce delay?" *IEEE Transactions on Computer-Aided Design,* vol. 10, no. 4, pp. 427–435. Describes the carry-skip adder. The paper describes the redundant logic that is added in a carry-skip adder and how to remove it without changing the function or delay of the circuit. [p. 82]

Lehman, M., and N. Burla. 1961. "Skip techniques for high-speed carry-propagation in binary arithmetic units." *IRE Transactions on Electronic Computers,* vol. 10, pp. 691–698. Original reference to carry-skip adder. [p. 82]

MacSorley, O. L. 1961. "High speed arithmetic in binary computers." *IRE Proceedings,* vol. 49, pp. 67–91. Early reference to carry-lookahead adder. Reprinted in Swartzlander [1990, vol. 1]. See also Weste [1993, pp. 526–529]. [p. 83]

Mead, C. A. 1989. *Analog VLSI and Neural Systems.* Reading, MA: Addison-Wesley, p.371. ISBN 0-201-05992-4. QA76.5.M39. Includes a description of MOS device operation.

Muller, R. S., and T. I. Kamins. 1977. *Device Electronics for Integrated Circuits.* New York: Wiley, p. 404. ISBN 0-471-62364-4. TK7871.85.M86. See also the second edition of this book (1986).

Mukherjee, A. 1986. *Introduction to nMOS and CMOS VLSI Systems Design.* Englewood Cliffs, NJ: Prentice-Hall, 370 p. ISBN 0-13-490947-X. TK7874.M86.

Rabaey, J. 1996. *Digital Integrated Circuits: A Design Perspective.* Englewood Cliffs, NJ: Prentice-Hall, pp. 700. ISBN 0-13-178609-1. TK7874.65.R33. Chapters 4 and 7 describe the design of full-custom CMOS datapath circuits.

Ranganathan, N. (Ed.). 1993. *VLSI Algorithms and Architectures: Fundamentals.* New York: IEEE Press, 305 p. ISBN 0-8186-4390-0. TK7874.V5554. See also N. Ranganathan (Ed.), 1993. *VLSI Algorithms and Architectures: Advanced Concepts.* New York: IEEE Press, 303 p. ISBN 0-8186-4400-1. TK7874.V555. Collections of articles mostly from *Computer* and *IEEE Transactions on Computers.*

Sato, T., et al. 1992. "An 8.5 ns 112-b transmission gate adder with a conflict-free bypass circuit." *IEEE Journal of Solid-State Circuits,* vol. 27, no. 4, pp. 657–659. Describes an implementation of a carry-bypass adder. [p. 81]

Sklansky, J. 1960. "Conditional-sum addition logic." *IRE Transactions on Electronic Computers,* vol. 9, pp. 226–231. Original reference to conditional-sum adder. Several texts have propagated an error in the spelling of Sklansky (two k's). See also [Weste, 1993] pp. 532–533; A. Rothermel et al., "Realization of transmission-gate conditional-sum (TGCS) adders with low latency time," *IEEE Journal of Solid-State Circuits,* vol. 24, no. 3, 1989, pp. 558–561; each of these are examples of adders based on Sklansky's design. [p. 83]

Swartzlander, E. E., Jr. 1990. *Computer Arithmetic.* Los Alamitos, CA: IEEE Computer Society Press, vols. 1 and 2. ISBN 0818689315 (vol. 1). QA76.6.C633. Volume 1 is a reprint (originally published: Stroudsberg, PA: Dowden, Hutchinson & Ross). Volume 2 is a sequel. Contains reprints of many of the early (1960–1970) journal articles on adder and multiplier architectures.

Sze, S. (Ed.). 1988. *VLSI Technology.* New York: McGraw-Hill, 676 p. ISBN 0-07-062735-5. TK7874.V566. Edited book on fabrication technology.

Trontelj, J., et al. 1989. *Analog Digital ASIC Design.* New York: McGraw-Hill, 249 p. ISBN 0-07-707300-2. TK7874.T76.

Uyemura, J. P. 1992. *Circuit Design for CMOS VLSI.* Boston: Kluwer, 450 p. ISBN 0-7923-9184-5. TK7874.U93. See also: J. P. Uyemura, 1988, *Fundamentals of MOS Digital Integrated Circuits,* Reading, MA: Addison-Wesley, 624 p. ISBN 0-201-13318-0. TK7874.U94. Includes basic circuit equations related to NMOS and CMOS logic design.

Wakerly, J. F. 1994. *Digital Design: Principles and Practices.* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 840 p. ISBN 0-13-211459-3. TK7874.65.W34. Undergraduate level introduction to logic design covering: binary arithmetic, CMOS and TTL, combinational logic, PLDs, sequential logic, memory, and the IEEE standard logic symbols.

Wallace, C. S. 1960. "A suggestion for a fast multiplier." *IEEE Transactions on Electronic Computers,* vol. 13, pp. 14–17. Original reference to Wallace-tree multiplier. Reprinted in Swartzlander [1990, vol. 1]. [p. 90]

Waser, S., and M. J. Flynn. 1982. *Introduction to Arithmetic for Digital Systems Designers.* New York: Holt, Rinehart, and Winston, 308 p. ISBN 0030605717. TK7895.A65.W37. [p. 114]

Weste, N. H. E., and K. Eshraghian. 1993. *Principles of CMOS VLSI Design: A Systems Perspective.* 2nd ed. Reading, MA: Addison-Wesley, 713 p. ISBN 0-201-53376-6. TK7874.W46. Chapter 5 covers CMOS logic gate design. Chapter 8 covers datapath elements. See also the first edition of this book. [p. 81]

ASIC LIBRARY DESIGN

3

Once we have decided to use an ASIC design style—using predefined and precharacterized cells from a library—we need to design or buy a cell library. Even though it is not necessary a knowledge of ASIC library design makes it easier to use library cells effectively.

3.1 Transistors as Resistors

In Section 2.1, "CMOS Transistors," we modeled transistors using ideal switches. If this model were accurate, logic cells would have no delay.

The ramp input, `v(in1)`, to the inverter in Figure 3.1(a) rises quickly from zero to V_{DD}. In response the output, `v(out1)`, falls from V_{DD} to zero. In Figure 3.1(b) we measure the **propagation delay** of the inverter, t_{PD}, using an input trip point of 0.5 and output trip points of 0.35 (falling, t_{PDf}) and 0.65 (rising, t_{PDr}). Initially the n-channel transistor, `m1`, is *off*. As the input rises, `m1` turns *on* in the saturation region ($V_{DS} > V_{GS} - V_{tn}$) before entering the linear region ($V_{DS} < V_{GS} - V_{tn}$). We model transistor `m1` with a resistor, R_{pd} (Figure 3.1c); this is the **pull-down resistance**. The equivalent resistance of `m2` is the **pull-up resistance**, R_{pu}.

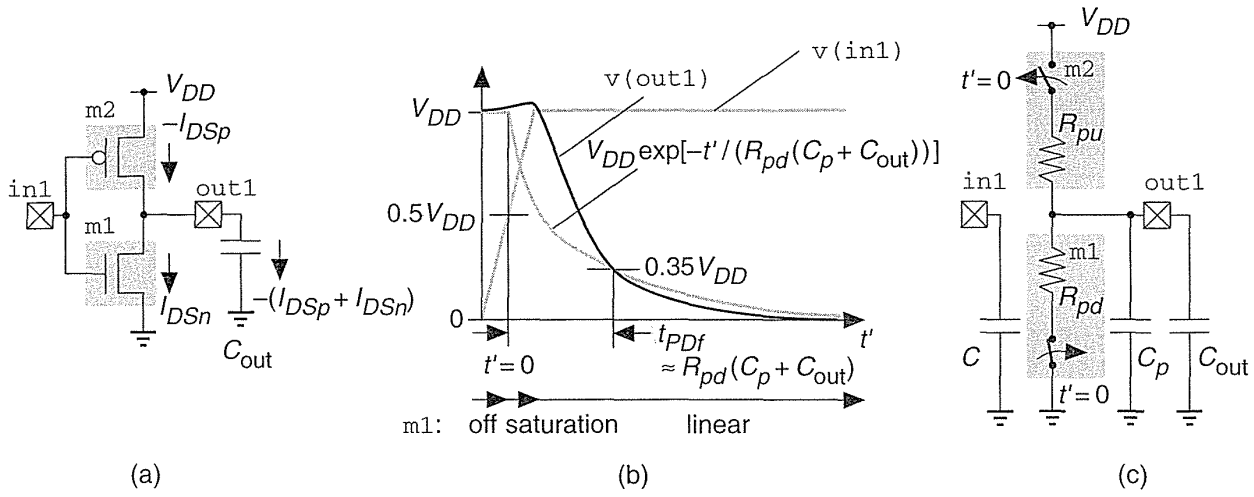


FIGURE 3.1 A model for CMOS logic delay. (a) A CMOS inverter with a load capacitance, C_{out}. (b) Input, `v(in1)`, and output, `v(out1)`, waveforms showing the definition of the falling propagation delay, t_{PDf}. In this case delay is measured from the input trip point of 0.5. The output trip points are 0.35 (falling) and 0.65 (rising). The model predicts $t_{PDf} \approx R_{pd}(C_p + C_{out})$. (c) The model for the inverter includes: the input capacitance, C; the pull-up resistance (R_{pu}) and pull-down resistance (R_{pd}); and the parasitic output capacitance, C_p.

Delay is created by the pull-up and pull-down resistances, R_{pd} and R_{pu}, together with the parasitic capacitance at the output of the cell, C_p (the **intrinsic output capacitance**) and the **load capacitance** (or **extrinsic output capacitance**), C_{out} (Figure 3.1c). If we assume a constant value for R_{pd}, the output reaches a lower trip point of 0.35 when (Figure 3.1b),

$$0.35V_{DD} = V_{DD}\exp\left[\frac{-t_{PDf}}{R_{pd}(C_{\text{out}} + C_p)}\right]. \tag{3.1}$$

An output trip point of 0.35 is convenient because $\ln(1/0.35) = 1.04 \approx 1$ and thus

$$t_{PDf} = R_{pd}(C_{\text{out}} + C_p)\ln\left(\frac{1}{0.35}\right) \approx R_{pd}(C_{\text{out}} + C_p). \tag{3.2}$$

The expression for the rising delay (with a 0.65 output trip point) is identical in form. Delay thus increases linearly with the load capacitance. We often measure load capacitance in terms of a **standard load**—the input capacitance presented by a particular cell (often an inverter or two-input NAND cell).

We may adjust the delay for different trip points. For example, for output trip points of 0.1/0.9 we multiply Eq. 3.2 by $-\ln(0.1) = 2.3$, because $\exp(-2.3) = 0.100$.

Figure 3.2 shows the DC characteristics of a CMOS inverter. To form Figure 3.2(b) we take the n-channel transistor surface (Figure 2.4b) and add that for a p-channel transistor (rotated to account for the connections). Seen from above, the intersection of the two surfaces is the static transfer curve of Figure 3.2(a)—along this path the transistor currents are equal and there is no output current to change the output voltage. Seen from one side, the intersection is the curve of Figure 3.2(c).

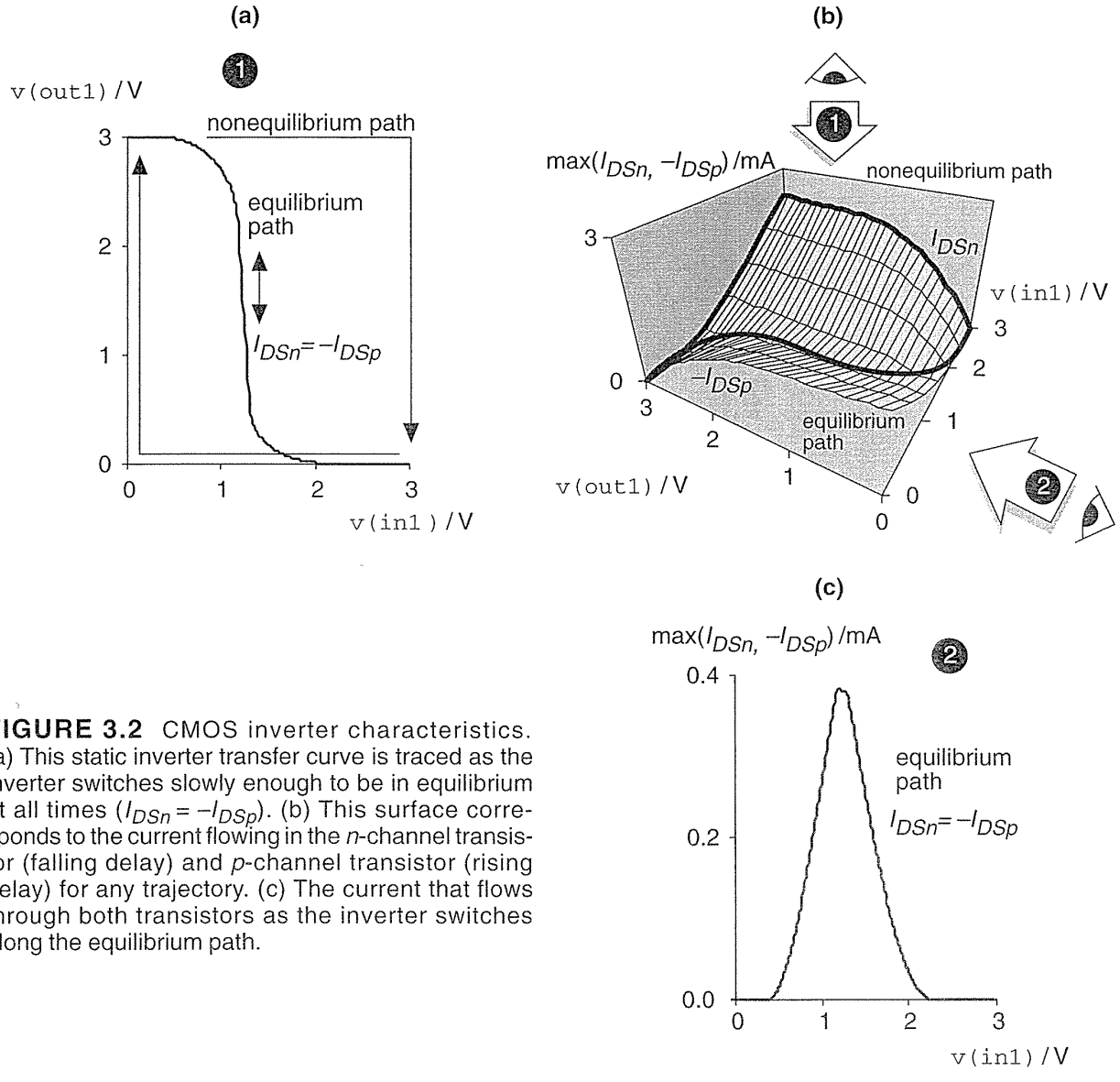


FIGURE 3.2 CMOS inverter characteristics. (a) This static inverter transfer curve is traced as the inverter switches slowly enough to be in equilibrium at all times ($I_{DSn} = -I_{DSp}$). (b) This surface corresponds to the current flowing in the n-channel transistor (falling delay) and p-channel transistor (rising delay) for any trajectory. (c) The current that flows through both transistors as the inverter switches along the equilibrium path.

The input waveform, `v(in1)`, and the output load (which determines the transistor currents) dictate the path we take on the surface of Figure 3.2(b) as the inverter switches. We can thus see that the currents through the transistors (and thus the pull-up and pull-down resistance values) will vary in a nonlinear way during switching. Deriving theoretical values for the pull-up and pull-down resistance values is difficult—instead we work the problem backward by picking the trip points, simulating the propagation delays, and then calculating resistance values that fit the model.

Figure 3.3 shows a simulation experiment (using the G5 process SPICE parameters from Table 2.1). From the results in Figure 3.3(c) we can see that $R_{pd} = 817\ \Omega$ and $R_{pu} = 1281\ \Omega$ for this inverter (with shape factors of 6/0.6 for the n-channel transistor and 12/0.6 for the p-channel) using 0.5 (input) and 0.35/0.65 (output) trip points. Changing the trip points would give different resistance values.

We can check that $817\ \Omega$ is a reasonable value for the pull-down resistance. In the saturation region $I_{DS(\text{sat})}$ is (to first order) independent of V_{DS}. For an n-channel transistor from our generic 0.5 μm process (G5 from Section 2.1) with shape factor W/L = 6/0.6, $I_{DSn(\text{sat})} = 2.5\,\text{mA}$ (at $V_{GS} = 3\text{V}$ and $V_{DS} = 3\text{V}$). The pull-down resistance, R_1, that would give the same drain–source current is

$$R_1 = 3.0\text{V}/(2.5 \times 10^{-3}\text{A}) = 1200\ \Omega. \tag{3.3}$$

This value is greater than, but not too different from, our measured pull-down resistance of $817\ \Omega$. We might expect this result since Figure 3.2b shows that the pull-down resistance reaches its maximum value at $V_{GS} = 3\text{V}$, $V_{DS} = 3\text{V}$. We could adjust the ratio of the logic so that the rising and falling delays were equal; then $R = R_{pd} = R_{pu}$ is the **pull resistance**.

Next, we check our model against the simulation results. The model predicts

$$\texttt{v(out1)} \approx V_{DD} \exp \frac{-t'}{R_{pd}(C_{\text{out}} + C_p)} \quad \text{for } t' > 0 \tag{3.4}$$

(t' is measured from the point at which the input crosses the 0.5 trip point, $t' = 0$ at $t = 20$ ps). With $C_p = 4$ standard loads $= 4 \times 0.034\,\text{pF} = 0.136\,\text{pF}$,

$$R_{pd}(C_{\text{out}} + C_p) = (38 + 817\,(0.136))\,\text{ps} = 149.112\ \text{ps}. \tag{3.5}$$

To make a comparison with the simulation we need to use $\ln(1/0.35) = 1.04$ and not approximately 1 as we have assumed, so that (with all times in ps)

$$\texttt{v(out1)} \approx 3.0 \exp\left[\frac{-t'}{149.112/1.04}\right]\text{V} = 3.0 \exp\left[\frac{-(t-20)}{143.4}\right]\text{V} \quad \text{for } t > 20\ \text{ps}. \tag{3.6}$$

Equation 3.6 is plotted in Figure 3.3(d). For `v(out1)` = 1.05 V (equal to the 0.35 output trip point), Eq. 3.6 predicts $t = 20 + 149.112 \approx 169$ ps and agrees with Figure 3.3(b)—it should because we derived the model from these results!

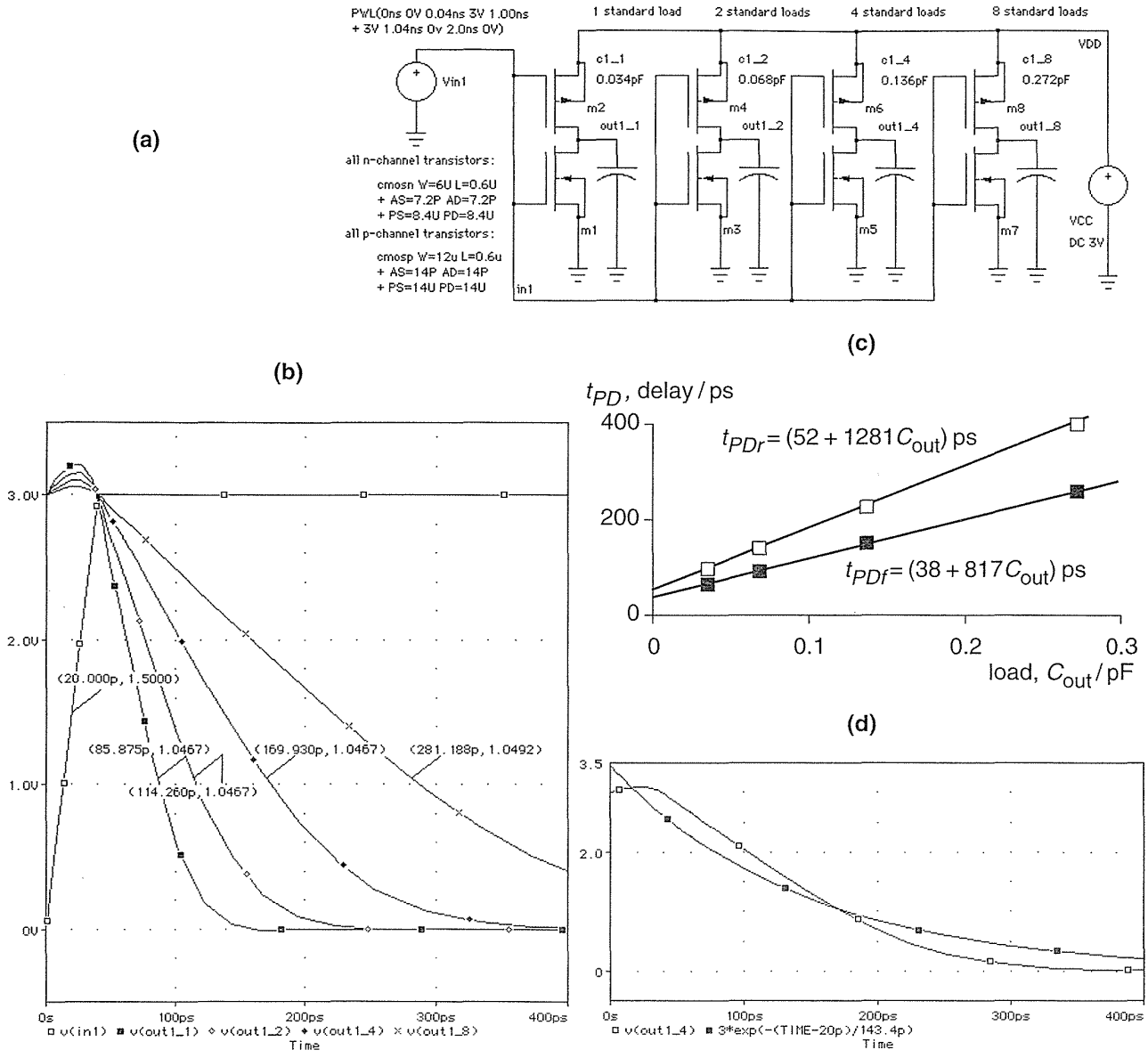


FIGURE 3.3 Delay. (a) LogicWorks schematic for inverters driving 1, 2, 4, and 8 standard loads (1 standard load = 0.034 pF in this case). (b) Transient response (falling delay only) from PSpice. The postprocessor Probe was used to mark each waveform as it crosses its trip point (0.5 for the input, 0.35 for the outputs). For example `v(out1_4)` (4 standard loads) crosses 1.0467 V ($\approx 0.35 V_{DD}$) at $t = 169.93$ ps. (c) Falling and rising delays as a function of load. The slopes in pspF^{-1} corresponds to the pull-up resistance (1281 Ω) and pull-down resistance (817 Ω). (d) Comparison of the delay model (valid for $t > 20$ ps) and simulation (4 standard loads). Both are equal at the 0.35 trip point.

Now we find C_p. From Figure 3.3(c) and Eq. 3.2

$$\begin{aligned} t_{PDr} &= (52 + 1281C_{\text{out}})\ \text{ps} \Rightarrow C_{pr} = 52/1281 = 0.041\text{pF} \quad \text{(rising)} \\ t_{PDf} &= (38 + 817C_{\text{out}})\ \text{ps} \Rightarrow C_{pf} = 38/817 = 0.047\text{pF} \quad \text{(falling)} \end{aligned} \tag{3.7}$$

These intrinsic parasitic capacitance values depend on the choice of output trip points, even though $C_{pf}R_{pdf}$ and $C_{pr}R_{pdr}$ are constant for a given input trip point and waveform, because the pull-up and pull-down resistances depend on the choice of output trip points. We take a closer look at parasitic capacitance next.

3.2 Transistor Parasitic Capacitance

Logic-cell delay results from transistor resistance, transistor (intrinsic) parasitic capacitance, and load (extrinsic) capacitance. When one logic cell drives another, the parasitic input capacitance of the driven cell becomes the load capacitance of the driving cell and this will determine the delay of the driving cell.

Figure 3.4 shows the components of transistor parasitic capacitance. SPICE prints all of the MOS parameter values for each transistor at the DC operating point. The following values were printed by PSpice (v5.4) for the simulation of Figure 3.3:

NAME	m1	m2
MODEL	CMOSN	CMOSP
ID	7.49E-11	-7.49E-11
VGS	0.00E+00	-3.00E+00
VDS	3.00E+00	-4.40E-08
VBS	0.00E+00	0.00E+00
VTH	4.14E-01	-8.96E-01
VDSAT	3.51E-02	-1.78E+00
GM	1.75E-09	2.52E-11
GDS	1.24E-10	1.72E-03
GMB	6.02E-10	7.02E-12
CBD	2.06E-15	1.71E-14
CBS	4.45E-15	1.71E-14
CGSOV	1.80E-15	2.88E-15
CGDOV	1.80E-15	2.88E-15
CGBOV	2.00E-16	2.01E-16
CGS	0.00E+00	1.10E-14
CGD	0.00E+00	1.10E-14
CGB	3.88E-15	0.00E+00

The parameters ID (I_{DS}), VGS, VDS, VBS, VTH (V_t), and VDSAT ($V_{DS(\text{sat})}$) are DC parameters. The parameters GM, GDS, and GMB are small-signal conductances (corresponding to $\partial I_{DS}/\partial V_{GS}$, $\partial I_{DS}/\partial V_{DS}$, and $\partial I_{DS}/\partial V_{BS}$, respectively). The

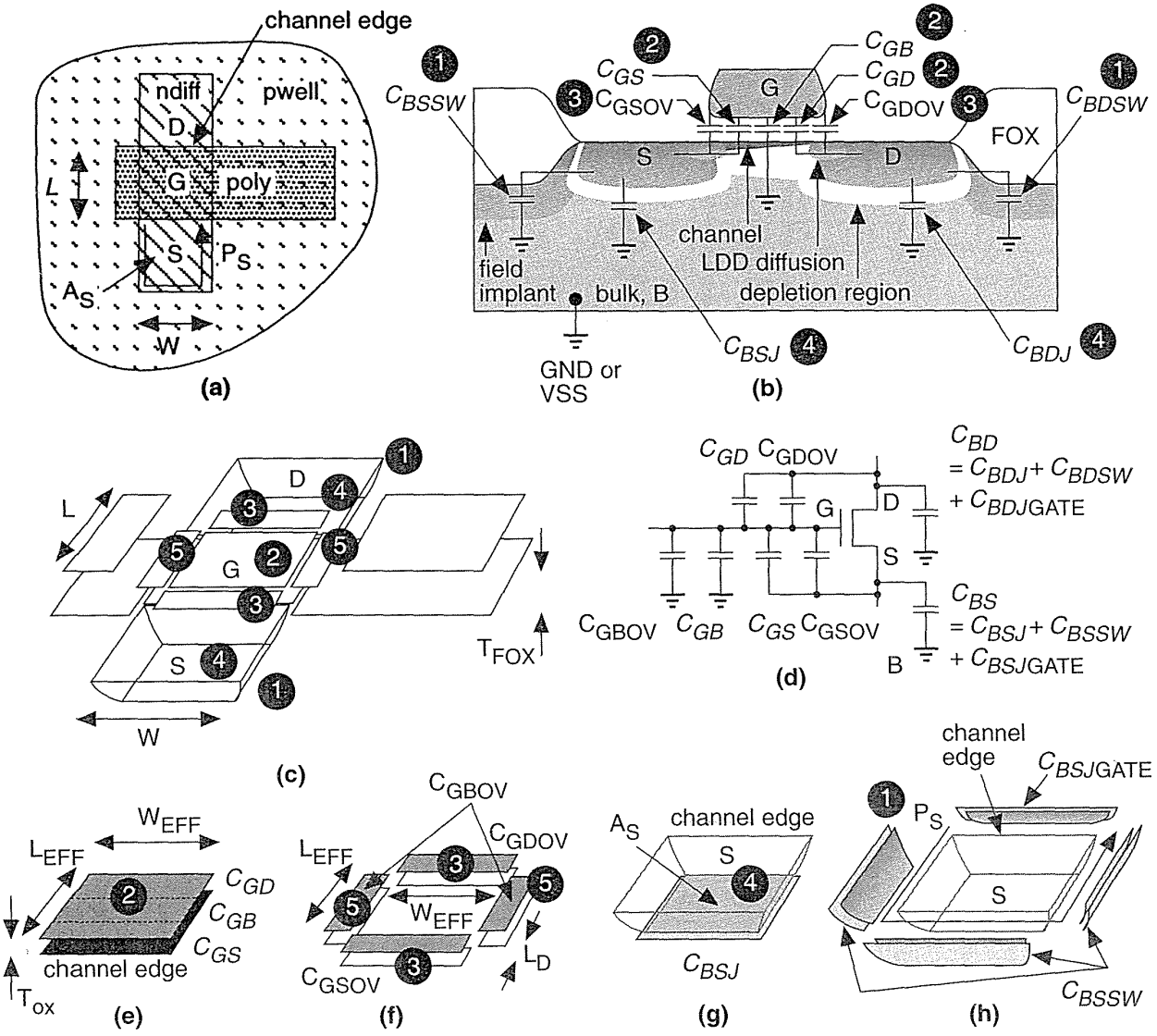


FIGURE 3.4 Transistor parasitic capacitance. (a) An *n*-channel MOS transistor with (drawn) gate length L and width W. (b) The gate capacitance is split into: the constant overlap capacitances C_{GSOV}, C_{GDOV}, and C_{GBOV} and the variable capacitances C_{GS}, C_{GB}, and C_{GD}, which depend on the operating region. (c) A view showing how the different capacitances are approximated by planar components (T_{FOX} is the field-oxide thickness). (d) C_{BS} and C_{BD} are the sum of the area (C_{BSJ}, C_{BDJ}), sidewall (C_{BSSW}, C_{BDSW}), and channel edge ($C_{BSJGATE}$, $C_{BDJGATE}$) capacitances. (e)–(f) The dimensions of the gate, overlap, and sidewall capacitances (L_D is the lateral diffusion).

remaining parameters are the parasitic capacitances. Table 3.1 shows the calculation of these capacitance values for the *n*-channel transistor `m1` (with $W = 6\,\mu m$ and $L = 0.6\,\mu m$) in Figure 3.3(a).

3.2.1 Junction Capacitance

The junction capacitances, C_{BD} and C_{BS}, consist of two parts: junction area and sidewall; both have different physical characteristics with parameters: `CJ` and `MJ` for the junction, `CJSW` and `MJSW` for the sidewall, and `PB` is common. These capacitances depend on the voltage across the junction (V_{DB} and V_{SB}). The calculations in Table 3.1 assume both source and drain regions are $6\,\mu m \times 1.2\,\mu m$ rectangles, so that $A_D = A_S = 7.2\,(\mu m)^2$, and the perimeters (excluding the $1.2\,\mu m$ channel edge) are $P_D = P_S = 6 + 1.2 + 1.2 = 8.4\,\mu m$. We exclude the channel edge because the sidewalls facing the channel (corresponding to $C_{BSJGATE}$ and $C_{BDJGATE}$ in Figure 3.4) are different from the sidewalls that face the field. There is no standard method to allow for this. It is a mistake to exclude the gate edge assuming it is accounted for in the rest of the model—it is not. A pessimistic simulation includes the channel edge in P_D and P_S (but a true worst-case analysis would use more accurate models and worst-case model parameters). In HSPICE there is a separate mechanism to account for the channel edge capacitance (using parameters `ACM` and `CJGATE`). In Table 3.1 we have neglected C_{JGATE}.

For the *p*-channel transistor `m2` ($W = 12\,\mu m$ and $L = 0.6\,\mu m$) the source and drain regions are $12\,\mu m \times 1.2\,\mu m$ rectangles, so that $A_D = A_S \approx 14\,(\mu m)^2$, and the perimeters are $P_D = P_S = 12 + 1.2 + 1.2 \approx 14\,\mu m$ (these parameters are rounded to two significant figures solely to simplify the figures and tables).

In passing, notice that a $1.2\,\mu m$ strip of diffusion in a $0.6\,\mu m$ process ($\lambda = 0.3\,\mu m$) is only 4λ wide—wide enough to place a contact only with aggressive spacing rules. The conservative rules in Figure 2.11 would require a diffusion width of at least 2 (rule 6.4a) + 2 (rule 6.3a) + 1.5 (rule 6.2a) = 5.5λ.

3.2.2 Overlap Capacitance

The overlap capacitance calculations for C_{GSOV} and C_{GDOV} in Table 3.1 account for lateral diffusion (the amount the source and drain extend under the gate) using SPICE parameter `LD = 5E-08` or $L_D = 0.05\,\mu m$. Not all versions of SPICE use the equivalent parameter for width reduction, `WD` (assumed zero in Table 3.1), in calculating C_{GDOV} and not all versions subtract W_D to form W_{EFF}.

3.2.3 Gate Capacitance

The gate capacitance calculations in Table 3.1 depend on the operating region. The gate–source capacitance C_{GS} varies from zero when the transistor is off to $0.5C_O$ ($0.5 \times 1.035 \times 10^{-15} = 5.18 \times 10^{-16}$ F) in the linear region to $(2/3)C_O$ in the saturation region (6.9×10^{-16} F). The gate–drain capacitance C_{GD} varies from zero (off) to $0.5C_O$ (linear region) and back to zero (saturation region).

TABLE 3.1 Calculations of parasitic capacitances for an n-channel MOS transistor.

PSpice	Equation	Values[1] for $V_{GS}=0V$, $V_{DS}=3V$, $V_{SB}=0V$
CBD	$C_{BD} = C_{BDJ} + C_{BDSW}$	$C_{BD} = 1.855\times10^{-15} + 2.04\times10^{-16} = 2.06\times10^{-15}$F
	$C_{BDJ} = A_D C_J (1 + V_{DB}/\phi_B)^{-m_J}$ (ϕ_B = PB)	$C_{BDJ} = (4.032\times10^{-15})\,(1 + (3/1))^{-0.56}$ $= 1.86\times10^{-15}$F
	$C_{BDSW} = P_D C_{JSW} (1 + V_{DB}/\phi_B)^{-m_{JSW}}$ (P_D may or may not include channel edge)	$C_{BDSW} = (4.2\times10^{-16})\,(1 + (3/1))^{-0.52}$ $= 2.04\times10^{-16}$F
CBS	$C_{BS} = C_{BSJ} + C_{BSSW}$	$C_{BS} = 4.032\times10^{-15} + 4.2\times10^{-16} = 4.45\times10^{-15}$F
	$C_{BSJ} = A_S C_J (1 + V_{SB}/\phi_B)^{-m_J}$	$A_S C_J = (7.2\times10^{-12})\,(5.6\times10^{-4}) = 4.03\times10^{-15}$F
	$C_{BSSW} = P_S C_{JSW} (1 + V_{SB}/\phi_B)^{-m_{JSW}}$	$P_S C_{JSW} = (8.4\times10^{-6})\,(5\times10^{-11}) = 4.2\times10^{-16}$F
CGSOV	$C_{GSOV} = W_{EFF} C_{GSO}$; $W_{EFF} = W - 2W_D$	$C_{GSOV} = (6\times10^{-6})\,(3\times10^{-10}) = 1.8\times10^{-16}$F
CGDOV	$C_{GDOV} = W_{EFF} C_{GSO}$	$C_{GDOV} = (6\times10^{-6})\,(3\times10^{-10}) = 1.8\times10^{-15}$F
CGBOV	$C_{GBOV} = L_{EFF} C_{GBO}$; $L_{EFF} = L - 2L_D$	$C_{GBOV} = (0.5\times10^{-6})\,(4\times10^{-10}) = 2\times10^{-16}$F
CGS	C_{GS}/C_O = 0 (off), 0.5 (lin.), 0.66 (sat.) C_O(oxide capacitance) = $\dfrac{W_{EFF} L_{EFF} \varepsilon_{ox}}{T_{ox}}$	$C_O = (6\times10^{-6})\,(0.5\times10^{-6})\,(0.00345)$ $= 1.03\times10^{-14}$F. $C_{GS} = 0.0$F
CGD	C_{GD}/C_O = 0 (off), 0.5 (lin.), ≈ 0 (sat.)	$C_{GD} = 0.0$F
CGB	C_{GB} = 0 (on), ≈ C_O in series with C_S (off)	$C_{GB} = 3.88\times10^{-15}$F, C_S = depletion capacitance

[1]Input

```
.MODEL CMOSN NMOS LEVEL=3 PHI=0.7 TOX=10E-09 XJ=0.2U TPG=1 VTO=0.65 DELTA=0.7
+ LD=5E-08 KP=2E-04 UO=550 THETA=0.27 RSH=2 GAMMA=0.6 NSUB=1.4E+17 NFS=6E+11
+ VMAX=2E+05 ETA=3.7E-02 KAPPA=2.9E-02 CGDO=3.0E-10 CGSO=3.0E-10 CGBO=4.0E-10
+ CJ=5.6E-04 MJ=0.56 CJSW=5E-11 MJSW=0.52 PB=1
m1  out1 in1 0 0 cmosn W=6U L=0.6U AS=7.2P AD=7.2P PS=8.4U PD=8.4U
```

The gate–bulk capacitance C_{GB} may be viewed as two capacitors in series: the fixed gate-oxide capacitance, $C_O = W_{EFF} L_{EFF} \varepsilon_{ox} / T_{ox}$, and the variable depletion capacitance, $C_S = W_{EFF} L_{EFF} \varepsilon_{Si} / x_d$, formed by the depletion region that extends under the gate (with varying depth x_d). As the transistor turns on the conducting channel appears and shields the bulk from the gate—and at this point C_{GB} falls to zero. Even with $V_{GS} = 0$ V, the depletion width under the gate is finite and thus $C_{GB} \approx 4 \times 10^{-15}$ F is less than $C_O \approx 10^{-16}$ F. In fact, since $C_{GB} \approx 0.5\, C_O$, we can tell that at $V_{GS} = 0$ V, $C_S \approx C_O$.

Figure 3.5 shows the variation of the parasitic capacitance values.

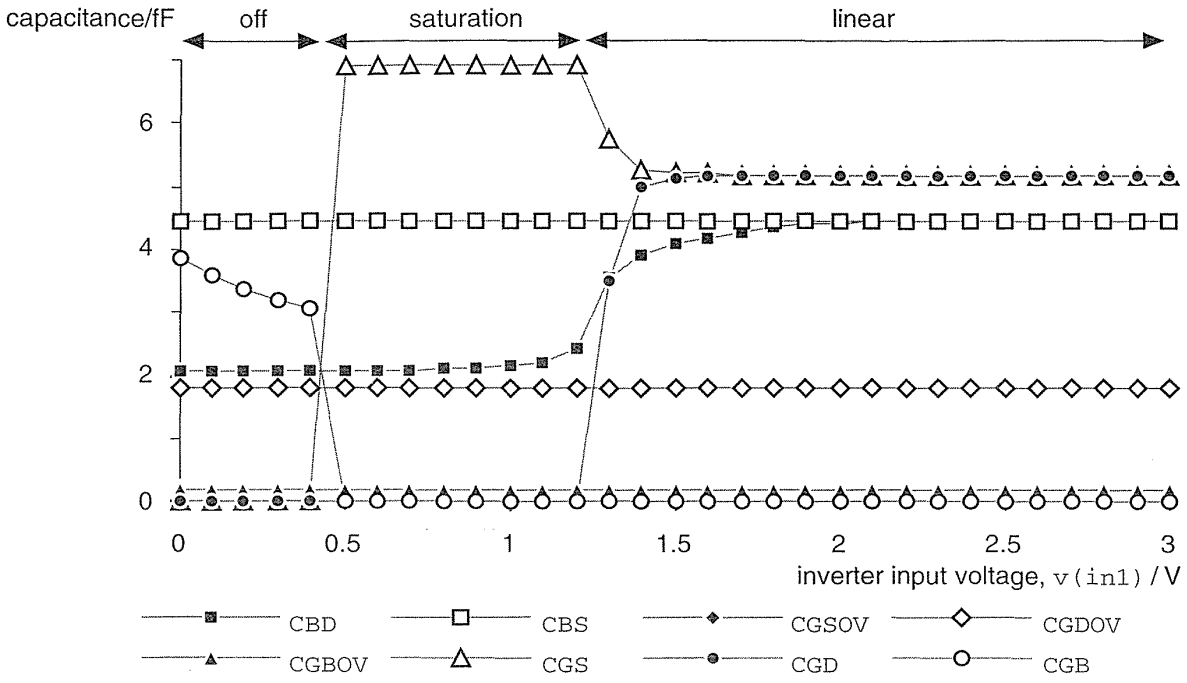


FIGURE 3.5 The variation of *n*-channel transistor parasitic capacitance. Values were obtained from a series of DC simulations using PSpice v5.4, the parameters shown in Table 3.1 (`LEVEL=3`), and by varying the input voltage, `v(in1)`, of the inverter in Figure 3.3(a). Data points are joined by straight lines. Note that `CGSOV` = `CGDOV`.

3.2.4 Input Slew Rate

Figure 3.6 shows an experiment to monitor the input capacitance of an inverter as it switches. We have introduced another variable—the delay of the input ramp or the slew rate of the input.

In Figure 3.6(b) the input ramp is 40 ps long with a slew rate of 3 V/40 ps or 75 GVs^{-1}—as in our previous experiments—and the output of the inverter hardly moves before the input has changed. The input capacitance varies from 20 to 40 fF with an average value of approximately 34 fF for both transitions—we can measure the average value in Probe by plotting `AVG(-i(Vin))`.

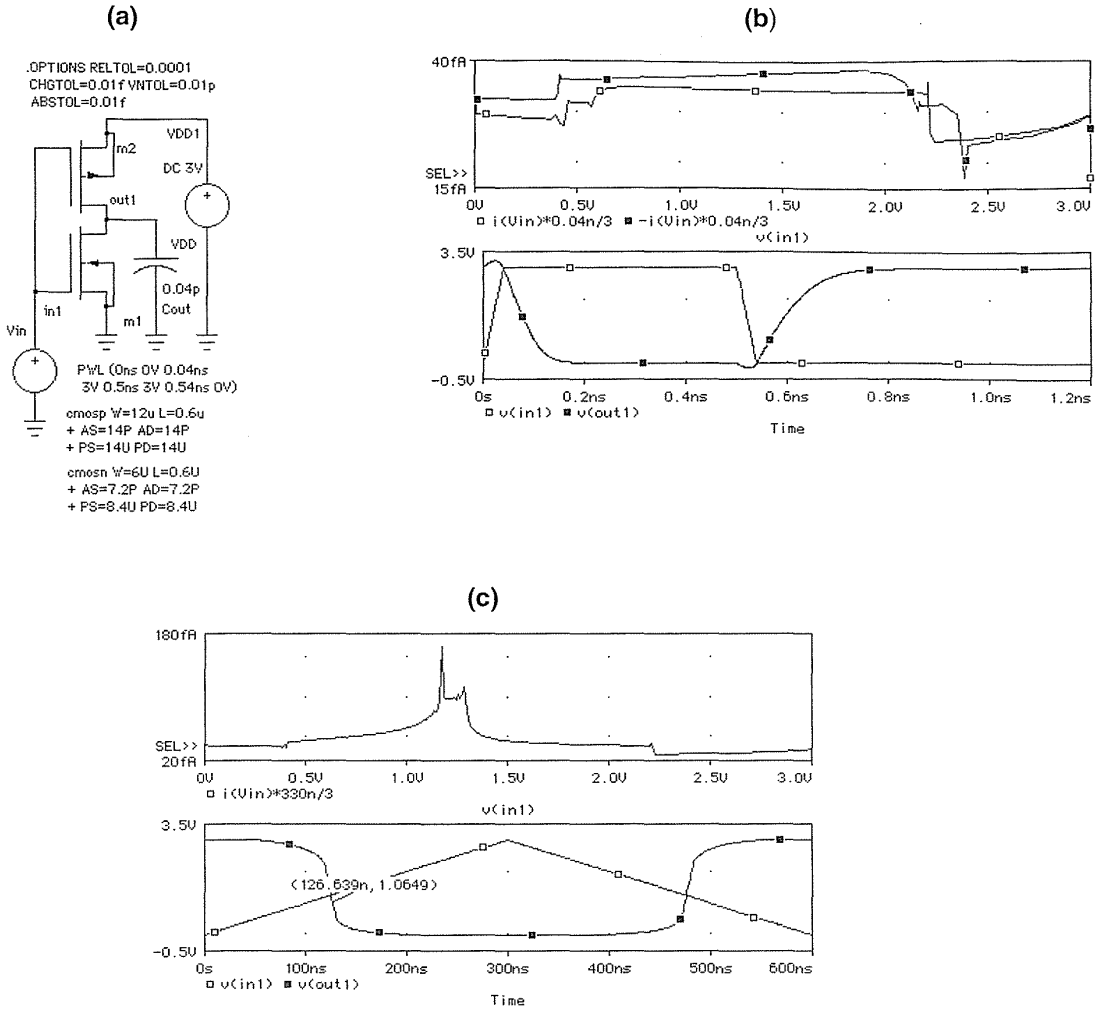


FIGURE 3.6 The input capacitance of an inverter. (a) Input capacitance is measured by monitoring the input current to the inverter, `i(Vin)`. (b) Very fast switching. The current, `i(Vin)`, is multiplied by the input ramp delay ($\Delta t = 0.04$ ns) and divided by the voltage swing ($\Delta V = V_{DD} = 3$ V) to give the equivalent input capacitance, $C = i\Delta t/\Delta V$. Thus an adjusted input current of 40 fA corresponds to an input capacitance of 40 fF. The current, `i(Vin)`, is positive for the rising edge of the input and negative for the falling edge. (c) Very slow switching. The input capacitance is now equal for both transitions.

In Figure 3.6(c) the input ramp is slow enough (300 ns) that we are switching under almost equilibrium conditions—at each voltage we allow the output to find its level on the static transfer curve of Figure 3.2(a). The switching waveforms are quite different. The average input capacitance is now approximately 0.04 pF (a 20 percent difference). The propagation delay (using an input trip point of 0.5 and an output trip point of 0.35) is negative and approximately $150 - 127 = -23$ ns. By changing the input slew rate we have broken our model. For the moment we shall ignore this problem and proceed.

The calculations in Table 3.1 and behavior of Figures 3.5 and 3.6 are very complex. How can we find the value of the parasitic capacitance, C, to fit the model of Figure 3.1? Once again, as we did for pull resistance and the intrinsic output capacitance, instead of trying to derive a theoretical value for C, we adjust the value to fit the model. Before we formulate another experiment we should bear in mind the following questions that the experiment of Figure 3.6 raises: Is it valid to replace the nonlinear input capacitance with a linear component? Is it valid to use a linear input ramp when the normal waveforms are so nonlinear?

Figure 3.7 shows an experiment crafted to answer these questions. The experiment has the following two steps:

1. Adjust `c2` to model the input capacitance of `m5/6`; then C = `c2` = 0.0335 pF.
2. Remove all the parasitic capacitances for inverter `m9/10`—except for the gate capacitances C_{GS}, C_{GD}, and C_{GB}—and then adjust `c3` (0.01 pF) and `c4` (0.025 pF) to model the effect of these missing parasitics.

We can summarize our findings from this and previous experiments as follows:

1. Since the waveforms in Figure 3.7 match, we can model the input capacitance of a logic cell with a linear capacitor. However, we know the input capacitance may vary (by up to 20 percent in our example) with the input slew rate.
2. The input waveform to the inverter `m3/m4` in Figure 3.7 is from another inverter—not a linear ramp. The difference in slew rate causes an error. The measured delay is 85 ps (0.085 ns), whereas our model (Eq. 3.7) predicts

$$t_{PDr} = (38 + 817C_{out})\,\text{ps} = (38 + (817)\ (0.0335))\,\text{ps} = 65\text{ps}. \tag{3.8}$$

3. The total gate-oxide capacitance in our inverter with $T_{ox} = 100\text{Å}$ is

$$\begin{aligned} C_O &= (W_n L_n + W_p L_p)\,\varepsilon_{ox} T_{ox} \\ &= (34.5\times10^{-4})\ ((6)\ (0.6) + (12)\ (0.6))\,\text{pF} = 0.037\ \text{pF}. \end{aligned} \tag{3.9}$$

4. All the transistor parasitic capacitances excluding the gate capacitance contribute 0.01 pF of the 0.0335 pF input capacitance—about 30 percent. The gate capacitances contribute the rest—0.025 pF (about 70 percent).

The last two observations are useful. Since the gate capacitances are nonlinear, we only see about 0.025/0.037 or 70 percent of the 0.037 pF gate-oxide capacitance, C_O, in the input capacitance, C. This means that it happens by chance that the total gate-oxide capacitance is also a rough estimate of the gate input capacitance, $C \approx C_O$. Using L and W rather than L_{EFF} and W_{EFF} in Eq. 3.9 helps this estimate. The accuracy of this estimate depends on the fact that the junction capacitances are approximately one-third of the gate-oxide capacitance—which happens to be true for many CMOS processes for the shapes of transistors that normally occur in logic cells. In the next section we shall use this estimate to help us design logic cells.

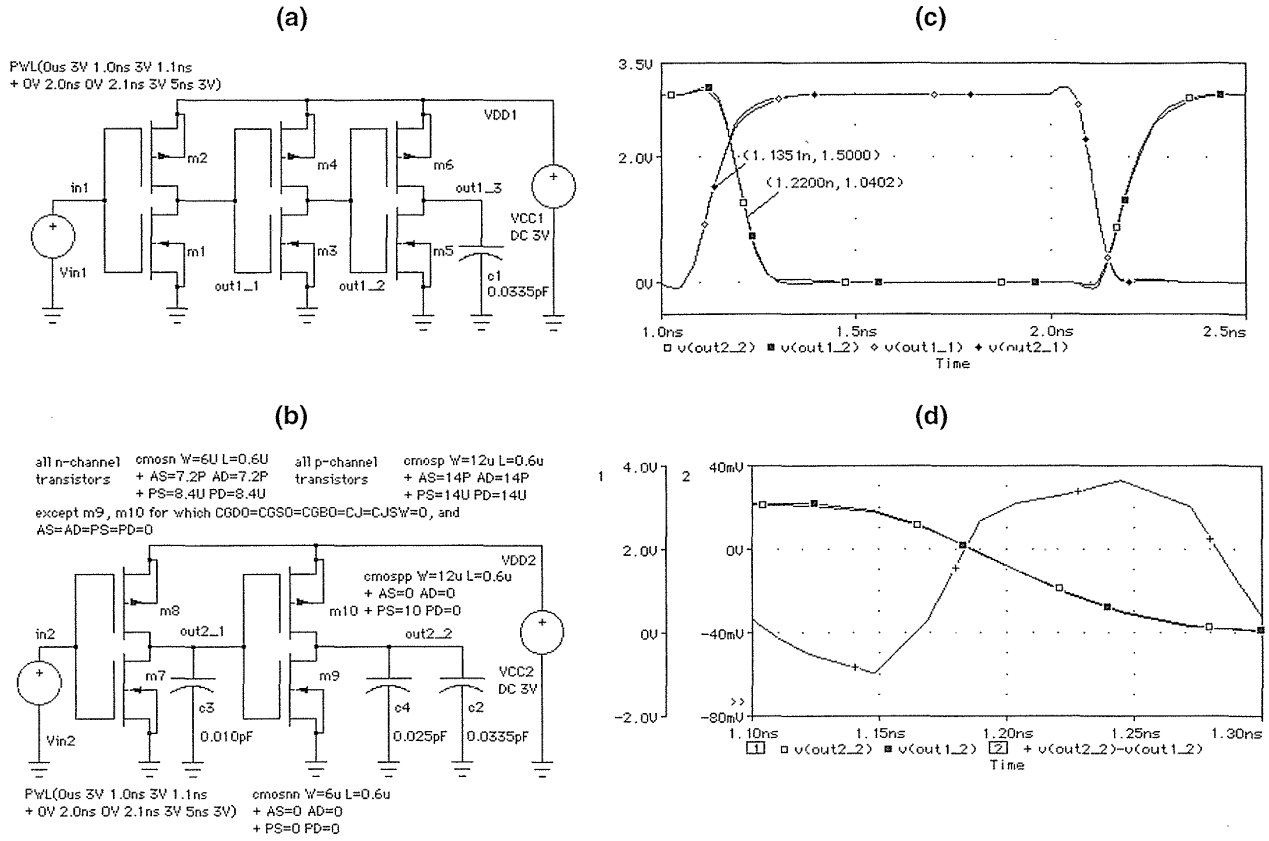


FIGURE 3.7 Parasitic capacitance. (a) All devices in this circuit include parasitic capacitance. (b) This circuit uses linear capacitors to model the parasitic capacitance of `m9/10`. The load formed by the inverter (`m5` and `m6`) is modeled by a 0.0335 pF capacitor (`c2`); the parasitic capacitance due to the overlap of the gates of `m3` and `m4` with their source, drain, and bulk terminals is modeled by a 0.01 pF capacitor (`c3`); and the effect of the parasitic capacitance at the drain terminals of `m3` and `m4` is modeled by a 0.025 pF capacitor (`c4`). (c) The two circuits compared. The delay shown (1.22 − 1.135 = 0.085 ns) is equal to t_{PDf} for the inverter `m3/4`. (d) An exact match would have both waveforms equal at the 0.35 trip point (1.05 V).

3.3 Logical Effort

In this section we explore a delay model based on *logical effort*, a term coined by Ivan Sutherland and Robert Sproull [1991], that has as its basis the time-constant analysis of Carver Mead, Chuck Seitz, and others.

We add a "catch all" nonideal component of delay, t_q, to Eq. 3.2 that includes: (1) delay due to internal parasitic capacitance; (2) the time for the input to reach the switching threshold of the cell; and (3) the dependence of the delay on the slew rate of the input waveform. With these assumptions we can express the delay as follows:

$$t_{PD} = R\,(C_{\text{out}} + C_p) + t_q. \tag{3.10}$$

(The input capacitance of the logic cell is C, but we do not need it yet.)

We will use a standard-cell library for a 3.3 V, 0.5 μm (0.6 μm drawn) technology (from Compass) to illustrate our model. We call this technology **C5**; it is almost identical to the G5 process from Section 2.1 (the Compass library uses a more accurate and more complicated SPICE model than the generic process). The equation for the delay of a 1X drive, two-input NAND cell is in the form of Eq. 3.10 (C_{out} is in pF):

$$t_{PD} = (0.07 + 1.46C_{\text{out}} + 0.15)\ \text{ns}. \tag{3.11}$$

The delay due to the intrinsic output capacitance (0.07 ns, equal to RC_p) and the nonideal delay ($t_q = 0.15$ ns) are specified separately. The nonideal delay is a considerable fraction of the total delay, so we may hardly ignore it. If data books do not specify these components of delay separately, we have to estimate the fractions of the constant part of a delay equation to assign to RC_p and t_q (here the ratio RC_p/t_q is approximately 2).

The data book tells us the input trip point is 0.5 and the output trip points are 0.35 and 0.65. We can use Eq. 3.11 to estimate the pull resistance for this cell as $R \approx 1.46\ \text{nspF}^{-1}$ or about 1.5 kΩ. Equation 3.11 is for the falling delay; the data book equation for the rising delay gives slightly different values (but within 10 percent of the falling delay values).

We can **scale** any logic cell by a scaling factor s (transistor gates become s times wider, but the gate lengths stay the same), and as a result the pull resistance R will decrease to R/s and the parasitic capacitance C_p will increase to sC_p. Since t_q is nonideal, by definition it is hard to predict how it will scale. We shall assume that t_q scales linearly with s for all cells. The total cell delay then scales as follows:

$$t_{PD} = \frac{R}{s}(C_{\text{out}} + sC_p) + st_q. \tag{3.12}$$

For example, the delay equation for a 2X drive ($s = 2$), two-input NAND cell is

$$t_{PD} = (0.03 + 0.75C_{\text{out}} + 0.51)\ \text{ns}. \tag{3.13}$$

Compared to the 1X version (Eq. 3.11), the output parasitic delay has decreased to 0.03 ns (from 0.07 ns), whereas we predicted it would remain constant (the difference is because of the layout); the pull resistance has decreased by a factor of 2 from 1.5 kΩ to 0.75 kΩ, as we would expect; and the nonideal delay has increased to 0.51 ns (from 0.15 ns). The differences between our predictions and the actual values give us a measure of the model accuracy.

We rewrite Eq. 3.12 using the input capacitance of the scaled logic cell, $C_{\text{in}} = sC$,

$$t_{PD} = RC\left(\frac{C_{\text{out}}}{C_{\text{in}}}\right) + RC_p + st_q. \tag{3.14}$$

Finally we normalize the delay using the time constant formed from the pull resistance R_{inv} and the input capacitance C_{inv} of a minimum-size inverter:

$$d = \frac{(RC)\left(\frac{C_{\text{out}}}{C_{\text{in}}}\right) + RC_p + st_q}{\tau} = f + p + q. \tag{3.15}$$

The time constant **tau**,

$$\tau = R_{\text{inv}}C_{\text{inv}}, \tag{3.16}$$

is a basic property of any CMOS technology. We shall measure delays in terms of τ.

The delay equation for a 1X (minimum-size) inverter in the C5 library is

$$t_{PD} = (0.06 + 1.60C_{\text{out}} + 0.10)\ \text{ns}. \tag{3.17}$$

Thus $t_{q\text{inv}} = 0.1$ ns and $R_{\text{inv}} = 1.60\ \text{k}\Omega$. The input capacitance of the 1X inverter (the standard load for this library) is specified in the data book as $C_{\text{inv}} = 0.036$ pF; thus $\tau = (0.036\ \text{pF})(1.60\ \text{k}\Omega) = 0.06$ ns for the C5 technology.

The use of logical effort consists of rearranging and understanding the meaning of the various terms in Eq. 3.15. The delay equation is the sum of three terms,

$$d = f + p + q. \tag{3.18}$$

We give these terms special names as follows:

$$\text{delay} = \text{effort delay} + \text{parasitic delay} + \text{nonideal delay}. \tag{3.19}$$

The **effort delay** f we write as a product of logical effort, g, and electrical effort, h:

$$f = gh. \tag{3.20}$$

So we can further partition delay into the following terms:

$$\text{delay} = \text{logical effort} \times \text{electrical effort} + \text{parasitic delay} + \text{nonideal delay}. \tag{3.21}$$

The **logical effort** g is a function of the type of logic cell,

$$g = \frac{RC}{\tau}. \tag{3.22}$$

What size of logic cell do the R and C refer to? It does not matter because the R and C will change as we scale a logic cell, but the RC product stays the same—the

logical effort is independent of the size of a logic cell. We can find the logical effort by scaling down the logic cell so that it has the same drive capability as the 1X minimum-size inverter. Then the logical effort, g, is the ratio of the input capacitance, C_{in}, of the 1X version of the logic cell to C_{inv} (see Figure 3.8).

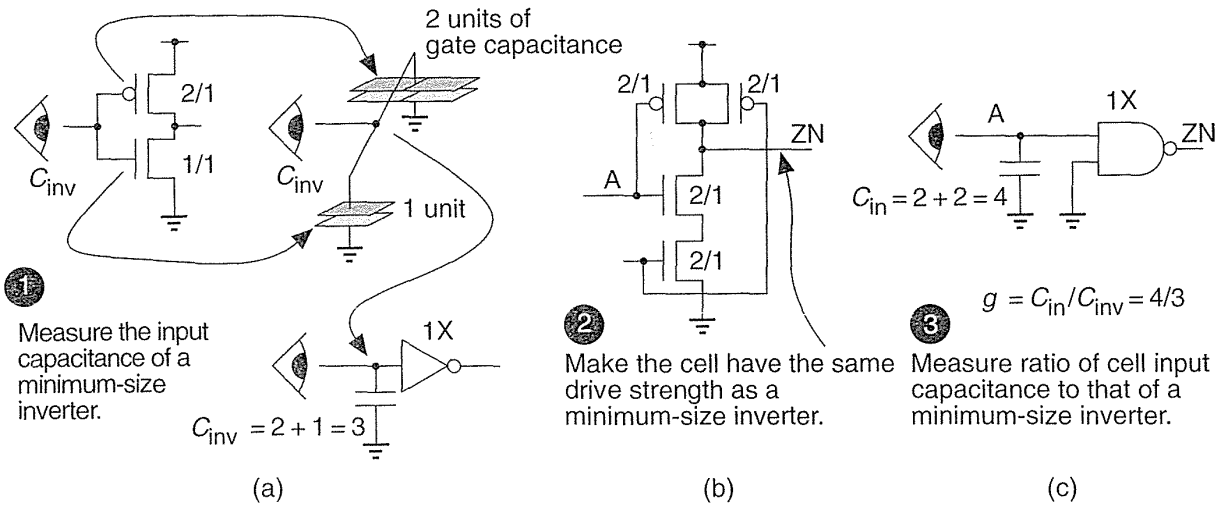


FIGURE 3.8 Logical effort. (a) The input capacitance, C_{inv}, looking into the input of a minimum-size inverter in terms of the gate capacitance of a minimum-size device. (b) Sizing a logic cell to have the same drive strength as a minimum-size inverter (assuming a logic ratio of 2). The input capacitance looking into one of the logic-cell terminals is then C_{in}. (c) The logical effort of a cell is C_{in}/C_{inv}. For a two-input NAND cell, the logical effort, $g = 4/3$.

The **electrical effort** h depends only on the load capacitance C_{out} connected to the output of the logic cell and the input capacitance of the logic cell, C_{in}; thus

$$h = \frac{C_{out}}{C_{in}}. \tag{3.23}$$

The **parasitic delay** p depends on the intrinsic parasitic capacitance C_p of the logic cell, so that

$$p = \frac{RC_p}{\tau}. \tag{3.24}$$

Table 3.2 shows the logical efforts for single-stage logic cells. Suppose the minimum-size inverter has an n-channel transistor with W/L = 1 and a p-channel transistor with W/L = 2 (logic ratio, r, of 2). Then each two-input NAND logic cell input is connected to an n-channel transistor with W/L = 2 and a p-channel transistor with W/L = 2. The input capacitance of the two-input NAND logic cell divided by

that of the inverter is thus 4/3. This is the logical effort of a two-input NAND when $r = 2$. Logical effort depends on the ratio of the logic. For an n-input NAND cell

TABLE 3.2 Cell effort, parasitic delay, and nonideal delay (in units of τ) for single-stage CMOS cells.

Cell	Cell effort (logic ratio = 2)	Cell effort (logic ratio = r)	Parasitic delay/τ	Nonideal delay/τ
inverter	1 (by definition)	1 (by definition)	p_{inv} (by definition)[1]	q_{inv} (by definition)[1]
n-input NAND	$(n+2)/3$	$(n+r)/(r+1)$	$n\,p_{inv}$	$n\,q_{inv}$
n-input NOR	$(2n+1)/3$	$(nr+1)/(r+1)$	$n\,p_{inv}$	$n\,q_{inv}$

[1]For the Compass 0.5 µm technology (C5): $p_{inv} = 1.0$, $q_{inv} = 1.7$, $R_{inv} = 1.5\,k\Omega$, $C_{inv} = 0.036\,pF$.

with ratio r, the p-channel transistors are W/L = r/1, and the n-channel transistors are W/L = n/1. For a NOR cell the n-channel transistors are 1/1 and the p-channel transistors are nr/1.

The parasitic delay arises from parasitic capacitance at the output node of a single-stage logic cell and most (but not all) of this is due to the source and drain capacitance. The parasitic delay of a minimum-size inverter is

$$p_{inv} = \frac{C_p}{C_{inv}}. \tag{3.25}$$

The parasitic delay is a constant, for any technology. For our C5 technology we know $RC_p = 0.06$ ns and, using Eq. 3.17 for a minimum-size inverter, we can calculate $p_{inv} = RC_p/\tau = 0.06/0.06 = 1$ (this is purely a coincidence). Thus C_p is about equal to C_{inv} and is approximately 0.036 pF. There is a large error in calculating p_{inv} from extracted delay values that are so small. Often we can calculate p_{inv} more accurately from estimating the parasitic capacitance from layout.

Because RC_p is constant, the parasitic delay is equal to the ratio of parasitic capacitance of a logic cell to the parasitic capacitance of a minimum-size inverter. In practice this ratio is very difficult to calculate—it depends on the layout. We can approximate the parasitic delay by assuming it is proportional to the sum of the widths of the n-channel and p-channel transistors connected to the output. Table 3.2 shows the parasitic delay for different cells in terms of p_{inv}.

The **nonideal delay** q is hard to predict and depends mainly on the physical size of the logic cell (proportional to the cell area in general, or width in the case of a standard cell or a gate-array macro),

$$q = \frac{st_q}{\tau}. \tag{3.26}$$

We define q_{inv} in the same way we defined p_{inv}. An n-input cell is approximately n times larger than an inverter, giving the values for nonideal delay shown in Table 3.2. For our C5 technology, from Eq. 3.17, $q_{inv} = t_{qinv}/\tau = 0.1\text{ ns}/0.06\text{ ns} = 1.7$.

3.3.1 Predicting Delay

As an example, let us predict the delay of a three-input NOR logic cell with 2X drive, driving a net with a fanout of four, with a total load capacitance (comprising the input capacitance of the four cells we are driving plus the interconnect) of 0.3 pF.

From Table 3.2 we see $p = 3p_{inv}$ and $q = 3q_{inv}$ for this cell. We can calculate C_{in} from the fact that the input gate capacitance of a 1X drive, three-input NOR logic cell is equal to gC_{inv}, and for a 2X logic cell, $C_{in} = 2gC_{inv}$. Thus,

$$gh = g\frac{C_{out}}{C_{in}} = \frac{g\,(0.3\text{pF})}{2g\,(C_{inv})} = \frac{(0.3\text{pF})}{2\,(0.036\text{pF})}. \tag{3.27}$$

(Notice that g cancels out in this equation, we shall discuss this in the next section.)

The delay of the NOR logic cell, in units of τ, is thus

$$\begin{aligned} d &= gh + p + q = \frac{(0.3 \times 10^{-12})}{(2)\ (0.036 \times 10^{-12})} + (3)\ (1) + (3)\ (1.7) \\ &= (4.1666667 + 3 + 5.1) \\ &= 12.266667\tau, \end{aligned} \tag{3.28}$$

equivalent to an absolute delay, $t_{PD} \approx 12.3 \times 0.06\text{ ns} = 0.74\text{ ns}$.

The delay for a 2X drive, three-input NOR logic cell in the C5 library is

$$t_{PD} = (0.03 + 0.72C_{out} + 0.60)\text{ ns}. \tag{3.29}$$

With $C_{out} = 0.3$ pF,

$$t_{PD} = 0.03 + 0.72\,(0.3) + 0.60 = 0.846\text{ns}, \tag{3.30}$$

compared to our prediction of 0.74 ns. Almost all of the error here comes from the inaccuracy in predicting the nonideal delay. Logical effort gives us a method to examine relative delays and not accurately calculate absolute delays. More important is that logical effort gives us an insight into why logic has the delay it does.

3.3.2 Logical Area and Logical Efficiency

Figure 3.9 shows a single-stage OR-AND-INVERT cell that has different logical efforts at each input. The logical effort for the OAI221 is the **logical-effort vector** $g = (7/3, 7/3, 5/3)$. For example, the first element of this vector, 7/3, is the logical effort of inputs A and B in Figure 3.9.

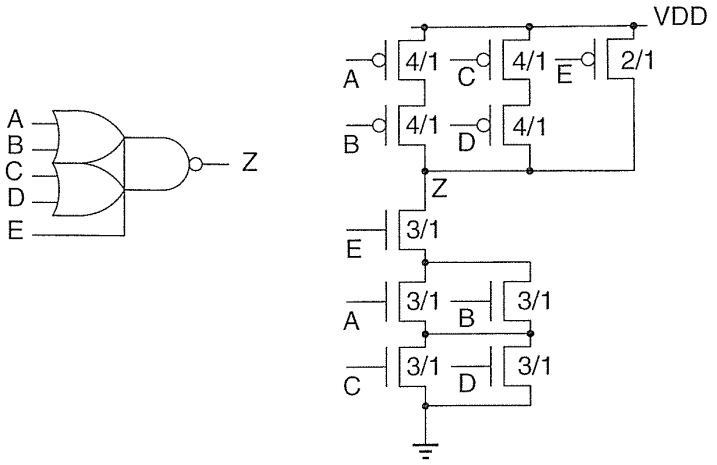


FIGURE 3.9 An OAI221 logic cell with different logical efforts at each input. In this case $g = (7/3, 7/3, 5/3)$. The logical effort for inputs A and B is 7/3, the logical effort for inputs C and D is also 7/3, and for input E the logical effort is 5/3. The logical area is the sum of the transistor areas, 33 logical squares.

We can calculate the area of the transistors in a logic cell (ignoring the routing area, drain area, and source area) in units of a minimum-size n-channel transistor—we call these units **logical squares**. We call the transistor area the **logical area**. For example, the logical area of a 1X drive cell, OAI221X1, is calculated as follows:

- n-channel transistor sizes: $3/1 + 4 \times (3/1)$
- p-channel transistor sizes: $2/1 + 4 \times (4/1)$
- total logical area $= 2 + (4 \times 4) + (5 \times 3) = 33$ logical squares

Figure 3.10 shows a single-stage AOI221 cell, with $g = (8/3, 8/3, 6/3)$. The calculation of the logical area (for a AOI221X1) is as follows:

- n-channel transistor sizes: $1/1 + 4 \times (2/1)$
- p-channel transistor sizes: $6/1 + 4 \times (6/1)$
- logical area $= 1 + (4 \times 2) + (5 \times 6) = 39$ logical squares

These calculations show us that the single-stage AOI221, with an area of 33 logical squares and logical effort of (7/3, 7/3, 5/3), is more **logically efficient** than the single-stage OAI221 logic cell with a larger area of 39 logical squares and larger logical effort of (8/3, 8/3, 6/3).

3.3.3 Logical Paths

When we calculated the delay of the NOR logic cell in Section 3.3.1, the answer did not depend on the logical effort of the cell, g (it cancelled out in Eqs. 3.27 and 3.28). This is because g is a measure of the input capacitance of a 1X drive logic cell. Since we were not driving the NOR logic cell with another logic cell, the input capacitance of the NOR logic cell had no effect on the delay. This is what we do in a data book—we measure logic-cell delay using an ideal input waveform that is the

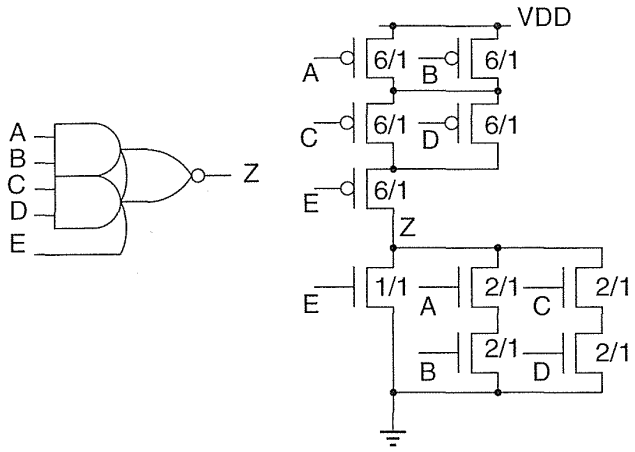


FIGURE 3.10 An AND-OR-INVERT cell, an AOI221, with logical-effort vector, $g = (8/3, 8/3, 7/3)$. The logical area is 39 logical squares.

same no matter what the input capacitance of the cell. Instead let us calculate the delay of a logic cell when it is driven by a minimum-size inverter. To do this we need to extend the notion of logical effort.

So far we have only considered a single-stage logic cell, but we can extend the idea of logical effort to a chain of logic cells or **logical path**. Consider the logic path when we use a minimum-size inverter ($g_0 = 1$, $p_0 = 1$, $q_0 = 1.7$) to drive one input of a 2X drive, three-input NOR logic cell with $g_1 = (nr + 1)/(r + 1)$, $p_1 = 3$, $q_1 = 3$, and a load equal to four standard loads. If the logic ratio is $r = 1.5$, then $g_1 = 5.5/2.5 = 2.2$.

The delay of the inverter is

$$\begin{aligned} d_0 &= g_0 h_0 + p_0 + q_0 = (1)\left(\frac{2g_1 C_{\text{inv}}}{C_{\text{inv}}}\right) + 1 + 1.7 \\ &= (1)\ (2)\ (2.2) + 1 + 1.7 \\ &= 7.1. \end{aligned} \tag{3.31}$$

Of this 7.1τ delay we can attribute 4.4τ to the loading of the NOR logic cell input capacitance, which is $2g_1 C_{\text{inv}}$. The delay of the NOR logic cell is, as before, $d_1 = g_1 h_1 + p_1 + q_1 = 12.3$, making the total delay $7.1 + 12.3 = 19.4$, so the absolute delay is $(19.4)\ (0.06\ \text{ns}) = 1.164\ \text{ns}$, or about 1.2 ns.

We can see that the **path delay** D is the sum of the logical effort, parasitic delay, and nonideal delay at each stage. In general, we can write the path delay as

$$D = \sum_{i \in \text{path}} g_i h_i + \sum_{i \in \text{path}} (p_i + q_i). \tag{3.32}$$

3.3.4 Multistage Cells

Consider the following function (a multistage AOI221 logic cell):

$$\begin{aligned} ZN(A1, A2, B1, B2, C) &= NOT(NAND(NAND(A1, A2), AOI21(B1, B2, C))) \\ &= (((A1 \cdot A2)' \cdot (B1 \cdot B2 + C)')')' = (A1 \cdot A2 + B1 \cdot B2 + C)' \\ &= AOI221(A1, A2, B1, B2, C). \end{aligned} \tag{3.33}$$

Figure 3.11(a) shows this implementation with each input driven by a minimum-size inverter so we can measure the effect of the cell input capacitance.

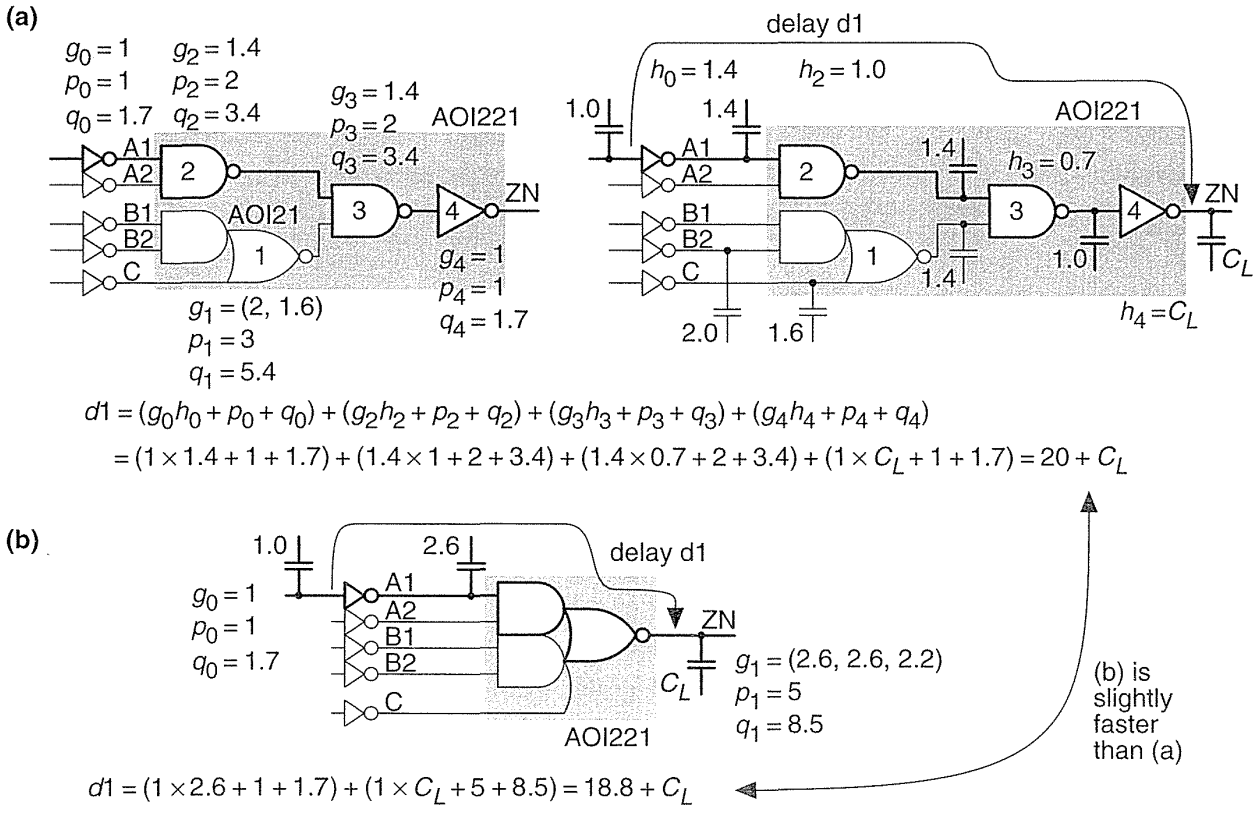


FIGURE 3.11 Logical paths. (a) An AOI221 logic cell constructed as a multistage cell from smaller cells. (b) A single-stage AOI221 logic cell.

The logical efforts of each of the logic cells in Figure 3.11(a) are as follows:

$$g_0 = g_4 = g(\text{NOT}) = 1,$$

$$g_1 = g\,(\text{AOI21}) = (2,\ (2r+1)/(r+1)) = (2,\ 4/2.5) = (2,\ 1.6),$$

$$g_2 = g_3 = g(\text{NAND2}) = (r+2)/(r+1) = (3.5)/(2.5) = 1.4. \tag{3.34}$$

Each of the logic cells in Figure 3.11 has a 1X drive strength. This means that the input capacitance of each logic cell is given, as shown in the figure, by gC_{inv}.

Using Eq. 3.32 we can calculate the delay from the input of the inverter driving A1 to the output ZN as

$$\begin{aligned} d1 &= (1)\ (1.4) + 1 + 1.7 + 1.4\,(1) + 2 + 3.4 \\ &\quad + (1.4)\ (0.7) + 2 + 3.4 + (1)\,C_L + 1 + 1.7 \\ &= (20 + C_L)\,. \end{aligned} \tag{3.35}$$

In Eq. 3.35 we have normalized the output load, C_L, by dividing it by a standard load (equal to C_{inv}). We can calculate the delays of the other paths similarly.

More interesting is to compare the multistage implementation with the single-stage version. In our C5 technology, with a logic ratio, $r = 1.5$, we can calculate the logical effort for a single-stage AOI221 logic cell as

$$\begin{aligned} g(\text{AOI221}) &= ((3r+2)/(r+1),\ (3r+2)/(r+1),\ (3r+1)/(r+1)) \\ &= (6.5/2.5,\ 6.5/2.5,\ 5.5/2.5) = (2.6,\ 2.6,\ 2.2). \end{aligned} \tag{3.36}$$

This gives the delay from an inverter driving the A input to the output ZN of the single-stage logic cell as

$$d1 = (\ (1)\,(2.6) + 1 + 1.7 + (1)C_L + 5 + 8.5) = (18.8 + C_L)\,. \tag{3.37}$$

The single-stage delay is very close to the delay for the multistage version of this logic cell. In some ASIC libraries the AOI221 is implemented as a multistage logic cell instead of using a single stage. It raises the question: Can we make the multistage logic cell any faster by adjusting the scale of the intermediate logic cells?

3.3.5 Optimum Delay

Before we can attack the question of how to optimize delay in a logic path, we shall need some more definitions. The **path logical effort** G is the product of logical efforts on a path:

$$G = \prod_{i \in \text{path}} g_i. \tag{3.38}$$

The **path electrical effort** H is the product of the electrical efforts on the path,

$$H = \prod_{i \in \text{path}} h_i = \frac{C_{\text{out}}}{C_{\text{in}}}, \tag{3.39}$$

where C_{out} is the last output capacitance on the path (the load) and C_{in} is the first input capacitance on the path.

The **path effort** F is the product of the path electrical effort and logical efforts,

$$F = GH. \tag{3.40}$$

The optimum effort delay for each stage is found by minimizing the path delay D by varying the electrical efforts of each stage h_i, while keeping H, the path electrical effort fixed. The optimum effort delay is achieved when each stage operates with equal effort,

$$\hat{f}_i = g_i h_i = F^{1/N}. \tag{3.41}$$

This a useful result. The optimum path delay is then

$$\hat{D} = NF^{1/N} + P + Q = N(GH)^{1/N} + P + Q, \tag{3.42}$$

where $P + Q$ is the sum of path parasitic delay and nonideal delay,

$$P + Q = \sum_{i \in \text{path}} p_i + q_i. \tag{3.43}$$

We can use these results to improve the AOI221 multistage implementation of Figure 3.11(a). Assume that we need a 1X cell, so the output inverter (cell 4) must have 1X drive strength. This fixes the capacitance we must drive as $C_{\text{out}} = C_{\text{inv}}$ (the capacitance at the input of this inverter). The input inverters are included to measure the effect of the cell input capacitance, so we cannot cheat by altering these. This fixes the input capacitance as $C_{\text{in}} = C_{\text{inv}}$. In this case $H = 1$.

The logic cells that we can scale on the path from the A input to the output are NAND logic cells labeled as 2 and 3. In this case

$$G = g_0 \times g_2 \times g_3 = 1 \times 1.4 \times 1.4 = 1.95. \tag{3.44}$$

Thus $F = GH = 1.95$ and the optimum stage effort is $1.95^{(1/3)} = 1.25$, so that the optimum delay $NF^{1/N} = 3.75$. From Figure 3.11(a) we see that

$$g_0 h_0 + g_2 h_2 + g_3 h_3 = 1.4 + 1.4 + 1 = 3.8. \tag{3.45}$$

This means that even if we scale the sizes of the cells to their optimum values, we only save a fraction of a τ ($3.8 - 3.75 = 0.05$). This is a useful result (and one that is true in general)—the delay is not very sensitive to the scale of the cells. In this case it means that we can reduce the size of the two NAND cells in the multicell imple-

mentation of an AOI221 without sacrificing speed. We can use logical effort to predict what the change in delay will be for any given cell sizes.

We can use logical effort in the design of logic cells and in the design of logic that uses logic cells. If we do have the flexibility to continuously size each logic cell (which in ASIC design we normally do not, we usually have to choose from 1X, 2X, 4X drive strengths), each logic stage can be sized using the equation for the individual stage electrical efforts,

$$\hat{h}_i = \frac{F^{1/N}}{g_i}. \tag{3.46}$$

For example, even though we know that it will not improve the delay by much, let us size the cells in Figure 3.11(a). We shall work backward starting at the fixed load capacitance at the input of the last inverter.

For NAND cell 3, $gh = 1.25$; thus (since $g = 1.4$), $h = C_{out}/C_{in} = 0.893$. The output capacitance, C_{out}, for this NAND cell is the input capacitance of the inverter—fixed as 1 standard load, C_{inv}. This fixes the input capacitance, C_{in}, of NAND cell 3 at $1/0.893 = 1.12$ standard loads. Thus, the scale of NAND cell 3 is 1.12/1.4 or 0.8X.

Now for NAND cell 2, $gh = 1.25$; C_{out} for NAND cell 2 is the C_{in} of NAND cell 3. Thus C_{in} for NAND cell 2 is $1.12/0.893 = 1.254$ standard loads. This means the scale of NAND cell 2 is 1.254/1.4 or 0.9X.

The optimum sizes of the NAND cells are not very different from 1X in this case because $H = 1$ and we are only driving a load no bigger than the input capacitance. This raises the question: What is the optimum stage effort if we have to drive a large load, $H >> 1$? Notice that, so far, we have only calculated the optimum stage effort when we have a fixed number of stages, N. We have said nothing about the situation in which we are free to choose, N, the number of stages.

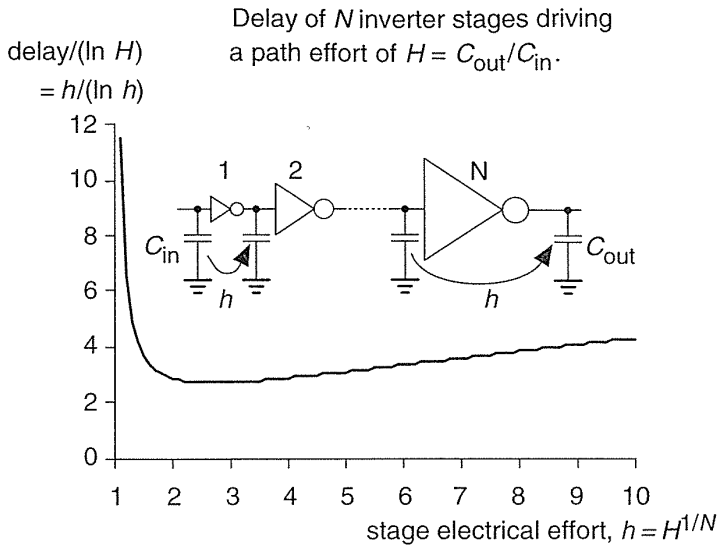
3.3.6 Optimum Number of Stages

Suppose we have a chain of N inverters each with equal stage effort, $f = gh$. Neglecting parasitic and nonideal delay, the total path delay is $Nf = Ngh = Nh$, since $g = 1$ for an inverter. Suppose we need to drive a path electrical effort H; then $h^N = H$, or $N \ln h = \ln H$. Thus the delay, $Nh = h \ln H / \ln h$. Since $\ln H$ is fixed, we can only vary $h/\ln(h)$. Figure 3.12 shows that this is a very shallow function with a minimum at $h = e \approx 2.718$. At this point $\ln h = 1$ and the total delay is $Ne = e \ln H$. This result is particularly useful in driving large loads either on-chip (the clock, for example) or off-chip (I/O pad drivers, for example).

Figure 3.12 shows us how to minimize delay regardless of area or power and neglecting parasitic and nonideal delays. More complicated equations can be derived, including nonideal effects, when we wish to trade off delay for smaller area or reduced power.

FIGURE 3.12 Stage effort.

h	h/(ln h)
1.5	3.7
2	2.9
2.7	2.7
3	2.7
4	2.9
5	3.1
10	4.3



3.4 Library-Cell Design

The optimum cell layout for each process generation changes because the design rules for each ASIC vendor's process are always slightly different—even for the same generation of technology. For example, two companies may have very similar 0.35 μm CMOS process technologies, but the third-level metal spacing might be slightly different. If a cell library is to be used with both processes, we could construct the library by adopting the most stringent rules from each process. A library constructed in this fashion may not be competitive with one that is constructed specifically for each process. Even though ASIC vendors prize their design rules as secret, it turns out that they are similar—except for a few details. Unfortunately, it is the details that stop us moving designs from one process to another. Unless we are a very large customer it is difficult to have an ASIC vendor change or **waive** design rules for us. We would like all vendors to agree on a common set of design rules. This is, in fact, easier than it sounds. The reason that most vendors have similar rules is because most vendors use the same manufacturing equipment and a similar process. It is possible to construct a highest common denominator library that extracts the most from the current manufacturing capability. Some library companies and the large Japanese ASIC vendors are adopting this approach.

Layout of library cells is either hand-crafted or uses some form of **symbolic layout**. Symbolic layout is usually performed in one of two ways: using either interactive graphics or a text layout language. Shapes are represented by simple lines or rectangles, known as **sticks** or **logs**, in symbolic layout. The actual dimensions of the sticks or logs are determined after layout is completed in a postprocessing step.

An alternative to graphical symbolic layout uses a text layout language, similar to a programming language such as C, that directs a program to assemble layout. The spacing and dimensions of the layout shapes are defined in terms of variables rather than constants. These variables can be changed after symbolic layout is complete to adjust the layout spacing to a specific process.

Mapping symbolic layout to a specific process technology uses 10–20 percent more area than hand-crafted layout (though this can then be further reduced to 5–10 percent with compaction). Most symbolic layout systems do not allow 45° layout and this introduces a further area penalty (my experience shows this is about 5–15 percent). As libraries get larger, and the capability to quickly move libraries and ASIC designs between different generations of process technologies becomes more important, the advantages of symbolic layout may outweigh the disadvantages.

3.5 Library Architecture

Figure 3.13(a) shows cell use data from over 150 CMOS gate array designs. These results are remarkably similar to that from other ASIC designs using different libraries and different technologies and show that typically 80 percent of an ASIC uses less than 20 percent of the cell library.

We can use the data in Figure 3.13(a) to derive some useful conclusions about the number and types of cells to be included in a library. Before we do this, a few words of caution are in order. First, the data shown in Figure 3.13(a) tells us about cells that are included a library. This data cannot tell us anything about cells that are not (and perhaps should be) included in a library. Second, the type of design entry we use—and the type of ASIC we are designing—can dramatically affect the profile of the use of different cell types. For example, if we use a high-level design language, together with logic synthesis, to enter an ASIC design, this will favor the use of the complex combinational cells (cells of the AOI family that are particularly area efficient in CMOS, but are difficult to work with when we design by hand).

Figure 3.13(a) tells us which cells we use most often, but does not take into account the cell area. What we really want to know are which cells are most important in determining the area of an ASIC. Figure 3.13(b) shows the area of the cells—normalized to the area of a minimum-size inverter. If we take the data in Figure 3.13(a) and multiply by the cell areas, we can derive a new measure of the contribution of each cell in a library (Figure 3.13c). This new measure, **cell importance,** is a measure of how much area each cell in a library contributes to a typical ASIC. For example, we can see from Figure 3.13(c) that a D flip-flop (with a cell importance of 3.5) contributes 3.5 times as much area on a typical ASIC than does an inverter (with a cell importance of 1).

Figure 3.13(c) shows cell importance ordered by the cell frequency of use and normalized to an inverter. We can rearrange this data in terms of cell importance, as

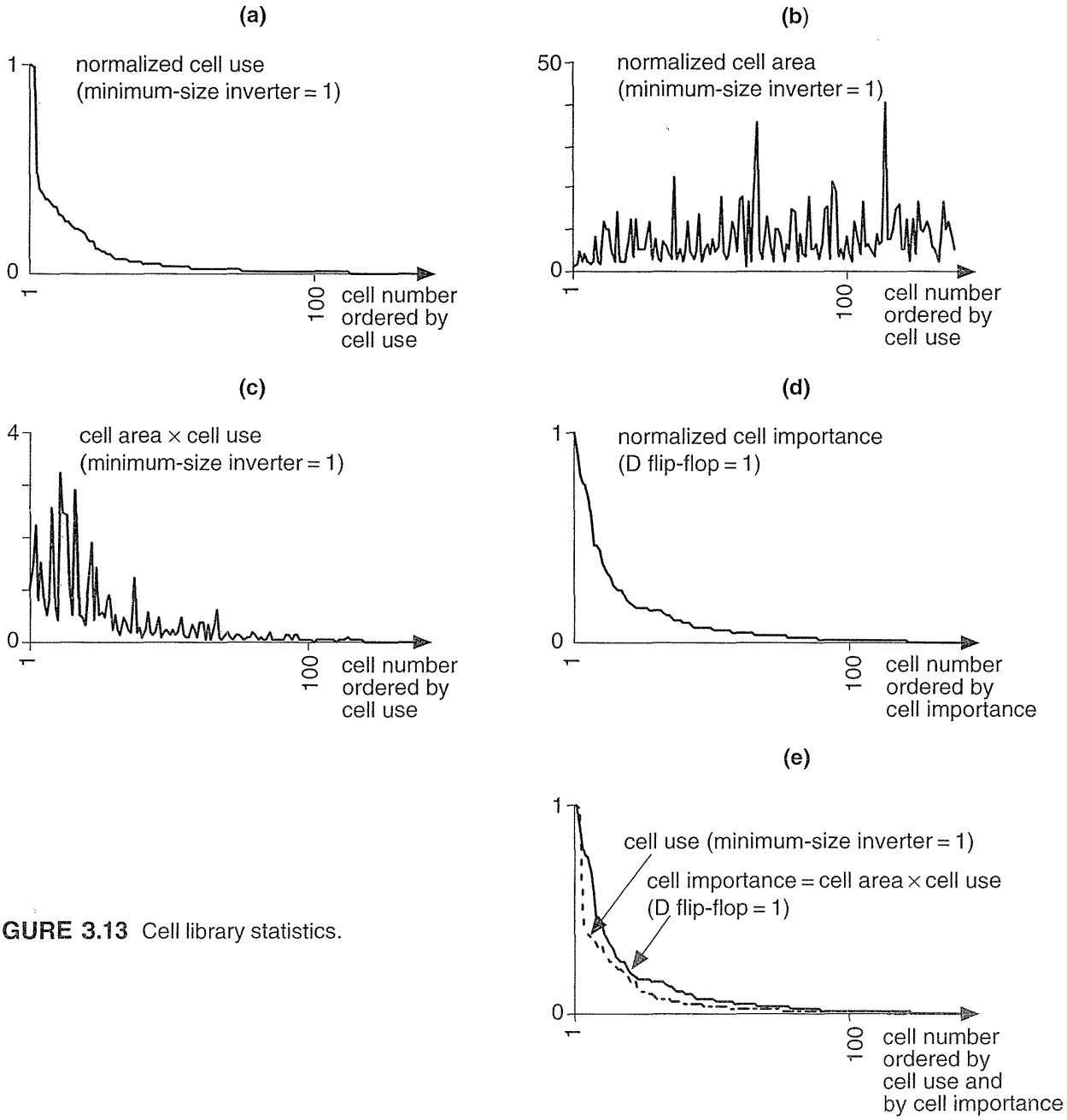


FIGURE 3.13 Cell library statistics.

shown in Figure 3.13(d), and normalized so that now the most important cell, a D flip-flop, has a cell importance of 1. Figure 3.13(e) includes the cell use data on the same scale as the cell importance data. Both show roughly the same shape, reflecting

that both measures obey an 80–20 rule. Roughly 20 percent of the cells in a library correspond to 80 percent of the ASIC area and 80 percent of the cells we use (but not the same 20 percent—that is why cell importance is useful).

Figure 3.13(e) shows us that the most important cells, measured by their contribution to the area of an ASIC, are not necessarily the cells that we use most often. If we wish to build or buy a dense library, we must concentrate on the area of those cells that have the highest cell importance—not the most common cells.

3.6 Gate-Array Design

Each logic cell or macro in a gate-array library is predesigned using fixed tiles of transistors known as the **gate-array base cell** (or just **base cell**). We call the arrangement of base cells across a whole chip in a complete gate array the **gate-array base** (or just **base**). ASIC vendors offer a selection of bases, with a different total numbers of transistors on each base. For example, if our ASIC design uses 48k equivalent gates and the ASIC vendor offers gate arrays bases with 50k-, 75k-, and 100k-gates, we will probably have to use the 75k-gate base (because it is unlikely that we can use 48/50 or 96 percent of the transistors on the 50k-gate base).

We isolate the transistors on a gate array from one another either with thick field oxide (in the case of oxide-isolated gate arrays) or by using other transistors that are wired permanently off (in gate-isolated gate arrays). Channeled and channelless gate arrays may use either gate isolation or oxide isolation.

Figure 3.14(a) shows a base cell for a **gate-isolated gate array**. This base cell has two transistors: one p-channel and one n-channel. When these base cells are placed next to each other, the n-diffusion and p-diffusion layers form continuous strips that run across the entire chip broken only at the poly gates that cross at regularly spaced intervals (Figure 3.14b). The metal interconnect spacing determines the separation of the transistors. The metal spacing is determined by the design rules for the metal and contacts. In Figure 3.14(c) we have shown all possible locations for a contact in the base cell. There is room for 21 contacts in this cell and thus room for 21 interconnect lines running in a horizontal direction (we use m1 running horizontally). We say that there are 21 **horizontal tracks** in this cell or that the cell is 21 tracks high. In a similar fashion the space that we need for a vertical interconnect (m2) is called a **vertical track**. The horizontal and vertical track widths are not necessarily equal, because the design rules for m1 and m2 are not always equal.

We isolate logic cells from each other in gate-isolated gate arrays by connecting transistor gates to the supply bus—hence the name, **gate isolation**. If we connect the gate of an n-channel transistor to V_{SS}, we isolate the regions of n-diffusion on each side of that transistor (we call this an **isolator transistor** or device, or just isolator). Similarly if we connect the gate of a p-channel transistor to V_{DD}, we isolate adjacent p-diffusion regions.

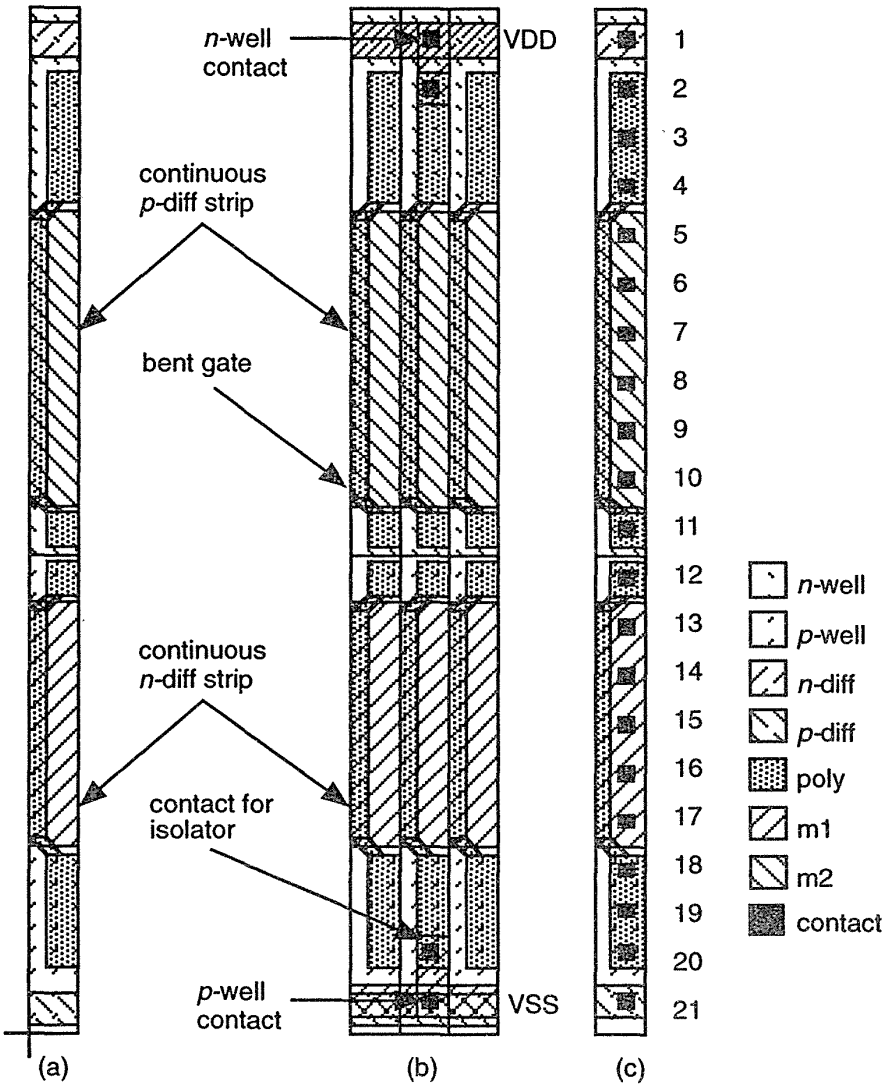


FIGURE 3.14 The construction of a gate-isolated gate array. (a) The one-track-wide base cell containing one *p*-channel and one *n*-channel transistor. (b) Three base cells: the center base cell is being used to isolate the base cells on either side from each other. (c) A base cell including all possible contact positions (there is room for 21 contacts in the vertical direction, showing the base cell has a height of 21 tracks).

Oxide-isolated gate arrays often contain four transistors in the base cell: the two *n*-channel transistors share an *n*-diffusion strip and the two *p*-channel transistors share a *p*-diffusion strip. This means that the two *n*-channel transistors in each base cell are electrically connected in series, as are the *p*-channel transistors. The base cells are isolated from each other using **oxide isolation**. During the fabrication pro-

cess a layer of the thick field oxide is left in place between each base cell and this separates the *p*-diffusion and *n*-diffusion regions of adjacent base cells.

Figure 3.15 shows an **oxide-isolated gate array**. This cell contains eight transistors (which occupy six vertical tracks) plus one-half of a single track that contains the well contacts and substrate connections that we can consider to be shared by each base cell.

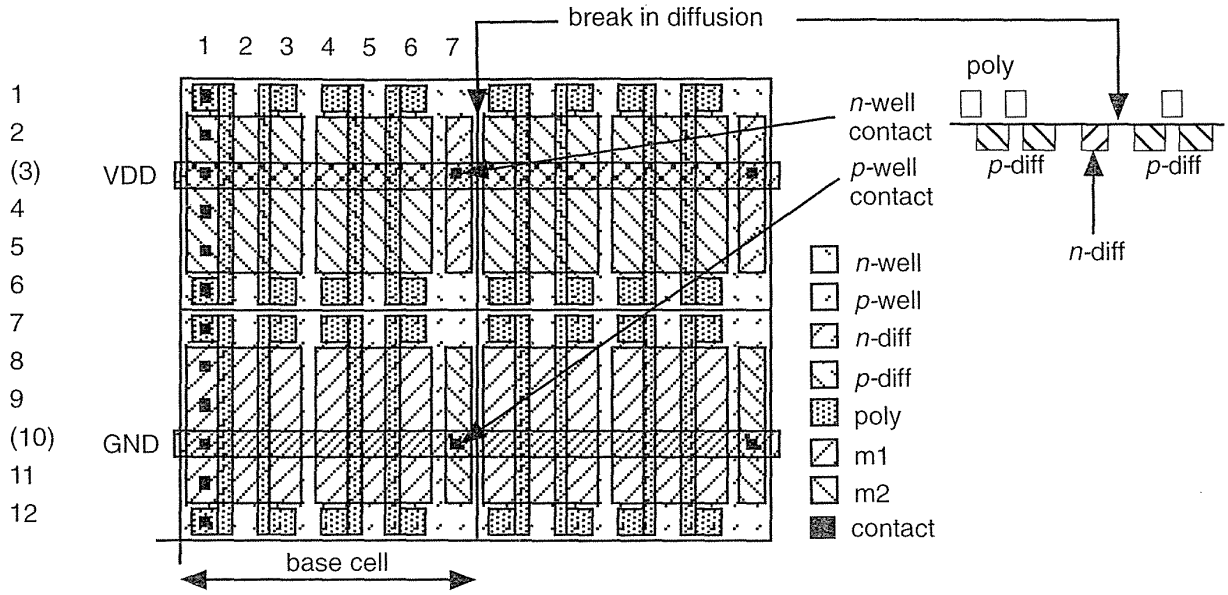


FIGURE 3.15 An oxide-isolated gate-array base cell. The figure shows two base cells, each containing eight transistors and two well contacts. The *p*-channel and *n*-channel transistors are each 4 tracks high (corresponding to the width of the transistor). The leftmost vertical track of the left base cell includes all 12 possible contact positions (the height of the cell is 12 tracks). As outlined here, the base cell is 7 tracks wide (we could also consider the base cell to be half this width).

Figure 3.16 shows a base cell in which the gates of the *n*-channel and *p*-channel transistors are connected on the polysilicon layer. Connecting the gates in poly saves contacts and a metal interconnect in the center of the cell where interconnect is most congested. The drawback of the preconnected gates is a loss in flexibility in cell design. Implementing memory and logic based on transmission gates will be less efficient using this type of base cell, for example.

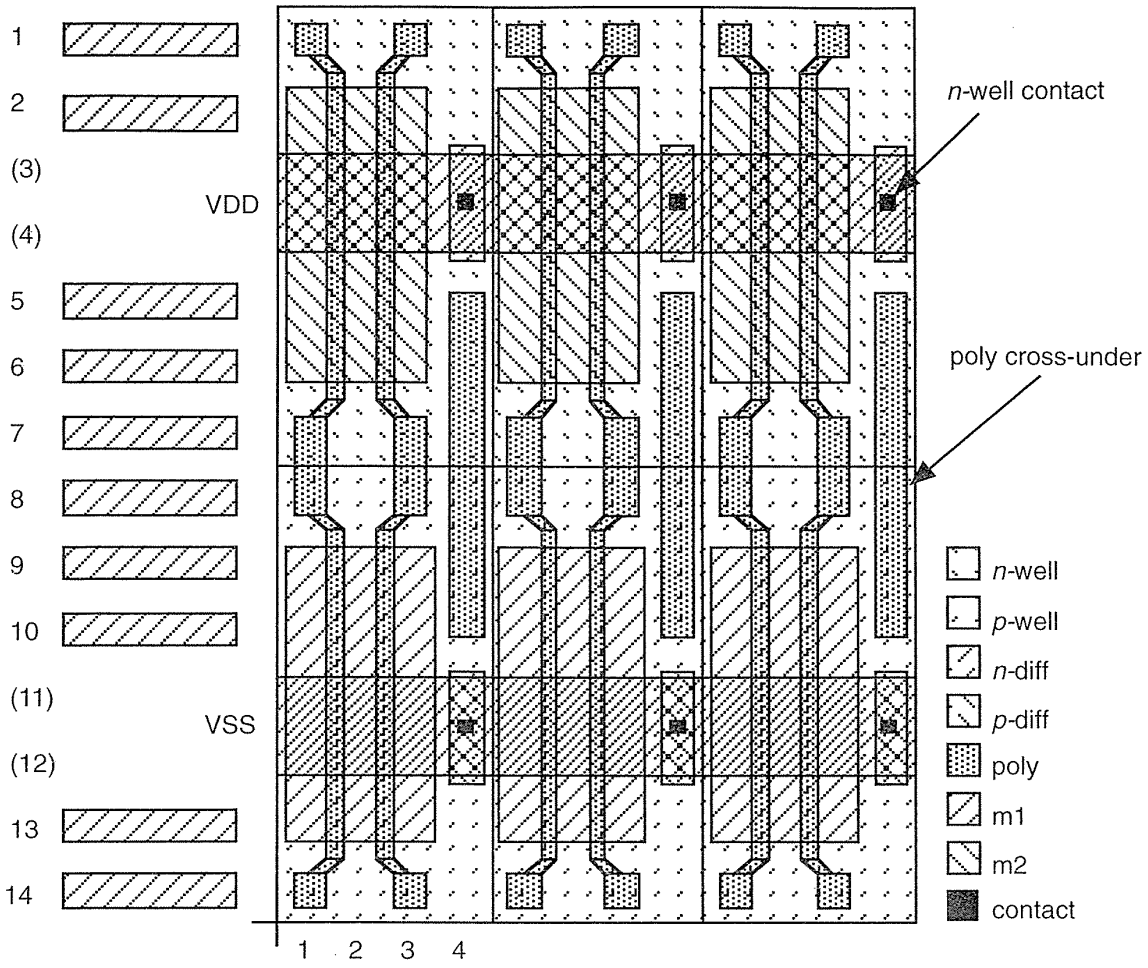


FIGURE 3.16 This oxide-isolated gate-array base cell is 14 tracks high and 4 tracks wide. VDD (tracks 3 and 4) and GND (tracks 11 and 12) are each 2 tracks wide. The metal lines to the left of the cell indicate the 10 horizontal routing tracks (tracks 1, 2, 5–10, 13, 14). Notice that the *p*-channel and *n*-channel polysilicon gates are tied together in the center of the cell. The well contacts are short, leaving room for a poly cross-under in each base cell.

Figure 3.17 shows the metal **personalization** for a D flip-flop macro in a gate-isolated gate array using a base cell similar to that shown in Figure 3.14(a). This macro uses 20 base cells, for a total of 40 transistors, equivalent to 10 gates.

The gates of the base cells shown in Figures 3.14–3.16 are bent. The **bent gate** allows contacts to the gates to be placed on the same grid as the contacts to diffusion. The polysilicon gates run in the space between adjacent metal interconnect lines. This saves space and also simplifies the routing software.

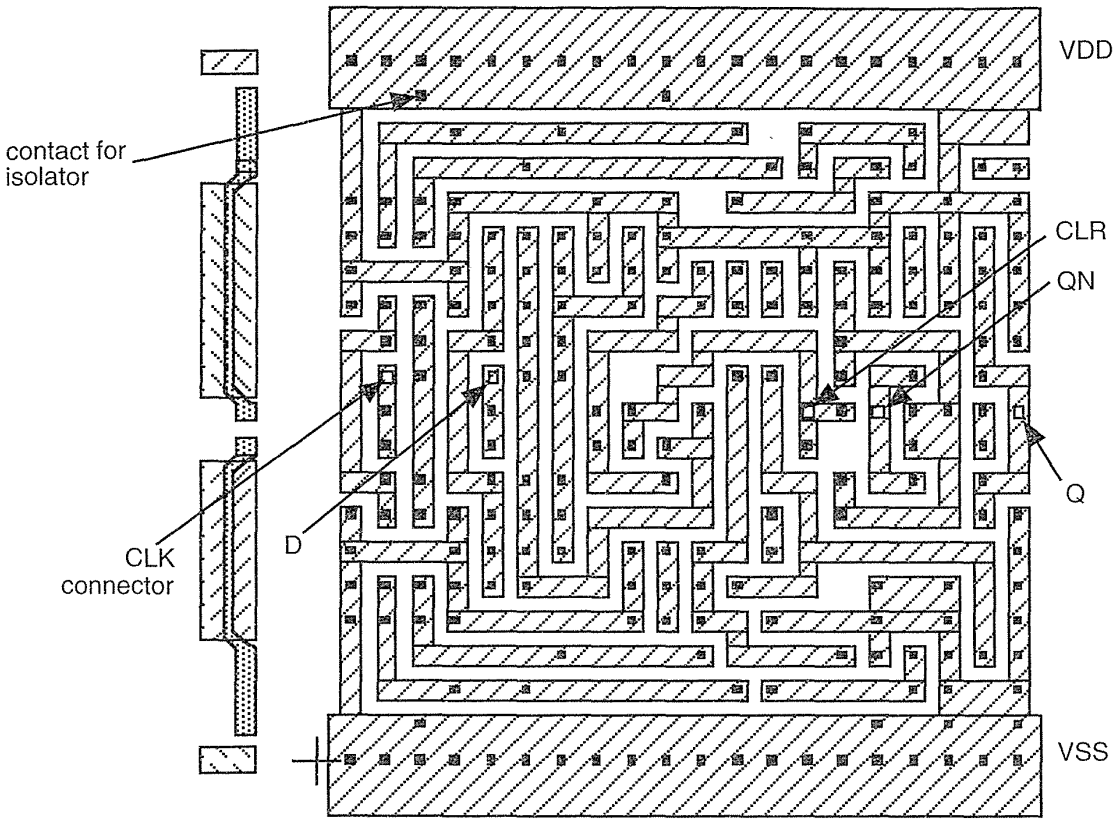


FIGURE 3.17 An example of a flip-flop macro in a gate-isolated gate-array library. Only the first-level metallization and contact pattern (the personalization) is shown on the right, but this is enough information to derive the schematic. The base cell is shown on the left. This macro is 20 tracks wide.

There are many trade-offs that determine the gate-array base cell height. One factor is the number of wires that can be run horizontally through the base cell. This will determine the capacity of the routing channel formed from an unused row of base cells. The base cell height also determines how easy it is to wire the logic macros since it determines how much space for wiring is available inside the macros.

There are other factors that determine the width of the base-cell transistors. The widths of the p-channel and n-channel transistors are slightly different in Figure 3.14(a). The p-channel transistors are 6 tracks wide and the n-channel transistors are 5 tracks wide. The ratio for this gate-array library is thus approximately 1.2. Most gate-array libraries are approaching a ratio of 1.

ASIC designers are using ever-increasing amounts of RAM on gate arrays. It is inefficient to use the normal base cell for a static RAM cell and the size of RAM on an embedded gate array is fixed. As an alternative we can change the design of the base cell. A base cell designed for use as RAM has extra transistors (either four—two *n*-channel and two *p*-channel—or two *n*-channel; usually minimum width) allowing a six-transistor RAM cell to be built using one base cell instead of the two or three that we would normally need. This is one of the advantages of the CBA (cell-based array) base cell shown in Figure 3.18.

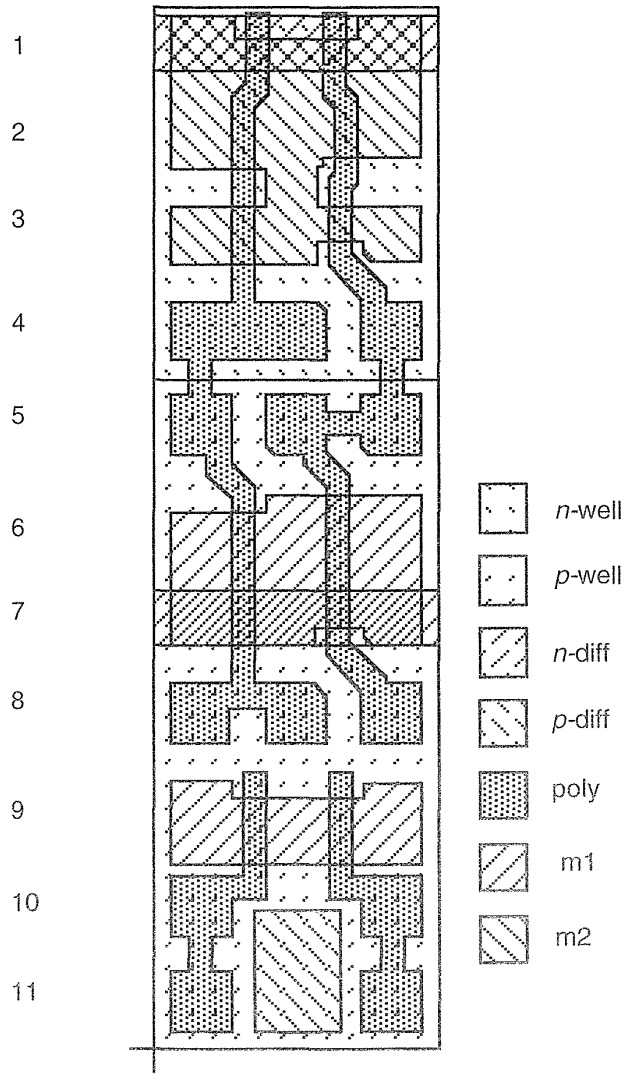


FIGURE 3.18 The SiARC/Synopsys cell-based array (CBA) basic cell.

3.7 Standard-Cell Design

Figure 3.19 shows the components of the standard cell from Figure 1.3. Each standard cell in a library is rectangular with the same height but different widths. The **bounding box** (**BB**) of a logic cell is the smallest rectangle that encloses all of the geometry of the cell. The cell BB is normally determined by the well layers. Cell connectors or terminals (the **logical connectors**) must be placed on the cell **abutment box** (**AB**). The **physical connector** (the piece of metal to which we connect wires) must normally overlap the abutment box slightly, usually by at least 1 λ, to assure connection without leaving a tiny space between the ends of two wires. The standard cells are constructed so they can all be placed next to each other horizontally with the cell ABs touching (we **abut** two cells).

A standard cell (a D flip-flop with clear) is shown in Figure 3.20 and illustrates the following features of standard-cell layout:

- Layout using 45° angles. This can save 10%–20% in area compared to a cell that uses only Manhattan or 90° geometry. Some ASIC vendors do not allow transistors with 45° angles; others do not allow 45° angles at all.
- Connectors are at the top and bottom of the cell on m2 on a routing grid equal to the vertical (m2) track spacing. This is a double-entry cell intended for a two-level metal process. A standard cell designed for a three-level metal process has connectors in the center of the cell.
- Transistor sizes vary to optimize the area and performance but maintain a fixed ratio to balance rise times and fall times.
- The cell height is 64 λ (all cells in the library are the same height) with a horizontal (m1) track spacing of 8 λ. This is close to the minimum height that can accommodate the most complex cells in a library.
- The power rails are placed at the top and bottom, maintaining a certain width inside the cell and abut with the power rails in adjacent cells.
- The well contacts (substrate connections) are placed inside the cell at regular intervals. Additional well contacts may be placed in spacers between cells.
- In this case both wells are drawn. Some libraries minimize the well or moat area to reduce leakage and parasitic capacitance.
- Most commercial standard cells use m1 for the power rails, m1 for internal connections, and avoid using m2 where possible except for cell connectors.

When a library developer creates a gate-array, standard-cell, or datapath library, there is a trade-off between using wide, high-drive transistors that result in large cells with high-speed performance and using smaller transistors that result in smaller cells that consume less power. A **performance-optimized library** with large cells might be used for ASICs in a high-performance workstation, for example. An **area-optimized library** might be used in an ASIC for a battery-powered portable computer.

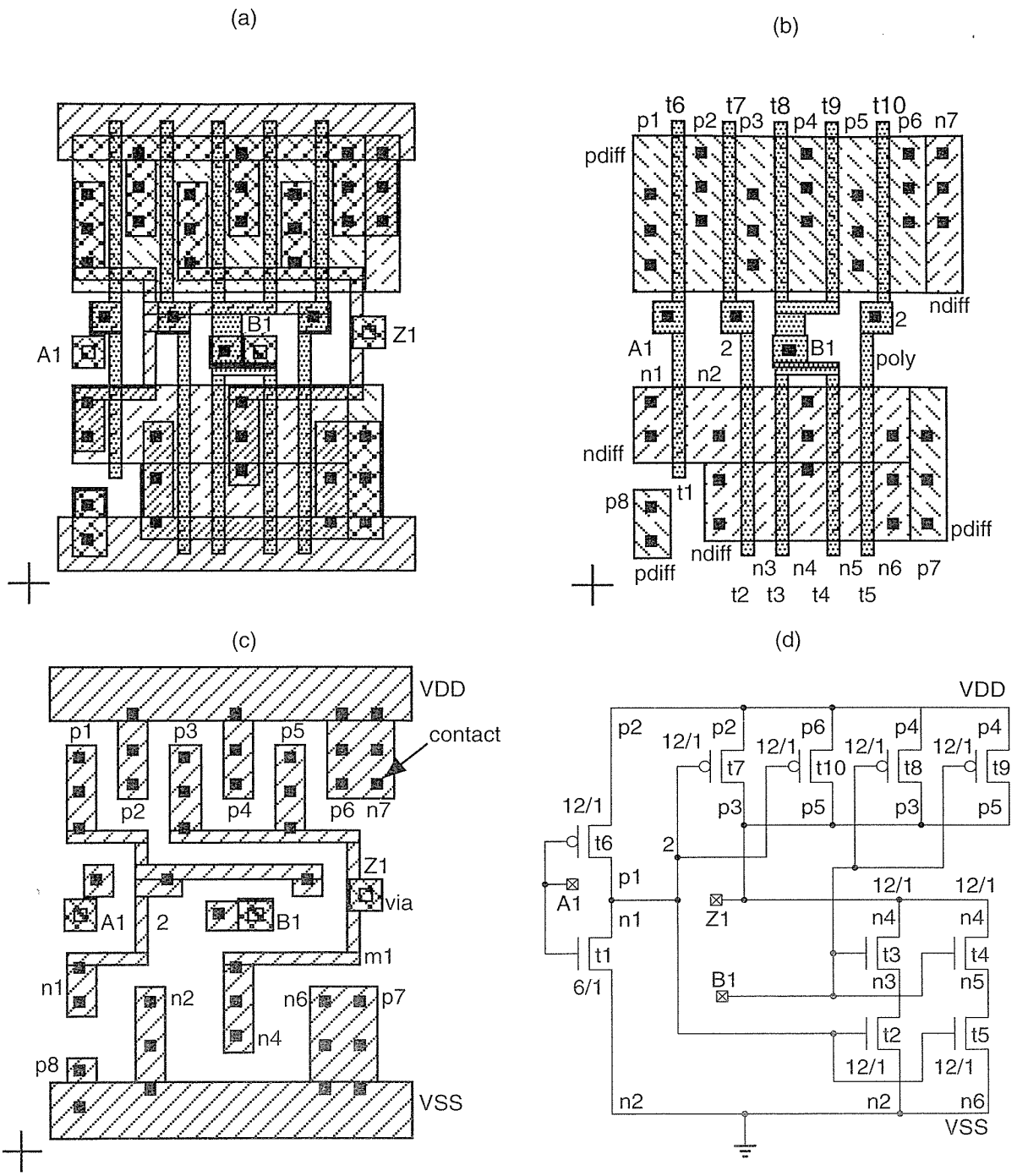


FIGURE 3.19 (a) The standard cell shown in Figure 1.3. (b) Diffusion, poly, and contact layers. (c) m1 and contact layers. (d) The equivalent schematic.

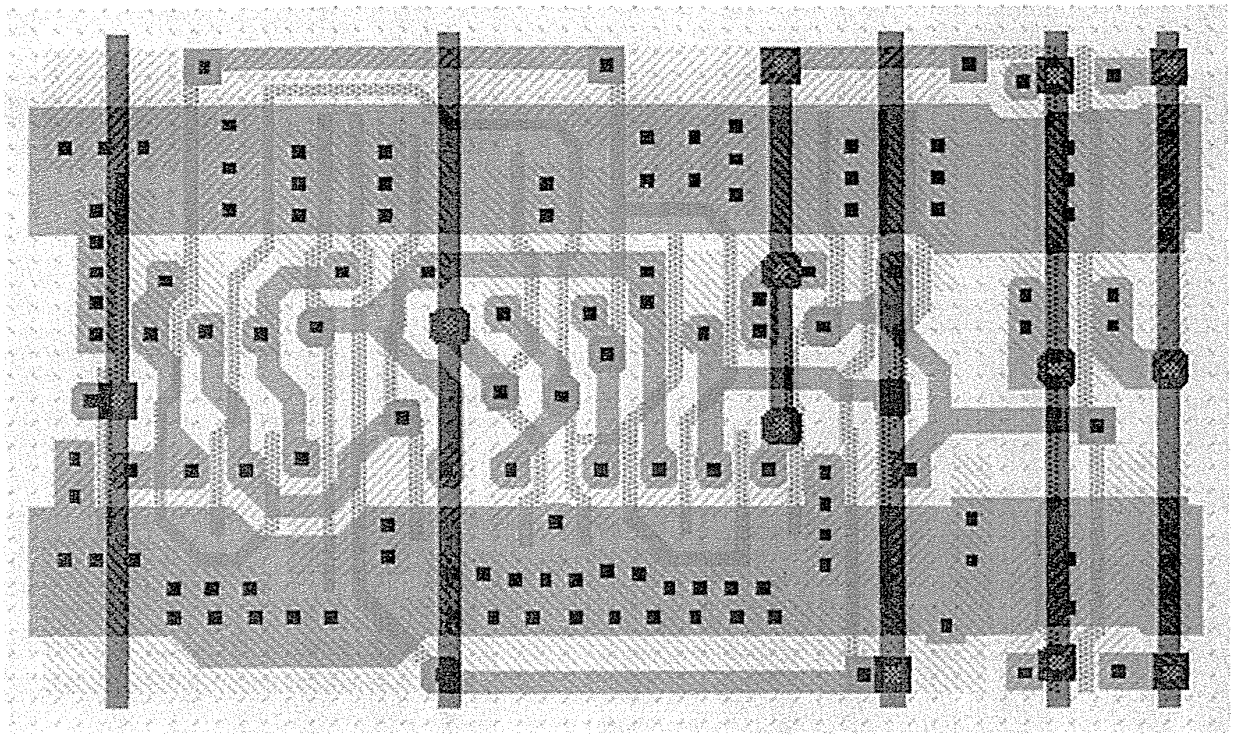


FIGURE 3.20 A D flip-flop standard cell. The wide power buses and transistors show this is a performance-optimized cell. This double-entry cell is intended for a two-level metal process and channel routing. The five connectors run vertically through the cell on m2 (the extra short vertical metal line is an internal crossover).

3.8 Datapath-Cell Design

Figure 3.21 shows a datapath flip-flop. The primary, thicker, power buses run vertically on m2 with thinner, internal power running horizontally on m1. The control signals (clock in this case) run vertically through the cell on m2. The control signals that are common to the cells above and below are connected directly in m2. The other signals (data, q, and qbar in this example) are brought out to the wiring channel between the rows of datapath cells.

Figure 3.22 is the schematic for Figure 3.21. This flip-flop uses a pair of cross-coupled inverters for storage in both the master and slave latches. This leads to a smaller and potentially faster layout than the flip-flop circuits that we use in gate-array and standard-cell ASIC libraries. The device sizes of the inverters in the datapath flip-flops are adjusted so that the state of the latches may be changed. Normally using this type of circuit is dangerous in an uncontrolled environment. However,

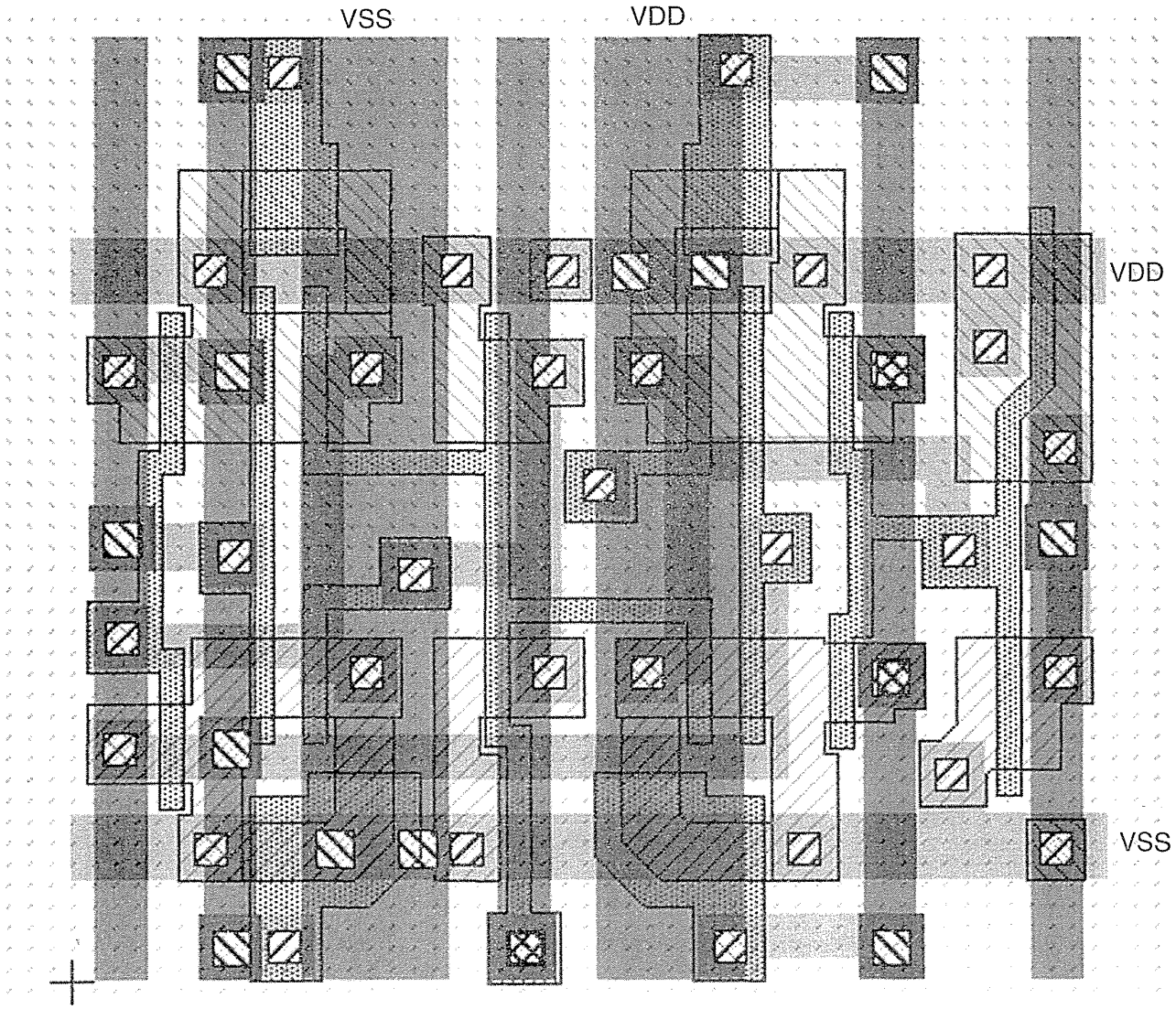


FIGURE 3.21 A datapath D flip-flop cell.

because the datapath structure is regular and known, the parasitic capacitances that affect the operation of the logic cell are also known. This is another advantage of the datapath structure.

Figure 3.23 shows an example of a datapath. Figure 3.23(a) depicts a two-level metal version showing the space between rows or slices of the datapath. In this case there are many connections to be brought out to the right of the datapath, and this causes the routing channel to be larger than normal and thus easily seen.

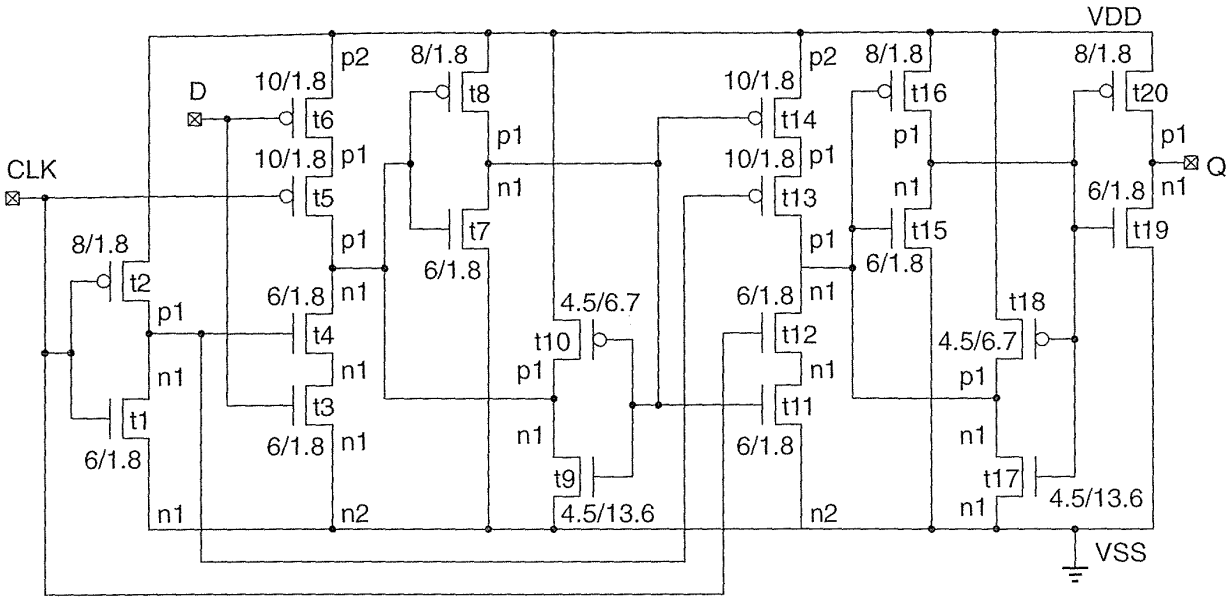


FIGURE 3.22 The schematic of the datapath D flip-flop cell shown in Figure 3.21.

Figure 3.23(b) shows a three-level metal version of the same datapath. In this case more of the routing is completed over the top of the datapath slices, reducing the size of the routing channel.

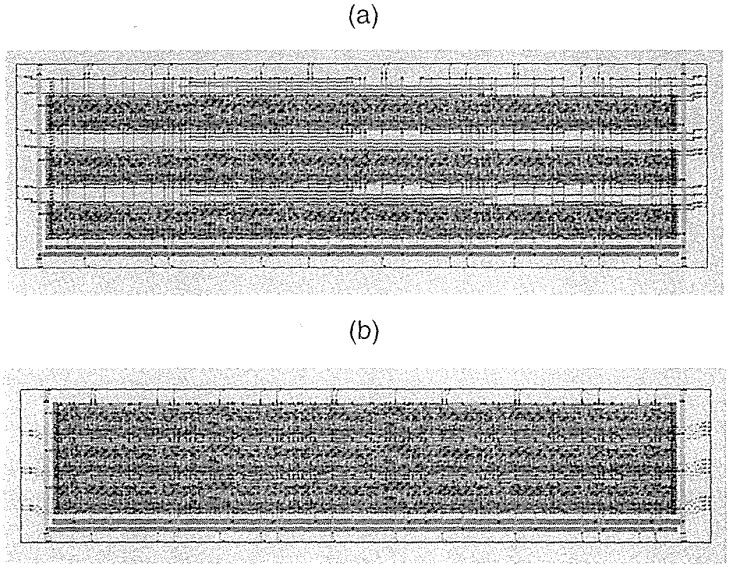


FIGURE 3.23 A datapath. (a) Implemented in a two-level metal process. (b) Implemented in a three-level metal process.

3.9 Summary

In this chapter we covered ASIC libraries: cell design, layout, and characterization. The most important concepts that we covered in this chapter were

- Tau, logical effort, and the prediction of delay
- Sizes of cells, and their drive strengths
- Cell importance
- The difference between gate-array macros, standard cells, and datapath cells

3.10 Problems

* = difficult, ** = very difficult, *** = extremely difficult

3.1 (Pull resistance, 10 min.)

a. Show that, for small V_{DS}, an n-channel transistor looks like a resistor, $R = 1/(\beta_n (V_{DD} - V_{tn}))$.

b. If $V_{GS} = V_{DD}$, $V_{DS} = 0$, and $k'_n = 200\ \mu AV^{-2}$ (equal to the n-channel transistor SPICE parameter KP in Table 2.1), find the pull resistance, R, for a 6/0.6 transistor in the linear region.

Answer: (b) 213 Ω.

3.2 (Inversion layer depth, 15 min.) In the absence of surface charge, Gauss's law demands continuity of the electric displacement vector, $\mathbf{D} = \varepsilon\mathbf{E}$, at the silicon surface, so that $\varepsilon_{ox} E_{ox} = \varepsilon_{Si} E_{Si}$, where $\varepsilon_{ox} = 3.9$, $\varepsilon_{Si} = 11.7$.

a. Assuming the potential at the surface is $V_{GS} - V_t = 2.5$ V, calculate E_{ox} and E_{Si} if $T_{ox} = 100$ Å.

b. Assume that carrier density $\propto \exp(-q\phi/kT)$, where ϕ is the potential; calculate the distance below the surface at which the inversion charge density falls to 10 percent of its value at the surface.

c. Comment on the accuracy of your answers.

Answer: (a) $2.5 \times 10^8\ Vm^{-1}$, $0.833 \times 10^8\ Vm^{-1}$. (b) 7.16 Å.

3.3 (Depletion layer depth, 15 min.) The depth of the depletion region under the gate is given by $x_d = \sqrt{(2\varepsilon_{Si}\phi_s)/(qN_A)}$, where $\phi_s = 2V_T \ln(N_A/n_i)$ is the surface potential at strong inversion. Calculate ϕ_s and x_d assuming: $\varepsilon_{Si} = 1.0359 \times 10^{-10}\ Fm^{-1}$, the substrate doping, $N_A = 1.4 \times 10^{17}\ cm^{-3}$, the **intrinsic carrier concentration** $n_i = 1.45 \times 10^{10}\ cm^{-3}$ (at room temperature), and the thermal voltage $V_T = kT/q = 25.9$ mV.

Answer: 0.833 V, 900 Å.

3.4 (Logical effort, 45 min.) Calculate the logical effort at each input of an AOI122 cell. Find an expression that allows you to calculate the logical effort for each input of an AOI*nnnn* cell for $n = 1, 2, 3$.

3.5 (Gate-array macro design, 120 min.) Draw a 1X drive, two-input NAND cell using the gate-array base cells shown in Figures 3.14(a)–3.16 (lay a piece of thin paper over the figures and draw the contacts and metal personalization only). Label the inputs and outputs. Lay out a 1X drive, four-input NAND cell using the same base array cells. Now lay out a 2X drive, four-input NAND cell (think about this one). Make sure that you size your transistors properly to balance rise times and fall times.

3.6 (Flip-flop library, 20 min.) Suppose we wish to build a library of flip-flops. We want to have flops with: positive-edge and negative-edge triggering: clear, preset (either, both, or neither); synchronous or asynchronous reset and preset controls if present (but not mixed on the same flip-flop); all flip-flops with or without scan as an option; flip-flops with Q and Qbar (either or both). How many flip-flops is that? (***) How would you attempt to prioritize which flip-flops to include in a library?

3.7 (AOI and OAI cell ratios, 30 min.) In Figure 2.13(c) we adjusted the sizes of the transistors assuming that there was only one path through the *n*-channel and *p*-channel stacks. Suppose that *p*-channel transistors A, B, C, and D are all on and *p*-channel transistor E turns on. What is the equivalent resistance of the *p*-channel stack in this case?

3.8 (**Eight-input AND, 60 min.) This question is an example in the paper by Sutherland and Sproull [1991] on logical effort. Figure 3.24 shows three different ways to design an eight-input AND cell, using NAND and NOR cells.

a. Find the logical effort at each input for A, B, C. Assume a logic ratio of 2.

b. Find the parasitic delay for A, B, C. Assume the parasitic delay of an inverter is 0.6.

c. Show that the path delays are given by the following equations where H is the path electrical effort, if we ignore the nonideal delays:

(i) $2\,(3.33H)^{0.5} + 5.4$ (alternative A)
(ii) $2\,(3.33H)^{0.5} + 3.6$ (alternative B)
(iii) $4\,(2.96H)^{0.25} + 4.2$ (alternative C)

d. Use these equations to determine the best alternative for $H = 2$ and $H = 32$.

3.9 (Special logic cells, 30 min.) Many ASIC cell libraries contain "special" logic cells. For example the Compass libraries contain a two-input NAND cell with an inverted input, FN01 = (A + B'). This saves routing area, is faster than using two separate cells, and is useful because the combination of a two-input NAND gate with one inverted input is heavily used by synthesis tools. Other "special" cells include:

- FN02 = MAJ3 = (A·B + A·C + B·C)'
- FN03 = AOI2-2 = ((A'·B') + (C·D))' = (A + B)(C' + D') = OA2-2

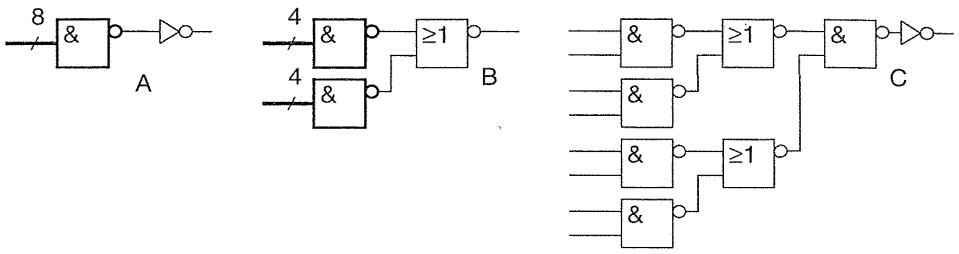


FIGURE 3.24 An eight-input AND cell (Problem 3.8).

- FN04 = OAI2-2
- FN05 = A·B' = (A' + B)'

a. Draw schematics for these cells.

b. Calculate the logical effort and logical area for each cell.

c. Can you explain where and why these cells might be useful?

3.10 (Euler paths, 60 min.) There are several ways to arrange the stacks in the AOI211 cell shown in Figure 3.25. For example, the *n*-channel transistor A can be below B without altering the function. Which arrangement would you predict gives a faster delay from A to Z and why? The *p*-channel transistors A and B can be above or below transistors C and D. How many distinct ways of arranging the transistors are there for this cell? What effect do the different arrangements have on layout? What effects do these different arrangements have on the cell performance?

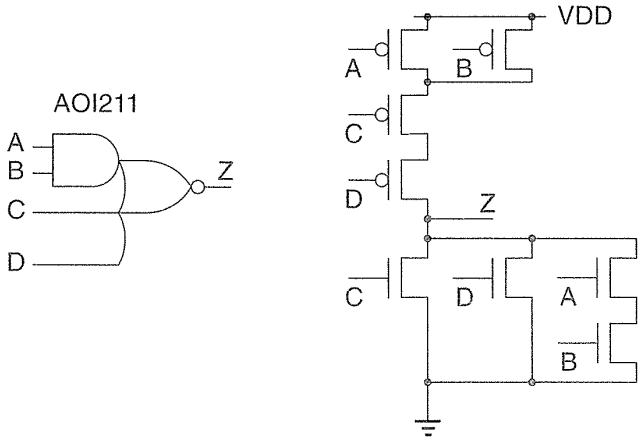


FIGURE 3.25 There are several ways to arrange the transistors in this AOI211 cell (Problem 3.10).

3.11 (*AOI and OAI cell efficiency, 60 min.) A standard-cell library data book contains the following data:

- AOI221: $t_R = 1.06–1.15$ ns; $t_F = 1.09–1.55$ ns; $C_{in} = 0.21–0.28$ pF; $W_C = 28.8$ μm
- OAI221: $t_R = 0.77–1.05$ ns; $t_F = 0.81–0.96$ ns; $C_{in} = 0.25–0.39$ pF; $W_C = 22.4$ μm

(W_C is the cell width, the cell height is 25.6 μm.) Calculate the **(a)** logical effort and **(b)** logical area for the AOI221 and OAI221 cells.

The implementation of the OAI221 in this library uses a single stage,

OAI221 = OAI221(a1, a2, b1, b2, c),

whereas the AOI221 uses the following multistage implementation:

AOI221 = NOT(NAND(NAND(a1, a2), AOI21(b1, b2, c))).

(c) What are the alternative implementations for these two cells? **(d)** From your answers attempt to explain the implementations chosen.

3.12 (**Logical efficiency, 60 min.) Extending Problem 3.11, let us compare an AOI33 with an OAI33 cell. **(a)** Calculate the logical effort and **(b)** logical areas for these cells.

The AOI33 uses a single-stage implementation as follows:

AOI33 = AOI33(a1, a2, a3, b1, b2, b3).

The OAI33 uses the following multistage implementation:

OAI33 = NOT[NOR[NOR(a1, a2, a3), NOR(b1, b2, b3)]].

(c) Calculate the path delay, D, as a function of path electrical effort, H, for both of these implementations ignoring parasitic and nonideal delays. **(d)** Use Eq. 3.42 to calculate the optimum path delay for these cells. **(e)** Compare and explain the differences between your answers to parts d and e for $H = 1$, 2, 4, and 8.

The timing data from the data book is as follows (the cell height is 25.6 μm):

- AOI33: $t_R = 0.70–1.06$ ns; $t_F = 0.72–1.15$ ns; $C_{in} = 0.21–0.28$ pF; $W_C = 35.2$ μm
- OAI33: $t_R = 1.06–1.70$ ns; $t_F = 1.42–1.98$ ns; $C_{in} = 0.31–0.36$ pF; $W_C = 48$ μm

(f) How does this data compare with your theoretical analysis?

3.13 (EXOR cells and logical effort, 60 min.) Show how to implement a two-input EXOR cell using an AOI22 and two inverters. Using logical effort, compare this with an implementation using an AOI21 cell and a NOR cell.

3.14 (***XNOR cells, 60 min.) Table 3.3 shows the implementation of XNOR cells in a standard-cell library. Analyze this data using the concept of logical effort.

3.15 (***Extensions to logical effort, 60 min.) The **path branching effort** B is the product of branching efforts:

$$B = \prod_{i \in \text{path}} b_i. \tag{3.47}$$

TABLE 3.3 Implementations of XNOR cells in CMOS (Problem 3.14).

Cell	Implementation
Library 1: XNOR2D1	NAND[OR(a1,a2),NAND(a1,a2)]
Library 2: XNOR2D1	NOT[NOT[MUX[a1, NOT(a1),a2)]]
Library 1: XNOR2D2	NOT[NOT[MUX(a1,NOT(a1),a2)]]
Library 2: XNOR2D2	NAND[OR(a1,a2),NAND(a1,a2)]
Library 1: XNOR3D1	NOT[NOT[MUX(a1, NOT(a1), NOT(MUX(a3, NOT(a3),a2)))]]
Library 1: XNOR3D2	NOT[NOT[MUX(a1, NOT(a1), NOT(MUX(a3, NOT(a3),a2)))]]

The **branching effort** is the ratio of the on-path plus off-path capacitance to the on-path capacitance. The **path effort** F becomes the product of the path electrical effort, path branching effort, and path logical effort:

$$F = GBH. \tag{3.48}$$

Show that the path delay D is

$$D = \sum_{i \in \text{path}} g_i b_i h_i + \sum_{i \in \text{path}} p_i. \tag{3.49}$$

(***) Show that the optimum path delay is then

$$\hat{D} = NF^{1/N} + P = N(GBH)^{1/N} + P. \tag{3.50}$$

3.16 (*Circuits from layout, 120 min.) Figure 3.26 shows a D flip-flop with clear from a 1.0 μm standard-cell library. Figure 3.27 shows two layout views of this D flip-flop. Construct the circuit diagram for this flip-flop, labeling the nodes and transistors as shown. Include the transistor sizes—use estimates for transistors with 45° gates—you only need W/L values, you can assume the gate lengths are all $L = 2\lambda$, equal to the minimum feature size. Label the inputs and outputs to the cell and identify their functions.

3.17 (Flip-flop circuits, 30 min.) Draw the circuit schematic for a positive-edge–triggered D flip-flop with active-high set and reset (base your schematic on Figure 2.18a, a negative-edge–triggered D flip-flop). Describe the problem when both SET and RESET are high.

If we want an active-high set or reset we can: (1) use an inverter on the set or reset signal or (2) we can substitute NOR cells. Since NOR cells are slower than NAND cells, which we do depends on whether we want to optimize for speed or area.

Thus, the largest flip-flop would be one with both Q and QN outputs, active high set and reset—requiring four TX gates, three inverters (four of the seven we

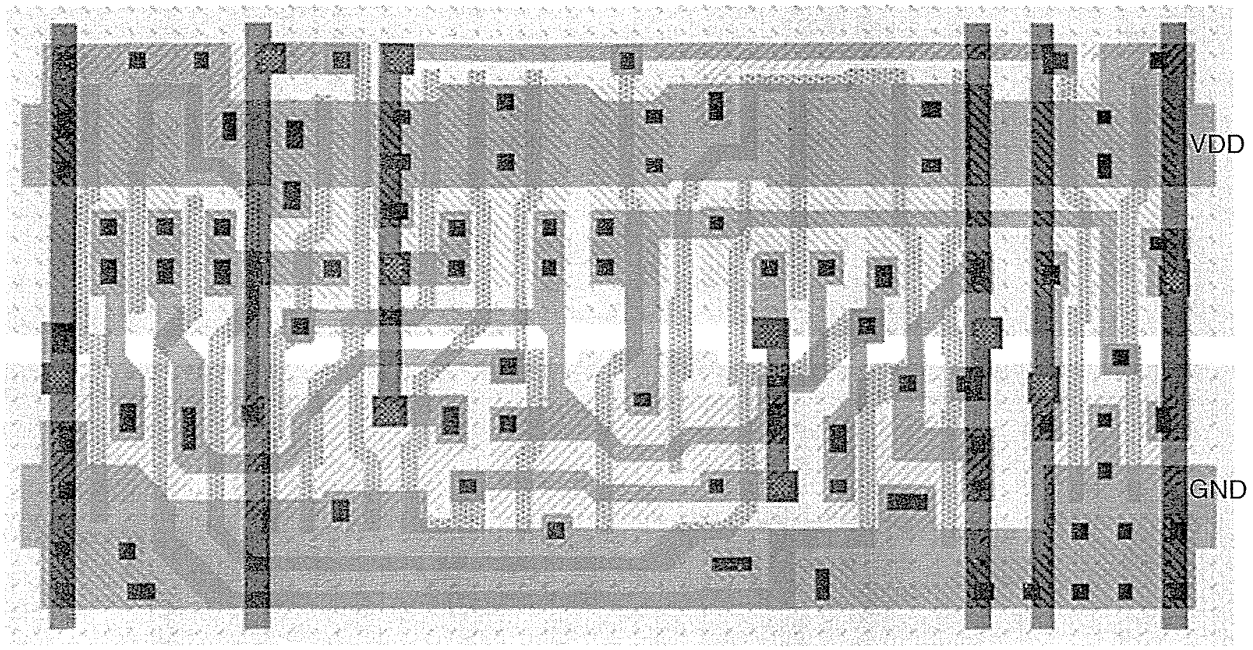


FIGURE 3.26 A D flip-flop from a 1.0 μm standard-cell library (Problem 3.16).

normally need are replaced with NAND cells), four NAND cells, and two inverters to invert the set and reset, making a total of 34 transistors, or 8.5 gates.

3.18 (Set and reset, 10 min.) Show how to add a **synchronous set** or a **synchronous reset** to the flip-flop of Figure 2.18(a) using a two-input MUX.

3.19 (Clocked inverters, 45 min.) Using PSpice compare the delay of an inverter with transmission gate with that of a clocked inverter using the G5 process SPICE parameters from Table 2.1.

3.20 (S-R, T, J-K flip-flops, 30 min.) The **characteristic equation** for a D flip-flop is $Q_{t+1} = D$. The characteristic equation for a J-K flip-flop is $Q_{t+1} = J(Q_t)' + K'Q_t$.

a. Show how you can build a J-K flip-flop using a D flip-flop.

b. The characteristic equation for a T flip-flop (toggle flip-flop) is $Q_{t+1} = (Q_t)'$. Show how to build a T flip-flop using a D flip-flop.

c. The characteristic equation does not show the timing behavior of a sequential element—the characteristic equation for a D latch is the same as that for a D flip-flop. The characteristic equation for an S-R latch and an S-R flip-flop is $Q_{t+1} = S + R'Q_t$. An S-R flip-flop is sometimes called a pulse-triggered flip-flop. Find out the behavior of an S-R latch and an S-R flip-flop and describe the differences between these elements and a D latch and a D flip-flop.

d. Explain why it is probably not a good idea to use an S-R flip-flop in an ASIC design.

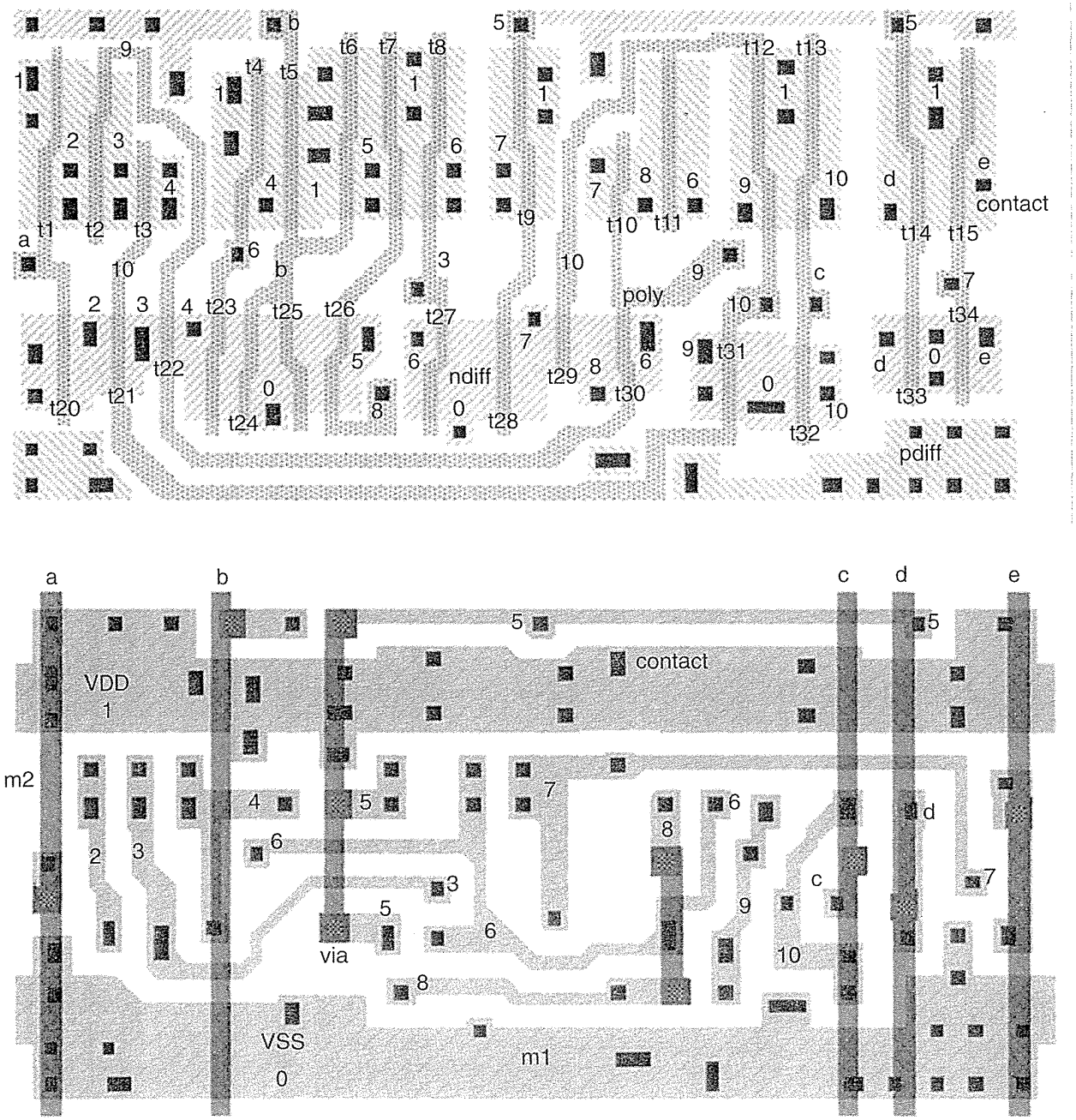


FIGURE 3.27 (Top) A standard cell showing the diffusion (*n*-diffusion and *p*-diffusion), poly, and contact layers (the *n*-well and *p*-well are not shown). (Bottom) Shows the m1, contact, m2, and via layers. Problem 3.16 traces this circuit for this cell.

3.21 (**Optimum logic, 60 min.) Suppose we have a fixed logic path of length n_1. We want to know how many (if any) buffer stages we should add at the output of

this path to optimize the total path delay given the output load capacitance.

a. If the total number of stages is N (logic path of length n_1 plus $N-n_1$ inverters), show that the total path delay is

$$\hat{D} = NF^{1/N} + \sum_{i=1}^{n_1} (p_i + q_i) + (N - n_1)\,(p_{inv} + q_{inv})\,. \tag{3.51}$$

The optimum number of stages is given by the solution to the following equation:

$$\frac{\partial \hat{D}}{\partial N} = \frac{\partial}{\partial N}(NF^{1/N} + (N - n_1)\,(p_{inv} + q_{inv}))\; = 0\,. \tag{3.52}$$

b. Show that the solutions to this equation can be written in terms of $F^{1/\hat{N}}$ (the optimum stage effort) where $\hat{N}$ is the optimum number of stages:

$$F^{1/\hat{N}}(1 - \ln F^{1/\hat{N}}) + (p_{inv} + q_{inv})\; = 0. \tag{3.53}$$

3.22 (XOR and XNOR cells, 60 min.) Table 3.4 shows the implementations of two- and three-input XOR cells in an ASIC standard-cell library (D1 are the 1X drive cells, and D2 are the 2X drive versions). Can you explain the choices for the two-input XOR cell and complete the table for the three-input XOR cell?

TABLE 3.4 Implementations of XOR cells (Problem 3.22).

Cell	Actual implementation[1]	Alternative implementation(s)
XOR2D1	AOI21[a1, a2, NOR(a1,a2)]	NOT[MUX(a1, NOT(a1), a2)]
		AOI22(a1, a2, NOT(a1), NOT(a2))
XOR2D2	NOT[MUX(a1, NOT(a1), a2)]	AOI21[a1, a2, NOR(a1, a2)]
		AOI22(a1, a2, NOT(a1), NOT(a2))
XOR3D1	NOT[MUX[a1, NOT(a1), NOT(MUX(a3, NOT(a3), a2))]]	?
XOR3D2	NOT[MUX[a1, NOT(a1), NOT(MUX(a3, NOT(a3), a2))]]	?

[1]MUX(a, b, c) = a·c + b·c'

3.23 (Library density, 10 min.) Derive an upper limit on cell density as follows: Assume a chip consists only of two-input NAND cells with no routing channels between rows (often achievable in a 3LM process with over-the-cell routing).

a. Explain how many vertical tracks you need to connect to a two-input NAND cell, assuming each connection requires a separate track.

b. If the NAND cell is 64λ high with a vertical track width of 8λ, calculate the NAND cell area, carefully explaining any assumptions.

c. Calculate the cell density (in gate/mil^2) for a 0.35 μm process, λ = 0.175 μm.

Answer: 3 tracks, 47 μm^2, 13.7 gates/mil^2 or 21 × 10^3 gates/mm^2.

3.24 (Gate-array density, 20 min.) The LSI Logic 10k and 100k gate arrays use a four-transistor base cell, equivalent to 1 gate, that is 12 tracks high and 3 tracks wide.

a. If a metal track is 8λ, where λ = 0.75 μm for a 1.5 μm technology, calculate the area of the LSI Logic base cell A_L in mil^2.

b. If we could use every base cell in the gate array, the cell density would be $D_G = 1/A_L$. Assume that, because of routing area and inefficiency of the gate array, we can use only 50 percent of the base cells for logic. What is D_G for the LSI Logic 1.5 μm array?

c. Chip cell density D_G is about 1.0 gate/mil^2 for a 1 μm technology (a two-input NAND cell occupies an area 25 μm on a side in a technology whose transistors are 1 μm long). This can change by a factor of 2 or more for a gate-array/standard-cell ASIC or high-density/high-performance library. Assume that cell density D_G scales ideally with technology. If the minimum feature size of a technology is 2λ, then $D_G \propto 1/\lambda^2$. Thus, for example, a 1.5 μm technology should have a cell density of roughly (1/1.5)^2 gates/mil^2. How does this agree with your estimate for the LSI Logic array?

3.25 (SiArc RAM, 10 min.) Suppose we need 16 k-bit of SRAM and 20 k-gate of random logic on a channelless gate array. Assume a base cell with four transistors and that we can build a RAM cell using two of these base cells. The RAM bits will require 32k base cells and the random logic will require 20k base cells. Suppose the base cell area is 12 tracks high, 3 tracks wide, and the horizontal and vertical track spacing is equal at 8λ.

a. Calculate the total area of the base cells we need. Now suppose we redesign the gate-array base cell so that we can build a RAM bit cell using a single base cell that is 20 tracks high, 3 tracks wide, and has 4 logic cell transistors and 4 RAM cell transistors. Assume that since the base cell now contains 8 transistors we only need 12 k base cells to implement 20 k-gate of random logic (the new base cell is less efficient than the old cell for implementing random logic).

b. Calculate the base cell area using the new base cell design.

c. Comment.

Answer: 1.2 × 10^8 λ^2, 1.1 × 10^8 λ^2.

3.26 (***Gate-array base cell, 60 min.) Figure 3.28 shows a simple gate-array base cell. Use the design rules shown in Table 2.16 (Problem 2.33) to calculate the minimum size of this base cell. Do this by determining which design rules apply to

the labels shown adjacent to each space or width in the figure. In most cases each of the spaces is determined by a single rule related to the region labeled, for example, the contact width labeled 'cc' is 2λ determined by rule C.1, the exact contact size. There is one exception, shown in the figure. Space 'aa' (bounding box, BB, to edge of pdiff) and width 'bb' (edge of pdiff to edge of contact) are determined by the minimum space labeled 'xx' (bounding box, BB, to poly edge) and width 'yy' (edge of poly to edge of contact). Space 'xx' is one half of the poly to poly spacing over field (rule P.4) because two base cells abut as shown in the figure. Width 'yy' is equal to the minimum poly overlap of contact (rule C.3). The distance 'aa + bb' is thus determined by the minimum distance 'xx + yy', as shown. The other distances are more straightforward to determine.

Answer: 40λ high by 26.25λ wide.

3.27 (CIF, 15 min.) Here is the part of the CIF for a standard cell that describes the *n*-well (CWN) and *p*-well (CWP) structure. The statement `B length height xCenter, yCenter` is CIF for a box (CIF dimensions are in centimicrons, 0.01 μm):

```
DS1;LCWN;B6000
1560 13600,3660;B2480 60 11840,2850;B2320 60 15440,2850;LCWP;B680 60
13740,2730;B6000 1380 13600,2010;
```

a. Draw the wells and BB. Label the dimensions in microns and λ (λ = 0.4 μm).

b. This is a double-entry cell with m2 connectors at top and bottom. For this cell library the cell AB is 3λ (120 centimicrons, determined by the well rules) inside the cell BB on all sides. What is the size of the cell AB in microns and λ?

c. The vertical (m2) routing pitch (the distance between centers of adjacent vertical m2 interconnect lines) is equal to the vertical track spacing and is 8λ (320 centimicrons). How many vertical tracks are there in this cell?

3.28 (CIF, 60 min.) Figure 3.29 shows an example of CIF that describes a single rectangle (box) of m1 with an accompanying label.

The CIF code has the following meaning:

- Lines 1–5 are CIF comments.
- Line 6 is a **definition start** for symbol 1 and marks the beginning of a **symbol definition** (a symbol is a piece of layout, symbol numbers are unique identifiers). The integers 2 and 8 define a **scaling factor** 2/8 (= 0.25) to be applied to distance measurements (the CIF unit, after scaling, is a **centimicron** or 0.01 μm).
- Line 7 is a **user extension** or expansion (all extensions begin with a digit). L-Edit uses user extension 9 for cell names (`Cell0` in this case).
- Line 8 is a user extension for a **cell label** located on layer `CM` (first-level metal in this technology) located at $x = 60$ units, $y = 180$ units (60, 180).

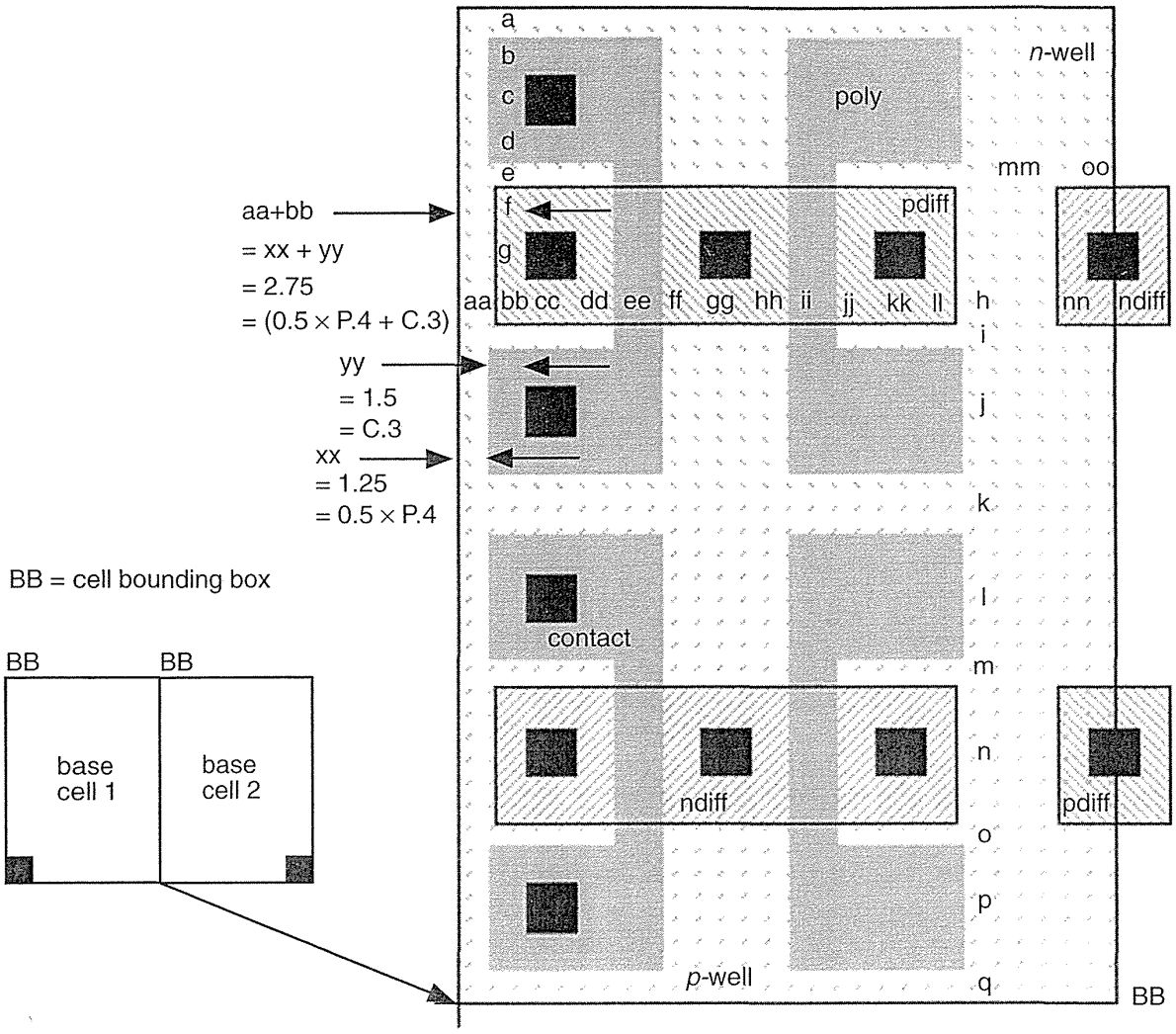


FIGURE 3.28 A simple gate-array base cell (Problem 3.26).

Applying the scaling factor of 0.25, this translates to (15, 45) in centimicrons or (0.5, 1.5) in lambda.

- Line 9 is a **layer specification** or command (begins with `L`).
- Line 10 is a **box command** and describes a box with (in order) length, `L`, of 240 units; width, `W`, of 120 units; and center at $x = 120$ units and $y = 300$ units. Applying the `DS` scaling factor of 0.25 gives `L` = 60, `W` = 30, center = (30, 75)(centimicrons) or `L` = 2, `W` = 1, center = (1, 2.5) in lambda.
- Line 11 is the **definition finish** (`DS` and `DF` must be paired).
- Line 12 is the **end command**.

```
(CIF written by the Tanner Research
layout editor: L-Edit);                    --1
(TECHNOLOGY: VLSIcmn6);                    --2
(DATE: Thu, Jun 27, 1996);                 --3
(FABCELL: NONE);                           --4
(SCALING: 1 CIF Unit = 1/120 Lambda, 1
Lambda = 3/10 Microns);                    --5
DS1 2 8;                                   --6
9 Cell0;                                   --7
94 LabelText 60 180 CM;                    --8
LCM;                                       --9
B 240 120 120 300;                         --10
DF;                                        --11
E                                          --12
```

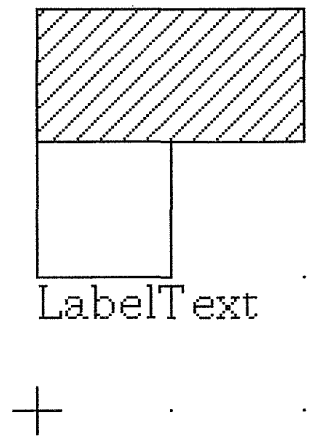


FIGURE 3.29 A simple CIF example (Problem 3.28).

You receive a CIF file whose mask-layer names are different from those in the technology file that you are using. The mapping between layer names is shown in Table 3.5.

a. Write an `awk` or `sed` script (or use another automated editing technique) to change the layer names. At this point you realize that there are several layer names (`LTRAN`, `LESD`) in the input file that are not required (or recognized) by your layout software (these particular examples are for software to recognize unused transistors in a gate array, and for an ESD implant in I/O devices).

b. (**) Enhance your script to completely remove an unwanted layer from the CIF file. There are some comments and CIF constructs that are not supported by your editor. Here is one example:

```
(BB: 39.2,82.6 72.8,122.5 in lambda);
```

Comments in this format specify the AB and BB for the cell. Other CIF user extensions, not recognized by your software, are used for labels for power supplies and connectors:

```
4A 1680 3360 2800 4844;
4M a 1 2292 4028 2356 4092 CM2;
4M z 4 2639 4090 2703 4154 CM2;
4X vdd 2 2800 4774 180 * * metal;
```

c. (**) Add code to remove all these constructs from the CIF file.

TABLE 3.5 Mapping CIF layer names (Problem 3.28).[1]

Input mask label	MOSIS mask label	Input mask label	MOSIS mask label	Input mask label	MOSIS mask label
LCNW	CWN	LCND[2]	CSN	LCM2	CMS
LCPW	CWP	LCPD[2]	CSP	LCC2	CVS
none[3]	CAA	LCC[4]	CCA	LCM3	CMT
LCP	CPG	LCM	CMF	none	COG

[1]This mapping is for input to a layout editor; the CIF may have to be modified again when written out from the layout editor.
[2]Map the input diffusion layers to the implant select layers. On output from the layout editor these layers should be sized up to generate the "real" implant select layers.
[3]There is no active layer in the input. Instead use the diffusion layers.
[4]There is only one contact layer in the input; map all contacts to CAA.There is no easy way to generate the MOSIS CCP layer. This prevents handling of poly and diffusion contacts separately.

3.11 Bibliography

The first part of this chapter is covered in greater detail in Weste and Eshraghian [1993]. The experiments presented in this chapter may be reproduced using PSpice and Probe from MicroSim (`http://www.microsim.com`). A free CD-ROM is available from MicroSim containing PC versions of their software together with reference manuals in Adobe Acrobat format that are readable on all platforms. Other PSpice and Probe versions are available online including the Apple Macintosh version used in this book (which requires a math coprocessor). Mukherjee [1986] covers CMOS process and fabrication issues. Analog ASIC design is covered by Haskard and May [1988] and Trontelj et al. [1989]. Chen [1990] and Uyemura [1992] provide more depth on analysis of combinational and sequential logic design. The book by Diaz [1995] includes material on I/O cell design for ESD protection that is hard to find. The patent literature is the best reference for high-speed and quiet I/O design. Wakerly's book [1994] on digital design is an excellent reference for logic design in general (including sequential logic, metastability, and binary arithmetic), though it emphasizes PLDs rather than ASICs.

3.12 References

Chen, J. Y. 1990. *CMOS Devices and Technology for VLSI.* Englewood Cliffs, NJ: Prentice-Hall, 348 p. ISBN 0-13-138082-6. TK7874.C523.

Diaz, C. H., et al. 1995. *Modeling of Electrical Overstress in Integrated Circuits.* Norwell, MA: Kluwer Academic, 148 p. ISBN 0-7923-9505-0. TK7874.D498. Includes 101 references. Good introduction to ESD problems and models. Most of the book deals with thermal analysis and thermal stress modeling.

Haskard, M. R., and I. C. May. 1988. *Analog VLSI Design: nMOS and CMOS.* Englewood Cliffs, NJ: Prentice-Hall, 243 p. ISBN 0-13-032640-2. TK7874.H392.

Mukherjee, A. 1986. *Introduction to nMOS and CMOS VLSI Systems Design.* Englewood Cliffs, NJ: Prentice-Hall, 370 p. ISBN 0-13-490947-X. TK7874.M86.

Sutherland, I. E., and R. F. Sproull. 1991. "Logical effort: designing for speed on the back of an envelope." In *Proceedings of the Advanced Research in VLSI,* Santa Cruz, CA, pp. 1–16. This reference may be hard to find, but similar treatments (without the terminology of logical effort) are given in Mead and Conway, or Weste and Eshraghian.

Trontelj, J., et al. 1989. *Analog Digital ASIC Design.* New York: McGraw-Hill, 249 p. ISBN 0-07-707300-2. TK7874.T76.

Uyemura, J. P. 1992. *Circuit Design for CMOS VLSI.* Boston: Kluwer Academic Publishers, 450 p. ISBN 0-7923-9184-5. TK7874.U93. See also: J. P. Uyemura, *Fundamentals of MOS Digital Integrated Circuits,* Reading, MA: Addison-Wesley, 1988, 624 p. ISBN 0-201-13318-0. TK7874.U94. Reference for basic circuit equations related to NMOS and CMOS logic design.

Wakerly, J. F. 1994. *Digital Design: Principles and Practices.* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 840 p. ISBN 0-13-211459-3. TK7874.65.W34. Introduction to logic design covering: binary arithmetic, CMOS and TTL, combinational logic, PLDs, sequential logic, memory, and the IEEE standard logic symbols.

Weste, N. H. E., and K. Eshraghian. 1993. *Principles of CMOS VLSI Design: A Systems Perspective.* 2nd ed. Reading, MA: Addison-Wesley, 713 p. ISBN 0-201-53376-6. TK7874.W46. See also the first edition of this book.

PROGRAMMABLE ASICs

4

There are two types of programmable ASICs: programmable logic devices (PLDs) and field-programmable gate arrays (FPGAs). The distinction between the two is blurred. The only real difference is their heritage. PLDs started as small devices that could replace a handful of TTL parts, and they have grown to look very much like their younger relations, the FPGAs. We shall group both types of programmable ASICs together as FPGAs.

An FPGA is a chip that you, as a systems designer, can program yourself. An IC foundry produces FPGAs with some connections missing. You perform design entry and simulation. Next, special software creates a string of bits describing the extra connections required to make your design—the **configuration file**. You then connect a computer to the chip and program the chip to make the necessary connections according to the configuration file. There is no customization of any mask level for an FPGA, allowing the FPGA to be manufactured as a standard part in high volume.

FPGAs are popular with microsystems designers because they fill a gap between TTL and PLD design and modern, complex, and often expensive ASICs. FPGAs are ideal for prototyping systems or for low-volume production. FPGA vendors do not need an IC fabrication facility to produce the chips; instead they contract IC foundries to produce their parts. Being fabless relieves the FPGA vendors of the huge burden of building and running a fabrication plant (a new submicron fab costs hundreds of millions of dollars). Instead FPGA companies put their effort into the FPGA

architecture and the software, where it is much easier to make a profit than building chips. They often sell the chips through distributors, but sell design software and any necessary programming hardware directly.

All FPGAs have certain key elements in common. All FPGAs have a regular array of basic logic cells that are configured using a **programming technology**. The chip inputs and outputs use special I/O logic cells that are different from the basic logic cells. A programmable interconnect scheme forms the wiring between the two types of logic cells. Finally, the designer uses custom software, tailored to each programming technology and FPGA architecture, to design and implement the programmable connections. The programming technology in an FPGA determines the type of basic logic cell and the interconnect scheme. The logic cells and interconnection scheme, in turn, determine the design of the input and output circuits as well as the programming scheme.

The programming technology may or may not be permanent. You cannot undo the permanent programming in **one-time programmable** (**OTP**) FPGAs. Reprogrammable or erasable devices may be reused many times. We shall discuss the different programming technologies in the following sections.

4.1 The Antifuse

An **antifuse** is the opposite of a regular fuse—an antifuse is normally an open circuit until you force a **programming current** through it (about 5 mA). In a poly–diffusion antifuse the high current density causes a large power dissipation in a small area, which melts a thin insulating dielectric between polysilicon and diffusion electrodes and forms a thin (about 20 nm in diameter), permanent, and resistive silicon **link**. The programming process also drives dopant atoms from the poly and diffusion electrodes into the link, and the final level of doping determines the resistance value of the link. Actel calls its antifuse a programmable low-impedance circuit element (**PLICE**™).

Figure 4.1 shows a poly–diffusion antifuse with an **oxide–nitride–oxide** (**ONO**) dielectric sandwich of: silicon dioxide (SiO_2) grown over the *n*-type antifuse diffusion, a silicon nitride (Si_3N_4) layer, and another thin SiO_2 layer. The layered ONO dielectric results in a tighter spread of blown antifuse resistance values than using a single-oxide dielectric. The effective electrical thickness is equivalent to 10 nm of SiO_2 (Si_3N_4 has a higher dielectric constant than SiO_2, so the actual thickness is less than 10 nm). Sometimes this device is called a fuse even though it is an *anti*fuse, and both terms are often used interchangeably.

The fabrication process and the programming current control the average resistance of a blown antifuse, but values vary as shown in Figure 4.2. In a particular technology a programming current of 5 mA may result in an average blown antifuse resistance of about 500 Ω. Increasing the programming current to 15 mA might reduce the average antifuse resistance to 100 Ω. Antifuses separate interconnect

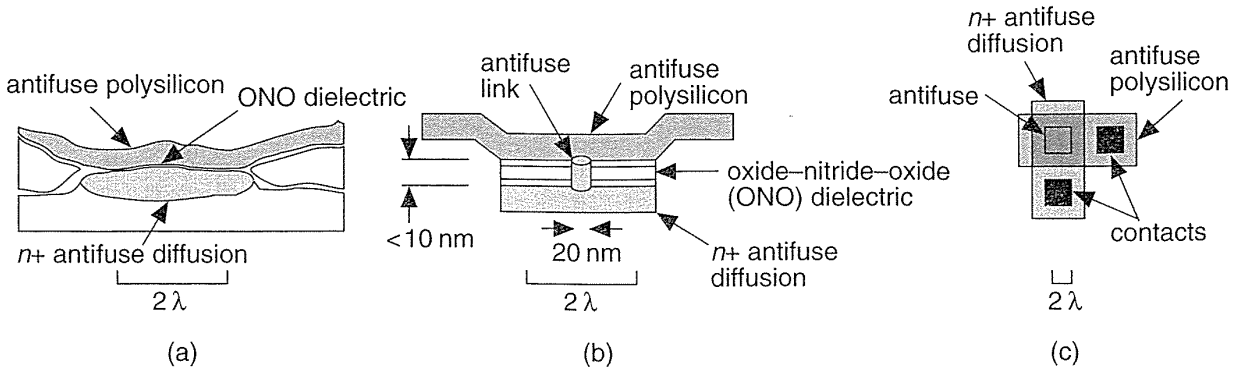


FIGURE 4.1 Actel antifuse. (a) A cross section. (b) A simplified drawing. The ONO (oxide–nitride–oxide) dielectric is less than 10 nm thick, so this diagram is not to scale. (c) From above, an antifuse is approximately the same size as a contact.

wires on the FPGA chip and the programmer blows an antifuse to make a permanent connection. Once an antifuse is programmed, the process cannot be reversed. This is an OTP technology (and radiation hard). An Actel 1010, for example, contains 112,000 antifuses (see Table 4.1), but we typically only need to program about 2 percent of the fuses on an Actel chip.

TABLE 4.1 Number of antifuses on Actel FPGAs.

Device	Antifuses
A1010	112,000
A1020	186,000
A1225	250,000
A1240	400,000
A1280	750,000

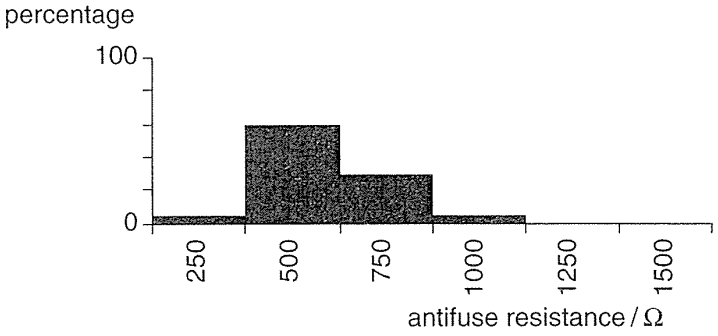


FIGURE 4.2 The resistance of blown Actel antifuses. The average antifuse resistance depends on the programming current. The resistance values shown here are typical for a programming current of 5 mA.

To design and program an Actel FPGA, designers iterate between design entry and simulation. When they are satisfied the design is correct they plug the chip into a socket on a special programming box, called an **Activator**, that generates

the programming voltage. A PC downloads the configuration file to the Activator instructing it to blow the necessary antifuses on the chip. When the chip is programmed it may be removed from the Activator without harming the configuration data and the chip assembled into a system. One disadvantage of this procedure is that modern packages with hundreds of thin metal leads are susceptible to damage when they are inserted and removed from sockets. The advantage of other programming technologies is that chips may be programmed after they have been assembled on a printed-circuit board—a feature known as **in-system programming (ISP)**.

The Actel antifuse technology uses a modified CMOS process. A double-metal, single-poly CMOS process typically uses about 12 masks—the Actel process requires an additional three masks. The *n*-type antifuse diffusion and antifuse polysilicon require an extra two masks and a 40 nm (thicker than normal) gate oxide (for the high-voltage transistors that handle 18 V to program the antifuses) uses one more masking step. Actel and Data General performed the initial experiments to develop the PLICE technology and Actel has licensed the technology to Texas Instruments (TI).

The programming time for an ACT 1 device is 5 to 10 minutes. Improvements in programming make the programming time for the ACT 2 and ACT 3 devices about the same as the ACT 1. A 5-day work week, with 8-hour days, contains about 2400 minutes. This is enough time to program 240 to 480 Actel parts per week with 100 percent efficiency and no hardware down time. A production schedule of more than 1000 parts per month requires multiple or gang programmers.

4.1.1 Metal–Metal Antifuse

Figure 4.3 shows a QuickLogic **metal–metal antifuse (ViaLink™)**. The link is an alloy of tungsten, titanium, and silicon with a bulk resistance of about 500 μΩcm.

There are two advantages of a metal–metal antifuse over a poly–diffusion antifuse. The first is that connections to a metal–metal antifuse are direct to metal—the wiring layers. Connections from a poly–diffusion antifuse to the wiring layers require extra space and create additional parasitic capacitance. The second advantage is that the direct connection to the low-resistance metal layers makes it easier to use larger programming currents to reduce the antifuse resistance. For example, the antifuse resistance $R \approx 0.8/I$, with the programming current I in mA and R in Ω, for the QuickLogic antifuse. Figure 4.4 shows that the average QuickLogic metal–metal antifuse resistance is approximately 80 Ω (with a standard deviation of about 10 Ω) using a programming current of 15 mA as opposed to an average antifuse resistance of 500 Ω (with a programming current of 5 mA) for a poly–diffusion antifuse.

The size of an antifuse is limited by the resolution of the lithography equipment used to makes ICs. The Actel antifuse connects diffusion and polysilicon, and both these materials are too resistive for use as signal interconnects. To connect the antifuse to the metal layers requires contacts that take up more space than the antifuse itself, reducing the advantage of the small antifuse size. However, the antifuse is so

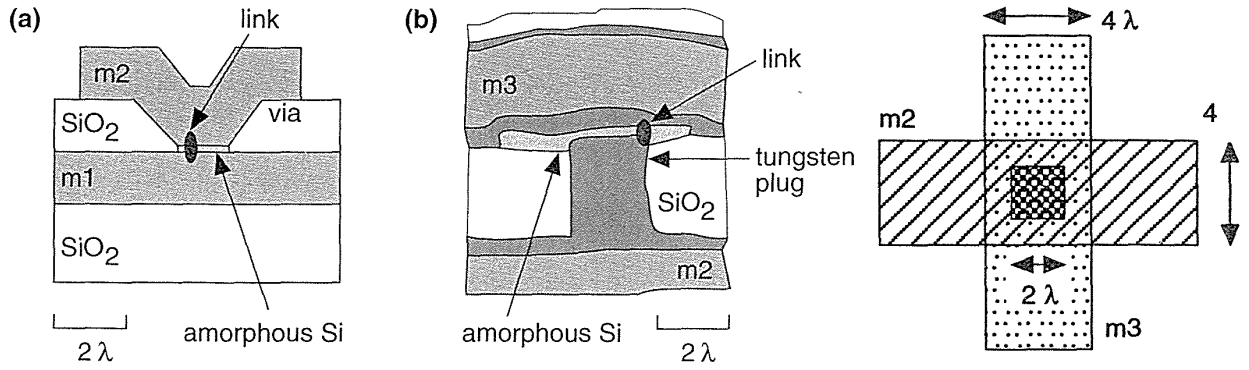


FIGURE 4.3 Metal–metal antifuse. (a) An idealized (but to scale) cross section of a QuickLogic metal–metal antifuse in a two-level metal process. (b) A metal–metal antifuse in a three-level metal process that uses contact plugs. The conductive link usually forms at the corner of the via where the electric field is highest during programming.

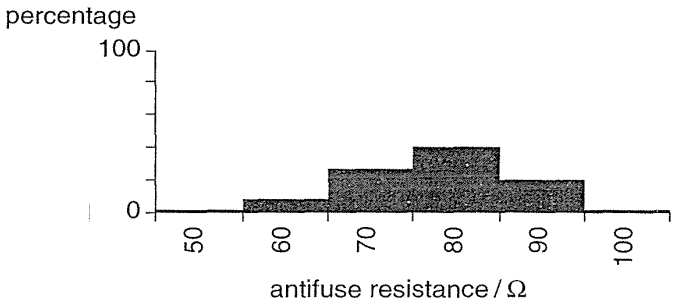


FIGURE 4.4 Resistance values for the QuickLogic metal–metal antifuse. A higher programming current (about 15 mA), made possible partly by the direct connections to metal, has reduced the antifuse resistance from the poly–diffusion antifuse resistance values shown in Figure 4.2.

small that it is normally the contact and metal spacing design rules that limit how closely the antifuses may be packed rather than the size of the antifuse itself.

An antifuse is resistive and the addition of contacts adds parasitic capacitance. The intrinsic parasitic capacitance of an antifuse is small (approximately 1–2 fF in a 1 μm CMOS process), but to this we must add the extrinsic parasitic capacitance that includes the capacitance of the diffusion and poly electrodes (in a poly–diffusion antifuse) and connecting metal wires (approximately 10 fF). These unwanted parasitic elements can add considerable *RC* interconnect delay if the number of antifuses connected in series is not kept to an absolute minimum. Clever routing techniques are therefore crucial to antifuse-based FPGAs.

The long-term reliability of antifuses is an important issue since there is a tendency for the antifuse properties to change over time. There have been some problems in this area, but as a result we now know an enormous amount about this

failure mechanism. There are many failure mechanisms in ICs—electromigration is a classic example—and engineers have learned to deal with these problems. Engineers design the circuits to keep the failure rate below acceptable limits and systems designers accept the statistics. All the FPGA vendors that use antifuse technology have extensive information on long-term reliability in their data books.

4.2 Static RAM

An example of **static RAM (SRAM)** programming technology is shown in Figure 4.5. This Xilinx SRAM configuration cell is constructed from two cross-coupled inverters and uses a standard CMOS process. The configuration cell drives the gates of other transistors on the chip—either turning pass transistors or transmission gates *on* to make a connection or *off* to break a connection.

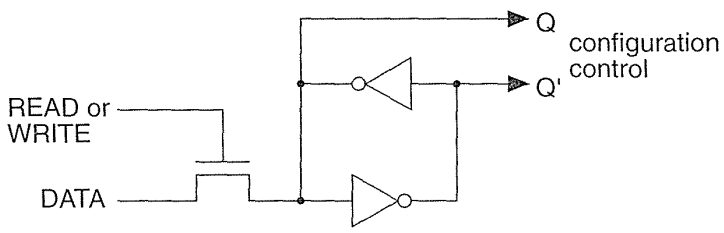


FIGURE 4.5 The Xilinx SRAM (static RAM) configuration cell. The outputs of the cross-coupled inverter (configuration control) are connected to the gates of pass transistors or transmission gates. The cell is programmed using the WRITE and DATA lines.

The advantages of SRAM programming technology are that designers can reuse chips during prototyping and a system can be manufactured using ISP. This programming technology is also useful for upgrades—a customer can be sent a new configuration file to reprogram a chip, not a new chip. Designers can also update or change a system on the fly in **reconfigurable hardware**.

The disadvantage of using SRAM programming technology is that you need to keep power supplied to the programmable ASIC (at a low level) for the volatile SRAM to retain the connection information. Alternatively you can load the configuration data from a permanently programmed memory (typically a **programmable read-only memory** or **PROM**) every time you turn the system on. The total size of an SRAM configuration cell plus the transistor switch that the SRAM cell drives is also larger than the programming devices used in the antifuse technologies.

4.3 EPROM and EEPROM Technology

Altera MAX 5000 EPLDs and Xilinx EPLDs both use UV-erasable **electrically programmable read-only memory (EPROM)** cells as their programming technology. Altera's EPROM cell is shown in Figure 4.6. The EPROM cell is almost as

small as an antifuse. An EPROM transistor looks like a normal MOS transistor except it has a second, floating, gate (gate1 in Figure 4.6). Applying a programming voltage V_{PP} (usually greater than 12 V) to the drain of the *n*-channel EPROM transistor programs the EPROM cell. A high electric field causes electrons flowing toward the drain to move so fast they "jump" across the insulating gate oxide where they are trapped on the bottom, floating, gate. We say these energetic electrons are *hot* and the effect is known as **hot-electron injection** or **avalanche injection**. EPROM technology is sometimes called **floating-gate avalanche MOS (FAMOS)**.

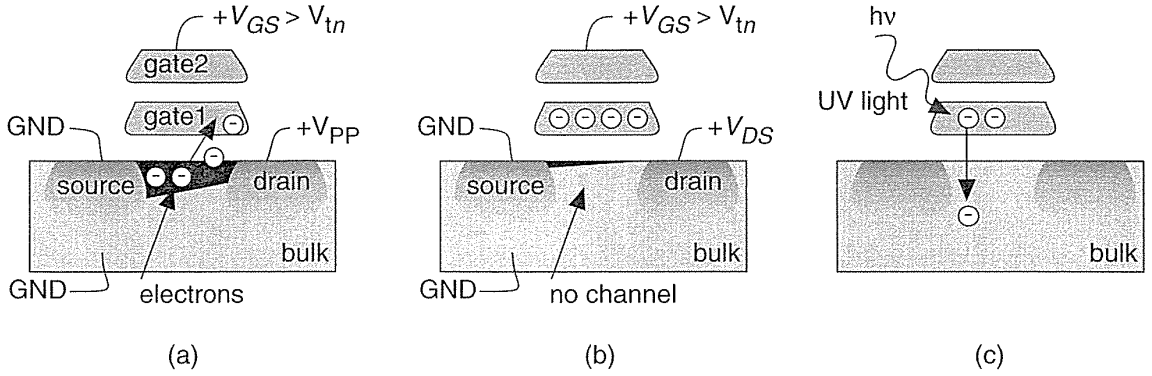


FIGURE 4.6 An EPROM transistor. (a) With a high (> 12 V) programming voltage, V_{PP}, applied to the drain, electrons gain enough energy to "jump" onto the floating gate (gate1). (b) Electrons stuck on gate1 raise the threshold voltage so that the transistor is always off for normal operating voltages. (c) Ultraviolet light provides enough energy for the electrons stuck on gate1 to "jump" back to the bulk, allowing the transistor to operate normally.

Electrons trapped on the floating gate raise the threshold voltage of the *n*-channel EPROM transistor (Figure 4.6b). Once programmed, an *n*-channel EPROM device remains *off* even with V_{DD} applied to the top gate. An unprogrammed *n*-channel device will turn *on* as normal with a top-gate voltage of V_{DD}. The programming voltage is applied either from a special programming box or by using on-chip charge pumps. Exposure to an ultraviolet (UV) lamp will erase the EPROM cell (Figure 4.6c). An absorbed light quantum gives an electron enough energy to jump from the floating gate. To erase a part we place it under a UV lamp (Xilinx specifies one hour within 1 inch of a 12,000 μWcm^{-2} source for its EPLDs). The manufacturer provides a software program that checks to see if a part is erased. You can buy an EPLD part in a windowed package for development, erase it, and use it again, or buy it in a nonwindowed package and program (or burn) the part once only for production. The packages get hot while they are being erased, so that windowed option is available with only ceramic packages, which are more expensive than plastic packages.

Programming an EEPROM transistor is similar to programming an UV-erasable EPROM transistor, but the erase mechanism is different. In an EEPROM transistor an electric field is also used to remove electrons from the floating gate of a programmed transistor. This is faster than using a UV lamp and the chip does not have to be removed from the system. If the part contains circuits to generate both program and erase voltages, it may use ISP.

4.4 Practical Issues

System companies often select an ASIC technology first, which narrows the choice of software design tools. The software then influences the choice of computer. Most **computer-aided engineering** (**CAE**) software for FPGA design uses some type of security. For workstations this usually means floating licenses (any of n users on a network can use the tools) or node-locked licenses (only n particular computers can use the tools) using the hostid (or host I.D., a serial number unique to each computer) in the boot EPROM (a chip containing start-up instructions). For PCs this is a hardware key, similar to the Viewlogic key illustrated in Figure 4.7. Some keys use the serial port (requiring extra cables and adapters); most now use the parallel port. There are often conflicts between keys and other hardware/software. For example, for a while some security keys did not work with the serial-port driver on Intel motherboards—users had to buy another serial-port I/O card.

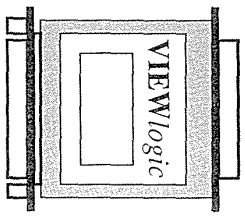


FIGURE 4.7 CAE companies use hardware security keys that fit at the back of a PC (this one is shown at about one-half the real size). Each piece of software requires a separate key, so that a typical design system may have a half dozen or more keys daisy-chained on one socket. This presents both mechanical and software conflict problems. Software will not run without a key, so it is easily possible to have $60,000 worth of keys attached to a single PC.

Most FPGA vendors offer software on multiple platforms. The performance difference between workstations and PCs is becoming blurred, but the time taken for the place-and-route step for Actel and Xilinx designs seems to remain constant—typically taking tens of minutes to over an hour for a large design—bounded by designers' tolerances.

A great deal of time during FPGA design is spent in schematic entry, editing files, and documentation. This often requires moving between programs and this is difficult on IBM-compatible PC platforms. Currently most large CAD and CAE programs completely take over the PC; for example you cannot always run third-party design entry and the FPGA vendor design systems simultaneously.

There are many other factors to be considered in choosing hardware:

- Software packages are normally less expensive on a PC.
- Peripherals are less expensive and easier to configure on a PC.
- Maintenance contracts are usually necessary and expensive for workstations.
- There is a much larger network of users to provide support for PC users.
- It is easier to upgrade a PC than a workstation.

4.4.1 FPGAs in Use

I once placed an order for a small number of FPGAs for prototyping and received a sales receipt with a scheduled shipping date three months away. Apparently, two customers had recently disrupted the vendor's product planning by placing large orders. Companies buying parts from suppliers often keep an **inventory** to cover emergencies such as a defective lot or manufacturing problems. For example, assume that a company keeps two months of inventory to ensure that it has parts in case of unforeseen problems. This **risk inventory** or safety supply, at a sales volume of 2000 parts per month, is 4000 parts, which, at an ASIC price of $5 per part, costs the company $20,000. FPGAs are normally sold through distributors, and, instead of keeping a risk inventory, a company can order parts as it needs them using a **just-in-time** (**JIT**) inventory system. This means that the distributors rather than the customer carry inventory (though the distributors wish to minimize inventory as well). The downside is that other customers may change their demands, causing unpredictable supply difficulties.

There are no standards for FPGAs equivalent to those in the TTL and PLD worlds; there are no standard pin assignments for VDD or GND, and each FPGA vendor uses different power and signal I/O pin arrangements. Most FPGA packages are intended for surface-mount **printed-circuit boards** (**PCBs**). However, surface mounting requires more expensive PCB test equipment and vapor soldering rather than bed-of-nails testers and surface-wave soldering. An alternative is to use socketed parts. Several FPGA vendors publish socket-reliability tests in their data books.

Using sockets raises its own set of problems. First, it is difficult to find wire-wrap sockets for surface-mount parts. Second, sockets may change the pin configuration. For example, when you use an FPGA in a PLCC package and plug it into a socket that has a PGA footprint, the resulting arrangement of pins is different from the same FPGA in a PGA package. This means you cannot use the same board layout for a prototype PCB (which uses the socketed PLCC part) as for the production PCB (which uses the PGA part). The same problem occurs when you use through-hole mounted parts for prototyping and surface-mount parts for production. To deal with this you can add a small piece to your prototype board that you use as a converter. This can be sawn off on the production boards—saving a board iteration.

Pin assignment can also cause a problem if you plan to convert an FPGA design to an MGA or CBIC. In most cases it is desirable to keep the same pin assignment as the FPGA (this is known as **pin locking** or **I/O locking**), so that the same PCB

can be used in production for both types of devices. There are often restrictions for custom gate arrays on the number and location of power pads and package pins. Systems designers must consider these problems before designing the FPGA and PCB.

4.5 Specifications

All FPGA manufactures are continually improving their products to increase performance and reduce price. Often this means changing the design of an FPGA or moving a part from one process generation to the next without changing the part number (and often without changing the specifications).

FPGA companies usually explain their part history in their data books.[1] The following history of Actel FPGA ACT 1 part numbers illustrates changes typical throughout the IC industry as products develop and mature:

- The Actel ACT 1 A1010/A1020 used a 2 μm process.
- The Actel A1010A/A1020A used a 1.2 μm process.
- The Actel A1020B was a die revision (including a shrink to a 1.0 μm process). At this time the A1020, A1020A, and A1020B all had different speeds.
- Actel graded parts into three speed bins as they phased in new processes, dropping the distinction between the different die suffixes.
- At the same time as the transition to die rev. 'B', Actel began specifying timing at worst-case commercial conditions rather than at typical conditions.

From this history we can see that it is often possible to have parts from the same family that use different circuit designs, processes, and die sizes, are manufactured in different locations, and operate at very different speeds. FPGA companies ensure that their products always meet the current published worst-case specifications, but there is no guarantee that the average performance follows the typical specifications, and there are usually no best-case specifications.

There are also situations in which two parts with identical part numbers can have different performance—when different ASIC foundries produce the same parts. Since FPGA companies are fabless, second sourcing is very common. For example, TI began making the TPC1010A/1020A to be equivalent to the original Actel ACT 1 parts produced elsewhere. The TI timing information for the TPC1010A/1020A was the same as the 2 μm Actel specifications, but TI used a faster 1.2 μm process. This meant that "equivalent" parts with the same part numbers were *much* faster than a designer expected. Often this type of information can only be obtained by large customers in the form of a **qualification kit** from FPGA vendors.

[1] See, for example, p.1-8 of the Xilinx 1994 data book.

A similar situation arises when the FPGA manufacturer adjusts its product mix by selling fast parts under a slower part number in a procedure known as **down-binning**. This is not a problem for synchronous designs that always work when parts are faster than expected, but is another reason to avoid asynchronous designs that may not always work when parts are much faster than expected.

4.6 PREP Benchmarks

Which type of FPGA is best? This is an impossible question to answer. The **Programmable Electronics Performance Company** (**PREP**) is a nonprofit organization that organized a series of benchmarks for programmable ASICs. The nine PREP benchmark circuits in the version 1.3 suite are:

1. An 8-bit datapath consisting of 4:1 MUX, register, and shift-register
2. An 8-bit timer–counter consisting of two registers, a 4:1 MUX, a counter and a comparator
3. A small state machine (8 states, 8 inputs, and 8 outputs)
4. A larger state machine (16 states, 8 inputs, and 8 outputs)
5. An ALU consisting of a 4×4 multiplier, an 8-bit adder, and an 8-bit register
6. A 16-bit accumulator
7. A 16-bit counter with synchronous load and enable
8. A 16-bit prescaled counter with load and enable
9. A 16-bit address decoder

The data for these benchmarks is archived at `http://www.prep.org`. PREP's online information includes Verilog and VHDL source code and test benches (provided by Synplicity) as well as additional synthesis benchmarks including a bit-slice processor, multiplier, and R4000 MIPS RISC microprocessor.

One problem with the FPGA benchmark suite is that the examples are small, allowing FPGA vendors to replicate multiple instances of the same circuit on an FPGA. This does not reflect the way an FPGA is used in practice. Another problem is that the FPGA vendors badly misused the results. PREP made the data available in a spreadsheet form and thus inadvertently challenged the marketing department of each FPGA vendor to find a way that company could claim to win the benchmarks (usually by manipulating the data using a complicated weighting scheme). The PREP benchmarks do demonstrate the large variation in performance between different FPGA architectures that results from differences in the type and mix of logic. This shows that designers should be careful in evaluating others' results and performing their own experiments.

4.7 FPGA Economics

FPGA vendors offer a wide variety of packaging, speed, and qualification (military, industrial, or commercial) options in each family. For example, there are several hundred possible part combinations for the Xilinx LCA series. Figure 4.8 shows the Xilinx part-naming convention, which is similar to that used by other FPGA vendors.

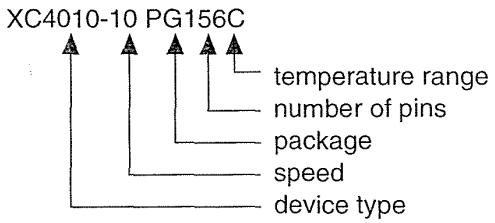


FIGURE 4.8 Xilinx part-naming convention.

Table 4.2 shows the various codes used by manufacturers in their FPGA part numbers. Not all possible part combinations are available, not all packaging combinations are available, and not all I/O options are available in all packages. For example, it is quite common for an FPGA vendor to offer a chip that has more I/O cells than pins on the package. This allows the use of cheaper plastic packages without having to produce separate chip designs for each different package. Thus a customer can buy an Actel A1020 that has 69 I/O cells in an inexpensive 44-pin PLCC package but uses only 34 pins for I/O—the other 10 (=44 – 34) pins are required for programming and power: three for GND, four for VDD, one for MODE (a pin that controls four other multifunction pins), and one for VPP (the programming voltage). A designer who needs all 69 I/Os can buy the A1020 in a bigger package. Tables in the FPGA manufacturers' data books show the availability, and these matrices change constantly.

4.7.1 FPGA Pricing

Asking "How much do FPGAs cost?" is rather like asking "How much does a car cost?" Prices of cars are published, but pricing schemes used by semiconductor manufactures are closely guarded secrets. Many FPGA companies use a pricing strategy based on a cost model that uses a series of multipliers or adders for each part option to calculate the suggested price for their distributors. Although the FPGA companies will not divulge their methods, it is possible to reverse engineer these factors to create a pricing matrix.

Many FPGA vendors sell parts through distributors. This can introduce some problems for the designer. For example, in 1992 the Xilinx XC3000 series offered the following part options:

- Five different size parts: XC30{20, 30, 42, 64, 90}
- Three different speed grades or bins: {50, 70, 100}

TABLE 4.2 Programmable ASIC part codes.

Item	Code	Description	Code	Description
Manufacturer's code	A	Actel	ATT	AT&T (Lucent)
	XC	Xilinx	isp	Lattice Logic
	EPM	Altera MAX	M5	AMD MACH 5 is on the device
	EPF	Altera FLEX	QL	QuickLogic
	CY7C	Cypress		
Package type	PL or PC	plastic J-leaded chip carrier, PLCC	VQ	very thin quad flatpack, VQFP
	PQ	plastic quad flatpack, PQFP	TQ	thin plastic flatpack, TQFP
	CQ or CB	ceramic quad flatpack, CQFP	PP	plastic pin-grid array, PPGA
	PG	ceramic pin-grid array, PGA	WB, PB	ball-grid array, BGA
Application	C	commercial	B	MIL-STD-883
	I	industrial	E	extended
	M	military		

TABLE 4.3 1992 base Actel FPGA prices.

Actel part	1H92 base price
A1010A-PL44C	$23.25
A1020A-PL44C	$43.30
A1225-PQ100C	$105.00
A1240-PQ144C	$175.00
A1280-PQ160C	$305.00

TABLE 4.4 1992 base Xilinx XC3000 FPGA prices.

Xilinx part	1H92 base price
XC3020-50PC68C	$26.00
XC3030-50PC44C	$34.20
XC3042-50PC84C	$52.00
XC3064-50PC84C	$87.00
XC3090-50PC84C	$133.30

- Ten different packages: {PC68, PC84, PG84, PQ100, CQ100, PP132, PG132, CQ184, PP175, PG175}
- Four application ranges or qualification types: {C, I, M, B}

where { } means "Choose one."

This range of options gave a total of 600 possible XC3000 products, of which 127 were actually available from Xilinx, each with a different part code. If a designer is uncertain as to exact size, speed, or package required, then they might easily need price information on several dozen different part numbers. Distributors know the price information—it is given to each distributor by the FPGA vendors. Sometimes the distributors are reluctant to give pricing information out—for the

TABLE 4.5 Actel price adjustment factors.

Purchase quantity, all types

(1–9)	(10–99)	(100–999)
100 %	96 %	84 %

Purchase time, in (100–999) quantity

1H92	2H92	93
100 %	80–95 %	60–80 %

Qualification type, same package

Commercial	Industrial	Military	883-B
100 %	120 %	150 %	230–300 %

Speed bin[1]

ACT 1-Std	ACT 1-1	ACT 1-2	ACT 2-Std	ACT 2-1
100 %	115 %	140 %	100 %	120 %

Package type

A1010:	PL44, 64, 84	PQ100	PG84		
	100 %	125 %	400 %		
A1020:	PL44, 64, 84	PQ100	JQ44, 68, 84	PG84	CQ84
	100 %	125 %	270 %	275 %	400 %
A1225:	PQ100	PG100			
	100 %	175 %			
A1240:	PQ144	PG132			
	100 %	140 %			
A1280:	PQ160	PG176	CQ172		
	100 %	145 %	160 %		

[1]Actel speed bins are: Std = standard speed grade; 1 = medium speed grade; 2 = fastest speed grade.

same reason car salespeople do not always like to advertise the pricing scheme for cars. However, pricing of the components of a microelectronics system is a vital fac-

tor in making decisions such as whether to use FPGAs or some alternative technology. Designers would like to know how FPGAs are priced and how prices may change.

4.7.2 Pricing Examples

Table 4.3 shows the prices of the least-expensive version of the Actel ACT 1 and ACT 2 FPGA families, the **base prices**, in the first half of 1992 (1H92). Table 4.4 shows the 1H92 base prices for the Xilinx XC3000 FPGA family. Current FPGA prices are much lower. As an example, the least-expensive XC3000 part, the XC3020A-7PC68C, was $13.75 in 1996—nearly half the 1992 price.

Using historical prices helps prevent accusations of bias or distortion, but still realistically illustrates the pricing schemes that are used. We shall use these base prices to illustrate how to estimate the sticker price of an FPGA by adding options—as we might for a car. To estimate the price of any part, multiply the base prices by the **adjustment factors** (shown in Table 4.5 for the Actel parts).

The adjustment factors in Table 4.5 were calculated by taking averages across a matrix of prices. Not all combinations of product types are available (for example, there was no military version of an A1280-1 in 1H92). The dependence of price over time is especially variable. An example price calculation for an Actel part is shown in Table 4.6. Many FPGA vendors use similar pricing models.

TABLE 4.6 Example Actel part-price calculation using the base prices of Table 4.3 and the adjustment factors of Table 4.5.

Example: A1020A-2-PQ100I in (100–999) quantity, purchased 1H92.

Factor	Example	Value
Base price	A1020A	$43.30
Quantity	100–999	84 %
Time	1H92	100 %
Qualification type	Industrial (I)	120 %
Speed bin[1]	2	140 %
Package	PQ100	125 %
Estimated price (1H92)		$76.38
Actual Actel price (1H92)		$75.60

[1]The speed bin is a manufacturer's code (usually a number) that follows the family part number and indicates the maximum operating speed of the device.

Some distributors now include FPGA prices and availability online (for example, Marshall at `http://marshall.com` for Xilinx parts) so that is possible to complete an up-to-date analysis at any time. Most distributors carry only one FPGA vendor; not all of the distributors publish prices; and not all FPGA vendors sell through distributors. Currently Hamilton-Avnet, at `http://www.hh.avnet.com`, carries Xilinx; and Wyle, at `http://www.wyle.com`, carries Actel and Altera.

4.8 Summary

In this chapter we have covered FPGA programming technologies including antifuse, SRAM, and EPROM technologies; the programming technology is linked to all the other aspects of a programmable ASIC. Table 4.7 summarizes the programming technologies and the fabrication processes used by programmable ASIC vendors.

TABLE 4.7 Programmable ASIC technologies.

	Actel	Xilinx LCA[1]	Altera EPLD	Xilinx EPLD
Programming technology	Poly–diffusion antifuse, PLICE	Erasable SRAM ISP	UV-erasable EPROM (MAX 5k) EEPROM (MAX 7/9k)	UV-erasable EPROM
Size of programming element	Small but requires contacts to metal	Two inverters plus pass and switch devices. Largest.	One *n*-channel EPROM device. Medium.	One *n*-channel EPROM device. Medium.
Process	Special: CMOS plus three extra masks.	Standard CMOS	Standard EPROM and EEPROM	Standard EPROM
Programming method	Special hardware	PC card, PROM, or serial port	ISP (MAX 9k) or EPROM programmer	EPROM programmer

	QuickLogic	Crosspoint	Atmel	Altera FLEX
Programming technology	Metal–metal antifuse, ViaLink	Metal–polysilicon antifuse	Erasable SRAM. ISP.	Erasable SRAM. ISP.
Size of programming element	Smallest	Small	Two inverters plus pass and switch devices. Largest.	Two inverters plus pass and switch devices. Largest.
Process	Special, CMOS plus ViaLink	Special, CMOS plus antifuse	Standard CMOS	Standard CMOS
Programming method	Special hardware	Special hardware	PC card, PROM, or serial port	PC card, PROM, or serial port

[1]Lucent (formerly AT&T) FPGAs have almost identical properties to the Xilinx LCA family.

All FPGAs have the following key elements:

- The programming technology
- The basic logic cells
- The I/O logic cells
- Programmable interconnect
- Software to design and program the FPGA

4.9 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

4.1 (Antifuse properties, 20 min.) In this problem we examine some of the physical and electrical features of the antifuse programming process.

a. If the programming current of an antifuse is 5 mA and the link diameter that is formed is 20 nm, what is the current density during programming?

b. If the average antifuse resistance is 500 Ω after programming is complete and the programming current is 5 mA, what is the voltage across the antifuse at completion of programming?

c. What power is dissipated in the antifuse link at the end of programming?

d. Suppose we wish to reduce the antifuse resistance from 500 Ω to 50 Ω. If the antifuse link is a tall, thin cylinder, what is the diameter of a 50 Ω antifuse?

e. Assume we need to keep the power dissipated per unit volume of the antifuse link the same at the end of the programming process for both 500 Ω and 50 Ω antifuses. What current density is required to program a 50 Ω antifuse?

f. With these assumptions what is the required programming current for a 50 Ω antifuse? Comment on your answer and the assumptions that you have made.

4.2 (Actel antifuse programming, 20 min.) In this problem we examine the time taken to program an antifuse-based FPGA.

a. We have stated that it takes about 5 to 10 minutes to program an Actel part. Given the number of antifuses on the smallest Actel part, and the number of antifuses that need to be blown on average, work out the equivalent time it takes to blow one antifuse. Does this seem reasonable?

b. Because of a failure process known as electromigration, the current density in a metal wire on a chip is limited to about 50 kAcm^{-2}. You can exceed this current for a short time as long as the time average does not exceed the limit. Suppose we want to use a minimum metal width to connect the programming transistors: Would these facts help explain your answer to part a?

c. What other factors might be involved in the process of blowing antifuses that may help explain your answer to part a?

4.3 (*Xilinx cell) Estimate the area components of a Xilinx cell as follows:

a. (30 min.) Assume the two inverters in the cross-coupled SRAM cell are minimum size (they are not, the *p*-channels—or *n*-channels—in one inverter need to be weak—long and narrow—but ignore this). Assume the read–write device is minimum size. Estimate the size of the SRAM cell including an allowance for wiring (state your assumptions clearly).

b. (15 min.) Assume a single *n*-channel pass transistor is connected to the SRAM cell and has an on-resistance of 500 Ω (equal to the average Actel ACT 1 antifuse resistance for comparison; the actual Xilinx pass transistors have closer to 1 kΩ on-resistance). Estimate the transistor size. Assume the gate voltage of the pass transistor is at 5 V, and the source and drain voltages are both at 0 V (the best case). *Hint:* Use the parameters from Section 3.1, "Transistors as Resistors."

c. (15 min.) Compare your total area estimates of the cell with other FPGA technologies. Explain why the assumptions you made may be too simple, and suggest ways to make more accurate estimates.

4.4 (FPGA vendors, 60 min.) Update the information shown in Table 4.7 using the online information provided by FPGA vendors.

4.5 (Prices) Adjustment factors, calculated from averages across the Xilinx price matrix, are shown in Table 4.8 (the adjustment factors for the Xilinx military and MIL-STD parts vary so wildly that it is not possible to use a simple model to predict these prices).

a. (5 min.) Estimate the price of a XC3042-70PG132I in 100+ quantity, purchased in 1H92.

b. (30 min.) Use the 1992 prices in Figure 4.9 to derive as much of the information shown in Table 4.8 as you can, explaining your methods.

3042	(1–24)	(25–99)	(100+)
50PC84C	$52.00	$47.30	$40.05
50PC84I	$67.30	$61.25	$51.80
70PC84C	$56.50	$51.40	$43.50
70PC84I	$73.30	$66.70	$56.45
100PC84C	$67.70	$61.60	$52.15
125PC84C	$114.00	$103.75	$87.80
50PP132C	$124.50	$113.30	$95.85
50PQ100C	$60.40		
50PG84C	$161.50		
50CQ100C	$194.50		
50PG132C	$191.20		

FIGURE 4.9 Xilinx XC3042 prices (1992). Problem 4.5 reconstructs part of Table 4.8 from this data.

TABLE 4.8 Xilinx price adjustment factors (1992) for Problem 4.5

Purchase quantity, all types

(1–24)	(25–99)	(100+)	(5000+)
100%	91%	77%	70%

Purchase time, in (100–999) quantity

1H92	+18 months
100%	60%

Qualification type, same package

Commercial	Industrial	Military	883-B
100%	130%	varies	varies

Speed bin

50	70	100	125
100%	110%	130%	220%

Package type

3020:	PC68	PC84	PQ100	PG84	CQ100	
	100%	106%	127%	340%	490%	
3030:	PC44	PC68	PC84	PQ100	PG84	
	100%	107%	113%	135%	330%	
3042:	PC84	PQ100	PP132	PG84	PG132	CQ100
	100%	175%	240%	310%	370%	375%
3064:	PC84	PQ160	PP132	PG132		
	100%	150%	190%	260%		
3090:	PC84	PQ160	PP175	PG175	CQ164	
	100%	130%	150%	230%	240%	

c. (Hours) Construct a table (using the format of Table 4.8) for a current FPGA family. You may have to be creative in capturing the HTML and filtering it into a spreadsheet. *Hint:* In Microsoft Word 5.0 you can select columns of text by holding down the Option key.

Answer: (a) $211.85 (the actual Xilinx price was $210.20).

4.6 (PREP benchmarks, 60 min.) Download the PREP 1.3 benchmark results as spreadsheets from `http://www.prep.org`. Split the participating companies among groups and challenge each group to produce an averaging or analysis scheme that shows the group's assigned company as a "winner." For hints on this problem, consult advertisements in past issues of *EE Times*.

4.7 (FPGA patents) Patents are a good place to find information on FPGAs.

a. Find U.S. Patent 5,440,245, Galbraith et al. "Logic module with configurable combinational and sequential blocks." Find and explain a method to paste the figures into a report.

b. Conduct a patent search on FPGAs. Good places to start are the **U.S. Patent and Trademark Office (PTO)** at `http://www.uspto.gov` and the IBM patent resource at `http://patent.womplex.ibm.com`. Until 1996 the full text of recent U.S. patents was available at `http://www.town.hall.org/patent`; this is still a good site to visit for references to other locations. Table 4.9 lists the patents awarded to the major FPGA companies up until 1996 (in the case of Actel and Altera the list includes only patents issued after 1990, corresponding roughly to patent numbers greater than number 5,000,000, which was issued in March 1990).

4.8 (**Maskworks, days) If you really want to find out about FPGA technology you tear chips apart. There is another way. Most U.S. companies register their chips as a type of copyright called a **Maskwork**. You will often see a little circle containing an "M" on a chip in the same way that a copyright sign is a circle surrounding the letter "C". Companies that require a Maskwork are required to deposit plots and samples of the chips with a branch of the Library of Congress. These plots are open for public inspection in Washington, D.C. It is perfectly legal to use this information. You have to sign a visitors' book, and most of the names in the book are Japanese. Research Maskworks and write a summary of its implications, the protection it provides, and (if you can find them) the rules for the materials that must be deposited with the authorities.

TABLE 4.9 FPGA Patents (U.S.).

QuickLogic	**Xilinx**	5,329,181	4,713,557	5,308,795	5,008,855	5,280,203
5,416,367	5,436,575	5,329,174	4,706,216	5,304,871		5,274,581
5,397,939	5,432,719	5,329,181	4,695,740	5,299,150	**Altera**	5,272,368
5,396,127	5,430,687	5,321,704	4,642,487	5,286,992	5,477,474	5,268,598
5,362,676	5,430,390	5,319,254		5,272,388	5,473,266	5,260,611
5,319,238	5,426,379	5,319,252	**Actel**	5,272,101	5,463,328	5,260,610
5,302,546	5,426,378	5,302,866	5,479,113	5,266,829	5,444,394	5,258,668
5,220,213	5,422,833	5,295,090	5,477,165	5,254,886	5,438,295	5,247,478
5,196,724	5,414,377	5,291,079	5,469,396	5,223,792	5,436,575	5,247,477
	5,410,194	5,245,277	5,464,790	5,208,530	5,436,574	5,243,233
Intel	5,410,189	5,224,056	5,457,644	5,198,705	5,434,514	5,241,224
4,543,594[1]	5,399,925	5,166,858	5,451,887	5,194,759	5,432,467	5,237,219
	5,399,924	5,155,432	5,449,947	5,191,241	5,414,312	5,220,533
Crosspoint	5,394,104	5,148,390	5,448,185	5,187,393	5,399,922	5,220,214
5,440,453	5,386,154	5,068,603	5,440,245	5,181,096	5,384,499	5,200,920
5,394,103	5,367,207	5,047,710	5,432,441	5,172,014	5,376,844	5,166,604
5,384,481	5,365,125	5,028,821	5,414,364	5,171,715	5,371,422	5,162,680
5,322,812	5,362,999	5,023,606	5,412,244	5,163,180	5,369,314	5,144,167
5,313,119	5,361,229	5,012,135	5,411,917	5,134,457	5,359,243	5,138,576
5,233,217	5,360,747	4,967,107	5,404,029	5,132,571	5,359,242	5,128,565
5,221,865	5,359,536	4,940,909	5,391,942	5,130,777	5,353,248	5,121,006
	5,349,691	4,902,910	5,387,812	5,126,282	5,352,940	5,111,423
Concurrent	5,349,250	4,870,302	5,373,169	5,111,262	5,350,954	5,097,208
5,218,240	5,349,249	4,855,669	5,371,414	5,107,146	5,349,255	5,091,661
5,144,166	5,349,248	4,855,619	5,369,054	5,095,228	5,341,308	5,066,873
5,089,973	5,343,406	4,847,612	5,367,208	5,087,958	5,341,048	5,045,772
	5,337,255	4,835,418	5,365,165	5,083,083	5,341,044	
Plus Logic	5,349,248	4,821,233	5,341,092	5,073,729	5,329,487	
5,028,821	5,343,406	4,820,937	5,341,043	5,070,384	5,317,210	
5,023,606	5,337,255	4,783,607	5,341,030	5,057,451	5,315,172	
5,012,135	5,332,929	4,758,985	5,317,698	5,055,718	5,301,416	
4,967,107	5,331,226	4,750,155	5,316,971	5,017,813	5,294,975	
4,940,909	5,331,220	4,746,822	5,309,091	5,015,885	5,285,153	

[1]Mohsen's patent on the antifuse structure.

4.10 Bibliography

Books by Ukeiley [1993], Chan [1994], and Trimberger [1994] are dedicated to FPGAs and their uses. The *International Workshop on Field-Programmable Logic and Applications* describes the latest developments and applications of FPGAs [Grünbacher and Hartenstein, 1992; Hartenstein and Servit, 1994; Moore and Luk, 1995; Hartenstein and Glesner, 1996]. Many of the FPGA vendors have Web sites that include white papers and technical documentation. The annual *IEEE International Electron Devices Meeting* (IEDM, ISSN 0163-1918, TK 7801.I53) is a forum for presenting new device and IC technology including new FPGA programming technologies. The *IEEE Transaction on Electron Devices* (ISSN 0018-9383) is the archival source for developments in device technology.

There is a large U.S. patent literature on FPGAs (see Table 4.9). Sometimes the FPGA vendors hide the basic low-level structures from the user to simplify their description or to prevent the competition from understanding their secrets. Patents have to explain the details of operation (otherwise they will not be awarded or cannot be enforced), so sometimes it can be useful to at least know where to look. One place to start is the front or back of the data book, which often contains a list of the manufacturer's patents.

4.11 References

Chan, P. K., and S. Mourad. 1994. *Digital Design Using Field Programmable Gate Arrays.* Englewood Cliffs, NJ: Prentice-Hall, 233 p. ISBN 0-13-319021-8. TK7888.4.C43.

Grünbacher, H., and R. W. Hartenstein. (Eds.). 1993. *International Workshop on Field-Programmable Logic and Applications* (2nd: 1992: Vienna). Berlin; New York: Springer-Verlag. ISBN 0387570918. TK7895.G36.I48.

Hartenstein, R. W., and M. Glesner (Eds.). 1996. *International Workshop on Field-Programmable Logic and Applications* (6th: 1996: Darmstadt). Berlin; New York: Springer-Verlag. ISBN 3540617302. TK7868.L6.I56.

Hartenstein, R. W., and M. Z. Servit. (Eds.). 1994. *International Workshop on Field-Programmable Logic and Applications* (4th: 1994: Prague). Berlin; New York: Springer-Verlag. ISBN 0387584196. TK7868.L6.I56.

Moore, W., and W. Luk. (Eds.). 1995. *International Workshop on Field-Programmable Logic and Applications* (5th: 1995: Oxford). Berlin; New York: Springer-Verlag. ISBN 3540602941. TK7895.G36.I48.

Trimberger, S. M. (Ed.). 1994. *Field-Programmable Gate Array Technology.* Boston: Kluwer Academic Publishers. ISBN 0-7923-9419-4. TK7895.G36.F54.

Ukeiley, R. L. 1993. *Field Programmable Gate Arrays (FPGAs): The 3000 Series.* Englewood Cliffs, NJ: Prentice-Hall, 173 p. ISBN 0-13-319468-X. TK7895.G36.U44.

PROGRAMMABLE ASIC LOGIC CELLS

5

All programmable ASICs or FPGAs contain a **basic logic cell** replicated in a regular array across the chip (analogous to a base cell in an MGA). There are the following three different types of basic logic cells: (1) multiplexer based, (2) look-up table based, and (3) programmable array logic. The choice among these depends on the programming technology. We shall see examples of each in this chapter.

5.1 Actel ACT

The basic logic cells in the Actel ACT family of FPGAs are called **Logic Modules**. The ACT 1 family uses just one type of Logic Module and the ACT 2 and ACT 3 FPGA families both use two different types of Logic Module.

5.1.1 ACT 1 Logic Module

The functional behavior of the Actel ACT 1 Logic Module is shown in Figure 5.1(a). Figure 5.1(b) represents a possible circuit-level implementation. We can build a logic function using an Actel Logic Module by connecting logic signals to some or all of the Logic Module inputs, and by connecting any remaining Logic Module inputs to VDD or GND. As an example, Figure 5.1(c) shows the connections to implement the function $F = A \cdot B + B' \cdot C + D$. How did we know what connections to make? To understand how the Actel Logic Module works, we take a detour via multiplexer logic and some theory.

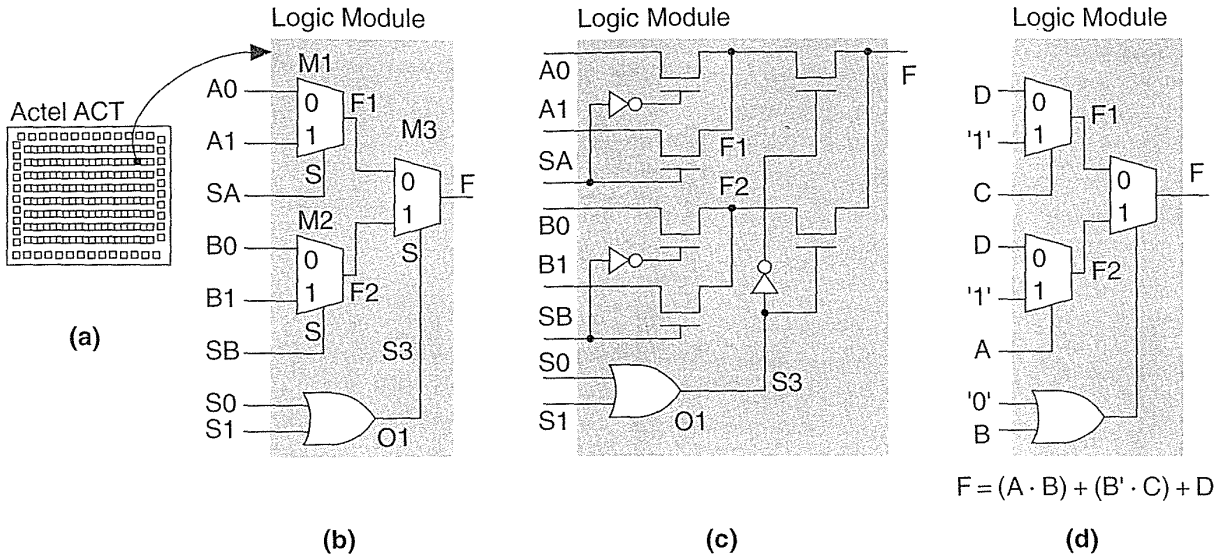


FIGURE 5.1 The Actel ACT architecture. (a) Organization of the basic logic cells. (b) The ACT 1 Logic Module. (c) An implementation using pass transistors (without any buffering). (d) An example logic macro. (*Source*: Actel.)

5.1.2 Shannon's Expansion Theorem

In logic design we often have to deal with functions of many variables. We need a method to break down these large functions into smaller pieces. Using the **Shannon expansion theorem,** we can **expand** a Boolean logic function F in terms of (or with respect to) a Boolean variable A,

$$F = A \cdot F(A = '1') + A' \cdot F(A = '0'), \tag{5.1}$$

where F (A = 1) represents the function F evaluated with A set equal to '1'.

For example, we can expand the following function F with respect to (I shall use the abbreviation *wrt*) A,

$$\begin{aligned} F &= A' \cdot B + A \cdot B \cdot C' + A' \cdot B' \cdot C \\ &= A \cdot (B \cdot C') + A' \cdot (B + B' \cdot C). \end{aligned} \tag{5.2}$$

We have split F into two smaller functions. We call $F(A = '1') = B \cdot C'$ the **cofactor** of F *wrt* A in Eq. 5.2. I shall sometimes write the cofactor of F *wrt* A as F_A (the cofactor of F *wrt* A' is $F_{A'}$). We may expand a function *wrt* any of its variables. For example, if we expand F *wrt* B instead of A,

$$\begin{aligned} F &= A' \cdot B + A \cdot B \cdot C' + A' \cdot B' \cdot C \\ &= B \cdot (A' + A \cdot C') + B' \cdot (A' \cdot C). \end{aligned} \tag{5.3}$$

We can continue to expand a function as many times as it has variables until we reach the **canonical form** (a unique representation for any Boolean function that uses only minterms. A **minterm** is a product term that contains all the variables of F—such as A · B' · C). Expanding Eq. 5.3 again, this time *wrt* C, gives

$$F = C \cdot (A' \cdot B + A' \cdot B') + C' \cdot (A \cdot B + A' \cdot B). \tag{5.4}$$

As another example, we will use the Shannon expansion theorem to implement the following function using the ACT 1 Logic Module:

$$F = (A \cdot B) + (B' \cdot C) + D. \tag{5.5}$$

First we expand F *wrt* B:

$$\begin{aligned} F &= B \cdot (A + D) + B' \cdot (C + D) \\ &= B \cdot F2 + B' \cdot F1. \end{aligned} \tag{5.6}$$

Equation 5.6 describes a 2:1 MUX, with B selecting between two inputs: F(A = '1') and F(A = '0'). In fact Eq. 5.6 also describes the output of the ACT 1 Logic Module in Figure 5.1! Now we need to split up F1 and F2 in Eq. 5.6. Suppose we expand $F2 = F_B$ *wrt* A, and $F1 = F_{B'}$ *wrt* C:

$$F2 = A + D = (A \cdot 1) + (A' \cdot D), \tag{5.7}$$

$$F1 = C + D = (C \cdot 1) + (C' \cdot D). \tag{5.8}$$

From Eqs. 5.6–5.8 we see that we may implement F by arranging for A, B, C to appear on the select lines and '1' and D to be the data inputs of the MUXes in the ACT 1 Logic Module. This is the implementation shown in Figure 5.1(d), with connections: A0 = D, A1 = '1', B0 = D, B1 = '1', SA = C, SB = A, S0 = '0', and S1 = B.

Now that we know that we can implement Boolean functions using MUXes, how do we know which functions we can implement and how to implement them?

5.1.3 Multiplexer Logic as Function Generators

Figure 5.2 illustrates the 16 different ways to arrange '1's on a Karnaugh map corresponding to the 16 logic functions, F(A, B), of two variables. Two of these functions are not very interesting (F = '0', and F = '1'). Of the 16 functions, Table 5.1 shows the 10 that we can implement using just one 2:1 MUX. Of these 10 functions, the following six are useful:

- INV. The MUX acts as an inverter for one input only.
- BUF. The MUX just passes one of the MUX inputs directly to the output.
- AND. A two-input AND.
- OR. A two-input OR.
- AND1-1. A two-input AND gate with inverted input, equivalent to an NOR-11.
- NOR1-1. A two-input NOR gate with inverted input, equivalent to an AND-11.

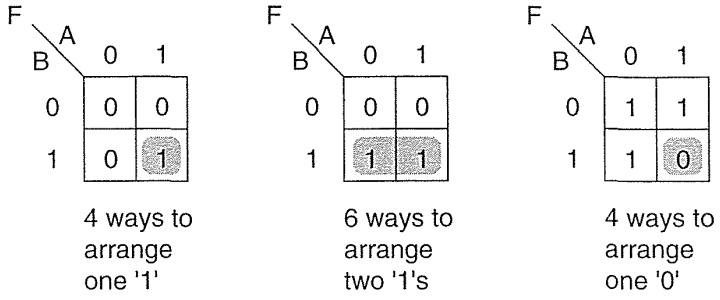


FIGURE 5.2 The logic functions of two variables.

TABLE 5.1 Boolean functions using a 2:1 MUX.

	Function, F	F =	Canonical form	Minterms[1]	Minterm code[2]	Function number[3]	M1[4]		
							A0	A1	SA
1	'0'	'0'	'0'	none	0000	0	0	0	0
2	NOR1-1(A, B)	(A+B')'	A'·B	1	0010	2	B	0	A
3	NOT(A)	A'	A'·B'+A'·B	0, 1	0011	3	0	1	A
4	AND1-1(A, B)	A·B'	A·B'	2	0100	4	A	0	B
5	NOT(B)	B'	A'·B'+A·B'	0, 2	0101	5	0	1	B
6	BUF(B)	B	A'·B+A·B	1, 3	1010	6	0	B	1
7	AND(A, B)	A·B	A·B	3	1000	8	0	B	A
8	BUF(A)	A	A·B'+A·B	2, 3	1100	9	0	A	1
9	OR(A, B)	A+B	A'·B+A·B'+A·B	1, 2, 3	1110	13	B	1	A
10	'1'	'1'	A'·B'+A'·B+A·B'+A·B	0, 1, 2, 3	1111	15	1	1	1

[1]The minterm numbers are formed from the product terms of the canonical form. For example, A·B' = 10 = 2.
[2]The minterm code is formed from the minterms. A '1' denotes the presence of that minterm.
[3]The function number is the decimal version of the minterm code.
[4]Connections to a two-input MUX: A0 and A1 are the data inputs and SA is the select input (see Eq. 5.11).

Figure 5.3(a) shows how we might view a 2:1 MUX as a **function wheel**, a three-input black box that can generate any one of the six functions of two-input variables: BUF, INV, AND-11, AND1-1, OR, AND. We can write the output of a function wheel as

$$F1 = \text{WHEEL1}(A, B). \tag{5.9}$$

where I define the wheel function as follows:

$$\text{WHEEL1}(A, B) = \text{MUX}(A0, A1, SA). \tag{5.10}$$

The MUX function is not unique; we shall define it as

$$\text{MUX}(A0, A1, SA) = A0 \cdot SA' + A1 \cdot SA. \tag{5.11}$$

The inputs (A0, A1, SA) are described using the notation

$$A0, A1, SA = \{A, B, '0', '1'\} \tag{5.12}$$

to mean that each of the inputs (A0, A1, and SA) may be any of the values: A, B, '0', or '1'. I chose the name of the wheel function because it is rather like a dial that you set to your choice of function. Figure 5.3(b) shows that the ACT 1 Logic Module is a function generator built from two function wheels, a 2:1 MUX, and a two-input OR gate.

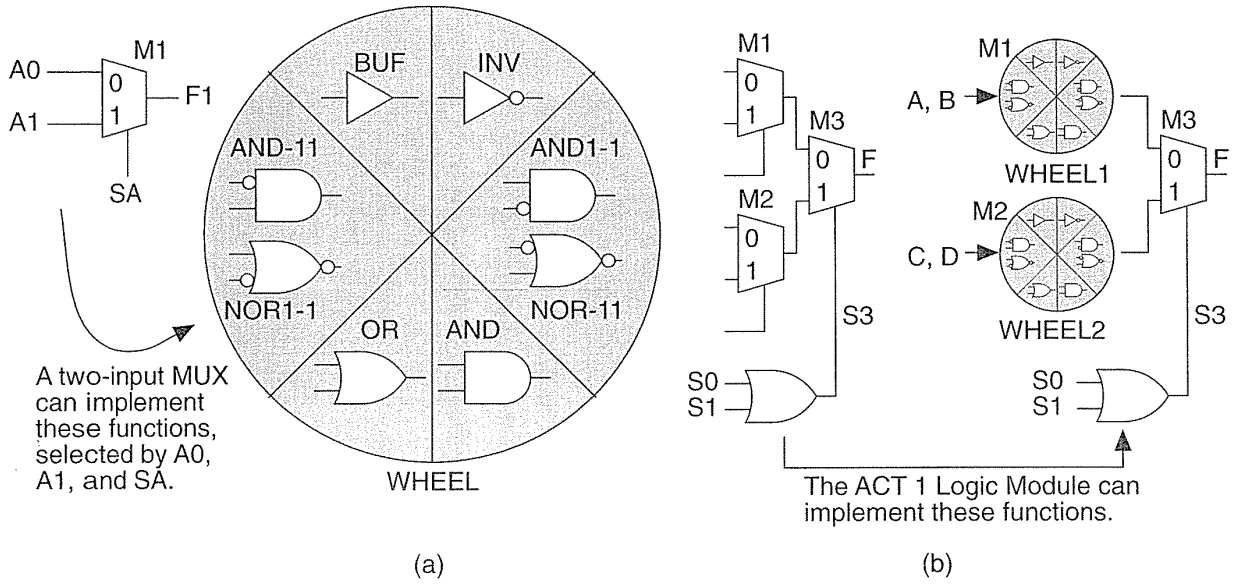


FIGURE 5.3 The ACT 1 Logic Module as a Boolean function generator. (a) A 2:1 MUX viewed as a function wheel. (b) The ACT 1 Logic Module viewed as two function wheels, an OR gate, and a 2:1 MUX.

We can describe the ACT 1 Logic Module in terms of two WHEEL functions:

$$F = \text{MUX}[\text{WHEEL1}, \text{WHEEL2}, \text{OR}(S0, S1)] \tag{5.13}$$

Now, for example, to implement a two-input NAND gate, F = NAND(A, B) = (A · B)', using an ACT 1 Logic Module we first express F as the output of a 2:1 MUX. To split up F we expand it *wrt* A (or *wrt* B; since F is symmetric in A and B):

$$F = A \cdot (B') + A' \cdot ('1') \tag{5.14}$$

Thus to make a two-input NAND gate we assign WHEEL1 to implement INV(B), and WHEEL2 to implement '1'. We must also set the select input to the MUX connecting WHEEL1 and WHEEL2, S0 + S1 = A—we can do this with S0 = A, S1 = '1'.

Before we get too carried away, we need to realize that we do not have to worry about how to use Logic Modules to construct combinational logic functions—this has already been done for us. For example, if we need a two-input NAND gate, we just use a NAND gate symbol and software takes care of connecting the inputs in the right way to the Logic Module.

How did Actel design its Logic Modules? One of Actel's engineers wrote a program that calculates how many functions of two, three, and four variables a given circuit would provide. The engineers tested many different circuits and chose the best one: a small, logically efficient circuit that implemented many functions. For example, the ACT 1 Logic Module can implement all two-input functions, most functions with three inputs, and many with four inputs.

Apart from being able to implement a wide variety of combinational logic functions, the ACT 1 module can implement sequential logic cells in a flexible and efficient manner. For example, you can use one ACT 1 Logic Module for a transparent latch or two Logic Modules for a flip-flop. The use of latches rather than flip-flops does require a shift to a two-phase clocking scheme using two nonoverlapping clocks and two clock trees. Two-phase synchronous design using latches is efficient and fast but, to handle the timing complexities of two clocks requires changes to synthesis and simulation software that have not occurred. This means that most people still use flip-flops in their designs, and these require two Logic Modules.

5.1.4 ACT 2 and ACT 3 Logic Modules

Using two ACT 1 Logic Modules for a flip-flop also requires added interconnect and associated parasitic capacitance to connect the two Logic Modules. To produce an efficient two-module flip-flop macro we could use extra antifuses in the Logic Module to cut down on the parasitic connections. However, the extra antifuses would have an adverse impact on the performance of the Logic Module in other macros. The alternative is to use a separate flip-flop module, reducing flexibility and increasing layout complexity. In the ACT 1 family Actel chose to use just one type of Logic Module. The ACT 2 and ACT 3 architectures use two different types of Logic Modules, and one of them does include the equivalent of a D flip-flop.

Figure 5.4 shows the ACT 2 and ACT 3 Logic Modules. The ACT 2 **C-Module** is similar to the ACT 1 Logic Module but is capable of implementing five-input logic functions. Actel calls its C-module a *combinatorial* module even though the module implements *combinational* logic. John Wakerly blames MMI for the introduction of the term combinatorial [Wakerly, 1994, p. 404].

The use of MUXes in the Actel Logic Modules (and in other places) can cause confusion in using and creating logic macros. For the Actel library, setting S = '0' selects input A of a two-input MUX. For other libraries setting S = '1' selects input A. This can lead to some very hard to find errors when moving schematics between libraries. Similar problems arise in flip-flops and latches with MUX inputs. A safer way to label the inputs of a two-input MUX is with '0' and '1', corresponding to the input selected when the select input is '1' or '0'. This notation can be extended to bigger MUXes, but in Figure 5.4, does the input combination S0 = '1' and S1 = '0' select input D10 or input D01? These problems are not caused by Actel, but by failure to use the IEEE standard symbols in this area.

The **S-Module** (**sequential module**) contains the same combinational function capability as the C-Module together with a **sequential element** that can be configured as a flip-flop. Figure 5.4(d) shows the sequential element implementation in the ACT 2 and ACT 3 architectures.

5.1.5 Timing Model and Critical Path

Figure 5.5(a) shows the **timing model** for the ACT family.[1] This is a simple timing model since it deals only with logic buried inside a chip and allows us only to estimate delays. We cannot predict the exact delays on an Actel chip until we have performed the place-and-route step and know how much delay is contributed by the interconnect. Since we cannot determine the exact delay before physical layout is complete, we call the Actel architecture **nondeterministic**.

Even though we cannot determine the preroute delays exactly, it is still important to estimate the delay on a logic path. For example, Figure 5.5(a) shows a typical situation deep inside an ASIC. Internal signal I1 may be from the output of a register (flip-flop). We then pass through some combinational logic, C1, through a register, S1, and then another register, S2. The register-to-register delay consists of a clock–Q delay, plus any combinational delay between registers, and the setup time for the next flip-flop. The speed of our system will depend on the slowest register–register delay or **critical path** between registers. We cannot make our clock period any longer than this or the signal will not reach the second register in time to be clocked.

Figure 5.5(a) shows an internal logic signal, I1, that is an input to a C-module, C1. C1 is drawn in Figure 5.5(a) as a box with a symbol comprising the overlapping letters "C" and "L" (borrowed from carpenters who use this symbol to mark the centerline on a piece of wood). We use this symbol to describe combinational logic.

[1] 1994 data book, p. 1-101.

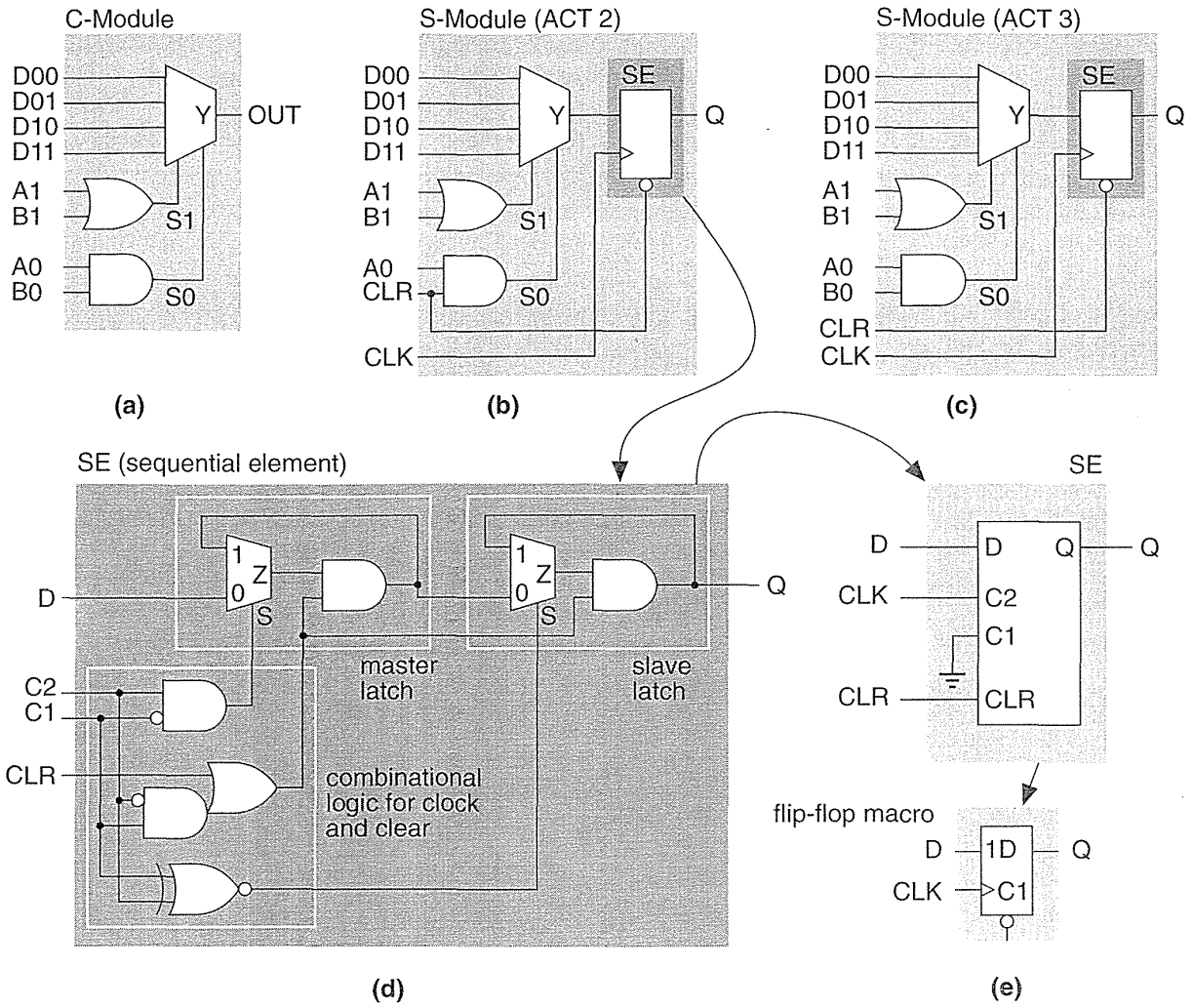


FIGURE 5.4 The Actel ACT 2 and ACT 3 Logic Modules. (a) The C-Module for combinational logic. (b) The ACT 2 S-Module. (c) The ACT 3 S-Module. (d) The equivalent circuit (without buffering) of the SE (sequential element). (e) The sequential element configured as a positive-edge–triggered D flip-flop. (*Source:* Actel.)

For the standard-speed grade ACT 3 (we shall look at speed grading in Section 5.1.6) the delay between the input of a C-module and the output is specified in the data book as a parameter, t_{PD}, with a maximum value of 3.0 ns.

The output of C1 is an input to an S-Module, S1, configured to implement combinational logic and a D flip-flop. The Actel data book specifies the minimum setup time for this D flip-flop as $t_{SUD} = 0.8$ ns. This means we need to get the data to the

input of S1 at least 0.8 ns before the rising clock edge (for a positive-edge–triggered flip-flop). If we do this, then there is still enough time for the data to go through the combinational logic inside S1 and reach the input of the flip-flop inside S1 in time to be clocked. We can guarantee that this will work because the combinational logic delay inside S1 is fixed.

The S-Module seems like good value—we get all the combinational logic functions of a C-module (with delay t_{PD} of 3 ns) as well as the setup time for a flip-flop for only 0.8 ns? ...not really. Next I will explain why not.

Figure 5.5(b) shows what is happening *inside* an S-Module. The setup and hold times, as measured *inside* (not outside) the S-Module, of the flip-flop are t'_{SUD} and t'_{H} (a prime denotes parameters that are measured inside the S-Module). The clock–Q propagation delay is t'_{CO}. The parameters t'_{SUD}, t'_{H}, and t'_{CO} are measured using the *internal* clock signal CLKi. The propagation delay of the combinational logic *inside* the S-Module is t'_{PD}. The delay of the combinational logic that drives the flip-flop clock signal (Figure 5.4d) is t'_{CLKD}.

From *outside* the S-Module, with reference to the outside clock signal CLK1:

$$\begin{aligned} t_{SUD} &= t'_{SUD} + (t'_{PD} - t'_{CLKD}), \\ t_{H} &= t'_{H} - (t'_{PD} - t'_{CLKD}), \\ t_{CO} &= t'_{CO} + t'_{CLKD}. \end{aligned} \tag{5.15}$$

Figure 5.5(c) shows an example of flip-flop timing. We have no way of knowing what the *internal* flip-flop parameters t'_{SUD}, t'_{H}, and t'_{CO} actually are, but we can assume some reasonable values (just for illustration purposes):

$$t'_{SUD} = 0.4 \text{ ns}, \quad t'_{H} = 0.9 \text{ ns}, \quad t'_{CO} = 0.4 \text{ ns}. \tag{5.16}$$

We do know the delay, t'_{PD}, of the combinational logic inside the S-Module. It is exactly the same as the C-Module delay, so $t'_{PD} = 3$ ns for the ACT 3. We do not know t'_{CLKD}; we shall assume a reasonable value of $t'_{CLKD} = 2.6$ ns (the exact value does not matter in the following argument).

Next we calculate the *external* S-Module parameters from Eq. 5.15 as follows:

$$t_{SUD} = 0.8 \text{ ns}, \quad t_{H} = 0.5 \text{ ns}, \quad t_{CO} = 3.0 \text{ ns}. \tag{5.17}$$

These are the same as the ACT 3 S-Module parameters shown in Figure 5.5(a), and I chose t'_{CLKD} and the values in Eq. 5.16 so that they would be the same. So now we see where the combinational logic delay of 3.0 ns has gone: 0.4 ns went into increasing the setup time and 2.6 ns went into increasing the clock–output delay, t_{CO}.

From the outside we can say that the combinational logic delay is *buried* in the flip-flop setup time. FPGA vendors will point this out as an advantage that they have. Of course, we are not getting something for nothing here. It is like borrowing money—you have to pay it back.

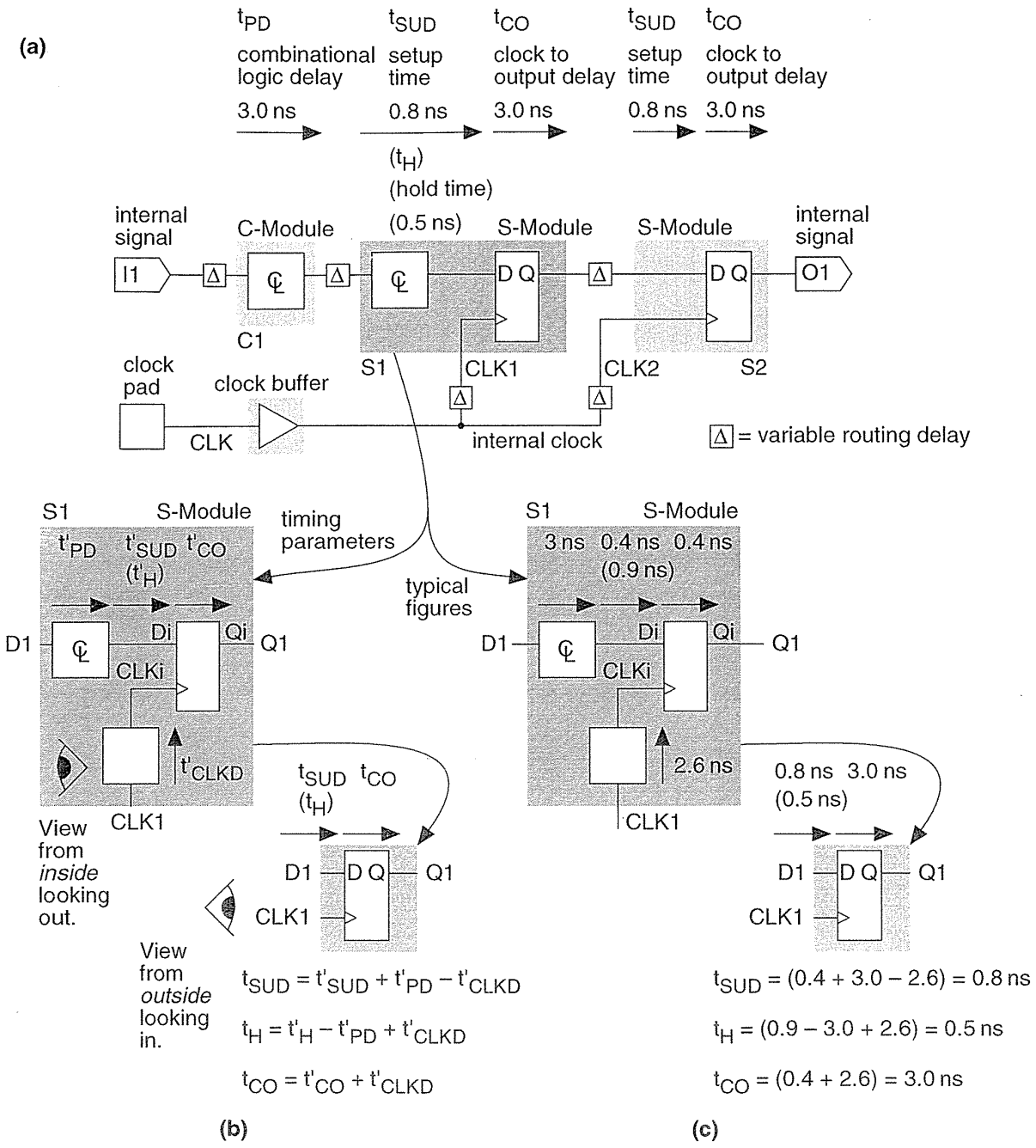


FIGURE 5.5 The Actel ACT timing model. (a) Timing parameters for a 'Std' speed grade ACT 3. (*Source*: Actel.) (b) Flip-flop timing. (c) An example of flip-flop timing based on ACT 3 parameters.

5.1.6 Speed Grading

Most FPGA vendors sort chips according to their speed (the sorting is known as **speed grading** or **speed binning**, because parts are automatically sorted into plastic bins by the production tester). You pay more for the faster parts. In the case of the ACT family of FPGAs, Actel measures performance with a special **binning circuit**, included on every chip, that consists of an input buffer driving a string of buffers or inverters followed by an output buffer. The parts are sorted from measurements on the binning circuit according to Logic Module propagation delay. The propagation delay, t_{PD}, is defined as the average of the rising (t_{PLH}) and falling (t_{PHL}) propagation delays of a Logic Module

$$t_{PD} = (t_{PLH} + t_{PHL}) / 2. \tag{5.18}$$

Since the transistor properties match so well across a chip, measurements on the binning circuit closely correlate with the speed of the rest of the Logic Modules on the die. Since the speeds of die on the same wafer also match well, most of the good die on a wafer fall into the same speed bin. Actel speed grades are: a 'Std' speed grade, a '1' speed grade that is approximately 15 percent faster, a '2' speed grade that is approximately 25 percent faster than 'Std', and a '3' speed grade that is approximately 35 percent faster than 'Std'.

5.1.7 Worst-Case Timing

If you use fully synchronous design techniques you only have to worry about how slow your circuit may be—not how fast. Designers thus need to know the maximum delays they may encounter, which we call the **worst-case timing**. Maximum delays in CMOS logic occur when operating under minimum voltage, maximum temperature, and slow–slow process conditions. (A slow–slow process refers to a process variation, or **process corner**, which results in slow *p*-channel transistors and slow *n*-channel transistors—we can also have fast–fast, slow–fast, and fast–slow process corners.)

Electronic equipment has to survive in a variety of environments and ASIC manufacturers offer several classes of qualification for different applications:

- Commercial. $V_{DD} = 5\,V \pm 5\%$, T_A (ambient) = 0 to +70 °C.
- Industrial. $V_{DD} = 5\,V \pm 10\%$, T_A (ambient) = –40 to +85 °C.
- Military: $V_{DD} = 5\,V \pm 10\%$, T_C (case) = –55 to +125 °C.
- Military: Standard MIL-STD-883C Class B.
- Military extended: Unmanned spacecraft.

ASICs for commercial application are cheapest; ASICs for the Cruise missile are very, very expensive. Notice that commercial and industrial application parts are specified with respect to the **ambient temperature** T_A (room temperature or the temperature inside the box containing the ASIC). Military specifications are relative to the package **case temperature**, T_C. What is really important is the temperature of

the transistors on the chip, the **junction temperature**, T_J, which is always higher than T_A (unless we dissipate zero power). For most applications that dissipate a few hundred mW, T_J is only 5–10 °C higher than T_A. To calculate the value of T_J we need to know the power dissipated by the chip and the thermal properties of the package—we shall return to this in Section 6.6.1, "Power Dissipation."

Manufacturers have to specify their operating conditions with respect to T_J and not T_A, since they have no idea how much power purchasers will dissipate in their designs or which package they will use. Actel used to specify timing under nominal operating conditions: $V_{DD} = 5.0$ V, and $T_J = 25$ °C. Actel and most other manufacturers now specify parameters under **worst-case commercial** conditions: $V_{DD} = 4.75$ V, and $T_J = +70$ °C.

Table 5.2 shows the ACT 3 commercial worst-case timing.[2] In this table Actel has included some estimates of the variable routing delay shown in Figure 5.5(a). These delay estimates depend on the number of gates connected to a gate output (the fanout).

When you design microelectronic systems (or design *anything*) you must use worst-case figures (just as you would design a bridge for the worst-case load). To convert nominal or typical timing figures to the worst case (or best case), we use measured, or empirically derived, constants called **derating factors** that are expressed either as a table or a graph. For example, Table 5.3 shows the ACT 3 derating factors from commercial worst-case to industrial worst-case and military worst-case conditions (assuming $T_J = T_A$). The ACT 1 and ACT 2 derating factors are approximately the same.[3]

TABLE 5.2 ACT 3 timing parameters.[1]

Family	Delay[2]	Fanout				
		1	2	3	4	8
ACT 3-3 (data book)	t_{PD}	2.9	3.2	3.4	3.7	4.8
ACT3-2 (calculated)	$t_{PD}/0.85$	3.41	3.76	4.00	4.35	5.65
ACT3-1 (calculated)	$t_{PD}/0.75$	3.87	4.27	4.53	4.93	6.40
ACT3-Std (calculated)	$t_{PD}/0.65$	4.46	4.92	5.23	5.69	7.38

Source: Actel.

[1] V_{DD} = 4.75 V, T_J(junction) = 70 °C. Logic module plus routing delay. All propagation delays in nanoseconds.
[2] The Actel '1' speed grade is 15 % faster than 'Std'; '2' is 25 % faster than 'Std'; '3' is 35 % faster than 'Std'.

[2] ACT 3: May 1995 data sheet, p. 1-173. ACT 2: 1994 data book, p. 1-51.

[3] 1994 data book, p. 1-12 (ACT 1), p. 1-52 (ACT 2), May 1995 data sheet, p. 1-174 (ACT 3).

TABLE 5.3 ACT 3 derating factors.[1]

	Temperature T_J (junction)/°C						
V_{DD}/V	−55	−40	0	25	70	85	125
4.5	0.72	0.76	0.85	0.90	1.04	1.07	1.17
4.75	0.70	0.73	0.82	0.87	1.00	1.03	1.12
5.00	0.68	0.71	0.79	0.84	0.97	1.00	1.09
5.25	0.66	0.69	0.77	0.82	0.94	0.97	1.06
5.5	0.63	0.66	0.74	0.79	0.90	0.93	1.01

Source: Actel.

[1]Worst-case commercial: V_{DD} = 4.75V, T_A (ambient) = +70 °C. Commercial: V_{DD} = 5V±5 %, T_A (ambient) = 0 to +70 °C. Industrial: V_{DD} = 5V±10%, T_A (ambient) = −40 to +85 °C. Military V_{DD} = 5V±10 %, T_C (case) = −55 to +125 °C.

As an example of a timing calculation, suppose we have a Logic Module on a 'Std' speed grade A1415A (an ACT 3 part) that drives four other Logic Modules and we wish to estimate the delay under worst-case industrial conditions. From the data in Table 5.2 we see that the Logic Module delay for an ACT 3 'Std' part with a fanout of four is t_{PD} = 5.7 ns (commercial worst-case conditions, assuming $T_J = T_A$).

If this were the slowest path between flip-flops (very unlikely since we have only one stage of combinational logic in this path), our estimated **critical path delay between registers**, t_{CRIT}, would be the combinational logic delay plus the flip-flop setup time plus the clock–output delay:

$$t_{CRIT}\,(\text{w-c commercial}) = t_{PD} + t_{SUD} + t_{CO}$$
$$= 5.7\ \text{ns} + 0.8\ \text{ns} + 3.0\ \text{ns} = 9.5\ \text{ns}. \qquad (5.19)$$

(I use w-c as an abbreviation for worst-case.) Next we need to adjust the timing to worst-case industrial conditions. The appropriate derating factor is 1.07 (from Table 5.3); so the estimated delay is

$$t_{CRIT}\,(\text{w-c industrial}) = 1.07 \times 9.5\ \text{ns} = 10.2\ \text{ns}. \qquad (5.20)$$

Let us jump ahead a little and assume that we can calculate that $T_J = T_A + 20\ °C = 105\ °C$ in our application. To find the derating factor at 105 °C we linearly interpolate between the values for 85 °C (1.07) and 125 °C (1.17) from Table 5.3). The interpolated derating factor is 1.12 and thus

$$t_{CRIT}\,(\text{w-c industrial}, T_J = 105\ °C) = 1.12 \times 9.5\ \text{ns} = 10.6\ \text{ns}, \qquad (5.21)$$

giving us an operating frequency of just less than 100 MHz.

It may seem unfair to calculate the worst-case performance for the slowest speed grade under the harshest industrial conditions—but the examples in the data books are always for the fastest speed grades under less stringent commercial conditions. If we want to illustrate the use of derating, then the delays can only get worse than the data book values! The ultimate word on logic delays for all FPGAs is the timing analysis provided by the FPGA design tools. However, you should be able to calculate whether or not the answer that you get from such a tool is reasonable.

5.1.8 Actel Logic Module Analysis

The sizes of the ACT family Logic Modules are close to the size of the base cell of an MGA. We say that the Actel ACT FPGAs use a **fine-grain architecture**. An advantage of a fine-grain architecture is that, whatever the mix of combinational logic to flip-flops in your application, you can probably still use 90 percent of an Actel FPGA. Another advantage is that synthesis software has an easier time mapping logic efficiently to the simple Actel modules.

The physical symmetry of the ACT Logic Modules greatly simplifies the place-and-route step. In many cases the router can swap equivalent pins on opposite sides of the module to ease channel routing. The design of the Actel Logic Modules is a balance between efficiency of implementation and efficiency of utilization. A simple Logic Module may reduce performance in some areas—as I have pointed out—but allows the use of fast and robust place-and-route software. Fast, robust routing is an important part of Actel FPGAs (see Section 7.1, "Actel ACT").

5.2 Xilinx LCA

Xilinx LCA (a trademark, denoting logic cell array) basic logic cells, **configurable logic blocks** or **CLBs**, are bigger and more complex than the Actel or QuickLogic cells. The Xilinx LCA basic logic cell is an example of a **coarse-grain architecture**. The Xilinx CLBs contain both combinational logic and flip-flops.

5.2.1 XC3000 CLB

The XC3000 CLB, shown in Figure 5.6, has five logic inputs (A–E), a common clock input (K), an asynchronous direct-reset input (RD), and an enable (EC). Using programmable MUXes connected to the SRAM programming cells, you can independently connect each of the two CLB outputs (X and Y) to the output of the flip-flops (QX and QY) or to the output of the combinational logic (F and G).

A 32-bit **look-up table** (LUT), stored in 32 bits of SRAM, provides the ability to implement combinational logic. Suppose you need to implement the function $F = A \cdot B \cdot C \cdot D \cdot E$ (a five-input AND). You set the contents of LUT cell number 31 (with address '11111') in the 32-bit SRAM to a '1'; all the other SRAM cells are set to '0'. When you apply the input variables as an address to the 32-bit SRAM, only

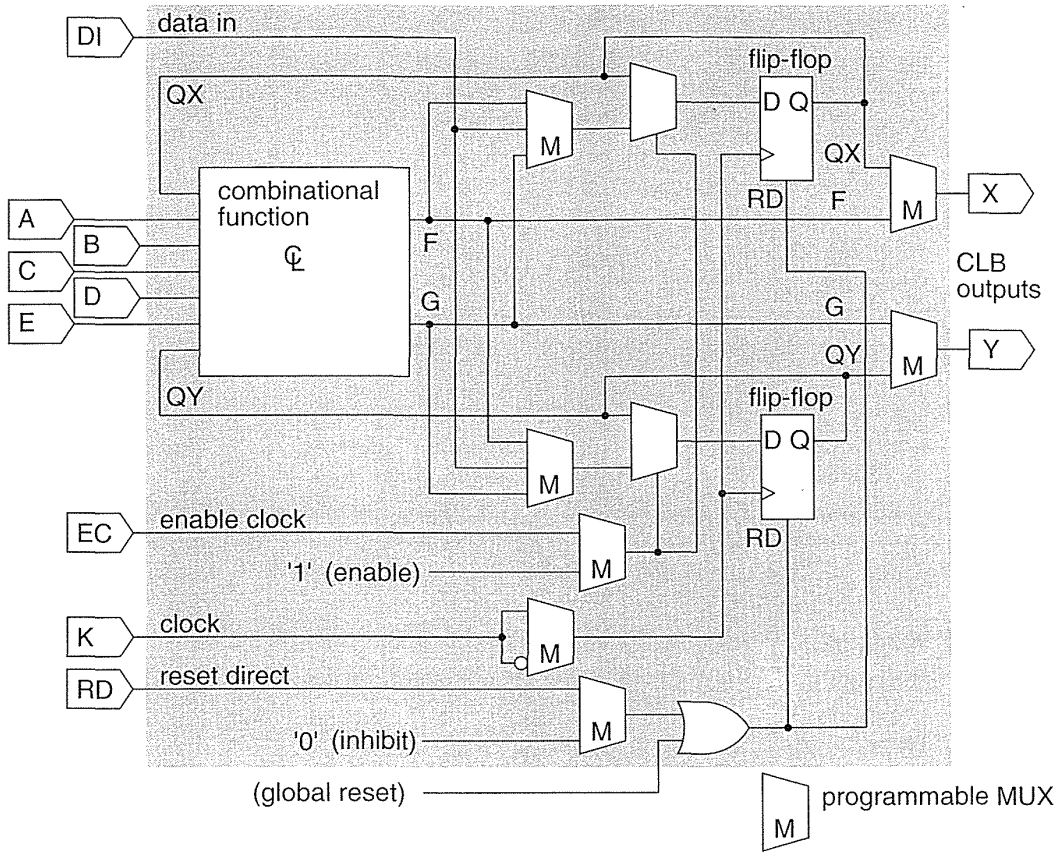


FIGURE 5.6 The Xilinx XC3000 CLB (configurable logic block). (*Source:* Xilinx.)

when ABCDE = '11111' will the output F be a '1'. This means that the CLB propagation delay is fixed, equal to the LUT access time, and independent of the logic function you implement.

There are seven inputs for the combinational logic in the XC3000 CLB: the five CLB inputs (A–E), and the flip-flop outputs (QX and QY). There are two outputs from the LUT (F and G). Since a 32-bit LUT requires only five variables to form a unique address ($32 = 2^5$), there are several ways to use the LUT:

- You can use five of the seven possible inputs (A–E, QX, QY) with the entire 32-bit LUT. The CLB outputs (F and G) are then identical.
- You can split the 32-bit LUT in half to implement two functions of four variables each. You can choose four input variables from the seven inputs (A–E, QX, QY). You have to choose two of the inputs from the five CLB inputs (A–E); then one function output connects to F and the other output connects to G.

- You can split the 32-bit LUT in half, using one of the seven input variables as a select input to a 2:1 MUX that switches between F and G. This allows you to implement some functions of six and seven variables.

5.2.2 XC4000 Logic Block

Figure 5.7 shows the CLB used in the XC4000 series of Xilinx FPGAs. This is a fairly complicated basic logic cell containing 2 four-input LUTs that feed a three-input LUT. The XC4000 CLB also has special fast carry logic hard-wired between CLBs. MUX control logic maps four control inputs (C1–C4) into the four inputs: LUT input H1, direct in (DIN), enable clock (EC), and a set/reset control (S/R) for the flip-flops. The control inputs (C1–C4) can also be used to control the use of the F' and G' LUTs as 32 bits of SRAM.

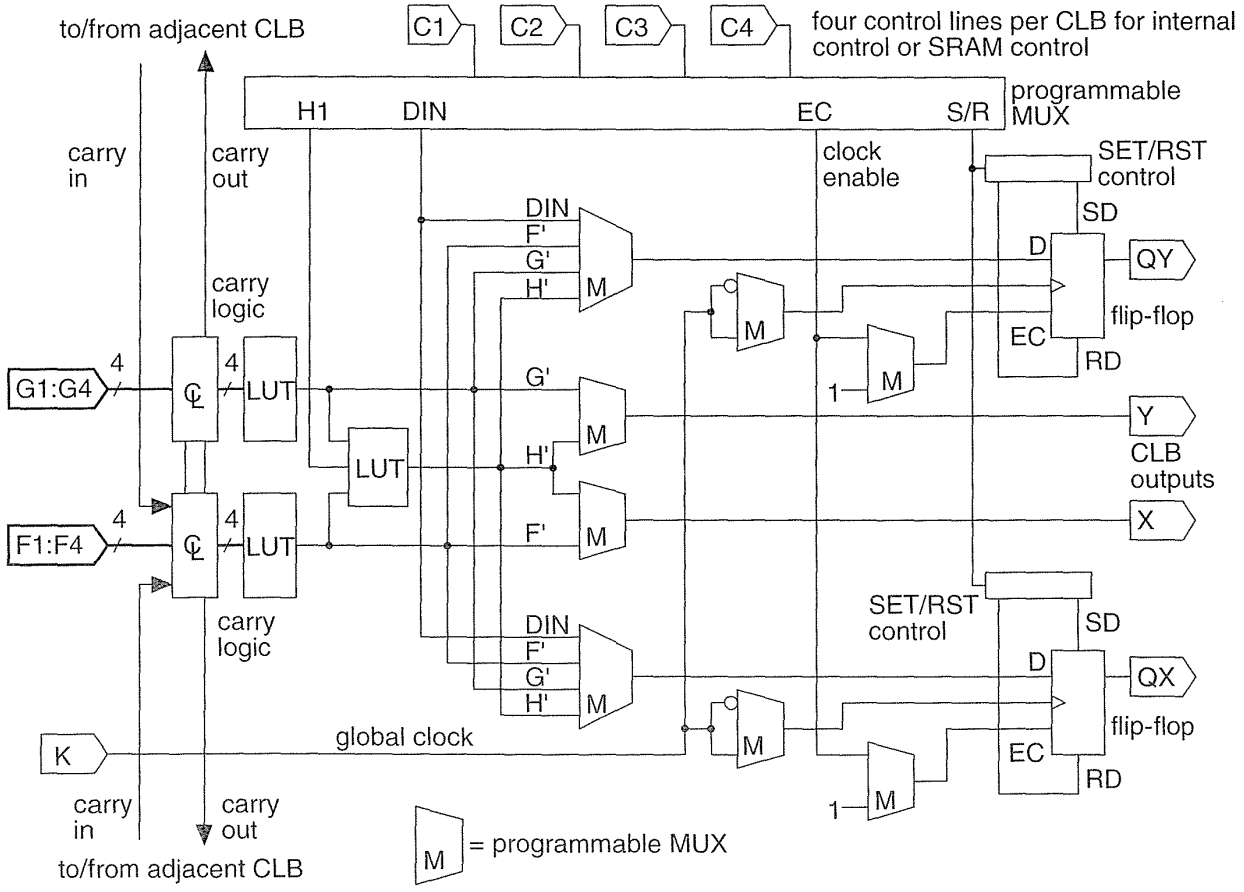


FIGURE 5.7 The Xilinx XC4000 family CLB (configurable logic block). (*Source:* Xilinx.)

5.2.3 XC5200 Logic Block

Figure 5.8 shows the basic logic cell, a **Logic Cell** or LC, used in the XC5200 family of Xilinx LCA FPGAs.[4] The LC is similar to the CLBs in the XC2000/3000/4000 CLBs, but simpler. Xilinx retained the term CLB in the XC5200 to mean a group of four LCs (LC0–LC3).

The XC5200 LC contains a four-input LUT, a flip-flop, and MUXes to handle signal switching. The arithmetic carry logic is separate from the LUTs. A limited capability to cascade functions is provided (using the MUX labeled F5_MUX in logic cells LC0 and LC2 in Figure 5.8) to gang two LCs in parallel to provide the equivalent of a five-input LUT.

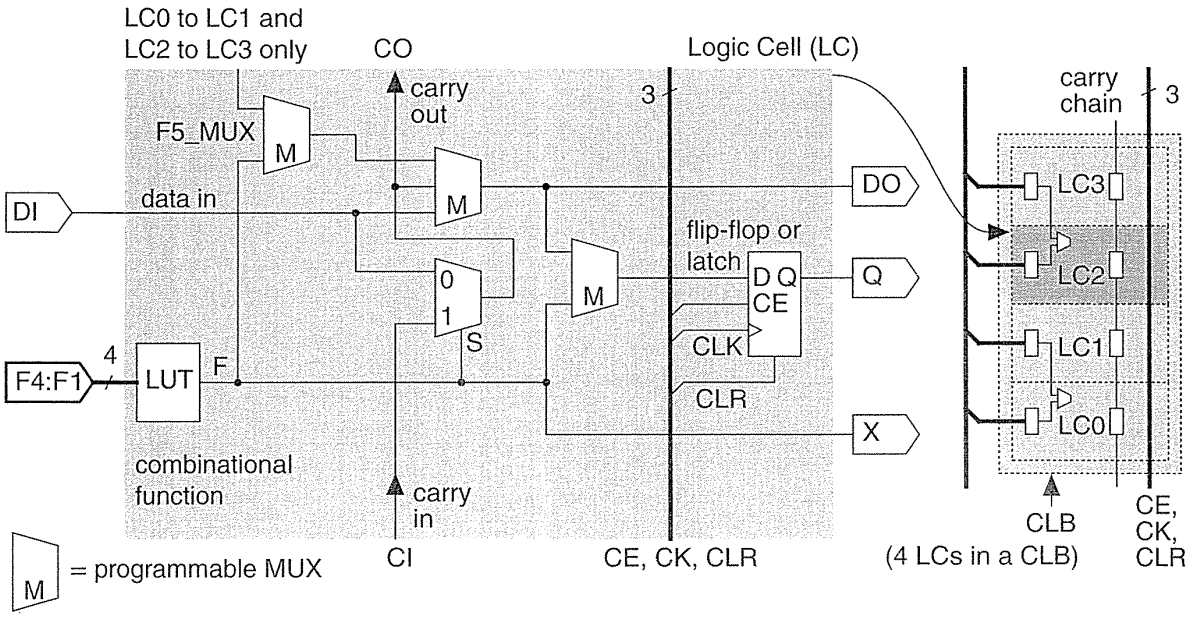


FIGURE 5.8 The Xilinx XC5200 family LC (Logic Cell) and CLB (configurable logic block). (*Source:* Xilinx.)

5.2.4 Xilinx CLB Analysis

The use of a LUT in a Xilinx CLB to implement combinational logic is both an advantage and a disadvantage. It means, for example, that an inverter is as slow as a five-input NAND. On the other hand a LUT simplifies timing of synchronous logic,

[4] Xilinx decided to use Logic Cell as a trademark in 1995 rather as if IBM were to use Computer as a trademark today. Thus we should now only talk of a Xilinx Logic Cell (with capital letters) and not Xilinx logic cells.

simplifies the basic logic cell, and matches the Xilinx SRAM programming technology well. A LUT also provides the possibility, used in the XC4000, of using the LUT directly as SRAM. You can configure the XC4000 CLB as a memory—either two 16×1 SRAMs or a 32×1 SRAM, but this is expensive RAM.

Figure 5.9 shows the timing model for Xilinx LCA FPGAs.[5] Xilinx uses two speed-grade systems. The first uses the maximum guaranteed toggle rate of a CLB flip-flop measured in MHz as a suffix—so higher is faster. For example a Xilinx XC3020-125 has a toggle frequency of 125 MHz. The other Xilinx naming system (which supersedes the old scheme, since toggle frequency is rather meaningless) uses the approximate delay time of the combinational logic in a CLB in nanoseconds—so lower is faster in this case. Thus, for example, an XC4010-6 has $t_{ILO} = 6.0$ ns (the correspondence between speed grade and t_{ILO} is fairly accurate for the XC2000, XC4000, and XC5200 but is less accurate for the XC3000).

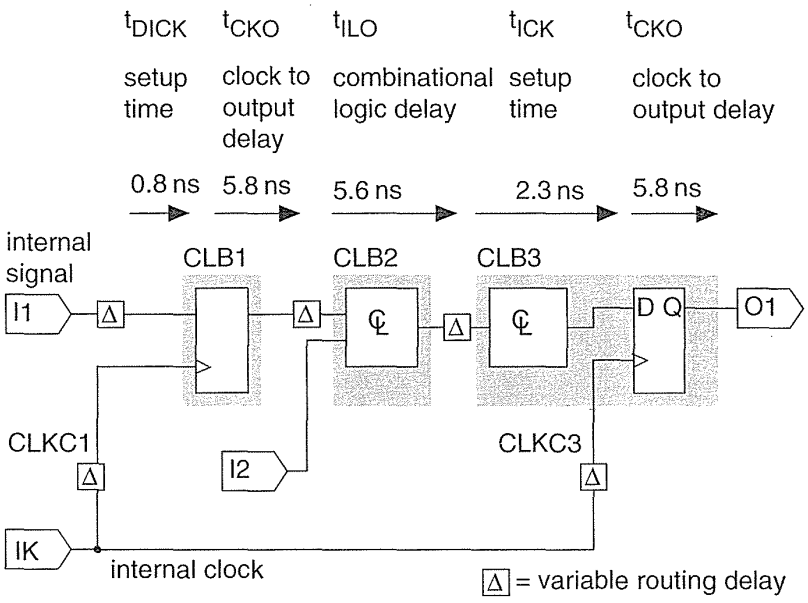


FIGURE 5.9 The Xilinx LCA timing model. The paths show different uses of CLBs (configurable logic blocks). The parameters shown are for an XC5210-6. (*Source:* Xilinx.)

The inclusion of flip-flops and combinational logic inside the basic logic cell leads to efficient implementation of state machines, for example. The coarse-grain architecture of the Xilinx CLBs maximizes performance given the size of the SRAM programming technology element. As a result of the increased complexity of the basic logic cell we shall see (in Section 7.2, "Xilinx LCA") that the routing between cells is more complex than other FPGAs that use a simpler basic logic cell.

[5]October 1995 (Version 3.0) data sheet.

5.3 Altera FLEX

Figure 5.10 shows the basic logic cell, a **Logic Element** (LE), that Altera uses in its FLEX 8000 series of FPGAs. Apart from the cascade logic (which is slightly simpler in the FLEX LE) the FLEX cell resembles the XC5200 LC architecture shown in Figure 5.8. This is not surprising since both architectures are based on the same SRAM programming technology. The FLEX LE uses a four-input LUT, a flip-flop, cascade logic, and carry logic. Eight LEs are stacked to form a Logic Array Block (the same term as used in the MAX series, but with a different meaning).

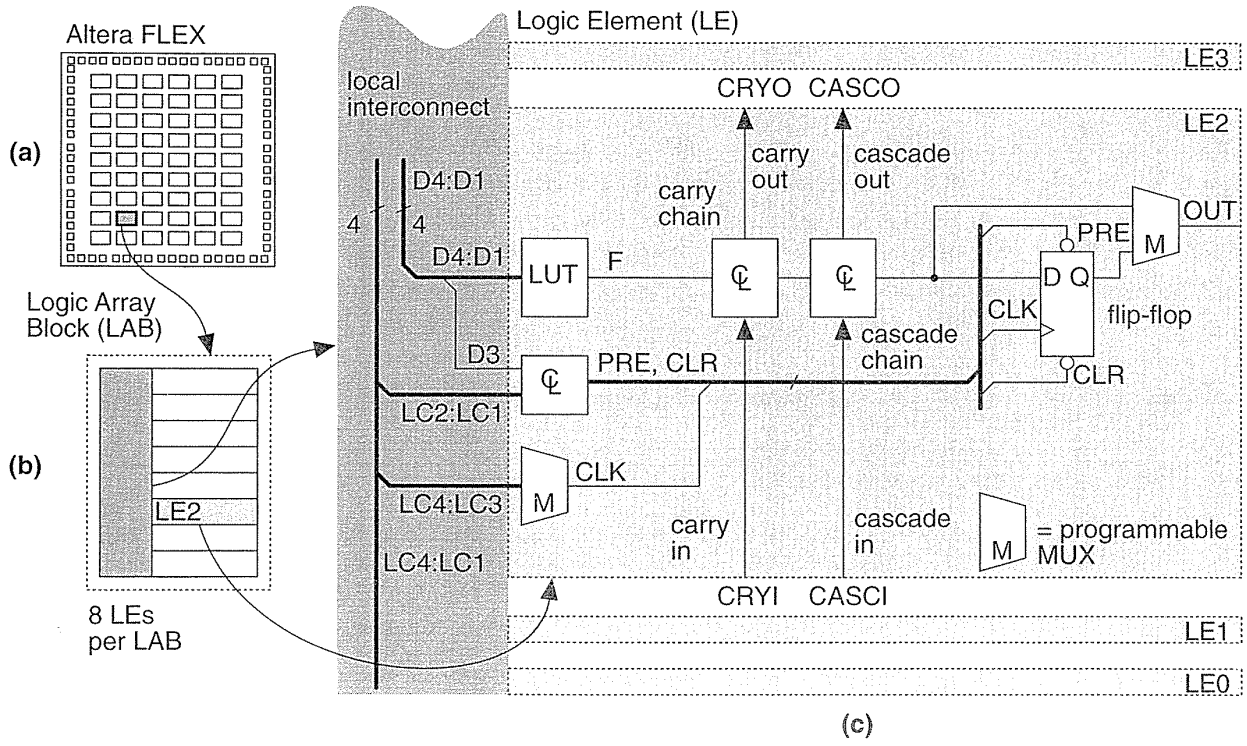


FIGURE 5.10 The Altera FLEX architecture. (a) Chip floorplan. (b) LAB (Logic Array Block). (c) Details of the LE (Logic Element). (*Source:* Altera (adapted with permission).)

5.4 Altera MAX

Suppose we have a simple two-level logic circuit that implements a sum of products as shown in Figure 5.11(a). We may redraw any two-level circuit using a regular structure (Figure 5.11b): a vector of buffers, followed by a vector of AND gates

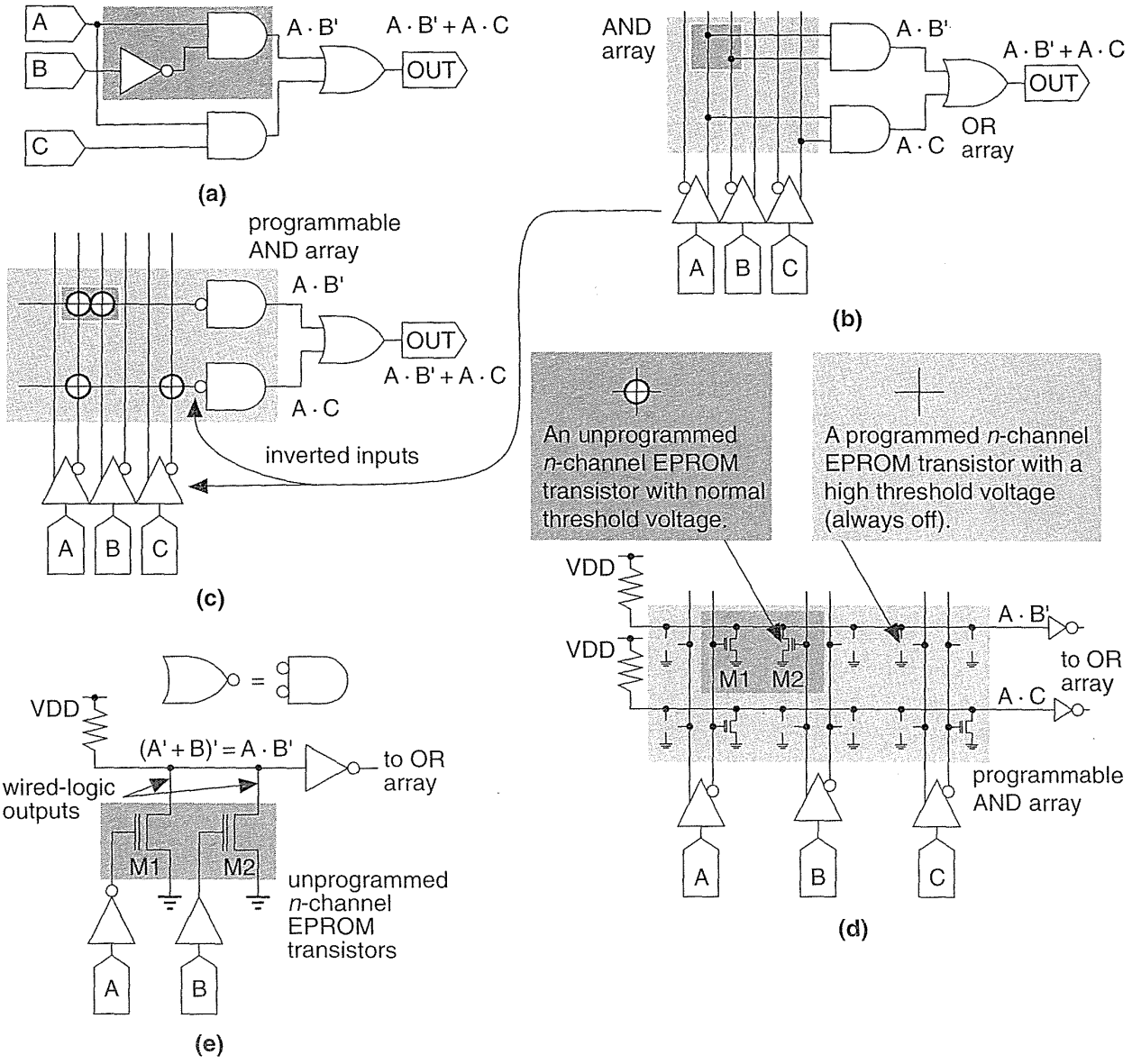


FIGURE 5.11 Logic arrays. (a) Two-level logic. (b) Organized sum of products. (c) A programmable-AND plane. (d) EPROM logic array. (e) Wired logic.

(which construct the product terms) that feed OR gates (which form the sums of the product terms). We can simplify this representation still further (Figure 5.11c), by drawing the input lines to a multiple-input AND gate as if they were one horizontal wire, which we call a **product-term line**. A structure such as Figure 5.11(c) is called **programmable array logic**, first introduced by Monolithic Memories as the PAL series of devices.

Because the arrangement of Figure 5.11(c) is very similar to a ROM, we sometimes call a horizontal product-term line, which would be the bit output from a ROM, the **bit line**. The vertical input line is the **word line**. Figure 5.11(d) and (e) show how to build the **programmable-AND array** (or product-term array) from EPROM transistors. The horizontal product-term lines connect to the vertical input lines using the EPROM transistors as pull-downs at each possible connection. Applying a '1' to the gate of an unprogrammed EPROM transistor pulls the product-term line low to a '0'. A programmed n-channel transistor has a threshold voltage higher than V_{DD} and is therefore always *off*. Thus a programmed transistor has no effect on the product-term line.

Notice that connecting the n-channel EPROM transistors to a **pull-up resistor** as shown in Figure 5.11(e) produces a **wired-logic** function—the output is high only if all of the outputs are high, resulting in a **wired-AND** function of the outputs. The product-term line is low when any of the inputs are high. Thus, to convert the wired-logic array into a programmable-AND array, we need to invert the sense of the inputs. We often conveniently omit these details when we draw the schematics of logic arrays, usually implemented as NOR–NOR arrays (so we need to invert the outputs as well). They are not minor details when you implement the layout, however.

Figure 5.12 shows how a programmable-AND array can be combined with other logic into a **macrocell** that contains a flip-flop. For example, the widely used **22V10** PLD, also called a registered PAL, essentially contains 10 of the macrocells shown in Figure 5.12. The part number, 22V10, denotes that there are 22 inputs (44 vertical input lines for both true and complement forms of the inputs) to the programmable AND array and 10 macrocells. The PLD or registered PAL shown in Figure 5.12 has an $2i \times jk$ programmable-AND array.

5.4.1 Logic Expanders

The basic logic cell for the Altera MAX architecture, a macrocell, is a descendant of the PAL. Using the **logic expander**, shown in Figure 5.13 to generate extra logic terms, it is possible to implement functions that require more product terms than are

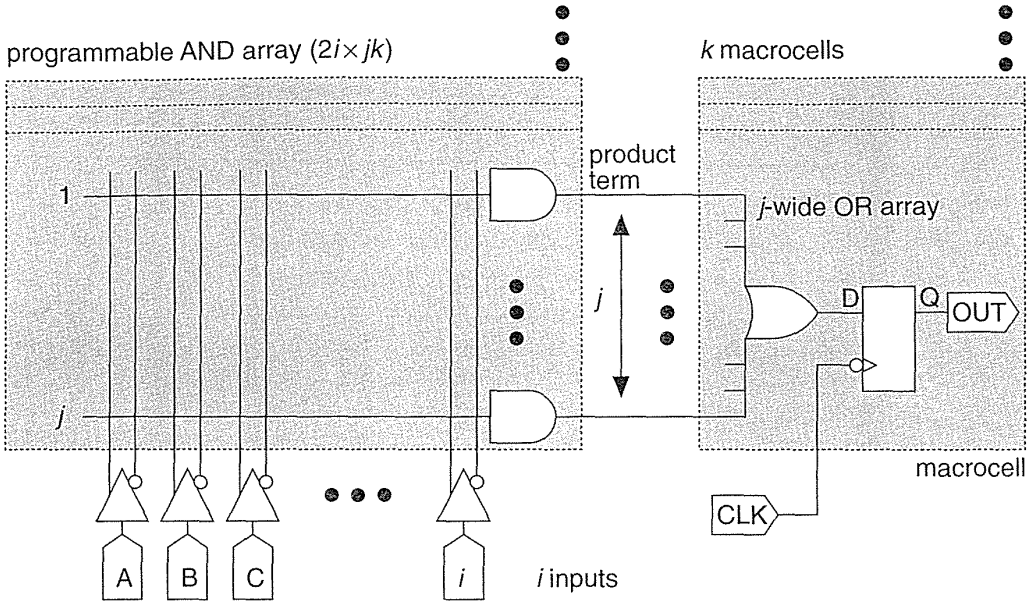


FIGURE 5.12 A registered PAL with *i* inputs, *j* product terms, and *k* macrocells.

available in a simple PAL macrocell. As an example, consider the following function:

$$F = A' \cdot C \cdot D + B' \cdot C \cdot D + A \cdot B + B \cdot C'. \tag{5.22}$$

This function has four product terms and thus we cannot implement F using a macrocell that has only a three-wide OR array (such as the one shown in Figure 5.13). If we rewrite F as a "sum of (products of products)" like this:

$$\begin{aligned} F &= (A' + B') \cdot C \cdot D + (A + C') \cdot B \\ &= (A \cdot B)' (C \cdot D) + (A' \cdot C)' \cdot B ; \end{aligned} \tag{5.23}$$

we can use logic expanders to form the **expander terms** $(A \cdot B)'$ and $(A' \cdot C)'$ (see Figure 5.13). We can even share these extra product terms with other macrocells if we need to. We call the extra logic gates that form these shareable product terms a **shared logic expander**, or just **shared expander**.

The disadvantage of the shared expanders is the extra logic delay incurred because of the second pass that you need to take through the product-term array. We usually do not know before the logic tools assign logic to macrocells (**logic assignment**) whether we need to use the logic expanders. Since we cannot predict the exact timing the Altera MAX architecture is not strictly **deterministic**. However,

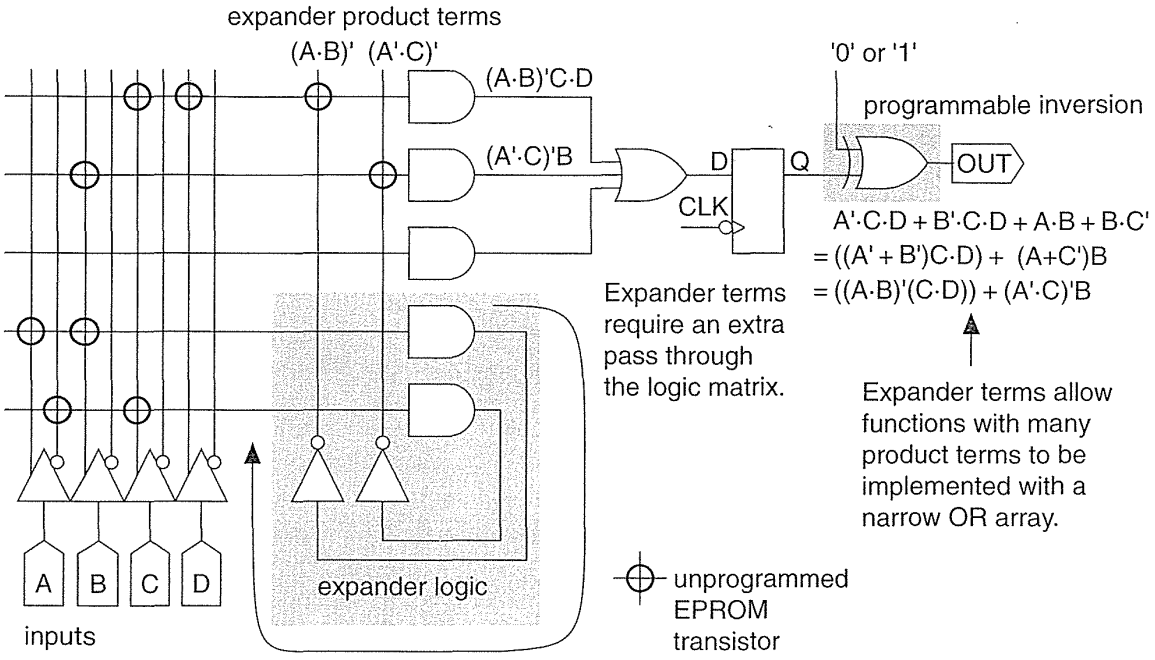


FIGURE 5.13 Expander logic and programmable inversion. An expander increases the number of product terms available and programmable inversion allows you to reduce the number of product terms you need.

once we do know whether a signal has to go through the array once or twice, we can simply and accurately predict the delay. This is a very important and useful feature of the Altera MAX architecture.

The expander terms are sometimes called **helper terms** when you use a PAL. If you use helper terms in a 22V10, for example, you have to go out to the chip I/O pad and then back into the programmable array again, using **two-pass logic**.

Another common feature in complex PLDs, also used in some PLDs, is shown in Figure 5.13. Programming one input of the XOR gate at the macrocell output allows you to choose whether or not to invert the output (a '1' for inversion or to a '0' for no inversion). This **programmable inversion** can reduce the required number of product terms by using a de Morgan equivalent representation instead of a conventional sum-of-products form, as shown in Figure 5.14.

As an example of using programmable inversion, consider the function

$$F = A \cdot B' + A \cdot C' + A \cdot D' + A' \cdot C \cdot D\,, \tag{5.24}$$

which requires four product terms—one too many for a three-wide OR array.

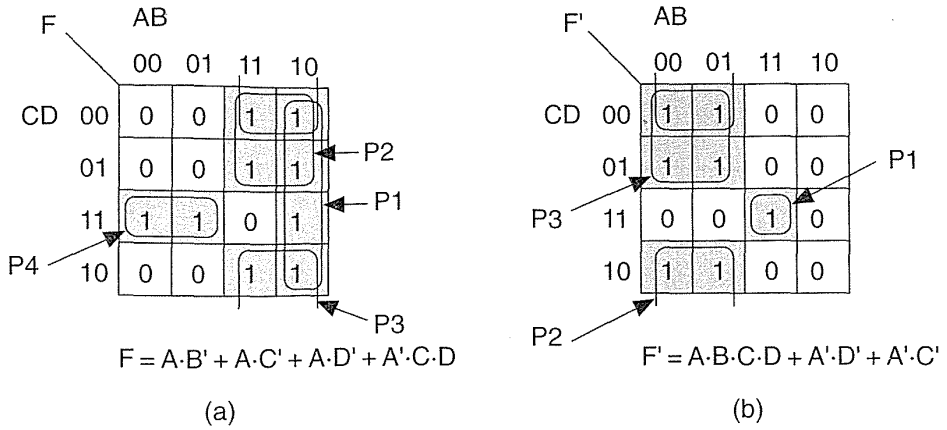


FIGURE 5.14 Use of programmed inversion to simplify logic: (a) The function $F = A \cdot B' + A \cdot C' + A \cdot D' + A' \cdot C \cdot D$ requires four product terms (P1–P4) to implement while (b) the complement, $F' = A \cdot B \cdot C \cdot D + A' \cdot D' + A' \cdot C'$ requires only three product terms (P1–P3).

If we generate the complement of F instead,

$$F' = A \cdot B \cdot C \cdot D + A' \cdot D' + A' \cdot C' , \tag{5.25}$$

this has only three product terms. To create F we invert F', using programmable inversion.

Figure 5.15 shows an Altera MAX macrocell and illustrates the architectures of several different product families. The implementation details vary among the families, but the basic features: wide programmable-AND array, narrow fixed-OR array, logic expanders, and programmable inversion—are very similar.[6] Each family has the following individual characteristics:

- A typical MAX 5000 chip has: 8 dedicated inputs (with both true and complement forms); 24 inputs from the chipwide interconnect (true and complement); and either 32 or 64 shared expander terms (single polarity). The MAX 5000 LAB looks like a 32V16 PLD (ignoring the expander terms).
- The MAX 7000 LAB has 36 inputs from the chipwide interconnect and 16 shared expander terms; the MAX 7000 LAB looks like a 36V16 PLD.
- The MAX 9000 LAB has 33 inputs from the chipwide interconnect and 16 local feedback inputs (as well as 16 shared expander terms); the MAX 9000 LAB looks like a 49V16 PLD.

[6] 1995 data book p. 274 (5000), p. 160 (7000), p. 126 (9000).

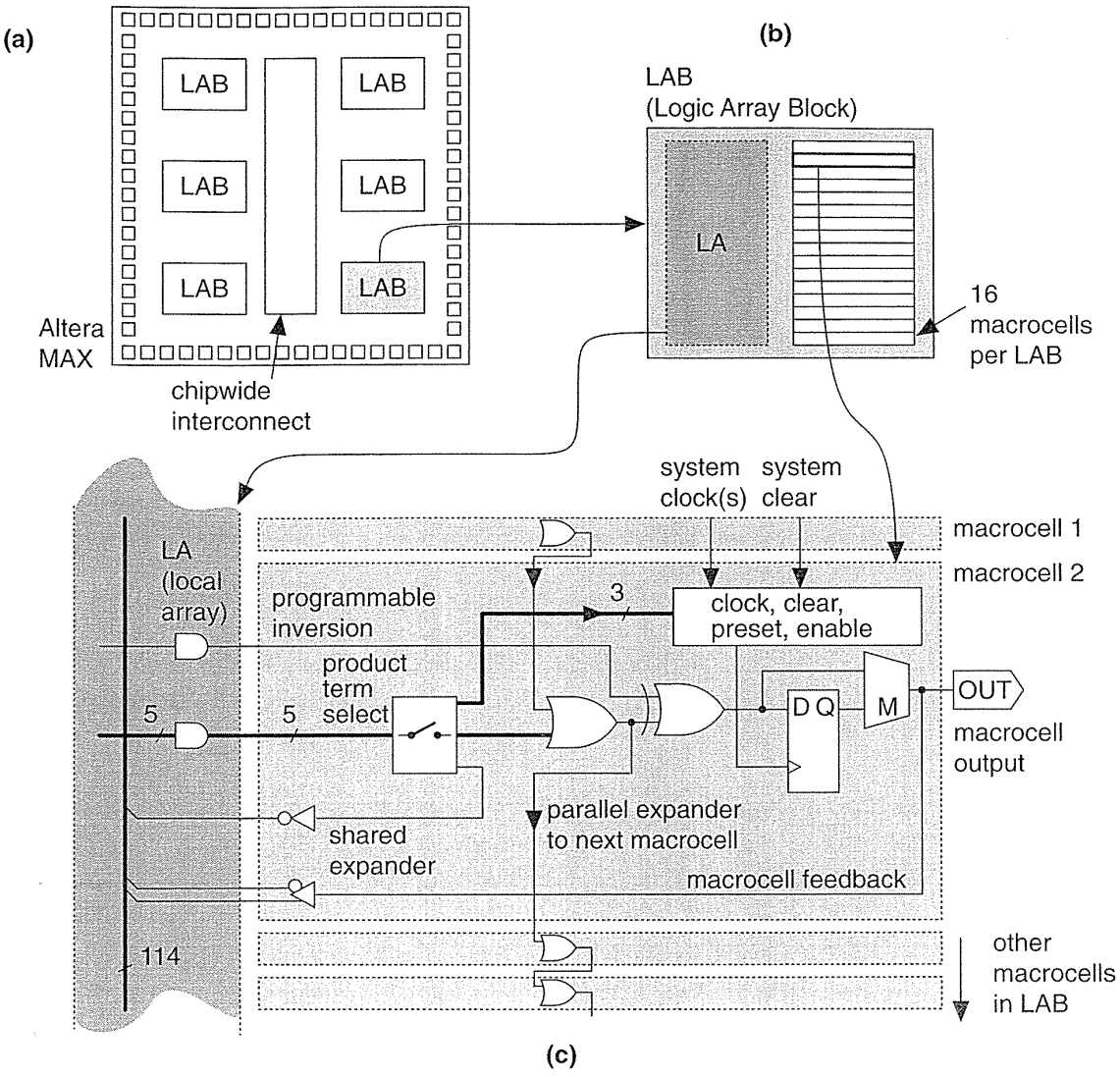


FIGURE 5.15 The Altera MAX architecture. (a) Organization of logic and interconnect. (b) A MAX family LAB (Logic Array Block). (c) A MAX family macrocell. The macrocell details vary between the MAX families—the functions shown here are closest to those of the MAX 9000 family macrocells.

5.4.2 Timing Model

Figure 5.16 shows the Altera MAX timing model for local signals.[7] For example, in Figure 5.16(a) an internal signal, I1, enters the local array (the LAB interconnect with a fixed delay $t_1 = t_{LOCAL} = 0.5$ ns), passes through the AND array (delay

[7] March 1995 data sheet, v2.

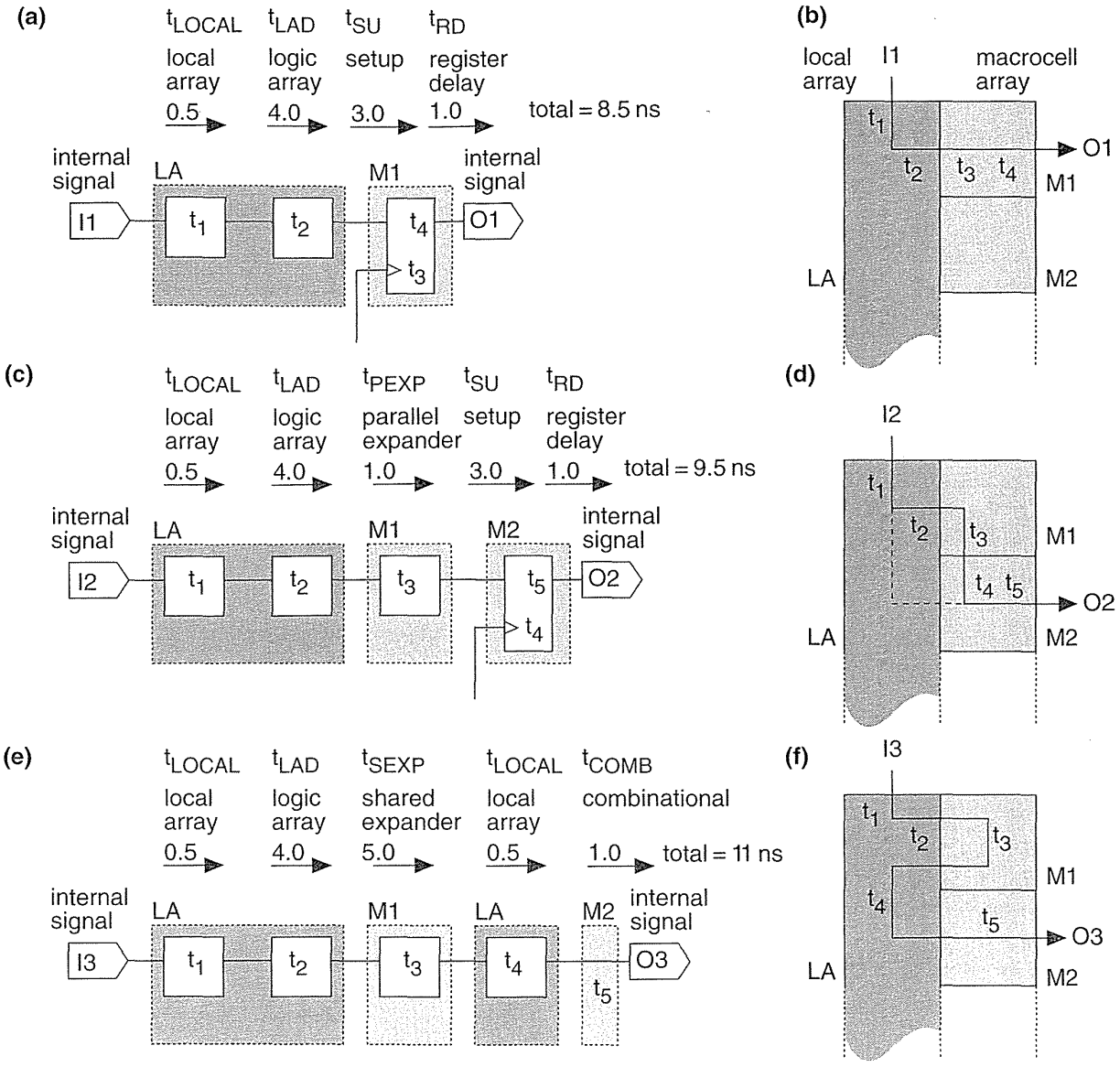


FIGURE 5.16 The timing model for the Altera MAX architecture. (a) A direct path through the logic array and a register. (b) Timing for the direct path. (c) Using a parallel expander. (d) Parallel expander timing. (e) Making two passes through the logic array to use a shared expander. (f) Timing for the shared expander (there is no register in this path). All timing values are in nanoseconds for the MAX 9000 series, '15' speed grade. (*Source:* Altera.)

$t_2 = t_{LAD} = 4.0$ ns), and to the macrocell flip-flop (with setup time, $t_3 = t_{SU} = 3.0$ ns, and clock–Q or **register delay**, $t_4 = t_{RD} = 1.0$ ns). The path delay is thus: $0.5 + 4 + 3 + 1 = 8.5$ ns.

Figure 5.16(c) illustrates the use of a **parallel logic expander**. This is different from the case of the *shared* expander (Figure 5.13), which required two passes in series through the product-term array. Using a parallel logic expander, the extra product term is generated in an adjacent macrocell in parallel with other product terms (not in series—as in a shared expander).

We can illustrate the difference between a parallel expander and a shared expander using an example function that we have used before (Eq. 5.22),

$$F = A' \cdot C \cdot D + B' \cdot C \cdot D + A \cdot B + B \cdot C' . \tag{5.26}$$

This time we shall use macrocell M1 in Figure 5.16(d) to implement F1 equal to the sum of the first three product terms in Eq. 5.26. We use F1 (using the parallel expander connection between adjacent macrocells shown in Figure 5.15) as an input to macrocell M2. Now we can form $F = F1 + B \cdot C'$ without using more than three inputs of an OR gate (the MAX 5000 has a three-wide OR array in the macrocell, the MAX 9000, as shown in Figure 5.15, is capable of handling five product terms in one macrocell—but the principle is the same). The total delay is the same as before, except that we add the delay of a parallel expander, $t_{PEXP} = 1.0$ ns. Total delay is then $8.5 + 1 = 9.5$ ns.

Figure 5.16(e) and (f) shows the use of a shared expander—similar to Figure 5.13.

The Altera MAX macrocell is more like a PLD than the other FPGA architectures discussed here; that is why Altera calls the MAX architecture a complex PLD. This means that the MAX architecture works well in applications for which PLDs are most useful: simple, fast logic with many inputs or variables.

5.4.3 Power Dissipation in Complex PLDs

A programmable-AND array in any PLD built using EPROM or EEPROM transistors uses a passive pull-up (a resistor or current source), and these macrocells consume **static power**. Altera uses a switch called the **Turbo Bit** to control the current in the programmable-AND array in each macrocell. For the MAX 7000, static current varies between 1.4 mA and 2.2 mA per macrocell in high-power mode (the current depends on the part—generally, but not always, the larger 7000 parts have lower operating currents) and between 0.6 mA and 0.8 mA in low-power mode. For the MAX 9000, the static current is 0.6 mA per macrocell in high-current mode and 0.3 mA in low-power mode, independent of the part size.[8] Since there are 16 macrocells in a LAB and up to 35 LABs on the largest MAX 9000 chip ($16 \times 35 = 560$ macrocells), just the static power dissipation in low-power mode can be substantial

[8] 1995 data book, p. 1-47.

($560 \times 0.3\,\text{mA} \times 5\,\text{V} = 840\,\text{mW}$). If all the macrocells are in high-power mode, the static power will double. This is the price you pay for having an (up to) 114-wide AND gate delay of a few nanoseconds ($t_{LAD} = 4.0\,\text{ns}$) in the MAX 9000. For any MAX 9000 macrocell in the low-power mode it is necessary to add a delay of between 15 ns and 20 ns to any signal path through the local interconnect and logic array (including t_{LAD} and t_{PEXP}).

5.5 Summary

Table 5.4 is a look-up table to Tables 5.5–5.9, which summarize the features of the logic cells used by the various FPGA vendors.

TABLE 5.4 Logic cell tables.

	Programmable ASIC family		Programmable ASIC family
Table 5.5	Actel (ACT 1) Xilinx (XC3000) Actel (ACT 2) Xilinx (XC4000)	Table 5.8	Actel (ACT 3) Xilinx LCA (XC5200) Altera FLEX (8000/10k)
Table 5.6	Altera MAX (EPM 5000) Xilinx EPLD (XC7200/7300) QuickLogic (pASIC 1)	Table 5.9	AMD MACH 5 Actel 3200DX Altera MAX (EPM 9000)
Table 5.7	Crosspoint (CP20K) Altera MAX (EPM 7000) Atmel (AT6000)		

The key points in this chapter are:

- The use of multiplexers, look-up tables, and programmable logic arrays
- The difference between fine-grain and coarse-grain FPGA architectures
- Worst-case timing design
- Flip-flop timing
- Timing models
- Components of power dissipation in programmable ASICs
- Deterministic and nondeterministic FPGA architectures

Next, in Chapter 6, we shall examine the I/O cells used by the various programmable ASIC families.

TABLE 5.5 Logic cells used by programmable ASICs.

	Actel ACT 1	Xilinx XC3000	Actel ACT 2	Xilinx XC4000
Basic logic cell	Logic module (LM)	CLB (Configurable Logic Block)	C-Module (combinatorial-module) and S-Module (sequential module)	CLB (Configurable Logic Block)
Logic cell contents	Three 2:1MUXes plus OR gate	32-bit LUT, 2 D flip-flops, 9 MUXes	C-Module: 4:1 MUX, 2-input OR, 2-input AND S-Module: 4-input MUX, 2-input OR, latch or D flip-flop	32-bit LUT, 2 D flip-flops, 10 MUXes, including fast carry logic E-suffix parts contain dual-port RAM.
Logic path delay	Fixed	Fixed with ability to bypass FF	Fixed	Fixed with ability to bypass FF
Combinational logic functions	Most 3-input, many 4-input functions (total 702 macros)	All 5-input functions plus 2 D flip-flops	Most 3- and 4-input functions (total 766 macros)	Two 4-input LUTs plus combiner with ninth input CLB as 32-bit SRAM (except D-suffix parts)
Flip-flop (FF) implementation	1 LM required for latch, 2 LMs required for flip-flops	2 D-flip-flops per CLB, latches can be built from pre-FF logic.	1 S-Module per D flip-flop; some FFs require 2 modules.	2 D flip-flops per CLB
Basic logic cells in each chip	LMs: A1010: 352 (8R × 44C) = 295 + 57 I/O A1020: 616 (14 R× 44C) = 547 + 69 I/O	64 (XC3020/A/L, XC3120/A) 100 (XC3030/A/L, XC3130/A) 144 (XC3042/A/L, XC3142/A) 224 (XC3064/A/L, XC3164/A) 320 (XC3090/A/L, XC3190/A) 484 (XC3195/A)	A1225: 451 = 231 S + 220 C A1240: 684 = 348 S + 336 C A1280: 1232 = 624 S + 608 C	64 (XC4002A) 100 (XC4003/A/E/H) 144 (XC4004A) 196 (XC4005/A/E/H) 256 (XC4006/E) 324 (XC4008/E) 400 (XC4010/D/E) 576 (XC4013/D/E) 784 (XC4020/E) 1024 (XC4025/E)

TABLE 5.6 Logic cells used by programmable ASICs.

	Altera MAX 5000	Xilinx XC7200/7300	QuickLogic pASIC 1
Basic logic cell	16 macrocells in a LAB (Logic Array Block) except EPM5032, which has 32 macrocells in a single LAB	9 macrocells within a FB (Functional Block), fast FBs (FFBs) omit ALU	Logic Cell (LC)
Logic cell contents	Macrocell: 64–106-wide AND, 3-wide OR array, 1 flip-flop, 2 MUXes, programmable inversion. 32–64 shared logic expander OR terms. LAB looks like a 32V16 PLD.	Macrocell: 21-wide AND, 16-wide OR array, 1 flip-flop, 1ALU FB looks like 21V9 PLD.	Four 2-input and two 6-input AND, three 2:1 MUXes and one D flip-flop
Logic path delay	Fixed (unless using shared logic expanders)	Fixed	Fixed
Combinational logic functions per logic cell	Wide input functions with ability to share product terms	Wide input functions with added 2-input ALU	All 3-input functions
Flip-flop (FF) implementation	1 D flip-flop or latch per macrocell. More can be constructed in arrays.	1 D flip-flop or latch per macrocell	1 D flip-flop per LC. LCs for other flip-flops not specified.
Basic logic cells in each chip	LABs: 32 (EPM5032) 64 (EPM5064) 128 (EPM5128) 128 (EPM5130) 192 (EPM5192)	FBs: 4 (XC7236A) 8 (XC7272A) 2 (XC7318) 4 (XC7336) 6 (XC7354) 8 (XC7372) 12 (XC73108) 16 (XC73144)	48 (QL6X8) 96 (QL8X12) 192 (QL12X16) 384 (QL16X24)

TABLE 5.7 Logic cells used by programmable ASICs.

	Crosspoint CP20K	Altera MAX 7k	Atmel AT6000
Basic logic cell	Transistor-pair tile (TPT), RAM-logic Tile (RLT)	16 macrocells in a LAB (Logic Array Block)	Cell
Logic cell contents	TPT: 2 transistors (0.5 gate). RLT: 3 inverters, two 3-input NANDs, 2-input NAND, 2-input AND.	Macrocell: wide AND, 5-wide OR array, 1 flip-flop, 3 MUXes, programmable inversion. 16 shared logic expander OR terms, plus parallel logic expander. LAB looks like a 36V16 PLD.	Two 5:1 MUXes, two 4:1 MUXes, 3:1 MUX, three 2:1 MUXes, 6 pass gates, four 2-input gates, 1 D flip-flop
Logic path delay	Variable	Fixed (unless using shared logic expanders)	Variable
Combinational functions per logic cell	TPT is smaller than a gate, approx. 2 TPTs = 1 gate.	Wide input functions with ability to share product terms	1-, 2-, and 3-input combinational configurations: 44 logical states and 72 physical states
Flip-flop (FF) implementation	D flip-flop requires 2 RLTs and 9 TPTs	1 D flip-flop or latch per macrocell. More can be constructed in arrays.	1 D flip-flop per cell
Basic logic cells in each chip	TPTs: 1760 (20220) 15,876 (22000) RLTs: 440 (20220) 3969 (22000)	Macrocells: 32 (EPM7032/V) 64 (EPM7064) 96 (EPM7096) 128 (EPM70128E) 160 (EPM70160E) 192 (EPM70192E) 256 (EPM70256E)	1024 (AT6002) 1600 (AT6003) 3136 (AT6005) 6400(AT6010)

TABLE 5.8 Logic cells used by programmable ASICs.

	Actel ACT 3	Xilinx XC5200	Altera FLEX 8000/10k
Basic logic cell	2 types of Logic Module: C-Module and S-Module (similar but not identical to ACT 2)	4 Logic Cells (LC) in a CLB (Configurable Logic Block)	8 Logic Elements (LE) in a Logic Array Block (LAB)
Logic cell contents (LUT = look-up table)	C-Module: 4:1 MUX, 2-input OR, 2-input AND. S-Module: 4:1 MUX, 2-input OR, latch or D flip-flop.	LC has 16-bit LUT, 1 flip-flop (or latch), 4 MUXes	16-bit LUT, 1 programmable flip-flop or latch, MUX logic for control, carry logic, cascade logic
Logic path delay	Fixed	Fixed	Fixed with ability to bypass FF
Combinational functions per logic cell	Most 3- and 4-input functions (total 766 macros)	One 4-input LUT per LC may be combined with adjacent LC to form 5-input LUT	4-input LUT may be cascaded with adjacent LE
Flip-flop (FF) implementation	1 D flip-flop (or latch) per S-Module; some FFs require 2 modules.	1 D flip-flop (or latch) per LC (4 per CLB)	1 D flip-flop (or latch) per LE
Basic logic cells in each chip	A1415: 104 S + 96 C A1425: 160 S + 150 C A1440: 288 S + 276 C A1460: 432 S + 416 C A14100: 697 S + 680 C	64 CLB (XC5202) 120 CLB (XC5204) 196 CLB (XC5206) 324 CLB (XC5210) 484 CLB (XC5215)	LEs: 208 (EPF8282/V/A /AV) 336 (EPF8452/A) 504 (EPF8636A) 672 (EPF8820/A) 1008 (EPF81188/A) 1296 (EPF81500/A) 576 (EPF10K10) 1152 (EPF10K20) 1728 (EPF10K30) 2304 (EPF10K40) 2880 (EPF10K50) 3744 (EPF10K70) 4992 (EPF10K100)

TABLE 5.9 Logic cells used by programmable ASICs.

	AMD MACH 5	Actel 3200DX	Altera MAX 9000
Basic logic cell	4 PAL Blocks in a Segment, 16 macrocells in a PAL Block	Based on ACT 2, plus D-module (decode) and dual-port SRAM	16 macrocells in a LAB (Logic Array Block)
Logic cell contents	20-bit to 32-bit wide OR array, switching logic, XOR gate, programmable flip-flop	C-Module: 4:1 MUX, 2-input OR, 2-input AND S-Module: 4-input MUX, 2-input OR, latch or D flip-flop D-module: 7-input AND, 2-input XOR	Macrocell: 114-wide AND, 5-wide OR array, 1 flip-flop, 5 MUXes, programmable inversion. 16 shared logic expander OR terms, plus parallel logic expander. LAB looks like a 49V16 PLD.
Logic path delay	Fixed	Fixed	Fixed (unless using expanders)
Combinational functions per logic cell	Wide input functions	Most 3- and 4-input functions (total 766 macros)	Wide input functions with ability to share product terms
Flip-flop (FF) implementation	1 D flip-flop or latch per macrocell	1 D flip-flop or latch per S-Module; some FFs require 2 modules.	1 D flip-flop or latch per macrocell. More can be constructed in arrays.
Basic logic cells in each chip	128 (M5-128) 192 (M5-192) 256 (M5-256) 320 (M5-320) 384 (M5-384) 512 (M5-512)	A3265DX: 510 S + 475 C + 20 D A32100DX: 700 S + 662 C + 20 D + 2 kSRAM A32140D): 954 S + 912 C + 24 D A32200DX: 1 230 S + 1 184 C + 24 D + 2.5 kSRAM A32300DX: 1 888 S + 1 833 C + 28 D + 3kSRAM A32400DX: 2 526 S + 2 466 C + 28 D + 4 kSRAM	Macrocells: 320 (EPM9320) 4×5 LABs 400 (EPM9400) 5×5 LABs 480 (EPM9480) 6×5 LABs 560 (EPM9560) 7×5 LABs

5.6 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

5.1 (Using the ACT 1 Logic Module, 30 min.) Consider the Actel ACT 1 Logic Module shown in Figure 5.1. Show how to implement: **(a)** a three-input NOR gate, **(b)** a three-input majority function gate, **(c)** a 2:1 MUX, **(d)** a half adder, **(e)** a three-input XOR gate, and **(f)** a four-input MUX.

5.2 (Worst-case and best-case timing, 10 min.) Seasoned digital CMOS designers do not worry too much when their designs stop working when they get too hot or when they reduce the supply voltage, but an ASIC that stops working either when increasing the supply voltage above normal or when it gets cold causes panic. Why?

5.3 (Typical to worst-case variation, 10 min.) The 1994 Actel data book (p. 1-5) remarks that: "the total derating factor from typical to worst-case for a standard ACT 1 array is only 1.19:1, compared to 2:1 for a masked gate array."

a. Can you explain why this is when the basic ACT 1 CMOS process is identical to a CMOS process for masked gate arrays?

b. There is a price to pay for the reduced spread in timing delays from typical to worst-case in an ACT 1 array. What is this disadvantage of the ACT 1 array over a masked gate array?

5.4 (ACT 2/3 sequential element, 30 min.). Show how the Actel ACT 2 and ACT 3 sequential element of Figure 5.4 (used in the S-Module) can be wired to implement:

a. a positive-edge–triggered flip-flop with clear,

b. a negative-edge–triggered flip-flop with clear,

c. a transparent-high latch,

d. a transparent-low latch, and

e. how it can be made totally transparent.

5.5 (*ACT 1 logic functions, 40 min.+)

a. How many different combinational functions of four logic variables are there?

b. of n variables? *Hint:* Consider the truth table.

c. The ACT 1 module can implement 213 of the 256 functions with three variables. How many of the 43 three-input functions that it cannot implement can you find?

d. (harder) Show that if you have access to both the true and complement form of the input variables you can implement all 256 logic functions of three variables with the ACT 1 Logic Module.

5.6 (Actel and Xilinx, 10 min.) The Actel Logic Modules (ACT 1, ACT 2, and ACT 3) have eight inputs and can implement most three-input logic functions and a few logic functions with four input variables. In contrast, the Xilinx XC5200 CLB,

for example, has only four inputs but can implement all logic functions with four or fewer variables. Why would Actel choose these logic cell designs and how can they be competitive with the Xilinx FPGA (which they are)?

5.7 (Actel address decoders, 10 min.) The maximum number of inputs that the ACT 1 Logic Module can handle is four. The ACT 2/ACT 3 C-module increases this to five.

a. How many ACT 1 Logic Modules do you need to implement a 32-bit wide address decoder (a 32-input AND gate)?

b. How many ACT 2/ACT 3 C-modules do you need?

5.8 (Altera shared logic expanders, 30 min.) Consider an Altera MAX 5000 logic array with three product-term lines. You cannot directly implement the function $Z = A \cdot B \cdot C + A \cdot B' \cdot C' + A' \cdot B \cdot C' + A' \cdot B' \cdot C$ with a programmable array logic macrocell that has only three product-term lines, since Z has four product terms.

a. How many Boolean functions of three variables are there that cannot be implemented with a programmable array logic macrocell that has only three product terms? *Hint:* Use a Karnaugh map to consider how many Boolean functions of three variables have more than three product terms in their sum-of-products representation.

b. Show how to use shared logic expanders that feed terms back into the product-term array to implement the function Z using a macrocell with three product terms.

c. How many shared expander lines do you need to add to be able to implement all the Boolean functions of three variables?

d. What is the largest number of product terms that you need to implement a Boolean function with n variables?

5.9 (Splitting the XC3000 CLB, 20 min.) In Section 5.2.1 we noted "You can split the (XC3000) 32-bit LUT in half, using one of the seven input variables to switch between the F and G outputs. This technique can implement some functions of six and seven variables."

a. Show which functions of six and seven variables can, and

b. which functions cannot, be implemented using this method.

5.10 (Programmable inversion, 20 min.) Section 5.4 described how the Altera MAX series logic cells can use programmable inversion to reduce the number of product terms needed to implement a function. Give another example of a function of four variables that requires four product terms. Is there a way to tell how many product terms a function may require?

5.11 (Table look-up mapping, 20 min.) Consider a four-input LUT (used in the CLB in the Xilinx XC2000, the first generation of Xilinx FPGAs, and in the XC5200 LE). This CLB can implement any Boolean function of four variables. Consider the function

$$Z = (A \cdot (B + C)) + (B \cdot D) + (E \cdot F \cdot G \cdot H \cdot I). \tag{5.27}$$

We can use four CLBs to implement Z as follows:

$$\begin{aligned} &\text{CLB1: } Z = Z1 + (B \cdot D) + Z3\,, \\ &\text{CLB2: } Z1 = A \cdot (B + C)\,, \\ &\text{CLB3: } Z3 = E \cdot F \cdot G \cdot Z5\,, \\ &\text{CLB4: } Z5 = H \cdot I\,. \end{aligned} \tag{5.28}$$

What is the length of the critical path? Find a better assignment in terms of area and critical path.

5.12 (Multiplexer mapping, 10 min.) Consider the function:

$$F = (A \cdot B) + (B' \cdot C) + D\,. \tag{5.29}$$

Use Shannon's expansion theorem to expand F *wrt* B:

$$F = B \cdot F1 + B' \cdot F2\,. \tag{5.30}$$

In other words express F in terms of B, B', F1, and F2 (*Hint:* F1 is a function of A and D only, F2 is a function of C and D only). Now expand F1 *wrt* A, and F2 *wrt* C. Using your answer, implement F using a single ACT 1 Logic Module.

5.13 (*Xilinx hazards, 10 min.) Explain why the outputs of the Xilinx CLBs are hazard-free for input changes in only one variable. Is this important?

5.14 (**Actel S-Modules, 10 min.) Notice that CLR is tied to the input corresponding to B0 of the C-module in the ACT 2 S-Module but the CLR input is separate from the B0 input in the ACT 3 version. Why?

5.15 (**Timing estimates, 60 min.) Using data book values for an FPGA architecture that you choose, and explaining your calculations carefully, estimate the (worst-case commercial) delay for the following functions: **(a)** 16-bit address decoder, **(b)** 8-bit ripple-carry adder, **(c)** 8-bit ripple-carry counter. Give your answers in terms of the data book symbols, and using actual parameters, for a speed grade that you specify, give an example calculation with the delay in ns.

5.16 (Actel logic. 30 min.) Table 5.10 shows how to use the Actel ACT 1 Logic Module to implement some of the 16 functions of two input variables. Complete this table.

5.17 (ACT 1 module implementation, 120 min.)

a. Show that the circuit shown in Figure 5.17, with buffered inputs and outputs, is equivalent to the one shown in Figure 5.1.

b. Show that the circuit for the ACT 1 Logic Module shown in Figure 5.18 is also the same.

c. Convert the circuit of Figure 5.18 to one that uses more efficient CMOS gates: inverters, AOI, and NAND gates.

d. (harder) Assume that the ACT 1 Logic Module has the equivalent of a 2X drive and the logic ratio is close to one. Compare your answer to part c against Figure 5.17 in terms of logical efficiency and logical area.

TABLE 5.10 Boolean functions using the ACT 1 Logic Module (Problem 5.16).

	Function, F	F =	Canonical form	Min-terms	M1			M2			OR1	
					A0	A1	SA	B0	B1	SB	S0	S1
1	0	0	0	—	0	0	0					
2	AND(A, B)	A · B	A · B	3	0	B	A					
3	AND1-1(A, B)	A · B'	A · B'	2	A	0	B					
4	NOR(A, B)	A + B	A' · B'	0								
5	NOR1-1(A, B)	A + B'	A' · B	1	B	0	A					
6	A	A	A · B' + A · B	2, 3	0	A	1					
7	B	B	A' · B + A · B	1, 3	0	B	1					
8	NOT(A)	A'	A' · B' + A' · B	0, 1	0	1	A					
9	NOT(B)	B'	A' · B' + A · B'	0, 2	0	1	B					
10	EXOR(A, B)	A ⊕ B	A' · B + A · B'	1, 2								
11	EXNOR(A, B)	(A ⊕ B)'	A' · B' + A · B	0, 3								
12	OR(A, B)	A + B	A' · B + A · B' + A · B	1, 2, 3	B	1	A					
13	OR1-1(A, B)	A + B'	A' · B' + A · B' + A · B	0, 2, 3								
14	NAND(A, B)	(A · B)'	A' · B' + A' · B + A · B'	0, 1, 2								
15	NAND1-1(A, B)	(A · B')'	A' · B' + A' · B + A · B	0, 1, 3								
16	1	1	A'·B'+A'·B+A·B'+A·B	0, 1, 2, 3	1	1	1					

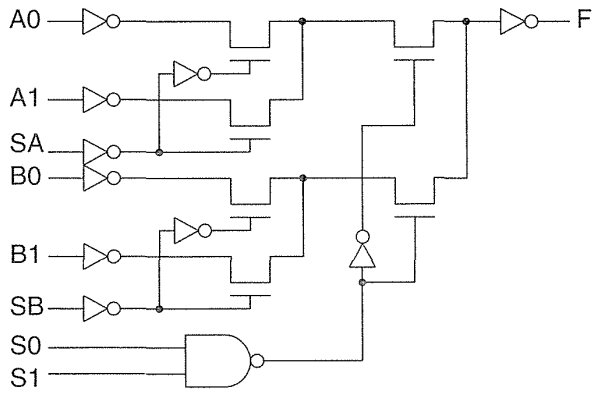


FIGURE 5.17 An alternative implementation of the ACT 1 Logic Module shown in Figure 5.1 (Problem 5.17).

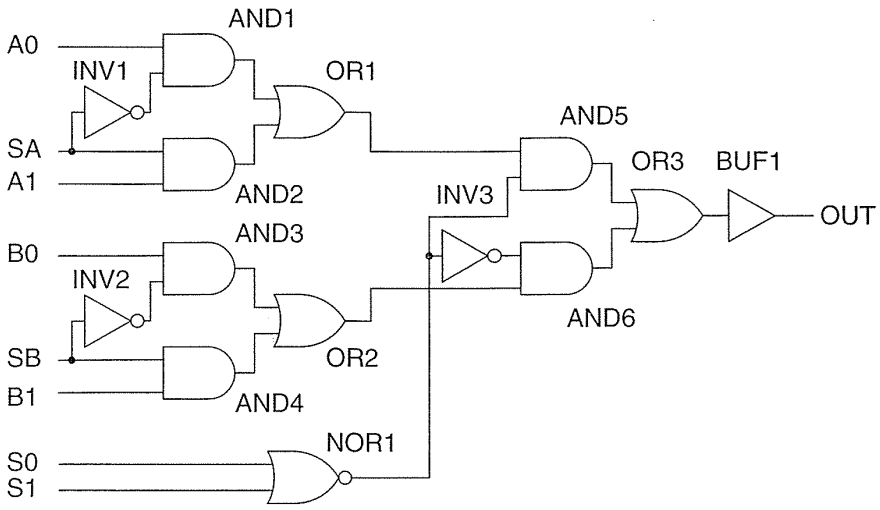


FIGURE 5.18 A schematic equivalent of the Actel ACT 1 Logic Module (Problem 5.17).

5.18 (**Xilinx CLB analysis, 60 min.) Table 5.11 shows some information derived from a die photo in the AT&T ATT3000 series data book that shows the eight by eight CLB matrix on an ATT3020 (equivalent to a XC3020) clearly. By measuring the die size in the photo and knowing the actual die size we can calculate the size of a CLB **matrix element** (ME) that includes a single XC3000 CLB as approximately 277 mil^2. The ME includes interconnect, SRAM, programming, and other resources as well as a CLB.

TABLE 5.11 ATT3020 die information (Problem 5.18).[1]

Parameter	Data book	Die photo	Calculated
3020 die width	183.5 mil	4.1 cm	—
3020 die height	219.3 mil	4.9 cm	—
3000 ME width	—	0.325 cm	14.55 mil = 370 μm
3000 ME height	—	0.425 cm	19.02 mil = 483 μm
3000 ME area	—	—	277 mil^2
3020 pad pitch	—	1.6 mm/pad	7.21 mil/pad

[1]Data from AT&T data book, July 1992, p. 3-76, MN92-024FPGA

a. The minimum feature size in the AT&T Holmdel twin-tub V process used for the ATT3000 family is 0.9 μm. Using a value of $\lambda = 0.45$ μm, calculate the Xilinx XC3000 ME size in λ^2.

b. Estimate, explaining your assumptions, the area of the XC4000 ME, and the XC5200 ME (both in λ^2).

c. Table 5.12 shows the ATT3000 die information. Using a value of 277 mil^2 for the ATT/XC3000 ME area, complete this table.

TABLE 5.12 ATT3000 die information (Problem 5.18).[1]

Die	Die height mil	Die width mil	Die area mil^2	Die area cm^2	CLBs	ME area mil^2	ME area cm^2
3020	219.3	183.5	40,242	0.26	8 × 8		
3030	259.8	215.0	55,857	0.36	10 × 10		
3042	295.3	242.5	71,610	0.46	12 × 12		
3064	270.9	366.5	99,285	0.64	16 × 14		
3090	437.0	299.2	130,750	0.84	16 × 20		

[1] Data from AT&T data book, July 1992, p. 3-75, MN92-024FPGA. 1 $mil^2 = 10^{-6}$ $in^2 = 2.54^2 \times 10^{-6}$ $cm^2 = 6.452 \times 10^{-6}$ cm^2

5.7 Bibliography

The book by Brown et al. [1992] on FPGAs deals with commercially available FPGAs and logic block architecture. There are several easily readable articles on FPGAs in the July 1993 issue of the *IEEE Proceedings* including articles by Rose et al. [1993] and Greene et al. [1993]. Greene's article is a good place to start digging deeper into the Actel FPGA architecture and gives an idea of the very complex problem of programming antifuses, something we have not discussed. Trimberger, who works at Xilinx, has edited a book on FPGAs [1994]. For those wishing to understand even more about the trade-offs in the different programmable ASIC architectures, a student of Stanford Professor Abbas El Gamal (one of the cofounders of Actel) has completed a Ph.D. on this topic [Kouloheris, 1993]. The best resources for information on FPGAs and their logic cells are the manufacturer's data sheets, data books, and application notes. The data books change every year or so as new products are released, so it is difficult to give specific references, but Xilinx, Actel, and Altera currently produce huge volumes complete with excellent design guides and application notes—you should obtain each of these even if you are not currently using that particular technology. Many of these are also online in Adobe Acrobat and PostScript format as well as in CD-ROM format (see also the bibliography in Chapter 4).

5.8 References

Brown, S. D., et al. 1992. *Field-Programmable Gate Arrays*. Norwell, MA: Kluwer Academic. 206 p. ISBN 0-7923-9248-5. TK7872.L64F54. Introduction to FPGAs, Commercially Available FPGAs, Technology Mapping for FPGAs, Logic Block Architecture, Routing for FPGAs, Flexibility of FPGA Routing Resources, A Theoretical Model for FPGA Routing. Includes an introduction to commercially available FPGAs. The rest of the book covers research on logic synthesis for FPGAs and FPGA architectures, concentrating on LUT-based architectures.

Greene, J., et al. 1993. "Antifuse field programmable gate arrays." *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1042–1056. Review article describing the Actel FPGAs. (Included in the Actel 1994 data book.)

Kouloheris, J. L. 1993. "Empirical study of the effect of cell granularity on FPGA density and performance." Ph.D. Thesis, Stanford, CA. 114 p. Detailed research study of the different FPGA architectures concentrating on structures similar to the Actel and Altera FPGAs.

Rose, J., et al. 1993. "A classification and survey of field-programmable gate array architectures." *Proceedings of the IEEE*, vol. 81, no. 7.

Trimberger, S. M. (Ed.). 1994. *Field-Programmable Gate Array Technology*. Boston: Kluwer Academic Publishers. ISBN 0-7923-9419-4. TK7895.G36.F54.

PROGRAMMABLE ASIC I/O CELLS

6

All programmable ASICs contain some type of **input/output cell (I/O cell)**. These I/O cells handle driving logic signals off-chip, receiving and conditioning external inputs, as well as handling such things as electrostatic protection. This chapter explains the different types of I/O cells that are used in programmable ASICs and their functions.

The following are different types of I/O requirements.

- *DC output*. Driving a resistive load at DC or low frequency (less than 1 MHz). Example loads are light-emitting diodes (LEDs), relays, small motors, and such. Can we supply an output signal with enough voltage, current, power, or energy?
- *AC output*. Driving a capacitive load with a high-speed (greater than 1 MHz) logic signal off-chip. Example loads are other logic chips, a data or address bus, ribbon cable. Can we supply a valid signal fast enough?
- *DC input*. Example sources are a switch, sensor, or another logic chip. Can we correctly interpret the digital value of the input?
- *AC input*. Example sources are high-speed logic signals (higher than 1 MHz) from another chip. Can we correctly interpret the input quickly enough?

- *Clock input.* Examples are system clocks or signals on a synchronous bus. Can we transfer the timing information from the input to the appropriate places on the chip correctly and quickly enough?
- *Power input.* We need to supply power to the I/O cells and the logic in the core, without introducing voltage drops or noise. We may also need a separate power supply to program the chip.

These issues are common to all FPGAs (and all ICs) so that the design of FPGA I/O cells is driven by the I/O requirements as well as the programming technology.

6.1 DC Output

Figure 6.1 shows a robot arm driven by three small motors together with switches to control the motors. The motor armature current varies between 50 mA and nearly 0.5 A when the motor is stalled. Can we replace the switches with an FPGA and drive the motors directly?

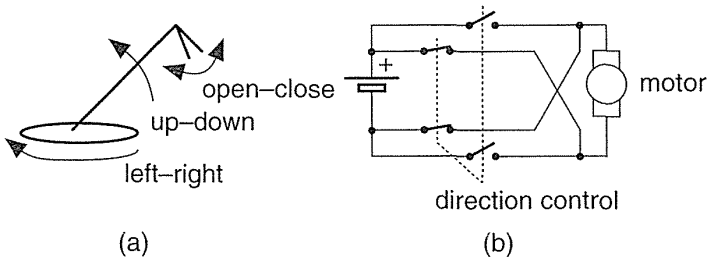


FIGURE 6.1 A robot arm. (a) Three small DC motors drive the arm. (b) Switches control each motor.

Figure 6.2 shows a CMOS complementary output buffer used in many FPGA I/O cells and its DC characteristics. Data books typically specify the output characteristics at two points, A (V_{OHmin}, I_{OHmax}) and B (V_{OLmax}, I_{OLmax}), as shown in Figure 6.2(d). As an example, values for the Xilinx XC5200 are as follows[1]:

- $V_{OLmax} = 0.4$ V, **low-level output voltage** at $I_{OLmax} = 8.0$ mA.
- $V_{OHmin} = 4.0$ V, **high-level output voltage** at $I_{OHmax} = -8.0$ mA.

By convention the **output current**, I_O, is positive if it flows into the output. Input currents, if there are any, are positive if they flow into the inputs. The Xilinx XC5200 specifications show that the output buffer can force the output pad to 0.4 V or lower and **sink** no more than 8 mA if the load requires it. CMOS logic inputs that may be connected to the pad draw minute amounts of current, but bipolar TTL inputs can require several milliamperes. Similarly, when the output is 4 V, the buffer

[1] XC5200 data sheet, October 1995 (v. 3.0).

can **source** 8 mA. It is common to say that $V_{OLmax} = 0.4$ V and $V_{OHmin} = 4.0$ V for a technology—without referring to the current values at which these are measured—strictly this is incorrect.

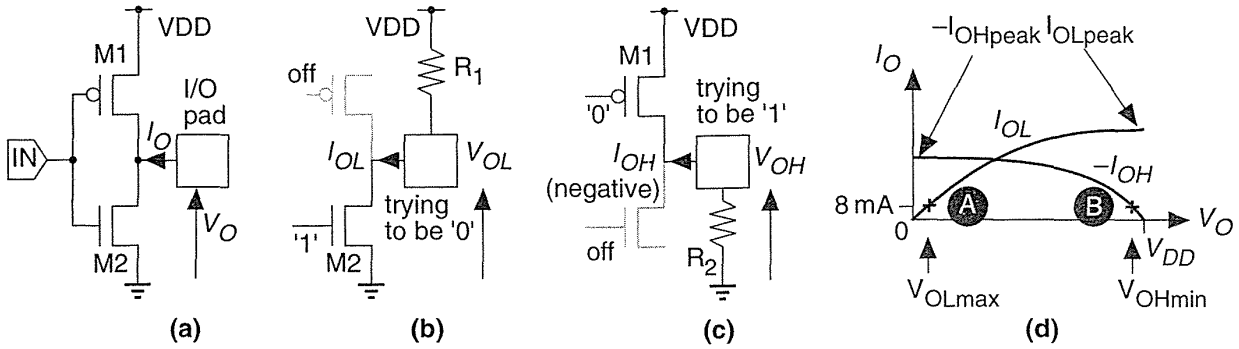


FIGURE 6.2 (a) A CMOS complementary output buffer. (b) Pull-down transistor M2 (M1 is off) sinks (to GND) a current I_{OL} through a pull-up resistor, R_1. (c) Pull-up transistor M1 (M2 is off) sources (from VDD) a current $-I_{OH}$ (I_{OH} is negative) through a pull-down resistor, R_2. (d) Output characteristics.

If we force the **output voltage**, V_O, of an output buffer, using a voltage supply, and measure the output current, I_O, that results, we find that a buffer is capable of sourcing and sinking far more than the specified I_{OHmax} and I_{OLmax} values. Most vendors do not specify output characteristics because they are difficult to measure in production. Thus we normally do not know the value of I_{OLpeak} or I_{OHpeak}; typical values range from 50 to 200 mA.

Can we drive the motors by connecting several output buffers in parallel to reach a peak drive current of 0.5 A? Some FPGA vendors do specifically allow you to connect adjacent output cells in parallel to increase the output drive. If the output cells are not adjacent or are on different chips, there is a risk of contention. Contention will occur if, due to delays in the signal arriving at two output cells, one output buffer tries to drive an output high while the other output buffer is trying to drive the same output low. If this happens we essentially short VDD to GND for a brief period. Although contention for short periods may not be destructive, it increases power dissipation and should be avoided.[2]

It is thus possible to parallel outputs to increase the DC drive capability, but it is not a good idea to do so because we may damage or destroy the chip (by exceeding the maximum metal electromigration limits). Figure 6.3 shows an alternative—a

[2]Actel specifies a maximum I/O current of ±20 mA for ACT3 family (1994 data book, p. 1-93) and its ES family. Altera specifies the maximum DC output current per pin, for example ±25 mA for the FLEX 10k (July 1995, v. 1 data sheet, p. 42).

simple circuit to boost the drive capability of the output buffers. If we need more power we could use two operational amplifiers (**op-amps**) connected as voltage followers in a bridge configuration. For even more power we could use discrete power MOSFETs or power op-amps.

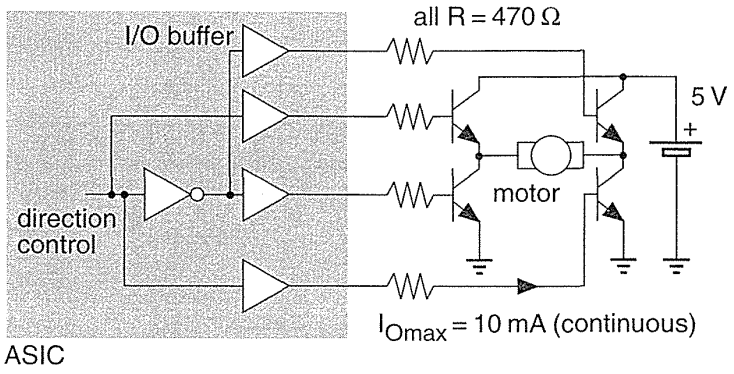


FIGURE 6.3 A circuit to drive a small electric motor (0.5 A) using ASIC I/O buffers. Any *npn* transistors with a reasonable gain ($\beta \approx 100$) that are capable of handling the peak current (0.5 A) will work with an output buffer that is capable of sourcing more than 5 mA. The 470 Ω resistors drop up to 5 V if an output buffer current approaches 10 mA, reducing the drive to the output transistors.

6.1.1 Totem-Pole Output

Figure 6.4(a) and (b) shows a **totem-pole** output buffer and its DC characteristics. It is similar to the TTL totem-pole output from which it gets its name (the totem-pole

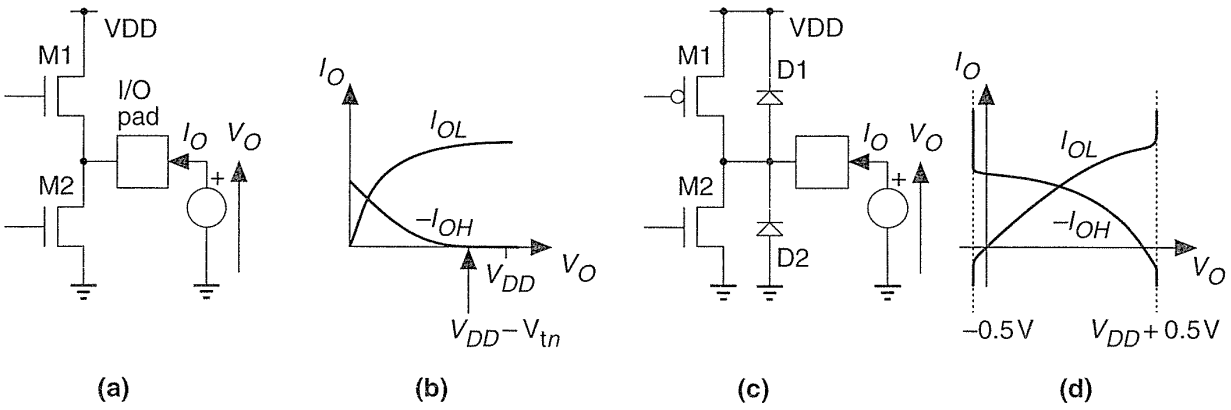


FIGURE 6.4 Output buffer characteristics. (a) A CMOS totem-pole output stage (both M1 and M2 are *n*-channel transistors). (b) Totem-pole output characteristics. (c) Clamp diodes, D1 and D2, in an output buffer (these diodes are present in all output buffers—totem-pole or complementary). (d) The clamp diodes start to conduct as the output voltage exceeds the supply voltage bounds.

circuit has two stacked transistors of the same type, whereas a complementary output uses transistors of opposite types). The high-level voltage, V_{OHmin}, for a totem pole is lower than V_{DD}. Typically V_{OHmin} is in the range of 3.5 V to 4.0 V (with $V_{DD} = 5$ V), which makes rising and falling delays more symmetrical and more closely matches TTL voltage levels. The disadvantage is that the totem pole will typically only drive the output as high as 3–4 V; so this would not be a good choice of FPGA output buffer to work with the circuit shown in Figure 6.3.

6.1.2 Clamp Diodes

Figure 6.4(c) show the connection of **clamp diodes** (D1 and D2) that prevent the I/O pad from voltage excursions greater than V_{DD} and less than V_{SS}. Figure 6.4(d) shows the resulting characteristics.

6.2 AC Output

Figure 6.5 shows an example of an off-chip three-state bus. Chips that have inputs and outputs connected to a bus are called **bus transceivers**. Can we use FPGAs to perform the role of bus transceivers? We will focus on one bit, B1, on bus BUSA, and we shall call it BUSA.B1. We need unique names to refer to signals on each chip; thus CHIP1.OE means the signal OE inside CHIP1. Notice that CHIP1.OE is not connected to CHIP2.OE.

Figure 6.6 shows the timing of part of a **bus transaction** (a sequence of signals on a bus):

1. Initially CHIP2 drives BUSA.B1 high (CHIP2.D1 is '1' and CHIP2.OE is '1').
2. The buffer output enable on CHIP2 (CHIP2.OE) goes low, **floating** the bus. The bus will stay high because we have a bus keeper, BK1.
3. The buffer output enable on CHIP3 (CHIP3.OE) goes high and the buffer drives a low onto the bus (CHIP3.D1 is '0').

We wish to calculate the delays involved in driving the off-chip bus in Figure 6.6. In order to find t_{float}, we need to understand how Actel specifies the delays for its I/O cells. Figure 6.7(a) shows the circuit used for measuring I/O delays for the ACT FPGAs. These measurements do not use the same trip points that are used to characterize the internal logic (Actel uses input and output trip points of 0.5 for internal logic delays).

Notice in Figure 6.7(a) that when the output enable E is '0' the output is **three-stated** (**high-impedance** or **hi-Z**). Different companies use different polarity and naming conventions for the "output enable" signal on a three-state buffer. To

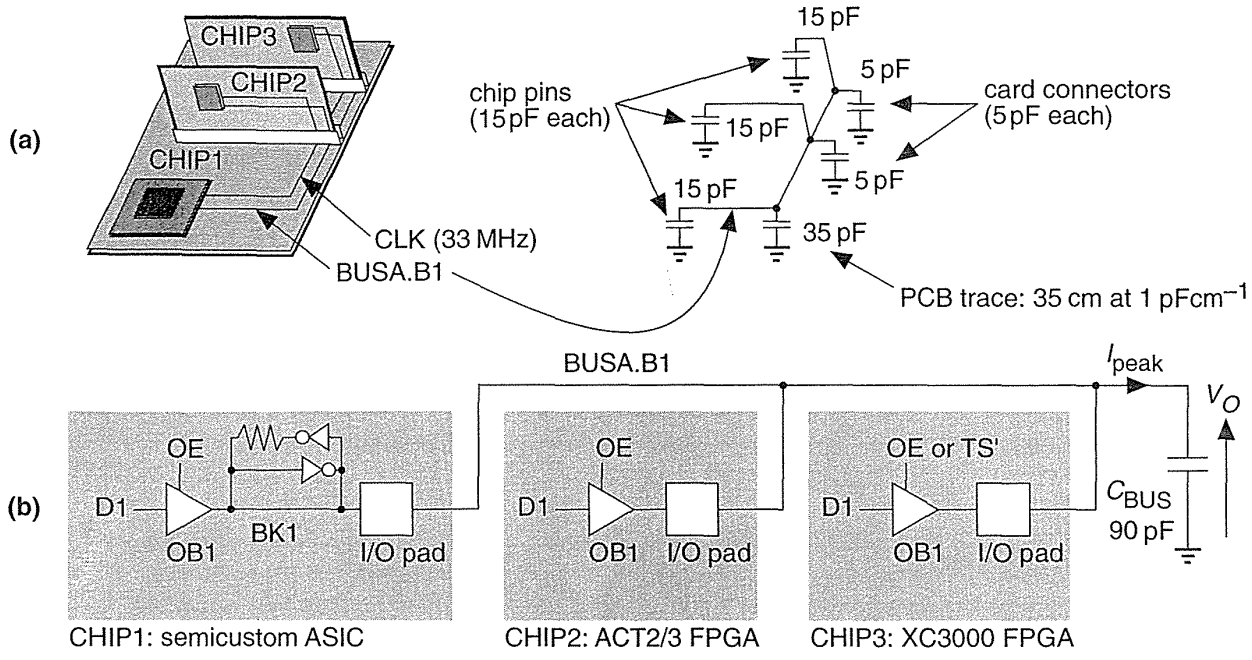


FIGURE 6.5 A three-state bus. (a) Bus parasitic capacitance. (b) The output buffers in each chip. The ASIC CHIP1 contains a bus keeper, BK1.

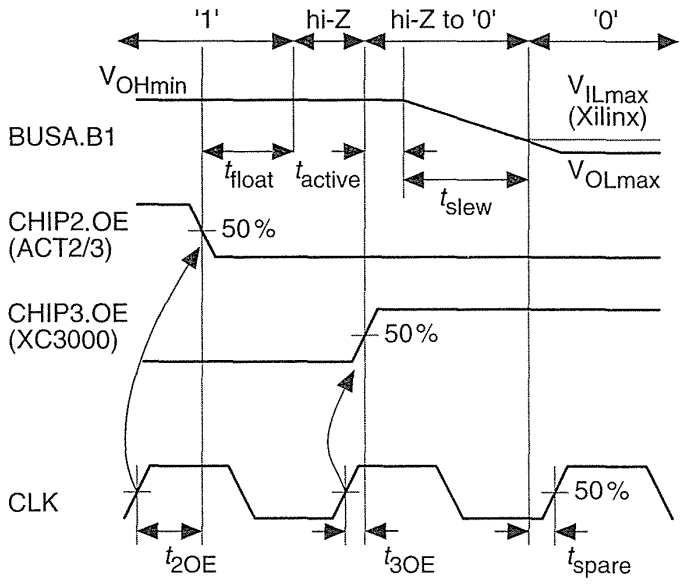


FIGURE 6.6 Three-state bus timing for Figure 6.5. The on-chip delays, t_{2OE} and t_{3OE}, for the logic that generates signals CHIP2.E1 and CHIP3.E1 are derived from the timing models described in Chapter 5 (the minimum values for each chip would be the clock-to-Q delay times).

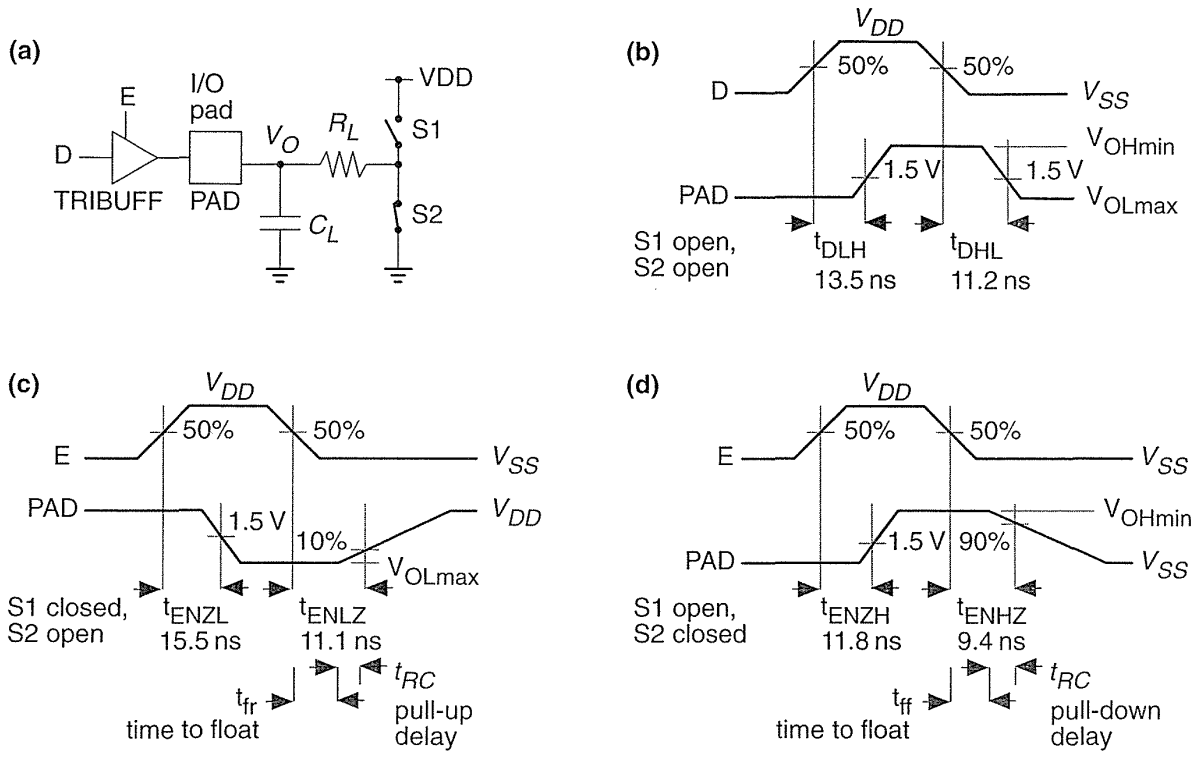


FIGURE 6.7 (a) The test circuit for characterizing the ACT 2 and ACT 3 I/O delay parameters. (b) Output buffer propagation delays from the data input to PAD (output enable, E, is high). (c) Three-state delay with D low. (d) Three-state delay with D high. Delays are shown for ACT 2 'Std' speed grade, worst-case commercial conditions ($R_L = 1$ kΩ, $C_L = 50$ pF, $V_{OHmin} = 2.4$ V, $V_{OLmax} = 0.5$ V). (The Actel three-state buffer is named TRIBUFF, an input buffer INBUF, and the output buffer, OUTBUF.)

measure the buffer delay (measured from the change in the enable signal, E) Actel uses a resistor load ($R_L = 1$ kΩ for ACT 2). The resistor pulls the buffer output high or low depending on whether we are measuring:

- t_{ENZL}, when the output switches from hi-Z to '0'.
- t_{ENLZ}, when the output switches from '0' to hi-Z.
- t_{ENZH}, when the output switches from hi-Z to '1'.
- t_{ENHZ}, when the output switches from '1' to hi-Z.

Other vendors specify the **time to float** a three-state output buffer directly (t_{fr} and t_{ff} in Figure 6.7c and d). This delay time has different names (and definitions): **disable time, time to begin hi-Z**, or **time to turn off**.

Actel does not specify the time to float but, since $R_L C_L = 50$ ns, we know $t_{RC} = -R_L C_L \ln 0.9$ or approximately 5.3 ns. Now we can estimate that

$$t_{fr} = t_{ENLZ} - t_{RC} = 11.1 - 5.3 = 5.8 \text{ ns}, \quad \text{and} \quad t_{ff} = 9.4 - 5.3 = 4.1 \text{ ns},$$

and thus the Actel buffer can float the bus in $t_{float} = 4.1$ ns (Figure 6.6).

The Xilinx FPGA is responsible for the second part of the bus transaction. The time to make the buffer CHIP2.B1 active is t_{active}. Once the buffer is active, the output transistors turn on, conducting a current I_{peak}. The output voltage V_O across the load capacitance, C_{BUS}, will **slew** or change at a steady rate, $dV_O/dt = I_{peak}/C_{BUS}$; thus $t_{slew} = C_{BUS}\Delta V_O / I_{peak}$, where ΔV_O is the change in output voltage.

Vendors do not always provide enough information to calculate t_{active} and t_{slew} separately, but we can usually estimate their sum. Xilinx specifies the time from the three-state input switching to the time the "pad is active and valid" for an XC3000-125 switching with a 50 pF load, to be $t_{active} = t_{TSON} = 11$ ns (fast option), and 27 ns (slew-rate limited option).[3] If we need to drive the bus in less than one clock cycle (30 ns), we will definitely need to use the fast option.

A supplement to the XC3000 timing data specifies the additional fall delay for switching large capacitive loads (above 50 pF) as $R_{fall} = 0.06 \text{ nspF}^{-1}$ (falling) and $R_{rise} = 0.12 \text{ nspF}^{-1}$ (rising) using the fast output option.[4] We can thus estimate that

$$I_{peak} \approx (5 \text{ V}) / (-0.06 \times 10^3 \text{ sF}^{-1}) \approx -84 \text{ mA} \qquad \text{(falling)}$$

and

$$I_{peak} \approx (5 \text{ V}) / (0.12 \times 10^3 \text{ sF}^{-1}) \approx 42 \text{ mA} \qquad \text{(rising)}.$$

Now we can calculate,

$$t_{slew} = R_{fall} (C_{BUS} - 50 \text{ pF}) = (90 \text{ pF} - 50 \text{ pF}) (0.06 \text{ nspF}^{-1}) \qquad \text{or } 2.4 \text{ ns},$$

for a total falling delay of $11 + 2.4 = 13.4$ ns. The rising delay is slower at $11 + (40 \text{ pF})(0.12 \text{ nspF}^{-1})$ or 15.8 ns. This leaves $(30 - 15.8)$ ns, or about 14 ns worst-case, to generate the output enable signal CHIP2.OE (t_{3OE} in Figure 6.6) and still leave time t_{spare} before the bus data is latched on the next clock edge. We can thus probably use a XC3000 part for a 30 MHz bus transceiver, but only if we use the fast slew-rate option.

An aside: Our example looks a little like the PCI bus used on Pentium and PowerPC systems, but the bus transactions are simplified. PCI buses use a **sustained three-state** system (s/t/s). On the PCI bus an s/t/s driver must drive the bus high for at least one clock cycle before letting it float. A new driver may not start driving the bus until a clock edge after the previous driver floats it. After such a **turnaround cycle** a new driver will always find the bus parked high.

[3] 1994 data book, p. 2-159.

[4] Application Note XAPP 024.000, Additional XC3000 Data, 1994 data book p. 8-15.

6.2.1 Supply Bounce

Figure 6.8(a) shows an n-channel transistor, M1, that is part of an output buffer driving an output pad, OUT1; M2 and M3 form an inverter connected to an input pad, IN1; and M4 and M5 are part of another output buffer connected to an output pad, OUT2. As M1 sinks current pulling OUT1 low (V_{o1} in Figure 6.8b), a substantial current I_{OL} may flow in the resistance, R_S, and inductance, L_S, that are between the on-chip GND net and the off-chip, external ground connection.

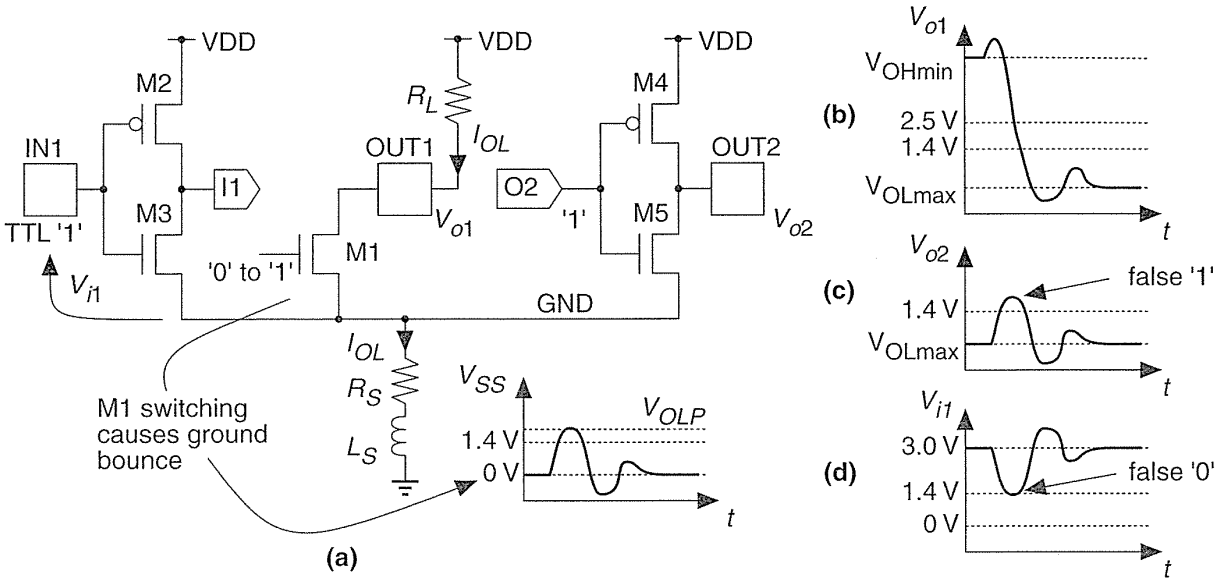


FIGURE 6.8 Supply bounce. (a) As the pull-down device, M1, switches, it causes the GND net (value V_{SS}) to bounce. (b) The supply bounce is dependent on the output slew rate. (c) Ground bounce can cause other output buffers to generate a logic glitch. (d) Bounce can also cause errors on other inputs.

The voltage drop across R_S and L_S causes a spike (or transient) on the GND net, changing the value of V_{SS}, leading to a problem known as **supply bounce**. The situation is illustrated in Figure 6.8(a), with V_{SS} bouncing to a maximum of V_{OLP}. This **ground bounce** causes the voltage at the output, V_{o2}, to bounce also. If the threshold of the gate that OUT2 is driving is a TTL level at 1.4 V, for example, a ground bounce of more than 1.4 V will cause a logic high **glitch** (a momentary transition from one logic level to the opposite logic level and back again).

Ground bounce may also cause problems at chip inputs. Suppose the inverter M2/M3 is set to have a TTL threshold of 1.4 V and the input, IN1, is at a fixed voltage equal to 3 V (a respectable logic high for bipolar TTL). In this case a ground bounce of greater than 1.6 V will cause the input, IN1, to see a logic low instead of a

high and a glitch will be generated on the inverter output, I1. Supply bounce can also occur on the VDD net, but this is usually less severe because the pull-up transistors in an output buffer are usually weaker than the pull-down transistors. The risk of generating a glitch is also greater at the low logic level for TTL-threshold inputs and TTL-level outputs because the low-level noise margins are smaller than the high-level noise margins in TTL.

Sixteen SSOs, with each output driving 150 pF on a bus, can generate a ground bounce of 1.5 V or more. We cannot simulate this problem easily with FPGAs because we are not normally given the characteristics of the output devices. As a rule of thumb we wish to keep ground bounce below 1 V. To help do this we can limit the maximum number of SSOs, and we can limit the number of I/O buffers that share GND and VDD pads.

To further reduce the problem, FPGAs now provide options to limit the current flowing in the output buffers, reducing the slew rate and slowing them down. Some FPGAs also have quiet I/O circuits that sense when the input to an output buffer changes. The quiet I/O then starts to change the output using small transistors; shortly afterwards the large output transistors "drop-in." As the output approaches its final value, the large transistors "kick-out," reducing the supply bounce.

6.2.2 Transmission Lines

Most of the problems with driving large capacitive loads at high speed occur on a bus, and in this case we may have to consider the bus as a transmission line. Figure 6.9(a) shows how a transmission line appears to a driver, D1, and receiver, R1, as a constant impedance, the **characteristic impedance** of the line, $\mathbf{Z_0}$. For a typical PCB trace, Z_0 is between 50 Ω and 100 Ω.

The voltages on a transmission line are determined by the value of the driver source resistance, R_0, and the way that we terminate the end of the transmission line. In Figure 6.9(a) the termination is just the capacitance of the receiver, C_{in}. As the driver switches between 5 V and 0 V, it launches a voltage wave down the line, as shown in Figure 6.9(b). The wave will be $Z_0/(R_0+Z_0)$ times 5 V in magnitude, so that if R_0 is equal to Z_0, the wave will be 2.5 V.

Notice that it does not matter what is at the far end of the line. The bus driver sees only Z_0 and not C_{in}. Imagine the transmission line as a tunnel; all the bus driver can see at the entrance is a little way into the tunnel—it could be 500 m or 5 km long. To find out, we have to go with the wave to the end, turn around, come back, and tell the bus driver. The final result will be the same whether the transmission line is there or not, but with a transmission line it takes a little longer for the voltages and currents to settle down. This is rather like the difference between having a conversation by telephone or by post.

The propagation delay (or time of flight), t_f, for a typical PCB trace is approximately 1 ns for every 15 cm of trace (the signal velocity is about one-half the speed of light). A voltage wave launched on a transmission line takes a time t_f to get to the end of the line, where it finds the load capacitance, C_{in}. Since no current can flow at

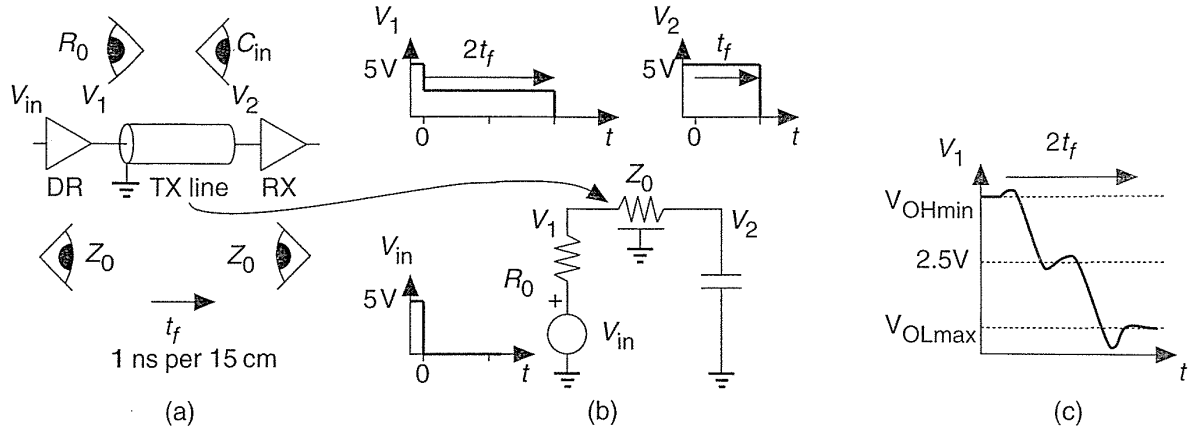


FIGURE 6.9 Transmission lines. (a) A printed-circuit board (PCB) trace is a transmission (TX) line. (b) A driver launches an incident wave, which is reflected at the end of the line. (c) A connection starts to look like a transmission line when the signal rise time is about equal to twice the line delay ($2t_f$).

this point, there must be a reflection that exactly cancels the incident wave so that the voltage at the input to the receiver, at V_2, becomes exactly zero at time t_f. The reflected wave travels back down the line and finally causes the voltage at the output of the driver, at V_1, to be exactly zero at time $2t_f$. In practice the nonidealities of the driver and the line cause the waves to have finite rise times. We start to see transmission line behavior if the rise time of the driver is less than $2t_f$, as shown in Figure 6.9(c).

There are several ways to terminate a transmission line. Figure 6.10 illustrates the following methods:

- *Open-circuit or capacitive termination.* The bus termination is the input capacitance of the receivers (usually less than 20 pF). The PCI bus uses this method.
- *Parallel resistive termination.* This requires substantial DC current (5 V / 100 Ω = 50 mA for a 100 Ω line). It is used by bipolar logic, for example emitter-coupled logic (ECL), where we typically do not care how much power we use.
- *Thévenin termination.* Connecting 300 Ω in parallel with 150 Ω across a 5 V supply is equivalent to a 100 Ω termination connected to a 1.6 V source. This reduces the DC current drain on the drivers but adds a resistance directly across the supply.
- *Series termination at the source.* Adding a resistor in series with the driver so that the sum of the driver source resistance (which is usually 50 Ω or even less) and the termination resistor matches the line impedance (usually around

100 Ω). The disadvantage is that it generates reflections that may be close to the switching threshold.

- *Parallel termination with a voltage bias.* This is awkward because it requires a third supply and is normally used only for a specialized high-speed bus.
- *Parallel termination with a series capacitance.* This removes the requirement for DC current but introduces other problems.

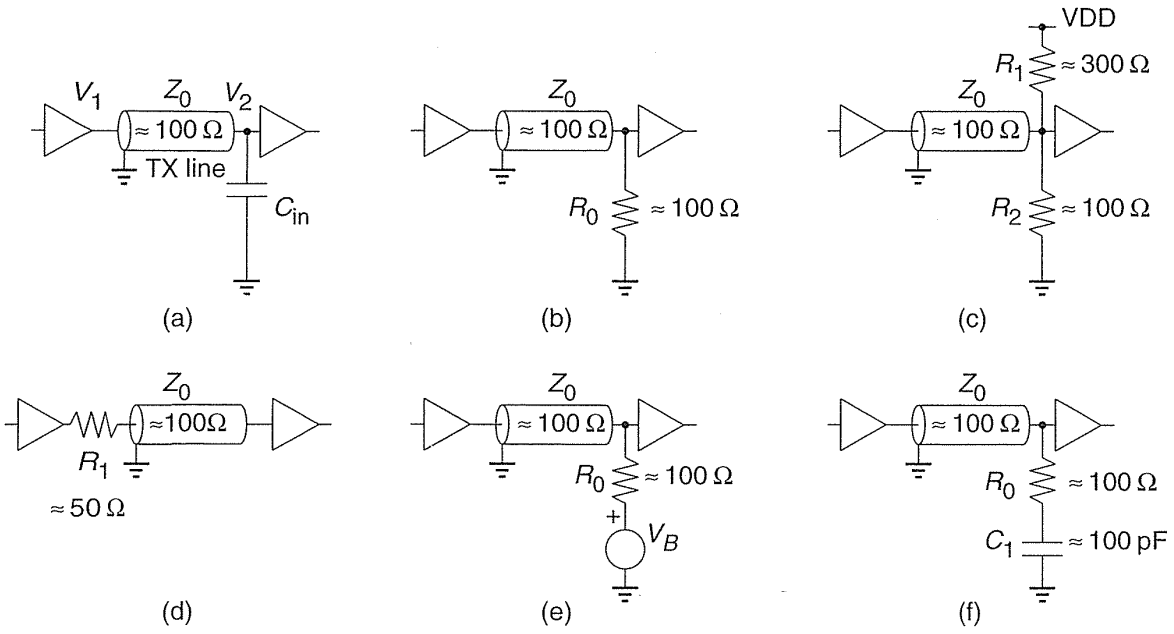


FIGURE 6.10 Transmission line termination. (a) Open-circuit or capacitive termination. (b) Parallel resistive termination. (c) Thévenin termination. (d) Series termination at the source. (e) Parallel termination using a voltage bias. (f) Parallel termination with a series capacitor.

Until recently most bus protocols required strong bipolar or BiCMOS output buffers capable of driving all the way between logic levels. The PCI standard uses weaker CMOS drivers that rely on reflection from the end of the bus to allow the intermediate receivers to see the full logic value. Many FPGA vendors now offer complete PCI functions that the ASIC designer can "drop in" to an FPGA [PCI, 1995].

An alternative to using a transmission line that operates across the full swing of the supply voltage is to use current-mode signaling or differential signals with low-voltage swings. These and other techniques are used in specialized bus structures and in high-speed DRAM. Examples are Rambus, and **Gunning transistor logic** (**GTL**). These are analog rather than digital circuits, but ASIC methods apply if the interface circuits are available as cells, hiding some of the complexity from the

designer. For example, Rambus offers a **Rambus access cell** (**RAC**) for standard-cell design (but not yet for an FPGA). Directions to more information on these topics are in the bibliography at the end of this chapter.

6.3 DC Input

Suppose we have a pushbutton switch connected to the input of an FPGA as shown in Figure 6.11(a). Most FPGA input pads are directly connected to a buffer. We need to ensure that the input of this buffer never floats to a voltage between valid logic levels (which could cause both *n*-channel and *p*-channel transistors in the buffer to turn on, leading to oscillation or excessive power dissipation) and so we use the optional pull-up resistor (usually about 100 kΩ) that is available on many FPGAs (we could also connect a 1 kΩ pull-up or pull-down resistor externally).

Contacts may bounce as a switch is operated (Figure 6.11b). In the case of a Xilinx XC4000 the effective pull-up resistance is 5–50 kΩ (since the specified pull-up current is between 0.2 and 2.0 mA) and forms an *RC* time constant with the parasitic capacitance of the input pad and the external circuit. This time constant (typically hundreds of nanoseconds) will normally be much less than the time over which the contacts bounce (typically many milliseconds). The buffer output may thus be a series of pulses extending for several milliseconds. It is up to you to deal with this in your logic. For example, you may want to **debounce** the waveform in Figure 6.11(b) using an SR flip-flop.

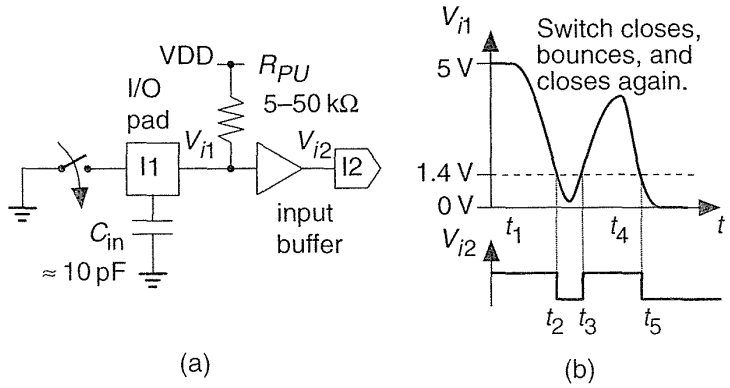


FIGURE 6.11 A switch input. (a) A pushbutton switch connected to an input buffer with a pull-up resistor. (b) As the switch bounces several pulses may be generated.

A bouncing switch may create a noisy waveform in the time domain, we may also have noise in the voltage level of our input signal. The **Schmitt-trigger** inverter in Figure 6.12(a) has a lower switching threshold of 2 V and an upper switching threshold of 3 V. The difference between these thresholds is the **hysteresis**, equal to 1 V in this case. If we apply the noisy waveform shown in Figure 6.12(b) to an

inverter with no hysteresis, there will be a glitch at the output, as shown in Figure 6.12(c). As long as the noise on the waveform does not exceed the hysteresis, the Schmitt-trigger inverter will produce the glitch-free output of Figure 6.12(d).

Most FPGA input buffers have a small hysteresis (the 200 mV that Xilinx uses is a typical figure) centered around 1.4 V (for compatibility with TTL), as shown in Figure 6.12(e). Notice that the drawing inside the symbol for a Schmitt trigger looks like the transfer characteristic for a buffer, but is backward for an inverter. Hysteresis in the input buffer also helps prevent oscillation and noise problems with inputs that have slow rise times, though most FPGA manufacturers still have a restriction that input signals must have a rise time faster than several hundred nanoseconds.

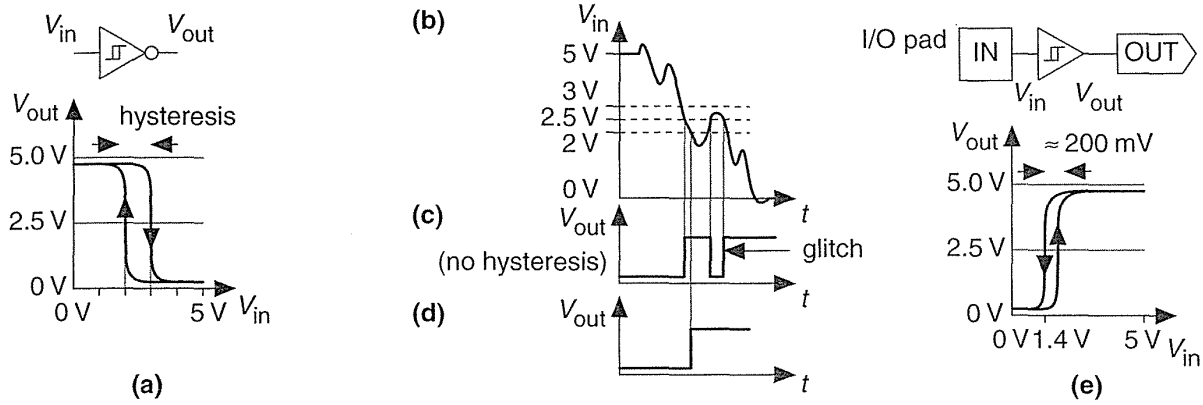


FIGURE 6.12 DC input. (a) A Schmitt-trigger inverter. (b) A noisy input signal. (c) Output from an inverter with no hysteresis. (d) Hysteresis helps prevent glitches. (e) A typical FPGA input buffer with a hysteresis of 200 mV centered around a threshold of 1.4 V.

6.3.1 Noise Margins

Figure 6.13(a) and (b) show the worst-case DC transfer characteristics of a CMOS inverter. Figure 6.13(a) shows a situation in which the process and device sizes create the lowest possible switching threshold. We define the maximum voltage that will be recognized as a '0' as the point at which the gain (V_{out}/V_{in}) of the inverter is −1. This point is $V_{ILmax} = 1V$ in the example shown in Figure 6.13(a). This means that any input voltage that is lower than 1V will definitely be recognized as a '0', even with the most unfavorable inverter characteristics. At the other worst-case extreme we define the minimum voltage that will be recognized as a '1' as $V_{IHmin} = 3.5V$ (for the example in Figure 6.13b).

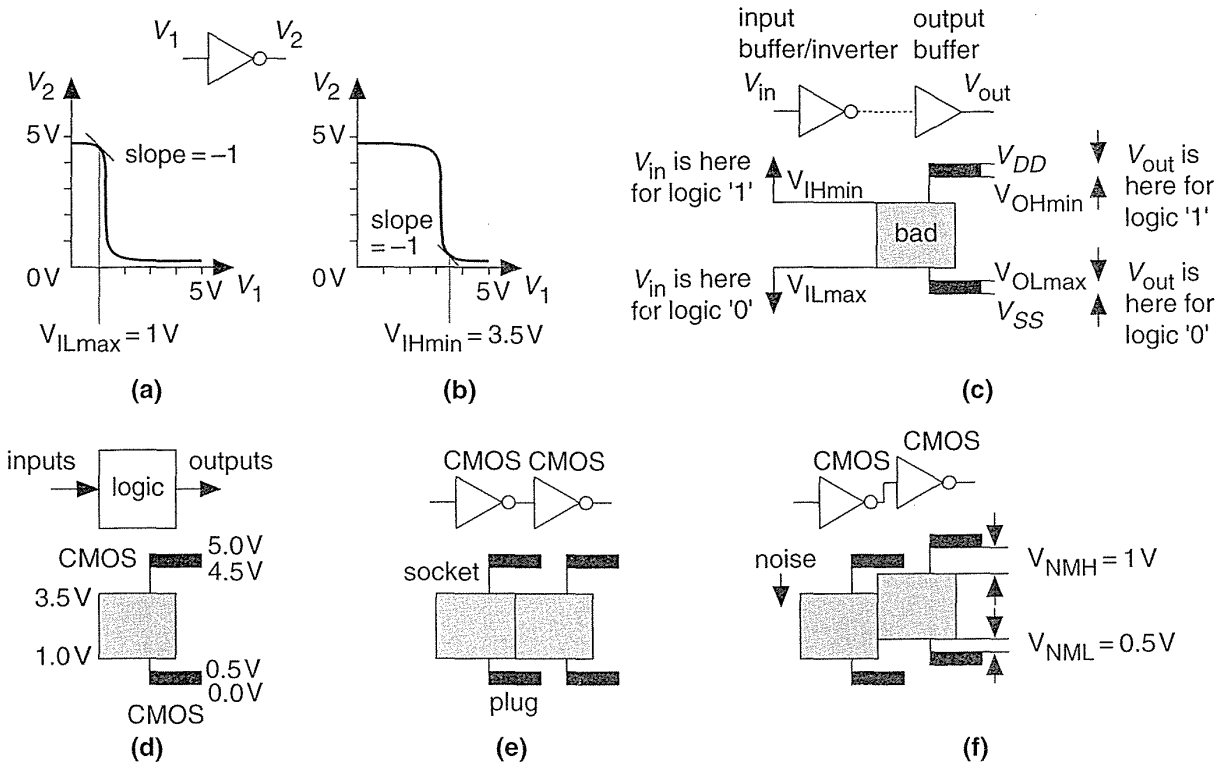


FIGURE 6.13 Noise margins. (a) Transfer characteristics of a CMOS inverter with the lowest switching threshold. (b) The highest switching threshold. (c) A graphical representation of CMOS logic thresholds. (d) Logic thresholds at the inputs and outputs of a logic gate or an ASIC. (e) The switching thresholds viewed as a plug and socket. (f) CMOS plugs fit CMOS sockets and the clearances are the noise margins.

Figure 6.13(c) depicts the following relationships between the various voltage levels at the inputs and outputs of a logic gate:

- A logic '1' output must be between V_{OHmin} and V_{DD}.
- A logic '0' output must be between V_{SS} and V_{OLmax}.
- A logic '1' input must be above the **high-level input voltage**, V_{IHmin}.
- A logic '0' input must be below the **low-level input voltage**, V_{ILmax}.
- Clamp diodes prevent an input exceeding V_{DD} or going lower than V_{SS}.

The voltages, V_{OHmin}, V_{OLmax}, V_{IHmin}, and V_{ILmax}, are the **logic thresholds** for a technology. A logic signal outside the areas bounded by these logic thresholds is "bad"—an unrecognizable logic level in an electronic no-man's land. Figure 6.13(d)

shows typical logic thresholds for a CMOS-compatible FPGA. The V_{IHmin} and V_{ILmax} logic thresholds come from measurements in Figure 6.13(a) and (b) and V_{OHmin} and V_{OLmax} come from the measurements shown in Figure 6.2(c).

Figure 6.13(d) illustrates how logic thresholds form a plug and socket for any gate, group of gates, or even a chip. If a plug fits a socket, we can connect the two components together and they will have compatible logic levels. For example, Figure 6.13(e) shows that we can connect two CMOS gates or chips together.

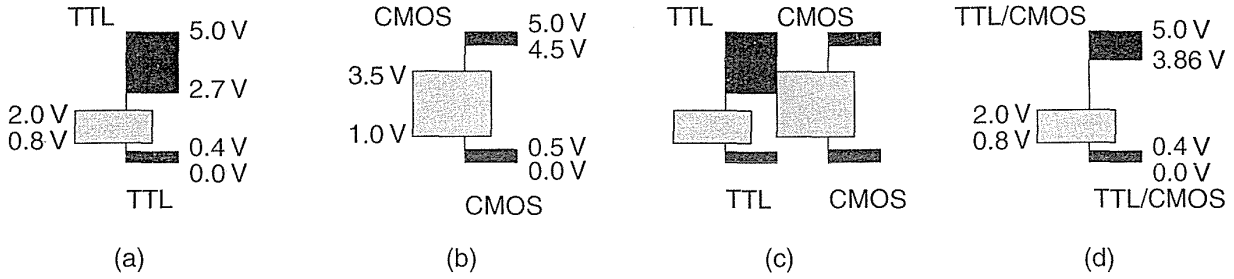


FIGURE 6.14 TTL and CMOS logic thresholds. (a) TTL logic thresholds. (b) Typical CMOS logic thresholds. (c) A TTL plug will not fit in a CMOS socket. (d) Raising V_{OHmin} solves the problem.

Figure 6.13(f) shows that we can even add some noise that shifts the input levels and the plug will still fit into the socket. In fact, we can shift the plug down by exactly $V_{OHmin} - V_{IHmin}$ $(4.5 - 3.5 = 1\ V)$ and still maintain a valid '1'. We can shift the plug up by $V_{ILmax} - V_{OLmax}$ $(1.0 - 0.5 = 0.5\ V)$ and still maintain a valid '0'. These clearances between plug and socket are the **noise margins**:

$$V_{NMH} = V_{OHmin} - V_{IHmin} \quad \text{and} \quad V_{NML} = V_{ILmax} - V_{OLmax}. \tag{6.1}$$

For two logic systems to be compatible, the plug must fit the socket. This requires both the **high-level noise margin** (V_{NMH}) and the **low-level noise margin** (V_{NML}) to be positive. We also want both noise margins to be as large as possible to give us maximum immunity from noise and other problems at an interface.

Figure 6.14(a) and (b) show the logic thresholds for TTL together with typical CMOS logic thresholds. Figure 6.14(c) shows the problem with trying to plug a TTL chip into a CMOS input level—the lowest permissible TTL output level, $V_{OHmin} = 2.7$ V, is too low to be recognized as a logic '1' by the CMOS input. This is fixed by most FPGA manufacturers by raising V_{OHmin} to around 3.8–4.0V (Figure 6.14d). Table 6.1 lists the logic thresholds for several FPGAs.

6.3.2 Mixed-Voltage Systems

To reduce power consumption and allow CMOS logic to be scaled below 0.5 μm it is necessary to reduce the power supply voltage below 5 V. The JEDEC 8 [JEDEC

I/O] series of standards sets the next lower supply voltage as 3.3 ± 0.3 V. Figure 6.15(a) and (b) shows that the 3 V CMOS I/O logic-thresholds can be made compatible with 5 V systems. Some FPGAs can operate on both 3 V and 5 V supplies, typically using one voltage for internal (or core) logic, V_{DDint} and another for the I/O circuits, $V_{DDI/O}$ (Figure 6.15c).

TABLE 6.1 FPGA logic thresholds.

	I/O options		Input levels		Output levels (high current)				Output levels (low current)			
	Input	Output	V_{IH} (min)	V_{IL} (max)	V_{OH} (min)	I_{OH} (max)	V_{OL} (max)	I_{OL} (max)	V_{OH} (min)	I_{OH} (max)	V_{OL} (max)	I_{OL} (max)
XC3000[1]	TTL		2.0	0.8	3.86	–4.0	0.40	4.0				
	CMOS		3.85[2]	0.9[3]	3.86	–4.0	0.40	4.0				
XC3000L			2.0	0.8	2.40	–4.0	0.40	4.0	2.80[4]	–0.1	0.2	0.1
XC4000[5]			2.0	0.8	2.40	–4.0	0.40	12.0				
XC4000H[6]	TTL	TTL	2.0	0.8	2.40	–4.0	0.50	24.0				
	CMOS	CMOS	3.85[2]	0.9[3]	4.00[7]	–1.0	0.50	24.0				
XC8100[8]	TTL	R	2.0	0.8	3.86	–4.0	0.50	24.0				
	CMOS	C	3.85[2]	0.9[3]	3.86	–4.0	0.40	4.0				
ACT 2/3			2.0	0.8	2.4	–8.0	0.50	12.0	3.84	–4.0	0.33	6.0
FLEX10k[9]		3V/5V	2.0	0.8	2.4	–4.0	0.45	12.0				

[1]XC2000, XC3000/A have identical thresholds. XC3100/A thresholds are identical to XC3000 except for ±8mA source–sink current. XC5200 thresholds are identical to XC3100A.
[2]Defined as $0.7\,V_{DD}$, calculated with $V_{DDmax} = 5.5$ V.
[3]Defined as $0.2\,V_{DD}$, calculated with $V_{DDmin} = 4.5$ V.
[4]Defined as $V_{DD} - 0.2$ V, calculated with $V_{DDmin} = 3.0$ V.
[5]XC4000, XC4000A have identical I/O thresholds except XC4000A has –24 mA sink current.
[6]XC4000H/E have identical I/O thresholds except XC4000E has –12 mA sink current. Options are independent.
[7]Defined as $V_{DD} - 0.5$ V, calculated with $V_{DDmin} = 4.5$ V.
[8]Input and output options are independent.
[9]MAX 9000 has identical thresholds to FLEX 10k.
Note: All voltages in volts, all currents in milliamperes.

There is one problem when we mix 3 V and 5 V supplies that is shown in Figure 6.15(d). If we apply a voltage to a chip input that exceeds the power supply of a chip, it is possible to power a chip inadvertently through the clamp diodes. In the worst case this may cause a voltage as high as 2.5 V (= 5.5 V – 3.0 V) to appear across the clamp diode, which will cause a very large current (several hundred milliamperes) to flow. One way to prevent damage is to include a series resistor between

the chips, typically around 1 kΩ. This solution does not work for all chips in all systems. A difficult problem in ASIC I/O design is constructing **5 V-tolerant I/O**. Most solutions may never surface (there is little point in patenting a solution to a problem that will go away before the patent is granted).

Similar problems can arise in several other situations:

- when you connect two ASICs with "different" 5 V supplies;
- when you power down one ASIC in a system but not another, or one ASIC powers down faster than another;
- on system power-up or system reset.

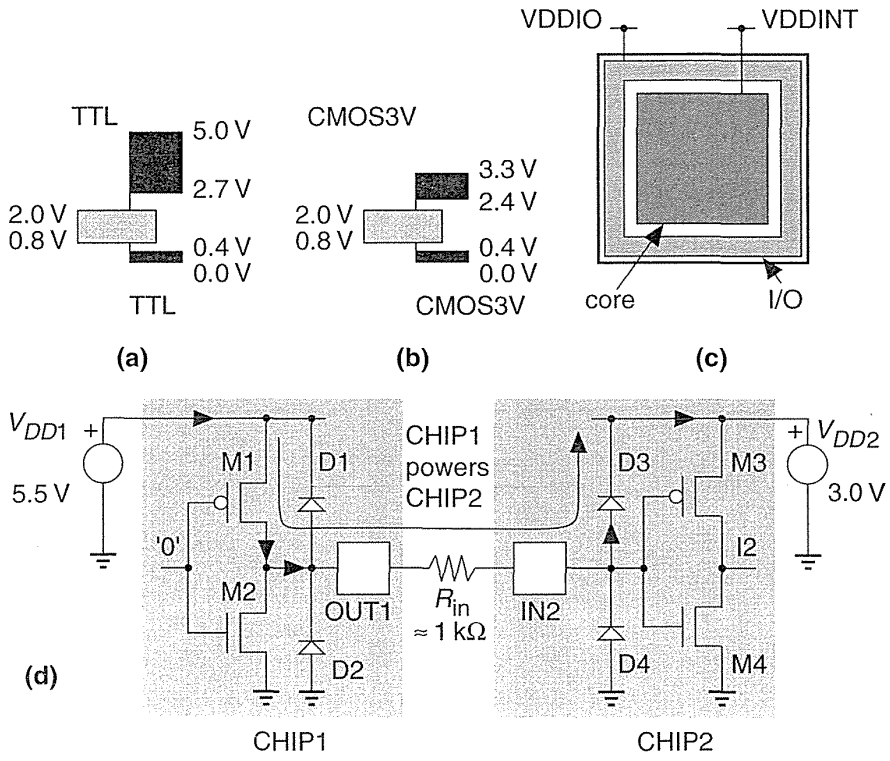


FIGURE 6.15 Mixed-voltage systems. (a) TTL levels. (b) Low-voltage CMOS levels. (c) A mixed-voltage ASIC. (d) A problem when connecting two chips with different supply voltages—caused by the input clamp diodes.

6.4 AC Input

Suppose we wish to connect an input bus containing sampled data from an **analog-to-digital converter** (**A/D**) that is running at a clock frequency of 100 kHz to an FPGA that is running from a system clock on a bus at 10 MHz (a NuBus). We are to perform some filtering and calculations on the sampled data before placing it on the

NuBus. We cannot just connect the A/D output bus to our FPGA, because we have no idea when the A/D data will change. Even though the A/D data rate (a sample every 10 µs or every 100 NuBus clock cycles) is much lower than the NuBus clock, if the data happens to arrive just before we are due to place an output on the NuBus, we have no time to perform any calculations. Instead we want to register the data at the input to give us a whole NuBus clock cycle (100 ns) to perform the calculations. We know that we should have the A/D data at the flip-flop input for at least the flip-flop setup time before the NuBus clock edge. Unfortunately there is no way to guarantee this; the A/D converter clock and the NuBus clock are completely independent. Thus it is entirely possible that every now and again the A/D data will change just before the NuBus clock edge.

6.4.1 Metastability

If we change the data input to a flip-flop (or a latch) too close to the clock edge (called a setup or hold-time **violation**), we run into a problem called **metastability**, illustrated in Figure 6.16. In this situation the flip-flop cannot decide whether its out-

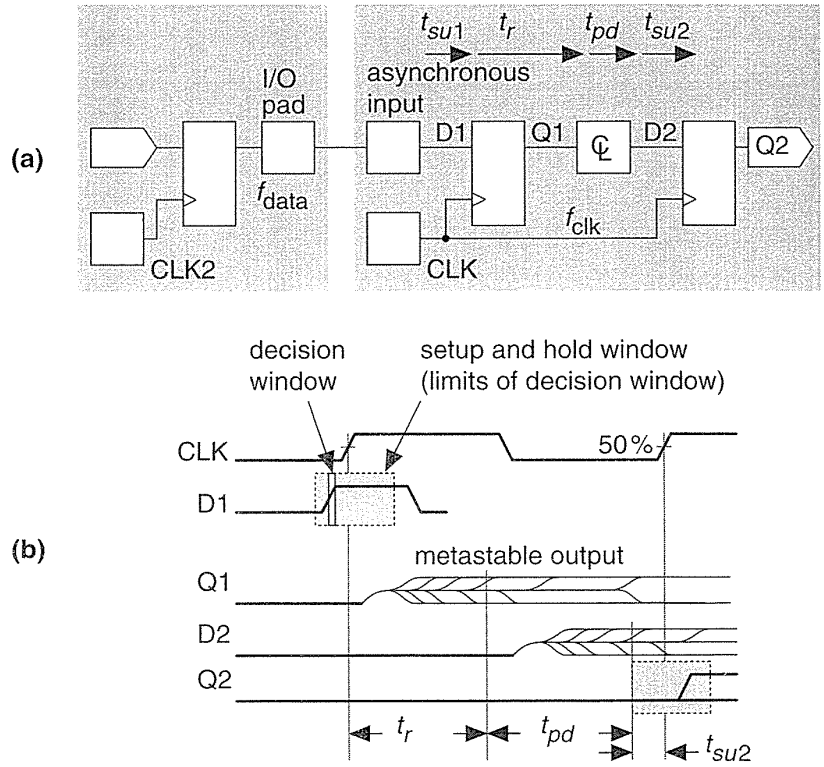


FIGURE 6.16 Metastability. (a) Data coming from one system is an asynchronous input to another. (b) A flip-flop has a very narrow decision window bounded by the setup and hold times. If the data input changes inside this decision window, the output may be metastable—neither '1' or '0'.

put should be a '1' or a '0' for a long time. If the flip-flop makes a decision, at a time t_r after the clock edge, as to whether its output is a '1' or a '0', there is a small, but finite, probability that the flip-flop will decide the output is a '1' when it should have been a '0' or vice versa. This situation, called an **upset**, can happen when the data is coming from the outside world and the flip-flop can't determine when it will arrive; this is an **asynchronous signal**, because it is not synchronized to the chip clock.

Experimentally we find that the **probability of upset**, p, is

$$p = T_0 \exp \frac{-t_r}{\tau_c}, \tag{6.2}$$

(per data event, per clock edge, in one second, with units $Hz^{-1} \cdot Hz^{-1} \cdot s^{-1}$) where t_r is the time a **sampler** (flip-flop or latch) has to **resolve** the sampler output; $\mathbf{T_0}$ and τ_c are constants of the sampler circuit design. Let us see how serious this problem is in practice. If $t_r = 5$ ns, $\tau_c = 0.1$ ns, and $T_0 = 0.1$ s, Eq. 6.2 gives the upset probability as

$$p = 0.1 \exp\left(\frac{-(5 \times 10^{-9})}{(0.1 \times 10^{-9})} \right) = 2 \times 10^{-23} \text{ s}, \tag{6.3}$$

which is very small, but the data and clock may be running at several MHz, causing the sampler plenty of opportunities for upset.

The **mean time between upsets** (**MTBU**, similar to MTBF—mean time between failures) is

$$\text{MTBU} = \frac{1}{p f_{\text{clock}} f_{\text{data}}} = \frac{\exp \frac{t_r}{\tau_c}}{T_0 f_{\text{clock}} f_{\text{data}}}, \tag{6.4}$$

where f_{clock} is the clock frequency and f_{data} is the data frequency.

If $t_r = 5$ ns, $\tau_c = 0.1$ ns, $T_0 = 0.1$ s (as in the previous example), $f_{\text{clock}} = 100$ MHz, and $f_{\text{data}} = 1$ MHz, then

$$\text{MTBU} = \frac{\exp\left(\frac{5 \times 10^{-9}}{0.1 \times 10^{-9}} \right)}{(100 \times 10^6)\ (1 \times 10^6)\ (0.1)} = 5.2 \times 10^8 \text{ seconds}, \tag{6.5}$$

or about 16 years (10^8 seconds is three years, and a day is 10^5 seconds). An MTBU of 16 years may seem safe, but suppose we have a 64-bit input bus using 64 flip-flops. If each flip-flop has an MTBU of 16 years, our system-level MTBF is three months. If

we ship 1000 systems we would have an average of 10 systems failing every day. What can we do?

The parameter τ_c is the inverse of the **gain–bandwidth product**, ***GB***, of the sampler at the instant of sampling. It is a constant that is independent of whether we are sampling a positive or negative data edge. It may be determined by a small-signal analysis of the sampler at the sampling instant or by measurement. It cannot be determined by simulating the transient response of the flip-flop to a metastable event since the gain and bandwidth both normally change as a function of time. We cannot change τ_c.

The parameter T_0 (units of time) is a function of the process technology and the circuit design. It may be different for sampling a positive or negative data edge, but normally only one value of T_0 is given. Attempts have been made to calculate T_0 and to relate it to a physical quantity. The best method is by measurement or simulation of metastable events. We cannot change T_0.

Given a good flip-flop or latch design, τ_c and T_0 should be similar for comparable CMOS processes (so, for example, all 0.5 μm processes should have approximately the same τ_c and T_0). The only parameter we can change when using a flip-flop or latch from a cell library is t_r, and we should allow as much resolution time as

TABLE 6.2 Metastability parameters for FPGA flip-flops. These figures are not guaranteed by the vendors.

FPGA	T_0/s	τ_c/s
Actel ACT 1	1.0E–09	2.17E–10
Xilinx XC3020-70	1.5E–10	2.71E–10
QuickLogic QL12x16-0	2.94E–11	2.91E–10
QuickLogic QL12x16-1	8.38E–11	2.09E–10
QuickLogic QL12x16-2	1.23E–10	1.85E–10
Xilinx XC8100	2.15E-12	4.65E–10
Xilinx XC8100 synchronizer	1.59E-17	2.07E–10
Altera MAX 7000	2.98E–17	2.00E–10
Altera FLEX 8000	1.01E–13	7.89E–11

Sources: Actel April 1992 data book, p. 5-1, gives C1 = $T_0 = 10^{-9}\text{Hz}^{-1}$, C2 = $1/\tau_c = 4.6052\ \text{ns}^{-1}$, or τ_c = 2.17E–10 s and T_0 = 1.0E–09 s. Xilinx gives K1 = T_0 = 1.5E–10 s and K2 = $1/\tau_c = 3.69\text{E}9\ \text{s}^{-1}$, τ_c = 2.71E–10 s, for the XC3020-70 (p. 8-20 of 1994 data book). QuickLogic pASIC 1 QL12X16: τ_c = 0.2 ns to 0.3 ns, T_0 = 0.3E–10 s to 1.2E–10 s (1994 data book, p. 5-25, Fig. 2). Xilinx XC8100 data, τ_c = 4.65E–10 s and T_0 = 2.15E–12 s, is from October 1995 (v. 1.0) data sheet, Fig.17 (the XC8100 was discontinued in August 1996). Altera 1995 data book p. 437, Table 1.

we can after the output of a *latch* before the signal is clocked again. If we use a *flip-flop* constructed from two latches in series (a master–slave design), then we are sampling the data twice. The resolution time for the first sample t_r is fixed, it is half the clock cycle (if the clock is high and low for equal times—we say the clock has a 50 percent **duty cycle**, or equal **mark–space ratio**). Using such a flip-flop we need to allow as much time as we can before we clock the second sample by connecting two flip-flops in series, without any combinational logic between them, if possible. If you are really in trouble, the next step is to divide the clock so you can extend the resolution time even further.

Table 6.2 shows flip-flop metastability parameters and Figure 6.17 graphs the metastability data for $f_{clock} = 10$ MHz and $f_{data} = 1$ MHz. From this graph we can see the enormous variation in MTBF caused by small variations in τ_c. For example, in the QuickLogic pASIC 1 series the range of T_0 from 0.3 to 1.2×10^{-10}s is 4:1, but it is the range of $\tau_c = 0.2 - 0.3$ ns (a variation of only 1:1.5) that is responsible for the enormous variation in MTBF (nearly four orders of magnitude at $t_r = 5$ ns). The variation in τ_c is caused by the variation in *GB* between the QuickLogic speed grades. Variation in the other vendors' parts will be similar, but most vendors do not show this information. To be safe, build a large safety margin for MTBF into any design—it is not unreasonable to use a margin of four orders of magnitude.

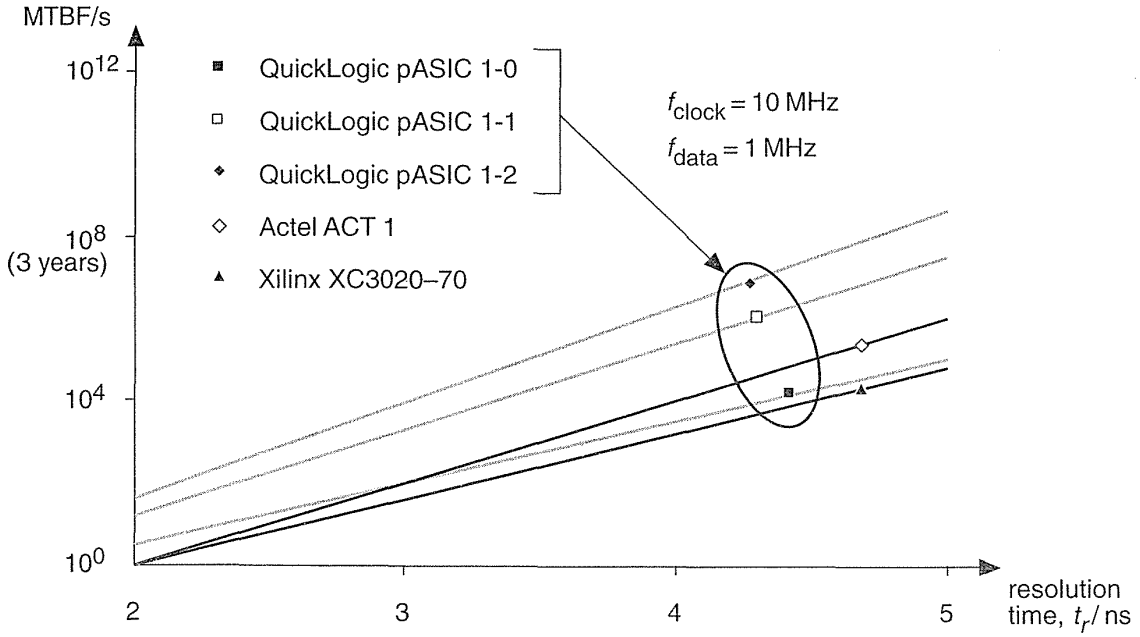


FIGURE 6.17 Mean time between failures (MTBF) as a function of resolution time. The data is from FPGA vendors' data books for a single flip-flop with clock frequency of 10 MHz and a data input frequency of 1 MHz (see Table 6.2).

Some cell libraries include a **synchronizer**, built from two flip-flops in cascade, that greatly reduces the effective values of τ_c and T_0 over a single flip-flop. The penalty is an extra clock cycle of latency.

To compare discrete TTL parts with ASIC flip-flops, the 74AS4374 TTL **metastable-hardened dual flip-flops**, from TI, have $\tau_c = 0.42$ ns and $T_0 = 4$ ns. The parameter T_0 ranges from about 10 s for the 74LS74 (a regular flip-flop) to 4 ns for the 74AS4374 (over nine orders of magnitude different); τ_c only varies from 0.42 ns (74AS374) to 1.3 ns (74LS74), but this small variation in τ_c is just as important.

6.5 Clock Input

When we bring the clock signal onto a chip, we may need to adjust the logic level (clock signals are often driven by TTL drivers with a high current output capability) and then we need to distribute the clock signal around the chip as it is needed. FPGAs normally provide special clock buffers and clock networks. We need to minimize the clock delay (or latency), but we also need to minimize the clock skew.

6.5.1 Registered Inputs

Some FPGAs provide a flip-flop or latch that you can use as part of the I/O circuit (registered I/O). For other FPGAs you have to use a flip-flop or latch using the basic logic cell in the core. In either case the important parameter is the input setup time. We can measure the setup with respect to the clock signal at the flip-flop or the clock signal at the clock input pad. The difference between these two parameters is the clock delay.

Figure 6.18 shows part of the I/O timing model for a Xilinx XC40005-6.[5]

- t_{PICK} is the fixed setup time for a flip-flop relative to the flip-flop clock.
- t_{skew} is the variable **clock skew**, the signed delay between two clock edges.
- t_{PG} is the variable clock delay or **latency**.

To calculate the flip-flop setup time ($t_{PSUFmin}$) relative to the clock pad (which is the parameter system designers need to know), we subtract the clock delay, so that

$$t_{PSUF} = t_{PICK} - t_{PG}. \tag{6.6}$$

The problem is that we cannot easily calculate t_{PG}, since it depends on the clock distribution scheme and where the flip-flop is on the chip. Instead Xilinx specifies

[5]The Xilinx XC4005-6 timing parameters are from the 1994 data book p. 2-50 to p. 2-53.

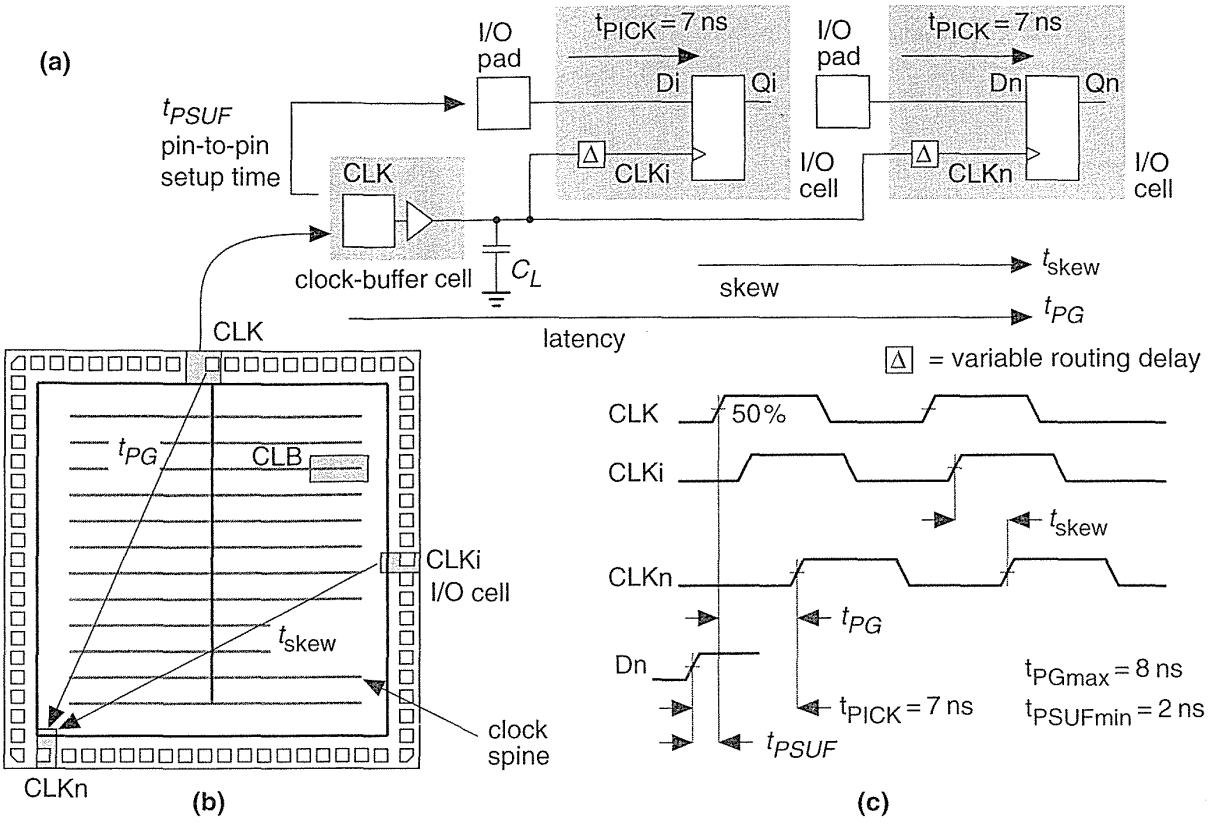


FIGURE 6.18 Clock input. (a) Timing model with values for a Xilinx XC4005-6. (b) A simplified view of clock distribution. (c) Timing diagram. Xilinx eliminates the variable internal delay t_{PG}, by specifying a pin-to-pin setup time, $t_{PSUFmin} = 2$ ns.

$t_{PSUFmin}$ directly, measured from the data pad to the clock pad; this time is called a **pin-to-pin timing parameter**. Notice $t_{PSUFmin} = 2 \text{ ns} \neq t_{PICK} - t_{PGmax} = -1$ ns.

Figure 6.19 shows that the hold time for a XC4005-6 flip-flop (t_{CKI}) with respect to the flip-flop clock is zero. However, the pin-to-pin hold time including the clock delay is $t_{PHF} = 5.5$ ns. We can remove this inconvenient hold-time restriction by delaying the input signal. Including a programmable delay allows Xilinx to guarantee the pin-to-pin hold time (t_{PH}) as zero. The penalty is an increase in the pin-to-pin setup time (t_{PSU}) to 21 ns (from 2 ns) for the XC4005-6, for example.

We also have to account for clock delay when we register an output. Figure 6.20 shows the timing model diagram for the clock-to-output delay.

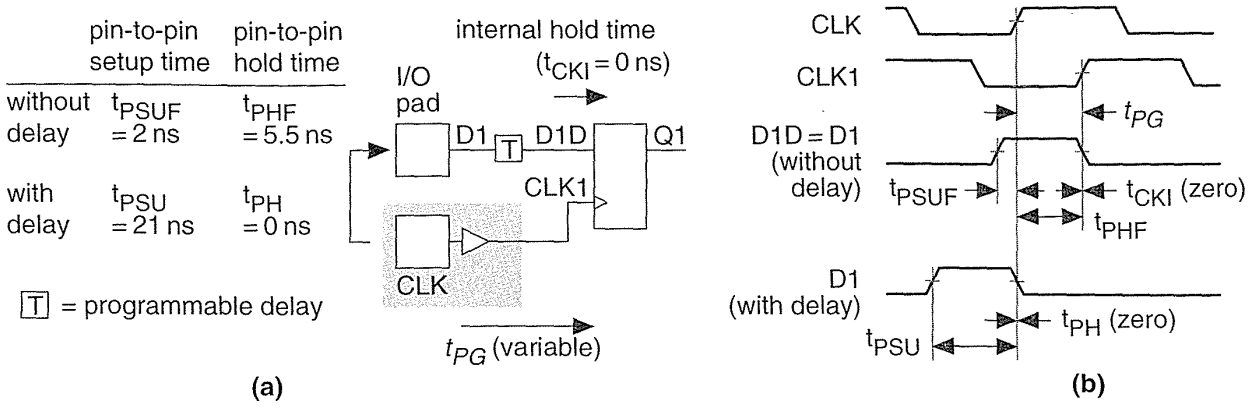


FIGURE 6.19 Programmable input delay. (a) Pin-to-pin timing model with values from an XC4005-6. (b) Timing diagrams with and without programmable delay.

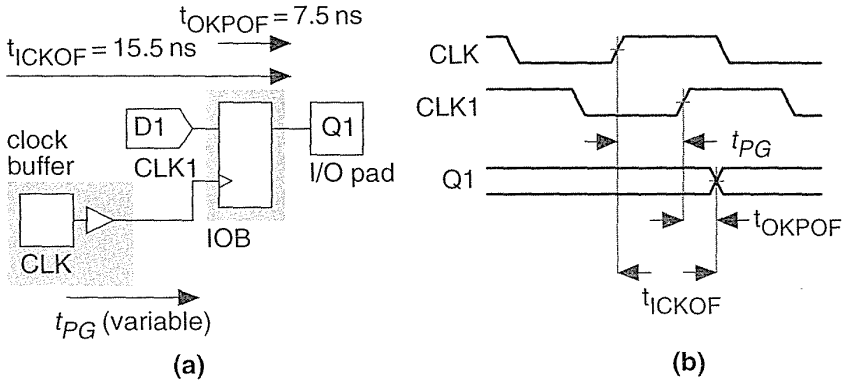


FIGURE 6.20 Registered output. (a) Timing model with values for an XC4005-6 programmed with the fast slew-rate option. (b) Timing diagram.

6.6 Power Input

The last item that we need to bring onto an FPGA is the power. We may need multiple VDD and GND power pads to reduce supply bounce or separate VDD pads for mixed-voltage supplies. We may also need to provide power for on-chip programming (in the case of antifuse or EPROM programming technology). The package type and number of pins will determine the number of power pins, which, in turn, affects the number of SSOs you can have in a design.

6.6.1 Power Dissipation

As a general rule a plastic package can dissipate about 1 W, and more expensive ceramic packages can dissipate up to about 2 W. Table 6.3 shows the thermal characteristics of common packages. In a high-speed (high-power) design the ASIC power

TABLE 6.3 Thermal characteristics of ASIC packages.

Package[1]	Pin count	Max. power P_{max}/W	θ_{JA}/°CW^{-1} (still air)[2,3]	θ_{JA}/°CW^{-1} (still air)[4]
CPGA	84		33	32–38
CPGA	100		35	
CPGA	132		30	
CPGA	175		25	16
CPGA	207		22	
CPGA	257		15	
CQFP	84		40	
CQFP	172		25	
PQFP	100	1.0	55	56–75
PQFP	160	1.75	33	30–33
PQFP	208	2.0	33	27-32
VQFP	80		68	
PLCC	44		52	44
PLCC	68		45	28–35
PLCC	84	1.5	44	
PPGA	132			33–34

[1]CPGA = ceramic pin-grid array; CQFP = ceramic quad flatpack; PQFP = plastic quad flatpack; VQFP = very thin quad flatpack; PLCC = plastic leaded chip carrier; PPGA = plastic pin-grid array.
[2] θ_{JA} varies with die size.
[3]Data from Actel 1994 data book p. 1-9, p. 1-45, and p. 1-94.
[4]Data from Xilinx 1994 data book p. 4-26 and p. 4-27.

consumption may dictate your choice of packages. Actel provides a formula for calculating typical dynamic chip power consumption of their FPGAs. The formula for the ACT 2 and ACT 3 FPGAs are complex; therefore we shall use the simpler formula for the ACT 1 FPGAs as an example[6]:

[6]1994 data book, p.1-9

$$\text{Total chip power} = 0.2\,(N \times F1) + 0.085\,(M \times F2) + 0.8\,(P \times F3)\ \text{mW} \qquad (6.7)$$

where

F1 = average logic module switching rate in MHz

F2 = average clock pin switching rate in MHz

F3 = average I/O switching rate in MHz

M = number of logic modules connected to the clock pin

N = number of logic modules used on the chip

P = number of I/O pairs used (input + output), with 50 pF load

As an example of a power-dissipation calculation, consider an Actel 1020B-2 with a 20 MHz clock. We shall initially assume 100 percent utilization of the 547 Logic Modules and assume that each switches at an average speed of 5 MHz. We shall also initially assume that we use all of the 69 I/O Modules and that each switches at an average speed of 5 MHz. Using Eq. 6.7, the Logic Modules dissipate

$$P_{LM} = (0.2)\,(547)\,(5) = 547\ \text{mW}, \qquad (6.8)$$

and the I/O Module dissipation is

$$P_{IO} = (0.8)\,(69)\,(5) = 276\ \text{mW}. \qquad (6.9)$$

If we assume the clock buffer drives 20 percent of the Logic Modules, then the additional power dissipation due to the clock buffer is

$$P_{CLK} = (0.085)\,(547)\,(0.2)\,(5) = 46.495\ \text{mW}. \qquad (6.10)$$

The total power dissipation is thus

$$P_D = (547 + 276 + 46.5) = 869.5\ \text{mW}, \qquad (6.11)$$

or about 900 mW (with an accuracy of certainly no better than ± 100 mW).

Suppose we intend to use a **very thin quad flatpack** (**VQFP**) with no cooling (because we are trying to save area and board height). From Table 6.3 the thermal

resistance, θ_{JA}, is approximately 68 °CW^{-1} for an 80-pin VQFP. Thus the maximum junction temperature under industrial worst-case conditions ($T_A = 85$ °C) will be

$$T_J = (85 + (0.87)\ (68)) = 144.16\ °C, \tag{6.12}$$

(with an accuracy of no better than 10 °C). Actel specifies the maximum junction temperature for its devices as $T_{Jmax} = 150$ °C (T_{Jmax} for Altera is also 150 °C, for Xilinx $T_{Jmax} = 125$°C). Our calculated value is much too close to the rated maximum for comfort; therefore we need to go back and check our assumptions for power dissipation. At or near 100 percent module utilization is not unreasonable for an Actel device, but more questionable is that all nodes and I/Os switch at 5 MHz.

Our real mistake is trying to use a VQFP package with a high θ_{JA} for a high-speed design. Suppose we use an 84-pin PLCC package instead. From Table 6.3 the thermal resistance, θ_{JA}, for this alternative package is approximately 44 °CW^{-1}. Now the worst-case junction temperature will be a more reasonable

$$T_J = (85 + (0.87)\ (44)) = 123.28\ °C. \tag{6.13}$$

It is possible to estimate the power dissipation of the Actel architecture because the routing is regular and the interconnect capacitance is well controlled (it has to be since we must minimize the number of series antifuses we use). For most other architectures it is much more difficult to estimate power dissipation. The exception, as we saw in Section 5.4 "Altera MAX," are the programmable ASICs based on programmable logic arrays with passive pull-ups where a substantial part of the power dissipation is static.

6.6.2 Power-On Reset

Each FPGA has its own power-on reset sequence. For example, a Xilinx FPGA configures all flip-flops (in either the CLBs or IOBs) as either SET or RESET. After chip programming is complete, the global SET/RESET signal forces all flip-flops on the chip to a known state. This is important since it may determine the initial state of a state machine, for example.

6.7 Xilinx I/O Block

The Xilinx I/O cell is the **input/output block** (**IOB**). Figure 6.21 shows the Xilinx XC4000 IOB, which is similar to the IOB in the XC2000, XC3000, and XC5200 but performs a superset of the options in these other Xilinx FPGAs.

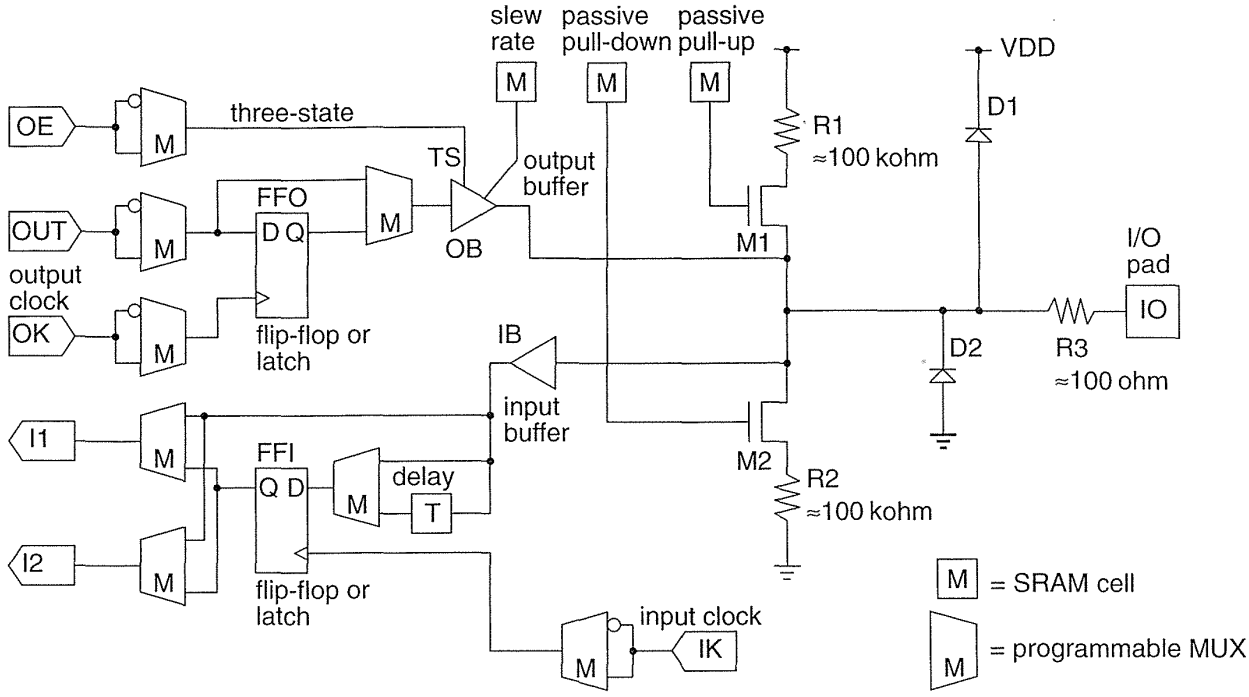


FIGURE 6.21 The Xilinx XC4000 family IOB (input/output block). (*Source:* Xilinx.)

The outputs contain features that allow you to do the following:

- Switch between a totem-pole and a complementary output (XC4000H).
- Include a passive pull-up or pull-down (both *n*-channel devices) with a typical resistance of about 50 kΩ.
- Invert the three-state control (output enable OE or three-state, TS).
- Include a flip-flop, or latch, or a direct connection in the output path.
- Control the slew rate of the output.

The features on the inputs allow you to do the following:

- Configure the input buffer with TTL or CMOS thresholds.
- Include a flip-flop, or latch, or a direct connection in the input path.
- Switch in a delay to eliminate an input hold time.

Figure 6.22 shows the timing model for the XC5200 family.[7] It is similar to the timing model for all the other Xilinx LCA FPGAs with one exception—the XC5200 does not have registers in the I/O cell; you go directly to the core CLBs to include a flip-flop or latch on an input or output.

[7]October 1995 (v. 3.0) data sheet.

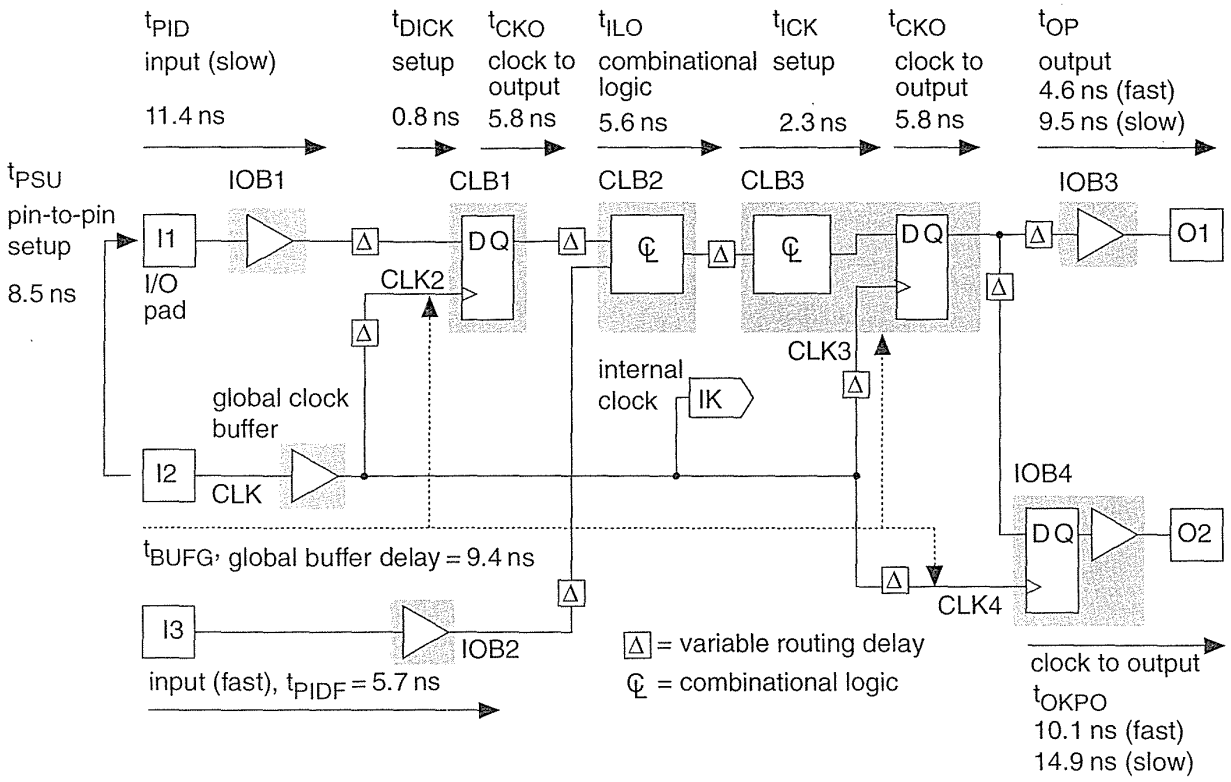


FIGURE 6.22 The Xilinx LCA (Logic Cell Array) timing model. The paths show different uses of CLBs (Configurable Logic Blocks) and IOBs (Input/Output Blocks). The parameters shown are for an XC5210-6. (*Source:* Xilinx.)

6.7.1 Boundary Scan

Testing PCBs can be done using a bed-of-nails tester. This approach becomes very difficult with closer IC pin spacing and more sophisticated assembly methods using surface-mount technology and multilayer boards. The IEEE implemented boundary-scan standard 1149.1 to simplify the problem of testing at the board level. The Joint Test Action Group (JTAG) developed the standard; thus the terms JTAG boundary scan or just JTAG are commonly used.

Many FPGAs contain a standard boundary-scan test logic structure with a four-pin interface. By using these four signals, you can program the chip using ISP, as well as serially load commands and data into the chips to control the outputs and check the inputs. This is a great improvement over bed-of-nails testing. We shall cover boundary scan in detail in Section 14.6, "Scan Test."

6.8 Other I/O Cells

The Altera MAX 5000 and 7000 use the **I/O Control Block** (IOC) shown in Figure 6.23. In the MAX 5000, all inputs pass through the chipwide interconnect. The MAX 7000E has special fast inputs that are connected directly to macrocell registers in order to reduce the setup time for registered inputs.

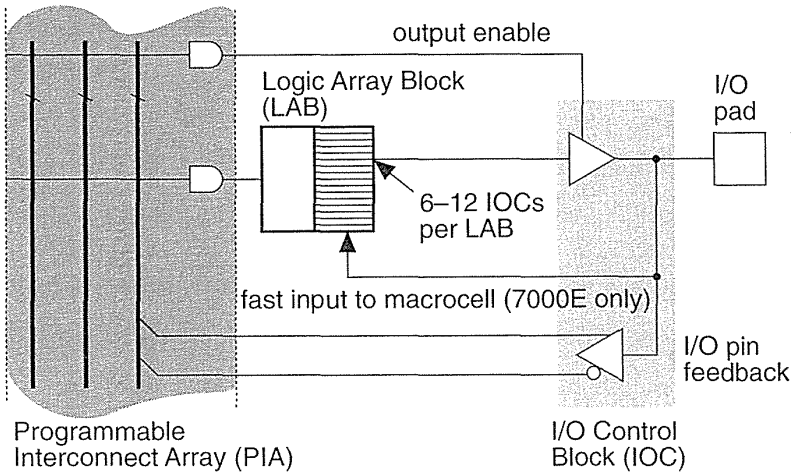


FIGURE 6.23 A simplified block diagram of the Altera I/O Control Block (IOC) used in the MAX 5000 and MAX 7000 series. The I/O pin feedback allows the I/O pad to be isolated from the macrocell. It is thus possible to use a LAB without using up an I/O pad (as you often have to do using a PLD such as a 22V10). The PIA is the chipwide interconnect.

The FLEX 8000 and 10k use the **I/O Element** (IOE) shown in Figure 6.24 (the MAX 9000 IOC is similar). The interface to the IOE is directly to the chipwide interconnect rather than the core logic. There is a separate bus, the **Peripheral Control Bus**, for the IOE control signals: clock, preset, clear, and output enable.

The AMD MACH 5 family has some I/O features not currently found on other programmable ASICs. The MACH 5 family has 3.3 V and 5 V versions that are both suitable for mixed-voltage designs. The 3 V versions accept 5 V inputs, and the outputs of the 3 V versions do not drive above 3.3 V. You can apply a voltage up to 5.5 V to device inputs before you connect VDD (this is known as **hot insertion** or hot switching, allowing you to swap cards with power still applied without causing latch-up). During power-up and power-down, all I/Os are three-state, and there is no I/O current during power-down, allowing power-down while connected to an active bus. All MACH 5 devices in the same package have the same pin configuration, so you can increase or reduce the size of device after completing the board layout.

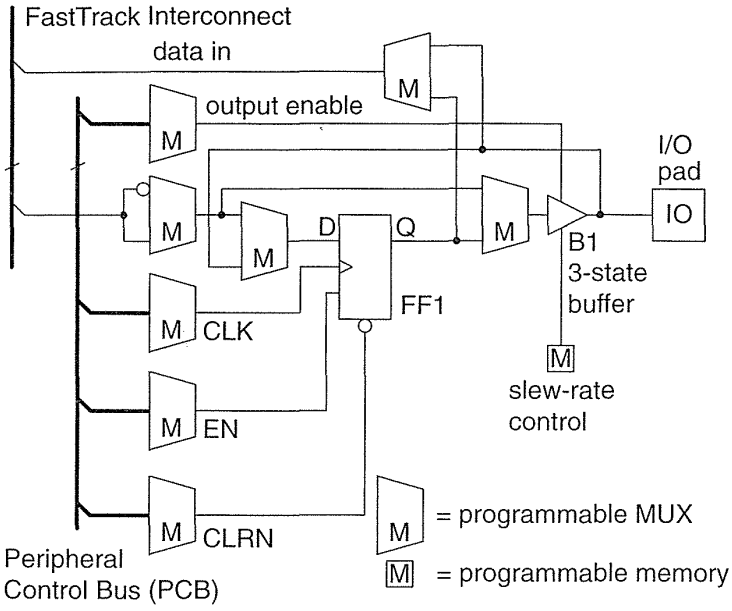


FIGURE 6.24 A simplified block diagram of the Altera I/O Element (IOE), used in the FLEX 8000 and 10k series. The MAX 9000 IOC (I/O Cell) is similar. The FastTrack Interconnect bus is the chipwide interconnect. The PCB is used for control signals common to each IOE.

6.9 Summary

Among the options available in I/O cells are: different drive strengths, TTL-compatibility, registered or direct inputs, registered or direct outputs, pull-up resistors, over-voltage protection, slew-rate control, and boundary-scan. Table 6.4 shows a list of features. Interfacing an ASIC with a system starts at the outputs where you check the voltage levels first, then the current levels. Table 6.5 is a look-up table for Tables 6.6 and 6.7, which show the I/O resources present in each type of programmable ASIC (using the abbreviations of Table 6.4).

Important points that we covered in this chapter are the following:

- Outputs can typically source or sink 5–10 mA continuously into a DC load, and 50–200 mA transiently into an AC load.
- Input buffers can be CMOS (threshold at $0.5V_{DD}$) or TTL (1.4 V).
- Input buffers normally have a small hysteresis (100–200 mV).
- CMOS inputs must never be left floating.
- Clamp diodes to GND and VDD are present on every pin.
- Inputs and outputs can be registered or direct.
- I/O registers can be in the I/O cell or in the core.
- Metastability is a problem when working with asynchronous inputs.

TABLE 6.4 I/O options for programmable ASICs.

Code[1]	I/O Option	Function
IT/C	TTL/CMOS input	Programmable input buffer threshold
OT/C	TTL/CMOS output	Complementary or totem-pole output
nSNK	Sink capability	Maximum current sink ability (e.g., 12SNK is $I_0 = 12$ mA sink)
nSRC	Source capability	Maximum current source ability (e.g., 12SRC is $I_0 = -12$ mA source)
5/3	5V/3V	Separate I/O and core voltage supplies
OD	Open drain/collector	Programmable open-drain at the output buffer
TS	Three-state	Output buffer with three-state control
SR	Slew-rate control	Fast or slew-rate limited output buffer to reduce ground bounce
PD	Pull-down	Programmable pull-down device or resistor at the I/O pad
PU	Pull-up	Programmable pull-up device or resistor at the I/O pad
EP	Enable polarity	Driver control can be positive (three-state) or negative (enable).
RI	Registered input	Inputs may be registered in I/O cell.
RO	Registered output	Outputs may be registered in I/O cell.
RIO	Registered I/O	Both inputs and outputs may be registered in I/O cell.
ID	Input delay	Input delay to eliminate input hold time
JTAG	JTAG	Boundary-scan test
SCH	Schmitt trigger	Schmitt trigger or input hysteresis
HOT	Hot insertion	Inputs protected from hot insertion
PCI	PCI compliant	Output buffer characteristics comply with PCI specifications.

[1]These codes are used in Tables 6.6 and 6.7.

6.10 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

6.1 (I/O resources, 60 min.) Obtain the specifications for the latest version of your choice of FPGA vendor from a data book or online data sheet and complete a table in the same format as Tables 6.6 and 6.7.

6.2 (I/O timing, 60 min.) On-chip delays are only half the battle in a typical design. Using data book parameters for an FPGA that you choose, estimate (worst-case commercial) how long it takes to bring a signal on-chip; through an input regis-

TABLE 6.5 I/O Cell Tables.

Programmable ASIC family	Table	Programmable ASIC family	Table
Actel (ACT 1)	Table 6.6	Actel (ACT 3)	Table 6.7
Xilinx (XC3000)		Xilinx LCA (XC5200)	
Actel (ACT 2)		Altera FLEX (8000/10k)	
Altera MAX (EPM 5k)		AMD MACH 5	
Xilinx EPLD (XC7200/7300)		Actel 3200DX	
QuickLogic (pASIC 1)		Altera MAX (EPM 9000)	
Crosspoint (CP20K)		Xilinx (XC8100)	
Altera MAX (EPM 7000)		AT&T ORCA (2C)	
Atmel (AT6000)		Xilinx (XC4000)	

ter (a flip-flop); through a combinational function (assume an inverter); and back off chip again through another (flip-flop) register. Give your answer in three parts:

a. The delay from a CMOS-level pad input (trip-point of 0.5) to the D input of the input register plus the flip-flop setup time.

b. The delay (measured from the clock, so include the clock-to-Q delay) through the inverter to the output register plus the setup time.

c. The delay from the output register (measured from the clock edge) to the output pad (trip point of 0.5) with a 50 pF load.

In each case give your answers: (i) Using data book symbols (specify which symbols and where in the data books you found them); and (ii) as calculated values, in nanoseconds, using a speed grade that you specify. State and explain very clearly any assumptions that you need to make about the clock to determine the setup times.

6.3 (Clock timing, 30 min.) When we calculate FPGA timing we need to include the time it takes to bring the clock onto the chip. For an FPGA you choose, estimate (worst-case commercial) the delay from the clock pad (0.5 trip-point) to the clock pin of an internal flip-flop

a. in terms of data book symbols (specify which and where you found them—t_{AB} on p. 2-32 of the ABC 1994 data book, for example), and

b. as calculated values in nanoseconds.

6.4 (**Bipolar drivers, 60 min.) The circuit in Figure 6.3 uses *npn* transistors.

a. Design a similar circuit that uses *pnp* transistors.

b. The *pnp* circuit may work better, why?

c. Design an even better circuit that uses *npn* and *pnp* transistors.

d. Explain why your circuit is even better.

e. Draw a diagram for a controller using op-amps instead of bipolar transistors.

TABLE 6.6 Programmable ASIC I/O logic resources.

	Actel (ACT 1)	Xilinx (XC3000)	Actel (ACT 2)
I/O cell name	I/O module	IOB (Input/Output Block)	I/O module
I/O cell functions[1]	TS, 10SRC, 10SNK	TS, RIO, IT/C, PU, 4SRC, 4SNK, 8SRC (3100), 8SNK (3100)	TS, (RIO)[2], 10SRC, 10SNK
Number of I/O cells	Max. I/O: 57 (1010) 69 (1020)	Max. I/O: 64 (3020) 144 (3090)	Max. I/O: 83 (1225) 140 (1280)

	Altera MAX 5000	Xilinx EPLD	QuickLogic (pASIC 1)
I/O cell name	I/O control block	I/O block	Bidirectional input/output cell & dedicated input cell
I/O cell functions	TS, 4SRC, 8SNK	(TS), (RI)[3], 5/3, 4SRC, 12SNK	TS
Number of I/O cells	8 (5016) –64 (5192)	38 (7336)–156 (73144) 36 (7236) –72 (7272)	32 (QL6X8)[4] –104 (QL16X24)

	Crosspoint (CP20K)	Altera MAX 7000	Atmel (AT6000)
I/O cell name	I/O cell	IOC (I/O Control Block)	Entrance and exit cells
I/O cell functions	TS, SR, IC/T , JTAG, SCH	TS, SR, 5/3, PCI, 4SRC, 12SNK	TS, SR, PU, OD, IT/C, 16SRC, 16SNK
Number of I/O cells	91 (20220) –270 (22000)	36 (7032) –164 (7256)	96 (6002) –160 (6010)

[1]Code definitions are listed in Table 6.4.
[2]ACT 2 I/O Module is separate from the I/O Pad Driver.
[3]Xilinx EPLD uses a mixture of I/O blocks, input-only blocks, and output-only blocks. The I/O blocks and input only blocks contain the equivalent of a D flip-flop (configured to be a flip-flop or latch).
[4]8 I/O are dedicated inputs on all parts.

6.5 (Xilinx output buffers, 15 min.) For the Xilinx XC2000 and XC3000 series[8]: $I_{OLpeak} = 120\,mA$ and $I_{OHpeak} = 80\,mA$; for the XC4000 family: $I_{OLpeak} = 160\,mA$ and $I_{OHpeak} = 130\,mA$; and for the XC7300 series: $I_{OLpeak} = 100\,mA$ and $I_{OHpeak} = 65\,mA$. For a typical 0.8–1.0 μm process:

[8]1994 databook, p. 8-15 and p. 9-23.

TABLE 6.7 Programmable ASIC I/O logic resources (contd.).

	Actel (ACT 3)	Xilinx LCA (XC5200)	Altera FLEX (8000/10k)
I/O cell name	I/O Module	IOB (I/O Block)	IOE (I/O Element)
I/O cell functions[1]	TS, SR, (RIO)[2], 8SRC, 12SNK	TS, PU, PD, JTAG	TS, SR, RI or RO, JTAG, PCI(8k), 4SRC, 12SNK
Number of I/O cells	80 (1415) –228 (14100)	84 (5202) –244 (5215)	78 (8282)–208 (81500) 150 (10K10) –406 (10K100)

	AMD MACH 5	Actel 3200DX	Altera MAX (EPM 9000)
I/O cell name	I/O Cell	I/O Module	IOE (I/O Element)
I/O cell functions	TS, 3.2SRC, 16SNK, PCI	Same as ACT 2	TS, SR, 5/3, PCI, JTAG, 4SRC, 8SNK
Number of I/O cells	120 (M5-128) –256 (M5-512)	126 (A3265DX) –292 (A32400DX)	168 (9320) –216 (9560)

	Xilinx (XC8100)[3]	AT&T ORCA 2C	Xilinx (XC4000)
I/O cell name	I/O Cell	PIC (Programmable input/output cells)	IOB (I/O Block)
I/O cell functions	TS, PU, IT/C (global), JTAG, PCI, 4SRC, 4/24SNK[4]	TS, IT/C, ID, PU, PD, OD, JTAG, PCI, (6SRC and 12SNK) or (3SRC and 6SNK), SCH	TS, RIO, JTAG, ID, IT/C, OT/C, PU, PD, 4SRC, 12SNK, 24SNK (4000A/H)
Number of I/O cells	32 (8100) –208 (8109)	160 (2C04) –480 (2C40)	80(4003) –256(4025)

[1]Code definitions are listed in Table 6.4.
[2] ACT 3 I/O Module is separate from the I/O Pad Driver.
[3]Discontinued August 1986.
[4]Two output modes: Capacitive (4SNK) and Resistive (24SNK).

p-channel (20/1): $I_{DS} = 3.0–5.0$ mA with $V_{DS} = -5$ V, $V_{GS} = -5$ V
n-channel (20/1): $I_{DS} = 7.5–10.0$ mA with $V_{DS} = 5$ V, $V_{GS} = 5$ V

a. Calculate the effective sizes of the transistors in the Xilinx output buffer.

b. Why might these only be "effective" sizes?

c. The Xilinx data book gives values for "source current and output high impedance" shown in Table 6.8. Graph the buffer characteristics when sourcing current.

d. Explain which parts in Table 6.8 use complementary output buffers and which use totem-pole outputs and explain how you can tell.

e. Can you explain how Xilinx arrived at the figures for impedance?

f. Comment on the method that Xilinx used.

g. Suggest and calculate a better measure of impedance.

TABLE 6.8 Xilinx output buffer characteristics.

	V_O (output voltage)[1]/V			
Part	**4**	**3**	**2**	**Impedance/Ω**
I_O (2018)	−30	−52	−60	30
I_O (3020)	−35	−60	−75	30
I_O (4005)	0	−12	−50	25
I_O (73108)	0	−10	−26	40

[1]Currents in milliamperes.

6.6 (Xilinx logic levels, 10 min.) Most manufacturers measure V_{OLmax} with V_{DD} set to its minimum value, Xilinx measures V_{OLmax} at V_{DDmax}. For example, for the Xilinx XC4000[9]: $V_{OLmax} = 0.4$ V at $I_{OLmax} = 12$ mA and V_{DDmax}. A footnote also explains that V_{OLmax} is measured with "50 % of the outputs simultaneously sinking 12 mA."

a. Can you explain why Xilinx measures V_{OLmax} this way?

b. What information do you need to know to estimate V_{OLmax} if all the other outputs were *not* sourcing or sinking any current.

6.7 (Output levels, 10 min.) In Figure 6.7(b–d) the PAD signal is labeled with different levels: In Figure 6.7(b) the PAD high and low levels are V_{OHmin} and V_{OLmax} respectively, in Figure 6.7(c) they are V_{DD} and V_{OLmax}, and in Figure 6.7(c) they are V_{OHmin} and V_{SS}.

a. Explain why this is.

b. In no more than 20 words explain the difference between V_{DD} and V_{OHmin} as well as the difference between V_{OLmax} and V_{SS}.

6.8 (TTL and CMOS outputs, 10 min.) The ACT 2 figures for t_{DLH} and t_{DHL} in Figure 6.7 are for the CMOS levels. For TTL levels the figures are (with the CMOS figures in parentheses): $t_{DLH} = 10.6$ ns (13.5 ns), and $t_{DHL} = 13.4$ ns (11.2 ns). The output buffer is the same in both cases, but the delays are measured using different levels. Explain the differences in these delays quantitatively.

[9] Xilinx 1994 data book, p. 2-48.

6.9 (Bus-keeper contention, 30 min.) Figure 6.25 shows a three-state bus, similar to Figure 6.5, that has a bus keeper on CHIP1 and a pull-up resistor that is part of a Xilinx IOB on CHIP2—we have a type of bus-keeper contention. For the XC3000 the **pull-up current** is 0.02–0.17 mA and thus RL1 is between 5 and 50 kΩ (1994 data book, p. 2-155).

a. Explain what might happen when both the bus drivers turn off.

b. Have you considered all possibilities?

c. Is bus-keeper contention a problem?

d. In the PCI specification control signals are required to be sustained three-state. A driver must deassert a control signal to the inactive state (high for the PCI control signals) for at least one clock cycle before three-stating the line. This means that a driver has to "put the signal back where it found it." Does this affect your answers?

e. Suggest a "fix" that stops you having to worry about any potential problems.

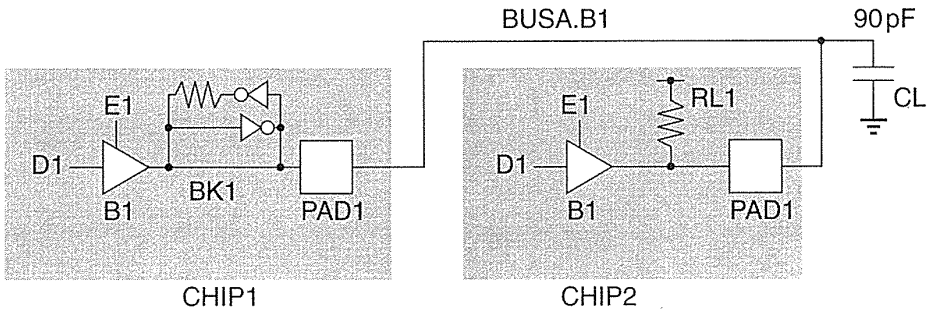


FIGURE 6.25 A bus keeper, BK1, and pull-up resistor, RL1, on the same bus.

6.10 (Short-circuit, 10 min.) What happens if you short-circuit the output of a complementary output buffer to (**a**) GND and (**b**) VDD? (**c**) What difference does it make if the output buffer is complementary or a totem-pole?

6.11 (Transmission line bias, 10 min.)

a. Why do we adjust the resistors in Figure 6.10(c) so that the Thévenin equivalent voltage source is 1.6 V?

b. What current does a driver have to sink if we want $V_{OLmax} = 0.4$ V?

c. What current does a driver have to source if we want $V_{OHmin} = 2.4$ V?

6.12 (Ground resistance, 10 min.) Calculate the resistance of an aluminum GND net that is 0.5 mm long and 10 μm wide.

6.13 (*Temperature) (**a**) (30 min.) You are about to ship a product and you have a problem with an FPGA. A high case temperature is causing it to be slower than you thought. You calculated the power dissipation, but you forgot that the InLet microprocessor is toasting the next door FPGA. You have no easy way to calculate T_J now, so we need to measure it in order to redesign the FPGA with fixed I/O loca-

tions. You remember that a diode forward voltage has a temperature coefficient of about $-2\,\text{mV°C}^{-1}$ and there are clamp diodes on the FPGA I/O. Explain, using circuit diagrams, how to measure the T_J of an FPGA in-circuit using: a voltage supply, DVM, thermometer, resistors, spoon, and a coffee maker. **(b)** (**120 min.) Try it.

6.14 (Delay measurement, 10 min.) Sumo Silicon has a new process ready before we do and Sumo's data book timing figures are much better than ours. Explain how to reduce our logic delays by changing our measurement circuits and trip points.

6.15 (Data sheets, 10 min.) In the 1994 data book Xilinx specifies $V_{ILmin} = 0.3$ V (and $V_{ILmax} = 0.8$ V) for the XC2000L. Why does this surprise you and what do you think the value for V_{ILmin} really is? FPGA vendors produce thousands of pages of data every year with virtually no errors. It is important to have the confidence to question a potential error.

6.16 (GTL, 60 min.) Find the original reference to Gunning transistor logic. Write a one-page summary of its uses and how it works.

6.17 (Thresholds, 10 min.) With some FPGAs it is possible to configure an output at TTL thresholds and an input (on the same pad) at CMOS thresholds. Can you think of a reason why you might want to do this?

6.18 (Input levels, 10 min.) When we define $V_{IHmin} = 0.7V_{DD}$, why do we calculate the *minimum* value of V_{IH} using $V_{DDmax} = 5.5$ V?

6.19 (Metastability equations, 30 min.)

a. From Eq. 6.4 show that if we make two measurements of t_r and MTBF then:

$$\tau_c = \frac{t_{r1} - t_{r2}}{\ln \text{MTBF}_1 - \ln \text{MTBF}_2}, \tag{6.14}$$

$$T_0 = \frac{\exp \frac{t_{r1}}{\tau_c}}{\text{MTBF}_1 f_c f_d}. \tag{6.15}$$

b. MTBU is extremely sensitive to variations in τ_c, show that:

$$\frac{d}{d\tau_c}\text{MTBU} = \frac{-t_r}{\tau_c^2}. \tag{6.16}$$

c. Show that the variation in MTBU is related to the variation in τ_c by the following expression:

$$\left(\frac{\Delta \text{MTBU}}{\text{MTBU}}\right) = \frac{-t_r}{\tau_c}\left(\frac{\Delta \tau_c}{\tau_c}\right). \tag{6.17}$$

6.20 (***Alternative metastability solutions, 120 min.) Write a minitutorial on metastability solutions. The best sources for this type of information are usually application notes written by FPGA and TTL manufacturers, many of which are available on the Web (TI is a good source on this topic).

6.21 (Altera 8000 I/O, 10 min) Figure 6.26 shows the Altera FLEX 8000 I/O characteristics. Determine as much as you are able to from these figures.

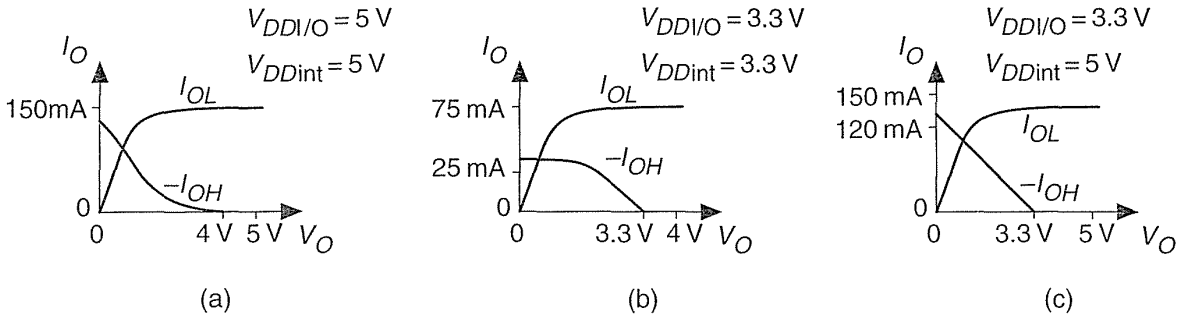


FIGURE 6.26 (a) Altera FLEX 8000 I/O characteristics operating at 5 V. (b) EPF8282V I/O operating at 3.3 V. (c) Characteristics with mixed 5V and 3.3 V I/O operation.

6.22 (Power calculation, 60 min.) Suppose we wish to limit power dissipation on an ACT 1 A1020 chip to below 1 W for a 44-pin PLCC package.

a. Derive an equation for the number of logic modules, number of I/O modules, number of modules connected to the clock and system clock frequency in terms of the package parameters and the worst-case T_A.

b. Assume:

100 percent utilization of I/Os,
50 percent are outputs connected to a 50 pF load,
100 percent utilization of logic modules,
10 percent of the logic modules are connected to the clock,
20 percent of the logic modules toggle every clock cycle,
20 percent of the I/Os toggle every clock cycle.

Determine an upper limit on clock frequency.

c. Next vary each of the assumptions you made in part b. Draw graphs showing the variation of clock frequency as you vary each of the above parameters, including the power dissipation limit (a spreadsheet will help).

d. Can you draw any conclusions from this exercise?

6.23 (Switch debounce, 30 min) Design a logic circuit to "debounce" the output from a buffer whose input is connected to a bounce-prone switch. Your system operates at a clock frequency of 1 MHz.

6.24 (Plugs and sockets, 30 min.) Draw the plugs and sockets (to scale) for the technologies in Table 6.9.

TABLE 6.9 TTL-compatible CMOS logic thresholds (Problems 6.24 and 6.25).[1]

	Input levels		Output levels driving TTL				Output levels driving CMOS[2]			
Family	V_{IHmin}	V_{ILmax}	V_{OHmin}	I_{OHmax}	V_{OLmax}	I_{OLmax}	V_{OHmin}	I_{OHmax}	V_{OLmax}	I_{OLmax}
74HCT	2.0	0.8	3.84	–4.0	0.33	4.0	4.4	–0.02	0.1	0.02
74HC	3.85	1.35	3.84	–4.0	0.33	4.0	4.4	–0.02	0.1	0.02
74ACT	2.0	0.8	3.76	–24.0	0.37	24.0	4.4	–0.05	0.1	0.05
74AC	3.85	1.35	3.76	–24.0	0.37	24.0	4.4	–0.05	0.1	0.05

[1]All voltages in volts, all currents in milliamperes.
[2]$I_{IHmax} = \pm 0.001$ mA, $I_{ILmax} = \pm 0.001$ mA for all families.

6.25 (TTL compatibility, 30 min.) Explain very carefully, giving an example using actual figures from the tables, how you would determine the compatibility between the TTL and CMOS logic thresholds shown in Table 6.9 and Table 6.10 and the FPGA logic thresholds in Table 6.1.

TABLE 6.10 TTL logic thresholds (Problem 6.25).[1]

TTL Family[2]	V_{IHmin}	V_{ILmax}	V_{OHmin}	I_{OHmax}	V_{OLmax}	I_{OLmax}	I_{IHmax}	I_{ILmax}
74S	2.0	0.8	2.7	–1.0	0.5	20.0	0.05	–2.0
74LS	2.0	0.8	2.7	–0.4	0.5	8.0	0.02	–0.4
74ALS	2.0	0.8	2.7	–0.4	0.5	8.0	0.02	–0.2
74AS	2.0	0.8	2.7	–2.0	0.5	20.0	0.02	–0.5
74F	2.0	0.8	2.7	–1.0	0.5	20.0	0.02	–0.6
74FCT	2.0	0.8	2.4	–15.0	0.5	48.0	±0.005	±0.005
74FCT-T	2.0	0.8	2.4	–8.0	0.5	48.0	±0.005	±0.005

[1]All voltages in volts, all currents in milliamperes
[2]Other (older) TTL and CMOS logic families include 4000, 74, 74H, and 74L

6.26 (ECL, 30 min.) Emitter-coupled logic (ECL) uses a positive supply, $V_{CC} = 0$ V, and a negative supply, $V_{EE} = -5.2$ V. The highest logic voltage allowed is –0.81 V and the lowest is –1.85 V. Table 6.11 shows the ECL 10K thresholds.

a. Calculate the high-level and low-level noise margins.

b. Find out the 100K thresholds and

c. calculate the 100K noise margins.

TABLE 6.11 ECL logic thresholds (Problem 6.26).

	V_{IHmin}/V	V_{ILmax}/V	V_{OHmin}/V	V_{OLmax}/V
ECL10K	–1.105	–1.475	–0.980	–1.630
ECL100K				

6.27 (Schmitt trigger, 30 min.) Find out the typical hysteresis for a TTL Schmitt trigger. What are the advantages and disadvantages of changing the hysteresis?

6.28 (Hysteresis, 20 min.)

a. Draw the transfer curve for an inverting buffer with very high gain that has a switching threshold centered at 2.2 V and 300 mV hysteresis.

b. If the center of the characteristic shifts by –0.3 V and +0.4 V and the hysteresis varies from 260 mV to 350 mV, calculate V_{IHmin} and V_{ILmax}.

6.29 (Driving an LED, 30 min.) Find out the typical current and voltage drive required by an LED and design a circuit to drive it. List your sources of information.

6.30 (**Driving TTL, 60 min.) Find out the input current requirements of different TTL families and write a minitutorial on the I/O requirements (in particular the current) when driving high and low levels onto a bus.

6.11 Bibliography

Wakerly's [1994] book describes TTL and CMOS logic thresholds as well as noise margins. The specification of digital I/O interfaces (voltage and current levels) is defined by the JEDEC (part of the Electronic Industries Association, EIA) JC-16 committee standards [JEDEC I/O]. Standards for ESD measurement are not as well defined; companies use a range of specifications: MIL-STD-883, EIAJ, a published model used by AT&T (see, for example, p. 5-13 to p. 5-19 in the AT&T 1995 FPGA data book) as well as JEDEC and ANSI/IEEE standards [JEDEC I/O, JEDEC ESD,

ANSI/IEEE ESD]. You are not likely to find any of these standards at the library, but they are available through specialist technical document distributors (typical 1996 costs were about $25 for the JEDEC documents; catalogs are generally free of charge). These standards are not technical reports, most only contain a few pages, but they are the source of the parameters that you see in data sheets.

6.12 References

Page numbers in brackets after a reference indicate its location in the chapter body.

[JEDEC I/O] [p. 246] In numerical (not chronological) order the relevant JEDEC standards for I/O are:

JESD8-A. Interface Standard for Nominal 3 V/3.3 V Supply Digital Integrated Circuits (June 1994). This standard replaces JEDEC Standards 8, 8-1, and 8-1-A and defines the DC interface parameters for digital circuits operating from a power supply of nominal 3 V/3.3 V.

JESD8-2. Standard for Operating Voltages and Interface Levels for Low Voltage Emitter-Coupled Logic (ECL) Integrated Circuits (March 1993). Describes 300K ECL (voltage and temperature compensated, with threshold levels compatible with 100K ECL).

JESD8-3. Gunning Transceiver Logic (GTL) Low-Level, High-Speed Interface Standard for Digital Integrated Circuits (Nov. 1993). Defines the DC input and output specifications for a low-level, high-speed interface for integrated circuits.

JESD8-4. Center-Tap-Terminated (CTT) Low-Level, High-Speed Interface Standard for Digital Integrated Circuits (Nov. 1993). Defines the DC I/O specifications for a low-level, high-speed interface for integrated circuits that can be a superset of LVCMOS and LVTTL.

JESD8-5. 2.5 V +/– 0.2 V (Normal Range), and 1.8 V–2.7 V (Wide Range) Power Supply Voltage and Interface Standard for Nonterminated Digital Integrated Circuit (Oct. 1995). Defines power supply voltage ranges, DC interface parameters for a high-speed, low-voltage family of nonterminated digital circuits.

JESD8-6. High Speed Transceiver Logic (HSTL): A 1.5 V Output Buffer Supply Voltage Based Interface Standard for Digital Integrated Circuits (Aug. 1995). Describes a 1.5 V high-performance CMOS interface suitable for high I/O count CMOS and BiCMOS devices operating at over 200 MHz.

JESD12-6. Interface Standard for Semicustom Integrated Circuits (March 1991). Defines logic interface levels for CMOS, TTL, and ECL inputs and outputs for 5 V operation.

[JEDEC ESD, ANSI/IEEE ESD] The JEDEC and IEEE standards for ESD are:

JESD22-C101. Field Induced Charged Device Model Test Method for Electrostatic Discharge Withstand Thresholds of Microelectronic Components (May 1995). Describes Charged Device Model that simulates charging/discharging events that occur in production equipment and processes. Potential for CDM ESD events occur with metal-to-metal contact in manufacturing.

ANSI/EOS/ESD S5.1-1993. Electrostatic Discharge (ESD) Sensitivity Testing, Human Body Model (HBM), Component Level.

ANSI/IEEE C62.47-1992. Guide on Electrostatic Discharge (ESD): Characterization of the ESD Environment.

ANSI/IEEE 1181-1991. Latchup Test Methods for CMOS and BiCMOS Integrated Circuit Process Characterization.

PCI Local Bus Specification, Revision 2.1, June 1, 1995. Available from PCI Special Interest Group, PO Box 14070, Portland OR 97214. (800) 433-5177 (U.S.), (503)797-4207 (International). 282 p. Detailed description of the electrical and mechanical requirements for the PCI Bus written for engineers who already understand the basic operation of the bus protocol. [p. 242]

Wakerly, J. F. 1994. *Digital Design: Principles and Practices.* 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 840 p. ISBN 0-13-211459-3. TK7874.65.W34. [p. 272]

PROGRAMMABLE ASIC INTERCONNECT

7

All FPGAs contain some type of **programmable interconnect**. The structure and complexity of the interconnect is largely determined by the programming technology and the architecture of the basic logic cell. The raw material that we have to work with in building the interconnect is aluminum-based metallization, which has a sheet resistance of approximately 50 mΩ/square and a line capacitance of $0.2\,\mathrm{pFcm}^{-1}$. The first programmable ASICs were constructed using two layers of metal; newer programmable ASICs use three or more layers of metal interconnect.

7.1 Actel ACT

The Actel ACT family interconnect scheme shown in Figure 7.1 is similar to a channeled gate array. The channel routing uses dedicated rectangular areas of fixed size within the chip called **wiring channels** (or just **channels**). The **horizontal channels** run across the chip in the horizontal direction. In the vertical direction there are similar **vertical channels** that run over the top of the basic logic cells, the Logic Modules. Within the horizontal or vertical channels wires run horizontally or vertically, respectively, within **tracks**. Each track holds one wire. The **capacity** of a fixed wiring channel is equal to the number of tracks it contains. Figure 7.2 shows a detailed view of the channel and the connections to each Logic Module—the **input stubs** and **output stubs**.

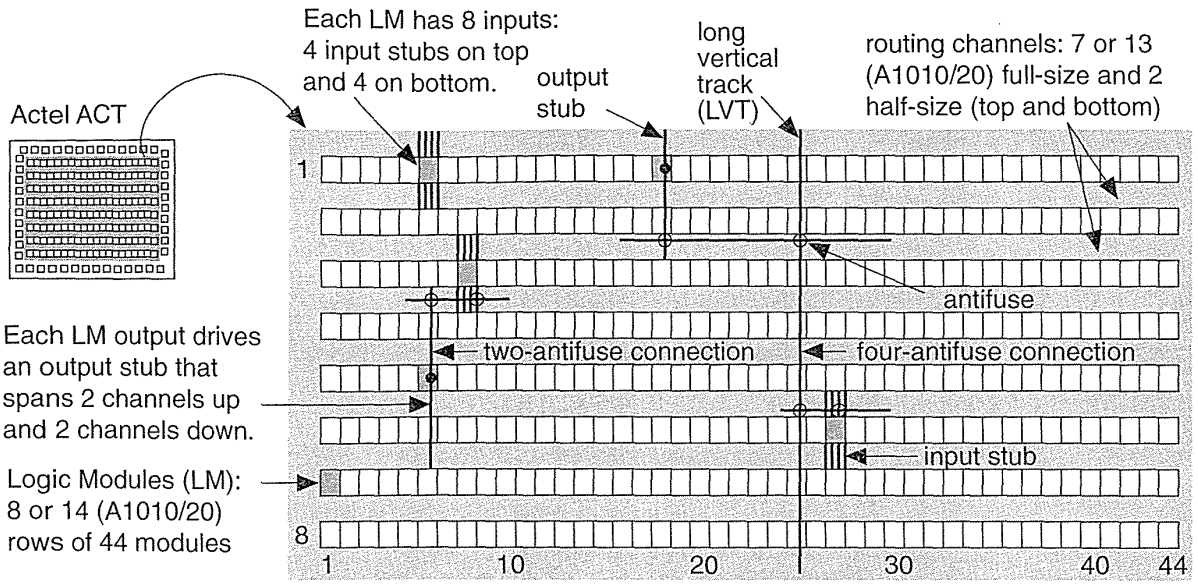


FIGURE 7.1 The interconnect architecture used in an Actel ACT family FPGA. (*Source:* Actel.)

In a channeled gate array the designer decides the location and length of the interconnect within a channel. In an FPGA the interconnect is fixed at the time of manufacture. To allow programming of the interconnect, Actel divides the fixed interconnect wires within each channel into various lengths or **wire segments**. We call this **segmented channel routing**, a variation on channel routing. Antifuses join the wire segments. The designer then programs the interconnections by blowing antifuses and making connections between wire segments; unwanted connections are left unprogrammed. A statistical analysis of many different layouts determines the optimum number and the lengths of the wire segments.

7.1.1 Routing Resources

The ACT 1 interconnection architecture uses 22 horizontal tracks per channel for signal routing with three tracks dedicated to VDD, GND, and the global clock (GCLK), making a total of 25 tracks per channel. Horizontal segments vary in length from four columns of Logic Modules to the entire row of modules (Actel calls these long segments **long lines**).

Four Logic Module inputs are available to the channel below the Logic Module and four inputs to the channel above the Logic Module. Thus eight vertical tracks per Logic Module are available for inputs (four from the Logic Module above the channel and four from the Logic Module below). These connections are the **input stubs**.

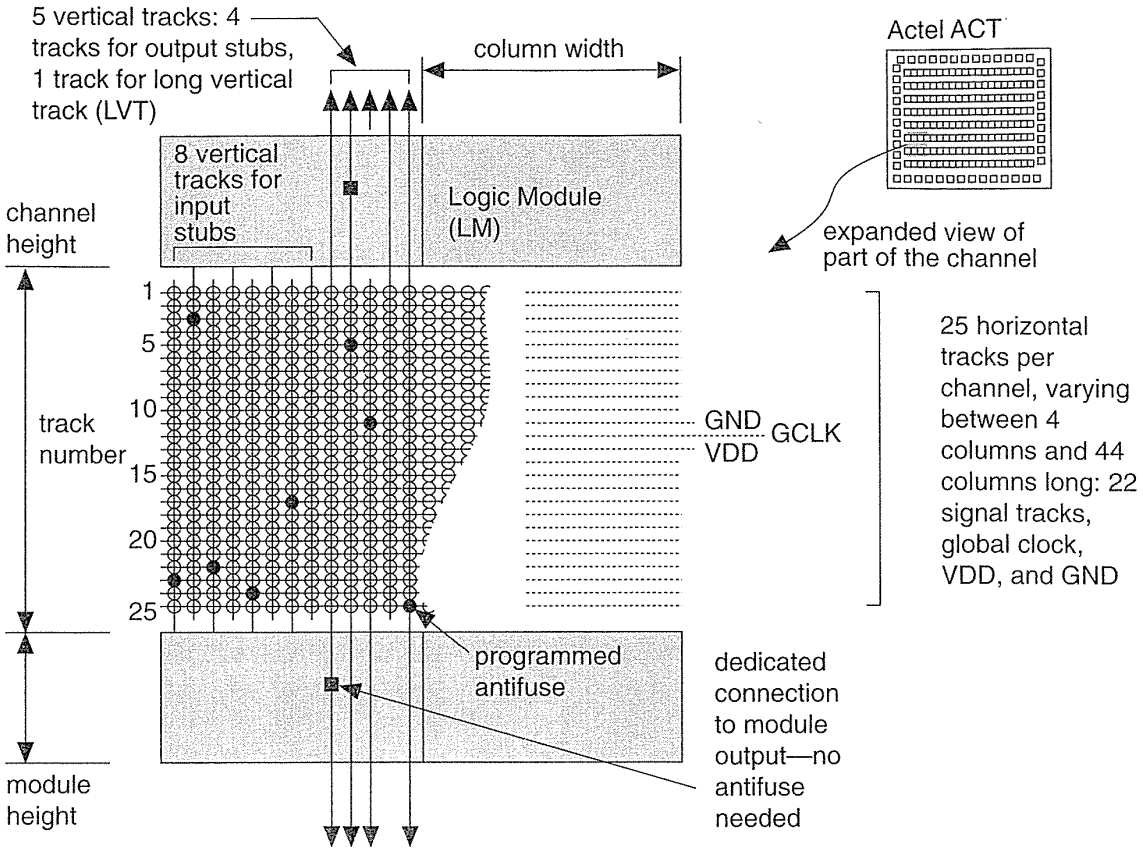


FIGURE 7.2 ACT 1 horizontal and vertical channel architecture. (*Source:* Actel.)

The single Logic Module output connects to a vertical track that extends across the two channels above the module and across the two channels below the module. This is the **output stub**. Thus module outputs use four vertical tracks per module (counting two tracks from the modules below, and two tracks from the modules above each channel). One vertical track per column is a **long vertical track** (LVT) that spans the entire height of the chip (the 1020 contains some segmented LVTs). There are thus a total of 13 vertical tracks per column in the ACT 1 architecture (eight for inputs, four for outputs, and one for an LVT).

Table 7.1 shows the routing resources for both the ACT 1 and ACT 2 families. The last two columns show the total number of antifuses (including antifuses in the I/O cells) on each chip and the total number of antifuses assuming the wiring channels are **fully populated** with antifuses (an antifuse at every horizontal and vertical interconnect intersection). The ACT 1 devices are very nearly fully populated.

TABLE 7.1 Actel FPGA routing resources.

	Horizontal tracks per channel, H	Vertical tracks per column, V	Rows, R	Columns, C	Total antifuses on each chip	H×V×R×C
A1010	22	13	8	44	112,000	100,672
A1020	22	13	14	44	186,000	176,176
A1225A	36	15	13	46	250,000	322,920
A1240A	36	15	14	62	400,000	468,720
A1280A	36	15	18	82	750,000	797,040

If the Logic Module at the end of a net is less than two rows away from the driver module, a connection requires two antifuses, a vertical track, and two horizontal segments. If the modules are more than two rows apart, a connection between them will require a long vertical track together with another vertical track (the output stub) and two horizontal tracks. To connect these tracks will require a total of four antifuses in series and this will add delay due to the resistance of the antifuses. To examine the extent of this delay problem we need some help from the analysis of RC networks.

7.1.2 Elmore's Constant

Figure 7.3 shows an **RC tree**—representing a net with a **fanout** of two. We shall assume that all nodes are initially charged to $V_{DD} = 1$ V, and that we short node 0 to ground, so $V_0 = 0$ V, at time $t = 0$ sec. We need to find the node voltages, V_1 to V_4, as a function of time. A similar problem arose in the design of wideband vacuum tube distributed amplifiers in the 1940s. Elmore found a measure of delay that we can use today [Rubenstein, Penfield, and Horowitz, 1983].

The current in branch k of the network is

$$i_k = -C_k \frac{dV_k}{dt}. \tag{7.1}$$

The linear superposition of the branch currents gives the voltage at node i as

$$V_i = -\sum_{k=1}^{n} R_{ki} C_k \frac{dV_k}{dt}, \tag{7.2}$$

where R_{ki} is the resistance of the path to V_0 (ground in this case) shared by node k and node i. So, for example, $R_{24} = R_1$, $R_{22} = R_1 + R_2$, and $R_{31} = R_1$.

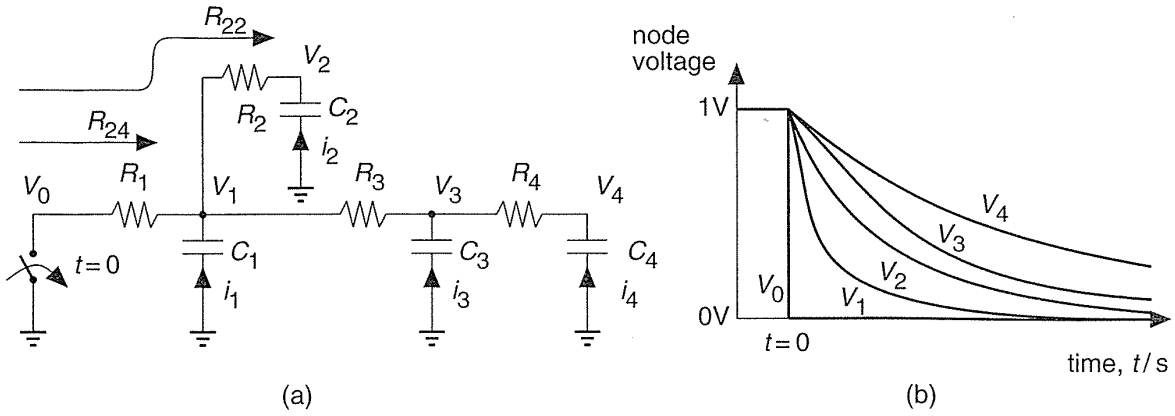


FIGURE 7.3 Measuring the delay of a net. (a) An RC tree. (b) The waveforms as a result of closing the switch at $t = 0$.

Unfortunately, Eq. 7.2 is a complicated set of coupled equations that we cannot easily solve. We know the node voltages have different values at each point in time, but, since the waveforms are similar, let us assume the slopes (the time derivatives) of the waveforms are related to each other. Suppose we express the slope of node voltage V_k as a constant, a_k, times the slope of V_i,

$$\frac{dV_k}{dt} = a_k \frac{dV_i}{dt}. \tag{7.3}$$

Consider the following measure of the error, E, of our approximation:

$$E = \sum_{k=1}^{n} R_{ki} C_k \int_0^{\infty} a_k \frac{dV_i}{dt} - \frac{dV_k}{dt} dt. \tag{7.4}$$

The error, E, is a minimum when $a_k = 1$ since initially $V_i(t=0) = V_k(t=0) = 1$ V (we normalized the voltages) and $V_i(t=\infty) = V_k(t=\infty) = 0$.

Now we can rewrite Eq. 7.2, setting $a_k = 1$, as follows:

$$V_i = -\sum_{k=1}^{n} R_{ki} C_k \frac{dV_i}{dt}. \tag{7.5}$$

This is a linear first-order differential equation with the following solution:

$$V_i(t) = e^{-t/\tau_{Di}}; \quad \tau_{Di} = \sum_{k=1}^{n} R_{ki} C_k. \tag{7.6}$$

The time constant τ_{Di} is often called the **Elmore delay** and is different for each node. We shall refer to τ_{Di} as the **Elmore time constant** to remind us that, if we approximate V_i by an exponential waveform, the delay of the RC tree using 0.35/0.65 trip points is approximately τ_{Di} seconds.

7.1.3 RC Delay in Antifuse Connections

Suppose a single antifuse, with resistance R_1, connects to a wire segment with parasitic capacitance C_1. Then a connection employing a single antifuse will delay the signal passing along that connection by approximately one time constant, or R_1C_1 seconds. If we have more than one antifuse, we need to use the Elmore time constant to estimate the interconnect delay.

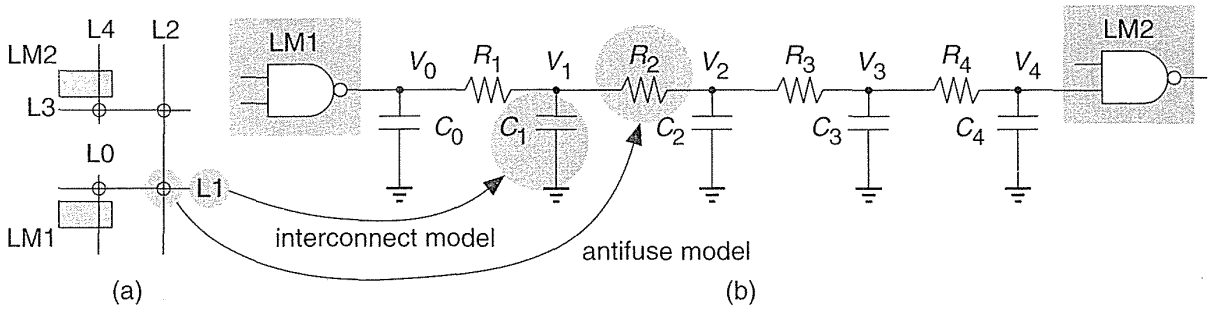


FIGURE 7.4 Actel routing model. (a) A four-antifuse connection. L0 is an output stub, L1 and L3 are horizontal tracks, L2 is a long vertical track (LVT), and L4 is an input stub. (b) An RC-tree model. Each antifuse is modeled by a resistance and each interconnect segment is modeled by a capacitance.

For example, suppose we have the four-antifuse connection shown in Figure 7.4. Then, from Eq. 7.6,

$$\begin{aligned}\tau_{D4} &= R_{14}C_1 + R_{24}C_2 + R_{34}C_3 + R_{44}C_4 \\ &= (R_1 + R_2 + R_3 + R_4)C_4 + (R_1 + R_2 + R_3)C_3 + (R_1 + R_2)C_2 + R_1C_1.\end{aligned}$$

If all the antifuse resistances are approximately equal (a reasonably good assumption) and the antifuse resistance is much larger than the resistance of any of the metal lines, L1–L5, shown in Figure 7.4 (a very good assumption) then $R_1 = R_2 = R_3 = R_4 = R$, and the Elmore time constant is

$$\tau_{D4} = 4RC_4 + 3RC_3 + 2RC_2 + RC_1. \tag{7.7}$$

Suppose now that the capacitance of each interconnect segment (including all the antifuses and programming transistors that may be attached) is approximately constant, and equal to C. A connection with two antifuses will generate a $3RC$ time constant, a connection with three antifuses a $6RC$ time constant, and a connection with four antifuses gives a $10RC$ time constant. This analysis is disturbing—it says that the interconnect delay grows quadratically ($\propto n^2$) as we increase the interconnect length and the number of antifuses, n. The situation is worse when the intermediate wire segments have larger capacitance than that of the short input stubs and output stubs. Unfortunately, this is the situation in an Actel FPGA where the horizontal and vertical segments in a connection may be quite long.

7.1.4 Antifuse Parasitic Capacitance

We can determine the number of antifuses connected to the horizontal and vertical lines for the Actel architecture. Each column contains 13 vertical signal tracks and each channel contains 25 horizontal tracks (22 of these are used for signals). Thus, assuming the channels are fully populated with antifuses,

- An input stub (1 channel) connects to 25 antifuses.
- An output stub (4 channels) connects to 100 (25×4) antifuses.
- An LVT (1010, 8 channels) connects to 200 (25×8) antifuses.
- An LVT (1020, 14 channels) connects to 350 (25×14) antifuses.
- A four-column horizontal track connects to 52 (13×4) antifuses.
- A 44-column horizontal track connects to 572 (13×44) antifuses.

A connection to the diffusion of an Actel antifuse has a parasitic capacitance due to the diffusion junction. The polysilicon of the antifuse has a parasitic capacitance due to the thin oxide. These capacitances are approximately equal. For a 2 μm CMOS process the capacitance to ground of the diffusion is 200 to 300 $aF\mu m^{-2}$ (area component) and 400 to 550 $aF\mu m^{-1}$ (perimeter component). Thus, including both area and perimeter effects, a 16 μm^2 diffusion contact (consisting of a 2 μm by 2 μm opening plus the required overlap) has a parasitic capacitance of 10–14 fF. If we assume an antifuse has a parasitic capacitance of approximately 10 fF in a 1.0 or 1.2 μm process, we can calculate the parasitic capacitances shown in Table 7.2.

We can use the figures from Table 7.2 to estimate the interconnect delays. First we calculate the following resistance and capacitance values:

1. The antifuse resistance is assumed to be $R = 0.5\ k\Omega$.
2. $C_0 = 1.2$ pF is the sum of the gate output capacitance (which we shall neglect) and the output stub capacitance (1.0 pF due to antifuses, 0.2 pF due to metal). The contribution from this term is zero in our calculation because we have neglected the pull resistance of the driving gate.
3. $C_1 = C_3 = 0.59$ pF (0.52 pF due to antifuses, 0.07 pF due to metal) corresponding to a minimum-length horizontal track.

TABLE 7.2 Actel interconnect parameters.

Parameter	A1010/A1020	A1010B/A1020B
Technology	2.0 μm, λ = 1.0 μm	1.2 μm, λ = 0.6 μm
Die height (A1010)	240 mil	144 mil
Die width (A1010)	360 mil	216 mil
Die area (A1010)	86,400 mil^2 = 56 $M\lambda^2$	31,104 mil^2 = 56 $M\lambda^2$
Logic Module (LM) height (Y1)	180 μm = 180 λ	108 μm = 180 λ
LM width (X)	150 μm = 150 λ	90 μm = 150 λ
LM area (X×Y1)	27,000 μm^2 = 27 $k\lambda^2$	9,720 μm^2 = 27 $k\lambda^2$
Channel height (Y2)	25 tracks = 287 μm	25 tracks = 170 μm
Channel area per LM (X×Y2)	43,050 μm^2 = 43 $k\lambda^2$	15,300 μm^2 = 43 $k\lambda^2$
LM and routing area (X×Y1 + X×Y2)	70,000 μm^2 = 70 $k\lambda^2$	25,000 μm^2 = 70 $k\lambda^2$
Antifuse capacitance	—	10 fF
Metal capacitance	0.2 $pFmm^{-1}$	0.2 $pFmm^{-1}$
Output stub length (spans 3 LMs + 4 channels)	4 channels = 1688 μm	4 channels = 1012 μm
Output stub metal capacitance	0.34 pF	0.20 pF
Output stub antifuse connections	100	100
Output stub antifuse capacitance	—	1.0 pF
Horiz. track length	4–44 cols. = 600–6600 μm	4–44 cols. = 360–3960 μm
Horiz. track metal capacitance	0.1–1.3 pF	0.07–0.8 pF
Horiz. track antifuse connections	52–572 antifuses	52–572 antifuses
Horiz. track antifuse capacitance	—	0.52–5.72 pF
Long vertical track (LVT)	8–14 channels = 3760–6580 μm	8–14 channels = 2240–3920 μm
LVT metal capacitance	0.08–0.13 pF	0.45–0.8 pF
LVT track antifuse connections	200–350 antifuses	200–350 antifuses
LVT track antifuse capacitance		2–3.5 pF
Antifuse resistance (ACT 1)		0.5 kΩ (typ.), 0.7 kΩ (max.)

4. $C_2 = 4.3$ pF (3.5 pF due to antifuses, 0.8 pF due to metal) corresponding to a LVT in a 1020B.
5. The estimated input capacitance of a gate is $C_4 = 0.02$ pF (the exact value will depend on which input of a Logic Module we connect to).

From Eq. 7.7, the Elmore time constant for a four-antifuse connection is

$$\tau_{D4} = (4)\,(0.5)\,(0.02) + (3)\,(0.5)\,(0.59) + (2)\,(0.5)\,(4.3) + (0.5)\,(0.59) \\ = 5.52\text{ns}. \tag{7.8}$$

This matches delays obtained from the Actel delay calculator. For example, an LVT adds between 5–10 ns delay in an ACT 1 FPGA (6–12 ns for ACT 2, and 4–14 ns for ACT 3). The LVT connection is about the slowest connection that we can make in an ACT array. Normally less than 10 percent of all connections need to use an LVT and we see why Actel takes great care to make sure that this is the case.

7.1.5 ACT 2 and ACT 3 Interconnect

The ACT 1 architecture uses two antifuses for routing nearby modules, three antifuses to join horizontal segments, and four antifuses to use a horizontal or vertical long track. The ACT 2 and ACT 3 architectures use increased interconnect resources over the ACT 1 device that we have described. This reduces further the number of connections that need more than two antifuses. Delay is also reduced by decreasing the population of antifuses in the channels, and by decreasing the antifuse resistance of certain critical antifuses (by increasing the programming current).

The **channel density** is the absolute minimum number of tracks needed in a channel to make a given set of connections (see Section 17.2.2, "Measurement of Channel Density"). Software to route connections using channeled routing is so efficient that, given complete freedom in location of wires, a channel router can usually complete the connections with the number of tracks equal or close to the theoretical minimum, the channel density. Actel's studies on segmented channel routing have shown that increasing the number of horizontal tracks slightly (by approximately 10 percent) above density can lead to very high routing completion rates.

The ACT 2 devices have 36 horizontal tracks per channel rather than the 22 available in the ACT 1 architecture. Horizontal track segments in an ACT 3 device range from a module pair to the full channel length. Vertical tracks are: input (with a two channel span: one up, one down); output (with a four-channel span: two up, two down); and long (LVT). Four LVTs are shared by each column pair. The ACT 2/3 Logic Modules can accept five inputs, rather than four inputs for the ACT 1 modules, and thus the ACT 2/3 Logic Modules need an extra two vertical tracks per channel. The number of tracks per column thus increases from 13 to 15 in the ACT 2/3 architecture.

The greatest challenge facing the Actel FPGA architects is the resistance of the polysilicon-diffusion antifuse. The nominal antifuse resistance in the ACT 1–2 1–2 μm processes (with a 5 mA programming current) is approximately 500 Ω and, in the worst case, may be as high as 700 Ω. The high resistance severely limits the number of antifuses in a connection. The ACT 2/3 devices assign a special antifuse to each output allowing a direct connection to an LVT. This reduces the number of antifuses in a connection using an LVT to three. This type of antifuse (a **fast fuse**) is

blown at a higher current than the other antifuses to give them about half the nominal resistance (about 0.25 kΩ for ACT 2) of a normal antifuse. The nominal antifuse resistance is reduced further in the ACT 3 (using a 0.8 μm process) to 200 Ω (Actel does not state whether this value is for a normal or fast fuse). However, it is the worst-case antifuse resistance that will determine the worst-case performance.

7.2 Xilinx LCA

Figure 7.5 shows the hierarchical Xilinx LCA interconnect architecture.

- The **vertical lines** and **horizontal lines** run between CLBs.
- The **general-purpose interconnect** joins **switch boxes** (also known as **magic boxes** or **switching matrices**).
- The **long lines** run across the entire chip. It is possible to form internal buses using long lines and the three-state buffers that are next to each CLB.
- The **direct connections** (not used on the XC4000) bypass the switch matrices and directly connect adjacent CLBs.
- The **Programmable Interconnection Points** (**PIPs**) are programmable pass transistors that connect the CLB inputs and outputs to the routing network.
- The **bidirectional (BIDI) interconnect buffers** restore the logic level and logic strength on long interconnect paths.

Table 7.3 shows the interconnect data for an XC3020, a typical Xilinx LCA FPGA, that uses two-level metal interconnect. Figure 7.6 shows the switching matrix. Programming a switch matrix allows a number of different connections between the general-purpose interconnect.

In Figure 7.6 (d), (g), and (h):

- $C_1 = 3C_{P1} + 3C_{P2} + 0.5C_{LX}$ is the parasitic capacitance due to the switch matrix and PIPs (F4, C4, G4) for CLB1, and half of the line capacitance for the double-length line adjacent to CLB1.
- C_{P1} and R_{P1} are the switching-matrix parasitic capacitance and resistance.
- C_{P2} and R_{P2} are the parasitic capacitance and resistance for the PIP connecting YQ of CLB1 and F4 of CLB3.
- $C_2 = 0.5C_{LX} + C_{LX}$ accounts for half of the line adjacent to CLB1 and the line adjacent to CLB2.
- $C_3 = 0.5C_{LX}$ accounts for half of the line adjacent to CLB3.
- $C_4 = 0.5C_{LX} + 3C_{P2} + C_{LX} + 3C_{P1}$ accounts for half of the line adjacent to CLB3, the PIPs of CLB3 (C4, G4, YQ), and the rest of the line and switch matrix capacitance following CLB3.

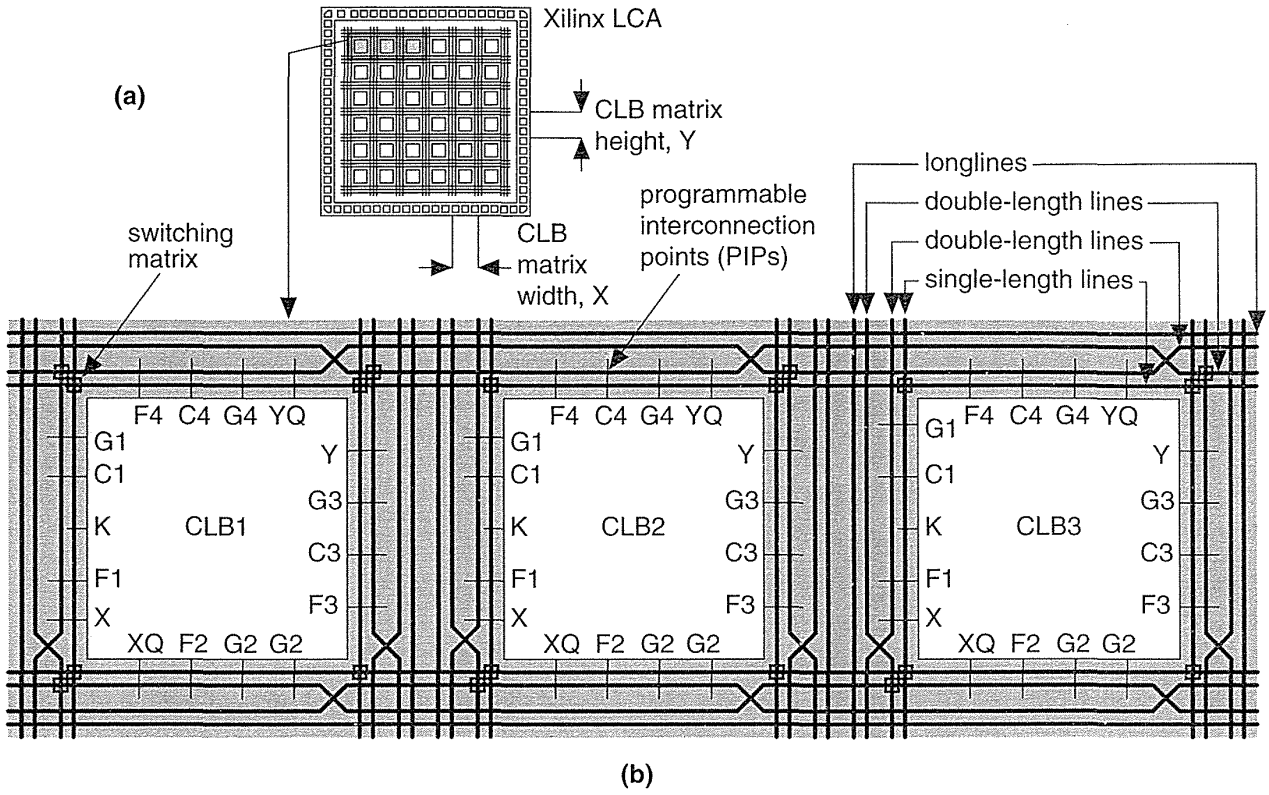


FIGURE 7.5 Xilinx LCA interconnect. (a) The LCA architecture (notice the matrix element size is larger than a CLB). (b) A simplified representation of the interconnect resources. Each of the lines is a bus.

We can determine Elmore's time constant for the connection shown in Figure 7.6 as

$$\tau_D = R_{P2}(C_{P2} + C_2 + 3C_{P1}) + (R_{P2} + R_{P1})(3C_{P1} + C_3 + C_{P2}) + (2R_{P2} + R_{P1})(C_{P2} + C_4). \quad (7.9)$$

If $R_{P1} = R_{P2}$, and $C_{P1} = C_{P2}$, then

$$\tau_D = (15 + 21) R_P C_P + (1.5 + 1 + 4.5) R_P C_{LX}. \quad (7.10)$$

We need to know the pass-transistor resistance R_P. For example, suppose $R_P = 1\text{k}\Omega$. If $k_n' = 50\,\mu\text{AV}^{-2}$, then (with $V_{tn} = 0.65$ V and $V_{DD} = 3.3$ V)

$$W/L = \frac{1}{k_n' R_P (V_{DD} - V_{tn})} = \frac{1}{(50 \times 10^{-6})(1 \times 10^{3})(3.3 - 0.65)} = 7.5. \quad (7.11)$$

TABLE 7.3 XC3000 interconnect parameters.

Parameter	XC3020
Technology	1.0 μm, λ = 0.5 μm
Die height	220 mil
Die width	180 mil
Die area	39,600 mil^2 = 102 Mλ^2
CLB matrix height (Y)	480 μm = 960 λ
CLB matrix width (X)	370 μm = 740 λ
CLB matrix area (X×Y)	17,600 μm^2 = 710 kλ^2
Matrix transistor resistance, R_{P1}	0.5–1kΩ
Matrix transistor parasitic capacitance, C_{P1}	0.01–0.02 pF
PIP transistor resistance, R_{P2}	0.5–1kΩ
PIP transistor parasitic capacitance, C_{P2}	0.01–0.02 pF
Single-length line (X, Y)	370 μm, 480 μm
Single-length line capacitance: C_{LX}, C_{LY}	0.075 pF, 0.1 pF
Horizontal Longline (8X)	8 cols. = 2960 μm
Horizontal Longline metal capacitance, C_{LL}	0.6 pF

If L = 1 μm, both source and drain areas are 7.5 μm long and approximately 3 μm wide (determined by diffusion overlap of contact, contact width, and contact-to-gate spacing, rules 6.1a + 6.2a + 6.4a = 5.5 λ in Table 2.7). Both drain and source areas are thus 23 μm^2 and the sidewall perimeters are 14 μm (excluding the sidewall facing the channel). If we have a diffusion capacitance of 140 aFμm^{-2} (area) and 500 aFμm^{-1} (perimeter), typical values for a 1.0 μm process, the parasitic source and drain capacitance is

$$C_P = (140 \times 10^{-18})\ (23) + (500 \times 10^{-18})\ (14) = 1.022 \times 10^{-14}\text{F}. \quad (7.12)$$

If we assume $C_P = 0.01$ pF and $C_{LX} = 0.075$ pF (Table 7.3),

$$\tau_D = (36)\ (1)\ (0.01) + (7)\ (1)\ (0.075) = 0.885 \text{ ns}. \quad (7.13)$$

A delay of approximately 1 ns agrees with the typical values from the XACT delay calculator and is about the fastest connection we can make between two CLBs.

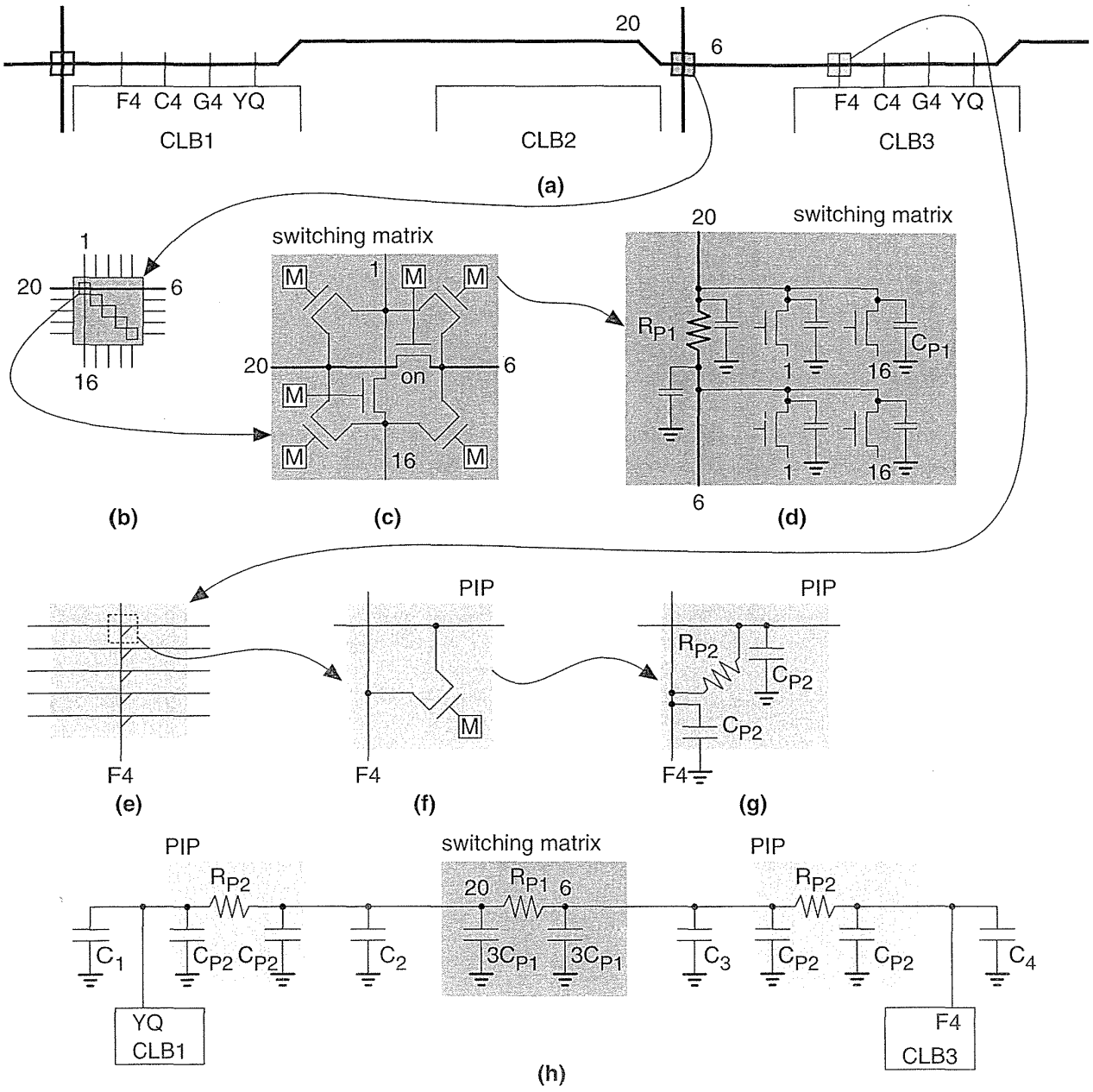


FIGURE 7.6 Components of interconnect delay in a Xilinx LCA array. (a) A portion of the interconnect around the CLBs. (b) A switching matrix. (c) A detailed view inside the switching matrix showing the pass-transistor arrangement. (d) The equivalent circuit for the connection between nets 6 and 20 using the matrix. (e) A view of the interconnect at a Programmable Interconnection Point (PIP). (f) and (g) The equivalent schematic of a PIP connection. (h) The complete RC delay path.

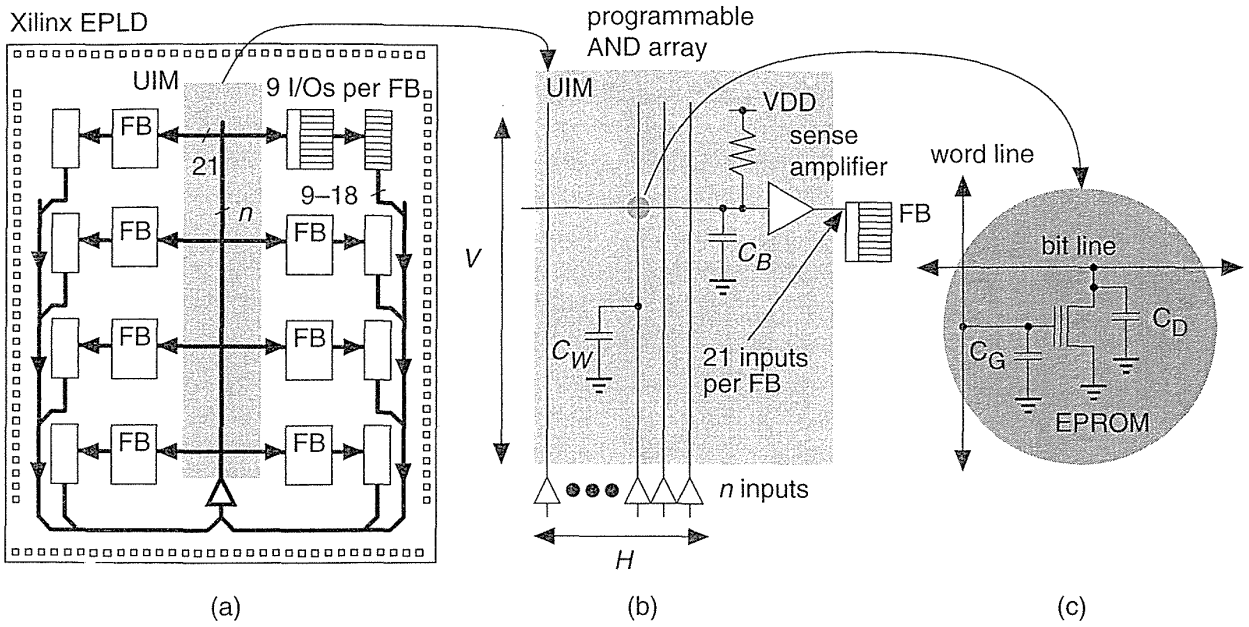


FIGURE 7.7 The Xilinx EPLD UIM (Universal Interconnection Module). (a) A simplified block diagram of the UIM. The UIM bus width, *n*, varies from 68 (XC7236) to 198 (XC73108). (b) The UIM is actually a large programmable AND array. (c) The parasitic capacitance of the EPROM cell.

7.3 Xilinx EPLD

The Xilinx EPLD family uses an interconnect bus known as **Universal Interconnection Module (UIM)** to distribute signals within the FPGA. The UIM, shown in Figure 7.7, is a programmable AND array with constant delay from any input to any output. In Figure 7.7:

- C_G is the fixed gate capacitance of the EPROM device.
- C_D is the fixed drain parasitic capacitance of the EPROM device.
- C_B is the variable horizontal bus ("bit" line) capacitance.
- C_W is the variable vertical bus ("word" line) capacitance.

Figure 7.7 shows the UIM has 21 output connections to each FB.[1] Thus the XC7272 UIM (with a 4×2 array of eight FBs as shown in Figure 7.7) has 168 (8×21) output connections. Most (but not all) of the nine I/O cells attached to each FB have two input connections to the UIM, one from a chip input and one feedback

[1] 1994 data book p. 3-62 and p. 3-78.

from the macrocell output. For example, the XC7272 has 18 I/O cells that are outputs only and thus have only one connection to the UIM, so $n = (18 \times 8) - 18 = 126$ input connections. Now we can calculate the number of tracks in the UIM: the XC7272, for example, has $H = 126$ tracks and $V = 168/2 = 84$ tracks. The actual physical height, V, of the UIM is determined by the size of the FBs, and is close to the die height.

The UIM ranges in size with the number of FBs. For the smallest XC7236 (with a 2×2 array of four FBs), the UIM has $n = 68$ inputs and 84 outputs. For the XC73108 (with a 6×2 array of 12 FBs), the UIM has $n = 198$ inputs. The UIM is a large array with large parasitic capacitance; it employs a highly optimized structure that uses EPROM devices and a **sense amplifier** at each output. The signal swing on the UIM uses less than the full $V_{DD} = 5$ V to reduce the interconnect delay.

7.4 Altera MAX 5000 and 7000

Altera MAX 5000 devices (except the EPM5032, which has only one LAB) and all MAX 7000 devices use a **Programmable Interconnect Array (PIA)**, shown in Figure 7.8. The PIA is a cross-point switch for logic signals traveling between LABs. The advantages of this architecture (which uses a fixed number of connections) over programmable interconnection schemes (which use a variable number of

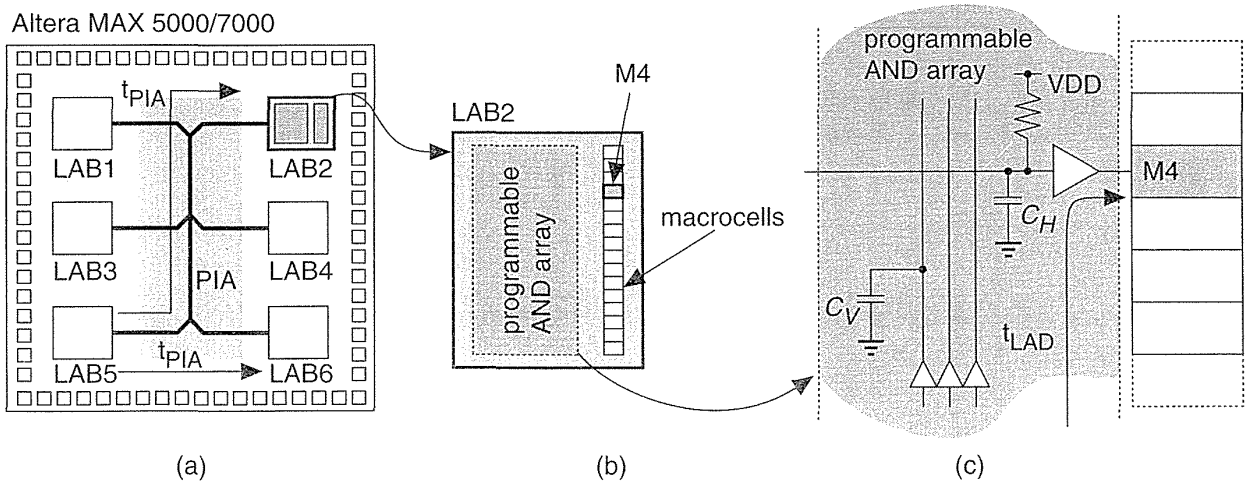


FIGURE 7.8 A simplified block diagram of the Altera MAX interconnect scheme. (a) The PIA (Programmable Interconnect Array) is deterministic—delay is independent of the path length. (b) Each LAB (Logic Array Block) contains a programmable AND array. (c) Interconnect timing within a LAB is also fixed.

connections) is the fixed routing delay. An additional benefit of the simpler nature of a large regular interconnect structure is the simplification and improved speed of the placement and routing software.

Figure 7.8(a) illustrates that the delay between any two LABs, t_{PIA}, is fixed. The delay between LAB1 and LAB2 (which are adjacent) is the same as the delay between LAB1 and LAB6 (on opposite corners of the die). It may seem rather strange to slow down all connections to the speed of the longest possible connection—a large penalty to pay to achieve a deterministic architecture. However, it gives Altera the opportunity to highly optimize all of the connections since they are completely fixed.

7.5 Altera MAX 9000

Figure 7.9 shows the Altera MAX 9000 interconnect architecture. The size of the MAX 9000 LAB arrays varies between 4 × 5 (rows × columns) for the EPM9320 and 7 × 5 for the EPM9560. The MAX 9000 is an extremely coarse-grained architecture, typical of complex PLDs, but the LABs themselves have a finer structure. Sometimes we say that complex PLDs with arrays (LABs in the Altera MAX family) that are themselves arrays (of macrocells) have a **dual-grain architecture**.

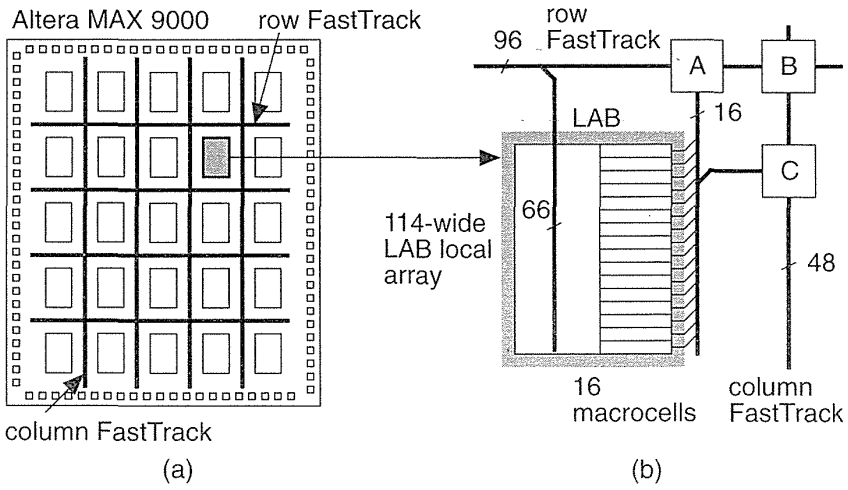


FIGURE 7.9 The Altera MAX 9000 interconnect scheme. (a) A 4 × 5 array of Logic Array Blocks (LABs), the same size as the EMP9400 chip. (b) A simplified block diagram of the interconnect architecture showing the connection of the FastTrack buses to a LAB.

In Figure 7.9(b), boxes A, B, and C represent the interconnection between the FastTrack buses and the 16 macrocells in each LAB:

- Box A connects a macrocell to one row channel.
- Box B connects three column channels to two row channels.
- Box C connects a macrocell to three column channels.

7.6 Altera FLEX

Figure 7.10 shows the interconnect used in the Altera FLEX family of complex PLDs. Altera refers to the FLEX interconnect and MAX 9000 interconnect by the same name, FastTrack, but the two are different because the granularity of the logic cell arrays is different. The FLEX architecture is of finer grain than the MAX arrays—because of the difference in programming technology. The FLEX horizontal interconnect is much denser (at 168 channels per row) than the vertical interconnect (16 channels per column), creating an aspect ratio for the interconnect of over 10:1 (168:16). This imbalance is partly due to the aspect ratio of the die, the array, and the aspect ratio of the basic logic cell, the LAB.

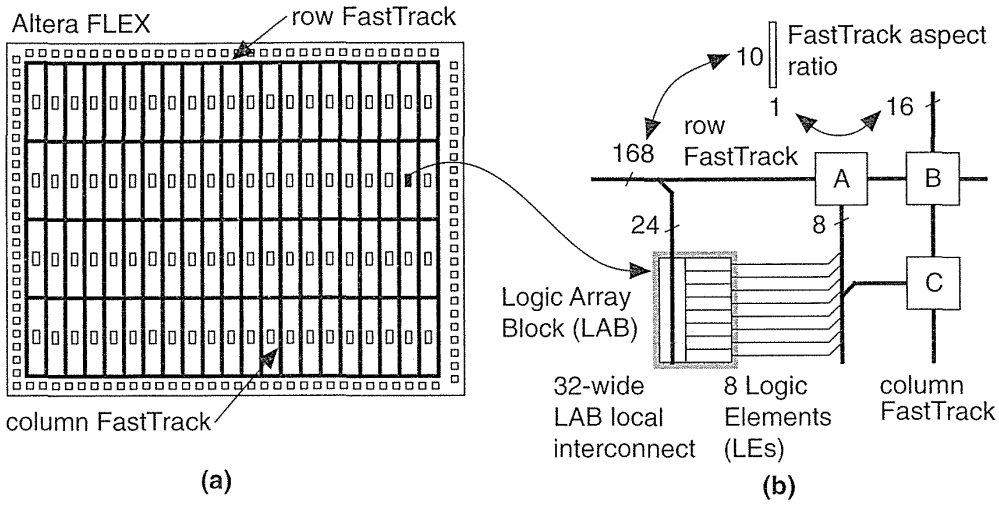


FIGURE 7.10 The Altera FLEX interconnect scheme. (a) The row and column FastTrack interconnect. The chip shown, with 4 rows × 21 columns, is the same size as the EPF8820. (b) A simplified diagram of the interconnect architecture showing the connections between the FastTrack buses and a LAB. Boxes A, B, and C represent the bus-to-bus connections.

As an example, the EPF8820 has 4 rows and 21 columns of LABs (Figure 7.10a). Ignoring, for simplicity's sake, what happens at the edge of the die we can total the routing channels as follows:

- Horizontal channels = 4 rows × 168 channels/row = 672 channels.
- Vertical channels = 21 rows × 16 channels/row = 336 channels.

It appears that there is still approximately twice (672:336) as much interconnect capacity in the horizontal direction as the vertical. If we look inside the boxes A, B, and C in Figure 7.10(b) we see that for individual lines on each bus:

- Box A connects an LE to two row channels.
- Box B connects two column channels to a row channel.
- Box C connects an LE to two column channels.

There is some dependence between boxes A and B since they contain MUXes rather than direct connections, but essentially there are twice as many connections to the column FastTrack as the row FastTrack, thus restoring the balance in interconnect capacity.

7.7 Summary

The RC product of the parasitic elements of an antifuse and a pass transistor are not too different. However, an SRAM cell is much larger than an antifuse which leads to coarser interconnect architectures for SRAM-based programmable ASICs. The EPROM device lends itself to large wired-logic structures. These differences in programming technology lead to different architectures:

- The antifuse FPGA architectures are dense and regular.
- The SRAM architectures contain nested structures of interconnect resources.
- The complex PLD architectures use long interconnect lines but achieve deterministic routing.

Table 7.4 is a look-up table for Tables 7.5 and 7.6, which summarize the features of the logic cells used by the various FPGA vendors.

TABLE 7.4 I/O Cell Tables.

Table	Programmable ASIC family
Table 7.5	Actel (ACT 1) Xilinx (XC3000) Actel (ACT 2) Xilinx (XC4000) Altera MAX (EPM 5000) Xilinx EPLD (XC7200/7300) Actel (ACT 3) QuickLogic (pASIC 1) Crosspoint (CP20K) Altera MAX (EPM 7000) Atmel (AT6000) Xilinx LCA (XC5200)
Table 7.6	Xilinx (XC8100) Lucent ORCA (2C) Altera FLEX (8000/10k) AMD MACH 5 Actel 3200DX Altera MAX (EPM 9000)

TABLE 7.5 Programmable ASIC interconnect.

	Actel (ACT 1)	Xilinx (XC3000)	Actel (ACT 2)	Xilinx (XC4000)
Interconnect between logic cells (tracks = trks)	Channeled array with segmented routing, long lines: 25 trks/ch. (horiz.); 13 trks/ch. (vert.); < 4 antifuses/path	Switch box, PIPs (Programmable Interconnect Points), 3-state internal bus, and long lines	Channeled array with segmented routing, long lines: 36 trks/ch. (horiz.); 15 trks/ch. (vert.); < 4 antifuses/path	Switch box, PIPs (Programmable Interconnect Points), 3-state internal bus, and long lines
Interconnect delay	Variable	Variable	Variable	Variable
Interconnect inside logic cells	Poly–diffusion antifuse	32-bit SRAM LUT	Poly–diffusion antifuse	32-bit SRAM LUT

	Altera (MAX 5000)	Xilinx EPLD	QuickLogic (pASIC 1)	Actel (ACT 3)
Interconnect between logic cells	Cross-bar PIA (Programmable Interconnect Architecture) using EPROM programmable-AND array	UIM (Universal Interconnect Matrix) using EPROM programmable-AND array	Programmable fully populated antifuse matrix	Channeled array with segmented routing, long lines: <4 antifuses/path
Interconnect delay	Fixed	Fixed	Variable	Variable
Interconnect inside logic cells	EPROM	EPROM	Metal–metal antifuse	Poly–diffusion antifuse

	Crosspoint (CP20K)	Altera MAX (MAX 7000)	Atmel (AT6000)	Xilinx LCA (XC5200)
Interconnect between logic cells	Programmable highly interconnected matrix	Fixed cross-bar PIA (Programmable Interconnect Architecture)	Programmable regular, local, and express bus scheme with line repeaters	Switch box, PIPs (Programmable Interconnect Points), 3-state internal bus, and long lines
Interconnect delay	Variable	Fixed	Variable	Variable
Interconnect inside logic cells	Metal–metal antifuse	EEPROM	SRAM	16-bit SRAM LUT

TABLE 7.6 Programmable ASIC interconnect (continued).

	Xilinx (XC8100)	Lucent ORCA 2C	Altera FLEX 8000/10k
Interconnect between logic cells	Channeled array with segmented routing, long lines. Programmable fully populated antifuse matrix.	Switch box, SRAM programmable interconnect, 3-state internal bus, and long lines	Row and column FastTrack between LABs
Interconnect delay	Variable	Variable	Fixed with small variation in delay in row FastTrack
Interconnect inside logic cells	Antifuse	SRAM LUTs and MUXs	LAB local interconnect between LEs. 16-bit SRAM LUT in LE.
	AMD MACH 5	**Actel 3200DX**	**Altera MAX 9000**
Interconnect between logic cells	EPROM programmable array	Channeled gate array with segmented routing, long lines	Row and column FastTrack between LABs
Interconnect delay	Fixed	Variable	Fixed
Interconnect inside logic cells	EPROM	Poly–diffusion antifuse	Programmable AND array inside LAB, EEPROM MUXes

The key points covered in this chapter are:

- The difference between deterministic and nondeterministic interconnect
- Estimating interconnect delay
- Elmore's constant

Next, in Chapter 8, we shall cover the software you need to design with the various FPGA families and explain how FPGAs are programmed.

7.8 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

7.1 (*Xilinx interconnect, 120 min.)

a. Write a minitutorial (one or two pages) explaining what you need to know to run and use the XACT delay calculator. Explain how to choose the part, set the display preferences, make connections to CLBs and the interconnect, and obtain timing figures.

b. Use the XACT editor to determine typical delays using the longlines, a switch matrix, the PIPs, and BIDI buffers (see the Xilinx data book for more detailed explanations of the interconnect structure). Draw six different typical paths using these elements and show the components of delay. Include screen shots showing the layout of the paths and cells with detailed explanations of the figures.

c. Construct a path using the TBUFs, the three-state buffers, driving a longline (do not forget the pull-up). Show the XACT calculated delay for your path and explain the number from data book parameters (list them and the page number from the data book).

d. Extend one simple path to the I/O and explain the input and output timing, again using the data book.

e. Include screen shots from the layout editor showing you example paths.

f. Bury all the ASCII (but not binary) files you used and the tools produced inside your report using "Hidden Text." Include explanations as to what these files are and which parts of the report they go with. This includes any schematic files, netlist files, and all files produced by the Xilinx tools. Use a separate directory for this problem and make a list in your report of all files (binary and ASCII) with explanations of what each file is.

7.2 (*Actel interconnect, 120 min.) Use the Actel chip editor to explore the properties of the interconnect scheme in a similar fashion to Problem 7.1 with the following changes: in part b make at least six different paths using various antifuse connections and explain the numbers from the delay calculator. Omit part c.

7.3 (*Altera MAX interconnect, 120 min.) Use the Altera tools to determine the properties of the MAX or FLEX interconnect in a similar fashion to Problem 7.1 with the following changes: In parts b and c construct at least six example circuits that show the various paths through the FastTrack or PIA chip-level interconnect, the local LAB array, the LAB, and the macrocells.

7.4 (**Custom ASICs, 120 min.)

a. Write a minitutorial (one or two pages) explaining how to run an ASIC tool (Compass/Mentor/Cadence/Tanner). Enter a simple circuit (using schematic entry or synthesis and cells from a cell library) and obtain a delay estimate.

b. Construct at least six example circuits that show various logic paths using various logic cells (for example: an inverter, a full adder).

c. Perform a timing simulation (either using a static timing verifier or using a logic simulator). Compare your results with those from a data book.

d. Extract the circuit to include the parasitic capacitances from layout in your circuit netlist and run a simulation to predict the delays.

e. Compare the results that include routing capacitance with the data book values for the logic cell delays and with the values predicted before routing.

f. Extend one simple path to the outputs of the chip by including I/O pads in your circuit and explain the input and output timing predictions.

g. Bury the ASCII files you used and the tools produced inside your report.

7.5 (**Actel stubs, 60 min.)

a. Which metal layers do you think Actel assigns to the horizontal and vertical interconnect in the ACT 1–3 architectures and why?

b. Why do the ACT 1–3 input stubs not extend over more than two channels above and below the Logic Modules, since this would reduce the need for LVTs?

c. The ACT 2 data sheet describes the output stubs as "twisted" (or interwoven) so that they occupy only four tracks. Show that the stubs occupy four vertical tracks whether they are twisted or not.

d. Suggest the real reason for the twisted stubs.

7.6 (A three-input NAND in ACT 1, 30 min.) The macros that require two ACT 1 modules include the three-input NAND (others include four-input NAND, AND, NOR).

a. What is the problem with trying to implement a three-input NAND gate using the Actel ACT 1 Logic Module?

b. Suggest a modification to the ACT 1 Logic Module that would allow the implementation of a three-input NAND using one of your new Logic Modules.

c. Can you think of a reason why Actel did not use your modification to its Logic Module design? *Hint:* The modification has to do with routing, and not the logic itself.

7.7 (*Actel architecture, 60 min.) This is a long but relatively straightforward problem that "reverse-engineers" the Actel architecture. If you measured the chip photo on the front of the April 1990 Actel data book, you would find the following:

1. Die height (scribe to scribe) = 170 mm.
2. Channel height = 8 mm (there are 7 full-height and 2 half-height channels).
3. Logic Module height = 5 mm (there are 8 rows of Logic Modules).
4. Column (Logic Module) width = 4.2 mm.

(The **scribe line** is an area at the edge of a die where a cut is made by a diamond saw when the dice are separated.) An Actel 1010 die in 2 μm technology is 240 mil high by 360 mil wide (p. 4-17 in the 1990 data book). Assuming these data book dimensions are scribe to scribe, calculate (**a**) the Logic Module height, (**b**) the channel height, and (**c**) the column (Logic Module) width.

Given that there are 25 tracks per horizontal channel, and 13 tracks per column in the vertical direction, calculate (**d**) the horizontal channel track spacing and (**e**) the vertical channel track spacing. (**f**) Using the fact that each output stub spans two channels above and below the Logic Module, calculate the height of the output stub.

We can now estimate the capacitance of the Logic Module stubs and interconnect. Assume the interconnect capacitance is $0.2\,\mathrm{pFmm}^{-1}$. (**g**) Calculate the capacitance of an output stub and an input stub. (**h**) Calculate the width and thus the capacitance of the horizontal tracks that are from four columns to 44 columns long.

You should not have to make any other assumptions to calculate these figures, but if you do, state them clearly. The figures you have calculated are summarized in Table 7.2.

7.8 (Xilinx bank shots, 20 min.) Figure 7.11 shows a magic box. Explain how to use a "bank shot" to enter one side of the box, bounce off another, and exit on a third side. What is the delay involved in this maneuver?

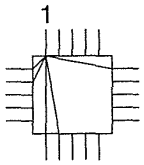


FIGURE 7.11 A Xilinx magic box showing one set of connections from connection 1 (Problem 7.8).

7.9 Bibliography

The paper by Greene et al. [1993] (reprinted in the 1994 Actel data book) is a good description of the Actel interconnect. The 1995 AT&T data book contains a very detailed account of the routing for the ORCA series of FPGAs, which is similar to the Xilinx LCA interconnect. You can learn a great deal about the details of the Lucent and Xilinx interconnect architecture from the AT&T data book. The Xilinx data book gives a good high-level overview of SRAM-based FPGA interconnect. The best way to learn about any FPGA interconnect is to use the software tools provided by the vendor. The Xilinx XACT editor that shows point-to-point routing delays on a graphical representation of the chip layout is an easy way to become familiar with the interconnect properties. The book by Brown et al. [1992] covers FPGA interconnect from a theoretical point of view, concentrating on routing for LUT based FPGAs, and also describes specialized routing algorithms for FPGAs.

7.10 References

Brown, S. D., et al. 1992. *Field-Programmable Gate Arrays*. Norwell, MA: Kluwer Academic, 206 p. ISBN 0-7923-9248-5. TK7872.L64F54. Contents: Introduction to FPGAs, Commercially Available FPGAs, Technology Mapping for FPGAs, Logic Block Architecture, Routing for FPGAs, Flexibility of FPGA Routing Resources, A Theoretical Model for FPGA Routing. Includes an introduction to commercially available FPGAs. The rest of the book

covers research on logic synthesis for FPGAs and FPGA architectures, concentrating on LUT-based architectures.

Greene, J., et al. 1993. "Antifuse field programmable gate arrays." *Proceedings of the IEEE,* vol. 81, no. 7, pp. 1042–1056, 1993. Review article describing the Actel FPGAs. (Included in the Actel 1994 data book.)

Rubenstein, J., P. Penfield, and M. A. Horowitz. 1983. "Signal delay in RC tree networks." *IEEE Transactions on CAD,* vol. CAD-2, no. 3, July 1983, pp. 202–211. Derives bounds for the response of RC networks excited by an input step voltage.

PROGRAMMABLE ASIC DESIGN SOFTWARE

8

There are five components of a programmable ASIC or FPGA: (1) the programming technology, (2) the basic logic cell, (3) the I/O cell, (4) the interconnect, and (5) the design software that allows you to program the ASIC. The design software is much more closely tied to the FPGA architecture than is the case for other types of ASICs.

8.1 Design Systems

The sequence of steps for FPGA design is similar to the sequence discussed in Section 1.2, "Design Flow." As for any ASIC a designer needs design-entry software, a cell library, and physical-design software. Each of the FPGA vendors sells **design kits** that include all the software and hardware that a designer needs. Many of these kits use design-entry software produced by a different company. Often designers buy that software from the FPGA vendor. This is called an **original equipment manufacturer** (**OEM**) arrangement—similar to buying a car with a stereo manufactured by an electronics company but labeled with the automobile company's name. Design entry uses cell libraries that are unique to each FPGA vendor. All of the FPGA vendors produce their own physical-design software so they can tune the algorithms to their own architecture.

Unfortunately, there are no standards in FPGA design. Thus, for example, Xilinx calls its 2:1 MUX an `M2_1` with inputs labeled `D0`, `D1`, and `S0` with output `O`. Actel calls a 2:1 MUX an `MX2` with inputs `A`, `B`, and `S` with output `Y`. This problem is not peculiar to Xilinx and Actel; each ASIC vendor names its logic cells, buffers, pads, and so on in a different manner. Consequently designers may not be able to transfer a netlist using one ASIC vendor library to another. Worse than this, designers may not even be able to transfer a design between two FPGA families made by the same FPGA vendor!

One solution to the lack of standards for cell libraries is to use a **generic cell library**, independent from any particular FPGA vendor. For example, most of the FPGA libraries include symbols that are equivalent to TTL 7400 logic series parts. The FPGA vendor's own software automatically handles the conversion from schematic symbols to the logic cells of the FPGA.

Schematic entry is not the only method of design entry for FPGAs. Some designers are happier describing control logic and state machines in terms of state diagrams and logic equations. A solution to some of the problems with schematic entry for FPGA design is to use one of several **hardware description languages (HDLs)** for which there are some standards. There are two sets of languages in common use. One set has evolved from the design of programmable logic devices (PLDs). The **ABEL** (pronounced "able"), **CUPL** ("cupple"), and **PALASM** ("pal-azzam") languages are simple and easy to learn. These languages are useful for describing state machines and combinational logic. The other set of HDLs includes **VHDL** and **Verilog**, which are higher-level and are more complex but are capable of describing complete ASICs and systems.

After completing design entry and generating a netlist, the next step is simulation. Two types of simulators are normally used for FPGA design. The first is a **logic simulator** for behavioral, functional, and timing simulation. This tool can catch any design errors. The designer provides input waveforms to the simulator and checks to see that the outputs are as expected. At this point, using a nondeterministic architecture, logic path delays are only estimates, since the wiring delays will not be known until after physical design (place-and-route) is complete. Designers then add or **back-annotate** the **postlayout timing information** to the **postlayout netlist** (also called a **back-annotated netlist**). This is followed by a **postlayout timing simulation**.

The second type of simulator, the type most often used in FPGA design, is a **timing-analysis** tool. A timing analyzer is a static simulator and removes the need for input waveforms. Instead the timing analyzer checks for critical paths that limit the speed of operation—signal paths that have large delays caused, say, by a high fanout net. Designers can set a certain delay restriction on a net or path as a **timing constraint**; if the actual delay is longer, this is a **timing violation**. In most design systems we can return to design entry and tag critical paths with attributes before completing the place-and-route step again. The next time we use the place-and-route software it will pay special attention to those signals we have labeled as critical in order to minimize the routing delays associated with those signals. The problem is

that this iterative process can be lengthy and sometimes nonconvergent. Each time timing violations are fixed, others appear. This is especially a problem with place-and-route software that uses random algorithms (and forms a chaotic system). More complex (and expensive) logic synthesizers can automate this iterative stage of the design process. The critical path information is calculated in the logic synthesizer, and timing constraints are created in a feedforward path (this is called **forward-annotation**) to direct the place-and-route software.

Although some FPGAs are reprogrammable, it is not a good idea to rely on this fact. It is very tempting to program the FPGA, test it, make changes to the netlist, and then keep programming the device until it works. This process is much more time consuming and much less reliable than performing thorough simulation. It is quite possible, for example, to get a chip working in an experimental fashion without really knowing why. The danger here is that the design may fail under some other set of operating conditions or circumstances. Simulation is the proper way to catch and correct these potential disasters.

8.1.1 Xilinx

Figure 8.1 shows the Xilinx design system. Using third-party design-entry software, the designer creates a netlist that forms the input to the Xilinx software. Utility software (`pin2xnf` for FutureNet DASH and `wir2xnf` for Viewlogic, for example) translate the netlist into a **Xilinx netlist format** (**XNF**) file. In the next step the Xilinx program `xnfmap` takes the XNF netlist and **maps** the logic into the Xilinx **Logic Cell Array** (**LCA**) architecture. The output from the mapping step is a MAP file. The schematic MAP file may then be **merged** with other MAP files using `xnfmerge`. This technique is useful to merge different pieces of a design, some created using schematic entry and others created, for example, using logic synthesis. A translator program `map2lca` translates from the logic gates (NAND gates, NOR gates, and so on) to the required CLB configurations and produces an unrouted LCA file. The Xilinx place-and-route software (`apr` or `ppr`) takes the unrouted LCA file and performs the allocation of CLBs and completes the routing. The result is a routed LCA file. A control program `xmake` (that works like the `make` program in C) can automatically handle the mapping, merging, and place-and-route steps. Following the place-and-route step, the logic and wiring delays are known and the postlayout netlist may be generated. After a postlayout simulation the **download file** or BIT file used to program the FPGA (or a PROM that will load the FPGA) is generated using the Xilinx `makebits` program.

Xilinx also provides a software program (Xilinx design editor, XDE) that permits manual control over the placement and routing of a Xilinx FPGA. The designer views a graphical representation of the FPGA, showing all the CLBs and interconnect, and can make or alter connections by pointing and clicking. This program is useful to check an automatically generated layout, or to explore critical routing paths, or to change and hand tune a critical connection, for example.

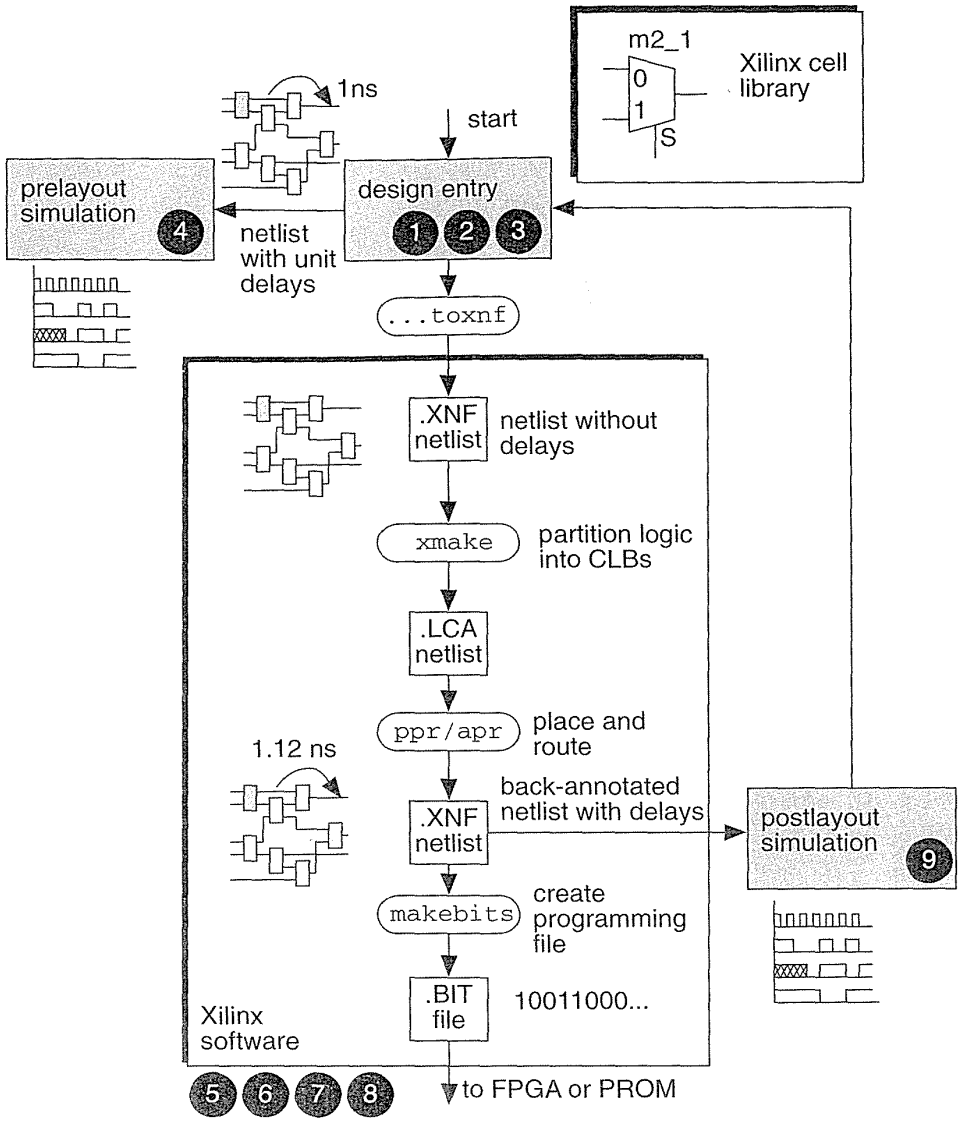


FIGURE 8.1 The Xilinx FPGA design flow. The numbers next to the steps in the flow correspond to those in the general ASIC design flow of Figure 1.10.

Xilinx uses a system called X-BLOX for creating regular structures such as vectored instances and datapaths. This system works with the Xilinx XNF netlist format. Other vendors, notably Actel and Altera, use a standard called **Relationally Placed Modules (RPM)**, based on the EDIF standard, that ensures that the pieces of an 8-bit adder, for example, are treated as a macro and stay together during placement.

8.1.2 Actel

Actel FPGA design uses third-party design entry and simulators. After creating a netlist, a designer uses the Actel software for the place-and-route step. The Actel design software, like other FPGA and ASIC design systems, employs a large number of file formats with associated filename extensions. Table 8.1 shows some of the Actel file extensions and their meanings.

TABLE 8.1 File types used by Actel design software.

ADL	Main design netlist
IPF	Partial or complete pin assignment for the design
CRT	Net criticality
VALIDATED	Audit information
COB	List of macros removed from design
VLD	Information, warning, and error messages
PIN	Complete pin assignment for the design
DFR	Information about routability and I/O assignment quality
LOC	Placement of non-I/O macros, pin swapping, and freeway assignment
PLI	Feedback from placement step
SEG	Assignment of horizontal routing segments
STF	Back-annotation timing
RTI	Feedback from routing step
FUS	Fuse coordinates (column-track, row-track)
DEL	Delays for input pins, nets, and I/O modules
AVI	Fuse programming times and currents for last chip programmed

Actel software can also map hardware description files from other programmable logic design software into the Actel FPGA architecture. As an example, Table 8.2 shows a text description of a state machine using an HDL from a company called LOG/iC. You can then convert the LOG/iC code to the PALASM code shown in Table 8.2. The Actel software can take the PALASM code and merge it with other PALASM files or netlists.

8.1.3 Altera

Altera uses a self-contained design system for its complex PLDs that performs design entry, simulation, and programming of the parts. Altera also provides an input and output interface to EDIF so that designers may use third-party schematic entry or a logic

TABLE 8.2 FPGA state-machine language.

LOG/iC state-machine language

```
*IDENTIFICATION
sequence detector
LOG/iC code
*X-NAMES
X; !input
*Y-NAMES
D; !output, D = 1 when three 1's appear on X
*FLOW-TABLE
;State, X input, Y output,  next state
  S1,     X1,       Y0,         F2;
  S1,     X0,       Y0,         F1;
  S2,     X1,       Y0,         F3;
  S2,     X0,       Y0,         F1;
  S3,     X1,       Y0,         F4;
  S3,     X0,       Y0,         F1;
  S4,     X1,       Y1,         F4;
  S4,     X0,       Y0,         F1;
*STATE-ASSIGNMENT
BINARY;
*RUN-CONTROL
PROGFORMAT = P-EQUATIONS;
*END
```

PALASM version

```
TITLE sequence detector
CHIP MEALY USER
CLK Z QQ2 QQ1 X
EQUATIONS
Z = X * QQ2 * QQ1
QQ2 := X * QQ1 + X * QQ2
QQ1 := X * QQ2 + X * /QQ1
```

synthesizer. We have seen that the interconnect scheme in the Altera complex PLDs is nearly deterministic, simplifying the physical-design software as well as eliminating the need for back-annotation and a postlayout simulation. As Altera FPGAs become larger and more complex, there are some exceptions to this rule. Some special cases require signals to make more than one pass through the routing structures or travel large distances across the Altera FastTrack interconnect. It is possible to tell if this will be the case only by trying to place and route an Altera device.

8.2 Logic Synthesis

Designers are increasingly using logic synthesis as a replacement for schematic entry. As microelectronic systems and their ASICs become more complex, the use of schematics becomes less practical. For example, a complex ASIC that contains over 10,000 gates might require hundreds of pages of schematics at the gate level. As another example, it is easier to write A = B + C than to draw a schematic for a 32-bit adder at the gate level.

The term *logic synthesis* is used to cover a broad range of software and software capabilities. Many logic synthesizers are based on **logic minimization**. Logic minimization is usually performed in one of two ways, either using a set of rules or using algorithms. Early logic-minimization software was designed using algorithms for two-level logic minimization and developed into multilevel logic-optimization software. Two-level and multilevel logic minimization is well suited to random logic that is to be implemented using a CBIC, MGA, or PLD. In these technologies, two-level logic can be implemented very efficiently. Logic minimization for FPGAs, including complex PLDs, is more difficult than other types of ASICs, because of the complex basic logic cells in FPGAs.

There are two ways to use logic synthesis in the design of FPGAs. The first and simplest method takes a hardware description, optimizes the logic, and then produces a netlist. The netlist is then passed to software that **maps** the netlist to an FPGA architecture. The disadvantage of this method is the inefficiency of decoupling the logic optimization from the mapping step. The second, more complicated, but more efficient method, takes the hardware description and directly optimizes the logic for a specific FPGA architecture.

Some logic synthesizers produce files in PALASM, ABEL, or CUPL formats. Software provided by the FPGA vendor then take these files and maps the logic to the FPGA architecture. The FPGA mapping software requires detailed knowledge of the FPGA architecture. This makes it difficult for third-party companies to create logic synthesis software that can map directly to the FPGA.

A problem with design-entry systems is the difficulty of moving netlists between different FPGA vendors. Once you have completed a design using an FPGA cell library, for example, you are committed to using that type of FPGA unless you repeat design entry using a different cell library. ASIC designers do not like this approach since it exposes them to the mercy of a single ASIC vendor. Logic synthesizers offer a degree of independence from FPGA vendors (universally referred to **vendor independence**, but this should, perhaps, be designer independence) by delaying the point in the design cycle at which designers need to make a decision on which FPGA to use. Of course, now designers become dependent on the synthesis software company.

8.2.1 FPGA Synthesis

For low-level logic synthesis, PALASM is a de facto standard as the lowest-common-denominator interchange format. Most FPGA design systems are capable of converting their own native formats into a PALASM file. The most common programmable logic design systems are ABEL from Data I/O, CUPL from P-CAD, LOG/iC from IsData, PALASM2 from AMD, and PGA-Designer from Minc. At a higher level, CAD companies (Cadence, Compass, Mentor, and Synopsys are examples) support most FPGA cell libraries. This allows you to map from a VHDL or Verilog description to an EDIF netlist that is compatible with FPGA design soft-

ware. Sometimes you have to buy the cell library from the software company, sometimes from the FPGA vendor.

TABLE 8.3 The VHDL code for the sequence detector of Table 8.2.

```
entity detector is port (X, CLK: in BIT; Z : out BIT); end;
architecture behave of detector is
   type states is (S1, S2, S3, S4);
   signal current, next: states;
begin
   combinational: process begin
      case current is
         when S1 =>
            if X = '1' then Z <= '0'; next <= S3; else Z <= '0'; next <= S1; end if;
         when S2 =>
            if X = '1' then Z <= '0'; next <= S2; else Z <= '0'; next <= S1; end if;
         when S3 =>
            if X = '1' then Z <= '0'; next <= S2; else Z <= '0'; next <= S1; end if;
         when S4 =>
            if X = '1' then Z <= '1'; next <= S4; else Z <= '0'; next <= S1; end if
      end case;
   end process
   sequential: process begin
      wait until CLK'event and CLK = '1'; current <= next ;
   end process;
end behave;
```

As an example, Table 8.3 shows a VHDL model for a pattern detector to check for a sequence of three '1's (excluding the code for the I/O pads). Table 8.4 shows a **script** or **command file** that runs the Synopsys software to generate an EDIF netlist from this VHDL that **targets** the TI version of the Actel FPGA parts. A script is a recipe that tells the software what to do. If we wanted to **retarget** this design to another type of FPGA or an MGA or CBIC ASIC, for example, we may only need a new set of cell libraries and to change the script (if we are lucky). In practice, we shall probably find we need to make a few changes in the VHDL code (in the areas of I/O pads, for example, that are different for each kind of ASIC). We now have a **portable design** and a measure of vendor independence. We have also introduced some dependence on the Synopsys software since the code in Table 8.3 might be portable, but the script (which is just as important a part of the design) in Table 8.4 may only be used with the Synopsys software. Nevertheless, using logic synthesis results in a more portable design than using schematic entry.

TABLE 8.4 The Synopsys script for the VHDL code of Table 8.3.

```
search_path = "."                          report_design > detector.rpt
/* use the TI cell libraries /*            /* optimize for area /*
link_library = tpc10.db                    max_area 0.0
target_library = tpc10.db                  compile
symbol_library = tpc10.sdb                 write -h -f db -o detector_opt.db
read -f vhdl detector.vhd                  report -area -cell -timing > detector.rpt
current_design = detector                  /* write EDIF netlist /*
write -n -f db -hierarchy -o detector.db   write -h -f edif -o detector.edf
/* design checking /*                      quit
check_design > detector.rpt
```

8.3 The Halfgate ASIC

This section illustrates FPGA design using a very simple ASIC—a single inverter. The hidden details of the design and construction of this "halfgate FPGA" are quite complicated. Fortunately, most of the inner workings of the design software are normally hidden from the designer. However, when software breaks, as it sometimes does, it is important to know how things work in order to fix the problem. The formats, filenames, and flow will change, but the information needed at each stage and the order in which it is conveyed will stay much the same.

8.3.1 Xilinx

Table 8.5 shows an FPGA design flow using Compass and Xilinx software. On the left of Table 8.5 is a script for the Compass programs—scripts for Cadence, Mentor, and Synopsys software are similar, but not all design software has the capability to be run on autopilot using scripts and a command language. The diagrams in Table 8.5 illustrate what is happening at each of the design steps. The following numbered comments, corresponding to the labels in Table 8.5, highlight the important steps:

1. The Verilog code, in `halfgate.v`, describes a single inverter.
2. The script runs the logic synthesizer that converts the Verilog description to an inverter (using elements from the Xilinx XC4000 library) and saves the result in a netlist, `halfgate_p.nls` (a Compass internal format).
3. The script next runs the logic optimizer for FPGAs. This program also adds the I/O pads. In this case, logic optimization implements the inverter by using an inverting output pad. The software writes out the netlist as `halfgate_p.xnf`.
4. A timing simulation is run on the netlist `halfgate_p.nls` (the Compass format netlist). This netlist uses the default delays—every gate has a delay of 1 ns.

TABLE 8.5 Design flow for the Xilinx implementation of the halfgate ASIC.

Script

```
# halfgate.xilinx.inp
shell setdef
   path working xc4000d xblox cmosch000x
quit
asic
   open [v]halfgate                          (1)
   synthesize
   save [nls]halfgate_p
quit
fpga                                         (2)
   set tag xc4000
   set opt area
   optimize [nls]halfgate_p
quit
qtv                                          (3)
   open [nls]halfgate_p
   trace critical
   print trace [txt]halfgate_p
quit
shell vuterm                                 (4)
   exec xnfmerge -p 4003PC84 halfgate_p > /dev/null
   exec xnfprep halfgate_p > /dev/null
   exec ppr halfgate_p > /dev/null
   exec makebits -w halfgate_p > /dev/null
   exec lca2xnf -g -v halfgate_p halfgate_b > /dev/null
quit
manager notice
utility netlist
   open [xnf]halfgate_b
   save [nls]halfgate_b
   save [edf]halfgate_b
quit
qtv
   open [nls]halfgate_b
   trace critical
   print trace [txt]halfgate_b
quit                                         (5)
```

Design flow

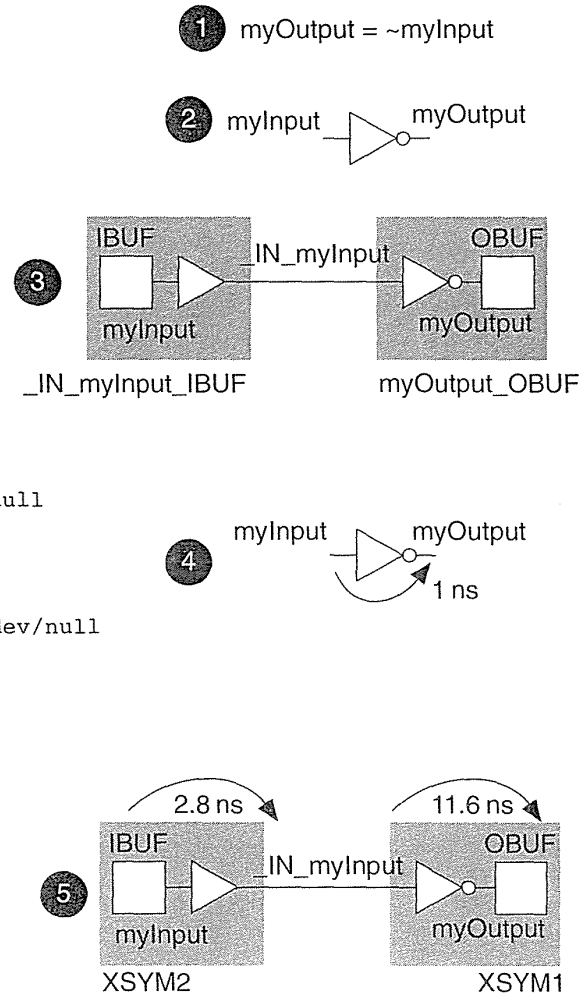


TABLE 8.6 The Xilinx files for the halfgate ASIC.

Verilog file (halfgate.v)

```
module halfgate(myInput, myOutput); input myInput; output myOutput; wire myOutput;
   assign myOutput = ~myInput;
endmodule
```

Preroute XNF file (halfgate_p.xnf)

```
LCANET, 5
USER, FPGA-Optimizer, 4.1, Date:960710 ,
Option: Area
PROG, FPGA-Optimizer, 4.1, "Lib=4000"
PART,  4010PG191
PWR, 0, GND
PWR, 1, VCC
SYM,_IN_myInput_IBUF,IBUF,LIBVER=2.0.0
PIN, I, I, myInput,
PIN, O, O, _IN_myInput,
END
EXT, myInput, I,
SYM, myOutput_obuf,OBUF,LIBVER=2.0.0,
PIN, I, I, _IN_myInput,, INV
PIN, O, O, myOutput,
END
EXT,  myOutput, O,
EOF
```

LCA file (halfgate_p.lca)

```
;: halfgate_p.lca (4003PC84-4), makebits
5.2.0, Tue Jul 16 20:09:43 1996
Version 2
Design 4003PC84 4 0
Speed -4
Addnet PAD_myInput PAD61.I2 PAD1.O
Netdelay PAD_myInput PAD1.O 3.1
Program PAD_myInput {65G521} {65G287}
{65G50} {63G50} {52G50} {45G50}
NProgram PAD_myInput col.B.long.3:PAD1.O
col.B.long.3:row.G.local.1
col.B.long.3:row.M.local.5-s MB.
40.1.14 MB.40.1.35 row.M.local.5:PAD61.I2
Editblk PAD61
Base IO
Config INFF: I1: I2:I O: OUT: PAD: TRI:
Endblk
Editblk PAD1
Base IO
Config INFF: I1: I2: O: OUT:O:NOT PAD:
TRI:
Endblk
Nameblk PAD61 myInput
Nameblk PAD1 myOutput
Intnet myOutput PAD myOutput
Intnet myInput PAD myInput
System FGG 0 VERS 2 !
System FGG 1 GD0 0 !
```

Postroute XNF file (halfgate_b.xnf)

```
LCANET, 4
PROG, LCA2XNF, 5.2.0, "COMMAND = -g -v
halfgate_p halfgate_b  TIME = Tue Jul 16
21:53:31 1996"
PART, 4003PC84-4
SYM, XSYM1, OBUF, SLOW
    PIN, O, O, myOutput, 3.0
    PIN, I, I, _IN_myInput, 8.6, INV
END
SYM, XSYM2, IBUF
    PIN, O, O, _IN_myInput, 2.8
    PIN, I, I, myInput
END
EXT, myOutput, O, 10
EXT, myInput, I, 29
EOF
```

5. At this point the script has run all of the Xilinx programs required to complete the place-and-route step. The Xilinx programs have created several files, the most important of which is `halfgate_p.lca`, which describes the FPGA layout. This postroute netlist is converted to `halfgate_b.nls` (the added suffix `'b'` stands for back-annotation). Next a timing simulation is performed on the postroute netlist, which now includes delays, to find the delay from the input (`myInput`) to the output (`myOutput`). This is the critical—and only—path. The simulation (not shown) reveals that the delay is 2.8 ns (for the input buffer) plus 11.6 ns (for the output buffer), for a total delay of 14.4 ns (this is for a XC4003 in a PC84 package, and default speed grade '4').

Table 8.6 shows the key Xilinx files that are created. The preroute file, `halfgate_p.xnf`, describes the IBUF and OBUF library cells but does not contain any delays. The LCA file, `halfgate_p.lca`, contains all the physical design information, including the locations of the pads and I/O cells on the FPGA (`PAD61` for `myInput` and `PAD1` for `myOutput`), as well as the details of the programmable connections between these I/O Cells. The postroute file, `halfgate_b.xnf`, is similar to the preroute version except that now the delays are included. Xilinx assigns delays to a pin (connector or terminal of a cell). In this case 2.8 ns is assigned to the output of the input buffer, 8.6 ns is assigned to the input of the output buffer, and finally 3.0 ns is assigned to the output of the output buffer.

8.3.2 Actel

The key Actel files for the halfgate design are the netlist file, `halfgate_io.adl`, and the STF delay file for back-annotation, `halfgate_io.stf`. Both of these files are shown in Table 8.7 (the STF file is large and only the last few lines, which contain the delay information, are shown in the table).

8.3.3 Altera

Because Altera complex PLDs use a deterministic routing structure, they can be designed more easily using a self-contained software package—an "all-in-one" software package using a single interface. We shall assume that we can generate a netlist that the Altera software can accept using Cadence, Mentor, or Compass software with an Altera design kit (the most convenient format is EDIF).

Table 8.8 shows the EDIF preroute netlist in a format that the Altera software can accept. This netlist file describes a single inverter (the line `'cellRef not'`). The majority of the EDIF code in Table 8.8 is a standard template to pass information about how the VDD and VSS nodes are named, which libraries are used, the name of the design, and so on. We shall cover EDIF in Chapter 9.

Table 8.9 shows a small part of the reports generated by the Altera software after completion of the place-and-route step. This report tells us how the software has used the basic logic cells, interconnect, and I/O cells to implement our design. With practice it is possible to read the information from reports such as Table 8.9

TABLE 8.7 The Actel files for the halfgate ASIC.

ADL file

```
; HEADER
; FILEID ADL ./halfgate_io.adl 85e8053b
; CHECKSUM 85e8053b
; PROGRAM certify
; VERSION 23/1
; ALSMAJORREV 2
; ALSMINORREV 3
; ALSPATCHREV .1
; NODEID 72705192
; VAR FAMILY 1400
; ENDHEADER
DEF halfgate_io; myInput, myOutput.
USE ADLIB:INBUF; INBUF_2.
USE ADLIB:OUTBUF; OUTBUF_3.
USE ADLIB:INV; u2.
NET DEF_NET_8;  u2:A, INBUF_2:Y.
NET DEF_NET_9;  myInput, INBUF_2:PAD.
NET DEF_NET_11;  OUTBUF_3:D, u2:Y.
NET DEF_NET_12;  myOutput, OUTBUF_3:PAD.
END.
```

STF file

```
; HEADER
; FILEID STF ./halfgate_io.stf c96ef4d8

... lines omitted ... (126 lines total)

DEF halfgate_io.
USE ; INBUF_2/U0;
  TPADH:'11:26:37',
  TPADL:'13:30:41',
  TPADE:'12:29:41',
  TPADD:'20:48:70',
  TYH:'8:20:27',
  TYL:'12:28:39'.
PIN u2:A;
  RDEL:'13:31:42',
  FDEL:'11:26:37'.
USE ; OUTBUF_3/U0;
  TPADH:'11:26:37',
  TPADL:'13:30:41',
  TPADE:'12:29:41',
  TPADD:'20:48:70',
  TYH:'8:20:27',
  TYL:'12:28:39'.
PIN OUTBUF_3/U0:D;
  RDEL:'14:32:45',
  FDEL:'11:26:37'.
END.
```

directly, but it is a little easier if we also look at the netlist. The EDIF version of postroute netlist for this example is large. Fortunately, the Altera software can also generate a Verilog version of the postroute netlist. Here is the generated Verilog postroute netlist, `halfgate_p.vo` (not `'.v'`), for the halfgate design:

```
// halfgate_p (EPM7032LC44) MAX+plus II Version 5.1 RC6 10/03/94
// Wed Jul 17 04:07:10 1996
`timescale 100 ps / 100 ps

module TRI_halfgate_p( IN, OE, OUT ); input IN; input OE; output OUT;
bufif1 ( OUT, IN, OE );
  specify
    specparam TTRI = 40; specparam TTXZ = 60; specparam TTZX  = 60;
    (IN => OUT) = (TTRI,TTRI);
```

TABLE 8.8 EDIF netlist in Altera format for the halfgate ASIC.

```
(edif halfgate_p
(edifVersion 2 0 0)
(edifLevel 0)
(keywordMap
 (keywordLevel 0))
(status
 (written
  (timeStamp 1996 7 10 23
55 8)
  (program "COMPASS Design
Automation -- EDIF Interface"
   (version "v9r1.2 last
updated 26-Mar-96"))
  (author "mikes")))
(library flex8kd
 (edifLevel 0)
 (technology
  (numberDefinition )
  (simulationInfo
   (logicValue  H)
   (logicValue  L)))
 (cell not
  (cellType GENERIC)
  (view COMPASS_mde_view
   (viewType NETLIST)
   (interface
    (port IN
     (direction INPUT))
    (port OUT
     (direction OUTPUT))
    (designator
"@@Label")))))
 (library working
  (edifLevel 0)
  (technology
   (numberDefinition )
   (simulationInfo
    (logicValue  H)
    (logicValue  L)))
  (cell halfgate_p
   (cellType GENERIC)
   (view COMPASS_nls_view
    (viewType NETLIST)
    (interface
     (port myInput
      (direction INPUT))
     (port myOutput
      (direction OUTPUT))
     (designator "@@Label"))
    (contents
     (instance B1_i1
      (viewRef
COMPASS_mde_view
       (cellRef not
        (libraryRef
flex8kd))))
     (net myInput
      (joined
       (portRef myInput)
       (portRef IN
        (instanceRef
B1_i1))))
     (net myOutput
      (joined
       (portRef myOutput)
       (portRef OUT
        (instanceRef
B1_i1))))
     (net VDD
      (joined )
      (property global
       (string "vcc")))
     (net VSS
      (joined )
      (property global
       (string "gnd")))))))
 (design halfgate_p
  (cellRef halfgate_p
   (libraryRef working))))
```

```
      (OE => OUT) = (0,0, TTXZ, TTZX, TTXZ, TTZX);
   endspecify
endmodule

module halfgate_p (myInput, myOutput);
   input myInput; output myOutput; supply0 gnd; supply1 vcc;
   wire B1_i1, myInput, myOutput, N_8, N_10, N_11, N_12, N_14;
   TRI_halfgate_p tri_2 ( .OUT(myOutput), .IN(N_8), .OE(vcc) );
   TRANSPORT transport_3 ( N_8, N_8_A );
   defparam transport_3.DELAY = 10;
   and delay_3 ( N_8_A, B1_i1 );
   xor xor2_4 ( B1_i1, N_10, N_14 );
   or or1_5 ( N_10, N_11 );
   TRANSPORT transport_6 ( N_11, N_11_A );
```

TABLE 8.9 Report for the halfgate ASIC fitted to an Altera MAX 7000 complex PLD.

```
** INPUTS **
                                          Shareable
                                          Expanders      Fan-In     Fan-Out
 Pin      LC   LAB  Primitive     Code   Total Shared n/a INP  FBK   OUT  FBK   Name
  43       -    -       INPUT              0      0   0    0    0     0    1   myInput

** OUTPUTS **
                                          Shareable
                                          Expanders      Fan-In     Fan-Out
 Pin      LC   LAB  Primitive     Code   Total Shared n/a INP  FBK   OUT  FBK   Name
  41      17     B      OUTPUT      t       0      0   0    1    0     0    0   myOutput

** LOGIC CELL INTERCONNECTIONS **
Logic Array Block 'B':
        +- LC17 myOutput
        |
LC      | | A B | Name

Pin
43   -> * | - * | myInput

* = The logic cell or pin is an input to the logic cell (or LAB) through the PIA.
- = The logic cell or pin is not an input to the logic cell (or LAB).
```

```
  defparam transport_6.DELAY = 60;
  and and1_6 ( N_11_A, N_12 );
  TRANSPORT transport_7 ( N_12, N_12_A );
  defparam transport_7.DELAY = 40;
  not not_7 ( N_12_A, myInput );
  TRANSPORT transport_8 ( N_14, N_14_A );
  defparam transport_8.DELAY = 60;
  and and1_8 ( N_14_A, gnd );
endmodule
```

The Verilog model for our ASIC, `halfgate_p`, is written in terms of other models: `and`, `xor`, `or`, `not`, `TRI_halfgate_p`, `TRANSPORT`. The first four of these are **primitive models** for basic logic cells and are built into the Verilog simulator. The model for `TRI_halfgate_p` is generated together with the rest of the code. We also need the following model for `TRANSPORT`, which contains the delay information for the Altera MAX complex PLD. This code is part of a file (`alt_max2.vo`) that is generated automatically.

```
// MAX+plus II Version 5.1 RC6 10/03/94 Wed Jul 17 04:07:10 1996
`timescale 100 ps / 100 ps
module TRANSPORT( OUT, IN ); input IN; output OUT; reg OUTR;
```

```
wire OUT = OUTR; parameter DELAY = 0;
`ifdef ZeroDelaySim
  always @IN OUTR <= IN;
`else
  always @IN OUTR <= #DELAY IN;
`endif
`ifdef Silos
  initial #0 OUTR = IN;
`endif
endmodule
```

The Altera software can also write the following VHDL postroute netlist:

```
-- halfgate_p (EPM7032LC44) MAX+plus II Version 5.1 RC6 10/03/94
-- Wed Jul 17 04:07:10 1996
LIBRARY IEEE; USE IEEE.std_logic_1164.all;
ENTITY n_tri_halfgate_p IS
GENERIC (ttri: TIME := 1 ns; ttxz: TIME := 1 ns; ttzx: TIME := 1 ns);
PORT (in0 : IN X01Z; oe  : IN X01Z; out0: OUT X01Z);
END n_tri_halfgate_p;

ARCHITECTURE behavior OF n_tri_halfgate_p IS
BEGIN
PROCESS (in0, oe) BEGIN
  IF oe'EVENT THEN
    IF oe = '0' THEN out0 <= TRANSPORT 'Z' AFTER ttxz;
    ELSIF oe = '1' THEN out0 <= TRANSPORT in0 AFTER ttzx;
    END IF;
  ELSIF oe = '1' THEN out0 <= TRANSPORT in0 AFTER ttri;
  END IF;
END PROCESS;
END behavior;

LIBRARY IEEE; USE IEEE.std_logic_1164.all; USE work.n_tri_halfgate_p;
ENTITY n_halfgate_p IS
  PORT ( myInput : IN X01Z; myOutput : OUT X01Z);
END n_halfgate_p;

ARCHITECTURE EPM7032LC44 OF n_halfgate_p IS
SIGNAL gnd : X01Z := '0'; SIGNAL vcc : X01Z := '1';
SIGNAL n_8, B1_i1, n_10, n_11, n_12, n_14 : X01Z;
COMPONENT n_tri_halfgate_p
  GENERIC (ttri, ttxz, ttzx: TIME);
  PORT (in0, oe  : IN X01Z; out0 : OUT X01Z);
END COMPONENT;
BEGIN
PROCESS(myInput) BEGIN ASSERT myInput /= 'X' OR Now = 0 ns
  REPORT "Unknown value on myInput" SEVERITY Warning;
END PROCESS;
n_tri_2: n_tri_halfgate_p
```

```
   GENERIC MAP (ttri => 4 ns, ttxz => 6 ns, ttzx => 6 ns)
   PORT MAP (in0 => n_8, oe => vcc, out0 => myOutput);
n_delay_3: n_8 <= TRANSPORT B1_i1 AFTER 1 ns;
n_xor_4: B1_i1 <= n_10 XOR n_14;
n_or_5: n_10 <= n_11;
n_and_6: n_11 <= TRANSPORT n_12 AFTER 6 ns;
n_not_7: n_12 <= TRANSPORT NOT myInput AFTER 4 ns;
n_and_8: n_14 <= TRANSPORT gnd AFTER 6 ns;
END EPM7032LC44;

LIBRARY IEEE; USE IEEE.std_logic_1164.all; USE work.n_halfgate_p;
ENTITY halfgate_p IS
   PORT ( myInput : IN std_logic; myOutput : OUT std_logic);
END halfgate_p;
ARCHITECTURE EPM7032LC44 OF halfgate_p IS
COMPONENT n_halfgate_p PORT (myInput : IN X01Z; myOutput : OUT X01Z);
END COMPONENT;
BEGIN
   n_0: n_halfgate_p
   PORT MAP ( myInput => TO_X01Z(myInput), myOutput => myOutput);
END EPM7032LC44;
```

The VHDL is a little harder to decipher than the Verilog, so the schematic for the VHDL postroute netlist is shown in Figure 8.2. This VHDL netlist is identical in function to the Verilog netlist, but the net names and component names are different. Compare Figure 8.2 with Figure 5.15(c) in Section 5.4, "Altera MAX," which shows the Altera basic logic cell and Figure 6.23 in Section 6.8, "Other I/O Cells," which describes the Altera I/O cell. The software has fixed the inputs to the various elements in the Altera MAX device to implement a single inverter.

8.3.4 Comparison

The halfgate ASIC design illustrates the differences between a nondeterministic coarse-grained FPGA (Xilinx XC4000), a nondeterministic fine-grained FPGA (Actel ACT 3), and a deterministic complex PLD (Altera MAX 7000). These differences, summarized as follows, were apparent even in the halfgate design:

1. The Xilinx LCA architecture does not permit an accurate timing analysis until after place and route. This is because of the coarse-grained nondeterministic architecture.
2. The Actel ACT architecture is nondeterministic, but the fine-grained structure allows fairly accurate preroute timing prediction.
3. The Altera MAX complex PLD requires logic to be fitted to the product steering and programmable array logic. The Altera MAX 7000 has an almost deterministic architecture, which allows accurate preroute timing.

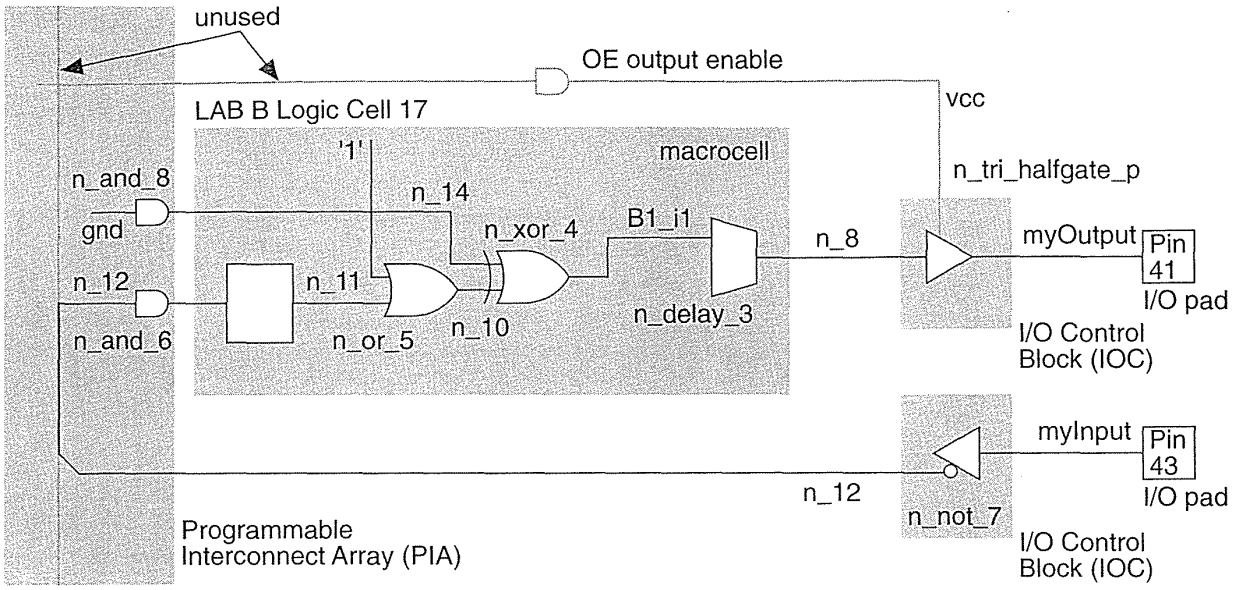


FIGURE 8.2 The VHDL version of the postroute Altera MAX 7000 schematic for the half-gate ASIC. Compare this with Figure 5.15(c) and Figure 6.23.

8.4 Summary

The important concepts covered in this chapter are:

- FPGA design flow: design entry, simulation, physical design, and programming
- Schematic entry, hardware design languages, logic synthesis
- PALASM as a common low-level hardware description
- EDIF, Verilog, and VHDL as vendor-independent netlist standards

8.5 Problems

* = Difficult, ** = Very difficult, *** = Extremely difficult

8.1 (Files, 60 min.) Create a version of Table 8.1 for your design system.

8.2 (Scripts, 60 min.) Create a version of Table 8.5 for your design system.

8.3 (Halfgate, 60 min.)

a. Using an FPGA of your choice, estimate the preroute delay of a single inverter (including I/O delays).

b. Complete a halfgate design and explain the postroute delays (make sure you know what conditions are being used—worst-case commercial, for example).

8.4 (***Xilinx die analysis, 120 min.) The data in Table 8.10 shows some information derived from a die photo of an ATT3020 (equivalent to a Xilinx 3020) in the AT&T data book. The die photo shows the CLBs clearly enough that we can measure their size. Then, knowing the actual die size, we can calculate the CLB size and other parameters. From your knowledge of the contents of the XC3020 CLB, as well as the programming and interconnect structures, make an estimate (showing all of your approximations and explaining all of your assumptions) of the CLB area and compare this to the value of 277 $mils^2$ shown in Table 8.10. You will need to calculate the number of logic gates in each CLB including the LUT resources. Estimate how many pass transistors and memory elements are required as well as calculate how many routing resources are assigned to each CLB. *Hint:* You may need to use the Xilinx software, look at the Xilinx data books, or even the AT&T (Lucent) Orca documentation.

TABLE 8.10 ATT3020 die information (Problem 8.4).

Parameter	Specified in data book	Measured on die photo	Calculated from die photo
3020 die width	183.5 mil	4.1 cm	—
3020 die height	219.3 mil	4.9 cm	—
3000 CLB width	—	0.325 cm	14.55 mil = 370 μm
3000 CLB height	—	0.425 cm	19.02 mil = 483 μm
3000 CLB area	—	—	277 $mils^2$
3020 pad pitch	—	1.61 mm/pad	7.21 mil/pad

Source: AT&T Data Book, July 1992, p. 3-76, MN92-024FPGA.

8.5 (***FPGA process, 120 min.) Table 8.11 describes AT&T's 0.9 μm twin-tub V CMOS process, with 0.75 μm minimum design rules and 0.6 μm effective channel length and silicided (TiS_2) poly, source, and drain. This is the process used by AT&T to second-source the Xilinx XC3000 family of FPGAs. Calculate the parasitic resistance and capacitance parameters for the interconnect.

8.6 (Xilinx die costs, 10 min.) Table 8.12 shows the AT&T ATT3000 series die information. Assume a 6-inch wafer that costs $2000 to fabricate and has a 90 percent yield. **(a)** What are the die costs? **(b)** Compare these figures to the costs of XC3020 parts in 1992 and comment.

TABLE 8.11 ATT3000 0.9 μm twin-tub V CMOS process (Problem 8.5).

Parameter	Value
Die thickness, t_{die}	21 mil
Wafer diameter, W_D	5 inch
Wafer thickness, W_t	25 mil
Minimum feature size, 2λ	0.75 μm
Effective gate length, L_{eff} (*n*-channel and *p*-channel)	0.6 μm
First-level metal, m1	Ti/AlCuSi
Second-level metal, m2	AlCuSi
m1 width	0.9 μm
m2 width	1.2 μm
m1 thickness	0.5 μm
m2 thickness	1.0 μm
m1 spacing	1.0 μm
m2 spacing	1.3 μm
D1 dielectric thickness, boron/phosphorus doped glass	3500 Å
D2 dielectric thickness, undoped glass	9000 Å
Minimum contact size	1.0 μm
Minimum via size	1.2 μm
Isolation oxide, FOX	3500 Å
Gate oxide	150 Å

Source: AT&T Data Book, July 1992, p. 2-37 and p. 3-76, MN92-024FPGA.

TABLE 8.12 ATT3000 die information (Problem 8.6).

Die	Die height /mils	Die width /mils	Die area /mils^2	Die area /cm^2	CLBs	Die perimeter /mils	I/O pads
3020	219.3	183.5	40,242	0.26	8 × 8	806	74
3030	259.8	215.0	55,857	0.36	10 × 10	950	98
3042	295.3	242.5	71,610	0.46	12 × 12	1076	118
3064	270.9	366.5	99,285	0.64	16 × 14	1275	142
3090	437.0	299.2	130,750	0.84	16 × 20	1472	166

Source: AT&T Data Book, July 1992, p. 3-75, MN92-024FPGA. 1 mil^2 = $2.54^2 \times 10^{-6}$ cm^2 = 6.452×10^{-6} cm^2.

8.7 (Pad density) Table 8.12 shows the number of pads on each of the AT&T 3000 (equivalent to the Xilinx XC3000) die. Calculate the pad densities in mil/pad for each part and compare with the figure for the ATT3020 in Table 8.10.

8.8 (Xilinx HardWire, 10 min.) Xilinx manufactures nonprogrammable versions of its LCA family of FPGAs. These **HardWire** chips are useful when a customer wishes to convert to high-volume production. The Xilinx 1996 Product overview (p. 16) shows two die photographs: one, an XC3090 (with the four quadrants of 8×10 CLB matrices visible), which is $32\,\text{mm} \times 47\,\text{mm}$; the other shows the HardWire version ($24\,\text{mm} \times 29\,\text{mm}$). Estimate the die size of the HardWire version from the data in Table 8.12 and estimate the percentage of a Xilinx LCA that is taken up by SRAM.

Answer: 60,500 mils^2; 50 %.

8.9 (Xilinx XDE, 10 min.) During his yearly appraisal Dewey explains to you how he improved three Xilinx designs last year and managed to use 100 percent of the CLBs on these LCA chips by means of the XDE manual place-and-route program. As Dewey's boss, rank Dewey from 1 (bad) to 5 (outstanding) and explain your ranking in a space that has room for no more than 20 words.

8.10 (Clocks, 60 min) (From a discussion on an Internet newsgroup including comments from Peter Alfke of Xilinx) "Xilinx guarantees that the minimum value for any delay parameter is always more than 25 % of the maximum value for that same parameter, as published for the fastest speed grade offered at any time. Many parameters have been reduced significantly over the years, but the clock delay has not. For example, comparing the fastest available XC3020-70 in 1988 with the fastest available XC3020A-6 (1996):

- logic delay (t_{ILO}) decreased from 9 ns to 4.1 ns
- output-to-pad delay decreased from 10 ns to 5 ns
- internal-clock-to-output pad delay decreased from 13 ns to 7 ns

The internal speed has more than doubled, but the worst-case clock distribution delay specification has only changed from 6.0 ns (1988) to 5.7 ns (1996)."

Comment on the reasons for these changes and their repercussions.

8.11 (State-machine design)

a. (10 min.) Draw the state diagram for the LOG/iC code in Table 8.2.

b. (10 min.) Show, using an example input sequence, that the detector works.

c. (10 min.) Show that the state equations and the encoding for the PALASM code in Table 8.2 correctly describe the sequence detector state machine.

d. (30 min.) Convert this design to a different format of your choice: schematic, low-level design language, or HDL.

e. (30 min.) Simulate and test your design.

8.12 (FPGA software, 60 min.) Write a minitutorial (less than 2 pages) on using your FPGA design system. An example set of instructions for the Altera MAX PLUS II software on a Unix system are shown below:

Setup:

1. Copy `~altera/M+2/maxplus2.ini` into `~you/yourDirectory` (call this the working directory).
2. Edit `maxplus2.ini` and point the `DESIGN_NAME` to your design
3. Copy `~altera/M+2/compass.lmf` and `~altera/M+2/compass.edc` into your working directory.
4. Copy `~altera/M+2/foo.acf` into your working directory and rename it `mydesign.acf` if your design name is `mydesign.edf`.
5. Set the environment as follows:

```
setenv LM_LICENSE_FILE ~altera/maxplus2/adm/license.altera
set path=($path ~altera/maxplus5.1/bin)
```

and run the programs in batch mode: `maxplus2 -c mydesign.edf`. Add to this information on any peculiarities of the system you are using (handling of overwriting of files, filename extensions and when they are created, arguments required to run the programs, and so on).

8.13 (Help, 20 min.) Print the "help" for the key programs in your FPGA system and form it into a condensed "cheat-sheet." Most programs echo help instruction when called with a `'-help'` or `'?'` argument (this ought to be a standard). For example, in the Actel system the key programs are `edn2adl`, `adl2edn`, and `als` (in newer versions `adl2edn` is now an option to `als`). *Hint:* Actel does not use `'-help'` argument, but you can get instructions on the syntax for each option individually. Table 8.13 shows an example for the Xilinx `xdelay` program.

8.6 Bibliography

There are few books on FPGA design software. Skahill's book [1996] covers PLD and FPGA design with Cypress FPGAs and the Cypress Warp design system. Connor has written two articles in EDN describing a complete FPGA design project [1992]. Most of the information on design software is available from the software companies themselves—increasingly in online form. There is still some material that is only available through the BBS or from a **file-transfer protocol (ftp)** site. There is also a great deal of valuable material available in data books printed between 1990 and 1995, prior to the explosion of the use of the Internet in the late-1990s. I have included pointers to these sources in the following sections.

TABLE 8.13 Xilinx `xdelay` arguments.

```
usage: xdelay [<options>] [<lcafile> ..]
where <options> are:
   -help                    Print this help.
   -timespec                Do timespec based delay analysis.
   -s                       Write short xdelay report.
   -x                       Write long xdelay report.
   -t <template file>       Read <template file>.
   -r                       Use two letter style block names in output.
   -o <file>                Send output to file.
   -w                       Write design file, after retiming net delays.
   -u <speed>               Use the <speed> speed grade.
   -d                       Don't trace delay paths.
   -convert <input .lca file> <new part type> <output .lca file>
                             Convert the input design to a new part type.
Specify no arguments to run xdelay in interactive mode.

        To Select Report           Specify Option
        -------------------------  --------------------------
        TimeSpec summary           -timespec
        Short path details         -s
        Long path details          -x
        Analyze summary            none of -s, -x or -timespec

A template file can be specified with the -t option to further filter the selected
report. Only those template commands relevant to the selected report will be used.

Using -w and -d options together will insert delay information into the design file(s),
without tracing any paths.

The -convert option may not be used with any other options.
```

8.6.1 FPGA Vendors

Actel (`http://www.actel.com`) has a Frequently Asked Questions (FAQ) guide that is an indication of the most common problems with FPGA design:

- Software versions, installation, and security, and not having enough computer memory
- X11, Motif, and OpenWindows—problems with paths and fonts. Compatibility problems with Windows 95 and NT
- Including I/O pads in a design using schematic entry and logic synthesis—problems with the commands and the exact syntax to use

- Using third-party software for schematic entry or logic synthesis and libraries—problems with versions and paths
- EDIF netlist issues

It seems most of these problems never go away—they just keep resurfacing. If you design a halfgate ASIC, an inverter, start-to-finish, as soon as you get a new set of software, this will alert you to most of the problems you are likely to encounter.

The May 1989 Actel data book contains details of the early antifuse experiments. The Actel April 1990 data book has a chip photo of the Actel 1010 on the cover (from which some useful information may be derived). Reliability reports and article reprints are now included in the data books (see, for example, [Actel, 1996]). There is PowerPoint presentation on FPGAs (`architec.exe`) and the Actel FPGA architecture at its Web site.

The Xilinx data book (see, for example, [Xilinx, 1996]) contains several hundred pages of information on LCA parts. Xilinx produced a separate *User Guide and Tutorials* book that contains over 600 pages of application notes, guides, and tutorials on designing with FPGAs and Xilinx FPGAs in particular. XCELL is the quarterly *Xilinx Newsletter*, first published in 1988. It is available online and contains useful tips and pointers to new application notes. There is an extensive set of Xilinx Application Notes at `http://www.xilinx.com/apps`. A 250-page guide to using the Synopsys software (`hdl_dg.pdf`) covers many of the problems users experience in using any logic synthesizer for FPGA design.

Xilinx provides design kits for its EPLD FPGAs for third-party software such as the Viewlogic design entry and simulation programs. The interconnect architecture in the Xilinx EPLD FPGA is deterministic and so postlayout timing results are close to prelayout estimates.

AMD, before it sold its stake in Xilinx, published the 1989/1990 Programmable Data Array Book, which was distinct from the Xilinx data book. The AMD data book contains useful information and code for programs to download configuration files to Xilinx FPGAs from a PC that are still useful.

Altera publishes a series of loose-leaf application notes on a variety of topics, some of them are in the data book (see, for example [Altera, 1996]), but some are not. Most of these application notes are available as the AN series of documents at `http://www.altera.com/html/literature`. This includes guides on using Cadence, Mentor, Viewlogic, and Synopsys software. The 100-page Synopsys guide (`as_sig.pdf`) explains many of the limitations of logic synthesizers for FPGA design and includes the complete VHDL source code for a voice-mail machine as an example.

Atmel has a series of data sheets and application notes for its PLD logic at `http://www.atmel.com`. Some of the data sheets (for the ATV2500, for example, available as `doc156.pdf`) also include examples of the use of CUPL and ABEL. An application note in Atmel's data book (available as `doc168.pdf`) includes the ABEL source code for a video frame grabber and a description of the NTSC video format. Atmel offers a review of its links to third-party software in a section "PLD Software

Tools Overview" in its data book (available online as `doc150.pdf` at `http://www.atmel.com/atmel/products`). Atmel uses an IBM-compatible PC-based system based on the Viewlogic software. Schematic entry uses Viewdraw and simulation uses Viewsim. Atmel provides a separate program, a **fitter**, to optimize a schematic for its FPGA architecture. The output from this software generates an optimized schematic. The place-and-route software then works with this new schematic. Atmel provides an interactive editor similar to the Xilinx design editor that allows the designer to perform placement manually. Atmel also supports PLD design software such as Synario from Data I/O.

The QuickLogic design kit uses the ECS (Engineering Capture System) developed by the CAD/CAM Group and now part of DATA I/O. Simulation uses X-SIM, a product of Silicon Automation Systems.

Cypress has a low-cost design system (for QuickLogic and its own series of complex PLDs) called Warp that uses VHDL for design entry.

8.6.2 Third-Party Software

There is a bewildering array of software and software companies that make, sell, and develop products for PLD and FPGA design. These are referred to as **third-party vendors**. In the remainder of this section we shall describe (in alphabetical order) some of the available third-party software. This list changes frequently and for more information you might search the EE sites from the Bibliography in Chapter 1.

Accel (`http://www.edac.org/EDAC/Companies`) produces Tango and P-CAD (which used to belong to Personal CAD Systems) that are a low-cost and popular schematic-entry and PCB layout software for PCs. Currently there are no FPGA vendors that support P-CAD or Tango directly. The missing ingredient is a set of libraries with the appropriate schematic symbols for the logic macros and cells used by the FPGA vendor.

AMD (`http://www.amd.com`) produces the Mach series of PLDs and is also the owner of PALASM. All of the FPGA vendors use the PALASM and PALASM2 languages as interchange formats. Using PALASM is an easy way to incorporate a PLD into an FPGA.

Antares (`http://www.anteresco.com`) is a spin-off from Mentor Corporation formed from Exemplar Logic, a company specializing in synthesis software for PLDs and FPGAs, and Model Technology, who produce a VHDL and Verilog simulator using a common kernel.

Cadence (`http://www.cadence.com`) is one of the largest EDA companies. They offer design kits for PLD and FPGA design with its schematic-entry (Composer) and logic-synthesis (Concept) software. The Cadence Web site has some pictures of ASIC and FPGA design flow in its third-party support area. To find these, search for "FPGA" from the main menu.

Compass Design Automation (`http://www.compass-da.com`) is a spin-off from VLSI Technology that specializes in ASIC design software and cell libraries. As part of its system design software, this vendor includes compilers and libraries for Xilinx, Actel, and Altera FPGAs.

Data I/O (`http://www.data-io.com`) makes the FutureNet DASH schematic-entry program primarily for IBM-compatible PCs. Version 5 also has an EDIF 2 0 0 netlist writer, and an optional program PLDlinx to convert designs to ABEL. Data I/O's ABEL is a very widely used PLD design standard. Most FPGA software allows the merging of ABEL files with netlists from schematic-entry programs. Usually you have to translate ABEL to PALASM first and then merge the PALASM file with any netlists that you created from schematics. ABEL is available on SUN workstations, IBM-compatible PC-DOS, and Macintosh platforms. The Macintosh version is available through Capilano Computing, using its DesignWorks program. Data I/O has extended its ABEL language for use with FPGA design. ABEL-FPGA is a set of software that can accept hardware descriptions in ABEL-HDL. ABEL-HDL is an extension of the ABEL language which is optimized for programmable logic. One of the features of ABEL-HDL is a set of naming extensions, **dot extensions**, which allow the designer to specify how certain signals will be mapped into an FPGA.

Data I/O also makes a number of programmers. For example, the Unisite PROM programmer can be used to program Actel, Altera MAX, and Xilinx EPLD devices.

Data I/O has recently launched a separate division called Synario Design Automation (`http://www.synario.com`) that has taken over ABEL and produces a new series of PLD and FPGA design software under the Synario banner.

Exemplar, now part of Antares, writes many of the software modules for logic synthesis used by other companies in their FPGA synthesis software. Exemplar provides a software package that allows you to enter hardware descriptions in ABEL, PALASM, CUPL, or Minc formats.

ISDATA produces a system called LOG/iC that can be used for FPGA design. LOG/iC produces **JEDEC fusemap** files, which can be converted and merged with netlists created with other vendors' software. An evaluation diskette contains LOG/iC software that programs the Lattice GAL16V8. ISDATA also makes a program called STATE/view for design using state diagrams and flow charts and works with LOG/iC and ABEL. HINT is a program that accepts a subset of VHDL and compiles to the LOG/iC language.

Logical Devices (`http://www.logicaldevices.com`) acquired **CUPL**, a widely used programming language for PLDs, from Personal CAD Systems in 1987. Most FPGA vendors allow you to use files in CUPL format indirectly. Usually you translate to the PALASM format first in order to incorporate any logic you design with CUPL. Logical Devices also sells EPROM programming hardware. They manufacture programmers for FPGAs.

Mentor Graphics Corporation (`http://www.mentorg.com`) is a large EDA company. Mentor produces schematic-entry and logic-synthesis software, IDEA Station and FPGA Station, that interface to the major FPGA vendors (see also Antares).

Minc's PLDesigner software allows the entry of PLD designs using a mixture of truth tables, waveforms, Minc's Design Synthesis Language (DSL), schematic entry, or a netlist (in EDIF format). Another Minc program PGADesigner includes the ability to target FPGAs as well as PLDs. This program is compatible with the OrCAD, P-CAD, and FutureNet DASH schematic-entry programs.

OrCAD (`http://www.orcad.com`) is a popular low-cost PC schematic-entry program supported directly by a number of FPGA vendors.

Simucad (`http://www.simucad.com`) produces PC-SILOS, a low-cost logic-simulation program for PCs machines. Xilinx used to bundle Simucad with FutureNet DASH in its least expensive, entry-level design kit.

Synopsys (`http://www.synopsys.com`) sells logic-synthesis software. There are two main products: the Design Compiler for ASIC design and the FPGA Compiler for FPGA design. FPGA Express is a PC-based FPGA logic synthesizer. There is an extensive on-line help system available for Synopsys customers.

Tanner Research (`http://www.tanner.com`) offers a variety of ASIC design software and a "burning service"; you send them the download files to program the FPGAs and Tanner Research programs the parts and ships them to you. Tanner Research also offers an Actel schematic library for its schematic-entry program S-Edit.

Texas Instruments (TI) and Minc produces mapping software between TI's gate arrays and FPGAs (TI's relationship with Actel is somewhere between a second-source and a partner). Mapping software allows designers to design for a TI gate array, for example, but prototype in FPGAs. Alternatively you could take an existing FPGA design and map it into a TI gate array. This type of design flow is popular with vendors such as AT&T (Lucent), TI, and Motorola who would like you to prototype with their FPGAs before transferring any high-volume products to their ASICs.

Viewlogic (`http://www.viewlogic.com`) produces the Workview and PRODesigner systems that are sets of ASIC design programs available on a variety of platforms. The Workview software consists of a schematic-entry program Viewdraw; two simulators: Viewsim and Viewfault; a synthesis tool, Viewgen; Viewplace for layout interface; Viewtrace for simulation analysis; and Viewwave for graphical display. There is also a package, Viewbase, that is a set of software routines enabling programmers to access Viewlogic's database in order to create EDIF, VHDL, and CFI (CAD Framework Initiative) interfaces. Most of the FPGA vendors have a means to incorporate Viewlogic's schematic netlists using Viewlogic's WIR netlist format. Viewlogic provides a number of applications notes (TECHniques) and includes a list of bug fixes, software limitations, and workarounds online.

8.7 References

Page numbers in brackets after a reference indicate its location in the chapter body.

Actel. 1996. *FPGA Data Book and Design Guide.* No catalog information. Available from Actel Corporation, 955 East Arques Avenue, Sunnyvale, CA 94086-4533, (408) 739-1010. Contains design guides and applications notes, including: Estimating Capacity and Performance for ACT 2 FPGA Designs (describes circuits to connect FPGAs to PALs); Binning Circuit of Actel FPGAs (describes circuits and data for performance measurement); Global Clock Networks (describes clock distribution schemes); Fast On and Off Chip Delays with ACT 2 I/O Latches (describes techniques to improve I/O performance); Board Level Considerations for Actel FPGAs (describes ground bounce and SSO problems); A Power-On Reset (POR) Circuit for Actel Devices (describes problems caused by slowly rising supply voltage); Implementing Load (*sic*) Latency Fast Counters with ACT 2 FPGAs; Oscillators for Actel FPGAs (describes crystal and RC oscillators); Designing a DRAM Controller Using Language-Based Synthesis (a detailed Verilog description of a 4 MB DRAM controller including refresh). See also the Actel Web site. [p. 322]

Altera. 1996. *Data Book.* No catalog information. Available from Altera Corporation, 2610 Orchard Parkway, San Jose, CA 95134-2020, (408) 944-0952. Contains information on the FLEX 10k and 8000 complex PLDs; MAX 9000, 7000, and 5000 complex PLDs; FLASHlogic; and EPLDs. A limited number of application notes are also included. More information may be found at the Altera Web site. [p. 322]

Connor, D. 1992. "Taking the first steps." *EDN*, April 9, p. 98. ISSN 0012-7515. The second part of this article, "Migrating to FPGAs: Any designer can do it," was published in *EDN*, April 23, 1992, p. 120. See also `http://www.ednmag.com`. Both articles are reprinted in the 1994 Actel Data Book. A description of designing, simulating, and testing a voicemail system using Viewlogic software. [p. 320]

Skahill, K. 1996. *VHDL for Programmable Logic.* Menlo Park, CA: Addison-Wesley, 593 p. ISBN 0-201-89573-0. TK7885.7.S55. Covers VHDL design for PLDs using Cypress Warp design system. [p. 320]

Xilinx. 1996. *The Programmable Logic Data Book.* No catalog information. Available from Xilinx Corporation, 2100 Logic Drive, San Jose, CA 95124-3400, (408) 559-7778. Contains details of XC9500, XC7300, and XC7200 CPLDs; XC5200, XC4000, XC3000 LCA FPGAs; and XC6200 sea-of-gates FPGAs. Earlier editions of this data book (the 1994 edition, for example) contained a section titled "Best of XCELL" that contained extremely useful design information. Much of this design material is now only available online, at the Xilinx Web site. [p. 322]

LOW-LEVEL DESIGN ENTRY

9

The purpose of **design entry** is to describe a microelectronic system to a set of **electronic-design automation** (**EDA**) tools. Electronic systems used to be, and many still are, constructed from off-the-shelf components, such as TTL ICs. Design entry for these systems now usually consists of drawing a picture, a **schematic**. The schematic shows how all the components are connected together, the **connectivity** of an ASIC. This type of design-entry process is called **schematic entry**, or **schematic capture**. A circuit schematic describes an ASIC in the same way an architect's plan describes a building.

The circuit schematic is a picture, an easy format for us to understand and use, but computers need to work with an ASCII or binary version of the schematic that we call a **netlist**. The output of a schematic-entry tool is thus a netlist file that contains a description of all the components in a design and their interconnections.

Not all the design information may be conveyed in a circuit schematic or netlist, because not all of the functions of an ASIC are described by the connectivity information. For example, suppose we use a programmable ASIC for some random logic functions. Part of the ASIC might be designed using a text language. In this case design entry also includes writing the code. What if an ASIC in our system contains a programmable memory (PROM)? Is the PROM microcode, the '1's and '0's, part of design entry? The operation of our system is certainly dependent on the correct programming of the PROM. So perhaps the PROM code ought to be considered part of design entry. On the other hand nobody would consider the operating-system code that is loaded into a RAM on an ASIC to be a part of design entry. Obviously, then,

there are several different forms of design entry. In each case it is important to make sure that you have completely specified the system—not only so that it can be correctly constructed, but so that someone else can understand how the system is put together. Design entry is thus an important part of **documentation**.

Until recently most ASIC design entry used schematic entry. As ASICs have become more complex, other design-entry methods are becoming common. Alternative design-entry methods can use graphical methods, such as a schematic, or text files, such as a programming language. Using a **hardware description language** (**HDL**) for design entry allows us to generate netlists directly using **logic synthesis**. We will concentrate on **low-level design-entry** methods together with their advantages and disadvantages in this chapter.

9.1 Schematic Entry

Schematic entry is the most common method of design entry for ASICs and is likely to be useful in one form or another for some time. HDLs are replacing conventional gate-level schematic entry, but new graphical tools based on schematic entry are now being used to create large amounts of HDL code.

Circuit schematics are drawn on **schematic sheets**. Standard schematic sheet sizes (Table 9.1) are ANSI A–E (more common in the United States) and ISO A4–A0 (more common in Europe). Usually a **frame** or **border** is drawn around the schematic containing boxes that list the name and number of the schematic page, the designer, the date of the drawing, and a list of any modifications or changes.

TABLE 9.1 ANSI (American National Standards Institute) and ISO (International Standards Organization) schematic sheet sizes.

ANSI sheet	Size (inches)	ISO sheet	Size (cm)
A	8.5 × 11	A5	21.0 × 14.8
B	11 × 17	A4	29.7 × 21.0
C	17 × 22	A3	42.0 × 29.7
D	22 × 34	A2	59.4 × 42.0
E	34 × 44	A1	84.0 × 59.4
		A0	118.9 × 84.0

Figure 9.1 shows the "spades" and "shovels," the recognized symbols for AND, NAND, OR, and NOR gates. One of the problems with these recommendations is that the corner points of the shapes do not always lie on a grid point (using a reasonable grid size).