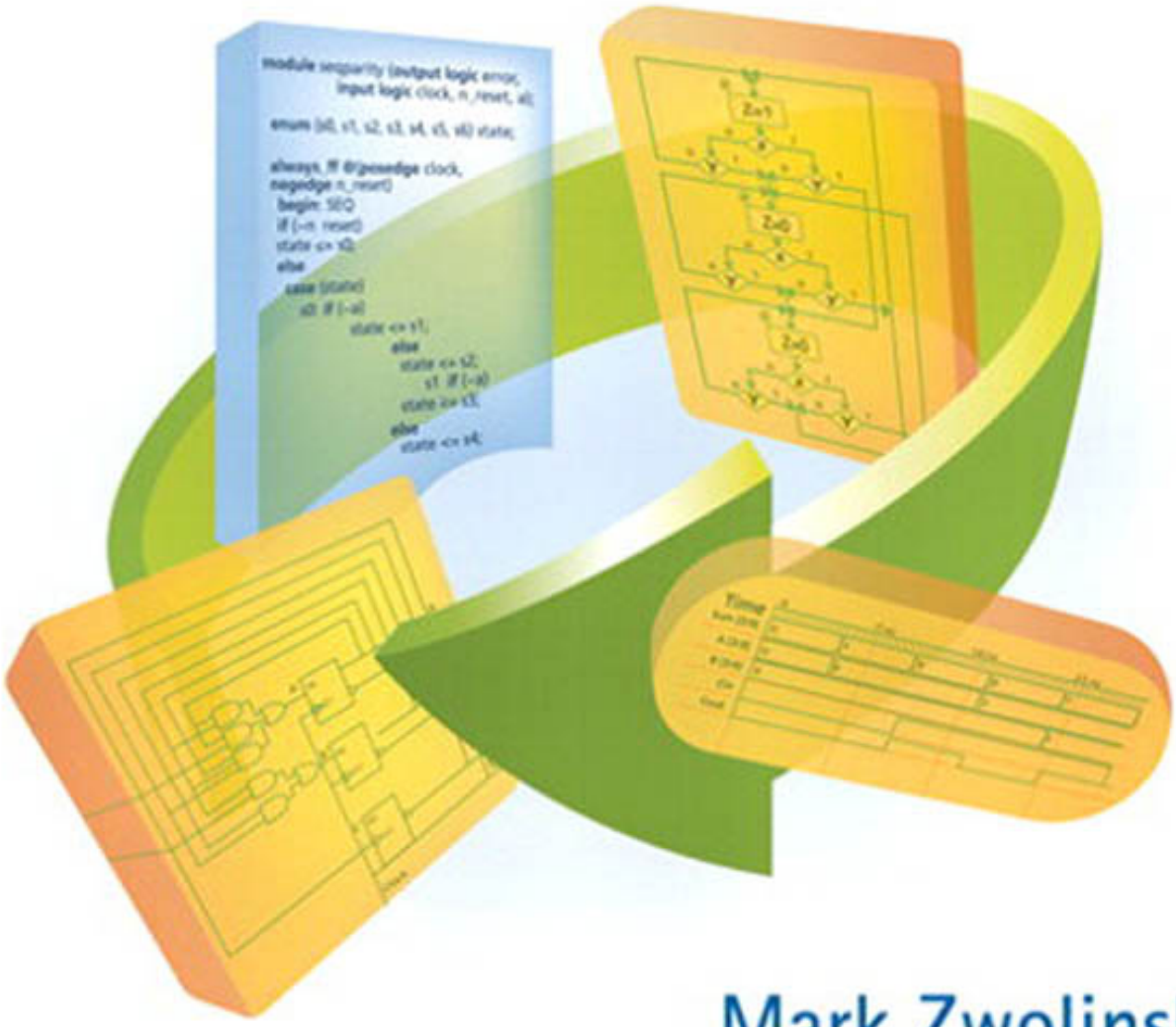
PRENTICE HALL

Digital System Design with SystemVerilog



Mark Zwolinski

Prentice Hall Modern Semiconductor Design Series

Library of Congress Cataloging-in-Publication Data
Zwolinski, Mark.
Digital system design with SystemVerilog / Mark Zwolinski.
p. cm.
Includes bibliographical references and index.
ISBN 0-13-704579-4 (hardback : alk. paper)
1. Verilog (Computer hardware description language) 2. Electronic digital computers–Design and construction. 3. Computer simulation. I. Title.

TK7885.7.Z86 2009
621.390285'53—dc22 2009034771

ISBN 13: 978-0-13-704579-2
ISBN 10: 0-13-704579-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, October 2009

Preface

About This Book

When *Digital System Design with VHDL* was published, the idea of combining a text on digital design with one on a hardware description language seemed novel. At about the same time, several other books with similar themes were published. *Digital System Design with VHDL* has now been adopted by several universities as a core text and has been translated into Polish, Chinese, Japanese, and Italian. I had thought about writing *Digital System Design with Verilog*, but I had (and still have) some doubts about using Verilog as a teaching language despite its widespread use. Soon after the second edition of *Digital System Design with VHDL* was published, a new hardware description language appeared—SystemVerilog. This new language removed many of my doubts about Verilog and even offered some noticeable advantages over VHDL. So the success of the first book and the appearance of the new language convinced me that the time had come for a new edition.

This book is intended as a student textbook for both undergraduate and postgraduate students. The majority of Verilog and SystemVerilog books are aimed at practicing engineers. Therefore, some features of SystemVerilog are not described at all in this book. Equally, aspects of digital design are covered that would not be included in a typical SystemVerilog book.

Syllabuses for electrical, electronic, and computer engineering degrees vary between countries and between universities or colleges. The material in this book has been developed over a number of years for second- and third-year undergraduates and for postgraduate students. It is assumed that students will be familiar with the principles of Boolean algebra and combinational logic design. At the University of Southampton, UK, the first-year undergraduate syllabus also includes introductions to synchronous sequential design and programmable logic. This book therefore builds upon these foundations. It has often been assumed that topics such as SystemVerilog are too specialized for second-year teaching and are best left to final year or postgraduate courses. There are several good reasons why SystemVerilog

should be introduced earlier into the curriculum. With increasing integrated circuit complexity, there is a need for graduates with knowledge of SystemVerilog and the associated design tools. If left to the final year, there is little or no time for the student to apply such knowledge in project work. Second, conversations with colleagues from many countries suggest that today's students are opting for computer science or computer engineering courses in preference to electrical or electronic engineering. SystemVerilog offers a means to interest computing-oriented students in hardware design. Finally, simulation and synthesis tools and FPGA design kits are now mature and available relatively inexpensively to educational establishments on PC platforms.

Structure of This Book

Chapter 1 introduces the ideas behind this book, namely the use of electronic design automation tools and CMOS and programmable logic technology. We also consider some engineering problems, such as noise margins and fan-out. In Chapter 2, the principles of Boolean algebra and combinational logic design are reviewed. The important matter of timing and the associated problem of hazards are discussed. Some basic techniques for representing data are discussed.

SystemVerilog is introduced in Chapter 3 through basic logic gate models. The importance of documented code is emphasized. We show how to construct netlists of basic gates and how to model delays through gates. We also discuss parameterized models. The idea of using SystemVerilog to verify models with testbenches is introduced.

In Chapter 4, a variety of modeling techniques are described. Combinational building blocks, buffers, decoders, encoders, multiplexers, adders, and parity checkers are modeled using a range of concurrent and sequential SystemVerilog coding constructs. The SystemVerilog models of hardware introduced in this chapter and in Chapters 5, 6, and 7 are, in principle, synthesizable, although discussion of exactly what is supported is deferred until Chapter 10. Testbench design styles are again discussed in Chapter 4. In addition, the IEEE dependency notation is introduced.

Chapter 5 introduces various sequential building blocks: latches, flip-flops, registers, counters, memory, and a sequential multiplier. The same style as Chapter 4 is used, with IEEE dependency notation, testbench design, and the introduction of SystemVerilog coding constructs.

Chapter 6 is probably the most important chapter of the book and discusses what might be considered the cornerstone of digital design: the design of finite state machines. The ASM chart notation is used. The design process from ASM chart to

D flip-flops and next state and output logic is described. SystemVerilog models of state machines are introduced.

In Chapter 7, the concepts of the previous three chapters are combined. The ASM chart notation is extended to include coupled state machines and registered outputs, and hence to datapath-controller partitioning. From this, we explain the idea of instructions in hardware terms and go on to model a very basic microprocessor in SystemVerilog. This provides a vehicle to introduce interfaces and packages.

The design of testbenches is discussed in more detail in Chapter 8. After recapping the techniques given in earlier chapters, we go on to discuss testbench architecture, constrained random test generation, and assertion-based verification.

SystemVerilog remains primarily a modeling language. Chapter 9 describes the operation of a SystemVerilog simulator. The idea of event-driven simulation is first explained, and the specific features of a SystemVerilog simulator are then discussed.

The other, increasingly important, role of SystemVerilog is as a language for describing synthesis models, as discussed in Chapter 10. The dominant type of synthesis tool available today is for RTL synthesis. Such tools can infer the existence of flip-flops and latches from a SystemVerilog model. These constructs are described. Conversely, flip-flops can be created in error if the description is poorly written, and common pitfalls are described. The synthesis process can be controlled by constraints. Because these constraints are outside of the language, they are discussed in general terms. Suitable constructs for FPGA synthesis are discussed. Finally, behavioral synthesis, which promises to become an important design technology, is briefly examined.

Chapters 11 and 12 are devoted to the topics of testing and design for test. This area has often been neglected, but is now recognized as being an important part of the design process. In Chapter 11, the idea of fault modeling is introduced. This is followed by test generation methods. The efficacy of a test can be determined by fault simulation.

In Chapter 12, three important design-for-test principles are described: scan path, built-in self-test (BIST), and boundary scan. This has always been a very dry subject, but a SystemVerilog simulator can be used, for example, to show how a BIST structure can generate different signatures for fault-free and faulty circuits.

We use SystemVerilog as a tool for exploring anomalous behavior in asynchronous sequential circuits in Chapter 13. Although the predominant design style is currently synchronous, it is likely that digital systems will increasingly consist of synchronous circuits communicating asynchronously with each other. We introduce the concept of the fundamental mode and show how to analyze and design

asynchronous circuits. We use SystemVerilog simulations to illustrate the problems of hazards, races, and setup and hold time violations. We also discuss the problem of metastability.

The final chapter introduces Verilog-AMS and mixed-signal modeling. Brief descriptions of digital-to-analog converters (DACs) and analog-to-digital converters (ADCs) are given. Verilog-AMS constructs to model such converters are given. We also introduce the idea of a phase-locked loop (PLL) here and give a simple mixed-signal model.

The Appendix briefly describes how SystemVerilog differs from earlier versions of Verilog.

At the end of each chapter a number of exercises have been included. These exercises are almost secondary to the implicit instruction in each chapter to simulate and, where appropriate, synthesize each SystemVerilog example. To perform these simulation and synthesis tasks, the reader may have to write his or her own testbenches and constraints files. The examples are available on the Web at zwolinski.org.

How to Use This Book

Obviously, this book can be used in a number of different ways, depending on the level of the course. At the University of Southampton, I have been using the material as follows.

Second Year of MEng/BEng in Electronic Engineering

Chapters 1 and 2 are review material, which the students would be expected to read independently. Lectures then cover the material of Chapters 3 through 7. Some of this material can be considered optional, such as Sections 5.3 and 5.7. Additionally, some constructs could be omitted if time is limited. The single-stuck fault model of Section 11.2 and the principles of test pattern generation in Section 11.3, together with the principles of scan design in Section 12.2, would also be covered in lectures.

Third Year of MEng/BEng in Electronic Engineering

Students would be expected to independently re-read Chapters 4 to 7. Lectures would cover Chapters 8 to 13. Verilog-AMS, Chapter 14, is currently covered in a fourth-year module.

In all years, students need to have access to a SystemVerilog simulator and an RTL synthesis tool in order to use the examples in the text. In the second year, a group

design exercise involving synthesis to an FPGA would be an excellent supplement to the material. In the third year at Southampton, all students do an individual project. Some of the individual projects will involve the use of SystemVerilog.

Web Resources

A Web site accompanies *Digital System Design with SystemVerilog* by Mark Zwolinski. Visit the site at zwolinski.org. Here you will find valuable teaching and learning material including all the SystemVerilog examples and links to sites with SystemVerilog tools.

2 Combinational Logic Design

Digital design is based on the processing of binary variables. In this chapter, we will review the principles of Boolean algebra and the minimization of Boolean expressions. Hazards and basic numbering systems will also be discussed.

2.1 Boolean Algebra

2.1.1 Values

Digital design uses a two-value algebra. Variables can take one of two values that can be represented by

ON and OFF,
TRUE and FALSE,
1 and 0.

2.1.2 Operators

The algebra of two values, known as Boolean algebra, after George Boole (1815–1864), has five basic operators. In decreasing order of precedence (i.e., in the absence

of parentheses, operations at the top of the list should be evaluated first) these are:

1. NOT
2. AND
3. OR
4. IMPLIES
5. EQUIVALENCE

The last two operators are not widely used in digital design. These operators can be used to form expressions. For example:

$$A = 1$$

$$B = C \text{ AND } 0$$

$$F = \overline{(A + B \cdot C)}$$

$$Z = (\bar{A} + B) \cdot (A + \bar{B})$$

The symbol "+" means "OR," "." means "AND," and the overbar, for example, "$\bar{A}$," means "NOT A."

2.1.3 Truth Tables

The meaning of an operator or expression can be described by listing all the possible values of the variables in that expression, together with the value of the expression in a *truth table*. The truth tables for the three basic operators are given in Tables 2.1, 2.2, and 2.3.

In digital design, three further operators are commonly used: NAND (Not AND), NOR (Not OR), and XOR (eXclusive OR); see Tables 2.4, 2.5, and 2.6.

The XNOR ($\overline{A \oplus B}$) operator is also used occasionally. XNOR is the same as EQUIVALENCE.

Table 2.1 NOT Operation

A	NOT A ($\bar{A}$)
0	1
1	0

Table 2.2 AND Operation

A	B	A AND B $(A \cdot B)$
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.3 OR Operation

A	B	A OR B $(A + B)$
0	0	0
0	1	1
1	0	1
1	1	1

Table 2.4 NAND Operation

A	B	A NAND B $\overline{(A \cdot B)}$
0	0	1
0	1	1
1	0	1
1	1	0

Table 2.5 NOR Operation

A	B	A NOR B $\overline{(A + B)}$
0	0	1
0	1	0
1	0	0
1	1	0

Table 2.6 XOR Operation

A	B	A XOR B $(A \oplus B)$
0	0	0
0	1	1
1	0	1
1	1	0

2.1.4 Rules of Boolean Algebra

There are a number of basic rules of Boolean algebra that follow from the precedence of the operators.

1. Commutativity

$$A + B = B + A$$
$$A \cdot B = B \cdot A$$

2. Associativity

$$A + (B + C) = (A + B) + C$$
$$A \cdot (B \cdot C) = (A \cdot B) \cdot C$$

3. Distributivity

$$A \cdot (B + C) = A \cdot B + A \cdot C$$

In addition, some basic relationships can be observed from the previous truth tables.

$$\bar{\bar{A}} = A$$

$$A \cdot 1 = A \qquad A + 0 = A$$

$$A \cdot 0 = 0 \qquad A + 1 = 1$$

$$A \cdot A = A \qquad A + A = A$$

$$A \cdot \bar{A} = 0 \qquad A + \bar{A} = 1$$

The right-hand column can be derived from the left-hand column by applying the *principle of duality*. The principle of duality states that if each AND is changed to an OR, each OR to an AND, each 1 to 0, and each 0 to 1, the value of the expression remains the same.

2.1.5 De Morgan's Law

There is a very important relationship that can be used to rewrite Boolean expressions in terms of NAND or NOR operations: de Morgan's Law. This is expressed as

$$\overline{(A \cdot B)} = \bar{A} + \bar{B} \quad \text{or} \quad \overline{(A + B)} = \bar{A} \cdot \bar{B}$$

2.1.6 Shannon's Expansion Theorem

Shannon's expansion theorem can be used to manipulate Boolean expressions.

$$F(A, B, C, D, \ldots) = A \cdot F(1, B, C, D, \ldots) + \bar{A} \cdot F(0, \mathrm{B}, \mathrm{C}, \mathrm{D}, \ldots)$$
$$= (A + F(0, B, C, D, \ldots)) \cdot (\bar{A} + F(1, B, C, D, \ldots))$$

$F(1, B, C, D, \ldots)$ means that all instances of A in F are replaced by a logic 1.

2.2 Logic Gates

The basic symbols for one and two input logic gates are shown in Figure 2.1. Three and more inputs are shown by adding extra inputs (but note that there is no such thing as a three input XOR gate). The ANSI/IEEE symbols can be used instead

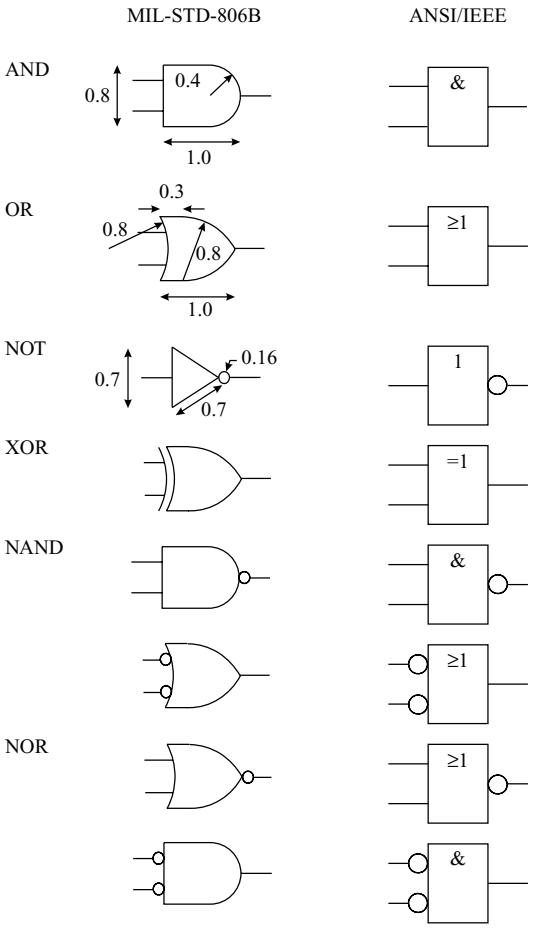


Figure 2.1 Logic symbols.

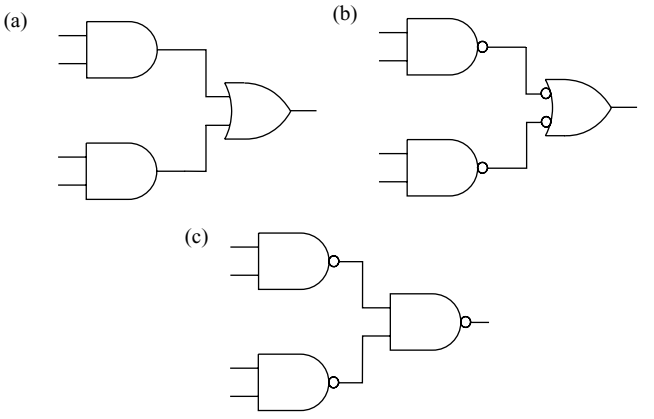


Figure 2.2 Equivalent circuit representations.

of the traditional "spade"-shaped symbols, but are "not preferred" according to IEEE Standard 91-1984. As will be seen in Chapter 3, IEEE notation is useful for describing complex logic blocks, but simple sketches are often clearer if done with the traditional symbols. A circle shows logic inversion. Note that there are two forms of the NAND and NOR gates. From de Morgan's law, it can be seen that the two forms are equivalent in each case.

In drawing circuit diagrams, it is desirable, for clarity, to choose the form of a logic gate that allows inverting circles to be joined. The circuits of Figure 2.2 are identical in function. If the circuit of Figure 2.2(a) is to be implemented using NAND gates, the diagram of Figure 2.2(b) may be preferable to that of Figure 2.2(c) because the function of the circuit is clearer.

2.3 Combinational Logic Design

The values of the output variables of combinational logic are dependent only on the input values and are independent of previous input values or states. Sequential logic, on the other hand, has outputs that depend on the previous states of the system. The design of sequential systems is described in Chapter 6.

The major design objective is usually to minimize the cost of the hardware needed to implement a logic function. That cost can usually be expressed in terms of the number of gates, although for technologies such as programmable logic, there are other limitations, such as the number of terms that may be implemented. Other design objectives may include testability (discussed in detail in Chapter 12) and reliability.

Table 2.7 Minterms and Maxterms

A	B	C	Z	
0	0	0	1	m_0
0	0	1	1	m_1
0	1	0	0	M_2
0	1	1	0	M_3
1	0	0	0	M_4
1	0	1	1	m_5
1	1	0	0	M_6
1	1	1	1	m_7

Before describing the logic design process, some terms have to be defined. In these definitions, it is assumed that we are designing a piece of combinational logic with a number of input variables and a single output.

A *minterm* is a Boolean AND function containing exactly one instance of each input variable or its inverse. A *maxterm* is a Boolean OR function with exactly one instance of each variable or its inverse. For a combinational logic circuit with n input variables, there are 2^n possible minterms and 2^n possible maxterms. If the logic function is true at row i of the standard truth table, that minterm exists and is designated by m_i. If the logic function is false at row i of the standard truth table, that maxterm exists and is designated by M_i. For example, Table 2.7 defines a logic function. The final column shows the minterms and maxterms for the function.

The logic function may be described by the logic OR of its minterms:

$$Z = m_0 + m_1 + m_5 + m_7$$

A function expressed as a logical OR of distinct minterms is in *sum of products* form.

$$Z = \bar{A} \cdot \bar{B} \cdot \bar{C} + \bar{A} \cdot \bar{B} \cdot C + A \cdot \bar{B} \cdot C + A \cdot B \cdot C$$

Each variable is inverted if there is a corresponding 0 in the truth table and not inverted if there is a 1.

Similarly, the logic function may be described by the logical AND of its maxterms.

$$Z = M_2 \cdot M_3 \cdot M_4 \cdot M_6$$

A function expressed as a logical AND of distinct maxterms is in *product of sums* form.

$$Z = (A + \bar{B} + C) \cdot (A + \bar{B} + \bar{C}) \cdot (\bar{A} + B + C) \cdot (\bar{A} + \bar{B} + C)$$

Table 2.8 Truth Table for $Z = A + \bar{A} \cdot \bar{B}$

A	B	Z
0	0	1
0	1	0
1	0	1
1	1	1

Each variable is inverted if there is a corresponding 1 in the truth table and not inverted if there is a 0.

An *implicant* is a term that covers at least one true value and no false values of a function. For example, the function $Z = A + \bar{A} \cdot \bar{B}$ is shown in Table 2.8.

The implicants of this function are $A \cdot B$, A, $\bar{B}$, $\bar{A} \cdot \bar{B}$, $A \cdot \bar{B}$. The non-implicants are $\bar{A}$, B, $\bar{A} \cdot B$.

A *prime implicant* is an implicant that covers one or more minterms of a function, such that the minterms are not all covered by another single implicant. In the example above, A, $\bar{B}$ are prime implicants. The other implicants are all covered by one of the prime implicants. An *essential prime implicant* is a prime implicant that covers an implicant not covered by any other prime implicant. Thus, A, $\bar{B}$ are essential prime implicants.

2.3.1 Logic Minimization

The function of a combinational logic circuit can be described by one or more Boolean expressions. These expressions can be derived from the specification of the system. It is very likely, however, that these expressions are not initially stated in their simplest form. Therefore, if these expressions were directly implemented as logic gates, the amount of hardware required would not be minimal. Therefore, we seek to simplify the Boolean expressions and hence minimize the number of gates needed. Another way of stating this is to say that we are trying to find the set of prime implicants of a function that is necessary to fully describe the function.

In principle, it is possible to simplify Boolean expressions by applying the various rules of Boolean algebra described in Section 2.1. It does not take long, however, to realize that this approach is slow and error prone. Other techniques have to be employed. The technique described here, *Karnaugh maps*, is a graphical method, although it is effectively limited to problems with six or fewer variables. The *Quine-McCluskey* algorithm is a tabular method, which is not limited in the number of variables and is well suited to tackling problems with more than one

output. Quine-McCluskey can be performed by hand, but it is generally less easy than the Karnaugh map method. It is better implemented as a computer program. Logic minimization belongs, however, to the *NP-complete* class of problems. This means that as the number of variables increases, the time to find a solution increases exponentially. Therefore, heuristic methods have been developed that find acceptable, but possibly less than optimal, solutions. The *Espresso* program implements heuristic methods that reduce to the Quine-McCluskey algorithm for small problems. Espresso has been used in a number of logic synthesis systems. Therefore, the approach adopted here is to use Karnaugh maps for small problems with a single output and up to six inputs. In general, it makes sense to use an EDA program to solve larger problems.

The Karnaugh map (or K-map, for short) method generates a solution in sum-of-products or product-of-sums form. Such a solution can be implemented directly as two-level AND-OR or OR-AND logic (ignoring the cost of generating the inverse values of inputs). AND-OR logic is equivalent to NAND-NAND logic, and OR-AND logic is equivalent to NOR-NOR logic. Sometimes, a cheaper solution (in terms of the number of gates) can be found by factorizing the two-level, minimized expression to generate more levels of logic—two-level minimization must be performed before any such factorization. Again, we shall assume that if such factorization is to be performed, it will be done using an EDA program, such as *SIS*.

2.3.2 Karnaugh Maps

A Karnaugh map is effectively another way to write a truth table. For example, the Karnaugh map of a general two-input truth table is shown in Figure 2.3.

Similarly, three- and four-input Karnaugh maps are shown in Figures 2.4 and 2.5, respectively. Note that along the top edge of the three-variable Karnaugh map and along both edges of the four-variable map, only one variable changes at a time—the sequence is 00, 01, 11, 10, not the normal binary counting sequence. Hence, for example, the columns in which *A* is true are adjacent. Therefore, the left and right edges, and the top and bottom in the four-variable map, are also adjacent—*B*

A	B	Z
0	0	Z_0
0	1	Z_1
1	0	Z_2
1	1	Z_3

Z:

B \ A	0	1
0	Z_0	Z_2
1	Z_1	Z_3

Figure 2.3 Two-input Karnaugh map.

Z: C \ AB	00	01	11	10
0	Z_0	Z_2	Z_6	Z_4
1	Z_1	Z_3	Z_7	Z_5

Figure 2.4 Three-input Karnaugh map.

Z: CD \ AB	00	01	11	10
00	Z_0	Z_4	Z_{12}	Z_8
01	Z_1	Z_5	Z_{13}	Z_9
11	Z_3	Z_7	Z_{15}	Z_{11}
10	Z_2	Z_6	Z_{14}	Z_{10}

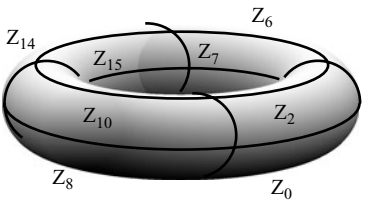


Figure 2.5 Four-input Karnaugh map.

is false in the leftmost and rightmost columns. The three variable map is therefore really a tube, and the four-variable map is a torus, as shown in Figure 2.5. Of course, the maps are drawn as squares for convenience!

A five-variable Karnaugh map is drawn as 2 four-variable maps, one representing the truth table when the fifth variable, E, is false, and the other when E is true. Squares at the same coordinates on both maps are considered to be adjacent. Similarly, a six-variable Karnaugh map is drawn as 4 four-variable maps corresponding to $\bar{E} \cdot \bar{F}$, $\bar{E} \cdot F$, $E \cdot \bar{F}$, and $E \cdot F$, respectively. For this to work, the Karnaugh maps have to be arranged in the pattern as the entries in the two-variable map. Hence, squares at the same location in adjacent maps can be considered adjacent. In practice, therefore, it is not feasible to consider Karnaugh maps with more than six variables.

Implicants can be read from Karnaugh maps by circling groups of $1, 2, 4, 8, \ldots 2^n$ true values. For example, the function $Z = \bar{A} \cdot \bar{B} + \bar{A} \cdot B$ can be expressed as shown in Table 2.9.

Table 2.9 Truth Table for $Z = \bar{A} \cdot \bar{B} + \bar{A} \cdot B$

A	B	Z
0	0	1
0	1	1
1	0	0
1	1	0

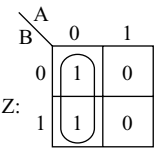


Figure 2.6 Karnaugh map for a two-input function.

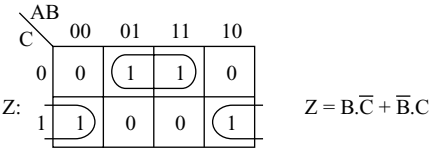


Figure 2.7 Groupings on a three-input Karnaugh map.

The corresponding Karnaugh map is shown in Figure 2.6. We can now circle the two adjacent 1s as shown. This grouping represents the function $Z = \bar{A}$ because it lies in the column $A = 0$, and because within the grouping, B takes both 0 and 1 values and hence we do not care about its value. Therefore, by grouping patterns of 1s, logic functions can be minimized. Examples of three- and four-variable Karnaugh maps are shown in Figures 2.7 and 2.8. In both cases, by considering that the edges of the Karnaugh maps are adjacent, groupings can be made that include 1s at two or four edges.

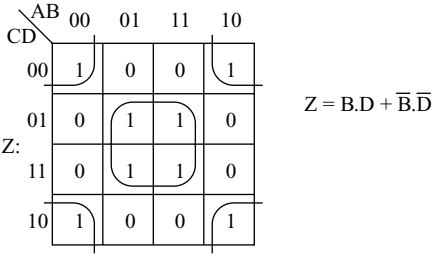


Figure 2.8 Groupings on a four-input Karnaugh map.

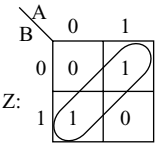


Figure 2.9 Exclusive OR grouping on a Karnaugh map.

The rules for reading *prime implicants* from a Karnaugh map are as follows.

- Circle the largest possible groups.
- Avoid circles inside circles (see the definition of a prime implicant).
- Circle 1s and read the sum of products for Z.
- Circle 0s and read the sum of products for $\bar{Z}$.
- Circle 0s and read the product of sums for Z.
- Circle 1s and read the product of sums for $\bar{Z}$.

Diagonal pairs, as shown in Figure 2.9, correspond to XOR functions.

The Karnaugh map of Figure 2.10 has three prime implicants circled. The function can be read as $Z = B \cdot \bar{C} \cdot D + \bar{A} \cdot C \cdot D + \bar{A} \cdot B \cdot D$. The vertical grouping, shown with a dashed line, covers 1s covered by the other groupings. This grouping is therefore *redundant* and can be omitted. Hence, the function can be read as $Z = B \cdot \bar{C} \cdot D + \bar{A} \cdot C \cdot D$.

Assuming that all the prime implicants have been correctly identified, the minimal form of the function can be read by selecting all the essential prime implicants (i.e., those circles that circle 1s—or 0s—not circled by any other group), together with sufficient other prime implicants needed to cover all the 1s (or 0s). Redundant groupings can be ignored, but under some circumstances it may be desirable to include them.

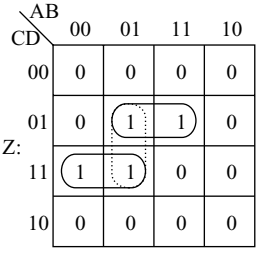


Figure 2.10 Redundant grouping on a Karnaugh map.

A	B	Z
0	0	1
0	1	-
1	0	0
1	1	1

Z:

B \ A	0	1
0	1	0
1	-	1

Figure 2.11 "Don't care" on a Karnaugh map.

Incompletely specified functions have "don't cares" in the truth tables. These don't cares correspond to input combinations that will not (or should not) occur. For example, consider the truth table of Figure 2.11.

The don't care entries can be included or excluded from groups as convenient, in order to get the largest possible groupings, and hence the smallest number of implicants. In the example, we could treat the don't care as a 0 and read $Z = \bar{A} \cdot \bar{B} + A \cdot B$, or treat the don't care as a 1 and read $Z = \bar{A} + B$.

2.4 Timing

The previous section dealt with minimizing Boolean expressions. The minimized Boolean expressions can then be directly implemented as networks of gates or on programmable logic. All gates have a finite delay between a change at an input and a change at an output. If gates are used, therefore, different paths may exist in the network, with different delays. This may cause problems.

To understand the difficulties, it is helpful to draw a *timing diagram*. This is a diagram of the input and output waveforms as a function of time. For example, Figure 2.12 shows the timing diagram for an inverter. Note the stylized (finite) rise

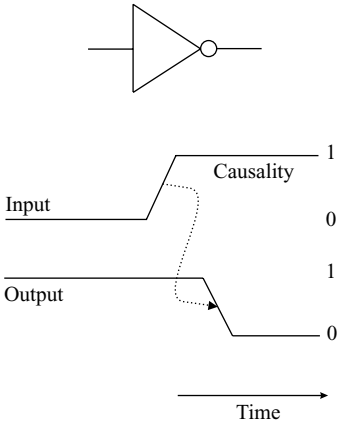


Figure 2.12 Timing diagram for inverter.

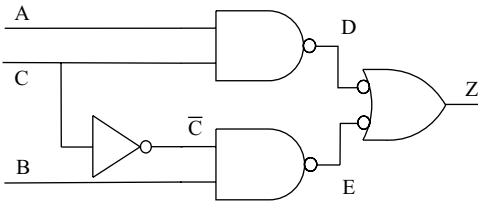


Figure 2.13 Circuit with static 1 hazard.

and fall times. An arrow shows causality, that is, the fact that the change in the output results from a change in the input.

A more complex circuit would implement the function

$$Z = A \cdot C + B \cdot \bar{C}$$

The value of $\bar{C}$ is generated from C by an inverter. A possible implementation of this function is therefore given in Figure 2.13. In practice, the delay through each gate and through each type of gate would be slightly different. For simplicity, however, let us assume that the delay through each gate is one unit of time. To start with, let $A = 1$, $B = 1$. The output, Z, should be at 1 irrespective of the value of C. Let us see, by way of the timing diagram in Figure 2.14, what happens when C changes from 1 to 0. One unit of time after C changes $\bar{C}$ and D change to 1. In turn, these changes cause E and Z to change to 0 another unit of time later. Finally, the change in E causes Z to change back to 1 a further unit of time later. This change

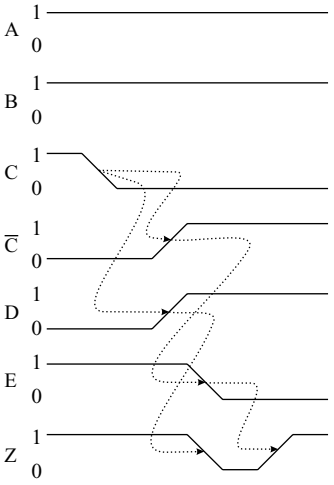


Figure 2.14 Timing diagram for the circuit of Figure 2.13.

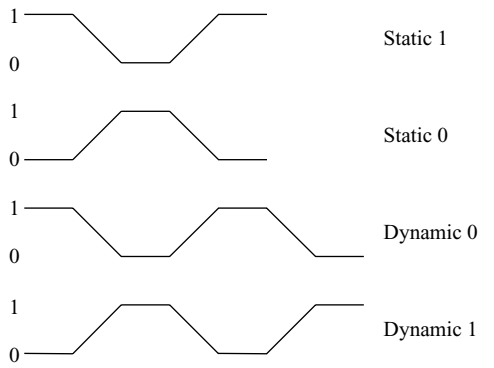


Figure 2.15 Types of hazard.

in Z from 1 to 0 and back to 1 is known as a *hazard*. A hazard occurs as a result of delays in a circuit.

Figure 2.15 shows the different types of hazard that can occur. The hazard in the circuit of Figure 2.13 is a static 1 hazard. Static 1 hazards can only occur in AND-OR or NAND-NAND logic. Static 0 hazards can only occur in OR-AND or NOR-NOR logic. Dynamic hazards do not occur in two-level circuits. They require three or more unequal signal paths. Dynamic hazards are often caused by poor factorization in multi-level minimization.

Static hazards, on the other hand, can be avoided by designing with redundant logic. For example, the Karnaugh map of the circuit function of Figure 2.13 is shown in Figure 2.16. The redundant prime implicant is shown as a dashed circle. The redundant gate corresponding to this prime implicant can be introduced to eliminate the hazard. The circuit function is therefore

$$Z = A \cdot C + B \cdot \bar{C} + A \cdot B$$

The circuit is shown in Figure 2.17. Now, F is independent of C. If $A = B = 1$, $F = 0$. F stays at 0 while C changes; therefore, Z stays at 1. (See Section 11.3.2 for another good reason why circuits with redundancy should be avoided.)

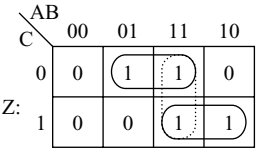


Figure 2.16 Redundant term on a Karnaugh map.

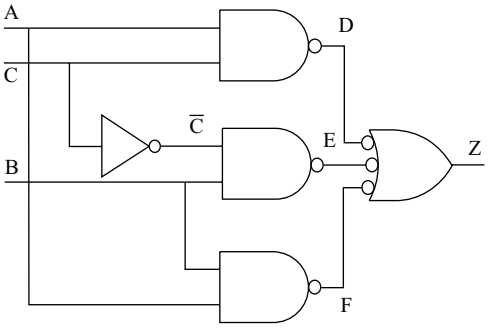


Figure 2.17 Hazard-free circuit.

2.5 Number Codes

Digital signals are either control signals of some kind or information. In general, information takes the form of numbers or characters. These numbers and characters have to be coded in a form suitable for storage and manipulation by digital hardware. Thus, one integer or one character may be represented by a set of bits. From the point of view of a computer or other digital system, no one system of coding is better than another. There do, however, need to be standards, so that different systems can communicate. The standards that have emerged are generally also designed such that a human being can interpret the data if necessary.

2.5.1 Integers

The simplest form of coding is that of positive integers. For example, a set of three bits would allow us to represent the decimal integers 0 to 7. In base 2 arithmetic, 000_2 represents 0_{10}, 011_2 represents 3_{10}, and 111_2 represents 7_{10}. As with decimal notation, the most significant bit is on the left.

For the benefit of human beings, strings of bits may be grouped into sets of three or four and written using *octal* (base 8) or *hexadecimal* (base 16) notation. For example, 66_8 is equal to 110 110_2 or 54_{10}. For hexadecimal notation, the letters A to F represent the decimal numbers 10 to 15. For example, EDA_{16} is 1110 1101 1010_2 or 7332_8 or 3802_{10}.

The simple translation of a decimal number into bits is sufficient for zero and positive integers. Negative integers require additional information. The simplest approach is to set aside one bit as a sign bit. Therefore, 0 110_2 might represent $+6_{10}$, while 1 110_2 would represent -6_{10}. While this makes translation between binary and decimal numbers simple, the arithmetic operations of addition and subtraction

require that the sign bits be checked before an operation can be performed on two numbers. It is common, therefore, to use a different notation for signed integers: *two's complement*. The principle of two's complement notation is that the code for $-b$, where b is a binary number represented using n bits, is the code given by $2^n - b$. For example, -6_{10} is represented by $10000_2 - 0110_2$, which is 1010_2. The same result is obtained by inverting all the bits and adding 1: -6_{10} is $1001_2 + 1 = 1010_2$.

The advantage of two's complement notation is that addition and subtraction may be performed using exactly the same hardware as for unsigned arithmetic; no sign checking is needed. The major disadvantage is that multiplication and division become much more complicated. Booth's algorithm, described in Section 5.7, is a technique for multiplying two's complement numbers.

2.5.2 Fixed Point Numbers

For many applications, non-integer data needs to be stored and manipulated. The binary representation of a *fixed-point* number is exactly the same as for an integer number, except that there is an implicit "decimal" point. For example, 6.25 is equal to $2^2 + 2^1 + 2^{-2}$ or 110.01_2. Instead of representing the point, the number 11001_2 (25_{10}) is stored with the implicit knowledge that it and the results of any operations involving it have to be divided by 2^2 to obtain the true value. Notice that all operations, including two's complement representations, are the same as for integer numbers.

2.5.3 Floating-Point Numbers

The number of bits that have been allocated to represent fractions limits the range of fixed point numbers. *Floating-point* numbers allow a much wider range of accuracy. In general, floating-point operations are only performed using specialized hardware because they are very computationally expensive. A typical *single precision* floating-point number has 32 bits, of which 1 is the sign bit (s), 8 are the exponent (e), biased by an offset ($2^e - 1 = 127$), and the remaining 23 are the mantissa (m), such that a decimal number is represented as

$$(-1)^s \times 1 \cdot m \times 2^e$$

IEEE standard 754-1985 defines formats for 32-, 64-, and 128-bit floating-point numbers, with special patterns for $\pm\infty$ and the results of invalid operations, such as $\sqrt{-1}$.

2.5.4 Alphanumeric Characters

Characters are commonly represented by 7 or 8 bits. The ASCII code is widely used. Seven bits allow the basic Latin alphabet in upper and lower cases, together with various punctuation symbols and control codes to be represented. For example, the letter A is represented by 1000001. For accented characters, 8-bit codes are commonly used. Manipulation of text is normally performed using general purpose computers rather than specialized digital hardware. Non-European languages may use 16 or 32 bits to represent individual characters.

2.5.5 Gray Codes

In the normal binary counting sequence, the transition from 0111 (7_{10}) to 1000 (8_{10}) causes 3 bits to change. In some circumstances, it may be undesirable that several bits should change at once because the bit changes may not occur at exactly the same time. The intermediate values might generate spurious warnings. A *Gray* code is one in which only 1 bit changes at a time. For example a 3-bit Gray code would count through the following sequence (other Gray codes can also be derived):

000
001
011
010
110
111
101
100

Note that the sequence of bits on a K-map is a Gray code. Another application of Gray codes is as a position encoder on a rotating shaft, as shown in Figure 2.18. Only 1-bit changes at each transition, so missed transitions are easily spotted.

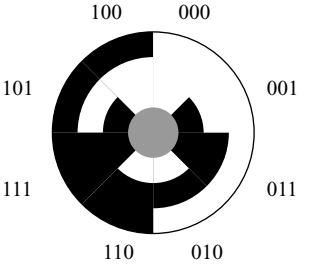


Figure 2.18 Gray code as shaft encoder.

Table 2.10 Two-Dimensional Parity

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Parity
Word 0	0	1	0	1	0	1	1	0	0
Word 1	0	1	0	0	1	0	0	0	0
Word 2	0	1	0	1	0	0	1	1	0
Word 3	0	1	0	0	1	0	0	1	1
Word 4	0	1	0	0	0	0	1	1	1
Word 5	0	1	0	0	1	0	0	0	0
Word 6	0	1	0	0	0	1	0	0	0
Word 7	0	1	0	0	1	1	0	0	1
Parity	0	0	0	0	0	1	1	1	1

2.5.6 Parity Bits

When data are transmitted either by wire or by using radio communications, there is always the possibility that noise may cause a bit to be misinterpreted. At the very least, it is desirable to know that an error has occurred, and it may be desirable to transmit sufficient information to allow any error to be corrected.

The simplest form of error detection is to use a parity bit with each word of data. For each 8 bits of data, a ninth bit is sent that is 0 if there is an even number of 1s in the data word (even parity) or 1 otherwise. Alternatively, odd parity can be used; in which case, the parity bit is inverted. This is sufficient if the chances of an error occurring are low. We cannot tell which bit is in error, but knowing that an error has occurred means that the data can be transmitted again. Unfortunately, if two errors occur, the parity bit might appear to be correct. A single error can be corrected by using a two-dimensional parity scheme, in which every ninth word is itself a set of parity bits, as shown in Table 2.10. If a single error occurs, both the row parity and column parity will be incorrect, allowing the erroneous bit to be identified and corrected. Certain multiple errors are also detectable and correctable.

By using a greater number of parity bits, each derived from part of the word, multiple errors can be detected and corrected. The simplest forms of such codes were derived by Hamming in 1948. Better codes were derived by Reed and Solomon in 1960.

Summary

Digital design is based on Boolean algebra. The rules of Boolean algebra allow logical expressions to be simplified. The basic logical operators can be implemented as digital building blocks—gates. Graphical methods, such as Karnaugh maps, are

suitable tools for finding the minimal forms of Boolean expressions with fewer than six variables. Larger problems can be tackled with computer-based methods. Gates have delays, which means that non-minimal forms of Boolean expressions may be needed to prevent timing problems, known as hazards. Data can be represented using sets of bits. Different types of data can be encoded to allow manipulation. Error detecting codes are used when data is transmitted over radio or other networks.

Further Reading

The principles of Boolean algebra and Boolean minimization are covered in many books on digital design. Recommended are those by Wakerly [25] and Hill and Peterson [6]. De Micheli [10] describes the Espresso algorithm, which sits at the heart of many logic optimization software packages. Espresso may be downloaded from www-cad.eecs.berkeley.edu/.

Error detection and correction codes are widely used in communications systems. Descriptions of these codes can be found in, for example, Hamming [8].

Exercises

2.1 Derive Boolean expressions for the circuits of Figure 2.19; use truth tables to discover if they are equivalent.

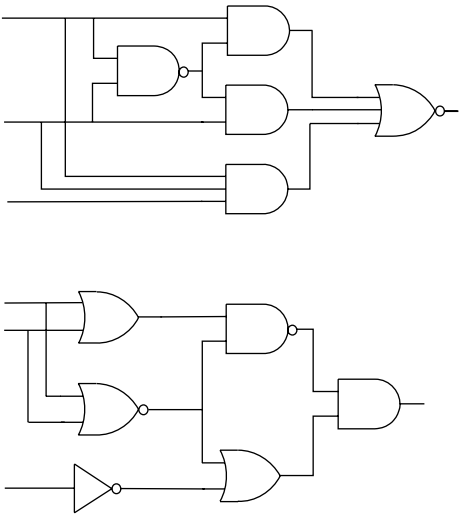


Figure 2.19 Circuits for Exercise 2.1.

2.2 Minimize

(a) $Z = m_0 + m_1 + m_2 + m_5 + m_7 + m_8 + m_{10} + m_{14} + m_{15}$

(b) $Z = m_3 + m_4 + m_5 + m_7 + m_9 + m_{13} + m_{14} + m_{15}$

2.3 Describe two ways of representing negative binary numbers. What are the advantages and disadvantages of each method?

2.4 A floating-point decimal number may be represented as:

$$(-1)^s \times 1 \cdot m \times 2^e$$

Explain what the binary numbers s, m, and e represent. How many bits would typically be used for s, m, and e in a single precision floating-point number?

Index

B

C

D

E

F

G

H

N

Q

R

S

T

U

V

W

X