Introduction to Logic Synthesis using Verilog HDL

Introduction to Logic Synthesis using Verilog HDL
Robert B. Reese and Mitchell A. Thornton
www.morganclaypool.com

ISBN-10: 1598291068 paperback
ISBN-13: 9781598291063 paperback

ISBN-10: 1598291076 ebook
ISBN-13: 9781598291070 ebook

DOI10.2200/S00060ED1V01Y200610DCS006

A Publication in the Morgan & Claypool Publishers' series
SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #6

Lecture #6
Series Editor and Affliation: Mitchell A. Thornton, Southern Methodist University

Series ISSN: 1930-3166 print
Series ISSN: 1930-3174 electronic

First Edition
10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Introduction to Logic Synthesis using Verilog HDL

Robert B. Reese
Mississippi State University

Mitchell A. Thornton
Southern Methodist University

SYNTHESIS LECTURES ON DIGITAL CIRCUITS AND SYSTEMS #6

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Introduction to Logic Synthesis Using Verilog HDL explains how to write accurate Verilog descriptions of digital systems that can be synthesized into digital system net lists with desirable characteristics. The book contains numerous Verilog examples that begin with simple combinational networks and progress to synchronous sequential logic systems. Common pitfalls in the development of synthesizable Verilog HDL are also discussed along with methods for avoiding them. The target audience is any one with a basic understanding of digital logic principles who wishes to learn how to model digital systems in the Verilog HDL in a manner that also allows for automatic synthesis. A wide range of readers, from hobbyists and undergraduate students to seasoned professionals, will find this a compelling and approachable work. This book provides concise coverage of the material and includes many examples, enabling readers to quickly generate high-quality synthesizable Verilog models.

KEYWORDS

Verilog, Digital System Design, Digital Logic Synthesis, HDL (Hardware Description Language), Combinational Logic, Sequential Logic.

Contents

1. Digital Logic Review with Verilog Quickstart 1

1.1 Learning Objectives 1

1.2 Logic Synthesis Introduction and Motivation 1

1.3 Combinational Logic in Verilog 4

1.3.1 *Assign* Statements 4

1.3.2 *Always* Procedural Blocks 8

1.4 Combinational Building Blocks in Verilog 11

1.4.1 Multibit/Multiinput Muxes, Verilog Hierarchical Design, and Bus Notation 11

1.4.2 Addition, Subtraction 14

1.4.3 Multiplication, Division 17

1.4.4 Shifting 18

1.4.5 Tri-State Logic 20

1.5 Sequential Logic in Verilog 21

1.5.1 One-bit Storage Elements 22

1.5.2 DFF Chains 23

1.5.3 Asynchronous Versus Synchronous Inputs 24

1.5.4 Registers, Counters, and Shift Registers 26

1.6 Event-Driven Simulation and Verilog 29

1.6.1 Event-Driven Simulation Basics 29

1.6.2 Timing Considerations 33

1.6.3 Presynthesis Versus Postsynthesis Simulation 34

1.6.4 Blocking Versus Nonblocking Assignments and Synthesis 35

1.7 Verilog Coding Guidelines 36

1.8 Summary 37

2. Synchronous Sequential Circuit Design 39

2.1 Learning Objectives 39

2.2 Sequential Circuits 39

2.2.1 Sequential Circuit Motivation 40

2.2.2 Synchronizing Signals: The Clock 41

2.2.3 Synchronous Sequential Circuit Architectures 42

2.3 Models of Finite State Machines 44
2.3.1 Basics of Algorithmic State Machine (ASM) Charts 45
2.3.2 The ASM Chart Model and an Example Controller 48
2.3.3 The State Diagram Model 52
2.4 State Assignment 54
2.5 Low-Level Models of Controllers 56
2.5.1 State Equations 56
2.5.2 State Tables 59
2.5.3 Controller Circuit Analysis 61
2.6 Mealy and Moore Machine Conversion 61
2.6.1 Mealy to Moore Machine Conversion 62
2.6.2 Moore to Mealy Conversion 63
2.6.3 State Machine Equivalence 63
2.7 Verilog Descriptions of Synchronous Sequential Circuits 64
2.7.1 Example Verilog Descriptions 66
2.7.2 Verilog Descriptions for the Mealy Machine Model of an Example Controller 66
2.7.3 Verilog Descriptions for the Moore Machine Model of an Example Controller 70
2.8 Summary 72

Biography 75

Preface

Modern digital logic design flows heavily utilize Hardware Description Languages (HDLs) for specification, simulation, automatic synthesis, and validation of target digital systems. The use of HDLs is an important skill for all digital designers to have and this book serves as a concise introduction to one of the most popular HDLs in common use today, Verilog. We introduce the Verilog language through examples as we review basic building blocks of combinational and sequential digital system design. The focus will be to restrict the discussion to synthesizable Verilog and to mention common pitfalls and how to avoid them.

The book is divided into two main chapters. The first chapter is devoted to a quick review of digital logic building blocks and their corresponding Verilog descriptions. Verilog descriptions are shown that correctly describe various logic elements as are those that show common errors and the unintentional digital logic that is produced by a synthesis tool.

A discussion of the basic internal operation of a discrete event simulator is included to help the reader gain an understanding of how the Verilog HDL is simulated. By understanding how a simulator operates, insight is gained into how to write and debug Verilog HDL.

The second chapter focuses on synchronous sequential circuits and their use as controllers for a datapath. Algorithmic State Machine (ASM) charts are described and emphasized throughout the chapter since they are easily understandable by designers and can be directly translated into a Verilog HDL module. An example design of a memory-zeroing circuit is used to illustrate the topics of controller modeling and synthesis.

CHAPTER 1

Digital Logic Review with Verilog Quickstart

This chapter assumes that the reader is already familiar with binary number representation and digital logic principles, and provides a refresher course on these topics while introducing the reader to the Verilog hardware description language (HDL), logic synthesis, and event-driven simulation.

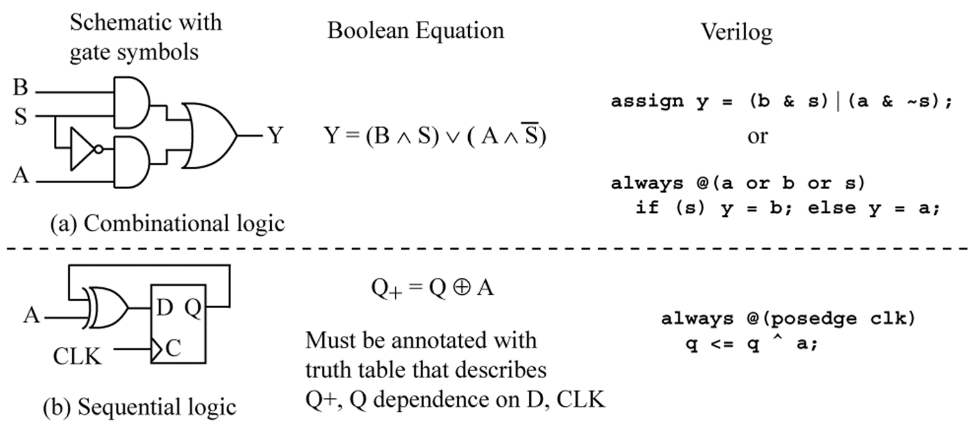
1.1 LEARNING OBJECTIVES

After reading this chapter, you will be able to perform the following tasks:

- Implement combinational gate networks and commonly used combinational building blocks in Verilog.
- Express sequential storage elements and commonly used sequential building blocks in Verilog.
- Discuss the role of logic synthesis in digital design.
- Discuss the principles behind event-driven simulation.

1.2 LOGIC SYNTHESIS INTRODUCTION AND MOTIVATION

Recall that a digital system performs logical and/or arithmetic computations on binary-valued (0, 1) inputs using a mixture of combinational gates and sequential storage elements and produces binary-valued outputs. Fig. 1.1 shows two representations for the combinational and sequential elements of a digital system: (a) a schematic with gate symbols (graphical), and (b) a Boolean equation (text). While a schematic offers a visual representation of the system's operation that can be intuitively easier to understand than a symbolic model, it becomes unwieldy and confusing once the schematic's symbol count exceeds a few 10s of symbols. Design entry and data sharing for schematics is also problematic. Schematic entry tools supplied by different vendors have different graphical user interfaces (GUIs), which must be relearned if a user switches design entry tools. Furthermore, there is no standard data format for schematics, and thus a schematic


Schematic with gate symbols
Boolean Equation
Verilog
B
S
A
Y
(a) Combinational logic
A
D Q
C
CLK
(b) Sequential logic
Must be annotated with truth table that describes Q+, Q dependence on D, CLK

$Y = (B \wedge S) \vee (A \wedge \overline{S})$

```
assign y = (b & s)|(a & ~s);
          or
always @(a or b or s)
  if (s) y = b; else y = a;
```

$Q_+ = Q \oplus A$

```
always @(posedge clk)
  q <= q ^ a;
```

FIGURE 1.1: Combinational and sequential logic representations

file created by one schematic entry tool is not easily portable to a schematic entry tool produced by a different vendor. This hinders design sharing between users who select different tools for design entry.

A text format solves the data format portability problem, as all text editor programs can process files that use the American Standard Code for Information Interchange (ASCII) for character encoding. However, the syntax used to express a digital system's operation must be agreed upon by all parties for the design to be portable between different design tools and users. There are several shortcomings in using Boolean equations for expressing complex digital systems:

- Boolean equations are a low-level description of a system's operation; it can be difficult for an external reader to grasp a system's functionality when it is specified as a sequence of Boolean equations.
- There is no standard method for representing sequential behavior using Boolean equations, as many different notations exist.
- There is no standard method for representing Boolean operations on groups of signals (busses), which is needed for reducing the number of Boolean statements that describe a system.
- Arithmetic operations such as addition, subtraction, multiplication, etc. have to be expressed as their component Boolean equations, increasing the number of Boolean statements for describing a system.

In the late 1970s and throughout the 1980s, as computer tools became essential for designing complex digital systems, many *hardware description languages* (HDLs) were created by different companies and universities for describing the structure and operation of digital systems. Two of these HDLs, Verilog and VHDL (a nested acronym that stands for VHISC HDL, where VHSIC is very high speed integrated circuit), eventually emerged as the standard HDLs for digital system representation. These two HDLs are adept at specifying digital system operation, with Verilog being the more popular language within the United States. This book uses Verilog as its HDL of choice. Verilog was initially a proprietary language, but was transitioned to the public domain by Cadence Design Systems and was made an IEEE standard in 1995, with a revised standard released in 2001 [1]. VHDL is an IEEE standard as well.

In Fig. 1.1(a), the Verilog representation of the combinational gate network is shown in two forms. One form uses a one-to-one mapping of the Boolean operations to Verilog logical operators (& is AND, | is OR, ~ is NOT), which are the same logical operators used in the C and C++ programming languages, and demonstrates that Boolean equations are easily represented in Verilog. The second form that contains the `always` block is an alternate view of the logic network's operation, as it uses an `if` statement that specifies the output `y` as a choice between the inputs of `a`, `b`. It is easier for an external reader to grasp the behavior of the logic network when it is expressed in this form, as it is a familiar representation for any person who has programmed in a high level language (HLL). Digital systems expressed in Verilog typically use these behavioral-type statements instead of Boolean equations, because of the increased clarity for external readers. The sequential network of Fig. 1.1(b) is also represented in Verilog by using an `always` block, using the syntax rules defined in the Verilog standard for expressing sequential behavior. Verilog syntax details are discussed on an *as needed* basis in the following review sections on combinational and sequential logic. Only the Verilog subset required by the design examples in this book is covered as the language contains many features that are outside the scope of these design examples.

There are many ways to implement digital logic; the most common methods in use today are field programmable gate arrays (FPGAs) or standard cells within an application specific integrated circuit (ASIC). In brief, an FPGA contains programmable gates and routing that can be configured to implement a digital system of the user's choice. A *logic synthesis tool* is used to convert a user's digital system specified in an HDL to an FPGA implementation. The logic synthesis tool performs optimizations to meet user-specified constraints concerning circuit speed and gate count. Fig. 1.2 shows that the role of a logic synthesis tool is similar to that of a compiler, whose function is to map a program specified in a high level language to an implementation on a particular central processing unit (CPU).

With respect to Fig. 1.2 and the syntactical similarities between Verilog and common high level languages, it is inaccurate to think of an HDL as a programming language. There are fundamental differences between HDLs and high level programming languages that are discussed in more detail in Sections 1.3 and 1.6.

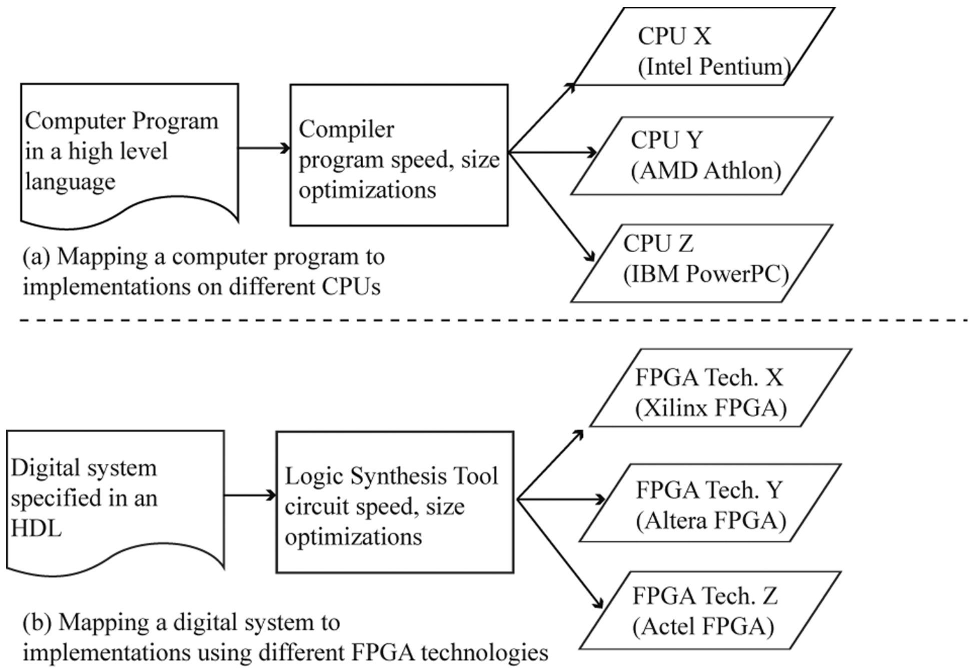

Computer Program in a high level language
Compiler program speed, size optimizations
CPU X (Intel Pentium)
CPU Y (AMD Athlon)
CPU Z (IBM PowerPC)
(a) Mapping a computer program to implementations on different CPUs
Digital system specified in an HDL
Logic Synthesis Tool circuit speed, size optimizations
FPGA Tech. X (Xilinx FPGA)
FPGA Tech. Y (Altera FPGA)
FPGA Tech. Z (Actel FPGA)
(b) Mapping a digital system to implementations using different FPGA technologies

FIGURE 1.2: HDLs and logic synthesis

1.3 COMBINATIONAL LOGIC IN VERILOG

Fig. 1.3 shows the common Boolean functions in their truth table, gate symbol, Boolean equation, and Verilog representations. The Verilog Boolean operators are the same as the bitwise logic operators used in the C and C++ programming languages. Verilog also possesses some gate-level primitives that implement the standard Boolean functions, but this book's examples use the Boolean operators exclusively.

1.3.1 *Assign* Statements

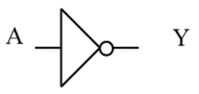
Fig. 1.4 shows the gate logic and symbol for a 1-bit 2-to-1 multiplexer (mux) function; recall that a 2-to-1 multiplexer passes one of the two inputs to the output based upon a select input. In Fig. 1.4, the output Y is equal to the input B if the select input S is "1," else the output Y is equal to A.

Three different Verilog implementations of the 2-to-1 mux are used as an introduction to Verilog combinational logic. Verilog is case sensitive with all keywords in lowercase; keywords are italicized for emphasis in Fig. 1.5. The basic design unit in Verilog is the *module*, which contains the module's interface signals and statements that describe its behavior. The `module` statement contains a list of the module interface signals, also known as *ports*, which can be given in any order. The three variations of Fig. 1.5 only differ in how the module's behavior is

Truth Table | Gate Symbol | Boolean | Verilog

NOT

A	Y
0	1
1	0

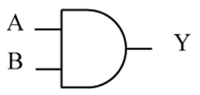


$Y = \overline{A}$

```
assign y = ~a;
```

AND

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

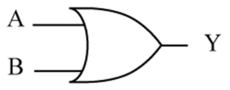


$Y = A \wedge B$

```
assign y = a & b;
```

OR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

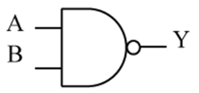


$Y = A \vee B$

```
assign y = a | b;
```

NAND

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

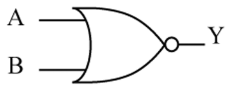


$Y = \overline{A \wedge B}$

```
assign y = ~(a & b);
```

NOR

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

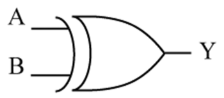


$Y = \overline{A \vee B}$

```
assign y = ~(a | b);
```

XOR

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0



$Y = A \oplus B$

```
assign y = a ^ b;
```

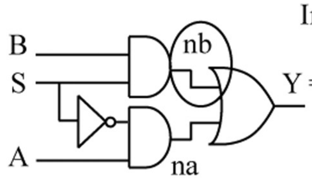
FIGURE 1.3: Boolean logic functions

Truth Table

S	B	A	Y
0	x	0	0
0	x	1	1
1	0	x	0
1	1	x	1

x - don't care

Gate Schematic



Internal net

$Y = (B \wedge S) \vee (A \wedge \overline{S})$

Symbol

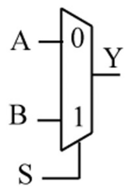


FIGURE 1.4: One-bit 2-to-1 multiplexer

(a) Single `assign` statement using boolean operators

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

//this is a comment
assign y=(b & s)|(a & ~s);



endmodule
```

(b) Multiple `assign` statements using Boolean operations and intermediate values

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

wire na, nb;

//this is a comment
assign nb = b & s;
assign na = a & ~s;
assign y = na | nb;

endmodule
```

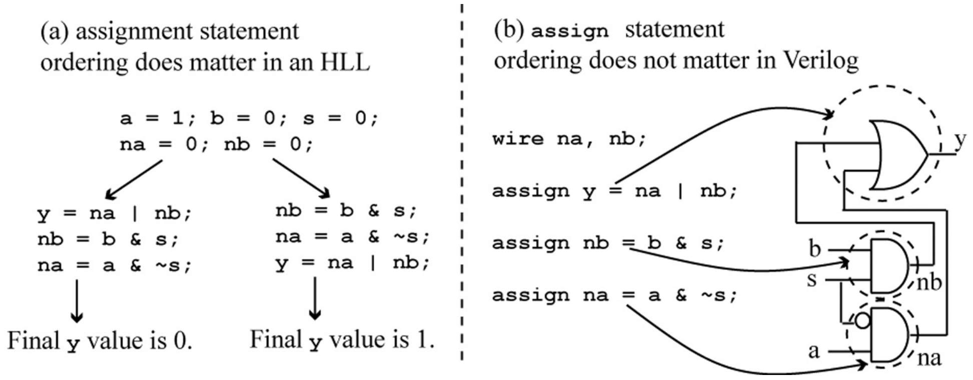
(c) Single `assign` statement using a conditional operator

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

//conditional statement
//              s=1 s=0
assign y = s ? b : a ;



endmodule
```

FIGURE 1.5: One-bit 2-to-1 multiplexer implementations using `assign` statements

written, so the interface of the three modules remains the same. These examples have the input signals first, followed by the output signal, but the order is arbitrary. The `input` and `output` statements that follow the module declaration define the direction of each interface signal. A port is declared bidirectional by using the `inout` keyword. These examples list all of the input ports using a single `input` statement; individual `input` statements could have also been used. Our examples use lowercase for user-defined names such as module and port names; uppercase is commonly used as well. It is suggested that mixed case not be used for user-defined names, as the tools that operate on the design data after they have been transformed from Verilog to some intermediate format may have difficulty with mixed-case names.

An `assign` statement makes a continuous assignment of the right-side expression to the net or port that appears on the left side, using the "=" operator. The best way to think of a continuous assignment is that the right-side expression defines a block of combinational logic, whose output is continuously connected, and thus continuously drives, the net or the port on the left side. Fig. 1.5(a) uses a single `assign` statement to specify a 2-to-1 multiplexer using Boolean operators. Fig. 1.5(b) divides the Boolean operators across multiple `assign` statements, using internal *nets* (gate ports joined by a wire) `na` and `nb` in the last `assign` statement to implement the final *or* gate. The `wire` statement is used to declare nets `na` and `nb`; this is not strictly necessary as only multibit nets (busses) have to be explicitly declared. Fig. 1.5(c) uses the *conditional* operator "?" in a single assignment to specify the multiplexer logic. The first operand in the conditional operator is the select operand (i.e., `s`); if this operand evaluates as nonzero then the second operand (i.e., `a`) is evaluated and returned, else the third operand is evaluated and returned. A module is terminated using the `endmodule` keyword. All of these implementations result in the same logic; none of them have any inherent advantages over the others.



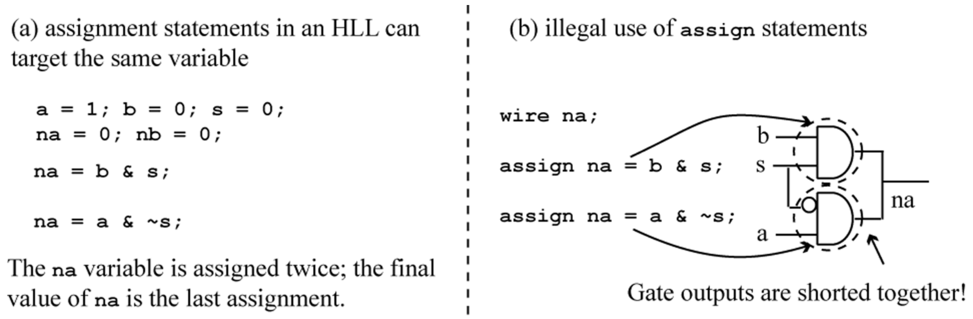
```
a = 1; b = 0; s = 0;
na = 0; nb = 0;

y = na | nb;
nb = b & s;
na = a & ~s;

nb = b & s;
na = a & ~s;
y = na | nb;
```

```
wire na, nb;
assign y = na | nb;
assign nb = b & s;
assign na = a & ~s;
```

FIGURE 1.6: Assignment statement ordering

It was previously stated that it is inaccurate to think of an HDL as just another form of high level programming language where wires are variables and a sequence of HDL statements are semantically equivalent to a sequence of HLL statements. Fig. 1.6 shows assignment statements in a HLL versus Verilog. In Fig. 1.6(a), the ordering of the assignment statements affects the final y value as the statements are evaluated sequentially. In Fig. 1.6(b), it is seen that Verilog `assign` statements describe hardware (gates) that operate concurrently, so the statement ordering does not affect the final hardware that is generated. Another way of stating this is that the arrangement of the gate symbols in a schematic does not affect the circuit function as long as the connections between the gate symbols are the same.

Fig. 1.7 shows another difference between an HLL and the Verilog HDL. In an HLL, multiple assignments to the same variable result in the variable's value being the result of the last assignment. In Verilog, multiple `assign` statements to the same wire causes the gate outputs of the logic implementation to be connected together. The only time that this is allowed is for tri-state logic implementation, in which only one of the tri-state gates driving the wire is


(a) assignment statements in an HLL can
target the same variable
The na variable is assigned twice; the final
value of na is the last assignment.
(b) illegal use of assign statements
b
s
na
a
Gate outputs are shorted together!

```
a = 1; b = 0; s = 0;
na = 0; nb = 0;
na = b & s;

na = a & ~s;
```

```
wire na;
assign na = b & s;
assign na = a & ~s;
```

FIGURE 1.7: Continuous assignments to the same wire

asserted at a given time. The Verilog `assign` statements in Fig. 1.7(b) are illegal as this does not implement tri-state drivers; correct implementation of tri-state logic is discussed in Section 1.4.

1.3.2 *Always* Procedural Blocks

While combinational logic can be described using `assign` statements, higher complexity combinational logic blocks are better described using `always` procedural blocks for several reasons:

- Powerful statements like `if`, `if-else`, `case` and looping constructs can only be used in `always` blocks; these statements are useful for implementing complex combinational blocks with greater clarity and in a more concise manner than is possible with `assign` statements.
- Multiple output nets can be assigned within a single `always` block.
- Sequential logic can only be specified within `always` blocks.

Fig. 1.8 serves as an introduction to `always` blocks by showing the implementation of the 2-to-1 multiplexer of Fig. 1.5 using three variations that each use an `always` block. For combinational logic, an `always` block header contains an *event list* that is designated by "@(*net1* or *net2* or . . . *netN*)" where changes (events) on nets 1, 2, . . . ,N cause the logic represented by the `always` block to evaluate their inputs. For combinational logic, any net that appears on the right side of an "=" operator in the `always` block should be included in the event list. The body

(a) Combinational `always` block using an `if` statement.

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y;

//use an if statement
always @(a or b or s)
begin
 if (s) y = b;
else y = a;
end

endmodule
```

(b) Combinational `always` block using Boolean operations.

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y;

//use boolean ops
always @(a or b or s)
begin
 y = (b & s)|(a & ~s);
end

endmodule
```

(c) Combinational `always` block using Boolean operations and intermediate values.

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y, na, nb;

//use intermediates
//and implicit event
//list
always @*
begin
 nb = b & s;
 na = a & ~s;
 y = na | nb;
end

endmodule
```

FIGURE 1.8: 1-bit 2-to-1 multiplexer using `always` procedural blocks

of an always block can be one or more statements; begin and end keywords are used to group multiple statements. The statement "reg y;" is included in each of the three modules of Fig. 1.8 as any net that is assigned within an always block must be declared as a reg type; this does not imply that this net is driven by a register or sequential logic.

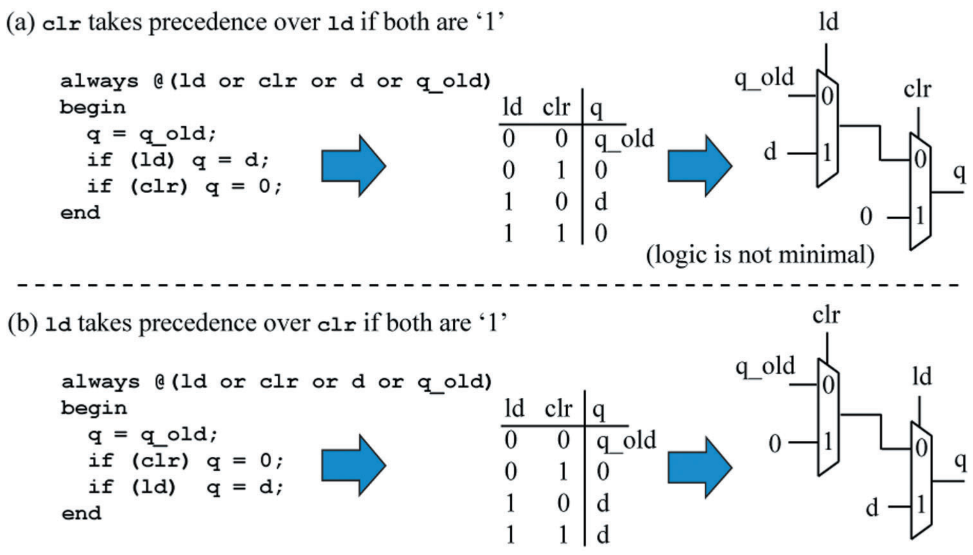
Fig. 1.8(a) implements the multiplexer using an if-else statement, with the if-body evaluated for a nonzero (true) conditional expression and the else clause evaluated otherwise. Keywords begin and end can be used to place multiple statements in an if-body or else clause. Fig. 1.8(b) implements the multiplexer using Boolean operators, while Fig. 1.8(c) distributes the Boolean operators using intermediate nets and multiple assignments. The "=" operator when used in an always block is called a *blocking* assignment, this terminology is discussed in more detail in Section 1.6, which covers event-driven simulation principles. The always block of Fig. 1.8(c) uses an *implicit* event list designated by "@*," which means that all nets on the right side of the assignments are included in the event list.

The semantics of the net assignments in an always blocks differs *significantly* from assign statements in that statements in an always block use the same sequential execution model as the statements in an HLL.

- The logic synthesized for an always block duplicates the assignments' behavior assuming that the assignments are evaluated *sequentially*. This means that the *order in which assignments are written* in an always blocks affects the logic that is synthesized.
- Because of the sequential nature of an always block, the same net can be assigned multiple times in an always block; the last assignment takes precedence.

Fig. 1.9 shows a case in which multiple blocking assignments are made to a net in the same always block, with assignment ordering affecting the synthesized logic. In Fig. 1.9(a), the clr input takes precedence over the ld input if both are "1," while in Fig. 1.9(b) the opposite holds true. Observe that in the two always blocks, if both ld and clr are "0," then the initial assignment of q=q_old sets the value of q. This is a common coding style used in combinational logic always blocks in that a default assignment is made to an output at the block's beginning, which is then overridden by later assignments in the block based upon the assertion of other inputs.

The use of a default assignment to the output net of a combinational always block *guarantees* that the output net is the target of an assignment for any input combination. If there is some logic path through the always block that does not assign a value to the output net, then a latch is inferred on that output as shown in Fig. 1.10. A latch is a sequential logic element, and should not be synthesized in an always block that is meant to implement combinational logic. Inferred latches are a common coding mistake for users who are new to Verilog and logic


(a) `clr` takes precedence over `ld` if both are '1'

```
always @(ld or clr or d or q_old)
begin
  q = q_old;
  if (ld) q = d;
  if (clr) q = 0;
end
```

ld	clr	q
0	0	q_old
0	1	0
1	0	d
1	1	0

(logic is not minimal)

(b) `ld` takes precedence over `clr` if both are '1'

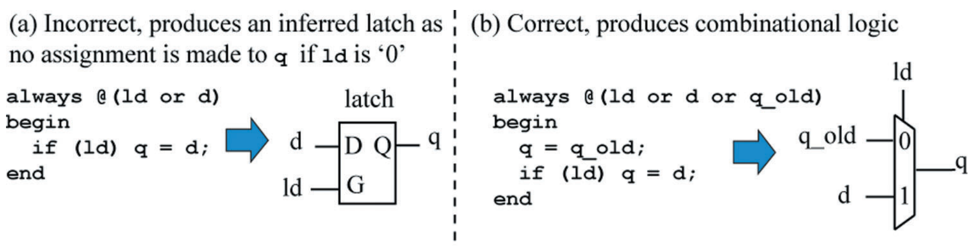
```
always @(ld or clr or d or q_old)
begin
  q = q_old;
  if (clr) q = 0;
  if (ld)  q = d;
end
```

ld	clr	q
0	0	q_old
0	1	0
1	0	d
1	1	d

FIGURE 1.9: The sequential nature of `always` blocks

synthesis in general. Fig. 1.10(a) shows that an inferred latch is placed on the net output `q` because there is no assignment to `q` if `ld` is "0." Fig. 1.10(b) corrects this mistake by assigning a default value to the `q` output net. Fig. 1.10 gives a peek at how to specify sequential logic in Verilog; Section 1.5 covers this in detail.

Another viewpoint of an `always` block's functionality is that it is a complex form of an `assign` statement. A module can have multiple `always` blocks, with each `always` block representing discrete logic blocks, just as a module can have multiple `assign` statements. The ordering of `always` blocks in a module does not affect the logic that is synthesized, for the same reason that `assign` statement ordering does not affect synthesized logic. Except for tri-state logic implementations, the same output net cannot be driven from multiple `always` blocks, just as the same net cannot be driven from multiple `assign` statements, because the gate drivers synthesized for the output net in each `always` block will be connected together.


(a) Incorrect, produces an inferred latch as no assignment is made to `q` if `ld` is '0'

```
always @(ld or d)
begin
  if (ld) q = d;
end
```

(b) Correct, produces combinational logic

```
always @(ld or d or q_old)
begin
  q = q_old;
  if (ld) q = d;
end
```

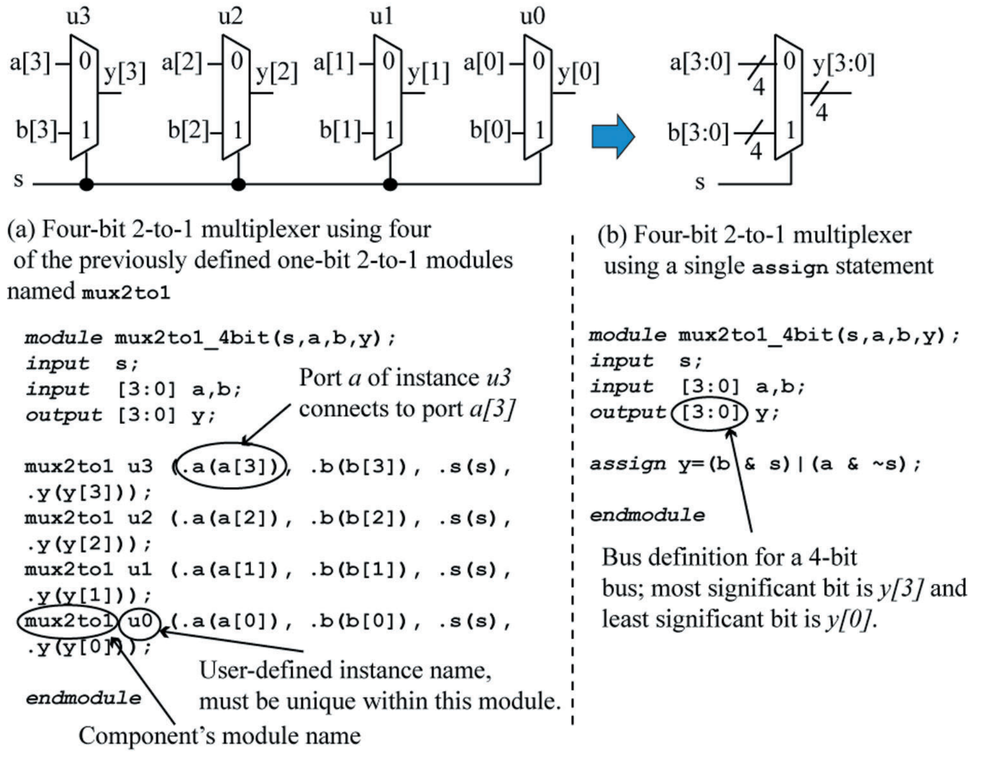
FIGURE 1.10: A common mistake with combinational `always` blocks: inferred latches

1.4 COMBINATIONAL BUILDING BLOCKS IN VERILOG

This section discusses common combinational building blocks and their implementation in Verilog, with new Verilog concepts introduced as they are required.

1.4.1 Multibit/Multiinput Muxes, Verilog Hierarchical Design, and Bus Notation

The 1-bit 2-to-1 mux of Fig. 1.4 can be extended to an N-bit 2-to-1 mux by paralleling N of the 1-bit muxes as shown in Fig. 1.11 for $N = 4$. The inputs a, b and the output y now become 4-bit wide busses, labeled as $a[3:0]$, $b[3:0]$, and $y[3:0]$, respectively. The individual bits of these busses are labeled as $a[0]$, $a[1]$, etc., and connect to the appropriate data ports on each of the 1-bit 2-to-1 muxes. The s control input of the four 1-bit muxes are tied together so that the s input selects the four data bits of the a and b inputs in parallel. Specifying this in Verilog requires introducing the Verilog syntax for busses and hierarchical modules. In Verilog, a port or wire that is greater than 1 bit in width is officially referred to as a *vector*, but is informally referred to as a *bus* in this book. The bus indices can be arbitrary values in Verilog, but our examples assume that a bus carries an N-bit quantity whose most significant bit is $N-1$ and whose least significant


u3
u2
u1
u0
a[3]
a[2]
a[1]
a[0]
b[3]
b[2]
b[1]
b[0]
y[3]
y[2]
y[1]
y[0]
s
0
1
a[3:0]
b[3:0]
y[3:0]
4
Port *a* of instance *u3* connects to port *a[3]*
User-defined instance name, must be unique within this module.
Component's module name
Bus definition for a 4-bit bus; most significant bit is *y[3]* and least significant bit is *y[0]*.

(a) Four-bit 2-to-1 multiplexer using four of the previously defined one-bit 2-to-1 modules named `mux2to1`

```
module mux2to1_4bit(s,a,b,y);
input  s;
input  [3:0] a,b;
output [3:0] y;

mux2to1 u3 (.a(a[3]), .b(b[3]), .s(s),
.y(y[3]));
mux2to1 u2 (.a(a[2]), .b(b[2]), .s(s),
.y(y[2]));
mux2to1 u1 (.a(a[1]), .b(b[1]), .s(s),
.y(y[1]));
mux2to1 u0 (.a(a[0]), .b(b[0]), .s(s),
.y(y[0]));

endmodule
```

(b) Four-bit 2-to-1 multiplexer using a single `assign` statement

```
module mux2to1_4bit(s,a,b,y);
input  s;
input  [3:0] a,b;
output [3:0] y;

assign y=(b & s)|(a & ~s);

endmodule
```

FIGURE 1.11: 4-bit 2-to-1 multiplexer implementations

bit is 0, to form a range $N-1$ to 0. In Verilog, this range is declared as "[$N-1:0$]," so a 4-bit bus has the range "[3:0]." Individual wires in the bus are labeled as *netname*[0], *netname*[1], etc. The 4-bit 2-to-1 mux in Fig. 1.11 thus has Verilog port names for the busses of `a[3:0]`, `b[3:0]`, and `y[3:0]`. Observe that in the `module` interface of Fig. 1.11(a), the `a`, `b` inputs use a separate `input` statement from the `s` input because their widths are different.

The module in Fig. 1.11(a) is a *hierarchical* module, because it uses four *instances* of the previously described `mux2to1` module to implement the 4-bit 2-to-1 mux. Each instance statement contains the module name (i.e., `mux2to1`), a user-defined instance name (i.e., `u3`), and a terminal list that describes how the instance's ports connect to nets or ports within the hierarchical module. This example uses *named association* within the terminal list, using the syntax ".*instance_port*(*netname*)" to connect an *instance_port* to the *netname* in the hierarchical module. A shorthand notation can be used in which the instance port names are not specified, with the hierarchical nets in the terminal list assumed to connect to instance ports in the order in which the instance ports are declared within the instance module. Named association is clearer and less prone to careless error, so all examples in this book used named association for terminal lists in instances.

Fig. 1.11(b) shows the 4-bit 2-to-1 mux implemented with the same `assign` statement used for the 1-bit version in Fig. 1.5(a), by simply changing the module interface to accommodate the 4-bit data ports. This illustrates Verilog's capability of specifying N-bit wide operations as easily as 1-bit wide operations. This is a powerful mechanism that improves a designer's productivity, allowing designers to focus on higher complexity designs.

Fig. 1.12 shows a 4-bit 4-to-1 mux implemented using 4-bit 2-to-1 muxes. While this is one of the traditional implementation methods for higher level muxes and the Verilog

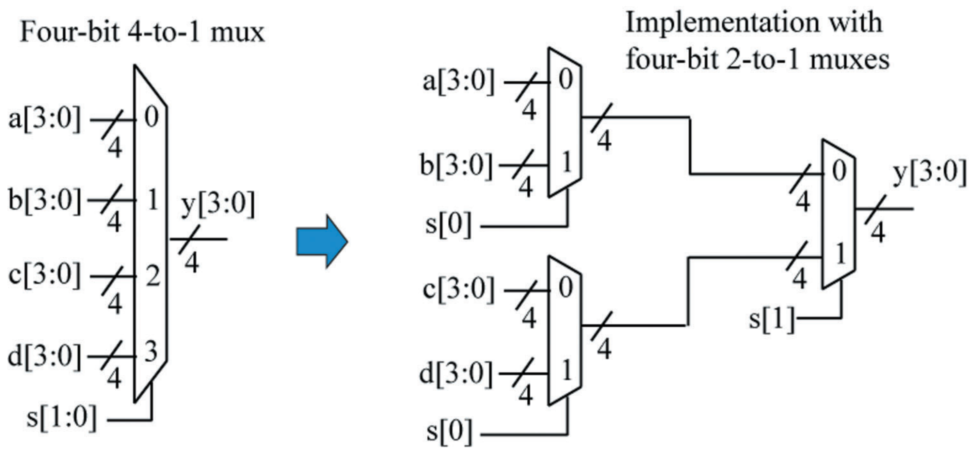
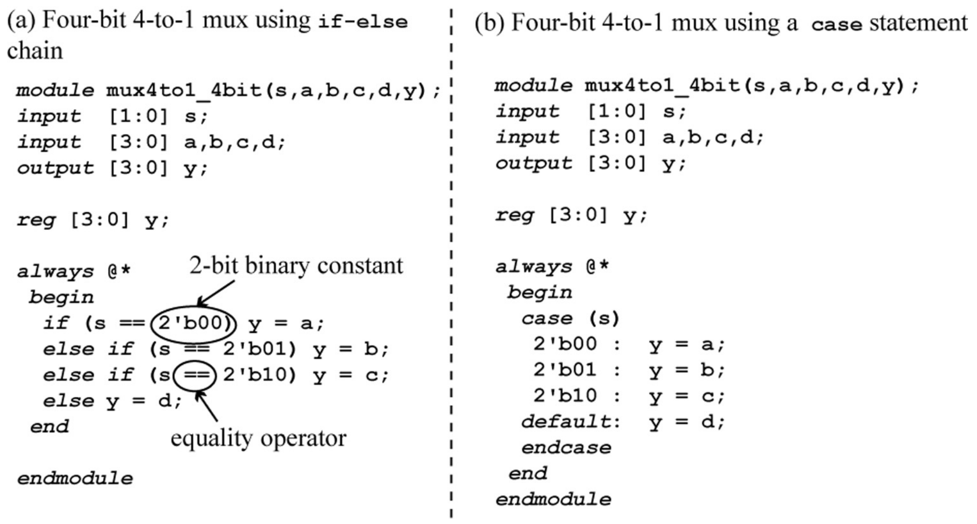


FIGURE 1.12: Four-bit 2-to-1 multiplexer implementations



(a) Four-bit 4-to-1 mux using `if-else` chain

```
module mux4to1_4bit(s,a,b,c,d,y);
input  [1:0] s;
input  [3:0] a,b,c,d;
output [3:0] y;

reg [3:0] y;

always @*
 begin
  if (s == 2'b00) y = a;
  else if (s == 2'b01) y = b;
  else if (s == 2'b10) y = c;
  else y = d;
 end

endmodule
```

(b) Four-bit 4-to-1 mux using a `case` statement

```
module mux4to1_4bit(s,a,b,c,d,y);
input  [1:0] s;
input  [3:0] a,b,c,d;
output [3:0] y;

reg [3:0] y;

always @*
 begin
  case (s)
   2'b00 :  y = a;
   2'b01 :  y = b;
   2'b10 :  y = c;
  default:  y = d;
  endcase
 end
endmodule
```

FIGURE 1.13: Verilog implementations of a 4-bit 2-to-1 multiplexer

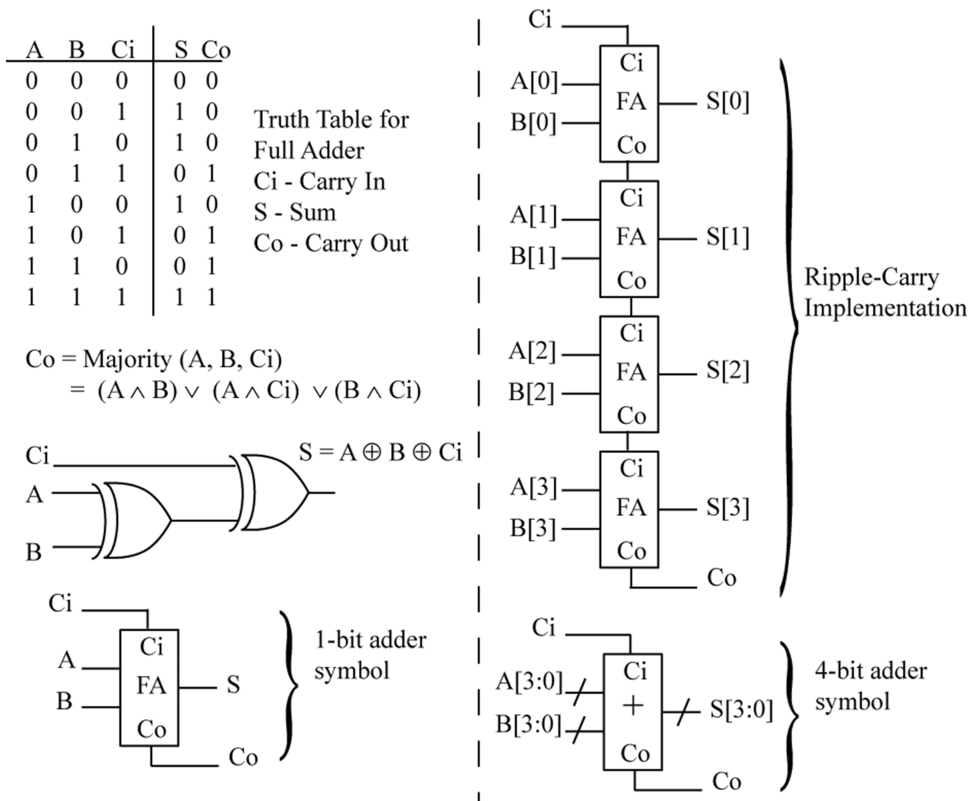
specification could be structured in this manner, it may or may not be an efficient implementation for a particular target technology, such as an FPGA. It is the function of the logic synthesis tool to map a Verilog specification to a target technology using the most efficient gate structure for that technology, so the Verilog descriptions of combinational building blocks that offer clarity and leave out implementation details are usually preferred.

Fig. 1.13 gives two Verilog specifications for this 4-bit 4-to-1 bit mux that only specify the multiplexer's functionality, not the implementation. Both Verilog modules use `always` blocks, but Fig. 1.13(a) uses an *if-else* chain for the logic while Fig. 1.13(b) uses a *case* statement. In Fig. 1.13(a), the equality operator `==` is used to compare the `s` select input against the 2-bit binary constants `2'b00`, `2'b01`, `2b'10` in the *if-else* chain, corresponding to the assignments of `y=a`, `y=b`, and `y=c`. If none of these comparisons are true, then the last *else* clause makes the assignment `y=d`. The `case` statement used in Fig. 1.13(b) is a shorthand notation for an *if-else* chain; there is no advantage to either form in terms of the logic that is synthesized. The `default` clause in the `case` statement is the clause that is chosen if `s` does match any of the previous values given for `s`. Observe that in both `always` blocks, the output `y` is guaranteed to be assigned some value due to the use of the last *else* clause in Fig. 1.13(a) and the *default* clause in Fig. 1.13(b). This is important, as it prevents a latch from being inferred on the `y` output.

This is the first time that a Verilog module has used a constant that was greater than a single bit in width. The general format of an unsigned integer constant is *[size]'baseDDDD..D*, where size is the number of bits in the constant, *base* is a single letter designating the base, and *DDDD..D* are digits in the specified base. Supported bases are d/D (decimal), b/B (binary), o/O (octal), and h/H (hex). The default base is decimal; all other bases must use a base specification. The size specification is optional; it is sized by the logic synthesis tool to match the sizes of other operands in the associated expression. A constant is left padded with zeros if necessary.

1.4.2 Addition, Subtraction

Fig. 1.14(a) shows the truth table, Boolean equations, and logic symbol for a binary full adder. The *A*, *B*, and *Ci* (carry-in) inputs are summed using binary arithmetic to produce the *S* (sum) and *Co* (carry-out) outputs. Fig. 1.14(b) shows a 4-bit adder using a *ripple-carry* adder configuration in which four 1-bit binary full adders are used, with the carry-out of full adder *i* as the carry-in to full adder $i + 1$. While this is the most gate-efficient implementation for an *N*-bit adder, it is also the slowest as the longest delay path passes through the carry chain of all *N* full adders. There are many other adder structures, such as *carry lookahead* and *carry select*, that are better suited for larger values of *N* given a particular maximum delay constraint,

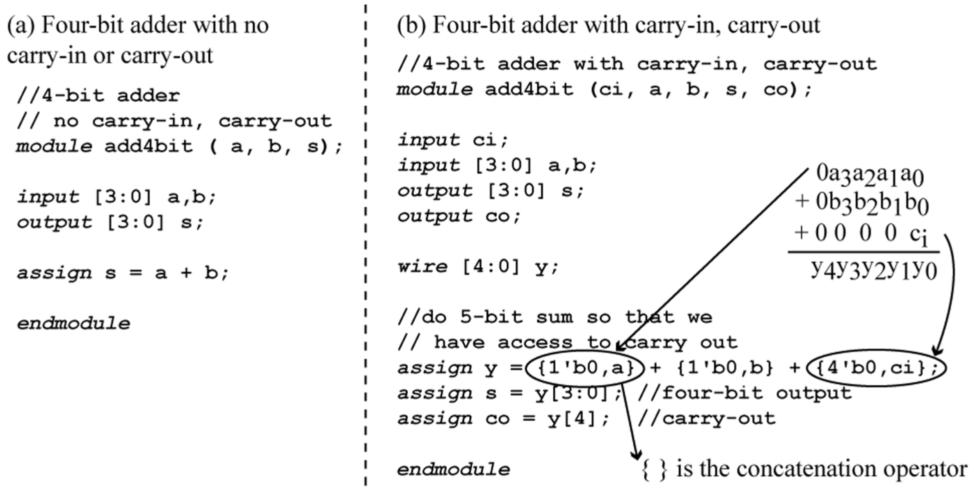


A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) Full Adder Truth table, Boolean equations, Symbol

(b) 4-bit adder symbol and Ripple Carry Implementation

FIGURE 1.14: Binary full adder and a ripple-carry 4-bit adder



(a) Four-bit adder with no carry-in or carry-out

```
//4-bit adder
// no carry-in, carry-out
module add4bit ( a, b, s);

input [3:0] a,b;
output [3:0] s;

assign s = a + b;

endmodule
```

(b) Four-bit adder with carry-in, carry-out

```
//4-bit adder with carry-in, carry-out
module add4bit (ci, a, b, s, co);

input ci;
input [3:0] a,b;
output [3:0] s;
output co;

wire [4:0] y;

//do 5-bit sum so that we
// have access to carry out
assign y = {1'b0,a} + {1'b0,b} + {4'b0,ci};
assign s = y[3:0]; //four-bit output
assign co = y[4]; //carry-out

endmodule
```

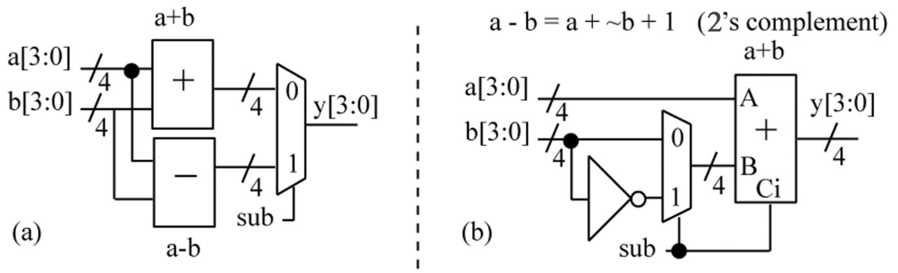
FIGURE 1.15: Verilog 4-bit adder implementations

but the N value at which one adder implementation becomes better than another is dependent on the implementation technology. For example, one FPGA vendor may implement fast carry chains so a ripple-carry structure may have the same delay as a carry lookahead structure for $N < 12$, while for a different FPGA vendor this may be true for $N < 8$. As such, for small to medium sized adders it is generally better to specify a behavioral implementation of the addition using the Verilog "+" operator instead of a structural implementation, and let the logic synthesis tool choose the best adder structure. For large adders, FPGA vendors will typically supply a parameterized library element that offers performance and gate count tradeoffs for their particular architecture. Increased performance comes at increased gate count, which is a well-known tradeoff in digital system design.

The Verilog module of Fig. 1.15(a) implements a 4-bit addition using the continuous assignment `s = a + b`. The use of the "+" operator leaves the logic synthesis tool free to choose the adder structure that will best meet any user-specified constraint, such as delay or area. The Verilog module of Fig. 1.15(b) also implements a 4-bit addition, except that it also provides carry-in and carry-out ports, similar to the adder of Fig. 1.14(b). The addition performed in the first assignment statement "`y =` " is a 5-bit sum with the fifth bit used as the carry-out signal. The concatenation operator {} is used to transform the 4-bit `a`, `b` operands to 5-bit operands by appending a most significant bit of 0. The carry-in bit is concatenated with four leading "0"s by the {`4'b0,ci`} operation and is also part of the sum in the first assignment. The sum output `s` is the lower 4 bits of the 5-bit `y` internal sum.

A *full subtractor* and an N-bit subtractor can be derived in a similar manner to that of the full adder and N-bit ripple-carry adder. As with addition, subtraction is supported within

Even though these two implementations are functionally equivalent, some synthesis tools may produce a slightly more efficient implementation for (b) than (a).



```
//adder/subtractor
module addsub4bit(sub,a,b,y);

input sub;
input [3:0] a,b;
output [3:0] y;
//                 sub=1   sub=0
assign y = sub ?(a-b):(a+b);

endmodule
```

```
//adder/subtractor
module addsub4bit ( sub, a, b, y);

input sub;
input [3:0] a,b;
output [3:0] y;

wire [3:0] btmp;
//                    sub=1  sub=0
assign btmp = sub ? ~b   : b;
assign y = a + btmp + {3'b0,sub};

endmodule
```

FIGURE 1.16: Adder/subtractor implementations

Verilog by the subtraction operator "−." Fig. 1.16(a) shows an adder/subtractor implementation that uses a multiplexer to select between addition "+" and subtraction "−" operations based on the `sub` input. While this may be the most intuitive way to construct an adder/subtractor, it is not necessarily the most efficient as it implies separate subtraction and addition blocks. The traditional implementation of an adder/subtractor is shown in Fig. 1.16(b), which uses the fact that `a-b` is equal to `a+~b+1`, that is, add the 2's complement of `b` to `a` to perform the subtraction. It is obvious that Fig. 1.16(b) can be implemented in fewer gates than that of Fig. 1.16(a) if both structures are mapped in a straightforward manner to gate primitives. However, because the two Verilog modules are functionally equivalent, when the Verilog operators are mapped to Boolean equations and logic optimization is done, one might expect that the same implementation is created for both. However, it is dependent on the particular logic synthesis tool that is used as to whether one of the two Verilog modules in Fig. 1.16 is implemented in a more efficient manner than the other. This example is given to illustrate that logic synthesis is not a magic panacea that removes all responsibility from the designer for creating efficient implementations. The manner in which the Verilog RTL (*Register Transfer Level*) is written can affect the efficiency of the resulting implementations, and a good designer is always aware of this.

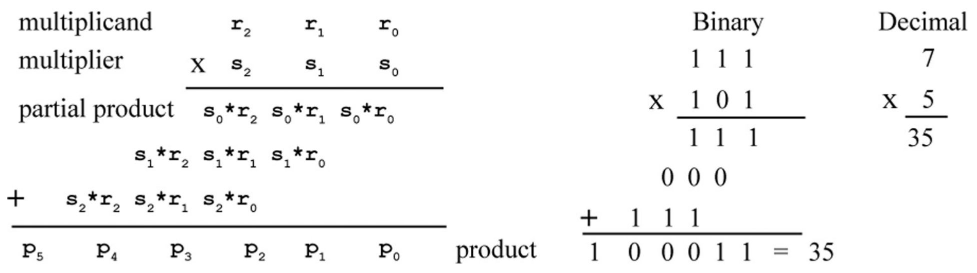
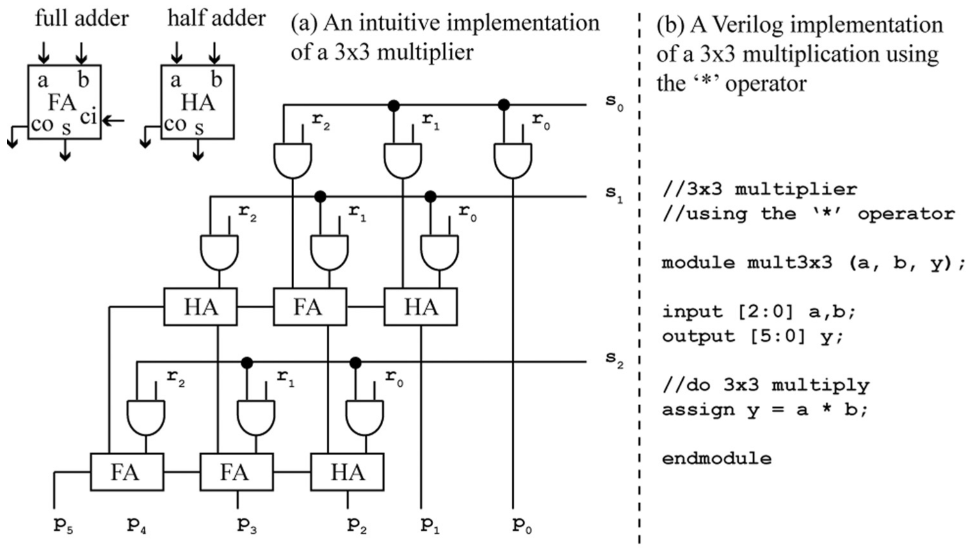


FIGURE 1.17: The multiplication operation for unsigned numbers

1.4.3 Multiplication, Division

Fig. 1.17 shows an unsigned 3 × 3 multiplication. A k-bit multiplicand and an m-bit multiplier produce a $(k + m)$-bit product. Typically, the operands in a multiplication are of the same length, so two n-bit operands form a $2n$-bit product. Depending on the implementation, some of the product bits may have to be discarded; choosing which bits to discard depends on the fixed-point representation chosen for the two operands.

As seen in Fig. 1.17, a binary multiplication operation is an AND operation. Each bit of the multiplier is AND'ed with the multiplicand to form *partial products*, which are summed to form the product. An intuitive implementation of a multiplier that directly maps from the operations of Fig. 1.17 is shown in Fig. 1.18(a). This is a combinational *array multiplier* and while it is intuitive and functionally correct, there are other array multiplier structures that offer higher performance as the operand size increases. Fig. 1.18(b) shows the 3 × 3


full adder
half adder
a b FA ci co s
a b HA co s
(a) An intuitive implementation of a 3x3 multiplier
s0
s1
s2
r2 r1 r0
HA FA HA
FA FA HA
p5 p4 p3 p2 p1 p0
(b) A Verilog implementation of a 3x3 multiplication using the '*' operator

```
//3x3 multiplier
//using the '*' operator

module mult3x3 (a, b, y);

input [2:0] a,b;
output [5:0] y;

//do 3x3 multiply
assign y = a * b;

endmodule
```

FIGURE 1.18: (a) An intuitive 3 × 3 array multiplier and (b) a 3 × 3 multiplier in Verilog

multiplication implemented in a Verilog module using the "∗" operator. For the same reasons as the "+" operation, it is usually advantageous to use the "∗" operator for small to medium width multipliers and let the logic synthesis tool choose the best multiplier structure. For multipliers with large width operands, most logic synthesis tools have a library of multiplier implementations that offer performance versus gate count tradeoffs. It must also be noted that the multiplier implementation also depends on whether the operands are unsigned or signed (2's complement); the multiplier synthesized by the Verilog "∗" operator is an unsigned multiplier.

The division operation is only mentioned briefly here as it is similar to multiplication in terms of logic synthesis. Verilog has a division operator ("/") that will synthesize to an unsigned combinational division implementation. However, as with multiplication, most logic synthesis tools offer a parameterized library module for division that provides choices for unsigned versus signed operands, the sizes of the dividend, remainder and quotient, and performance versus gate count tradeoffs.

1.4.4 Shifting

Right and left shift operations by a single bit position on 8-bit values are shown in Figs. 1.19(a) and 1.19(b), respectively. The s_i value is the bit shifted into the vacant bit position, which is the most significant bit for a right shift, and the least significant bit for a left shift. A 1-bit shift implementation is simply wiring, as shown by the Verilog concatenation statements that accomplish the shift. The right shift operation copies the upper 7 bits of `a` to the lower 7 bits of `y`, with the most significant bit of `y` filled by s_i. The left shift operation copies the lower 7 bits of `a` to the upper 7 bits of `y`, with the least significant bit filled by s_i.

Fig. 1.20(a) shows a multiplexer implementation of an 8-bit shift block that can shift left by 1 bit, right by 1 bit, or pass its input through unchanged. The Verilog implementation of this shift block is shown in Fig. 1.20(b). Verilog also has left (<<) and right shift (>>) operators that can be used for shifting; both of these operators use a zero for the shift input bit. The Verilog module in Fig. 1.20(b) allows the user to control the value of the shift input bit (s_i).

The shifter implemented in Fig. 1.20 only shifts by one position; a shifter that can shift multiple positions is called a *barrel shifter*. Fig. 1.21(a) shows the traditional multiplexer implementation of a 32-bit barrel shifter that shifts left by 0 to 31 positions based on the 5-bit *s*

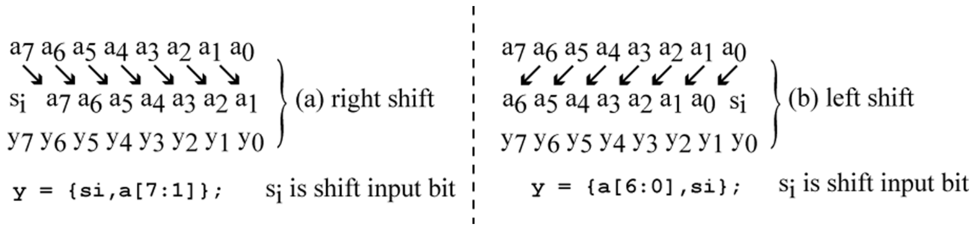
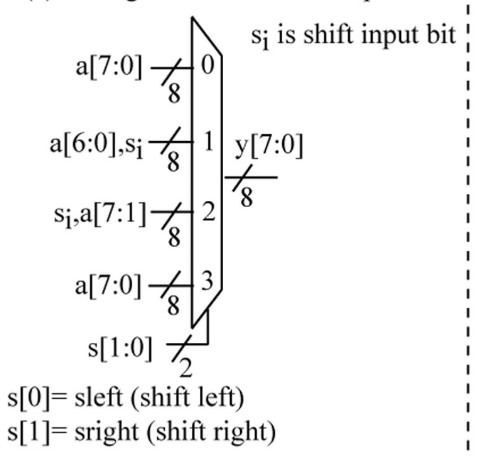


FIGURE 1.19: Right/left shift operations

(a) left/right shift with a multiplexer



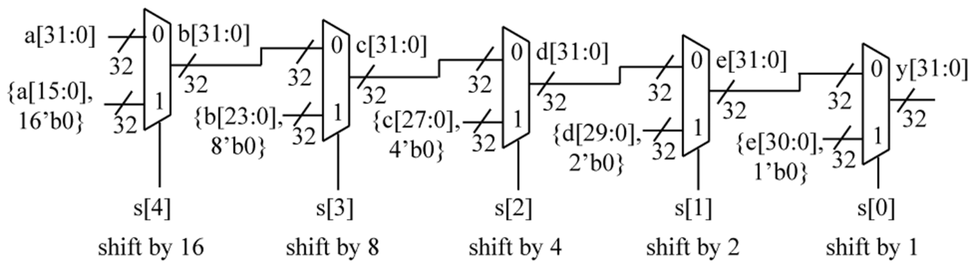
s	*y*
00	*y* = *a*
01	*y* is *a* shifted to left
10	*y* is *a* shifted to right
11	*y* = *a*

(b) Verilog left/right shift

```
module lrshift_8bit (si,sleft,sright,a,y);

input si,sleft,sright;
input [7:0] a;
output [7:0] y;

reg [7:0] y;
wire [1:0] s;

assign s[0] = sleft;
assign s[1] = sright;

always @(s or a) begin
 case (s)
  2'b01   : y = {a[6:0],si}; //left shift
  2'b10   : y = {si,a[7:1]}; //right shift
  default: y = a;
 endcase
end

endmodule
```

FIGURE 1.20: Verilog left/right shift by one example

(a) Multiplexer implementation of a 32-bit left-shift barrel shifter



(b) Verilog implementation

```
module bshift_32bit (s, a, y);
input [4:0] s;
input [31:0] a;
output [31:0] y;

assign y = a << s;

endmodule
```

FIGURE 1.21: Barrel shifter (32-bit, left shift)

input. Each stage of the barrel shifter corresponds to a fixed shift by a power of 2 (16, 8, 4, 2, 1) controlled by a bit in *s*. The composition of the five stages produces a shift amount between 0 and 31. The Verilog implementation of this shifter is implemented by a single continuous assignment statement of `y = a << s`. This again illustrates the power of an HDL to hide implementation details from a designer and to specify complex logic blocks in a very compact manner.

1.4.5 Tri-State Logic

Fig. 1.22(a) shows the operation of a *tri-state* buffer with a high true enable. A tri-state buffer can be thought of as a noninverting buffer with a switch on its output. When the buffer is enabled, the output is driven by the noninverting buffer. When the buffer is disabled, the output is disconnected from the noninverting buffer, and the output is left floating or in the *high-impedance* logic state. The high-impedance logic state is traditionally indicated by the "*Z*" value. Two common uses of tri-state logic are shown in Figs. 1.22(b) and Figs. 1.22(c). The half-duplex link shown in Fig. 1.22(b) between blocks 1 and 2 allows communication in either

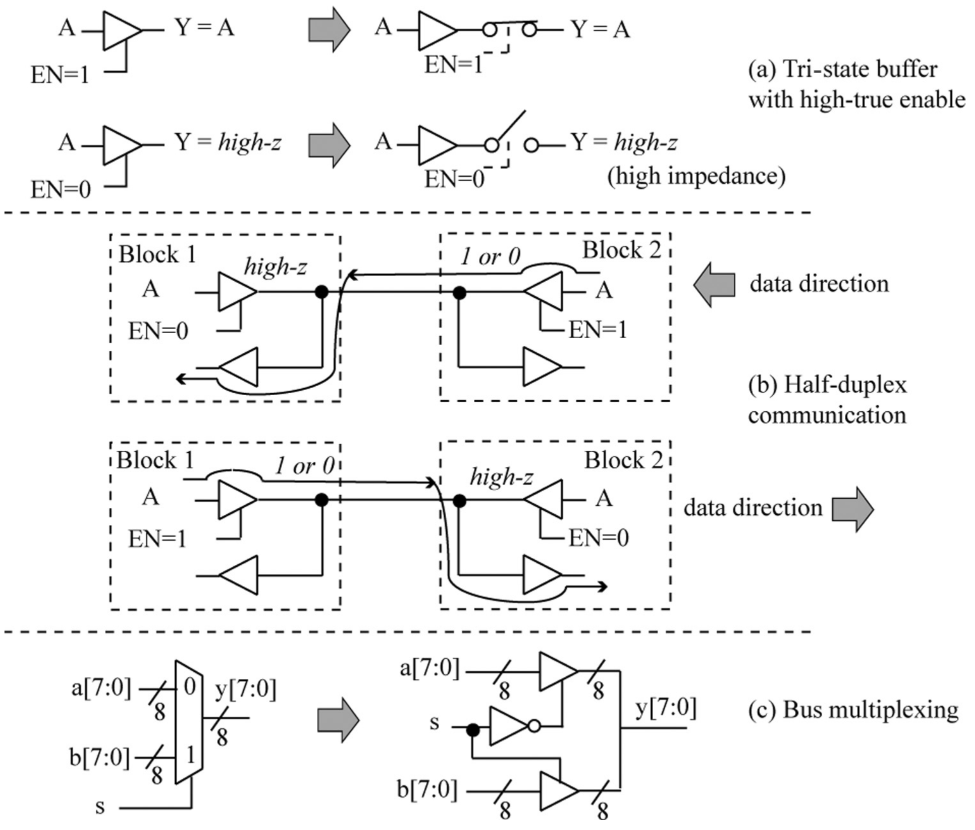
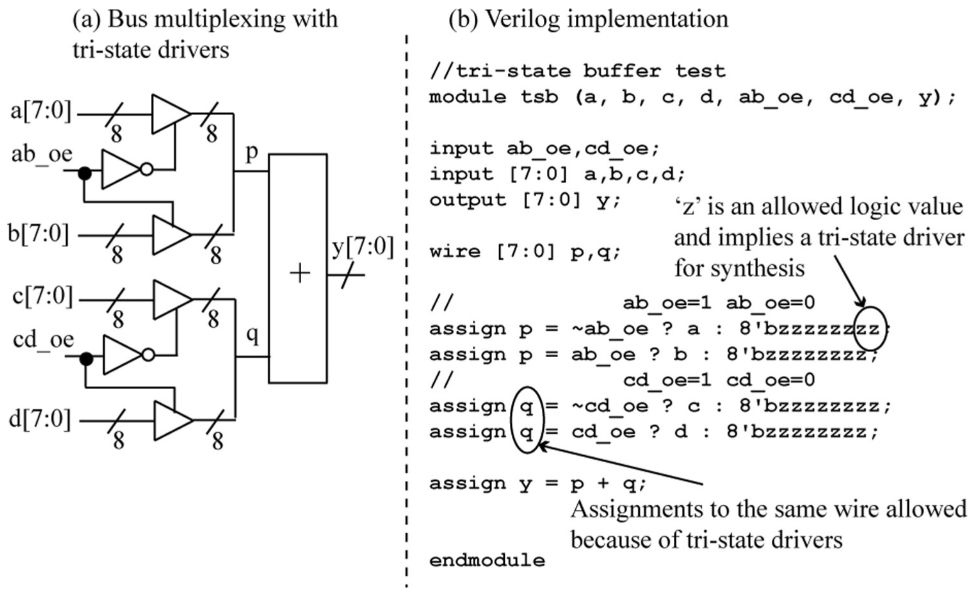


FIGURE 1.22: Tri-state logic and sample uses


(a) Bus multiplexing with tri-state drivers
a[7:0]
ab_oe
b[7:0]
c[7:0]
cd_oe
d[7:0]
8
p
q
+
y[7:0]
(b) Verilog implementation
'z' is an allowed logic value and implies a tri-state driver for synthesis
Assignments to the same wire allowed because of tri-state drivers

```
//tri-state buffer test
module tsb (a, b, c, d, ab_oe, cd_oe, y);

input ab_oe,cd_oe;
input [7:0] a,b,c,d;
output [7:0] y;

wire [7:0] p,q;

//              ab_oe=1 ab_oe=0
assign p = ~ab_oe ? a : 8'bzzzzzzzz;
assign p = ab_oe ? b : 8'bzzzzzzzz;
//              cd_oe=1 cd_oe=0
assign q = ~cd_oe ? c : 8'bzzzzzzzz;
assign q = cd_oe ? d : 8'bzzzzzzzz;

assign y = p + q;

endmodule
```

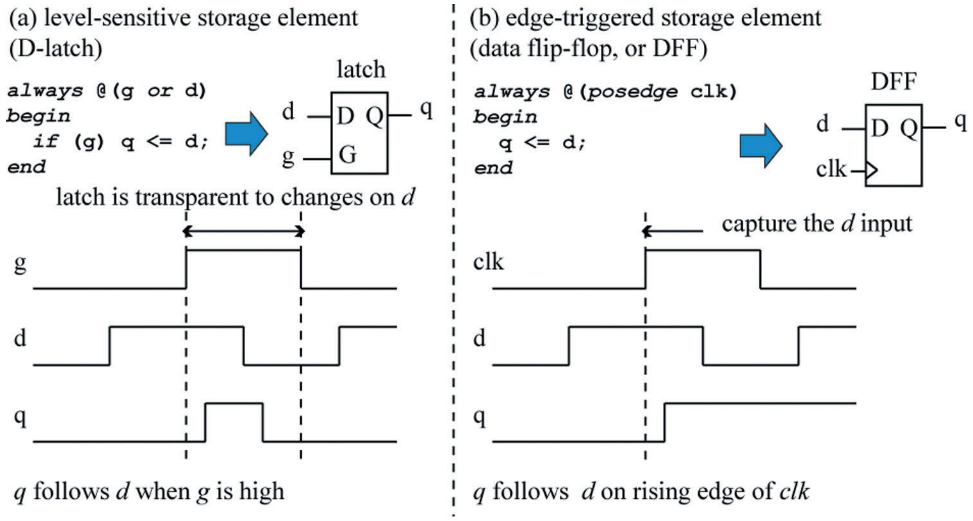
FIGURE 1.23: Tri-state logic in Verilog

direction over a single wire, but only one of the blocks can be driving the wire at a particular time. Tri-state buffers can also be used to implement bus multiplexing as shown in Fig. 1.22(c). Tri-state logic is most commonly used for signals that drive off-chip to a shared bus, but some FPGA vendors implement on-chip tri-state logic as well.

An example of tri-state bus multiplexing and its implementation in Verilog is given in Fig. 1.23. Each adder input in Fig. 1.23(a) is selected from two different 8-bit values, where the multiplexing is done by tri-state buffers. The Verilog implementation in Fig. 1.23(b) uses internal wires p and q as the adder inputs. Wire p has two continuous assignments, with each assignment using a conditional statement that selects between an input port value (a, b) and an 8-bit tri-state value given as `8'bzzzzzzzz`. The assignment of "z" to a bit value in Verilog implies a tri-state driver when the Verilog is synthesized. Inferred tri-state drivers are the only time that multiple continuous assignments can be made to the same wire in Verilog. Some synthesis tools may replace tri-state bus multiplexing with equivalent logic multiplexing if the target implementation technology does not support tri-state drivers.

1.5 SEQUENTIAL LOGIC IN VERILOG

This section reviews sequential systems, 1-bit storage elements, and common sequential building blocks, along with their Verilog implementations.



```
always @(g or d)
begin
  if (g) q <= d;
end
```

```
always @(posedge clk)
begin
  q <= d;
end
```

(a) level-sensitive storage element
(D-latch)
latch
d
g
D Q
G
q
latch is transparent to changes on *d*
g
d
q
q follows *d* when *g* is high
(b) edge-triggered storage element
(data flip-flop, or DFF)
DFF
d
clk
D Q
q
capture the *d* input
clk
d
q
q follows *d* on rising edge of *clk*

FIGURE 1.24: Level-sensitive and edge-triggered devices in Verilog

1.5.1 One-bit Storage Elements

A *sequential* system differs from a combinational system in that the sequential system has internal state (or memory) and thus its outputs are dependent upon both its external inputs and internal state. One-bit storage elements are used for state storage in a sequential system. Fig. 1.24(a) shows the logic symbol, Verilog implementation, and timing diagram for a *level-sensitive* 1-bit storage element known as a *D-latch*. The device is termed *level-sensitive* because when the gate (*G*) signal is high, then the *Q* output follows the *D* input, i.e., the latch is in *transparent* mode. When *G* drops low, then the latch's internal state becomes equal to the last *D* input value when *G* was high. The Verilog implementation of a *D*-latch is an `always` block that makes a *nonblocking* assignment ("<=") of `d` to `q` when the `g` input is nonzero. Observe that if the `g` input is zero, then the `always` block does not make any assignment to `q`, causing the synthesis tool to infer a latch on the `q` output as the `q` output must retain its last known `d` value when `g` was nonzero. Nonblocking assignments ("<=") as opposed to blocking assignments ("=") should be used in `always` blocks that are used to synthesize sequential logic; this is discussed in more detail in Section 1.6.

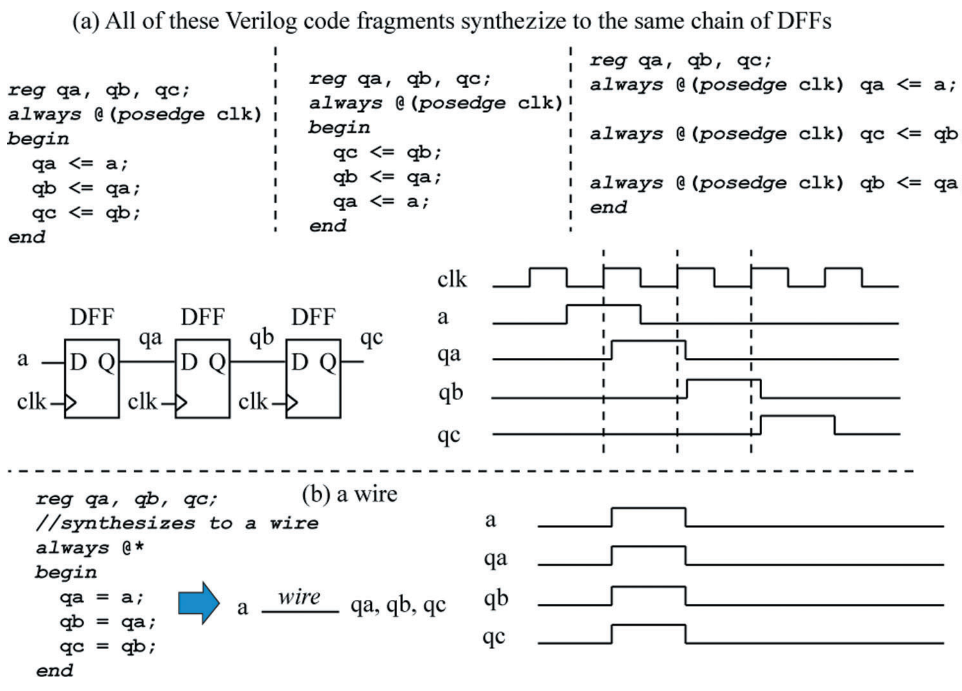
Fig. 1.24(b) gives the logic symbol, Verilog implementation, and timing diagram for a *rising edge-triggered* 1-bit storage element known as a data flip-flop (DFF). The DFF is said to be edge-triggered as the DFF's internal state is only affected on the active edge of the clock (*clk*) input, which causes the internal state to become equal to the *D* input value. The `q` output always reflects the internal state of the DFF. The DFF shown in Fig. 1.24(b) is rising-edge triggered; the logic symbol for a falling-edge triggered DFF has a bubble on the clock input.

The *D* input of the DFF is said to be *synchronous* with respect to the clock input as the *D* input only affects the *Q* output on the active clock edge. The Verilog DFF implementation indicates the sensitivity of the DFF to the rising clock edge by using the keyword *posedge* (positive edge) in the event list of the `always` block. The `d` input does not appear on the event list of the `always` block as it is the `clk` input that triggers a possible change to the `q` output, not the `d` input. The body of the `always` block simply makes a nonblocking assignment of `d` to `q`.

In general, edge-triggered storage elements are preferred to level-sensitive storage elements because of simpler timing requirements, and as such, this book uses DFFs exclusively in its designs. You may be familiar with other types of edge-triggered storage elements such as the JK flip-flop or T (toggle) flip-flop; these devices are DFFs with extra logic placed around them. The 1-bit edge-triggered storage elements provided by FPGA vendors are DFFs because of their simplicity and speed.

1.5.2 DFF Chains

The `always` blocks of Fig. 1.25(a) can be somewhat confusing to readers who are new to logic synthesis in that all of these Verilog code fragments synthesize to the same chain of DFFs as shown. This occurs because each nonblocking assignment synthesizes to a single DFF whose


(a) All of these Verilog code fragments synthezize to the same chain of DFFs

```
reg qa, qb, qc;
always @(posedge clk)
begin
  qa <= a;
  qb <= qa;
  qc <= qb;
end
```

```
reg qa, qb, qc;
always @(posedge clk)
begin
  qc <= qb;
  qb <= qa;
  qa <= a;
end
```

```
reg qa, qb, qc;
always @(posedge clk) qa <= a;

always @(posedge clk) qc <= qb;

always @(posedge clk) qb <= qa;
end
```

(b) a wire

```
reg qa, qb, qc;
//synthesizes to a wire
always @*
begin
  qa = a;
  qb = qa;
  qc = qb;
end
```

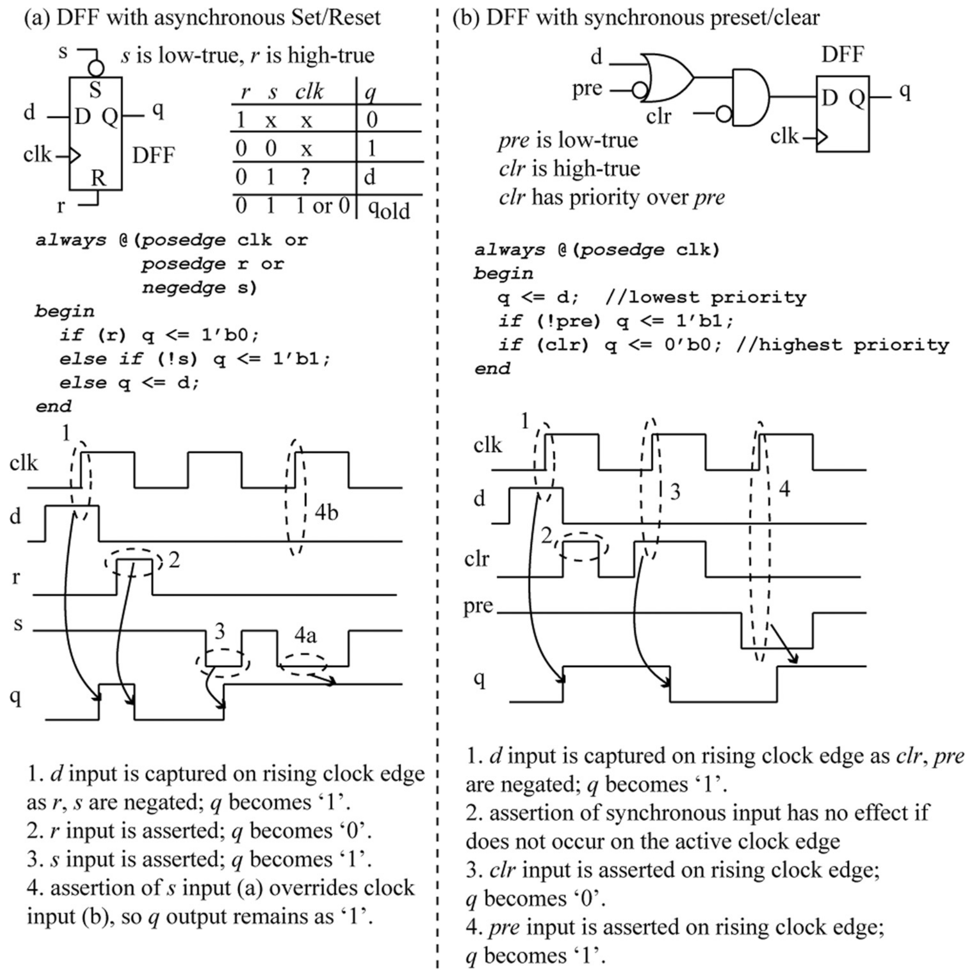
FIGURE 1.25: Synthesizing a chain of DFFs versus a wire

input happens to be the output of another nonblocking assignment. The ordering of these nonblocking assignments within an `always` block does not matter since they are assignments to different outputs, and these nonblocking assignments can be either in a single `always` block or in separate `always` blocks as shown. The key factor is that each of the nonblocking assignments is protected by "`posedge clk`" in the event list of the various `always` blocks, causing a DFF to be synthesized for that assignment. The timing diagram in Fig. 1.25(a) shows that the edge-triggered nature of the DFF causes the `qa`, `qb`, and `qc` outputs to change to the DFF input value at the rising clock edge. Contrast this to the blocking assignments in Fig. 1.25(b) that are not protected by "`posedge clk`" in the event list. This causes combinational logic to be synthesized, which simplifies to a wire from the input `a` to the three outputs `qa`, `qb`, and `qc`. Changes on the input `a` are immediately propagated by the wire to the `qa`, `qb`, and `qc` outputs as shown in the timing diagram.

1.5.3 Asynchronous Versus Synchronous Inputs

Fig. 1.26 illustrates the differences between asynchronous and synchronous inputs to a DFF. An asynchronous input affects the DFF's internal state independent of the active clock edge, while a synchronous input requires an active clock edge. The DFF in Fig. 1.26(a) has high-true reset (R) and low-true set (S) asynchronous inputs; the polarity of these inputs is arbitrary and opposite polarities were chosen for example purposes. The event list of the Verilog `always` block uses "`posedge r`" and "`negedge s`" to indicate that these inputs are high-true and low-true inputs, respectively. It is unfortunate that the Verilog keywords `posedge/negedge` are required for use with asynchronous inputs as well as with the clock input, as it implies that asynchronous inputs are edge-triggered in the same way as the clock input. However, this is not the case as asynchronous inputs are *level-sensitive* inputs, and force the DFF output either low or high as long as they are asserted, overriding the clock input. The `posedge/negedge` keywords used with asynchronous inputs indicate the leading edge of the assertion level for that input. In the `always` block of Fig. 1.26(a), observe that the *if-else* chain has the asynchronous input behavior specified first (`r`, `s`), and the synchronous behavior in the last *else* clause, indicating that the asynchronous inputs take precedence over the clock input. This *if-else* chain format must be followed with the asynchronous behavior specified first and synchronous behavior specified last for a logic synthesis tool to correctly infer a DFF with asynchronous inputs. The ordering of asynchronous inputs in the *if-else* chain determines their priority. The timing diagram of Fig. 1.26(a) indicates that the DFF's internal state is affected upon assertion of an asynchronous input.

Fig. 1.26(b) shows a DFF with synchronous clear (`clr`) and preset (`pre`) inputs. The naming of these inputs is arbitrary; in this example different names were chosen from the asynchronous set and reset inputs of Fig. 1.26(a) to avoid confusion. Observe that the only


(a) DFF with asynchronous Set/Reset
s is low-true, *r* is high-true
s
S
d
D Q
q
clk
DFF
R
r
1
clk
d
4b
r
2
s
3
4a
q
(b) DFF with synchronous preset/clear
d
pre
clr
DFF
D Q
q
clk
pre is low-true
clr is high-true
clr has priority over *pre*
1
clk
d
3
4
2
clr
pre
q

r	*s*	*clk*	*q*
1	x	x	0
0	0	x	1
0	1	?	d
0	1	1 or 0	q_{old}

```
always @(posedge clk or
         posedge r or
         negedge s)
begin
  if (r) q <= 1'b0;
  else if (!s) q <= 1'b1;
  else q <= d;
end
```

1. *d* input is captured on rising clock edge as *r*, *s* are negated; *q* becomes '1'.
2. *r* input is asserted; *q* becomes '0'.
3. *s* input is asserted; *q* becomes '1'.
4. assertion of *s* input (a) overrides clock input (b), so *q* output remains as '1'.

```
always @(posedge clk)
begin
  q <= d;  //lowest priority
  if (!pre) q <= 1'b1;
  if (clr) q <= 0'b0; //highest priority
end
```

1. *d* input is captured on rising clock edge as *clr*, *pre* are negated; *q* becomes '1'.
2. assertion of synchronous input has no effect if does not occur on the active clock edge
3. *clr* input is asserted on rising clock edge; *q* becomes '0'.
4. *pre* input is asserted on rising clock edge; *q* becomes '1'.

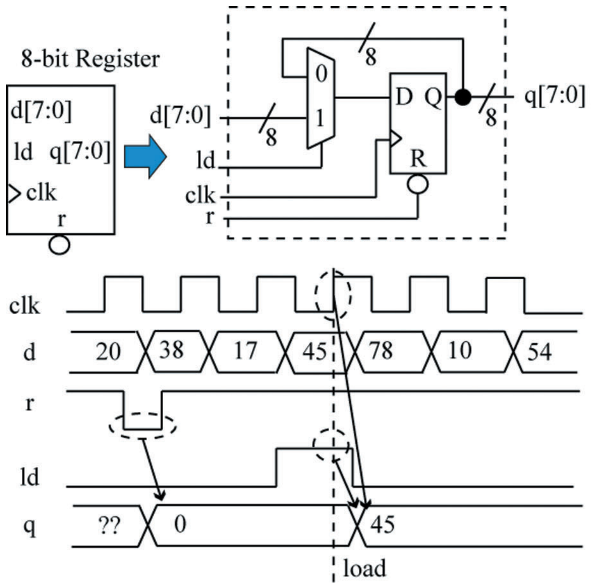
FIGURE 1.26: Asynchronous versus synchronous inputs to a DFF

input that appears in the event list of the `always` block is the clock (`clk`) input as this DFF has no asynchronous inputs. The logic shown is a conceptual gate-level implementation of the synchronous clear and preset operations that are specified in the `always` block; be aware that the target implementation technology determines the actual synthesized logic. The `always` block is written such that the lowest priority assignment to the `q` output is done first, and the highest priority assignment to `q` is made last. An *if-else* chain similar to that of Fig. 1.26(a) could have also been written; unlike the asynchronous `always` block, there is no particular format for specification of synchronous logic. The timing diagram of Fig 1.26(b) indicates that assertion of a synchronous input does not affect the DFF's internal state until the next active clock edge.

In general, asynchronous inputs to DFFs are reserved for power-on logic, while synchronous inputs are used during normal operation.

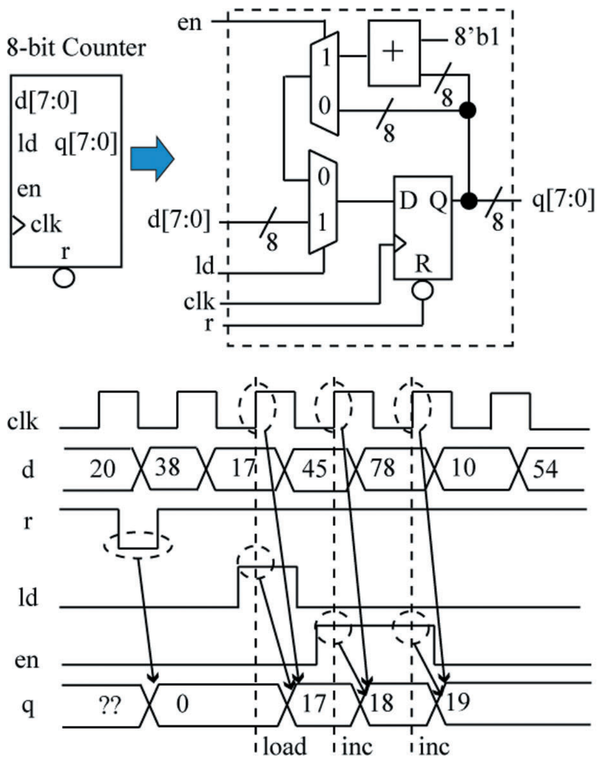
1.5.4 Registers, Counters, and Shift Registers

The combinational building blocks of Section 1.4 and DFFs can be combined to form commonly used sequential building blocks such as registers, counters, and shift registers. A common need in a sequential system is the ability to store an N-bit value over several clock periods. Paralleling N-DFFs forms a block that can store an N-bit value, but the problem with DFFs is that they can change value on every active clock edge. Fig. 1.27 shows that a *register* is built by combining a DFF with a 2-to-1 multiplexer to form a device that only changes its value when the load (`ld`) input is asserted on the active clock edge. The register contained in Fig. 1.27 is an 8-bit register with an asynchronous low-true reset (`r`). The multiplexer steers the external `d` input of the register to the D input of the DFF when `ld` is asserted, causing a new value to be loaded on the next active clock edge. If the `ld` input is negated, then the register's output is steered back to the DFF input so that the next active clock edge simply loads the register's old value, thus retaining the register's contents. The gate-level implementation in Fig. 1.27 is provided for conceptual purposes; the actual gate-level implementation depends on the target technology. The Verilog implementation is an `always` block of sequential logic whose synchronous section



```
module reg8bit(clk,r,d,ld,q);

input clk,r,ld;
input [7:0] d;
output [7:0] q;

reg [7:0] q;

//register
always @(posedge clk
         or negedge r)
begin
  //async reset
  if (!r) q <= 8'b0;
  else begin
  //synchronous inputs
    if (ld) q <= d;
  end
end

endmodule
```

FIGURE 1.27: Register with low-true asynchronous reset

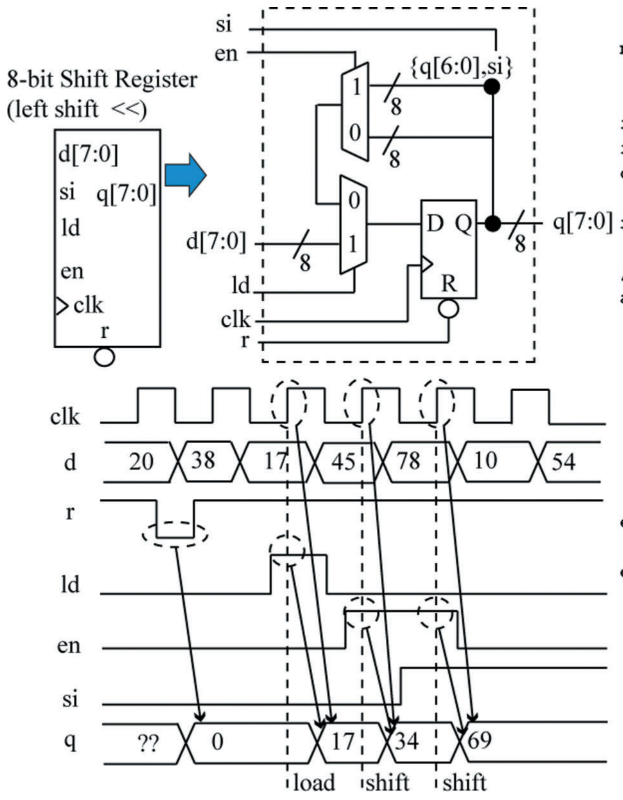


```
module cnt8bit (clk,r,d,
                ld,en,q);

input clk,r,ld,en;
input [7:0] d;
output [7:0] q;

reg [7:0] q;

//register
always @(posedge clk
         or negedge r) begin
 //async reset
 if (!r) q <= 8'b0;
 else begin
 //synchronous inputs
  if (en) q <= q + 8'b1;
  if (ld) q <= d;
 end
end

endmodule
```

FIGURE 1.28: Binary up-counter with low-true asynchronous reset

is an *if* statement that assigns the `d` input to the `q` output whenever the `ld` input is nonzero. If the `ld` input is zero, then no assignment is made to `q`, causing it to retain its last value.

Fig. 1.28 gives the logic symbol, conceptual gate-level logic diagram, timing diagram, and Verilog implementation for an 8-bit up-counter. In comparison to the register, the up-counter logic symbol has one extra control signal named `en`; the counter increases by one when `en` is asserted on an active clock edge. The timing diagram shows the counter loaded with the value 17 by asserting `ld` on an active clock edge, followed by assertion of `en` for two consecutive clock cycles causing the counter to increase to 18, and then to 19. The counter then holds the value 19 stable as both `ld` and `en` are negated for the remaining clock cycles. The Verilog implementation is the same as for the register except that the `always` block contains an extra *if* statement that performs the increment of `q<=q + 8'b1` when `en` is nonzero. The *if* statement for the load capability follows the *if* statement for the increment operation, giving load precedence over increment if both `ld` and `en` are asserted in the same clock cycle. This matches the multiplexer structure that is shown in the conceptual gate-level logic diagram. As one might expect, counters are useful for counting the number of operations performed in a digital system. However, another



```
module sreg8bit (clk,r,d,ld,
                 en,si,q);

input clk,r,ld,en,si;
input [7:0] d;
output [7:0] q;

reg [7:0] q;

//shift register
always @(posedge clk
         or negedge r) begin
 //async reset
 if (!r) q <= 8'b0;
 else begin
 //synchronous inputs
  if (en) q <= {q[6:0],si};
  if (ld) q <= d;
 end
end

endmodule
```

FIGURE 1.29: Shift register (left shift) with low-true asynchronous reset

common use of counters is for providing addresses to a memory system, as memory locations are typically accessed in a sequential fashion.

Fig. 1.29 gives the logic symbol, conceptual gate-level logic diagram, timing diagram, and Verilog implementation for an 8-bit shift register (left shift). This has the same structure as the counter except that the incrementer in the multiplexer feedback path has been replaced by a 1-bit wired shift-left using the s_i input as shift-in for the least significant bit. The timing diagram loads the shift register with the value 17, and then shifts this value to the left in the next consecutive clock cycle by asserting the en input. The result of the first shift left is 34 (17*2+0) as the s_i input is "0," while the result of the following left shift is 69 (34*2 +1) because s_i is now "1." The Verilog implementation is the same as for the counter, except that the *if* statement that implemented the increment operation has been replaced by an *if* statement that implements the shift-left capability. One use of shift registers is for fast multiplication or division by 2. Another more common usage is for serial communication, which sends or receives an *N*-bit datum 1 bit at a time over a serial link.

1.6 EVENT-DRIVEN SIMULATION AND VERILOG

In previous sections, combinational and sequential building blocks have been presented along with their Verilog representations; these building blocks are used to form digital systems. To verify the functionality of a digital system, both the Verilog code and the synthesized gate-level netlist can be simulated in a *digital logic simulator*. This section covers the basics of how digital logic simulators operate, and the differences between Verilog simulation (presynthesis) and gate-level simulation (postsynthesis).

1.6.1 Event-Driven Simulation Basics

Various approaches have been taken in the past in creating digital logic simulators. One of the first approaches is that of a so-called Levelized Compiled Code (LCC) simulation. The advantage of this approach is speed since each circuit is transformed into a computer program that is compiled. The disadvantage of this approach is that all timing information is lost and thus these types of simulations are often referred to as 0-delay (zero-delay) simulations.

In order to generate timing information, an event-based simulation model is typically used. An event is defined as a logic-value change in a net at some instant in time. Event-driven (ED) simulations are quite different from LCC since events can occur simultaneously in time in real circuits and thus the parallelism inherent in logic circuits is more accurately modeled. To illustrate how a basic ED simulator operates, consider the example circuit in Fig. 1.30.

In the example circuit, all of the nets are given labels (A–F, H, K–N, P). Events are defined for a net when a logic level changes. In a basic ED simulator, a list or a queue is maintained that contains every net for which an event occurred at some instant of time. After this event list is built, it is traversed and a new list is built, called the gate queue. Whenever an event occurs on

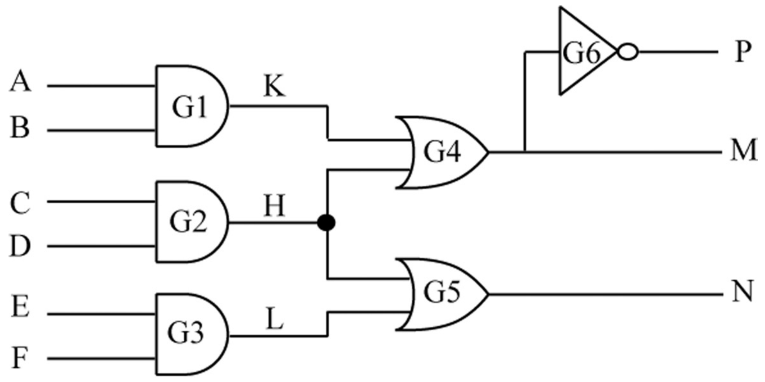


FIGURE 1.30: Example circuit for ED simulation

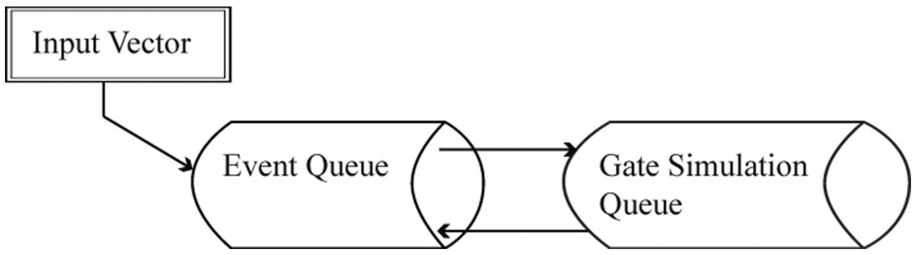


FIGURE 1.31: Timing wheel diagram

an input net to a gate, the simulator must simulate the gate to determine if the gate output net undergoes a corresponding event. The event queue and the gate queue are alternately formed and processed and the simulation is over when the queues are empty. Because the event and gate queues are alternately filled, processed, and emptied, this structure is sometimes referred to as the "timing wheel" and a diagram of a timing wheel is depicted in Fig. 1.31. The smaller box in the upper-left corner represents a new set of input stimulus values that initiate events on the input nets and cause a new simulation to occur. These are typically referred to as "test vectors" and the choice of an appropriate set of test vectors is a very important factor in effective simulation as well as using an efficient and timing accurate simulator. Here, we will focus on the anatomy of the timing accurate simulator only.

Using the timing wheel structure and the concept of an event-driven simulation, the example circuit is used to describe how the ED simulation occurs. Consider that the example circuit has been fully simulated for a test vector of A=1, B=1, C=0, D=0, E=1, F=0 and that at some instant in time the input test vector changes to A=0, B=0, C=0, D=0, E=1, F=1. This will cause events to occur on nets A, B, and F as shown in Fig. 1.32. In the figure, affected nets are shown with a previous logic value and the current logic value separated by a

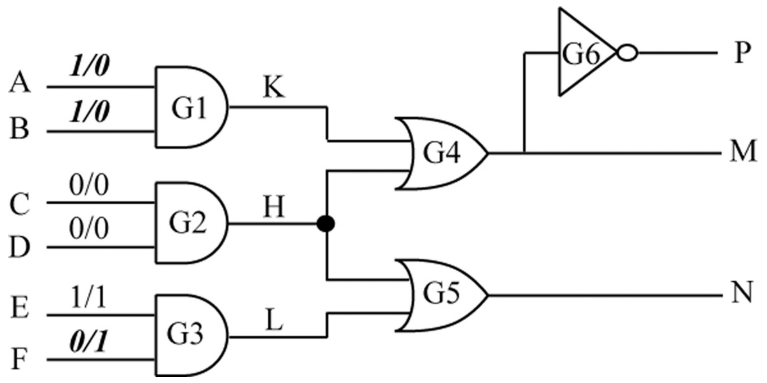


FIGURE 1.32: ED simulation example

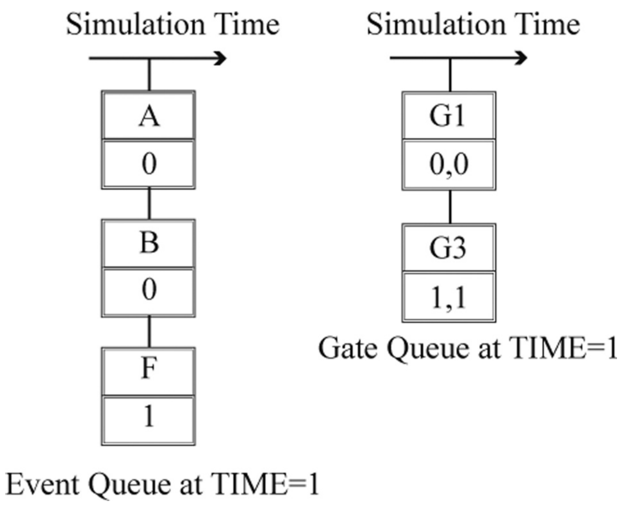


FIGURE 1.33: Event and gate queue content

slash. Whenever these two values are different, an event has occurred and these are denoted by the use of a ***bold-italic*** font.

The first step of the simulation is to build a list of events in the event queue. Each event may have other information included in the queue entry such as the net identifier, the new logic value, and other information. A very simplified figure illustrating the event content is shown in Fig. 1.33.

At this point in the simulation, the name of each net and the current logic value are stored in the event queue for all detected events. The simulator engine will next process this list of events and together with a graph depicting the circuit structure, the gate queue will be filled with a list of gates for which each event is driving (or fanning in to). As an example, event entries for nets A and B both serve as inputs to gate G1; thus this gate is placed in the gate queue along with all current input values. Note that G1 is placed in the gate queue only once.

As each event is processed, it is actually removed from the queue. However to better describe these concepts, we will leave previous events in the event queue diagram and draw the newer events to the left. As sets of events progress from left to right, a notion of time can be inferred from the simulation. For now we are assuming all gates have an equal delay and this is referred to as a "unit-time" simulation model. To complete this part of the simulation, each gate in the gate queue is simulated and if an event occurs on the gate output a corresponding entry is placed into the now empty event queue. Two time units later, the queue diagrams are shown in Fig. 1.34 and the corresponding circuit (or netlist) is shown in schematic form in Fig. 1.35. At this point one more gate will be scheduled for simulation (the inverter G6) and no more gates

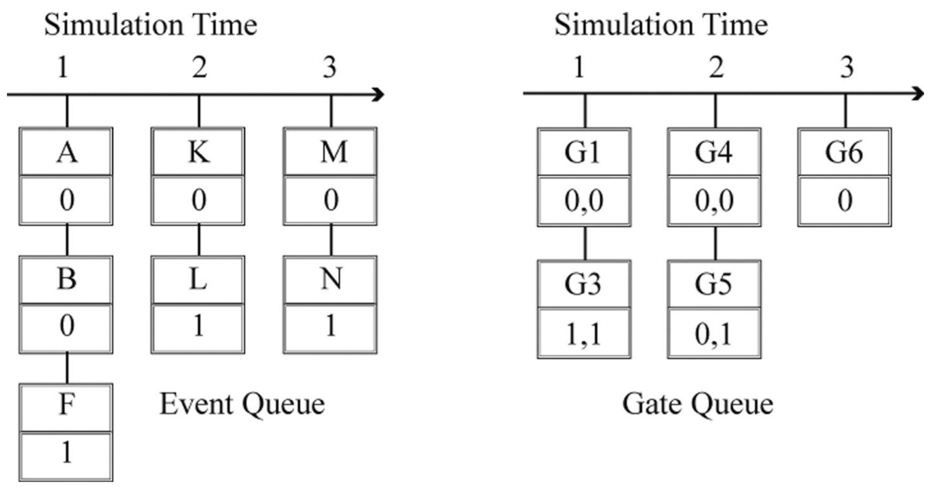


FIGURE 1.34: Event and gate queues at simulation time 3

will be scheduled afterward, yielding a total simulation time of four units. **Bold** font is used for events (previous/current logic values that differ) that have been processed during this time, with ***bold-italic*** font used for events that are yet to be processed. Since actual gate-level simulations are being performed, only those gates in the circuit that require simulation are simulated. As an example, gate G2 was never simulated for this test vector. Although only gates that require simulation are simulated, it is possible that the same gate can be simulated more than once for a single simulation run. Another important aspect of ED simulation is that the order in which the gates are simulated inside the gate queue does not matter; the scheduling mechanism ensures that all gates present in the gate queue for a given time instant are operating in parallel.

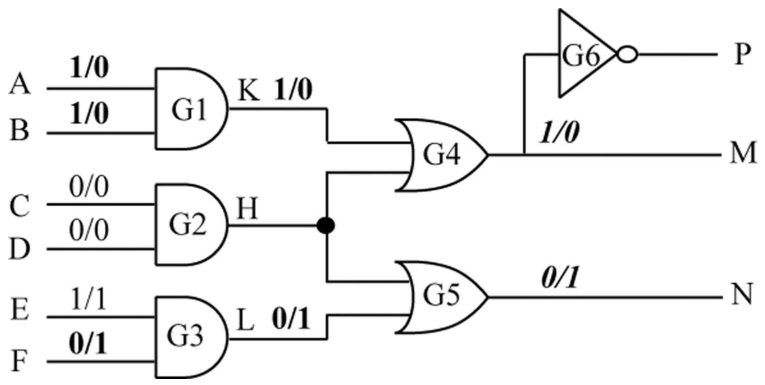


FIGURE 1.35: Example circuit with event annotations at simulation time 3

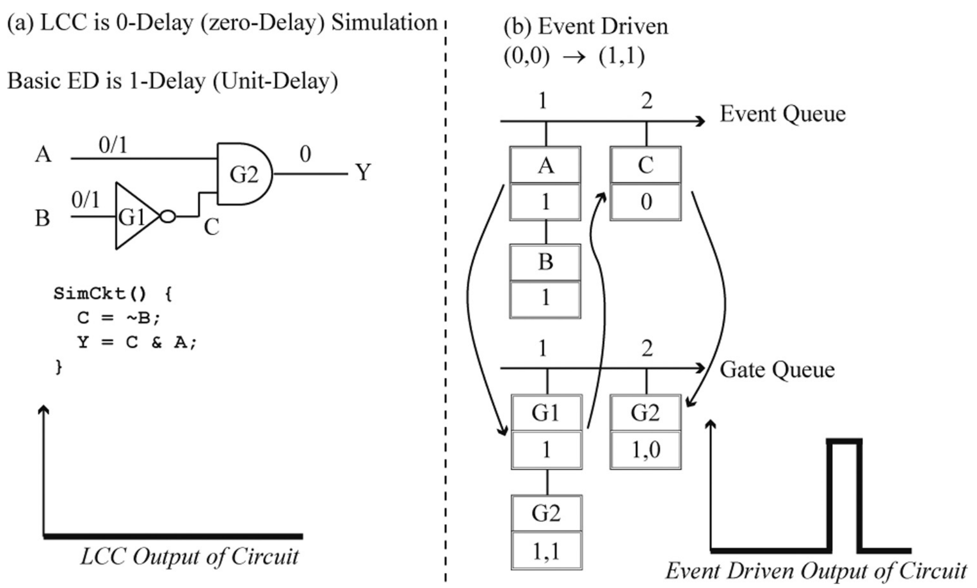


FIGURE 1.36: Example showing two simulations for same gate

1.6.2 Timing Considerations

As mentioned previously, a gate may be scheduled for simulation more than once for a particular test vector. This occurs when unequal delay paths are present in a circuit and actually allows for more accurate timing behavior in the simulated circuit. As an example, consider the simple circuit and the associated event and gate queue content in Fig. 1.36. In this example, the bottom-left corner of the figure contains the LCC code (in the C programming language) and the associated output waveform (all zero-valued) when the input vector changes from (0,0) to (1,1). On the right side of the figure, the queue content and the resulting output waveform are shown for a unit-delay ED simulation. Note that the AND gate is scheduled for simulation twice, once when the top-most input net undergoes an event, and then one time unit later when the output of the inverter causes an event to occur. This more accurately describes real circuit behavior and the output waveform shows that the AND gate outputs a logic-1 value briefly due to the delay in the inverter.

The unequal delays in the paths from the primary circuit inputs to the AND gate inputs cause the circuit output to go high for an amount of time equal to the unit-delay associated with the inverter. For more accurate timing, variable-delay nominal values could be used since a single time value is still inadequate to accurately capture detailed timing information. It is possible to also run three simulations using minimal, maximal, and nominal delays but this increases the overall design time and some timing errors can still be masked. Most event-driven simulators operate with worst-case delay models and thus produce pessimistic timing results.

1.6.3 Presynthesis Versus Postsynthesis Simulation

Verilog simulators are event driven, and the Verilog code that has been used in the examples to this point all use zero-delay assignments, either blocking, nonblocking, or continuous. Nonzero delays can be specified in Verilog assignments by using a # operator and a delay value. However, the examples in this book all use zero-delay assignments as the provided Verilog code is written for synthesis purposes, and the gate-level netlist synthesized from the Verilog code provides the delays. When simulating Verilog, one must be aware if the simulation is *presynthesis* (also known as *functional*) or *postsynthesis*, as the postsynthesis simulation reflects the timing delays of the implementation technology. Fig. 1.37(a) shows the C code of Fig. 1.36 written as a Verilog `always` block and synthesized to two different implementation technologies. The functional simulation in Fig. 1.37(a) shows no glitch on the output *Y* when *A* and *B* both transition to 1 simultaneously as it is a zero-delay simulation. The postsynthesis simulation in Fig. 1.37(b) has a timing glitch on the output *Y* as the implementation technology uses discrete gates with physical delays. Fig. 1.37(c) shows a different implementation technology in which the synthesis tool maps the `always` block logic to a 4 location by a 1-bit (4 × 1) memory device (a *lookup table*). This eliminates the intermediate *C* value, and thus there is no glitch in the *Y* output of the simulation even though the memory device has a nonzero delay. Presynthesis simulation

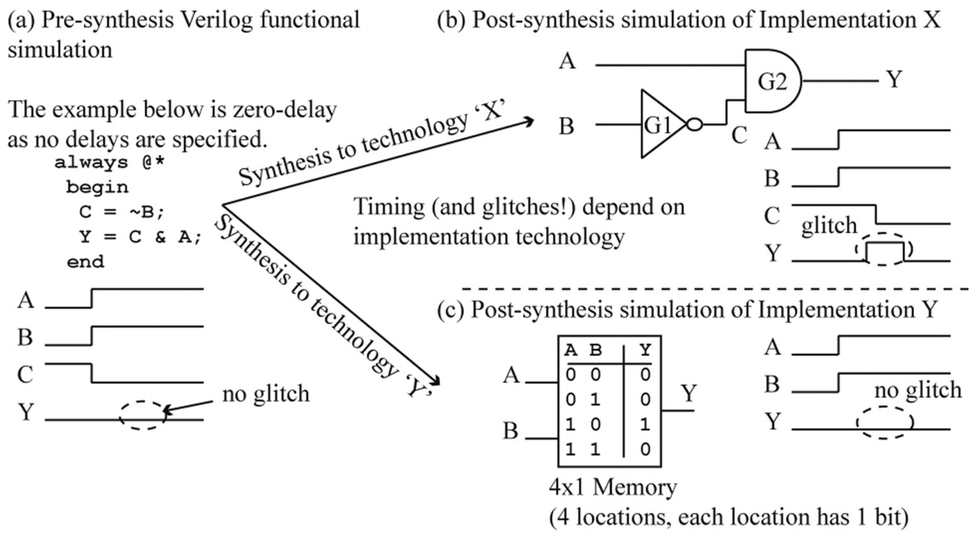

(a) Pre-synthesis Verilog functional simulation
The example below is zero-delay as no delays are specified.
always @*
begin
C = ~B;
Y = C & A;
end
A
B
C
Y
no glitch
Synthesis to technology 'X'
Synthesis to technology 'Y'
Timing (and glitches!) depend on implementation technology
(b) Post-synthesis simulation of Implementation X
A
B
G1
G2
C
Y
glitch
(c) Post-synthesis simulation of Implementation Y
A B Y
0 0 0
0 1 0
1 0 1
1 1 0
4x1 Memory
(4 locations, each location has 1 bit)
no glitch

FIGURE 1.37: Presynthesis versus postsynthesis simulation

requires much less CPU time than postsynthesis simulation for complex systems that require a large number of test vectors for verification.

1.6.4 Blocking Versus Nonblocking Assignments and Synthesis

In the synthesis examples presented to this point, blocking assignments ("=") are used in `always` blocks that synthesize combinational logic, while nonblocking assignments ("<=") are used in `always` blocks that synthesize to sequential logic. This is due to the manner in which these two assignment types are handled by the Verilog event-driven simulation model, and its translation to hardware elements. What follows is a simplified explanation of the differences between these two assignment types; the reader is referred to [1] for a complete discussion. Zero-delay blocking assignments are so named because the assignment of the right-hand side (RHS) to the left-hand side (LHS) is completed without any intervening Verilog code allowed to execute, i.e., the assignment *blocks* the execution of the other Verilog code. For nonblocking assignments within an always block, all RHS expressions are evaluated, and are only assigned to the LHS targets after the `always` block completes. This causes a logic synthesis tool to treat blocking assignments differently from nonblocking assignments in terms of the synthesized logic. Fig. 1.38(a) shows an `always` block with two blocking assignments whose event list contains "`posedge clk`," inferring that the logic to be synthesized is edge triggered. The first blocking assignment "`q1 = d`" is triggered on the rising clock edge, and thus a DFF is synthesized to represent this assignment. Because this is a blocking assignment, the assignment is assumed to complete before the next assignment. Thus, the `q1` on the RHS of the second assignment "`q2 = q1`" is the `q1` after the clock edge has occurred, or is the `q1` that represents the output of the synthesized

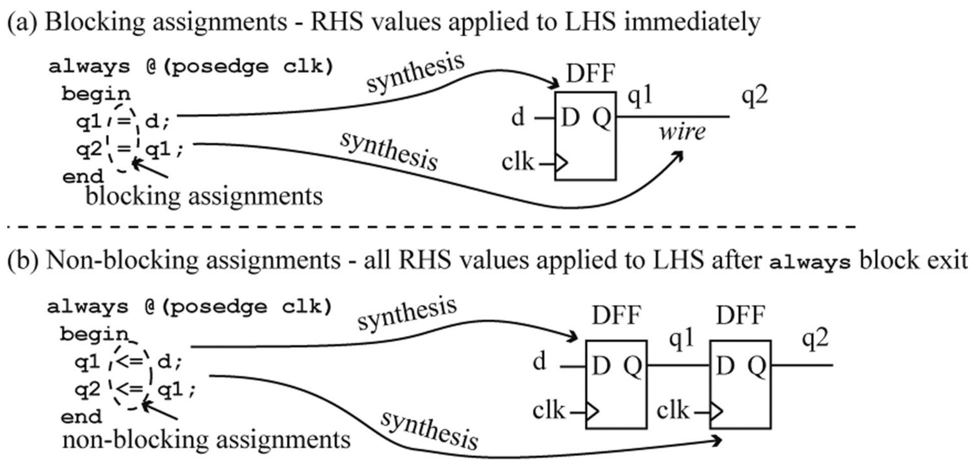


FIGURE 1.38: Blocking versus nonblocking assignments

DFF. The second blocking assignment "`q2 = q1`" copies this `q1` to the `q2`, which synthesizes to a wire as previously seen in Fig. 1.25(b).

Contrast this with Fig. 1.38(b), which is the same `always` block except that the blocking assignments have been replaced by nonblocking assignments. In this case, the RHS values of `d` and `q1` are their values at the time the process is triggered by the rising clock edge as all RHS values of all nonblocking assignments are evaluated before any nonblocking LHS assignments are made. This causes the synthesis tool to infer a DFF for each of these assignments, forming the DFF chain as shown in Fig. 1.38(b).

1.7 VERILOG CODING GUIDELINES

The following list contains a few guidelines adopted from [3] for the Verilog HDL that will help users who are new to logic synthesis to write the Verilog RTL code that will synthesize to the user's expected hardware realization.

1. Use blocking assignments ("=") in `always` blocks that are meant to represent combinational logic.
2. Use nonblocking assignments ("<=") in `always` blocks that are meant to represent sequential logic.
3. Do not mix blocking and nonblocking assignments in the same `always` block. If an `always` block contains a significant amount of combinational logic that requires intermediate wires (and thus, intermediate assignments), then place this logic in a separate `always` block.
4. If an `always` block for combinational logic contains complicated logic pathways due to *if-else* branching or other logic constructs, then assign every output a default value at the beginning of the block. This ensures that all outputs are assigned a value regardless of the path taken through the logic, avoiding inferred latches on outputs.
5. Do not make assignments to the same output from multiple `always` blocks.

Also, pay attention to any warnings that the logic synthesis tool issues when compiling and synthesizing the Verilog RTL. These warnings provide alerts for unusual conditions that often indicate a mistake in coding. A few common warnings provided by Verilog logic synthesis tools are as follows (exact wording is simulation tool dependent):

1. "Input X is unused (does not drive any logic)." This means that the synthesized logic does not make use of a particular input.
2. "Output X is stuck at VDD (or GND)." This means that in the synthesized logic there is an output that has been reduced to a fixed "1" or "0" with no gating.

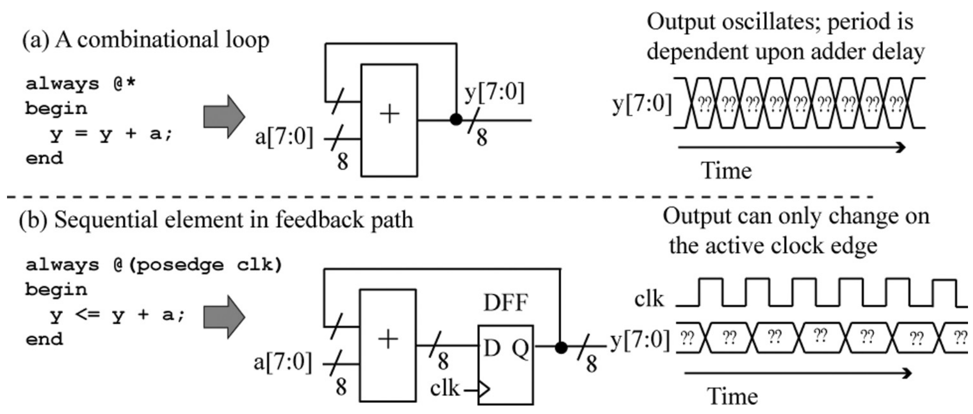


FIGURE 1.39: Combinational loop

3. "Outputs X and Y share the same net." This means that the logic for outputs *X* and *Y* is the same, and that outputs *X* and *Y* are driven by the same gate.
4. "Output X has no driver." This means that an output has never been assigned, and thus no logic has been synthesized to drive it.
5. "Combinational loop detected on net X." This means that the synthesis tool has found a feedback path from a combinational gate output back to one of its inputs without an intervening latch or DFF. Unless an *asynchronous* (i.e., no clock is used) digital system is being designed, this is an error in coding as all feedback paths should be broken by a sequential element. 1.39(a) shows an example of a combinational loop formed by feeding the output of an adder back to one of its inputs. For a nonzero value on the input a, this causes the adder to oscillate with a period equal to the delay path through the adder. To correctly sum the previous adder output with a new input value, a sequential element such as a DFF must be placed in the feedback path. This allows the clock signal to control the sequencing of new output values from the adder.

1.8 SUMMARY

Hardware description languages and logic synthesis offer a powerful mechanism for the specification and implementation of digital systems. This chapter has reviewed digital logic fundamentals in a Verilog HDL context, which is our HDL of choice in this book.

CHAPTER 2
Synchronous Sequential Circuit Design

This chapter will review the underlying theory of sequential circuits and will heavily emphasize synchronous sequential circuits. Topics such as ASM charts, state encoding, Verilog descriptions, and their effect on resulting automatically synthesized circuits are included. These types of circuits are very common and useful when they are used as controllers for guiding input data through a data processing circuit.

2.1 LEARNING OBJECTIVES

After reading this chapter, you will be able to perform the following tasks:

- Describe a synchronous sequential circuit in contrast to other types of circuits.
- Express the operation of a synchronous sequential circuit in terms of an *Algorithmic State Machine* (ASM) chart and other common models.
- Understand how a synchronous sequential circuit can be used as a controller for a datapath.
- Describe what state encoding is and what the effects are in terms of the resulting synchronous circuit.
- Develop a *Register Transfer Level* (RTL) Verilog description of a synchronous sequential circuit.
- Be familiar with different Verilog coding styles for a synchronous sequential circuit.
- Develop either Mealy- or Moore-type synchronous sequential circuits and understand their differences in output signal timing.

2.2 SEQUENTIAL CIRCUITS

Sequential circuits differ from combinational logic circuits in that the outputs of these circuits depend on both the input signals and the value of an internal state. The state value is present by way of a memory capability in the sequential circuit. Memory circuits can be constructed in a variety of ways including the 1-bit storage elements described in Chapter 1. Sequential circuits can be broadly classified as being synchronous or asynchronous referring to the presence or absence of a periodic signal as an input.

2.2.1 Sequential Circuit Motivation

The common sequential devices described in Chapter 1 are used so commonly that we refer to them as sequential logic building blocks. In general, a sequential circuit may have any set of arbitrary states and these types of circuits are extremely useful in the design of digital circuits in the role of a controller. We will use the terms "controller" and "sequential circuit" interchangeably in the following text, although we will always assume that we are dealing with synchronous controllers, those that depend on a synchronizing clock signal for state changes.

There is interest in synchronous controllers since they are very commonly used as part of an entire digital system, including processors, bus bridges, physical layer protocol devices, and other digital circuits. These types of digital systems can be viewed as being composed of two main subcircuits: a datapath and a controller. Although this chapter is not devoted entirely to datapath + controller circuit design, a brief description of the architecture of these types of circuits is provided here to give context for the remainder of the discussion on controllers.

A general block diagram of a digital system composed of a datapath and a controller is shown in Fig. 2.1. The datapath is the portion of the circuit that contains components that transform input data signals into output data signals. Datapaths may be purely combinational or they may also contain synchronous components such as dedicated counters. The controller

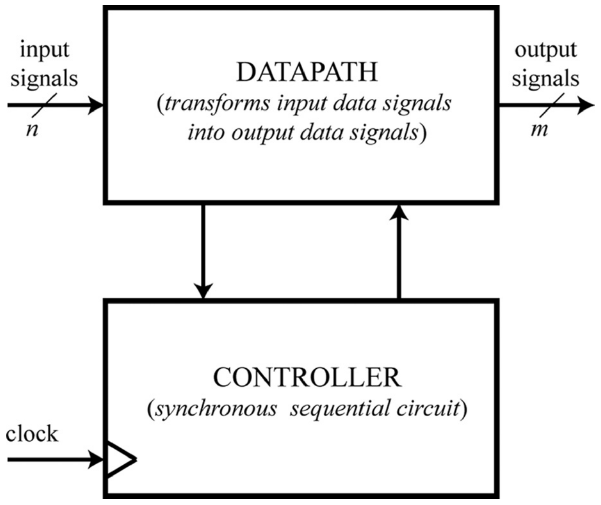

input signals
n
DATAPATH
(transforms input data signals into output data signals)
output signals
m
CONTROLLER
(synchronous sequential circuit)
clock

FIGURE 2.1: Diagram of digital circuit composed of a datapath and a controller

subcircuit is responsible for guiding the input signals through various components in the datapath and can allow the "sharing" of components in a datapath. For example, if a digital system needs to apply a multiplication operation to two different sets of input signals, but only one multiplier circuit is present in the datapath, the controller can generate signals that sequences and allows different input signals to be present at the multiplier circuit inputs at different times. Common types of signals that the controller circuit provides to a datapath include device reset signals, multiplexer select signals, register load and clear signals, and in general any signal that is considered to be a "control" versus a "data" signal. The controller may also receive input signals generated by the datapath. An example might be a completion signal generated by a datapath component that would in turn be used by the controller to change state.

2.2.2 Synchronizing Signals: The Clock

In this chapter, we focus on synchronous circuits, which are those that require a synchronizing periodic signal (or a clock signal) as an input. Fig. 2.2 illustrates a typical clock signal as a voltage versus time plot with various parameters such as period (τ), frequency (f), pulse width (P_W), pulse height (P_H), and duty cycle (D_C) as a percentage depicted. The clock signal in the figure is ideal in that it is depicted with zero-valued rise- and fall-times. In reality, the rise- and fall-times are finite valued and not necessarily equal; however, unless stated specifically, we can assume this ideal model for the purposes of our discussion. The P_H parameter is shown in units of voltage, which is the most common unit used for modern digital circuits since they are based on voltage mode circuits at the transistor level.

During the simulation of synchronous circuits using Verilog, it is necessary to generate a clock signal. This can be accomplished by a separate Verilog module for that purpose. Two Verilog modules are shown in Fig. 2.3 that can serve as clock signal generators. The first module `clk_gen1` produces an output clock signal that is composed of exactly 50 clock cycles. The

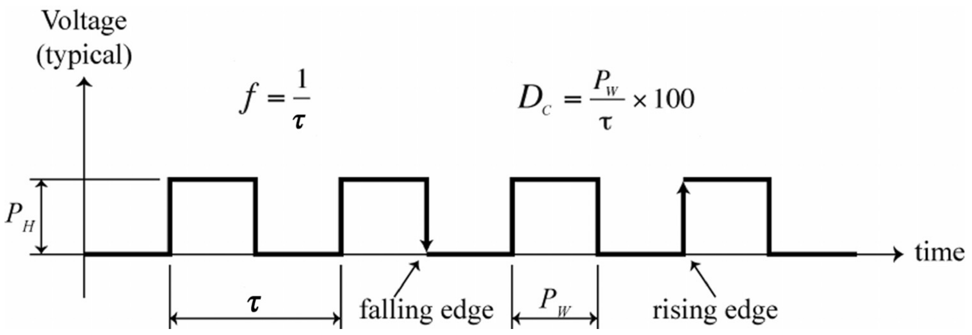


FIGURE 2.2: Typical synchronous sequential circuit clock signal

```
module clk_gen1(clk);
output clk;
reg clk;                      module clk_gen2(clk);
initial                       output clk;
    begin                     reg clk;
       clk = 1'b0;            initial  clk = 1'b0;
       repeat (100)           always    #5 clk = ~clk;
       #5 clk = ~clk;          endmodule
    end
endmodule
```

FIGURE 2.3: Verilog modules for clock signal generation

`repeat (100)` statement causes the Verilog line `#5 clk = ~clk;` to be repeated 100 times. Every 5 simulation time units, the clock signal changes polarity so 10 time units are required for one complete period. The `clk_gen2` module generates a clock signal that continually oscillates until the simulation is halted, because the `#5 clk = ~clk;` statement is present in an `always` block. The clock signal produced by the Verilog modules in Fig. 2.3 have duty cycles of 50%, a pulse width of 5 simulation time units, and a period of 10 simulation time units. More sophisticated clock generators can be written that allow for varying the duty cycle of the clock generation circuit.

For practical reasons in terms of circuit synthesis, the clock generator module should be written as a separate module and not included as part of the module containing your circuit design description, since the purpose of the clock generator module is for simulation only. After your circuit has been simulated and you are ready to synthesize it, you will not want the synthesis tool to produce clock generation circuitry. By keeping the clock generator as a separate module, you can remove it from your file when you are ready to synthesize the circuit and provide the synthesis tool with only the module describing the circuit under design.

Many synthesis tools only support a synthesizable subset of the Verilog language. As an example, the clock signal generation modules in Fig. 2.3 cannot be simulated using the Altera Quartus tool since the timing delay parameters (e.g., `#5`) are not supported since the Quartus tool is designed for synthesis and supports only the synthesizable constructs of Verilog. Other simulation tools may require the insertion of simulator directives such as `$stop` or `$finish` in the clock modules for proper operation.

2.2.3 Synchronous Sequential Circuit Architectures

Just as combinational logic circuits are commonly modeled as binary-valued Boolean functions, it is also common and convenient to model controller circuits based on a mathematical model known as an automaton. The two most popular automata models for controller circuits are referred to as Moore or Mealy machines, named after researchers who published early papers on their structure [4, 5]. The distinguishing characteristic between Mealy and Moore machines is that the output of a Moore machine depends only upon the state of the sequential circuit whereas the output signals of a Mealy machine depend upon both the state and the input signals of the circuit. This has a practical effect in that the output signals of a Moore machine only change after output logic delays following a clock signal edge whereas the output signals of a Mealy machine may change at any time shortly after an input signal changes value. Because the outputs may change at any time for a Mealy machine, the outputs for a given state transition can only be considered to be valid shortly after a state transition that occurs in relation to a clock edge event.

Block diagrams of the architecture of synchronous Mealy and Moore machines are shown in Fig. 2.4. Each of these is shown with n input lines, m output lines, k present state values, and q excitation values. The k present state values are the content of the k 1-bit storage memory elements called the state of the circuit. The q excitation values are used as inputs to the storage devices so that their content may change during the next clock event. If the storage devices are simple registers or D flip-flops, then the $k = q$ and the q signals are referred to as the next state. The m output signals are produced by the lower combinational logic blocks in the figure. Note that the output combinational logic is optional for the Moore model. Many times Moore machines are designed such that some or all of the k state values are the desired circuit output signals. This is generally desirable in terms of timing since the output signals of a Moore machine that are actually the present state values will only change shortly after a clock event and will remain stable until the next clock event.

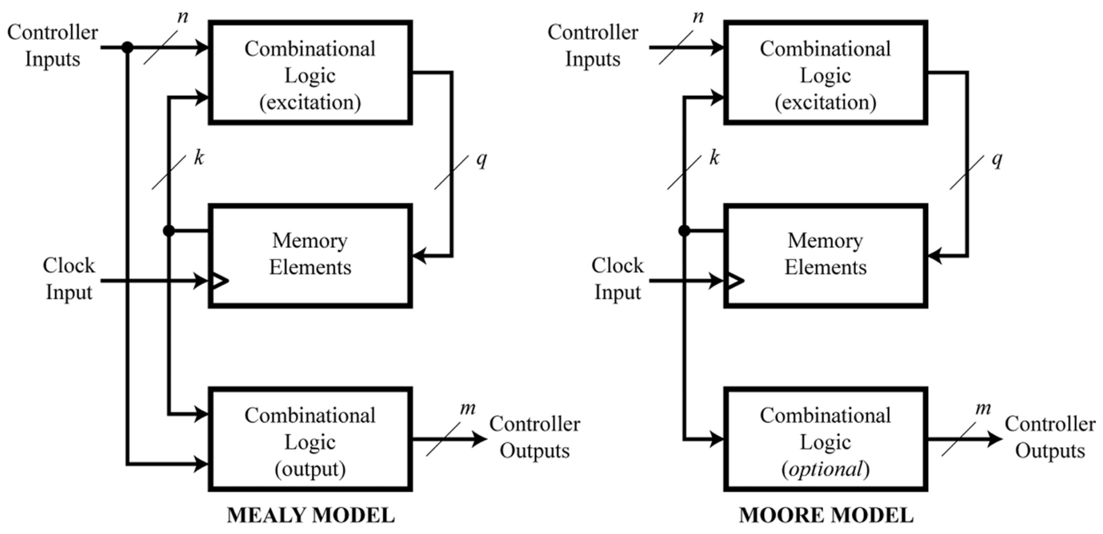


FIGURE 2.4: Block diagrams of Mealy and Moore machine digital circuitry

In comparing the two diagrams in Fig. 2.4, the Moore model has the same structure as the Mealy model with the exception of a missing bus connecting the input signals to the output combinational logic block. This has a practical effect in terms of building circuits that adhere to the Mealy- or Moore-type structure. In reality, it is usually the case that some outputs are derived directly from the present state values only and we will refer to these as "Moore-type" outputs. Other outputs may depend on both the input values and the present state; these are the "Mealy-type" outputs. Circuits containing both Moore-type and Mealy-type outputs are still technically Mealy machines. It is usually desirable from a design point of view to build circuits with as many Moore-type outputs as possible since those output signals have the desired property of changing only shortly after a clock event and they remain stable throughout the entire clock period.

Since Moore and Mealy machine models are mathematical structures, there is a rigorous theoretical background underlying the Mealy and Moore models referred to as the automata theory. The automata theory is the basis behind the traditional model of computation and is used for many purposes other than controller circuit design, including computer program compiler construction, proofs of algorithm complexity, and the specification and classification of computer programming languages [6]. While these subjects are not covered here, we mention this in order to point out that the synchronous sequential circuits we are studying here are a subset of the more general category of automata models.

Because automata are mathematical models that produce values dependent upon some internal state and possibly some dependent input values, they are also often referred to as *state machines*. An automaton may allow for a finite or an infinite set of possible states and furthermore, they may have deterministic or nondeterministic behavior. A deterministic state machine is one whose outputs are always the same for a given internal state and a set of dependent input values. A *finite state machine* (FSM) is one where all possible state values form a finite set. The synchronous sequential circuits that are the focus of this chapter are conveniently modeled as deterministic finite state machines that are modeled as either Mealy or Moore machines. For this reason, these types of circuit models are sometimes referred to as FSMs in the literature. Although an FSM refers to a type of mathematical model of an automaton, it is sometimes the case that designers refer to a controller as an FSM when technically a controller is a circuit that is a realization or embodiment of an FSM model.

2.3 MODELS OF FINITE STATE MACHINES

FSMs are commonly modeled by digital designers in a variety of ways, including state diagrams, state equations, state tables, and *algorithmic state machine* (ASM) charts. This section will provide an overview of each of these types of models for an example FSM. Our method for controller circuit design will involve specifying the behavior by first developing an FSM model and then translating the model into a Verilog description, which in turn will be synthesized yielding a circuit realization. Verilog descriptions are also models but we will describe the development of these models in a separate section since our preferred approach for controller design is to first develop an ASM chart and then to translate the chart into a Verilog description.

2.3.1 Basics of Algorithmic State Machine (ASM) Charts

Algorithmic State Machine (ASM) charts are a way to model a synchronous sequential circuit and these are our preferred model to use before the generation of a Verilog description commences. There are several reasons for this preference:

- ASM charts strictly adhere to a few rules and this allows them to be easily translated directly into the Verilog HDL description.
- ASM charts are usually much easier to comprehend by a human than a Verilog listing.
- The combination of ASM charts and a Verilog listing provides a very powerful and comprehensive form of documentation for a synchronous sequential circuit.
- ASM charts provide an easy way to distinguish between Moore- and Mealy-type outputs.

ASM charts are a rigorous form of flowcharts that were initially introduced for describing software programs. ASM charts consist of three different symbols connected by directed edges that depict the flow of events within a circuit. As shown in Fig. 2.5, a single rectangular box is present for each state, an oval is present for circuit outputs that depend on both input values and the current state (note that these dependent output ovals are only present in Mealy model descriptions), and the diamond shaped symbols are used for decisions in next-state transitions based on circuit inputs.

The rectangular state-representing symbol is sometimes optionally shown with a state encoding or a state name. If these are present in the ASM chart, they should be shown outside the box since everything inside the state-representing symbol refers to an unconditional (or Moore-type) output signal. It is also common to designate one of the state-representing signals as a "reset" or "initialization" state. While there is no strict convention for denoting this state, common practice is to have a flow direction symbol pointing to the rectangle with the other end labeled "RESET." State-representing symbols may have one or more flow direction symbols pointing to them but they should always have exactly one exiting flow direction symbol.

The diamond-shaped decision symbol is used to allow signal flow to vary in a conditional manner based on an input signal. Decision symbols should always contain an input signal name or a Boolean expression composed of input signals. Basic decision symbols are single-bit valued which implies that they have exactly two exiting flow direction symbols, one indicating that

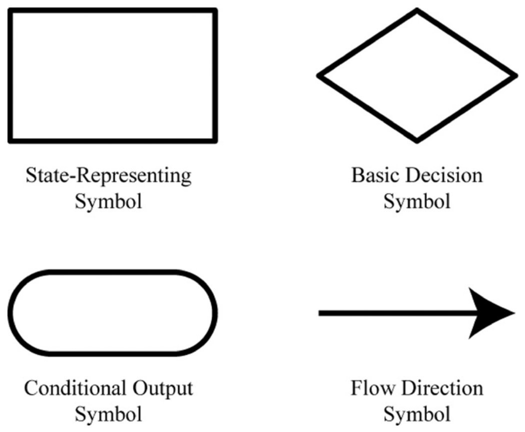


FIGURE 2.5: Symbols used in ASM charts

the contained expression or input signal is a "1" and the other indicating evaluation to a "0." Each of the two exiting flow direction arrows is labeled by having either a 0 or a 1 next to them. A basic decision symbol always has a single input flow direction symbol originating from either a state-representing symbol or another basic decision symbol. The two outputs of a basic decision symbol should either point to a conditional output symbol or to a state-representation symbol.

A conditional output symbol is only used in Mealy machine ASM representations. These symbols always have exactly one entering and one exiting direction flow symbol. Output signals or equations defining output signals are contained within conditional flow boxes. In a correctly drawn ASM chart, the entering direction flow symbol to a conditional symbol must always originate from an output of a decision flow symbol. For this reason, the output designated by the conditional symbol depends (or is conditional upon) on the input signal in the decision flow symbol. These ASM chart symbols are sometimes referred to as Mealy output symbols since they denote outputs that depend on both the current state and an input signal value. The exiting flow direction arrow points to a state-representing symbol or another basic decision symbol.

We refer to the diamond-shaped symbol as a basic decision symbol since some designers use a more general decision symbol with more than two output flow direction arrows that depend on more than one input signal. The more complex generalized decision symbol can always be represented by a cascade of basic decision symbols, each labeled with a single input signal and having two (binary) exiting flow arrows. The order of the cascade of basic symbols does not matter since evaluation of a path through the decision boxes is assumed to occur simultaneously, shortly after a clock event in a synchronous sequential circuit; nevertheless, some people prefer

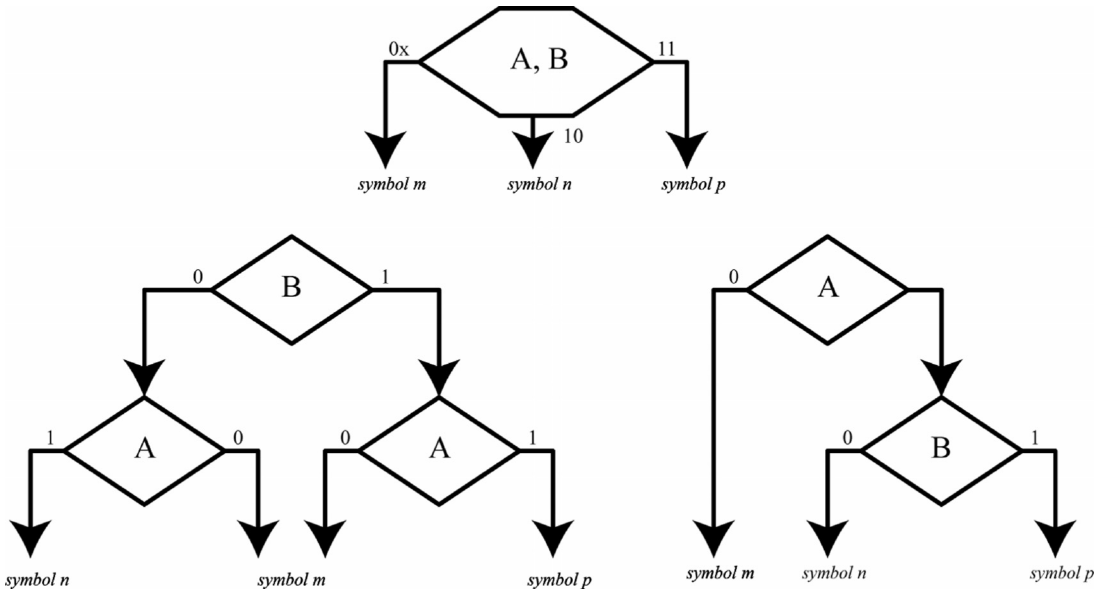


FIGURE 2.6: Complex and corresponding basic ASM charts

using these more general decision symbols since they avoid the issue of ordering in a cascade of basic decision symbols. A common way to draw a general decision symbol is shown in Fig. 2.6 on the top and two corresponding cascades of basic decision symbols are shown below.

All the three ASM chart portions shown in Fig. 2.6 are equivalent. The top decision symbol has three exiting flow symbols that correspond in total with all four different 2-bit values of the input signal pair A, B. In the worst case, the general decision symbol would have four different exiting flow symbols corresponding to the four different 2-bit valuations of the ordered pair of input signals A, B = {00,01,10,11}. For conciseness, the don't care x is used to indicate that if the input signal A = 0, the activated path in the chart is to the ASM symbol m. The two cascades of basic decision symbols are identical to the generalized symbol. There are advantages and disadvantages of each convention. The advantage of the generalized symbol is clearly the conciseness of the ASM chart. Also, if fully generalized decision boxes are used everywhere, the rule about the exiting flow symbol from conditional output symbols may be simplified to state that they must always point to a state-representing symbol. It is permissible to insert conditional output symbols in intermediate stages of cascades of flow symbols, provided that they follow all decision symbols whose input signals affect the conditional output signal. The disadvantage of the generalized decision symbol is that the designer must ensure that all possible combinations of input variables are uniquely covered. That is, there must be one and only one exiting decision symbol for all possible combinations of input signals inside the generalized decision symbol.

Finally, because there are two possible ways to express the generalized decision symbol using basic decision symbol cascades in Fig. 2.6, many designers who are just learning about ASM charts mistakenly assume that the order in which the decision boxes are encountered have some bearing on the underlying circuit. This is not the case since ASM charts represent synchronous sequential circuits. In synchronous sequential circuits, all decision boxes are evaluated as soon as the input signals change regardless of the order in which they are drawn in an ASM chart. This can allow the paths through the decision cascades to change as input signals change. However, in terms of effecting state transitions, only the activated path shortly after a clock event matters.

2.3.2 The ASM Chart Model and an Example Controller

The same example FSM is used for all the models described in this chapter. The example is intended to represent a controller circuit that receives inputs from a datapath that consists of a memory with extra circuitry that allows for writing a sequence of zero values over a range of addresses. The additional circuitry in the datapath consists of two registers for holding the lower and upper addresses, a counter for generating intermediate addresses, a comparator for determining when the upper address value has been reached, and multiplexers for switching the memory device input signals between the "zeroing" function and normal operation. A simplified block diagram of the datapath is shown in Fig. 2.7 with the controller circuit represented as a large box at the bottom of the diagram.

The datapath is designed to allow the memory to either function normally or in a *zeroing* mode. During normal functioning, addresses (addr[5:0]) are connected to the memory address input (addr) through the uppermost multiplexer (M1) and data to be written to the memory is passed through the lowermost multiplexer (M2) to the memory data input port. In the zeroing mode, the multiplexer select lines are changed to connect the output of the counter to the memory address input and the connection to a constant logic "0" to the memory data input port (din). One of the purposes of the controller circuit is to generate the appropriate multiplexer select line values.

In this example, it is assumed that the memory with a zeroing capability circuit is interfaced to other circuitry that is responsible for ensuring that the appropriate addresses are stored in registers R1 and R2 before a zeroing operation is asserted. The zeroing operation is asserted by external circuitry to indicate that the system should switch from normal operation to the zeroing mode. This external input signal, `zero`, serves as input to the controller which in turn will issue the appropriate signals to the datapath to initiate the zeroing mode. The overall circuit generates

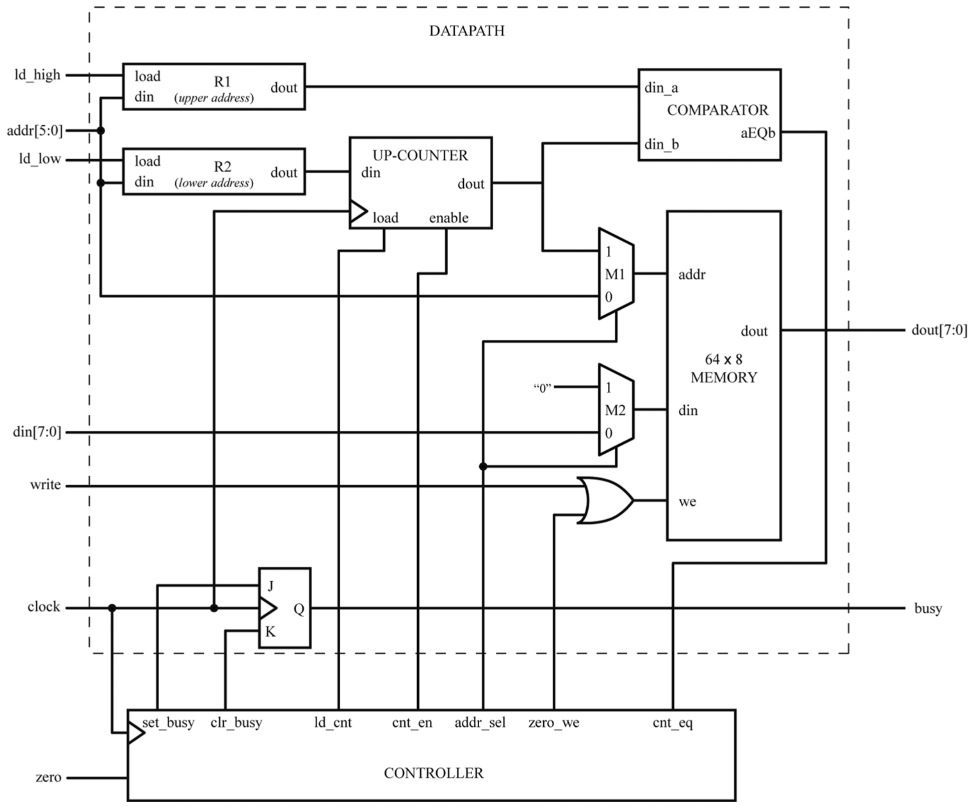


FIGURE 2.7: Block diagram of a datapath and controller for an example digital system

one output, `busy`, that is generated by the controller outputs `clr_busy` and `set_busy` via a JK flip-flop in the datapath. The output signal, `busy`, can be used by other external circuitry to determine when normal memory reads and writes can occur. The two-input OR gate in the datapath is needed to allow the single memory write enable (`we`) to be controlled externally during normal memory operations, or by the controller during the zeroing operation. The counter is responsible for generating the next incremental address to which a "0" value is to be written during each clock cycle of a zeroing operation. This counter has the capability to have an initial value loaded, in this case from R2, and is an up-counter. The output of the counter is fanned out to both the M1 multiplexer and an input of a comparator. The purpose of the comparator is to determine when the zeroing operation is complete. This is accomplished by comparing each generated address from the counter with the last address to be zeroed present in register R1. When the addresses are the same, the comparator outputs the signal `cnt_eq`

TABLE 2.1: Example Controller Input and Output Signal Names

INPUTS		OUTPUTS	
DESCRIPTIVE NAME	PURPOSE	DESCRIPTIVE NAME	PURPOSE
		`Set_busy`	Controller output indicating zeroing operation has begun
`zero`	External input causing zeroing operation to begin	`Clr_busy`	Controller output indicating zeroing operation has ended
		`load_cnt`	Controller output causing R2 content to be loaded into counter
`cnt_eq`	Input from datapath causing zeroing operation to halt	`addr_sel`	Controller output connected to select inputs of M1 and M2
		`zero_we`	Controller output asserting the we input of the memory
		`cnt_en`	Controller output asserting the enable input of the counter

that serves as an input to the controller, which in turn generates the control signals that allow the datapath to resume operation as a normal memory unit.

Examination of the Fig. 2.7 indicates that our example controller has two input signals from the datapath and produces six output signals that control the datapath. The inputs and outputs of the controller are summarized in Table 2.1.

Fig. 2.8 contains an ASM chart that models the example controller. When constructing the ASM chart, it is natural and easy to determine which outputs are Moore-type versus Mealy-type—this is an advantage of the ASM chart. State `s0` represented by the top state-representing symbol is the state at which our example circuit is behaving as a normal memory. The decision box below `s0` indicates that at each clock event, the input zero is checked to determine if an external request for a zeroing operation has been asserted, if so the controller transitions into

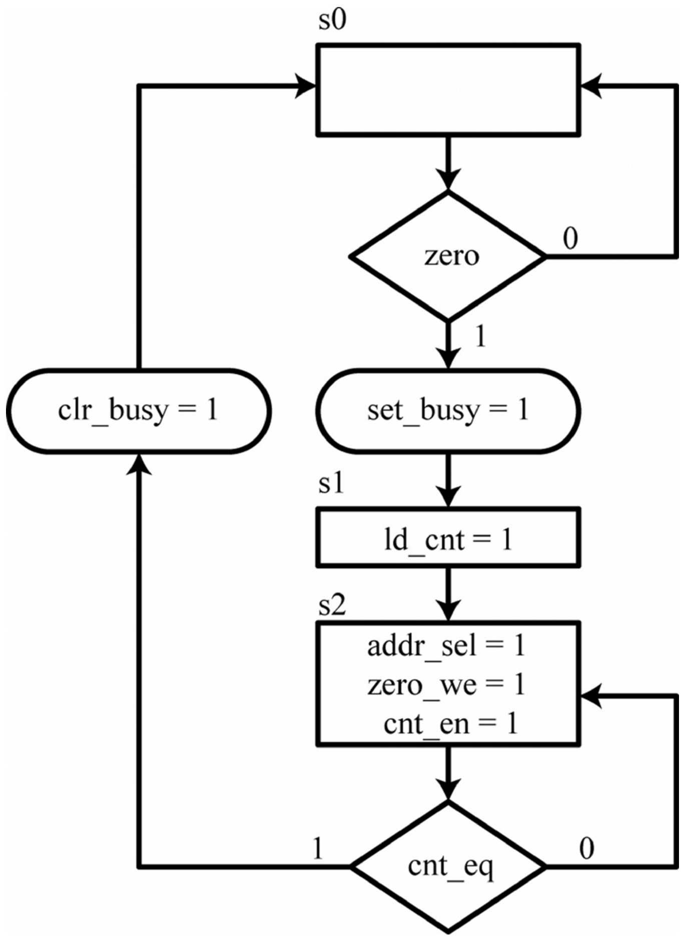


FIGURE 2.8: Example ASM chart for a controller

state `s1`, otherwise it remains in state `s0`. Note that whenever the controller is in state `s0` and `zero` is asserted, the conditional output `set_busy` is asserted. This implies that `set_busy` is a Mealy-type output since it depends upon the input signal `zero`.

After `zero` has been detected as asserted, and the controller has transitioned into state `s1`, it is necessary to load the counter in the datapath with a low-range value of the memory address where the zeroing operation is to begin. This is accomplished when the controller asserts the unconditional output `ld_cnt` signal that is connected to the load input of the counter. The purpose of state `s1` is to initialize the counter to begin at the address that was loaded into register R2. The controller stays in state `s1` for exactly one clock cycle since there is no decision block after the `s1` block in the ASM chart.

The controller transitions into state `s2` and remains there as long as the zeroing operation is occurring. While in state `s2`, the controller input `cnt_eq` is checked at each clock cycle as indicated by the lowermost decision box. The `cnt_eq` signal is the output of the comparator in the datapath and is asserted whenever the counter is equal to the upper range address present in register R1. When `cnt_eq` is asserted, the controller can then transition back to state `s0` and generate the appropriate control signals that cause the unit to resume operation as a normal memory device. While the controller is in state `s2`, the three unconditional outputs `addr_sel`, `zero_we`, and `cnt_en` are asserted. The `addr_sel` signal is connected to the select lines of the multiplexers M1 and M2 and causes the `addr` input of the memory to receive data from the counter and the `din` input of the memory device to be connected to a logic "0" value. The `zero_we` signal that is connected to the input of the OR gate is asserted, causing the memory device to be in the write mode. The `cnt_en` signal is the enable signal for the counter, and its assertion causes the counter to increase by one at each clock edge. Note that the `clr_busy` conditional output signal is asserted when the controller is in state `s2` and the `cnt_eq` signal is asserted. The `clr_busy` signal is also clearly a Mealy output since it depends upon both the state (`s2`) and the controller input `cnt_eq`.

The four unconditional outputs of the controller are the signals `ld_cnt`, `addr_sel`, `zero_we`, and `cnt_en`. These depend only upon the state of the controller and for this reason they appear (when asserted) inside the state-representation symbols. Typically, when unconditional outputs such as these four signals are not asserted, they are not present in the state boxes of the ASM chart. For example, `ld_cnt=0` is not written inside the `s0` or `s2` box in the ASM chart. The nonasserted values are referred to as default values and they have a direct translation into default Verilog statements when a controller is described with Verilog.

2.3.3 The State Diagram Model

The state diagram model for a deterministic FSM is a directed graph with a vertex set where each member of the set uniquely maps to each possible state and an edge set that represents all possible transitions from one state to another during a single clock event. The vertices are usually drawn as circles or ovals that are annotated with a label representing a state and the edges may be labeled with circuit input and/or output values.

It is easy to recognize whether a state diagram represents a Mealy or a Moore machine by examining the annotations on the graph edges. The state diagram for a Mealy machine contains edge annotations of the form *input/output* where the "/" symbol is used to delimit the input labels or values from the output labels or values. The inputs and outputs may be given in terms of symbolic or numerical values. Fig. 2.9 is a Mealy-type state diagram representation of our example controller with the numerically labeled edges. The edge labels should be interpreted as the input and output signal values in the order (`zero cnt_eq/set_busy`

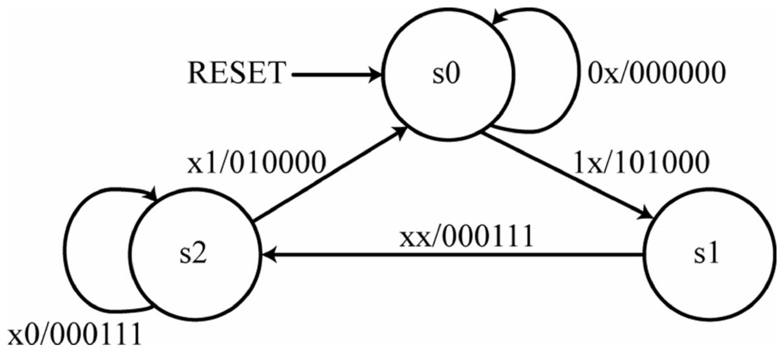


FIGURE 2.9: State diagram for a FSM representing a two-input/six-output Mealy machine

`clr_busy ld_cnt addr_sel zero_we cnt_en`). If symbolic edge labels were used then instead of 1x/101000 for the edge label indicating a transition from state `s0` to `s1`, we would use (`zero x/set_busy` $\overline{\texttt{clr\_busy}}$ ld_cnt $\overline{\text{addr\_sel}}$ $\overline{\text{zero\_we}}$ $\overline{\text{cnt\_en}}$).

If the state diagram has no "input/output" designations on each transition arc, then the machine is a Moore model. Moore machines have outputs that are dependent only on current state values. In some cases the output signals may be equivalent to some or all of the state bits. In general, the outputs are specified values in the vertices of the state transition diagram, typically separated by the state values by using a "/". Alternatively, the particular output signals can directly be encoded into each state vector. Fig. 2.10 contains two state diagrams representing Moore machines that are equivalent to the Mealy machine model shown in Fig. 2.9.

While these two state diagrams look the same, there are differences in the underlying circuits that are synthesized from the two. The leftmost state diagram indicates the state

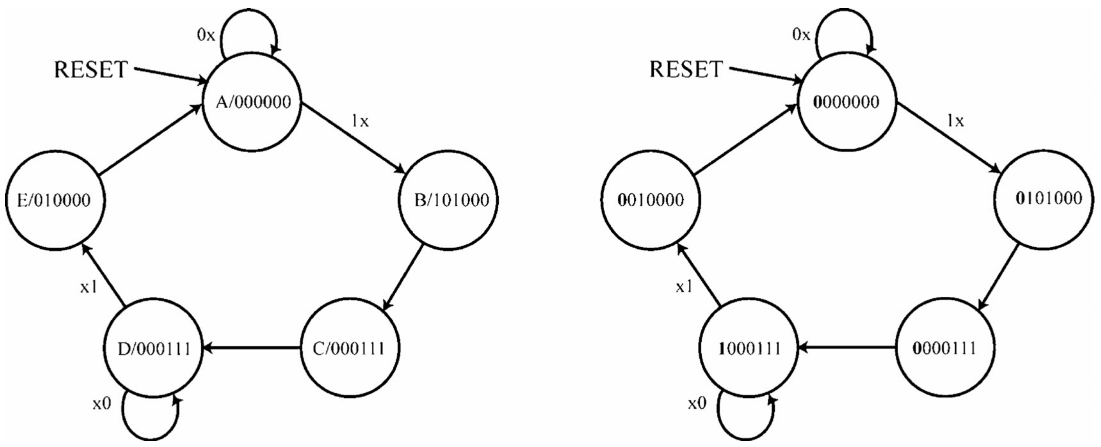


FIGURE 2.10: Two state diagram representations of the same Moore machine

symbolically {A, B, C, D, E} with the output signals shown to the right of the "/" symbol. The rightmost state diagram gives the state encoding values explicitly and the output signals are the least significant 6 bits of each state vector that are not present in bold. Note that an additional bit is needed in each state vector (the bolded bit) since the output signals are identical for states D and C. Referring to Fig. 2.4 where block diagrams of the architecture of controller circuits are shown, note that the Moore model has a lower block labeled optional output logic. This combinational logic is present when a circuit is synthesized from the state diagram on the left in Fig. 2.10. In this case, it is a combinational logic circuit whose inputs are only the current state values that are assigned to {A, B, C, D, E}. Because the outputs depend only on the state values, this satisfies the definition of a Moore machine; however in this case, the outputs are not registered. Conversely, when the circuit is synthesized from the state diagram on the right, outputs are directly particular state values and are thus registered.

These two types of Moore circuits have advantages and disadvantages. When the combinational output logic block is used, the number of bits representing each state may be minimized resulting in using fewer memory devices; however fewer memory devices can cause an increase in the amount excitation combinational logic as well as requiring an output logic block. Additionally, the production of the output signals by propagating current state values through an output logic block causes their stabilization to occur at different times and can cause the signals to exhibit static hazard behavior (bouncing or ringing) after each state transition. For these reasons, we recommend that Moore machine always be implemented where output values are state values directly when possible.

2.4 STATE ASSIGNMENT

State assignment is the process of assigning a unique bit string to each state represented in a description of a controller circuit. An example of state assignment was given in the rightmost state diagram of Fig. 2.10 where each state was labeled by a unique bit string rather than a symbolic symbol as is shown in the ASM chart example in Fig. 2.8, the Mealy state diagram in Fig. 2.9, and the leftmost Moore machine state diagram in Fig. 2.10. Before a controller can be implemented, the symbolic symbols representing each state must be transformed into a unique set of identifying bit strings. For each bit in these binary-valued state names, a memory element is synthesized.

While any arbitrary assignment can be made to a state machine as long as each state has a unique bit string, the exact assignment will affect the resulting synthesized logic in terms of both area and maximum clock frequency. The theory of optimal state assignment has been studied for many years and this problem has been proven to be intractable, which is to say that no existing method is known for finding the best assignment that does not require runtime proportional to trying all possibilities. Furthermore, the state assignment that yields the least amount of circuitry is likely not the same as the state assignment that allows the circuit to be clocked as fast as possible, so tradeoffs are present.

The minimum number of bits possible in a state encoding for a controller with N states is $\lceil \log_2(N) \rceil$; thus for the example ASM chart above we need $\lceil \log_2(3) \rceil = \lceil 1.5849 \rceil = 2$ bits for each state. Since there are 4! different possibilities, there are 24 different possible controller circuits possible for a minimum state-bit encoding. As the number of states increases, this combinatorial property increases dramatically, and this is only for minimal-bit encoding. Furthermore, studies have shown that fewer than 5% of all possible minimal-bit encodings are optimal in terms of area [7] for some controllers.

Because the problem is intractable, designers generally rely on using "rules of experience" or heuristic methods to perform state encoding. A heuristic is an observation of behavior that usually yields good results. As an example, for most controllers, if the state bits are mostly the same (but at least one bit must differ) between adjacent states, then the resulting circuit is generally a good tradeoff between speed and area. Adjacent states are any two states in an ASM chart that are connected either by a single direction arrow or by a path through decision and conditional outputs only. In the best case, we would have only a single bit changing in the state encoding of two adjacent states and we would have as few state bits as possible labeling each state. When this can be accomplished, it is referred to as a Gray code state encoding. However, Gray code state encoding is not always possible depending on the shape of the ASM chart. In our example ASM chart, Gray code state encoding is impossible since there are no possible ways to assign unique bit strings to the adjacent pairs {(`s0`, `s1`), (`s1`, `s2`), (`s2`, `s0`)}.

The next best alternative is to assign bit patterns that differ by only two bits for every adjacent pair of states. It turns out that this is always possible, and one way to achieve this is to use the so-called *one-hot* encoding method. In one-hot encoding, the number of bits in each state label is equal to the total number of states in the controller. The name one-hot comes from the fact that for each bit string only a single bit is a "1" and all others are a "0." As an example, a one-hot encoding for the example ASM chart in Fig. 2.8 is {`s0`=001, `s2`=010, `s3`=100}. Because only a single bit is a "1" in each encoding, we are guaranteed to have a distance of 2 between any two adjacent states regardless of the shape of the ASM chart.

Many times, state encoding simply does not matter and a simple arbitrary minimum assignment method works fine. When you develop your Verilog module, you will need to make an arbitrary state assignment before synthesis (note that newer HDLs such as SystemVerilog allow for symbolic enumerations that avoid this). We recommend that you make such an arbitrary assignment and validate that your controller is functionally correct first. Only in later stages of design iterations should you consider using a heuristic state assignment technique or an automated tool to optimize your design; however it is important to be aware of this issue. In highly optimized designs, such as modern microprocessors, good state assignment is essential.

It has also been our experience that one-hot encoding is generally only beneficial if your controller has more than approximately 10 states, and this benefit is generally in terms of increasing the maximum clock frequency. For small controllers, we recommend using a minimal-bit encoding strategy and assigning the codes such that as many of the adjacent state pairs as possible differ by a single bit. Finally, we note that some synthesis tools such as Synopsys Design Compiler allow for separate scripts to be employed where a designer can specify a type of state encoding to be used, such as one-hot. These tools analyze the Verilog code and automatically invoke an automated state assignment algorithm to determine the actual state assignments based on an area versus performance tradeoff constraint.

2.5 LOW-LEVEL MODELS OF CONTROLLERS

The preferred approach for controller design is to generate an ASM chart and then to immediately transform this description into a corresponding Verilog RTL-level description. This description can then be supplied as an input to a synthesis tool to produce a netlist file.

The synthesis task can be performed manually resulting in a low-level model of the controller; however, this is not practical for controllers of even moderate size. For the sake of completeness and pedagogy, we describe two low-level models of controllers in this section: state equation descriptions and a state table.

2.5.1 State Equations

A set of state equations describing a controller consists of a complete set of transition equations and output signal equations. State equations differ from binary Boolean logic equations in that the concept of discrete time is included within them. For synchronous sequential circuits, state equations are often written using a discrete time notation where t represents values in the current state and $t+1$ represents the next state. Transition equations are those that give information about next state values, states at time $t+1$, as a function of inputs and present state values, states at time t. Output equations are those that provide information about output values at time t as a function of input and state values at time t.

State equations may be written either before or after state assignment and memory element selection. When they are written before state assignment, there is exactly one transition equation for each state in the ASM chart and when they are written after state assignment and memory element selection, there is exactly one transition equation for each bit in the state encoding bit string. We refer to these forms as state-level and bit-level state equations, respectively.

2.5.1.1 Example of State-Level State Equations

State-level state equations can be directly derived from an ASM chart. Referring to the example ASM chart in Fig. 2.8, we examine each state box to determine each state transition equation.

The left-hand side of each transition equation is the state at time $t+1$ and the right-hand side consists of an expression that has a product term for every path entering the state box with all products joined by the disjunctive (OR) operator. Consider state `s0` in the ASM chart; there are two paths possible to enter this state. One path occurs when the input zero is not asserted and the current state is `s0`. The other path to enter state `s0` occurs when `cnt_eq` is asserted and the current state is `s2`. Thus, the transition equation for state `s0` becomes

$$\texttt{s0}(t+1) = \overline{\texttt{zero}} \cdot \texttt{s0}(t) + \texttt{cnt\_eq} \cdot \texttt{s2}(t).$$

The state box for state `s1` has a single path entering it and that path occurs when the input `zero` is asserted and the current state is `s0`. Note that we do not include the information about the conditional output `set_busy`, which is also in the path since a separate output equation will be formed for this quantity. The product term formed for the transition into `s1` is the conjunction (AND) of `zero` and state `s0`:

$$\texttt{s1}(t+1) = \texttt{zero} \cdot \texttt{s0}(t).$$

The state box for `s2` also has a single entering path and this path originates directly from the state box of `s1`; thus the transition equation is very simple:

$$\texttt{s2}(t+1) = \texttt{s1}(t).$$

Next, we need to determine the output equations. The output equations for the unconditional outputs are very simple; they are equal to the state(s) in which they are asserted. If unconditional outputs are asserted in more than one state (not the case in our example), all the states in which they are asserted are joined together by the OR operation. The ASM chart in Fig. 2.8 has four unconditional outputs and the respective output equations are

$$\begin{aligned}
\texttt{ld\_cnt}(t) &= \texttt{s1}(t),\\
\texttt{addr\_sel}(t) &= \texttt{s2}(t),\\
\texttt{zero\_we}(t) &= \texttt{s2}(t),\\
\texttt{cnt\_en}(t) &= \texttt{s2}(t).
\end{aligned}$$

The final step is to develop the output equations for the conditional outputs. Conditional outputs depend on both the present state and the inputs. The conditional output `set_busy` is asserted only when the input `zero` is asserted and the present state is `s0`. Likewise, the conditional output `clr_busy` depends on the input `cnt_eq` and a present state value of `s2`. The two conditional output equations are

$$\begin{aligned}
\texttt{set\_busy}(t) &= \texttt{zero} \cdot \texttt{s0}(t),\\
\texttt{clr\_busy}(t) &= \texttt{cnt\_eq} \cdot \texttt{s2}(t).
\end{aligned}$$

2.5.1.2 Example of Bit-Level State Equations

Bit-level state equations depend on controller inputs and various output bits of the state holding devices rather than the entire state. This means that bit-level state equations can only be developed after state assignment has been performed. At the bit level, transition equations are formed for each bit in the state values rather than each state.

It is the case that if one-hot encoding is used, the bit-level state equations can be immediately determined from the state-level state equations. Assume that the one-hot state assignment {s0=001, s1=010, s2=100} is made with three flip-flops used as state holding elements. The current state of the controller, $\mathtt{s_i}$, is given as $\mathtt{s_i} = Q_C Q_B Q_A$ where each Q represents a flip-flop output. With this constraint, the bit-level state equations are the same as the state-level equations by substituting the flip-flop output value for each appropriate state value. This is possible for two reasons:

1. In one-hot encoding there are the same number of state-level and bit-level transition equations.
2. There is exactly one unique state bit asserted in each state of a one-hot encoded system.

In our example, Q_C replaces s0, Q_B replaces s1 and Q_A replaces s2, and the bit-level state equations are

$$\begin{aligned}
Q_C(t+1) &= \overline{\mathtt{zero}} \cdot Q_C(t) + \mathtt{cnt\_eq} \cdot Q_A(t),\\
Q_B(t+1) &= \mathtt{zero} \cdot Q_C(t),\\
Q_A(t+1) &= Q_B(t),\\
\mathtt{ld\_cnt}(t) &= \overline{Q_A}(t) Q_B(t) \overline{Q_C}(t),\\
\mathtt{addr\_sel}(t) &= Q_A(t) \overline{Q_B}(t) \overline{Q_C}(t),\\
\mathtt{zero\_we}(t) &= Q_A(t) \overline{Q_B}(t) \overline{Q_C}(t),\\
\mathtt{cnt\_en}(t) &= Q_A(t) \overline{Q_B}(t) \overline{Q_C}(t),\\
\mathtt{set\_busy}(t) &= \mathtt{zero} \cdot \overline{Q_A}(t) \overline{Q_B}(t) Q_C(t),\\
\mathtt{clr\_busy}(t) &= \mathtt{cnt\_eq} \cdot Q_A(t) \overline{Q_B}(t) \overline{Q_C}(t).
\end{aligned}$$

After the bit-level state equations are determined, excitation equations may be formed from the transition equations. Recall that an excitation equation is one that describes the input to the state holding device while a transition equation is one that describes the output of a state-holding device in the next state as a function of controller inputs and other present state bit values. The transformation from transition to excitation equations can be performed by using the state-holding device characteristic. In this book we are only considering D flip-flops as state-holding elements since they are the simplest to use and are the most predominant in programmable logic. The characteristic input/output relation for the D flip-flop is very simple:

$$Q(t+1) = D,$$

where $Q(t+1)$ represents the output of the D flip-flop after a clock edge and D is the synchronous input. If we use three D flip-flops, D_A, D_B, and D_C, to represent the present state of the circuit, and use the D flip-flop characteristic equation and the bit-level transition equations, then the excitation equations become

$$\begin{aligned} D_C &= \overline{\texttt{zero}} \cdot Q_C(t) + \texttt{cnt\_eq} \cdot Q_A(t), \\ D_B &= \texttt{zero} \cdot Q_C(t), \\ D_A &= Q_B(t). \end{aligned}$$

2.5.2 State Tables

In the previous section we showed how bit-level state equations are directly obtained from state-level state equations when one-hot encoding is used. If one-hot encoding is not used, a direct translation from state-level to bit-level state equations is not possible. One convenient way to obtain the bit-level state equations is to use a state table.

State tables are tabular descriptions of synchronous state machines. Because these types of circuits can be described as ASM charts or state diagrams which are both directed graphs, the state table can be considered an adjacency matrix representation of these directed graphs with extra information related to the external inputs and outputs. State tables are usually written with a number of rows equivalent to the number of distinct states the corresponding circuit possesses and with rows for each external input, each present-state bit, each next-state bit, and each circuit output bit. If a state encoding of {s0=00, s1=01, s2=10} is used for the example controller where the current state is $\texttt{s}_i = Q_A Q_B$, the corresponding state table is shown in Table 2.2.

TABLE 2.2: State Table Representation of the Example Controller with Minimal State Encoding

		PRESENT STATE		NEXT STATE			
zero	Cnt_eq	$Q_A(t)$	$Q_B(t)$	$Q_A(t+1)$	$Q_B(t+1)$	set_busy	clr_busy
0	X	0	0	0	0	0	0
1	X	0	0	0	1	1	0
x	X	0	1	1	0	0	0
x	0	1	0	1	0	0	0
x	1	1	0	0	0	0	1

Each row in the state table corresponds to a path in the ASM chart. As an example the first in the table indicates that if the present state is $Q_A(t)Q_B(t)$=s0=00 and the input zero is not asserted (i.e., zero=0), then the next state is $Q_A(t+1)Q_B(t+1)$=s0=00 and the conditional outputs are set_busy=0 and clr_busy=0. From the information in the state table, the bit-level transition equations can be derived. $Q_A(\mathrm{t}+1)$ is asserted as shown in rows 5 and 6 of the table and the corresponding transition equation is

$$Q_A(t+1) = Q_B(t) + \overline{\texttt{cnt\_eq}} \cdot Q_A(t)\overline{Q_B}(t).$$

Likewise, the state table indicates that $Q_B(t+1)$ is asserted in the third row when zero=1, $Q_A(t)$=0, and $Q_B(t)$=0. This yields the transition equation

$$Q_B(t+1) = \texttt{zero} \cdot \overline{Q_A}(t)\overline{Q_B}(t).$$

Using the state table, a similar analysis as that for determining the transition equations is performed for the conditional outputs yielding

$$\begin{aligned} \texttt{set\_busy}(t) &= \texttt{zero} \cdot \overline{Q_A}(t)\overline{Q_B}(t), \\ \texttt{clr\_busy}(t) &= \texttt{cnt\_eq} \cdot Q_A(t)\overline{Q_B}(t). \end{aligned}$$

The unconditional output equations are equal to the state values in which they are asserted as before and they are given as

$$\begin{aligned} \texttt{ld\_cnt}(t) &= \texttt{s1}(t) = \overline{Q_A}(t)Q_B(t), \\ \texttt{addr\_sel}(t) &= \texttt{s2}(t) = Q_A(t)\overline{Q_B}(t), \\ \texttt{zero\_we}(t) &= \texttt{s2}(t) = Q_A(t)\overline{Q_B}(t), \\ \texttt{cnt\_en}(t) &= \texttt{s2}(t) = Q_A(t)\overline{Q_B}(t). \end{aligned}$$

Using the D flip-flop characteristic, the corresponding excitation equations are

$$\begin{aligned} D_A &= Q_B(t) + \overline{\texttt{cnt\_eq}} \cdot Q_A(t)\overline{Q_B}(t), \\ D_B &= \texttt{zero} \cdot \overline{Q_A}(t)\overline{Q_B}(t). \end{aligned}$$

Using the excitation equations and the output equations, a logic diagram can be drawn and is shown in Fig. 2.11. Although basic logic gates are shown in Fig. 2.11, the actual logic generated depends on the target technology. For example if a device in the Altera Flex10K is used, four-input lookup tables (i.e., 16×1 memory circuits) are used instead of discrete logic gates.

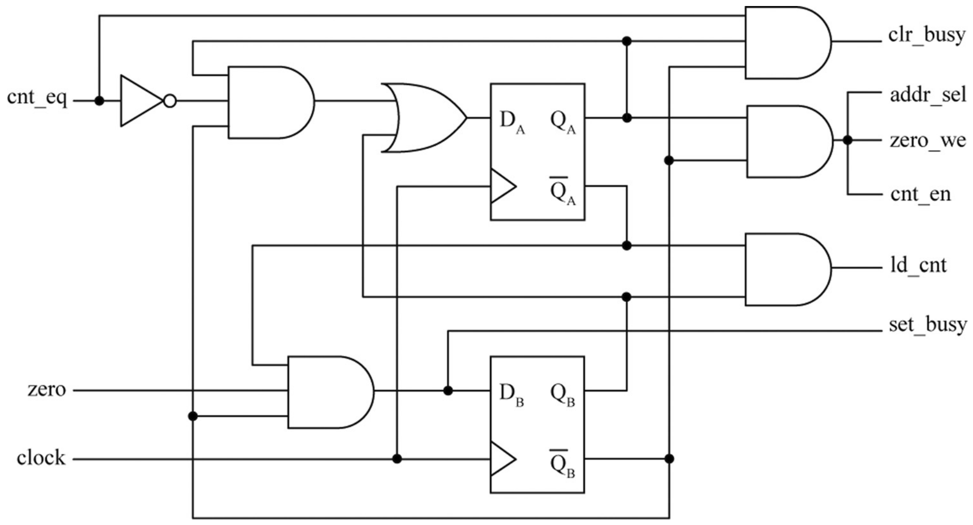


FIGURE 2.11: Logic diagram corresponding to bit-level state equations with minimal encoding

2.5.3 Controller Circuit Analysis

The inverse problem of logic synthesis is that of analysis. The analysis problem gives a logic diagram and requires the development of state equations. The procedure for determining a set of state equations for a given circuit diagram is as follows:

1. Determine the combinational logic expressions that represent the memory device synchronous input signals in terms of external inputs and present state variables.
2. Use the characteristic equations of the memory storage devices to find the next state equations in terms of the memory device inputs.
3. Substitute the expressions found in step 1 into those of step 2 yielding the next-state equations for each state bit.
4. Determine the output equations for each output bit by forming an equation based on the present state values and the external circuit inputs.

2.6 MEALY AND MOORE MACHINE CONVERSION

All synchronous sequential circuits may be implemented based on either Mealy or Moore machine models. Occasionally it is desirable to convert between Mealy and Moore machine models of synchronous state machines. As an example, it may be advantageous to realize a sequential circuit with outputs that are registered to avoid output signal changes in the middle of a clock period, or in other words, to ensure that the output is always valid except during state transitions. One way to do this is to convert a Mealy machine description to that of a Moore machine so that the circuit outputs are actually portions of the state encodings, or that they at least depend only on the state encodings. Conversely, transformation of a Moore machine to a Mealy machine can lead to circuits that require fewer states and hence memory storage elements that may result in a savings in area or component usage. Although the examples of Mealy to Moore conversion and vice versa are shown using state diagrams, the methodology for this conversion can also be accomplished directly with ASM charts.

2.6.1 Mealy to Moore Machine Conversion

The procedure for conversion from a Mealy to a Moore machine is to augment the state vectors with the output bits. All of the transition edges in a state diagram labeled with output bits are replaced with edges with the output bits omitted and the next states have the output bits appended to the original state vector encodings. As an example, consider the portion of the state transition diagram in Fig. 2.12 that is originally in Mealy form on the left and is transformed to Moore form on the right.

From the example in Fig. 2.12, it is seen that the transformation from a Mealy to a Moore state transition diagram is accomplished by splitting states with more than one incoming edge into separate states and then by preserving all destination states. The output values in the Mealy machine are changed into additional bits in the state encodings. In the example these new bits were appended to the end of the state vectors of the Mealy machine encodings but in general they may be added anywhere within the state vectors as long as each resulting bit

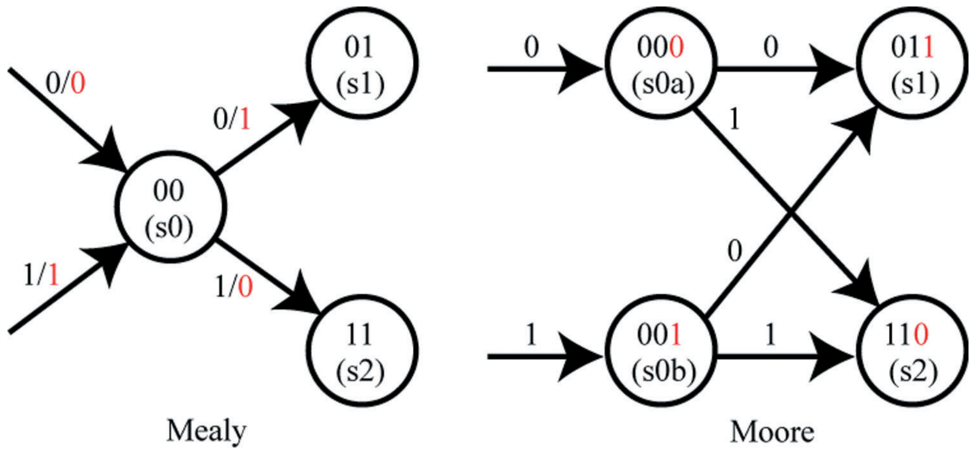


FIGURE 2.12: Portion of a Mealy state machine transformed to a Moore machine

string is unique. It is easy to see that the Moore equivalent of a Mealy machine will generally require more flip-flops since the number of states is increased and the state vectors grow in length.

2.6.2 Moore to Mealy Conversion

When a designer wishes to minimize the number of memory elements required and it is not necessary to have registered outputs of a synchronous sequential circuit, conversion from a Mealy to a Moore model of the circuit can be advantageous. Since sequential circuits realized from Mealy models generally require fewer states, the resulting circuit will usually require fewer state holding elements; however, the amount of combinational logic may also increase so a careful tradeoff should be performed if savings in area is the ultimate goal. The Moore to Mealy translation process is performed in two steps. The first step translates a Moore to Mealy machine and results in a state diagram with the same number of states. Clearly, there is no savings in terms of states after this translation is performed thus a second step should be performed that searches for equivalent states and collapses them into a single state. As an example of the first step, Fig. 2.13 contains a portion of a state diagram for a Moore machine on the left with the corresponding portion of the Mealy machine on the right. Essentially this transformation involves pushing the output values from the state vector to the outgoing edge of each vertex in the state diagram. The output values labeled A and B in Fig. 2.8 are those of the preceding states that each respective edge originates from.

2.6.3 State Machine Equivalence

Two state machines are said to be equivalent if for all possible inputs the same outputs are produced. In general, a given input/output behavior does not require a unique state machine.

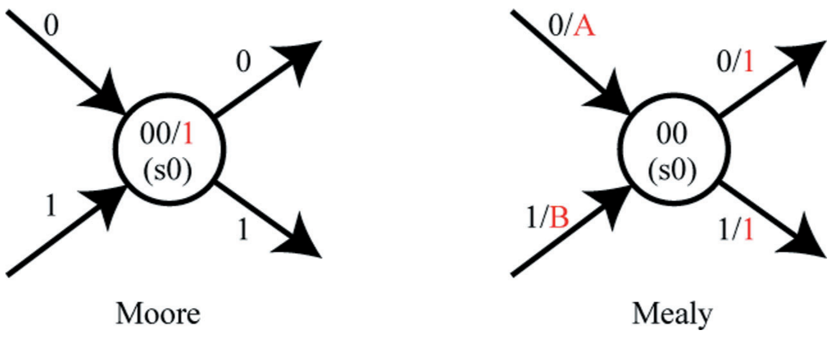


FIGURE 2.13: Portion of a Moore state machine transformed to a Mealy machine

Many different state machines can exist with the same input/output behavior with different numbers of internal states. As synchronous sequential circuit designers, we are interested in specifying state machines with as few states as possible to minimize the number of state holding elements.

During the process of transforming a Moore machine to a Mealy machine, it is always desirable to determine if the transformed Mealy machine can be replaced by another equivalent Mealy machine that is composed of fewer internal states. A methodology for minimizing the number of required states in a state machine is to generate the state table model of the circuit and then to search for equivalent states. Equivalent states are those for which the same input/output behavior occurs. When two equivalent states are found, one is discarded from the state table and all other occurrences of the discarded state are relabeled with the state that is retained.

The methodology for finding equivalent states in a Mealy machine is a classic problem in logic synthesis and design. One of the most common ways to find state equivalence involves the use of implication tables as described in [8] and many other texts on basic digital logic design and we refer the reader to these for more details.

2.7 VERILOG DESCRIPTIONS OF SYNCHRONOUS SEQUENTIAL CIRCUITS

The design of synchronous sequential circuits usually begins with a high-level description or notion of what the circuit is to accomplish in a behavioral sense. The first step in the design process is to transform this behavior into a state machine model. Given the model, the synchronous sequential circuit can be derived by a variety of means. In this text, we recommend the following procedure:

1. Transform the behavioral description of the circuit into an ASM chart. It may be easier for the designers who are more familiar with state diagrams to first generate another model such as a state diagram before finally deriving the ASM chart; however after experience with ASM chart generation has been obtained, most designers can generate the ASM chart as a first step.
2. Once the ASM chart has been developed, the next step is to generate a Verilog description of the ASM chart. Different styles of Verilog descriptions can be generated at this point, but all should synthesize into a controller with essentially the same timing characteristics.

Many designers accomplish this first step by drawing an ASM chart or a state diagram. Typically, the state encoding is unknown at this point and states are often given symbolic labels that may be letters of alphabets or descriptive words. After the first model of the state machine has been determined, an ASM chart is created if it has not already been produced. We strongly emphasize and suggest that ASM charts be generated before writing the Verilog code description. The "ASM first" approach is recommended because the rules of proper ASM chart production allow a designer to easily transform the chart into a Verilog description without performing low-level and tedious design work, such as state assignment and derivation of bit-level transition and excitation equations that are required for tabular or equation descriptions. The latter is to be avoided since simple changes in the circuit behavior result in more work to update the state table or state equations and then the resulting Verilog code.

It is very easy to fall into the trap of generating the initial ASM chart and corresponding Verilog description and to then modify only the Verilog code during design debugging. This should be avoided at all costs since the ASM chart provides a quick graphical depiction of the operation of the circuit and bugs can easily be detected by updating the ASM chart in each design iteration.

In the previous section on ASM charts we mentioned the rules of their proper construction as we described each symbol. We will group the rules together and restate them here for conciseness.

1. State-representation symbols
    a. Should only contain zero or more unconditional output expressions inside.
    b. May or may not have state names or state encodings appear outside but near the symbol.
    c. Must have one or more flow direction symbols (arrows) pointing to them.
    d. Must have a single flow direction symbol (arrow) exiting them.
2. Decision symbols
    a. Should contain input signals or expressions dependent upon input signals only.
    b. Must have a single flow direction symbol pointing to them that originates from a state-representation symbol.
    c. Must have two (single bit, basic decision symbol) or more (generalized decision symbol) outgoing flow direction symbols.
    d. Each outgoing flow direction symbol must be labeled with a unique and complete set of all possible values of the input signals or set of expressions contained within them.
    e. All outgoing signal flow arrows must point to another decision symbol, a conditional output symbol, or another state-representation symbol.

3. Conditional output symbols
    a. Must have exactly one incoming flow symbol in ASMs where generalized decision symbols are used.
    b. May have more than one incoming flow symbol if they all originate from decision symbols in ASM charts containing cascades of decision symbols.
    c. Must have exactly one outgoing flow symbol that points to another decision symbol or a state-representation symbol.
    d. The incoming flow signal must always originate from a decision symbol and never from a state-representation symbol.

Although the rules above allow conditional output symbols to be interspersed among cascades of decision symbols, we recommend that as a matter of practice, conditional output symbols always follow complete cascades of decision symbols, which implies that all outgoing flow arrows from conditional symbols will always point to state-representation symbols.

2.7.1 Example Verilog Descriptions

There are a variety of ways to describe a controller given an ASM chart description. One way to classify these different styles is through the number of `always` blocks used in the implementation. We will show three different Verilog listings that implement the example controller described by the ASM chart in Fig. 2.8. An easy way to classify these different approaches is to note that sections of the code represent the different blocks shown in the architecture models shown in Fig. 2.4. A fourth Verilog example is shown that synthesizes the Moore machine description of the controller modeled by the rightmost state diagram in Fig. 2.10.

2.7.2 Verilog Descriptions for the Mealy Machine Model of an Example Controller

In each of the three following Verilog modules we adhere to the coding guideline rule of using blocking assignments inside `always` blocks that synthesize combinational logic and nonblocking assignments for those always blocks that synthesize registered logic. Verilog `always` blocks that synthesize into registered logic are those that utilize the Verilog keywords `posedge` or `negedge` in the signal activation lists (i.e., the arguments of the `always` statement). It is possible to use blocking assignments in `always` blocks that produce registered logic but this can be tricky since the sequential order of the assignments matters and an incorrect order can produce the wrong results. For this reason, we promote and recommend using nonblocking assignments in `always` blocks that produce registered logic as this allows designers to not be concerned about the order of the assignment statements. The three examples that follow were constructed with a correct ordering of the assignments meaning that it would be safe to use blocking assignments in the following code listings as well as nonblocking.

In each of the following Verilog examples, we have included output ports for the current state values as indicated by the comments in the modules. These output ports are for debugging purposes only and are convenient during simulation. After the controller is verified to be working properly, these output ports should be removed prior to synthesis.

Fig. 2.14 contains a Verilog code listing that reflects the behavior of the ASM chart in Fig. 2.8.

The Verilog listing in Fig. 2.14 contains a single `always` block. When synthesized, the `always` block in Fig. 2.14 is responsible for generating the logic present in the top two

```
module ramfsm_ex1 (state, addr_sel, cnt_en, ld_cnt, zero_we, set_busy,
clr_busy, clk, reset, zero, cnt_eq);
   input        clk, reset, zero, cnt_eq;
   output [1:0] state;  //state output for debugging
   output       addr_sel, cnt_en, ld_cnt, zero_we;
   output       set_busy, clr_busy;

   reg    [1:0] state;

//  State Encoding Here
   parameter S0=2'b00, S1=2'b01, S2=2'b10;

// Register and Combinational Transition logic here
   always @(posedge clk or posedge reset)
       begin
           if (reset == 1'b1)
                state<=S0;
           else
                case (state)
                S0: if (zero == 1'b1) state <= S1;
                S1: state <= S2;
                S2: if (cnt_eq == 1'b1) state <= S0;
                default: state <= S0;
                endcase
        end
// Combinational output logic here
assign set_busy  = (state==S0 && zero==1'b1) ? 1'b1 : 1'b0;
assign ld_cnt    = (state==S1) ? 1'b1 : 1'b0;
assign addr_sel  = (state==S2) ? 1'b1 : 1'b0;
assign zero_we   = (state==S2) ? 1'b1 : 1'b0;
assign cnt_en    = (state==S2) ? 1'b1 : 1'b0;
assign clr_busy  = (state==S2 && cnt_eq==1'b1) ? 1'b1 : 1'b0;

endmodule
```

FIGURE 2.14: Verilog code listing of an example controller with a single `always` block

rectangles in Fig. 2.4 of the Mealy machine model. These are the memory storage elements and the combinational logic representing the excitation equations. The six concurrent statements that occur just before the `endmodule` are responsible for generating the output logic as shown in the bottommost rectangle in the Mealy model of Fig. 2.4. Referring back to the discussion on discrete-event simulation, we know that these are asynchronous statements that may be scheduled for simulation at any time an event occurs, meaning a signal on the right-hand side of the statement changes value. This in turn represents unregistered outputs in terms of synthesized combinational logic whose output changes shortly after any of the dependent signals on the right-hand side of the statements toggle. It is also interesting to note that the signals `ld_cnt`, `addr_sel`, `zero_we`, and `cnt_en` depend only on the present state. Hence, they will change value only after a short time that the state changes and are Moore-type outputs although they are not registered outputs. Although not shown in this example, it is possible to have outputs that depend only on the present state but with more complicated logic combining various bits of the present state in other controller examples. In this case output signals will all change shortly after a state change but not necessarily at the exact same time due to propagation through the output logic. The other two output signals, `set_busy` and `clr_busy`, depend not only upon the present state, but also the input signals `zero` and `cnt_eq`. This means that these output signals may change either shortly after a state change or shortly after one of these input signals toggles in value.

The next example, in Fig. 2.15, contains a Verilog code listing that has all the output signals and transition logic generated from within an `always` block. Since this logic is generated from within an `always` block that is not protected by a clock edge, the synthesized outputs are not registered and combinational logic is produced. The first `always` block is responsible for producing the memory elements represented by the middle block in the Mealy architecture diagram in Fig. 2.4 while the second `always` block produces the combinational logic in the top and bottom blocks of Fig. 2.4. Note that nonblocking (<=) assignments are used in the first `always` block while blocking (=) assignments are used in the second `always` block. Also note the inclusion of the default assignments for the outputs at the beginning of the combinational `always` block. These are very important to remember to include to prevent the outputs from being "stuck" in an asserted state. These assignments correspond to the nonasserted output values that are generally not shown in the ASM chart.

The next example code listing is shown in Fig. 2.16 and contains sections of Verilog code that each correspond to one of the blocks shown in the Mealy architecture model in Fig. 2.4. The first `always` block is responsible for synthesizing the memory elements in the center block, the second `always` block produces combinational logic that synthesizes the excitation logic block at the top of the diagram, and the bottom set of six continuous assignment statements synthesizes into the output combinational logic block.

```
module ramfsm_ex2  (pstate, addr_sel, cnt_en, ld_cnt, zero_we,
                set_busy, clr_busy, clk, reset, zero, cnt_eq);
   input        clk, reset, zero, cnt_eq;
   output [1:0] pstate;  //state output for debugging
   output       addr_sel, cnt_en, ld_cnt, zero_we;
   output       set_busy, clr_busy;
   reg          addr_sel, cnt_en, ld_cnt, zero_we, set_busy, clr_busy;
   reg    [1:0] pstate, nstate;

   //  State Encoding Here
   parameter S0=2'b00, S1=2'b01, S2=2'b10;

   // Register logic here
   always @(posedge clk or posedge reset)
       begin
           if (reset == 1'b1) pstate <= S0;
             else pstate <= nstate;
       end

// Combinational transition and output logic here
   always @(pstate)
      begin
         // We must include default values here
         // to avoid inferred latches
         set_busy = 1'b0;
         ld_cnt   = 1'b0;
         clr_busy = 1'b0;
         addr_sel = 1'b0;
         zero_we  = 1'b0;
         cnt_en   = 1'b0;
// Transition and output logic in case statement
         case (pstate)
                S0: if (zero == 1'b1)
                      begin
                       nstate = S1;
                       set_busy = 1'b1;
                      end
                    else nstate = S0;
                S1: begin
                       nstate = S2;
                       ld_cnt = 1'b1;
                    end
                S2: begin
                     if (cnt_eq == 1'b1)
                       begin
                        nstate = S0;
                        clr_busy = 1'b1;
                       end
                      else nstate = S2;
                     addr_sel = 1'b1;
                     zero_we = 1'b1;
                     cnt_en = 1'b1;
                    end
                default: nstate = S0;
          endcase
      end
endmodule
```

FIGURE 2.15: Verilog listing of an example controller with two `always` blocks

```
module ramfsm_ex3 (pstate, addr_sel, cnt_en, ld_cnt, zero_we,
               set_busy, clr_busy, clk, reset, zero, cnt_eq);
   input        clk, reset, zero, cnt_eq;
   output [1:0] pstate;  //state output for debugging
   output       addr_sel, cnt_en, ld_cnt, zero_we;
   output       set_busy, clr_busy;
   reg    [1:0] pstate, nstate;

   //  State Encoding Here
   parameter S0=2'b00, S1=2'b01, S2=2'b10;

   // Register logic here
   always @(posedge clk or posedge reset)
       begin
           if (reset == 1'b1) pstate<=S0;
             else pstate <= nstate;
       end

// Combinational transition logic here
   always @(pstate)
      begin
         case (pstate)
                S0: if (zero == 1'b1)
                        nstate = S1;
                    else
                        nstate = S0;
                S1: nstate = S2;
                S2: if (cnt_eq == 1'b1)
                        nstate = S0;
                    else
                        nstate = S2;
                default: nstate = S0;
          endcase
      end

// Combinational output logic here
assign set_busy  = (pstate==S0 && zero==1'b1) ? 1'b1 : 1'b0;
assign ld_cnt    = (pstate==S1) ? 1'b1 : 1'b0;
assign addr_sel  = (pstate==S2) ? 1'b1 : 1'b0;
assign zero_we   = (pstate==S2) ? 1'b1 : 1'b0;
assign cnt_en    = (pstate==S2) ? 1'b1 : 1'b0;
assign clr_busy  = (pstate==S2 && cnt_eq==1'b1) ? 1'b1 : 1'b0;

endmodule
```

FIGURE 2.16: Verilog listing of an example controller with two `always` blocks and concurrent assignments for output logic

2.7.3 Verilog Descriptions for the Moore Machine Model of an Example Controller

The example controller described by the ASM chart in Fig. 2.8 was transformed to a Moore machine represented by the rightmost state diagram in Fig. 2.10. As discussed previously, the timing characteristics for the outputs of a Moore type controller differ in that they change value only slightly after a clock event since the output signals are actually state bits.

```
module ramfsm_ex4 (state, addr_sel, cnt_en, ld_cnt, zero_we, set_busy,
clr_busy, clk, reset, zero, cnt_eq);
   input        clk, reset, zero, cnt_eq;
   output [6:0] state;  //state output for debugging
   output       addr_sel, cnt_en, ld_cnt, zero_we;
   output       set_busy, clr_busy;

   reg    [6:0] state;

//  State Encoding Here
   parameter A=7'b0000000, B=7'b0101000, C=7'b0000111, D=7'b1000111 ,
E=7'b0010000;

// Register and Combinational Transition logic here
   always @(posedge clk or posedge reset)
       begin
           if (reset == 1'b1)
                state<=A;
           else
                case (state)
                A: if (zero == 1'b1)
                       state <= B;
                   else
                       state <= A;
                B: state <= C;
                C: state <= D;
                D: if (cnt_eq == 1'b1)
                       state <= E;
                   else
                       state <= D;
                D: state <= E;
                E: state <= A;
                default: state <= A;
                endcase
        end

// Outputs are directly the state encoding signals
assign set_busy  = state[5];
assign clr_busy  = state[4];
assign ld_cnt    = state[3];
assign addr_sel  = state[2];
assign zero_we   = state[1];
assign cnt_en    = state[0];
endmodule
```

FIGURE 2.17: Verilog listing of an example controller with two `always` blocks and concurrent assignments for output logic

Fig. 2.17 contains the Verilog description of the Moore-type controller and consists of a single registered `always` block followed by a set of continuous assignments statements that simply map certain state bits to the various output signals. This type of controller requires more preliminary work to be performed since our preferred approach is to generate an ASM chart that usually contains outputs that are both Mealy-type (i.e., conditional outputs) and

Moore-type (unconditional outputs). After generation of this type of ASM chart, the chart must be converted into a chart with unconditional outputs only, and this involves performing state assignments that map to the output signals. In certain cases, where it is desired to have outputs that only change at clock edges, this is one way to generate such controllers.

2.8 SUMMARY

This chapter has provided a definition of sequential digital logic systems and focused specifically on synchronous or clocked sequential logic. We have described how this class of circuits is very useful for serving as controllers in a system containing a datapath. Models at various levels of abstraction were described including ASM charts, state diagrams, state equations, and state tables. Several design issues were discussed including conversions from Mealy- to Moore-type models and vice versa, timing aspects of output signals from Moore versus Mealy machine based controllers, and state assignment. We concluded this chapter with a discussion of various styles for describing a controller in synthesizable Verilog and emphasized the design approach of first creating an ASM chart and then translating it into a Verilog module.

References

[1] IEEE Std. 1364-2001, IEEE Standard Verilog® Hardware Description Language, p. 856.

[2] IEEE Std. 1364.1-2002, IEEE Standard for Verilog® Register Transfer Level Synthesis, p. 100.

[3] C. E. Cummings, "Nonblocking assignments in verilog synthesis, coding styles that kill!," *SNUG-2000*, San Jose, CA, 2000.

[4] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Tech. J.*, Vol. 34, No. 5, pp. 1045–1079, 1955.

[5] E. F. Moore, "Gedanken experiments on sequential machines," *Automata Studies*. Princeton, NJ: Princeton University Press, pp. 129–153, 1956.

[6] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.

[7] C. Maxfield, *Bebop to the Boolean Boogie*. Solana Beach, CA: Hightext Publications, 1995.

[8] M. M. Mano, *Digital Design*, 3rd edition, Upper Saddle River, NJ: Prentice Hall, 2002.

Biography

Robert B. Reese received the B.S. degree from Louisiana Tech University, Ruston, in 1979 and the M.S. and Ph.D. degrees from Texas A&M University, College Station, in 1982 and 1985, respectively, all in electrical engineering. He served as a Member of the Technical Staff of the Microelectronics and Computer Technology Corporation (MCC), Austin, TX, from 1985 to 1988. Since 1988, he has been with the Department of Electrical and Computer Engineering at Mississippi State University, Mississippi State, where he is an Associate Professor. Courses that he teaches include VLSI systems, Digital System design, and Microprocessors. His research interests include self-timed digital systems and computer architecture.

Mitchell A. Thornton received the BSEE degree from Oklahoma State University in 1985, the MSEE degree from the University of Texas in Arlington in 1990, and the MSCS in 1993 and Ph.D. in computer engineering in 1995 from Southern Methodist University in Dallas, Texas. His industrial experience includes full-time employment at E-Systems (now L-3 communications) in Greenville, Texas and the Cyrix Corporation in Richardson, Texas where he served in a variety of engineering positions between 1985 through 1992. From 1995 through 1999, he was a faculty member in the Department of Computer Systems Engineering at the University of Arkansas and from 1999 through 2002 in the Department of Electrical and Computer Engineering at Mississippi State University. Currently, he is a Professor of Computer Science and Engineering and, by courtesy, Electrical Engineering at Southern Methodist University. His research and teaching interests are in the general area of digital circuits and systems design with specific emphasis in EDA/CAD methods including asynchronous circuit and computer arithmetic circuit synthesis, formal verification/validation and simulation of digital systems, multiple-valued logic, and spectral techniques.