FPGA PROTOTYPING BY VERILOG EXAMPLES

This Page Intentionally Left Blank

FPGA PROTOTYPING BY VERILOG EXAMPLES

Xilinx Spartan™-3 Version

Pong P. Chu
Cleveland State University

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Chu, Pong P., 1959–
FPGA prototyping by Verilog examples / Pong P. Chu.
p. cm.
Includes index.
ISBN 978-0-470-18532-2 (cloth)
1. Field programmable gate arrays—Design and construction. 2. Prototypes, Engineering. 3.Verilog (Computer hardware description language) I. Title.
TK7895.G36C484 2008
621.39'5—dc22 2008003732

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

In memory of my father, Chia Chi Chu

This Page Intentionally Left Blank

CONTENTS

This Page Intentionally Left Blank

PREFACE

HDL (hardware description language) and *FPGA* (field-programmable gate array) devices allow designers to quickly develop and simulate a sophisticated digital circuit, realize it on a prototyping device, and verify operation of the physical implementation. As these technologies mature, they have become mainstream practice. We can now use a PC and an inexpensive FPGA prototyping board to construct a complex and sophisticated digital system. This book uses a "learning by doing" approach and illustrates the FPGA and HDL development and design process by a series of examples. A wide range of examples is included, from a simple gate-level circuit to an embedded system with an 8-bit soft-core microcontroller and customized I/O peripherals. All examples can be synthesized and physically tested on a prototyping board.

Focus and audience

Focus The main focus of this book is on the effective derivation of hardware, not the syntax of HDL. Instead of explaining every language construct, the book focuses on a small synthesizable subset and uses about a dozen code templates to provide the skeletons of various types of circuits. These templates are general and can easily be integrated to construct a large, complex system. Although this approach limits the "freedom" of syntactic expression, it will not prevent us from developing innovative hardware architecture. Because of the generality and flexibility of HDL, the same circuit can usually be described by a wide variety of language constructs and coding styles. Many of these codes are intended for modeling. They may lead to unnecessarily complex hardware implementation and sometimes cannot be synthesized at all. The template approach actually forces us to think more about hardware and develop a good coding practice for synthesis. Since we are

more interested in hardware, it is more beneficial to spend time on developing 10 different hardware architectures with the same code template rather than describing the same circuit with 10 different versions of codes.

There are two popular HDLs, *VHDL* and *Verilog*. Both languages are used widely and are IEEE standards. This book uses Verilog, and a separate book with a similar title uses VHDL. Despite the drastic syntactic differences in the two languages, their capabilities are very similar, particularly for our purposes. After we comprehend the design practice and coding methodology in one language, learning the other language is rather straightforward.

Although the book is intended for beginning designers, the examples follow strict design guidelines and prepare readers for future endeavors. The coding and design practice is "forward compatible," which means that:

- The same practice can be applied to large design in the future.
- The same practice can aid other system development tasks, including simulation, timing analysis, verification, and testing.
- The same practice can be applied to ASIC technology and different types of FPGA devices.
- The code can be accepted by synthesis software from different vendors.

In summary, the book is a hands-on, hardware-centric text that involves *minimal HDL overhead* and follows good design and coding practice to achieve *maximal forward comparability*.

Audience and perquisites The book contains three major parts: basic digital circuits, peripheral modules, and embedded microcontroller. The intended audience is students in an introductory or advanced digital system design course as well as practicing engineers who wish to learn FPGA- and HDL-based development. For the materials in the first two parts, readers need to have a basic knowledge of digital systems, usually a required course in electrical engineering and computer engineering curricula. For the materials in the third part, prior exposure to assembly language programming will be helpful.

Logistics

Although a major goal of this book is to teach readers to develop software-independent and device-neutral HDL codes, we have to choose a software package and a prototyping board to synthesize and implement the design examples. The synthesis software and FPGA devices from Xilinx, a leading manufacture in this area, are used in the book.

Software The synthesis software used in the book is the Web version of the Xilinx *ISE* package. The functionality of this version is similar to that of the full version but supports only a limited number of devices. Most introductory development boards use FPGA devices from the inexpensive Spartan-3 family. Since the Web version supports the Spartan-3 device, it fits our needs. The simulation software used in the book is the starter version of Mentor Graphics' *ModelSim XE III* package. It is a customized edition of *ModelSim*. Both software packages are free and can be downloaded from Xilinx's Web site.

FPGA prototyping board This book is prepared to be used with several entry-level FPGA prototyping boards manufactured by Digilent Inc., including the *Spartan-3 Starter*, *Nexys-2*, and *Basys* boards, all of which contain a Spartan-3/3E FPGA device and have

similar I/O peripherals. The design examples in the book are based on the Spartan-3 Starter board (or simply the *S3 board*), but most of them can be used directly on other boards as well. The applicability of the HDL codes is summarized below.

- **Spartan-3 Starter (S3) board**. The S3 board contains all the peripherals and no additional accessory module is needed. All HDL codes and discussions can be applied to this board directly.
- **Nexys-2 board**. The Nexys-2 board is a newer board, which contains a larger FPGA device and a larger memory chip. Its peripherals are similar to those on the S3 board. There are two differences. First, the "color depth" of its VGA interface is expanded from 3 bits to 8 bits. Thus, the output of the VGA interface circuits discussed in Chapters 13 and 14 needs to be modified accordingly. Second, the Nexys-2 board contains a more sophisticated external memory device. Although the device can be configured as an asynchronous SRAM, the timing characteristics are different from those of the S3 board's memory device, and thus the HDL codes for the memory controller in Chapter 11 cannot be used directly. However, the same design principle can be applied to construct a new controller.
- **Basys board**. The Basys board is a simpler board. It lacks the RS-232 connector. To implement the UART module and the serial interface discussed in Chapter 8, we need Digilent's *RS-232 converter peripheral module*. The Basys board has no external memory devices, and thus the discussion of the memory controller in Chapter 11 is not applicable.
- **Other FPGA boards**. Most peripherals discussed in this book are de facto industrial standards, and the corresponding HDL codes can be used as long as a board provides proper analog interface circuits and connectors. Except for the Xilinx-specific portions, the codes can be applied to the boards based on the FPGA devices from other manufacturers as well.

PC Accessories The design examples include interfaces to several PC peripheral devices. A keyboard, a mouse, and a VGA monitor are required for the respective modules, and a "straight-through" serial cable (the most commonly used type) is required for the UART module. These accessories are widely available and can probably be obtained from an old PC.

Book organization

The book is divided into three major parts. Part I introduces the elementary HDL constructs and their hardware counterparts, and demonstrates the construction of a basic digital circuit with these constructs. It consists of six chapters:

- Chapter 1 describes the skeleton of an HDL program, basic language syntax, and logical operators. Gate-level combinational circuits are derived with these language constructs.
- Chapter 2 provides an overview of an FPGA device, prototyping board, and development flow. The development process is demonstrated by a tutorial on Xilinx ISE synthesis software and a tutorial on Mentor Graphics ModelSim simulation software.
- Chapter 3 introduces HDL's relational and arithmetic operators and routing constructs. These correspond to medium-sized components, such as comparators, adders, and multiplexers. Module-level combinational circuits are derived with these language constructs.

- Chapter 4 covers the codes for memory elements and the construction of "regular" sequential circuits, such as counters and shift registers, in which the state transitions exhibit a regular pattern.
- Chapter 5 discusses the construction of a finite state machine (FSM), which is a sequential circuit whose state transitions do not exhibit a simple, regular pattern.
- Chapter 6 presents the construction of an FSM with data path (FSMD). The FSMD is used to implement register transfer (RT) methodology, in which the system operation is described by data transfers and manipulations among registers.
- Chapter 7 discusses several more advanced topics on language constructs and coding techniques and introduces the development of more sophisticated testbenches. This chapter can be skipped without affecting the remaining chapters.

Part II applies the techniques from Part I to design an array of peripheral modules for the prototyping board. Each chapter covers the development, implementation, and verification of an individual peripheral. These modules can be incorporated to a larger project. Part II consists of seven chapters:

- Chapter 8 discusses the design of a universal asynchronous receiver and transmitter (UART), which provides a serial link to receive and transmit data via the prototyping board's RS-232 port.
- Chapter 9 covers the design of a keyboard interface, which reads scan code from a keyboard. The keyboard is connected via the prototyping board's PS2 port.
- Chapter 10 covers the design of a mouse interface, which obtains the button and movement information from a mouse. The mouse is also connected via the prototyping board's PS2 port.
- Chapter 11 discusses the implementation and timing issues of a memory controller. The controller is used to read data from and write data to the two static random access memory (SRAM) devices on the S3 board.
- Chapter 12 discusses the inference and application of Spartan-3 device-specific components. The focus is on the FPGA's internal memory blocks.
- Chapter 13 presents the design and implementation of a video controller. The discussion covers the generation of video synchronization signals and shows the construction of simple bit- and object-mapped graphical interfaces. The monitor is connected to the prototyping board's VGA port.
- Chapter 14 continues development of the video controller. The discussion illustrates the construction of text interface and general tile-mapped scheme.

Part III introduces an FPGA-based soft-core microcontroller, known as *PicoBlaze*, and demonstrates the integration of a general-purpose processor and customized circuit. It includes four chapters:

- Chapter 15 provides an overview of the organization and instruction set of PicoBlaze.
- Chapter 16 introduces the basic assembly programming and provides an overview of the development process.
- Chapter 17 discusses PicoBlaze's I/O feature and illustrates the procedure to derive customized circuits to interface other I/O peripherals.
- Chapter 18 discusses PicoBlaze's interrupt capability and demonstrates the construction of a customized interrupt-handling circuit.

In addition to regular chapters, the appendix summarizes and lists all code templates.

Special marks$^{Xilinx\ specific}$ We use two special paragraph marks in the book: one for a *Xilinx-specific feature* and one for *Verilog-1995 constructs*. While the examples

described in the book are implemented on a Xilinx-based prototyping board and the codes are synthesized by Xilinx ISE software, we try to make the HDL codes as device independent and software neutral as possible. Most discussions and codes can be applied to different target devices and different synthesis software as well. However, certain codes or device features are unique to Xilinx ISE software or Spartan-3 FPGA devices. We use the *Xilinx specific* superscript, as in the heading of this section, to indicate that the discussion in the corresponding section or chapter is unique to Xilinx.

Similarly, we use marginal notes, as shown on the outer edge, to indicate that the discussion in a paragraph is unique to Xilinx. This note indicates that the code or design is no longer portable and needs to be revised when a different software package or target device is used.

Xilinx specific

The Verilog language was first ratified in 1995 (referred to as Verilog-1995) and then revised in 2001 (referred to as Verilog-2001). Many useful enhancements are added in the revised version. We use Verilog-2001 in this book. If a language construct differs in the two versions, we describe the old syntax briefly in a separate paragraph and use a marginal note, as shown on the outer edge, for this type of discussion. It indicates "for your information" and the materials are included to help readers understand the older Verilog codes.

FYI

Instructional use

The book can be a good companion text for an introductory digital systems course or an advanced project-oriented course. In an introductory digital systems course, the book supplies the lab portion of the curriculum. The chapters in Part I basically follow the sequence of a typical curriculum and can be presented along with regular lectures. One or two peripheral modules can be selected as case studies, and corresponding experiments can be used as term projects.

In an advanced project-oriented course, the book provides a base for independent projects. The materials in Part I should be treated as an overview or refresher, which provides a general background on HDL, synthesis, and FPGA boards. Some modules in Part II can be used to demonstrate the design of more complex circuits. These modules can also be considered as building blocks (i.e., IPs) or subsystems to be integrated into final projects. The PicoBlaze microcontroller discussed in Part III can be used as a general-purpose processor if an embedded-system type of project is desired.

Companion Web site

An accompanying Web site (http://academic.csuohio.edu/chu_p/rtl) provides additional information, including the following materials:

- Errata
- Code templates
- HDL code listing and relevant files
- Links to synthesis and simulation software
- Links to referenced materials
- Additional project ideas

Errata The book is self-prepared, which means that the author has produced all aspects of the text, including illustrations, tables, code listings, indexing, and formatting. As errors

are always bound to happen, the accompanying Web site provides an updated errata sheet and a place to report errors.

P. P. CHU

Cleveland, Ohio
January 2008

ACKNOWLEDGMENTS

The author would like to express his gratitude to Professor George L. Kramerich for his encouragement and help.

The author also thanks John Wiley & Sons, Inc. for giving permission to use Figures 3.1, 3.2, 4.2, 4.10, 4.11, 6.5, and 7.2 from my text *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, and Xilinx, Inc. for giving permission to use Figures 2.3 and 9.3 from the *Spartan-3 Starter Kit Board User Guide*.

All trademarks used or referred to in this book are the property of their respective owners.

P. P. Chu

This Page Intentionally Left Blank

PART I

BASIC DIGITAL CIRCUITS

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

1.1 INTRODUCTION

Verilog is a hardware description language. It was developed in the mid-1980s and later transferred to the IEEE (Institute of Electrical and Electronics Engineers). The language is formally defined by IEEE Standard 1364. The standard was ratified in 1995 (referred to as Verilog-1995) and revised in 2001 (referred to as Verilog-2001). Many useful enhancements are added in the revised version. We use Verilog-2001 in this book.

Verilog is intended for describing and modeling a digital system at various levels and is an extremely complex language. The focus of this book is on hardware design rather than the language. Instead of covering every aspect of Verilog, we introduce the key Verilog synthesis constructs by examining a collection of examples. Several advanced topics are examined further in Chapter 7 and detailed Verilog coverage may be explored through the sources listed in the bibliographic section at the end of the chapter.

Although the syntax of Verilog is somewhat like that of the C language, its semantics (i.e., "meaning") is based on concurrent hardware operation and is totally different from the sequential execution of C. The subtlety of some language constructs and certain inherent non-deterministic behavior of Verilog can lead to difficult-to-detect errors and introduce a discrepancy between simulation and synthesis. The coding of this book follows a "better-safe-than-buggy" philosophy. Instead of writing quick and short codes, the focus is on style and constructs that are clear and synthesizable and can accurately describe the desired hardware.

Table 1.1 Truth table of 1-bit equality comparator

input $i0\ i1$	**output** eq
0 0	1
0 1	0
1 0	0
1 1	1

In this chapter, we use a simple comparator to illustrate the skeleton of a Verilog program. The description uses only logic operators and represents a gate-level combinational circuit, which is composed of simple logic gates. In Chapter 3, we cover the remaining Verilog operators and constructs and examine the register-transfer-level combinational circuits, which are composed of intermediate-sized components, such as adders, comparators, and multiplexers.

1.2 GENERAL DESCRIPTION

Consider a 1-bit equality comparator with two inputs, `i0` and `i1`, and an output, `eq`. The `eq` signal is asserted when `i0` and `i1` are equal. The truth table of this circuit is shown in Table 1.1.

Assume that we want to use basic logic gates, which include *not*, *and*, *or*, and *xor cells*, to implement the circuit. One way to describe the circuit is to use a sum-of-products format. The logic expression is

$$eq = i0 \cdot i1 + i0' \cdot i1'$$

One possible Verilog code is shown in Listing 1.1. We examine the language constructs and statements of this code in the following subsections.

Listing 1.1 Gate-level implementation of a 1-bit comparator

```
module eq1
    // I/O ports
    (
     input wire i0, i1,
     output wire eq
    );

    // signal declaration
    wire p0, p1;

    // body
    // sum of two product terms
    assign eq = p0 | p1;
    // product terms
    assign p0 = ~i0 & ~i1;
    assign p1 = i0 & i1;

endmodule
```

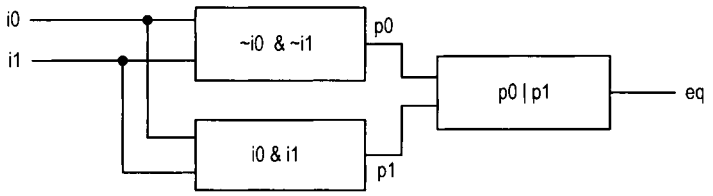


Figure 1.1 Graphical representation of a comparator program.

The best way to understand an HDL (hardware description language) program is to think in terms of hardware circuits. This program consists of three portions. The I/O port portion describes the input and output ports of this circuit, which are `i0` and `i1`, and `eq`, respectively. The signal declaration portion specifies the internal connecting signals, which are `p0` and `p1`. The body portion describes the internal organization of the circuit. There are three continuous assignments in this code. Each can be thought of as a circuit part that performs certain simple logical operations. We examine the language constructs and statements of this code in the next section.

The graphical representation of this program is shown in Figure 1.1. The three continuous assignments constitute the three circuit parts. The connections among these parts are specified implicitly by the signal and port names.

1.3 BASIC LEXICAL ELEMENTS AND DATA TYPES

1.3.1 Lexical elements

Identifier An *identifier* gives a unique name to an object, such as `eq1`, `i0`, or `p0`. It is composed of letters, digits, the underscore character (_), and the dollar sign ($). $ is usually used with a system task or function.

The first character of an identifier must be a letter or underscore. It is a good practice to give an object a descriptive name. For example, `mem_addr_en` is more meaningful than `mae` for a memory address enable signal.

Verilog is a *case-sensitive language*. Thus, `data_bus`, `Data_bus`, and `DATA_BUS` refer to three different objects. To avoid confusion, we should refrain from using the case to create different identifiers.

Keywords *Keywords* are predefined identifiers that are used to describe language constructs. In this book, we use boldface type for Verilog keywords, such as **module** and **wire** in Listing 1.1.

White space White space, which includes the space, tab, and newline characters, is used to separate identifiers and can be used freely in the Verilog code. We can use proper white spaces to format the code and make it more readable.

Comments A *comment* is just for documentation purposes and will be ignored by software. Verilog has two forms of comments. A one-line comment starts with //, as in

```
// This is a comment.
```

A multiple-line comment is encapsulated between /* and */, as in

```
/* This is comment line 1.
   This is comment line 2.
   This is comment line 3.  */
```

In this book, we use italic type for comments, as in the examples above.

1.4 DATA TYPES

1.4.1 Four-value system

Four basic values are used in most data types:

- 0: for "logic 0", or a false condition
- 1: for "logic 1", or a true condition
- z: for the high-impedance state
- x: for an unknown value

The z value corresponds to the output of a tri-state buffer. The x value is usually used in modeling and simulation, representing a value that is not 0, 1, or z, such as an uninitialized input or output conflict.

1.4.2 Data type groups

Verilog has two main groups of data types: *net* and *variable*.

Net group The data types in the net group represent the physical connections between hardware components. They are used as the outputs of *continuous assignments* and as the connection signals between different modules. The most commonly used data type in this group is **wire**. As the name indicates, it represents a connecting wire.

The **wire** data type represents a 1-bit signal, as in

```
wire p0, p1;    // two 1-bit signals
```

When a collection of signals is grouped into a bus, we can represent it using a one-dimensional array (vector), as in

```
wire [7:0] data1, data2; // 8-bit data
wire [31:0] addr;        // 32-bit address
wire [0:7] revers_data;  // ascending index should be avoided
```

While the index range can be either descending (as in [7:0]) or ascending (as in [0:7]), the former is preferred since the leftmost position (i.e., 7) corresponds to the MSB of a binary number.

A two-dimensional array is sometimes needed to represent a memory. For example, a 32-by-4 memory (i.e., a memory has 32 words and each word is 4 bits wide) can be represented as

```
wire [3:0] mem1 [31:0]; // 32-by-4 memory
```

The other data types in the net group imply certain logical behavior or functionality, such as **wand** (for wired-and connection) and **supply0** (for circuit ground connection). We don't use these data types in this book. Verilog-2001 also allows the **signed** data type and this issue is discussed in Section 7.3.

Variable group The data types in the variable group represent abstract storage in behavioral modeling and are used in the outputs of *procedural assignments*. There are five data types in this group: **reg**, **integer**, **real**, **time**, and **realtime**. The most commonly used data type in this group is **reg** and it can be synthesized. The inferred circuit may or *may not* contain physical storage components. The last three data types can only be used in modeling and simulation, and the use of the **integer** data type is discussed in Section 7.3.

In Verilog-1995, the variable group is known as the *register group*. Since this term is the same for a physical hardware register (i.e., a collection of flip-flops), it is changed in the Verilog-2001 documentation to avoid confusion. In this book, we use the term *variable* for the data type, and use the term *register* for the physical register circuit. **FYI**

1.4.3 Number representation

An integer constant in Verilog can be represented in various formats. Its general form is

```
[sign][size]'[base][value]
```

The `[base]` term specifies the base of the number, which can be the following:

- `b` or `B`: binary
- `o` or `O`: octal
- `h` or `H`: hexadecimal
- `d` or `D`: decimal

The `[value]` term specifies the value of the number in the corresponding base. The underline character (_) can be included for clarity.

The `[size]` term specifies the number of bits in a number. It is optional. The number is known as a *sized number* when a `[size]` term exists and is known as an *unsized number* otherwise.

Sized number A sized number specifies the number of bits explicitly. If the size of the value is smaller than the `[size]` term specified, zeros are padded in front to extend the number, except in several special cases. The `z` or `x` value is padded if the MSB of the value is `z` or `x`, and the MSB is padded if the **signed** data type is used. Several sized number examples are shown in the top portion of Table 1.2.

Unsized number An unsized number omits the `[size]` term. Its actual size depends on the host computer but must be at least 32 bits. The `'[base]` term can also be omitted if the number is in decimal format. Assume that 32 bits are used in the host machine. Several unsized number examples are shown in the bottom portion of Table 1.2.

1.4.4 Operators

Verilog has about two dozen operators. For the gate-level description, we need only the following bitwise operators: ~ (not), & (and), | (or), and ^ (xor). These operators infer basic gate-level cells. Other operators are discussed in Section 3.2.

1.5 PROGRAM SKELETON

As its name indicates, HDL is used to describe hardware. When we develop or examine a Verilog code, it is much easier to comprehend if we think in terms of "hardware organization"

Table 1.2 Examples of sized and unsized numbers

number	stored value	comment
5'b11010	11010	
5'b11_010	11010	_ ignored
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	00000000000000000000000000011010	extended to 32 bits
'hee	00000000000000000000000011101110	extended to 32 bits
1	00000000000000000000000000000001	extended to 32 bits
-1	11111111111111111111111111111111	extended to 32 bits

rather than "sequential algorithm." Most Verilog codes in this book follow the basic skeleton shown in Listing 1.1. It consists of three portions: I/O port declaration, signal declaration, and module body.

1.5.1 Port declaration

The module declaration and port declaration of Listing 1.1 are

```
module eq1
   (
    input wire i0, i1,
    output wire eq
   );
```

The I/O declaration specifies the modes, data types, and names of the module's I/O ports. The simplified syntax is

```
module [module_name]
   (
    [mode] [data_type] [port_names],
    [mode] [data_type] [port_names],
    . . .
    [mode] [data_type] [port_names]
   );
```

The [mode] term can be **input**, **output**, or **inout**, which represent the input, output, or bidirectional port, respectively. Note that there is no comma in the last declaration. The [data_type] term can be omitted if it is **wire**.

FYI

Verilog-1995 port declaration In Verilog-1995, port names, modes, and data types are declared separately. For example, the preceding port declaration becomes

```
module eq1 (i0, i1, eq);  // only port names in brackets
   // declare mode
   input i0, i1;
   output eq;
   // declare data type
   wire i0, i1;
   wire eq;
```

We do not use this format in this book.

1.5.2 Program body

Unlike a program in the C language, in which the statements are executed sequentially, the program body of a synthesizable Verilog module can be thought of as a collection of circuit parts. These parts are operated in parallel and executed concurrently. There are several ways to describe a part:

- Continuous assignment
- "Always block"
- Module instantiation

The first way to describe a circuit part is by using a *continuous assignment*. It is useful for simple combinational circuits. Its simplified syntax is

```
assign [signal_name] = [expression];
```

Each continuous assignment can be thought as a circuit part. The signal on the left-hand side is the output and the signals used in the right-hand-side expression are the inputs. The expression describes the function of this circuit. For example, consider the statement

```
assign eq = p0 | p1;
```

It is a circuit that performs the or operation. When p0 or p1 changes its value, this statement is activated and the expression is evaluated. The new value is assigned to eq after the propagation delay. There are three continuous assignments in Listing 1.1 and they correspond to the three circuit parts shown in Figure 1.1. Since the assignments correspond to the circuit parts, the order of these statements does not matter.

The second way to describe a circuit part is by using an *always block*. More abstract *procedural assignments* are used inside the always block and thus it can be used to describe more complex circuit operation. The always block is discussed in Section 3.3.

The third way to describe a circuit part is by using *module instantiation*. Instantiation creates an instance of another module and allows us to incorporate predesigned modules as subsystems of the current module. Instantiation is discussed in Section 1.6.

1.5.3 Signal declaration

The declaration portion specifies the internal signals and parameters used in the module. The internal signals can be thought of as the interconnecting wires between the circuit parts, as shown in Figure 1.1.

The simplified syntax of signal declaration is

```
[data_type] [port_names];
```

Two internal signals are declared in Listing 1.1:

```
wire p0, p1;
```

FYI

Implicit net In Verilog, an identifier does not need to be declared explicitly. If a declaration is omitted, it is assumed to be an *implicit net*. The default data type is **wire**. We can remove the explicit declarations in Listing 1.1 and the simplified code is shown in Listing 1.2.

Listing 1.2 Code with implicit net

```
module eq1_implicit
   (
    input i0, i1,  // no data type declaration
    output eq
   );

// no internal signal declaration

// product terms must be placed in front
assign p0 = ~i0 & ~i1;    //implicit declaration
assign p1 = i0 & i1;      //implicit declaration
// sum of two product terms
assign eq = p0 | p1;

endmodule
```

Although the code is more compact, it may introduce subtle errors of misspelled identifiers. For clarity and documentation, we always use explicit declarations in this book.

1.5.4 Another example

We can expand the comparator to 2-bit inputs. Let the input be a and b and the output be aeqb. The aeqb signal is asserted when both bits of a and b are equal. The code is shown in Listing 1.3.

Listing 1.3 Gate-level implementation of a 2-bit comparator

```
module eq2_sop
  (
  input wire[1:0] a, b,
  output wire aeqb
  );

  // internal signal declaration
  wire p0, p1, p2, p3;

  // sum of product terms
  assign aeqb = p0 | p1 | p2 | p3;
  // product terms
  assign p0 = (~a[1] & ~b[1]) & (~a[0] & ~b[0]);
  assign p1 = (~a[1] & ~b[1]) & (a[0] & b[0]);
  assign p2 = (a[1] & b[1]) & (~a[0] & ~b[0]);
  assign p3 = (a[1] & b[1]) & (a[0] & b[0]);

endmodule
```

The a and b ports are now declared as a two-element array. Derivation of the architecture body is similar to that of the 1-bit comparator. The p0, p1, p2, and p3 signals represent

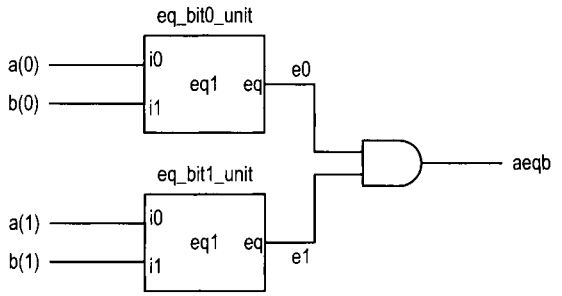


Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

the results of the four product terms, and the final result, aeqb, is the logic expression in sum-of-products format.

1.6 STRUCTURAL DESCRIPTION

A digital system is frequently composed of several smaller subsystems. This allows us to build a large system from simpler or predesigned components. Verilog provides a mechanism, known as *module instantiation*, to perform this task. This type of code is called *structural description*.

An alternative to the design of the 2-bit comparator of Section 1.5.4 is to utilize previously constructed 1-bit comparators as the building blocks. The diagram is shown in Figure 1.2, in which two 1-bit comparators are used to check the two individual bits and their results are fed to an and cell. The aeqb signal is asserted only when both bits are equal. The corresponding code is shown in Listing 1.4.

Listing 1.4 Structural description of a 2-bit comparator

```
module eq2
   (
    input  wire[1:0] a, b,
    output wire aeqb
   );

   // internal signal declaration
   wire e0, e1;

   // body
   // instantiate two 1-bit comparators
   eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
   eq1 eq_bit1_unit (.eq(e1), .i0(a[1]), .i1(b[1]));

   // a and b are equal if individual bits are equal
   assign aeqb = e0 & e1;

endmodule
```

The code includes two module instantiation statements. The simplified syntax of module instantiation is

```
[module_name] [instance_name]
  (
   .[port_name]([signal_name]),
   .[port_name]([signal_name]),
   . . .
   );
```

The first portion of the statement specifies which component is used. The `[module_name]` term indicates the name of the module and the `[instance_name]` term gives a unique id for an instance. The second portion is port connection, which indicates the connections between the I/O ports of an instantiated module (the lower-level module) and the external signals used in the current module (the higher-level module). This form of mapping is known as *connection by name*. The order of the port-name and signal-name pairs does not matter.

In Listing 1.4, the first component instantiation statement is

```
eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
```

The `eq1` is the module name defined in Listing 1.1. The port mapping reflects the connections shown in Figure 1.2. The component instantiation statement represents a circuit that is encompassed in a "black box" whose function is defined in another module.

This example demonstrates the close relationship between a block diagram and code. The code is essentially a textual description of a schematic. Although it is a clumsy way for humans to comprehend the diagram, it puts all representations into a single HDL framework.

Xilinx specific The Xilinx ISE package includes a simple schematic editor utility that can perform schematic capture in graphic format and then convert the diagram into an HDL structural description.

FYI

Connection by ordered list An alternative scheme to associate the ports and external signals is *connection by ordered list* (sometimes also known as *connection by position*). In this scheme, the port names of the lower-level module are omitted and the signals of the higher-level module are listed in the same order as the lower-level module's port declaration. With this scheme, the two module instantiation statements in Listing 1.4 can be rewritten as

```
eq1 eq_bit0_unit (a[0], b[0], e0);
eq1 eq_bit1_unit (a[1], b[1], e1);
```

Although this scheme makes the code more compact, it is error prone, especially for a module with many I/O ports. For example, if we modify the code of the lower-level module and switch the order of two ports in the port declaration, all the instantiated modules need to be corrected as well. If this is done accidentally during code editing, the altered port order may be left undetected during synthesis and leads to difficult-to-find bugs. We always use the connection-by-name scheme in this book.

FYI

Verilog primitive Verilog includes a set of predefined *primitives* that can be instantiated as modules. These primitives correspond to simple gate-level function blocks, such as the *and*, *or*, and *not cells*. For example, the `eq1` circuit can be implemented by using simple cells, as shown in Figure 1.3. The corresponding primitive-based code is shown in Listing 1.5.

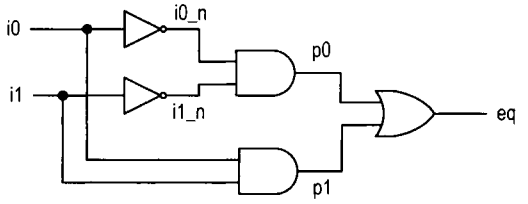


Figure 1.3 Low-level diagram of a 1-bit comparator.

Listing 1.5 Implementation with Verilog primitive

```
module eq1_primitive
  (
   input wire i0, i1,
   output wire eq
  );

  // internal signal declaration
  wire i0_n, i1_n, p0, p1;

  //primitive gate instantiations
  not unit1 (i0_n, i0);          // i0_n = ~i0;
  not unit2 (i1_n, i1);          // i1_n = ~i1;
  and unit3 (p0, i0_n, i1_n); // p0 = i0_n & i1_n;
  and unit4 (p1, i0, i1);        // p1 = i0 & i1;
  or unit5 (eq, p0, p1);         // eq = p0 | p1;

endmodule
```

This form of code is very tedious and can easily be replaced with simple bitwise logical operators. We do not use primitives in this book.

In addition to the predefined primitives, we can also define customized primitives, known as *user-defined primitives* (UDPs). For example, we can define a 1-bit comparator circuit in a UDP, as shown in Listing 1.6.

Listing 1.6 UDP of a 1-bit comparator

```
primitive eq1_udp(eq, i0, i1);
   output eq;
   input i0, i1;

  table
  // i0 i1 : eq
     0  0  : 1;
     0  1  : 0;
     1  0  : 0;
     1  1  : 1;
  endtable

endprimitive
```

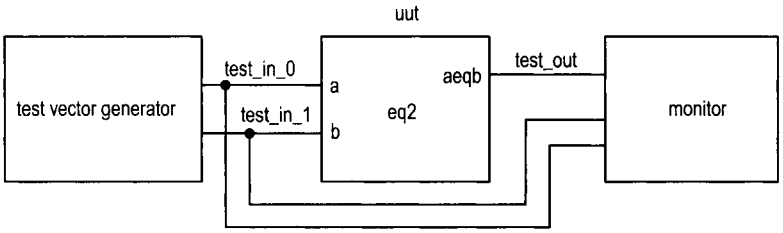


Figure 1.4 Testbench for a 2-bit comparator.

A UPD is essentially a table-based description of a circuit. The same table can also be described by a case statement (discussed in Section 3.5). We use the latter approach and do not use UDPs in this book.

1.7 TESTBENCH

After code is developed, it can be *simulated* in a host computer to verify the correctness of the circuit operation and can be *synthesized* to a physical device. Simulation is usually performed within the same HDL framework. We create a special program, known as a *test-bench*, to mimic a physical lab bench. The sketch of a 2-bit comparator testbench is shown in Figure 1.4. The `uut` block is the unit under test, the `test vector generator` block generates testing input patterns, and the `monitor` block examines the output responses. A simple testbench for the 2-bit comparator is shown in Listing 1.7.

Listing 1.7 Testbench for a 2-bit comparator

```
// The 'timescale directive specifies that
// the simulation time unit is 1 ns  and
// the simulation timestep is 10 ps
`timescale 1 ns/10 ps

module eq2_testbench;
   // signal declaration
   reg  [1:0] test_in0, test_in1;
   wire  test_out;

   // instantiate the circuit under test
   eq2 uut
      (.a(test_in0), .b(test_in1), .aeqb(test_out));

   //  test vector generator
   initial
   begin
      // test vector 1
      test_in0 = 2'b00;
      test_in1 = 2'b00;
      # 200;
      // test vector 2
      test_in0 = 2'b01;
      test_in1 = 2'b00;
```

```
        # 200;
        // test vector 3
        test_in0 = 2'b01;
        test_in1 = 2'b11;
        # 200;
        // test vector 4
        test_in0 = 2'b10;
        test_in1 = 2'b10;
        # 200;
        // test vector 5
        test_in0 = 2'b10;
        test_in1 = 2'b00;
        # 200;
        // test vector 6
        test_in0 = 2'b11;
        test_in1 = 2'b11;
        # 200;
        // test vector 7
        test_in0 = 2'b11;
        test_in1 = 2'b01;
        # 200;
        // stop simulation
        $stop;
    end

endmodule
```

The code consists of a module instantiation statement, which creates an instance of the 2-bit comparator, and an *initial block*, which generates a sequence of test patterns. The initial block is a special Verilog construct, which is executed once when simulation starts. The statements inside an initial block are executed sequentially. Each test pattern is generated by three statements, as in

```
        // test vector 2
        test_in0 = 2'b01;
        test_in1 = 2'b00;
        # 200;
```

The first two statements specify the values for the `test_in0` and `test_in1` signals and the third indicates that the two values will last for 200 time units. The last statement, **$stop**, is a Verilog system function that stops the simulation and returns the control to simulation software.

The code has no monitor. We can observe the input and output waveforms on a simulator's display, which can be treated as a "virtual logic analyzer." The simulated timing diagram of this testbench is shown in Figure 2.16.

Writing code for a comprehensive test vector generator and a monitor requires detailed knowledge of Verilog. For now, this listing can serve as a testbench template for other combinational circuits. We can substitute the `uut` instance and modify the test patterns according to the new circuit. We provide a review of additional modeling and simulation-related language constructs and demonstrate the construction of a more sophisticated testbench in Section 7.5.

1.8 BIBLIOGRAPHIC NOTES

A short bibliographic section appears at the end of each chapter to provide some of the most relevant references for further exploration. A comprehensive bibliography is included at the end of the book.

Verilog is a complex language. The standard is specified in *IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001. Verilog HDL, 2nd edition,* by S. Palnitkar and *Starter's Guide to Verilog 2001* by M. D. Ciletti provide detailed coverage of the language's syntax and constructs. Verilog-2001 includes many improvements over the old standard. The article "The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It" by S. Sutherland summarizes the new features. Derivation of the testbench for a large digital system is a difficult task. *Writing Testbenches: Functional Verification of HDL Models, 2nd edition*, by J. Bergeron focuses on this topic.

1.9 SUGGESTED EXPERIMENTS

At the end of each chapter, some experiments are suggested as exercises. The experiments help us to better understand the concepts and provide a hands-on opportunity to design and debug actual circuits.

1.9.1 Code for gate-level greater-than circuit

Develop the HDL codes in Experiment 2.9.1. The code can be simulated and synthesized after we complete Chapter 2.

1.9.2 Code for gate-level binary decoder

Develop the HDL codes in Experiment 2.9.2. The code can be simulated and synthesized after we complete Chapter 2.

CHAPTER 2

OVERVIEW OF FPGA AND EDA SOFTWARE

2.1 INTRODUCTION

Developing a large FPGA-based system is an involved process that consists of many complex transformations and optimization algorithms. Software tools are needed to automate some of the tasks. We use the Web version of the Xilinx *ISE* package for synthesis and implementation, and use the starter version of Mentor Graphics *ModelSim XE III* package for simulation. In this chapter, we give a brief overview of the FPGA device and the S3 prototyping board, and provide short tutorials for the two software packages to "jump-start" the learning process.

2.2 FPGA

2.2.1 Overview of a general FPGA device

A *field-programmable gate array* (FPGA) is a logic device that contains a two-dimensional array of generic logic cells and programmable switches. The conceptual structure of an FPGA device is shown in Figure 2.1. A logic cell can be configured (i.e., *programmed*) to perform a simple function, and a programmable switch can be customized to provide interconnections among the logic cells. A custom design can be implemented by specifying the function of each logic cell and selectively setting the connection of each programmable switch. Once the design and synthesis are completed, we can use a simple adaptor cable to download the desired logic cell and switch configuration to the FPGA device and obtain

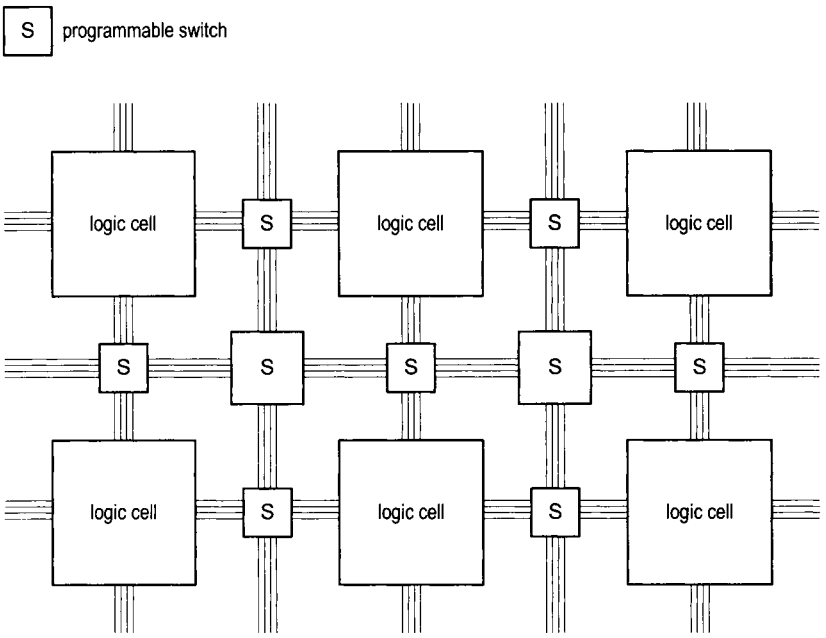
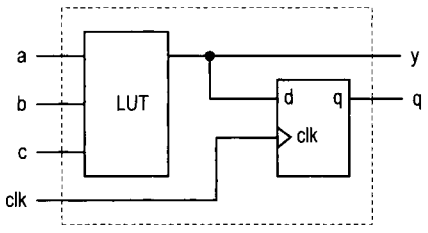


Figure 2.1 Conceptual structure of an FPGA device.



(a) Conceptual diagram

a b c	y
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

(b) Example table

Figure 2.2 Three-input LUT-based logic cell.

the custom circuit. Since this process can be done "in the field" rather than "in a fabrication facility (fab)," the device is known as *field programmable.*

LUT-based logic cell A logic cell usually contains a small configurable combinational circuit with a D-type flip-flop (D FF). The most common method to implement a configurable combinational circuit is a *look-up table* (LUT). An n-input LUT can be considered as a small 2^n-by-1 memory. By properly writing the memory content, we can use an LUT to implement any n-input combinational function. The conceptual diagram of a three-input LUT-based logic cell is shown in Figure 2.2(a). An example of three-input LUT implementation of $a \oplus b \oplus c$ is shown in Figure 2.2(b). Note that the output of the LUT

can be used directly or stored to the D FF. The latter can be used to implement sequential circuits.

Macro cell Most FPGA devices also embed certain *macro cells* or *macro blocks*. These are designed and fabricated at the transistor level, and their functionalities complement the general logic cells. Commonly used macro cells include memory blocks, combinational multipliers, clock management circuits, and I/O interface circuits. Advanced FPGA devices may even contain one or more prefabricated processor cores.

2.2.2 Overview of the Xilinx Spartan-3 devices

This book uses Xilinx Spartan-3 family FPGA devices. Based on the ratio between the number of logic cells and the I/O counts, the family is further divided into several subfamilies. Our discussion applies to all the subfamilies.

Logic cell, slice, and CLB The most basic element of the Spartan-3 device is a *logic cell* (LC), which contains a four-input LUT and a D FF, similar to that in Figure 2.2. In addition, a logic cell contains a carry circuit, which is used to implement arithmetic functions, and a multiplexing circuit, which is used to implement wide multiplexers. The LUT can also be configured as a 16-by-1 static random access memory (SRAM) or a 16-bit shift register.

To increase flexibility and improve performance, eight logic cells are combined with a special internal routing structure. In Xilinx terms, two logic cells are grouped to form a *slice*, and four slices are grouped to form a *configurable logic block* (CLB).

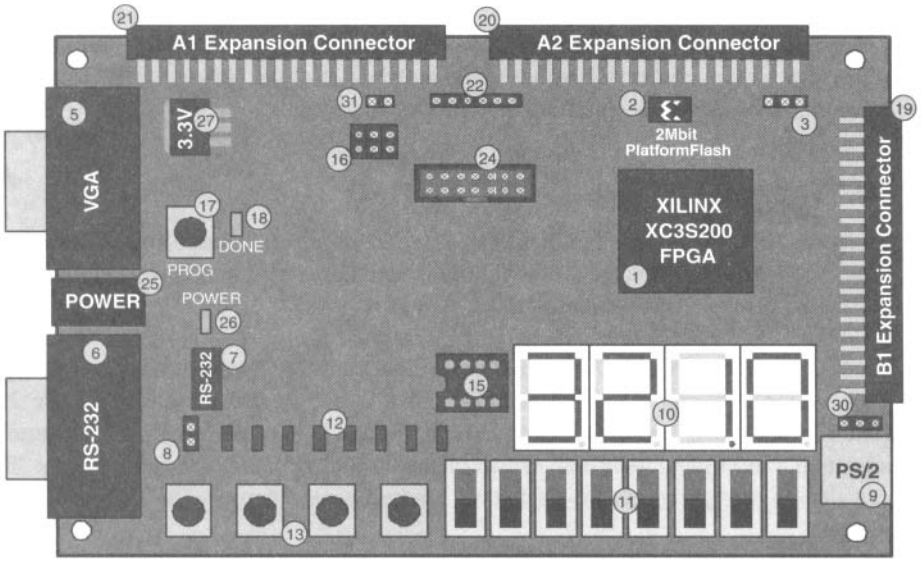
Macro cell The Spartan-3 device contains four types of macro blocks: *combinational multiplier*, *block RAM*, *digital clock manager* (DCM), and *input/output block* (IOB). The combinational multiplier accepts two 18-bit numbers as inputs and calculates the product. The block RAM is an 18K-bit synchronous SRAM that can be arranged in various types of configurations. A DCM uses a digital-delayed loop to reduce clock skew and to control the frequency and phase shift of a clock signal. An IOB controls the flow of data between the device's I/O pins and the internal logic. It can be configured to support a wide variety of I/O signaling standards.

Devices in the Spartan-3 subfamily Although Spartan-3 FPGA devices have similar types of logic cells and macro cells, their densities differ. Each subfamily contains an array of devices of various densities. The numbers of LCs, block RAMs, multipliers, and DCMs of the devices from the Spartan-3 subfamily are summarized in Table 2.1.

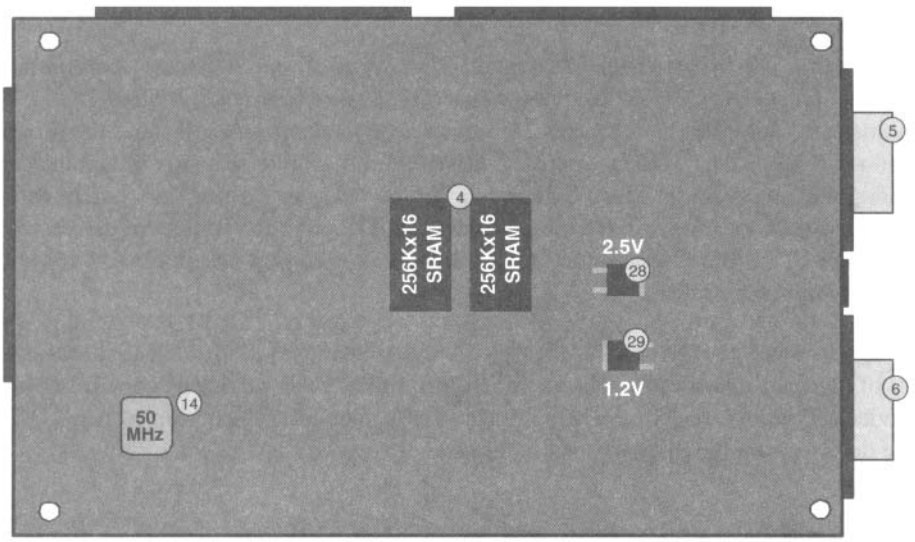
2.3 OVERVIEW OF THE DIGILENT S3 BOARD

The Digilent S3 board is based on a Spartan-3 device (usually an XC3S200) and has an array of built-in peripherals. The simplified layouts of the board are shown in Figure 2.3(a) and (b). The main components and connectors are as follows:

1. Xilinx Spartan-3 XC3S200 FPGA device (XC3S200FT256)
2. 2M-bit Xilinx XCF02S platform flash configuration PROM
3. Jumper to select the configuration source
4. Two 256K-by-16 asynchronous SRAM devices (ISSI IS61LV25616AL-10T)



(a) Top view



(b) Bottom view

Figure 2.3 Layout of an S3 board. (Courtesy of Xilinx, Inc. © Xilinx, Inc. 1994–2007. All rights reserved.)

Table 2.1 Devices in the Spartan-3 family

Device	Number of LCs	Number of block RAMs	Block RAM bits	Number of multipliers	Number of DCMs
XC3S50	1,728	4	72K	4	2
XC3S200	4,320	12	216K	12	4
XC3S400	8,064	16	288K	16	4
XC3S1000	17,280	24	432K	24	4
XC3S1500	29,952	32	576K	32	4
XC3S2000	46,080	40	720K	40	4
XC3S4000	62,208	96	1,728K	96	4
XC3S5000	74,880	104	1,872K	104	4

5. VGA display port
6. RS-232 serial port
7. RS-232 transceiver/voltage-level convertor
8. Second RS-232 transmit and receive channel
9. PS/2 mouse/keyboard port
10. Four-digit seven-segment LED display
11. Eight slide switches
12. Eight discrete LED outputs
13. Four momentary-contact pushbutton switches
14. 50-MHz crystal oscillator clock source
15. Socket for an auxiliary crystal oscillator clock source
16. Jumper to select an FPGA configuration mode
17. Pushbutton switch to force FPGA reconfiguration
18. LED to indicate whether the FPGA is successfully configured
19. 40-pin expansion connector 1 (labeled B1)
20. 40-pin expansion connector 2 (labeled A2)
21. 40-pin expansion connector 3 (labeled A1)
22. JTAG connector for Digilent download cable
23. Digilent low-cost download cable (included in the S3 kit but not shown in Figure 2.3)
24. JTAG port (to be used with the Xilinx Parallel Cable IV and MultiPRO Desktop Tool, which are not included in the S3 kit)
25. Power connector for an unregulated 5-V power supply (included in the S3 kit)
26. Power-on LED indicator
27. 3.3-V voltage regulator
28. 2.5-V voltage regulator
29. 1.2-V voltage regulator
30. Selector for PS2 port voltage supply (3.3 or 5 V)

2.4 DEVELOPMENT FLOW

The simplified development flow of an FPGA-based system is shown in Figure 2.4. To facilitate further reading, we follow the terms used in the Xilinx documentation. The left portion of the flow is the refinement and programming process, in which a system is transformed from an abstract textual HDL description to a device cell-level configuration

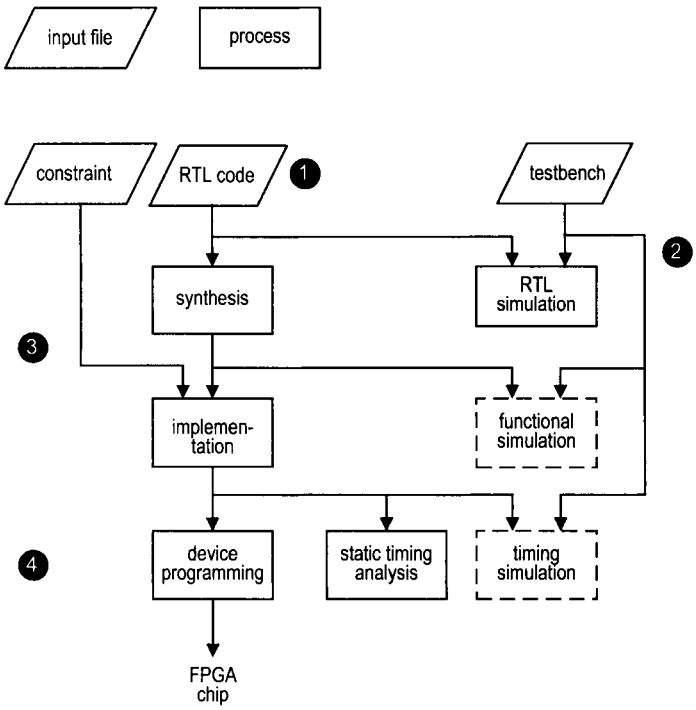


Figure 2.4 Development flow.

and then downloaded to the FPGA device. The right portion is the validation process, which checks whether the system meets the functional specification and performance goals. The major steps in the flow are:

1. Design the system and derive the HDL file(s). We may need to add a separate constraint file to specify certain implementation constraints.
2. Develop the testbench in HDL and perform *RTL simulation*. The RTL term reflects the fact that the HDL code is done at the register transfer level.
3. Perform *synthesis* and *implementation*. The synthesis process is generally known as *logic synthesis*, in which the software transforms the HDL constructs to generic gate-level components, such as simple logic gates and FFs. The *implementation* process consists of three smaller processes: translate, map, and place and route. The *translate process* merges multiple design files to a single netlist. The *map process,* which is generally known as *technology mapping*, maps the generic gates in the netlist to FPGA's logic cells and IOBs. The *place and route process*, which is generally known as *placement and routing*, derives the physical layout inside the FPGA chip. It places the cells in physical locations and determines the routes to connect various signals. In the Xilinx flow, *static timing analysis*, which determines various timing parameters, such as maximal propagation delay and maximal clock frequency, is performed at the end of the implementation process.
4. Generate and download the programming file. In this process, a configuration file is generated according to the final netlist. This file is downloaded to an FPGA device serially to configure the logic cells and switches. The physical circuit can be verified accordingly.

The optional *functional simulation* can be performed after synthesis, and the optional *timing simulation* can be performed after implementation. Functional simulation uses a synthesized netlist to replace the RTL description and checks the correctness of the synthesis process. Timing simulation uses the final netlist, along with detailed timing data, to perform simulation. Because of the complexity of the netlist, functional and timing simulation may require a significant amount of time. If we follow good design and coding practices, the HDL code will be synthesized and implemented correctly. We only need to use RTL simulation to check the correctness of the HDL code and use static timing analysis to examine the relevant timing information. Both functional and timing simulations may be omitted from the development flow.

2.5 OVERVIEW OF THE XILINX ISE PROJECT NAVIGATOR

Xilinx ISE (integrated software environment) controls all aspects of the development flow. *Project Navigator* is a graphical interface for users to access software tools and relevant files associated with a project. We use it to launch all development tasks except ModelSim simulation. The discussion in this section and the tutorial in the next section are based on ISE WebPack version 8.2.

The default ISE window is shown in Figure 2.5. It is divided into four subwindows:

- *Sources window* (top left): hierarchically displays the files included in the project
- *Processes window* (middle left): displays available processes for the source file currently selected
- *Transcript window* (bottom): displays status messages, errors, and warnings
- *Workplace window* (top right): contains multiple document windows (such as HDL code, report, schematic, and so on) for viewing and editing

Each subwindow may be resized, moved, docked, or undocked. The default layout can be restored by selecting View ≻ Restore. Note that a subwindow may contain multiple pages. The tabs at the bottom are used to select the desired page.

Sources window The sources window is used mainly to display files associated with the current project. A typical sources window, which corresponds to the design of Listing 2.2, is shown in Figure 2.6. The top drop-down list, labeled Sources for:, specifies the current design view. The `synthesis/implementation` view should be selected since we use ISE only for synthesis and implementation.

There are three tabs at the bottom, labeled Sources, Snapshots, and Libraries. The Sources tab displays the project name, the FPGA device specified, and user documents and design files. The modules are displayed according to the internal design hierarchy. In Figure 2.6, the eq2 and `eq1` entities reflect the hierarchy of Listing 2.2. The eq2 module also includes the `eq_s3.ucf` file, which specifies the constraints of the design. We can open a file in the workplace window by double-clicking the corresponding module. A *top-level module* icon can be placed next to a module, as in the eq2 module, to invoke synthesis and implementation for this particular module.

The Snapshots tab displays project's "snapshots," which are copies of previously stored project files. The Libraries tab shows all libraries associated with the project.

Processes window The processes window displays the processes available. The display is *context sensitive* and the available processes are based on the source type selected in the sources window. For example, the eq2 module, which is set as the top-level module,

Figure 2.5 Typical ISE window.

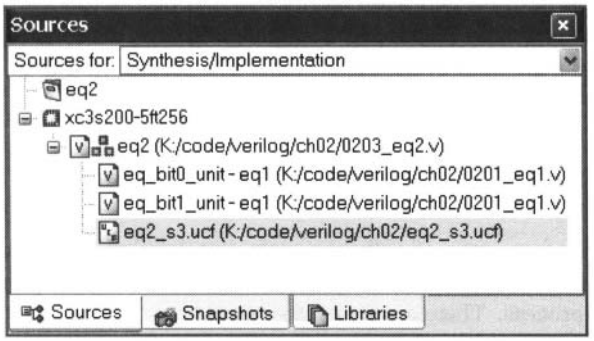


Figure 2.6 Typical sources window.

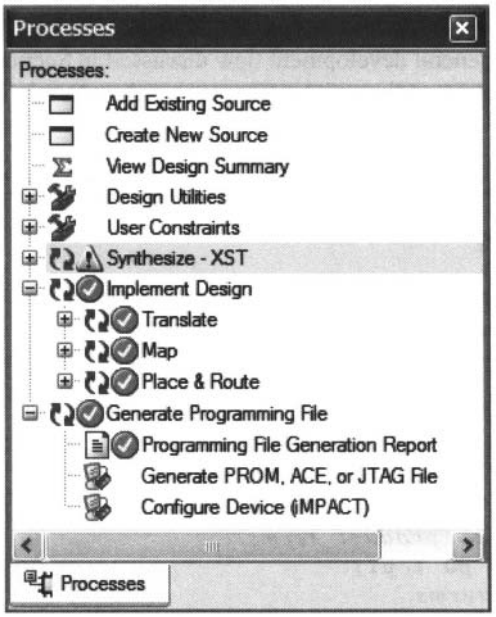


Figure 2.7 Typical processes window.

is selected in Figure 2.6. The available processes are displayed in the processes window, as shown in Figure 2.7. Some processes may also contain several subprocesses. We can initiate a process by clicking on the corresponding icon. ISE incorporates the "auto make" technology, which automatically runs the processes necessary to get to the desired step. For example, when we initiate the Generate Programming File process, ISE automatically invokes the Synthesize and Implement Design processes since file generation is dependent on the implementation result, which, in turn, is dependent on the synthesis result.

Transcript window The transcript window is used to display the progress of a process and relevant messages. The Console page displays errors, warnings, and information messages. An error is signified by a red X mark next to the message and a warning is signified by a yellow ! mark. The Warnings and Errors pages display only warning and error messages.

Workplace window The workplace window is for users to view and edit various types of files. We use it to perform two main tasks. The first task is to view and edit the HDL and constraint files. The default editor is the *ISE Text Editor*, which is a simple text editor with features to assist creation of the HDL code. The second task is to check the design summary and various reports.

2.6 SHORT TUTORIAL ON ISE PROJECT NAVIGATOR

Xilinx ISE consists of an array of software tools, but detailed discussion of their use is beyond the scope of this book. We present a short tutorial in this section to illustrate the basic development process. There are four major steps:

1. Create the design project and HDL codes.
2. Create a testbench and perform RTL simulation.
3. Add a constraint file and synthesize and implement the code.
4. Generate and download the configuration file to an FPGA device.

These steps follow the general development flow discussed in Section 2.4.

We use the 2-bit comparator discussed in Chapter 1 in the tutorial. The codes are repeated in Listings 2.1 and 2.2.

Listing 2.1 Gate-level implementation of a 1-bit comparator

```
module eq1
    // I/O ports
    (
     input wire i0, i1,
     output wire eq
    );

    // signal declaration
    wire p0, p1;

    // body
    // sum of two product terms
    assign eq = p0 | p1;
    // product terms
    assign p0 = ~i0 & ~i1;
    assign p1 = i0 & i1;

endmodule
```

Listing 2.2 Structural description of a 2-bit comparator

```
module eq2
    (
     input  wire[1:0] a, b,
     output wire aeqb
    );

    // internal signal declaration
    wire e0, e1;

    // body
```

```
    // instantiate two 1-bit comparators
    eq1 eq_bit0_unit (.i0(a[0]), .i1(b[0]), .eq(e0));
    eq1 eq_bit1_unit (.eq(e1), .i0(a[1]), .i1(b[1]));

15  // a and b are equal if individual bits are equal
    assign aeqb = e0 & e1;

endmodule
```

2.6.1 Create the design project and HDL codes

There are three tasks in this step:

- Create a project.
- Add or create HDL files.
- Check the HDL syntax.

Create a project An ISE project contains basic information of a design, which includes the source files and a target device. A new project can be created as follows:

1. Select Start ≻ All Programs ≻ Xilinx ISE ≻ Project Navigator (or wherever ISE resides) to launch the ISE project navigator.
2. In Project Navigator, select File ≻ New Project. The New Project Wizard - Create New project dialog appears. Enter the project name as eq2 and the location, and verify that HDL is selected in the Top-level Source Type field. Click Next.
3. The New Project Wizard - Device Properties dialog appears. We need to enter the desired target device in this dialog. This information can be found in the FPGA board manual or by checking the marking on the top of the FPGA chip. For a typical S3 board, select the following:
   - Product Category: `All`
   - Family: `Spartan3`
   - Device: `XC3S200`
   - Package: `FT256`
   - Speed: `-4`

   We also need to verify that the Xilinx XST software is selected for synthesis:
   - Synthesis Tool: `XST (VHDL/Verilog)`
4. Click Next a few times to go through the remaining dialogs and then click Finish to complete the creation.

After a project is created, we can create or add the relevant HDL files and a constraint file.

Create a new HDL file If a file does not exist, we must create a new source file. The procedure to create a new HDL file is:

1. Select Project ≻ New Source. The New Source Wizard - Select Source Type dialog appears. Select Verilog Module and type the file name, eq2. Click Next.
2. The next dialog appears. This dialog allows us to enter port names to be embedded in the Verilog code. However, since the code generated uses the old style of port declaration, we do not use this feature. Click Next.
3. Click Finish and a new HDL text editor window appears in the workplace window. The software automatically generates a comment header and module delimiters.
4. Use the editor to enter the HDL code in Listing 2.2 and save the file.

5. Repeat the process to create another file for the code in Listing 2.1.

Add existing files If a file already exists, it can be added to the project as follows:

1. Select Project ≻ Add Source. A dialog window appears.
2. Go to the appropriate directory and select the desired files. Click Open and a new dialog appears.
3. Click OK to complete the addition. These files now appear in the sources window of the project navigator.

Check the code syntax After completing a new HDL file, we need to check the syntax of the code:

1. Select the desired file in the source window.
2. In the processes window, click the + icon next to Synthesize to expand the process hierarchy.
3. Double-click the Check Syntax process.

The bottom transcript displays the progress of the process and reports errors and warnings, which begin with red X and yellow ! marks. Double-clicking the message leads to the offending line in the file. We can correct the problem, save the file, and repeat the syntax checking process until all syntax errors are eliminated.

2.6.2 Create a testbench and perform the RTL simulation

The testbench functions as a virtual lab bench. It consists of the HDL module to be tested and a code segment to generate the stimulus. The RTL simulation verifies operation of the HDL module in the host computer. ISE contains a built-in ISE simulator and can launch the *ModelSim* simulator manufactured by Mentor Graphics Corporation. Since the latter is more robust and versatile, we use it in the book. Although ModelSim can be invoked from ISE Project Navigator, we treat it as an individual software tool and illustrate its use in Section 2.7.

2.6.3 Add a constraint file and synthesize and implement the code

There are three tasks in this step:

- Add a constraint file.
- Perform synthesis and implementation.
- Check the design summary.

Add a constraint file *Constraints* are certain conditions imposed on the synthesis and implementation processes. For our purposes, the main type of constraint is the pin assignment of a top-level I/O port and the minimal clock rate. During the implementation process, an I/O signal of the top-level module must be mapped to a physical pin of the FPGA device. Since the peripherals' I/O signals are already permanently connected to the designated FPGA's pins on the prototyping board, we must ensure that the signals are mapped to the corresponding pins. The other type of constraint is about timing, which specifies the minimal clock frequency to facilitate the oscillator of the board.

The constraint information is stored in a text file with an extension of .ucf (for the user constraint file). In the eq2 circuit, we can connect the a and b ports to four switches and the aeqb port to an LED to verify the physical operation of the circuit. For the S3 board, the corresponding pins are F12, G12, H14, H13, and K12. The constraint file becomes

```
# 4 slide switches
NET "a<0>"   LOC = "F12"  ;  # switch 0
NET "a<1>"   LOC = "G12"  ;  # switch 1
NET "b<0>"   LOC = "H14"  ;  # switch 2
NET "b<1>"   LOC = "H13"  ;  # switch 3
# led
NET "aeqb"   LOC = "K12"  ;  # led 0
```

Note that the # sign is used for a comment and the text after it is ignored. This file must be added to the design in the sources window.

Several ISE tools are available to specify and generate the constraint file. Since all of our experiments are done in the same prototyping board, the constraints (i.e., pin assignment and clock frequency) remain the same. A constraint template file that includes all connected I/O peripheral signals of the S3 board is provided in the Appendix. One easy method to create a constraint file is simply to copy and edit the template file according to the I/O port names of the current design. The procedure to create the .ucf file for the eq2 circuit proceeds as follows:

1. Copy the template constraint file and rename it eq2_s3.ucf.
2. Follow the procedure in Section 2.6.1 to add the new constraint file to the eq2 module in the sources window.
3. Select the constraint file.
4. In the processes window, click the + icon next to User Constraints to expand the process hierarchy.
5. Double-click the Edit Constraints (Text) process to launch the ISE text editor.
6. Rename the I/O names as needed and then delete the unused pin assignments.
7. Save the file.

The default option of ISE version 8.2 only allows the pin assignments of the existing top-level I/O ports. If unused pin assignments are not deleted from the ucf template, error messages will be generated. We can override the default option as follows:

1. Select the top-level HDL file.
2. Right-click the Implement Design process in the processes window and then select Properties... from the menu. A dialog window appears.
3. In the dialog window, check the Allow Unmatched LOC Constraints option and then click OK.

After this option is turned on, we can use the same ucf template for all designs as long as the same I/O port names are kept in the top-level module, and we don't need to edit the ucf file each time.

Perform synthesis and implementation Invoking the synthesis and implementation procedure is very simple:

1. Select the module to be synthesized and make sure that it is designated as the top-level module (with a green square next to the module icon).
2. Double-click the Implement Design process in the processes window.
3. Although the syntax is checked earlier, the code may contain constructs that cannot be synthesized or may lead to poor implementation (such as a combinational loop). The error and warning messages are displayed in the console tab of the transcript window.
4. Correct the problems and repeat the simulation and synthesis processes if needed.

EQ2 Project Status			
Project File:	eq2.ise	**Current State:**	Placed and Routed
Module Name:	eq2	• **Errors:**	No Errors
Target Device:	xc3s200-5ft256	• **Warnings:**	No Warnings
Product Version:	ISE, 8.1i	• **Updated:**	Sun Jan 21 18:04:45 2007

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of 4 input LUTs	1	3,840	1%	
Logic Distribution				
Number of occupied Slices	1	1,920	1%	
Number of Slices containing only related logic	1	1	100%	
Number of Slices containing unrelated logic	0	1	0%	
Total Number of 4 input LUTs	1	3,840	1%	
Number of bonded IOBs	5	173	2%	
Total equivalent gate count for design	6			
Additional JTAG gate count for IOBs	240			

Performance Summary			
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints		

Detailed Reports					
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Sat Jan 20 22:22:32 2007	0	0	0
Translation Report	Current	Sat Jan 20 22:22:46 2007	0	0	0
Map Report	Current	Sat Jan 20 22:23:00 2007	0	0	2 Infos
Place and Route Report	Current	Sat Jan 20 22:23:18 2007	0	0	1 Info
Static Timing Report	Current	Sat Jan 20 22:23:30 2007	0	0	2 Infos
Bitgen Report					

Figure 2.8 Design summary.

Check the design summary As the project progresses, a report is generated in each process. These reports and key statistics are summarized in a design summary window. We can check the size of the resulting circuit (in terms of the numbers of slices, FFs, and LUTs) and, for a sequential circuit, check whether the clock rate meets the timing constraints. The summary can be invoked by double-clicking the View Design Summary process in the processes window. The summary for the eq2 circuit is shown in Figure 2.8. We can check the use of slices, LUTs, and so on, in the Device Utilization Summary portion. A more detailed report can be invoked by clicking the corresponding link.

2.6.4 Generate and download the configuration file to an FPGA device

The last step is to generate the configuration file and download the file to the FPGA device. There are three tasks in this step:

- Connect the download cable.
- Generate the configuration file.
- Download the configuration file.

The S3 kit comes with a parallel-port JTAG download cable, and the following discussion is based on this cable. The procedures for other cables are similar and detailed instructions may be found in their manuals.

Connect the download cable The procedure to prepare the board is as follows:

1. Make sure that the *PROM* and *Mode* jumpers (labeled 3 and 16 in Figure 2.3) are in their default setting (as the board is shipped).
2. Connect the power cable.
3. Connect one end of the download cable to the parallel port of a PC and connect the other end to the JTAG port (labeled 22 in Figure 2.3) on the S3 board.

Generate the configuration file Generating a configuration file is very straightforward:

1. Make sure that the top-level module is selected in the source window.
2. Click Generate Programming File in the processes window.

After this process is completed, a configuration file, eq2.bit, is generated.

Download the configuration file Downloading the configuration file to an FPGA device is done by a software tool known as *iMPACT*, which can be invoked from ISE Project Navigator. The procedure is as follows:

1. In the processes window, click the + sign to expand the Generate Programming File hierarchy.
2. Double-click the Configure Device (iMPACT) process. The Welcome to iMPACT dialog appears, as shown in Figure 2.9. Check Configure devices using Boundary-Scan (JTAG) and verify that Automatically connect to a cable and identify Boundary-Scan chain is selected in the drop-down list. Click Finish.
3. If a message indicating that two devices are found is displayed, click OK to continue.
4. The main iMPACT window, along with the Assign New Configuration File dialog, appears, as shown in Figure 2.10. The devices connected to the JTAG chain on the board should be detected and displayed.
5. Select the eq2.bit file and click Open to assign this configuration file to the xc3s200 device in the JTAG chain.
6. If a warning message appears, ignore it and click OK.
7. Select Bypass to skip the other device.
8. Right-click on the xc3s200 device image, and select Program .... The Programming Properties dialog opens. Click OK to program the device.
9. The Program Succeeded message appears when the downloading process is completed.

Now the FPGA device is configured and we can test the circuit with the switches and observe the output LED.

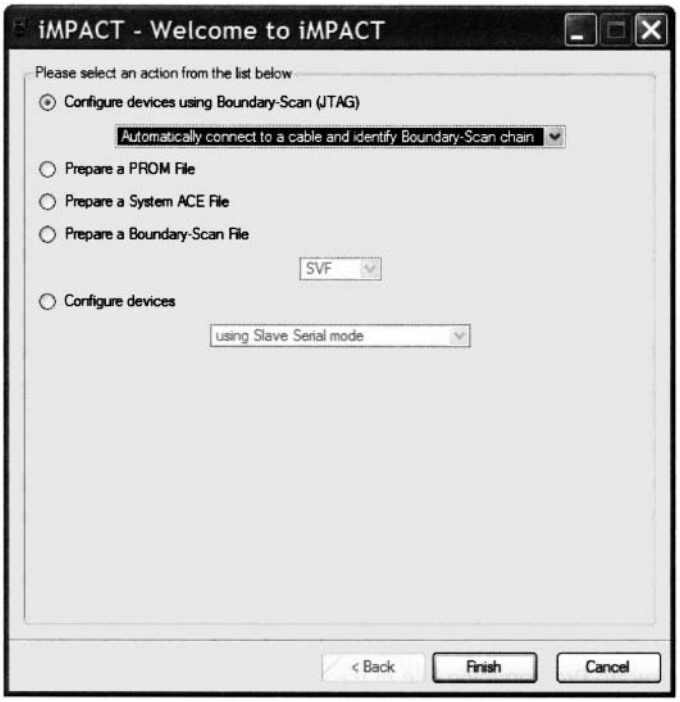


Figure 2.9 iMPACT welcome dialog.

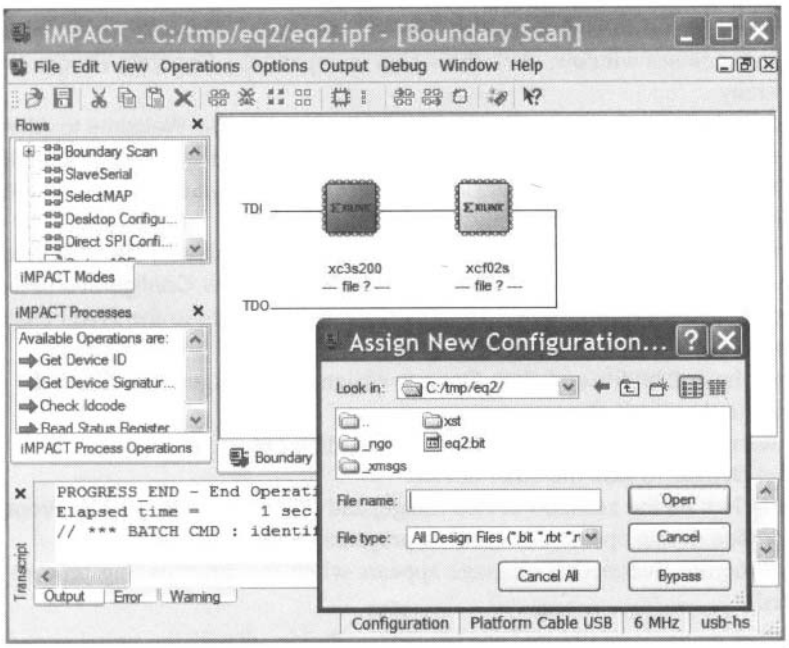


Figure 2.10 iMPACT main window.

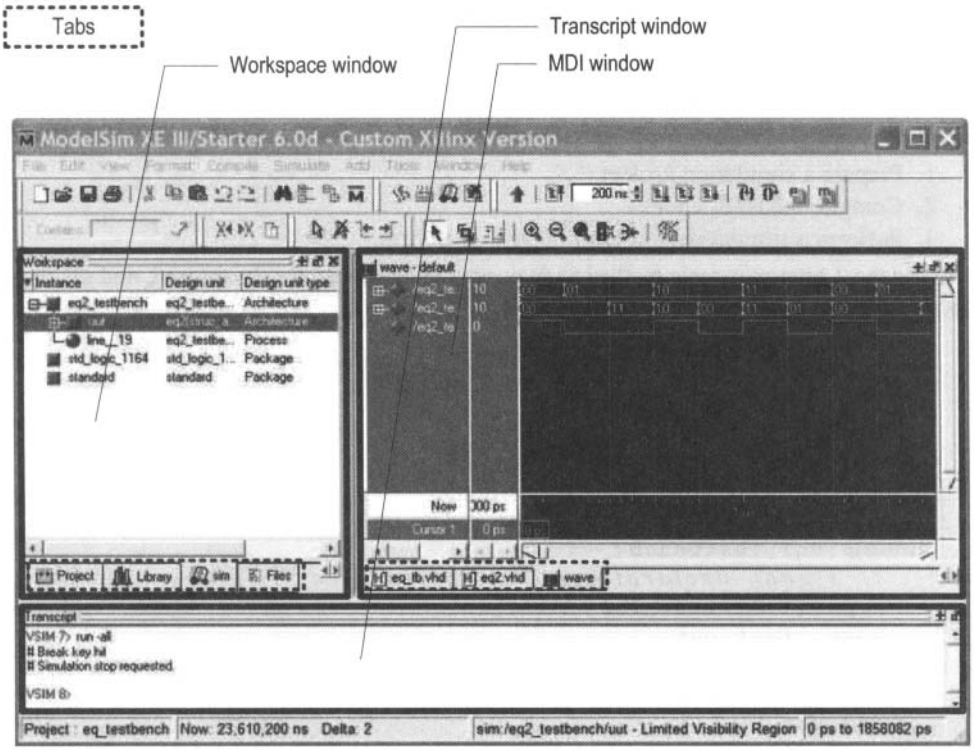


Figure 2.11 Typical ModelSim window.

An alternative way to configure the FPGA is to download the configuration file to a PROM and load the configuration file from the PROM. More information may be found in the sources cited in the bibliographic section.

2.7 SHORT TUTORIAL ON THE MODELSIM HDL SIMULATOR

The ModelSim software is an HDL simulator manufactured by Mentor Graphics Corporation and can run independently without ISE. The discussion in this section is based on ModelSim XE III Starter version 6.0d.

The default ModelSim window is shown in Figure 2.11. It is divided into three subwindows: Transcript window (bottom), Workspace window, and multiple document interface (MDI) window. The Workspace window displays information on the current process. The bottom tab is used to select the desired process page, which can be Project, Library, Sim, and so on. The Transcript window keeps track of command history and messages. It can also be used as a command-line interface to enter ModelSim commands. The MDI window is an area to display HDL text, waveform, and so on. The bottom tab selects the desired pages.

Each subwindow may be resized, moved, docked, or undocked. Additional windows may appear for some operations. The default layout can be restored by selecting Window ≻ Initial Layout.

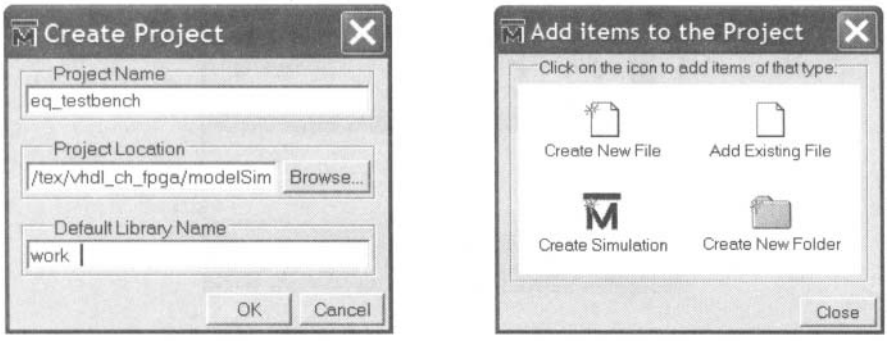
We present a short tutorial in this section to illustrate the basic simulation process. There are three steps:

1. Prepare a simulation project.
2. Compile the HDL codes.
3. Perform a simulation and examine the waveform.

We use the 2-bit comparator testbench discussed in Chapter 1 for the tutorial, and the code is repeated in Listing 2.3.

Listing 2.3 Testbench of a 2-bit comparator

```
// The 'timescale directive specifies that
// the simulation time unit is 1 ns and
// the simulation timestep is 10 ps
`timescale 1 ns/10 ps

module eq2_testbench;
   // signal declaration
   reg  [1:0] test_in0, test_in1;
   wire  test_out;

   // instantiate the circuit under test
   eq2 uut
      (.a(test_in0), .b(test_in1), .aeqb(test_out));

   //  test vector generator
   initial
   begin
      // test vector 1
      test_in0 = 2'b00;
      test_in1 = 2'b00;
      # 200;
      // test vector 2
      test_in0 = 2'b01;
      test_in1 = 2'b00;
      # 200;
      // test vector 3
      test_in0 = 2'b01;
      test_in1 = 2'b11;
      # 200;
      // test vector 4
      test_in0 = 2'b10;
      test_in1 = 2'b10;
      # 200;
      // test vector 5
      test_in0 = 2'b10;
      test_in1 = 2'b00;
      # 200;
      // test vector 6
      test_in0 = 2'b11;
      test_in1 = 2'b11;
```



(a) `Create Project` dialog (b) `Add items` dialog

Figure 2.12 New project dialogs.

```
        # 200;
        // test vector 7
        test_in0 = 2'b11;
        test_in1 = 2'b01;
        # 200;
        // stop simulation
        $stop;
    end

endmodule
```

Prepare a simulation project A ModelSim simulation project consists of the library definition and a collection of HDL files. A testbench is an HDL program and can be created by using the ISE text editor, as discussed in Section 2.6.1. Alternatively, ModelSim also has a built-in editor. We assume that all HDL files are already constructed. The procedure to create a project is as follows:

1. Select Start ≻ All Programs ≻ ModelSim XE III 6.0d ≻ ModelSim (or wherever ModelSim resides) to launch the ModelSim program.
2. Select File ≻ New ≻ Project and the Create Project dialog appears, as shown in Figure 2.12(a). Enter the project name as `eq_testbench`, select the project location, and set Default Library Name to `work`. Click OK. A blank Project page appears in the main window and the Add items to the project dialog appears, as shown in Figure 2.12(b).
3. In the Add items to the project dialog, click Add Existing File and add the necessary HDL files. Click OK. The project tab appears in the workplace subwindow and displays the selected files, as shown in Figure 2.13.

Compile the HDL code The *compile* term here means to convert the HDL code into ModelSim internal format. In Verilog, compiling is done on the module basis. The procedure is:

1. Highlight the `eq1` file and right-click the mouse. Select Compile ≻ Compile Selected. Note that the compiling should be started from the modules at the bottom of the design hierarchy. The progress and messages are displayed in the transcript window.

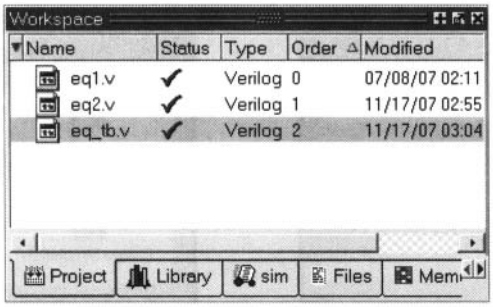


Figure 2.13 Project tab of the workplace panel.

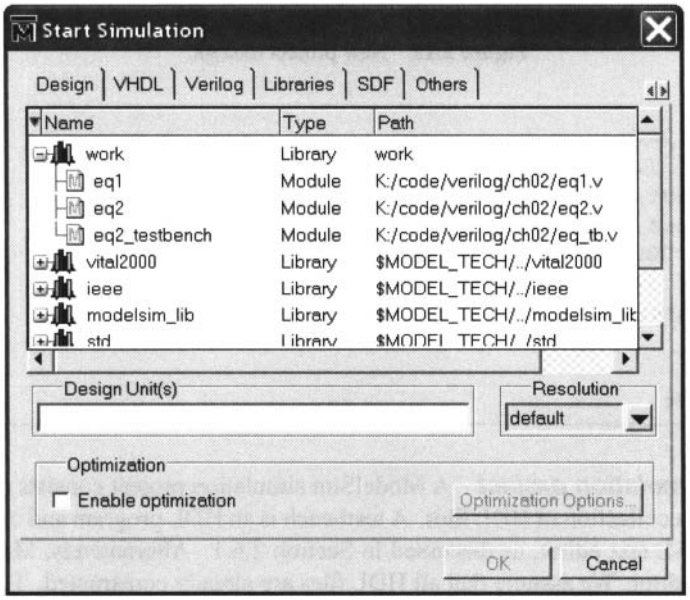


Figure 2.14 Simulate dialog.

2. If the file contains no syntactical error, a check mark shows up. Otherwise, an X mark shows up. Click the red error line in the transcript window to locate the errors. Correct the problems, save the file, and recompile the file.
3. Repeat the preceding steps to compile the eq2 file and then the `eq_tb` file.

Perform a simulation and examine the waveform After compiling the testbench and corresponding files, we can perform the simulation and examine the resulting waveform. This corresponds to running the circuit in a virtual lab bench and checking the waveform in a virtual logic analyzer. The procedure is:

1. Select Simulate ≻ Simulate and the Simulate dialog appears.
2. In the Design tab, find and expand the work library, which is the one defined when we create the project. All compiled units are displayed, as shown in Figure 2.14.
3. Load `eq2_testbench` by double-clicking the corresponding icon. The sim tab appears in the workplace window and the corresponding page displays the structure of

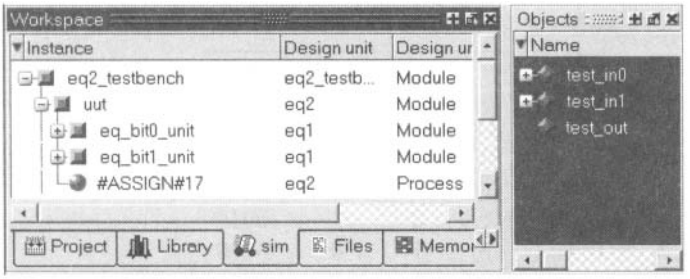


Figure 2.15 Sim panel of the workplace panel.

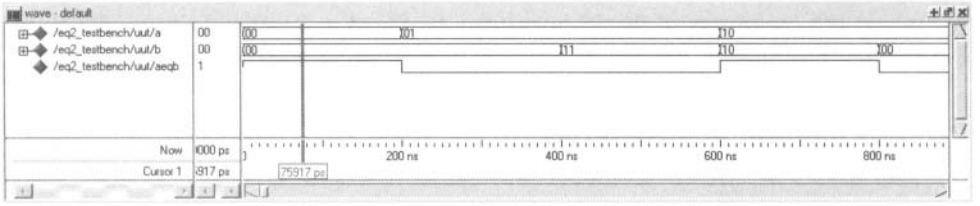


Figure 2.16 Waveform window.

the `eq2_testbench` module, as shown in Figure 2.15. An object window, which contains the signals in the selected module, may also appear.

4. Highlight the uut unit and right-click the mouse. Select Add ≻ Add to Wave. This adds all the signals of the uut unit to the waveform page. The waveform page appears in the MDI window.
5. If necessary, rearrange the signals order and set them to the proper formats (decimal, hex, and so on).
6. Select Simulate ≻ Run. There are several commands to control the simulation: Restart (restart the simulation), Run (run the simulation one step), Continue run (resume the run from the interrupt), Run All (run the simulation forever), and Break (break the simulation). These commands are also shown as icons at the top of the window.
7. The waveform window displays the simulated result, shown in Figure 2.16. We can scroll the window, zoom in, or zoom out to check the correctness of the design.

2.8 BIBLIOGRAPHIC NOTES

Both Xilinx ISE and Mentor Graphics ModelSim are complex software packages, and their documentation exceeds several thousand pages. Most documentation can be accessed via the Help menu. ISE has a short 30-page tutorial, *ISE 8.1i Quick Start Tutorial*, and a more comprehensive 170-page tutorial, *ISE In-Depth Tutorial*. ModelSim also has a similar tutorial, *ModelSim Tutorial*. These tutorials provide an overview on all features of the software package. Relevant information for the Spartan-3 device can be found in its data sheets, *DS099 Spartan-3 FPGA Family: Complete Data Sheet*, which includes the detailed explanation on the logic cells and macro cells. *The Design Warrior's Guide to FPGAs* by Clive Maxfield provides a comprehensive review of FPGA-related issues. The detailed

Table 2.2 Truth table of a 2-to-4 decoder with enable

input			output
en	$a(1)$	$a(0)$	$bcode$
0	–	–	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

layout and I/O connectors of the S3 board may be found in *Spartan-3 Starter Kit Board User Guide*. Information on other prototyping boards can be found in their manuals.

2.9 SUGGESTED EXPERIMENTS

2.9.1 Gate-level greater-than circuit

The greater-than circuit compares two inputs, a and b, and asserts an output when a is greater than b. We want to create a 4-bit greater-than circuit from the bottom up and use only gate-level logical operators. Design the circuit as follows:

1. Derive the truth table for a 2-bit greater-than circuit and obtain the logic expression in the sum-of-products format. Based on the expression, derive the HDL code using only logical operators.
2. Derive a testbench for the 2-bit greater-than circuit. Perform a simulation and verify the correctness of the design.
3. Use four switches as the inputs and one LED as the output. Synthesize the circuit and download the configuration file to the prototyping board. Verify its operation.
4. Use the 2-bit greater-than circuits and 2-bit equality comparators and a minimal number of "glue gates" to construct a 4-bit greater-than circuit. First draw a block diagram and then derive the structural HDL code according to the diagram.
5. Derive a testbench for the 4-bit greater-than circuit. Perform a simulation and verify the correctness of the design.
6. Use eight switches as the inputs and one LED as the output. Synthesize the circuit and download the configuration file to the prototyping board. Verify its operation.

2.9.2 Gate-level binary decoder

An n-to-2^n binary decoder asserts one of 2^n bits according to the input combination. The functional table of a 2-to-4 decoder with an enable signal is shown in Table 2.2. We want to create several decoders using only gate-level logical operators. The procedure is as follows:

1. Determine the logic expressions for the 2-to-4 decoder with enable and derive the HDL code using only logical operators.
2. Derive a testbench for the decoder. Perform a simulation and verify the correctness of the design.
3. Use two switches as the inputs and four LEDs as the outputs. Synthesize the circuit and download the configuration file to the prototyping board. Verify its operation.

4. Use the 2-to-4 decoders to derive a 3-to-8 decoder. First draw a block diagram and then derive the structural HDL code according to the diagram.
5. Derive a testbench for the 3-to-8 decoder. Perform a simulation and verify the correctness of the design.
6. Use three switches as the inputs and eight LEDs as the outputs. Synthesize the circuit and download the configuration file to the prototyping board. Verify its operation.
7. Use the 2-to-4 decoders to derive a 4-to-16 decoder. First draw a block diagram and then derive the structural HDL code according to the diagram.
8. Derive a testbench for the 4-to-16 decoder. Perform a simulation and verify the correctness of the design.

CHAPTER 3

RT-LEVEL COMBINATIONAL CIRCUIT

3.1 INTRODUCTION

The gate-level circuits discussed in Chapter 1 utilize simple bitwise operators to describe gate-level design, which is composed of simple logic cells. In this chapter, we examine the HDL description of circuits that are composed of intermediate-sized components, such as adders, comparators, and multiplexers. Since these components are the basic building blocks used in the *register transfer methodology*, it is sometimes referred to as RT-level design. We discuss more sophisticated Verilog operators, the *always block*, and routing constructs, and then demonstrate the RT-level combinational circuit design through a series of examples.

3.2 OPERATORS

Verilog consists of about two dozen operators. In addition to the bitwise operators discussed in Chapter 1, there are arithmetic, shift, and relational operators. These operators correspond to intermediate-sized components, such as adders and comparators. We examine these operators in this section and also cover miscellaneous synthesis-related Verilog constructs. Table 3.1 summarizes the operators.

Table 3.1 Verilog operators

Type of operation	Operator symbol	Description	Number of operands
Arithmetic	+	addition	2
	-	subtraction	2
	*	multiplication	2
	/	division	2
	%	modulus	2
	**	exponentiation	2
Shift	>>	logical right shift	2
	<<	logical left shift	2
	>>>	arithmetic right shift	2
	<<<	logical left shift	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
Equality	==	equality	2
	!=	inequality	2
	===	case equality	2
	!==	case inequality	2
Bitwise	~	bitwise negation	1
	&	bitwise and	2
	|	bitwise or	2
	^	bitwise xor	2
Reduction	&	reduction and	1
	|	reduction or	1
	^	reduction xor	1
Logical	!	logical negation	1
	&&	logical and	2
	||	logical or	2
Concatenation	{ }	concatenation	any
	{ { } }	replication	any
Conditional	? :	conditional	3

Table 3.2 Operator precedence

Operator	Precedence
! ~ + - (unary)	highest
**	
* / %	
+ - (binary)	
>> << >>> <<<	
< <= > >=	
== != === !==	
&	
^	
|	
&&	
||	
?:	lowest

3.2.1 Arithmetic operators

There are six arithmetic operators: +, -, *, /, %, and **. They represent addition, subtraction, multiplication, division, modulus, and exponentiation operations, respectively. The + and - operators can also be used as unary operators, as in -a. During synthesis, the + and - operators infer the adder and subtractor and they are synthesized by FPGA's logic cells.

Multiplication is a complicated operation and synthesis of the multiplication operator * depends on synthesis software and target device technology. The Xilinx Spartan-3 FPGA family contains prefabricated combinational multiplier blocks. The Xilinx XST software can infer these blocks during synthesis and thus the multiplication operator can be used in HDL code. The XCS200 device of the S3 board consists of twelve 18-by-18 multiplier blocks. Although the synthesis of the multiplication operator is supported, we need to be aware of the limitation on the number and input width of these blocks and use them with care. **Xilinx specific**

The /, %, and ** operators usually cannot be synthesized automatically.

3.2.2 Shift operators

There are four shift operators: >>, <<, >>>, and <<<. The first two represent the logical shift right and left and the last two represent the arithmetic shift right and left.

The 0's are shifted in for a logical shift operation (i.e., >> and <<). The sign bits (i.e., the MSB) are shifted in for the >>> operation and the 0's are shifted in for the <<< operation. Note that there is no difference between the << and <<< operations. The latter is included for completeness. Some shifting examples are shown in Table 3.3.

If both operands of a shift operator are signals, as in a << b, the operator infers a *barrel shifter*, which is a fairly complex circuit. On the other hand, if the shifted amount is fixed, as in a << 2, the operation infers no logic and involves only routing of the input signals.

Table 3.3 Shift operation examples

a	a >> 2	a >>> 2	a << 2	a <<< 2
0100_1111	0001_0011	0001_0011	0011_1100	0011_1100
1100_1111	0011_0011	1111_0011	0011_1100	0011_1100

This type of operation can also be described by using the catenation operator discussed in Section 3.2.5.

3.2.3 Relational and equality operators

There are four relational operators: >, <, <=, and >=. These operators compare two operands and return a Boolean result, which can be *false* (represented by 1-bit scalar value 0) or *true* (represented by 1-bit scalar value 1).

There are four equality operators: ==, !=, ===, and !==. As with the relational operators, they return *false* (1-bit 0) or *true* (1-bit 1). The === and !== operators, known as *case equality* and *case inequality* operators, take into consideration of the matches of the x and z bits in the operands. They cannot be synthesized.

The relational operators and the == and != operators infer comparators during synthesis.

3.2.4 Bitwise, reduction, and logical operators

The bitwise, reduction, and logical operators are somewhat similar and perform the and, or, xor, as well as not operations. These operators are implemented by basic logic cells.

Bitwise operators There are four basic bitwise operators: & (and), | (or), ^ (xor), and ~ (not). The first three operators require two operands. Negation and xor operation can be combined, as in ~^ or ^~, to form the xnor operator. The operations are performed on a bit-by-bit basis and thus are known as bitwise operators. For example, let a, b, and c be 4-bit signals:

```
wire [3:0] a, b, c;
```

The statement

```
assign  c = a | b;
```

is the same as

```
assign  c[3] = a[3] | b[3];
assign  c[2] = a[2] | b[2];
assign  c[1] = a[1] | b[1];
assign  c[0] = a[0] | b[0];
```

Reduction operators The previous &, |, and ^ operators may have only one operand and then are known as reduction operators. The single operand usually has an array data type. The designated operation is performed on all elements of the array and returns a 1-bit result. For example, let a be a 4-bit signal and y be a 1-bit signal:

```
wire [3:0] a;
wire y;
```

Table 3.4 Logical and bitwise operation examples

a	b	a&b	a|b	a&&b	a||b
0	1	0	1	0 (false)	1 (true)
000	000	000	000	0 (false)	0 (false)
000	001	000	001	0 (false)	1 (true)
011	001	001	011	1 (true)	1 (true)

The statement

```
assign  y = | a;  // only one operand
```

is the same as

```
assign  y = a[3] | a[2] | a[1] | a[0];
```

Logical operators There are three logical operators: && (logical and), || (logical or), and ! (logical negate). The logical operators are different from the bitwise operators. If we assume that no x or z is used, the operands of a logical operator are interpreted as false (when all bits are 0's) or true (when at least one bit is 1), and the operation always returns a 1-bit result. As the name suggests, the logical operators should be used as logical connectives of Boolean expressions, as in

```
(state==idle) || ((state==op) && (count>10))
```

Some examples are shown in Table 3.4. The corresponding bitwise operations are also included to illustrate the difference between the two types of operations. Since Verilog uses 0 and 1 to represent the false and true values, bitwise and logical operators can be used interchangeably in some situations. However, it is good practice to use logical operators for Boolean expressions and use bitwise operators for signal manipulation.

3.2.5 Concatenation and replication operators

The concatenation operator, { }, combines segments of elements and small arrays to form a large array. The following example illustrates its use:

```
wire a1;
wire [3:0] a4;
wire [7:0] b8, c8, d8;
. . .
assign b8 = {a4, a4};
assign c8 = {a1, a1, a4, 2'b00};
assign d8 = {b8[3:0], c8[3:0]};
```

Implementation of the concatenation operator involves reconnection of the input and output signals and only requires "wiring."

One application of the concatenation operator is to shift and rotate a signal by a fixed amount, as shown in the following example:

```
wire [7:0] a;
wire [7:0] rot, shl, sha;
. . .
```

```
// rotate a to right 3 bits
assign rot = {a[2:0], a[8:3]};
// shift a to right 3 bits and insert 0 (logic shift)
assign shl = {3'b000, a[8:3]};
// shift a to right 3 bits and insert MSB
// (arithmetic shift)
assign sha = {a[8], a[8], a[8], a[8:3]};
```

The concatenation operator, `N{ }`, replicates the enclosed string. The replication constant, `N`, specifies the number of replications. For example, `{4{2'b01}}` returns `8'b01010101`. The previous arithmetic shift operation can be simplified:

```
assign sha = {3{a[8]}, a[8:3]};
```

3.2.6 Conditional operators

The conditional operator, `?:`, takes three operands and its general format is

```
[signal] = [boolean_exp] ? [true_exp] : [false_exp];
```

The `[boolean_exp]` is a Boolean expression that returns true (`1'b1`) or false (`1'b0`). The `[signal]` gets `[true_exp]` if it is true and `[false_exp]` if it is false. For example, the following circuit obtains the maximum of `a` and `b`:

```
assign max = (a>b) ? a : b;
```

The operator can be thought as a simplified if-then-else statement:

```
if [boolean_exp] then
   [signal] = [true_exp];
else
   [signal] = [false_exp];
```

Despite its simplicity, the conditional operators can be cascaded or nested to specify the desired selection. For example, the `eq1` circuit described in Table 1.1 can be rewritten using conditional operators:

```
assign eq = (~i1 & ~i0) ? 1'b1 :
            (~i1 & i0)  ? 1'b0 :
            (i1 & ~i0)  ? 1'b0 :
            1'b1;
```

We can extend the maximal circuit to return the maximum of `a`, `b`, and `c`:

```
assign max = (a>b) ? ((a>c) ? a : c) :
                     ((b>c) ? b : c);
```

While synthesized, a conditional operator infers a 2-to-1 multiplexing circuit. The detailed derivation is discussed in Section 3.6.

3.2.7 Operator precedence

The operator precedence specifies the order of evaluation. The precedence is shown in Table 3.2. When an expression is evaluated, the operator with higher precedence is evaluated first. For example, in the `a + b >> 1` expression, `a + b` is evaluated first and then `>> 1` is evaluated. We can use parentheses to alter the precedence, as in `a + (b >> 1)`. It is a good practice to use parentheses to make an expression clearer, as in `(a + b) >> 1`, even when they are not required.

3.2.8 Expression bit-length adjustment

As signals in real hardware, nets and variables in a Verilog program usually have different numbers of bits (i.e., *bit lengths* or *widths*). In a Verilog statement, the bit lengths of operands can be different and the adjustment is determined by a set of implicit rules:

- Determine the maximal bit length of the operands in the context, which includes the right-hand-side expression and the left-hand-side signal.
- Extend the bit lengths of operands on the right-hand side to the maximum and evaluate the expression.
- Assign the result to the left-hand-side signal. Truncate the MSBs if the signal's bit length is smaller.

Let us first consider a simple example:

```
wire [7:0] a, b;

assign a = 8'b00000000;
assign b = 0;
```

The first statement assigns an 8-bit value, "00000000", to a. The second statement assigns the integer 0 to b. Recall that the integer in Verilog is 32 bits and thus 0 is represented as "00000000000000000000000000000000". Since b is 8 bits wide, it is truncated to "00000000" during the assignment. Although both statements assign an all-zero pattern to the signals, we need to be aware of how the values are obtained.

Let us consider another example:

```
wire [7:0] a, b;
wire [7:0] sum8;
wire [8:0] sum9;

assign sum8 = a + b;
assign sum9 = a + b;
```

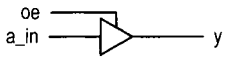
In the first assignment, all operands are 8 bits wide and an 8-bit addition is performed. The carry-out bit of the addition is discarded. In the second assignment, the a and b signals are extended to 9 bits, the bit length of the sum9 signal, and a 9-bit addition is performed. The sum[9] bit gets the resulting carry-out bit. We can also use a concatenation operator if an explicit carry-out signal is desired:

```
assign {c_out,sum8} = a + b;
```

Although the basic conversion rule is simple and intuitive, the subtleties can be error-prone. For example, let a, b, sum1, and sum2 be 8-bit signals. The following statements give a different result:

```
// shift 0 to MSB of sum1
assign sum1 = (a + b) >> 1;
// shift carry-out of a+b to MSB of sum2
assign sum2 = (0 + a + b) >> 1;
```

In the first assignment, all operands are 8 bits wide and an 8-bit addition is performed. The carry-bit is discarded. When the shift operation is performed, 0 is shifted into the MSB. In the second assignment, 0 is an integer and thus is 32 bits wide. The a and b are extended to 32 bits for addition and the summation is shifted. The result is then truncated to 8 bits when assigned to sum2 and sum2[7] gets the original carry-out bit. The conversion becomes more involved when the **signed** data type is used (discussed in Section 7.3).



oe	y
0	Z
1	a_in

Figure 3.1 Symbol and functional table of a tri-state buffer.

A safe but somewhat cumbersome alternative is to adjust the bit lengths of the operands manually. For example, an alternative that may be used to obtain `sum2` is

```
wire [8:0] sum_ext;          // extend sum to 9 bits
. . .
assign sum_ext = {1'b0,a} + {1'b0,b};
assign sum2 = sum_ext[9:1];
```

The code is longer but is more descriptive and less prone to error.

In summary, we must be aware of the Verilog's automatic bit-length adjustment mechanism. Unintended bit-length mismatch may lead to subtle, difficult-to-find errors. Except for trivial adjustments, such as assigning an all-zero pattern with an integer 0, we should either adjust the bit lengths manually or thoroughly document the desired automatic adjustment.

3.2.9 Synthesis of `z` and `x` values

In addition to the regular logic 0 and logic 1, net and variable can contain `z` and `x` values. Although they are not operators, we discuss the synthesis aspect of these two values in this subsection.

Synthesis of `z` The `z` value implies *high impedance* or an open circuit. It is not a normal logic value and can only be synthesized by a *tri-state buffer*. The symbol and function table of a tri-state buffer are shown in Figure 3.1. The operation of the buffer is controlled by an enable signal, `oe` (for "output enable"). When it is 1, the input is passed to output. On the other hand, when it is 0, the y output appears to be an open circuit. The code of the tri-state buffer is

```
assign y = (oe) ? a_in : 1'bz;
```

The most common application for a tri-state buffer is to implement a *bidirectional port* to better utilize a physical I/O pin. A simple example is shown in Figure 3.2. The `dir` signal controls the direction of signal flow of the `bi` pin. When it is 0, the tri-state buffer is in a high-impedance state and the `sig_out` signal is blocked. The pin is used as an input port and the input signal is routed to the `sig_in` signal. When the `dir` signal is 1, the pin is used as an output port and the `sig_out` signal is routed to an external circuit. The HDL code can be derived according to the diagram:

```
module bi_demo(
      inout wire bi,
      . . .
   )

   assign sig_out = output_expression;
```

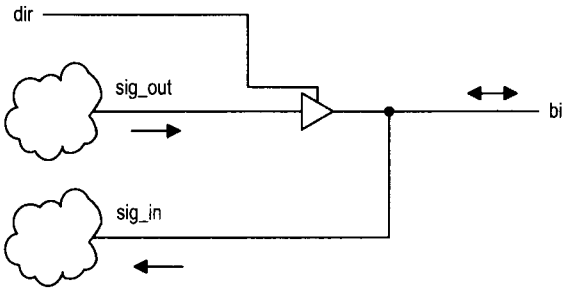


Figure 3.2 Single-buffer bidirectional I/O port.

Table 3.5 Truth table with don't-care

input i	output y
0 0	0
0 1	1
1 0	1
1 1	x

```
. . .
assign some_signal = expression_with_sig_in;
. . .
assign bi = (dir) ?  sig_out : 1'bz;
assign sig_in = bi;
. . .
```

Note that the mode of the `bi` port must be declared as **inout** for bidirectional operation.

For a Xilinx Spartan-3 device, a tri-state buffer exists only in the I/O block (IOB) of a physical pin. Thus, the tri-state buffer can be used only for I/O ports that are mapped to the physical pins of an FPGA device. **Xilinx specific**

Synthesis of `x` In some combinational circuits, certain input patterns may never occur and thus the output value is irrelevant. We frequently assign a "don't-care" value to the output. During synthesis, the don't-care will be assigned a value (either 0 or 1) that can help the optimization process. Consider the truth table shown in Table 3.5. We assume that the `i` will never be `11` and thus the corresponding output is specified as don't-care. In synthesis, we can use `x` for the don't-care value. One possible code for the previous table is

```
assign y = (i==2'b00) ? 1'b0 :
           (i==2'b01) ? 1'b1 :
           (i==2'b10) ? 1'b1 :
           1'bx;   // i==2'b11
```

Although this approach helps to minimize the circuit, it introduces a discrepancy between simulation and synthesis. In simulation, `x` is a unique value rather than "0 or 1". If the input is `11` in simulation, the output becomes `x` and is not consistent with the synthesized result (which can be either 0 or 1). However, since the `11` pattern should never occur in the

original specification, the appearance of the x value can be used to signal potential errors in the testbench.

3.3 ALWAYS BLOCK FOR A COMBINATIONAL CIRCUIT

To facilitate system modeling, Verilog contains a number of *procedural statements*, which are executed in sequence. Since their behavior is different from the normal concurrent circuit model, these statements are encapsulated inside an *always block* or *initial block*. The initial block is executed once when the simulation is started. It can be used in simulation, as in the testbench example in Listing 1.7. Only the always block can be synthesized and it is discussed in this section. Since the procedural statement is more abstract, this type of code is sometimes known as *behaviorial* description.

An always block can be thought of as a black box whose behavior is described by the internal procedural statements. Procedural statements include a rich variety of constructs but many of them don't have clear hardware counterparts. A poorly coded always block frequently leads to unnecessarily complex implementation or cannot be synthesized at all. The focus of this section is on the synthesis of combinational circuits and we limit the discussion to three types of statements:

- Blocking procedural assignment
- If statement
- Case statement

The latter two can be considered as constructs that infer routing structure.

3.3.1 Basic syntax and behavior

The simplified syntax of an always block with a *sensitivity list* (also known as *event control expression*) is

```
always @([sensitivity_list])
begin [optional name]
   [optional local variable declaration];

   [procedural statement];
   [procedural statement];
  . . .
end
```

The `[sensitivity_list]` term is a list of signals and events to which the always block responds (i.e., is "sensitive to"). For a combinational circuit, all the input signals should be included in this list. The body is composed of any number of procedural statements. The **begin** and **end** delimiters can be omitted if there is only one procedural statement in the body. The `@([sensitivity_list])` term is actually a timing control construct. It is usually the *only* timing control construct in a synthesizable always block.

An always block can be considered as a complex circuit part. It can be suspended or activated. When any signal of the sensitivity list changes or an event occurs, the part is activated and executes the internal procedural statements. Since there is no other timing control construct, the execution continues to the end and the part is suspended. Thus, an always block actually "loops forever" and the initiation of each loop is controlled by the sensitivity list.

3.3.2 Procedural assignment

A procedural assignment can only be used within an always block or initial block. There are two types of assignments: *blocking assignment* and *nonblocking assignment*. Their basic syntax is

```
[variable_name] = [expression];    // blocking assignment
[variable_name] <= [expression];   // nonblocking assignment
```

In a blocking assignment, the expression is evaluated and then assigned to the variable immediately, before execution of the next statement (the assignment thus "blocks" the execution of other statements). It behaves like the normal variable assignment in the C language. In a nonblocking assignment, the evaluated expression is assigned at the end of the always block (the assignment thus does not block the execution of other statements).

The blocking and nonblocking assignments frequently confuse new Verilog users and failing to comprehend their differences can lead to unexpected behavior or race conditions. The basic rule of thumb is:

- *Use blocking assignments for a combinational circuit.*
- *Use nonblocking assignments for a sequential circuit.*

This topic is explained in detail in Section 7.1. Since we focus on combinational circuits in this chapter, only the blocking statement is used.

3.3.3 Variable data types

In a procedural assignment, an expression can only be assigned to an output with one of the *variable* data types, which are **reg**, **integer**, **real**, **time**, and **realtime**. The **reg** data type is like the **wire** data type, but used with a procedural output. The **integer** data type represents a fixed-size (usually 32 bits) signed number in 2's-complement format. Since its size is fixed, we usually don't use it in synthesis. The other data types are for modeling and simulation and cannot be synthesized.

3.3.4 Simple examples

We use two simple examples to illustrate the use and behavior of the always block and procedural blocking assignment.

1-bit comparator We can rewrite the previous 1-bit comparator circuit in Listing 1.1 using an always block. The code is shown in Listing 3.1.

Listing 3.1 Always block implementation of a 1-bit comparator

```
module eq1_always
  (
   input wire i0, i1,
   output reg eq  // eq declared as reg
);

  // p0 and p1 declared as reg
  reg p0, p1;

  always @(i0, i1) // i0 an i1 must be in sensitivity list
  begin
```

```
    // the order of statements is important
    p0 = ~i0 & ~i1;
    p1 = i0 & i1;
    eq = p0 | p1;
  end

endmodule
```

Since the eq, p0, and p1 signals are assigned within the always block, they are declared as the **reg** data type. The sensitivity list consists of i0 and i1, which are separated by a comma. When one of them changes, the always block is activated. The three blocking assignments are executed sequentially, much like the statements in a C program. The order of the statements is important and p0 and p1 must be assigned values before being used.

FYI

In Verilog-1995, the keyword **or** is used in place of the comma in a sensitivity list. For example, the list

```
always @(a, b, c)
```

is written as

```
always @(a or b or c)
```

We use only commas in this book.

A combinational circuit must include all its input signals in the sensitivity list to correctly model the desired behavior. Missing a signal can lead to discrepancy between synthesis and simulation. In Verilog-2001, we can use the notation

```
always @*
```

to implicitly include all the input signals. In this book, we use this construct for the combinational circuit.

Three-input and circuit The similarity of the codes in Listings 1.1 and 3.1 is somewhat misleading. The behavior of continuous assignments and procedural statements is quite different.

Consider the code in Listing 3.2. It is a circuit that performs an and operation over a, b, and c (i.e., a & b & c).

Listing 3.2 Behaviorial reduced and circuit using a variable

```
module and_block_assign
  (
  input wire a, b, c,
  output reg y
  );

  always @*
  begin
    y = a;
    y = y & b;
    y = y & c;
  end

endmodule
```

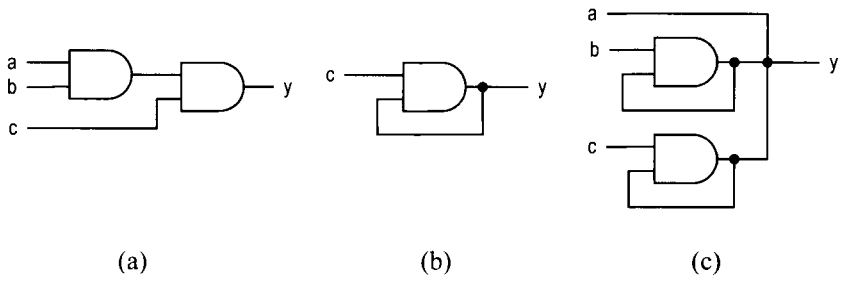


Figure 3.3 Circuits inferred from correct and incorrect code segments.

The inferred circuit is shown in Figure 3.3(a). If we use continuous assignments in a similar way, as shown in Listing 3.3, the description is incorrect.

Listing 3.3 Incorrect code for a reduced and circuit

```
module and_cont_assign
  (
  input wire a, b, c,
  output wire y
);

  assign y = a;
  assign y = y & b;
  assign y = y & c;

endmodule
```

In this code, each continuous assignment infers a circuit part. The three appearances of y on the left-hand side imply that the three outputs are tied together. The corresponding circuit diagram is shown in Figure 3.3(c) and it is clearly not the desired circuit.

3.4 IF STATEMENT

3.4.1 Syntax

The simplified syntax of an if statement is

```
if [boolean_expr]
   begin
      [procedural statement];
      [procedural statement];
      . . .
   end
else
   begin
      [procedural statement];
      [procedural statement];
      . . .
   end
```

Table 3.6 Function table of a four-request priority encoder

input r	output pcode
1 - - -	100
0 1 - -	011
0 0 1 -	010
0 0 0 1	001
0 0 0 0	000

The [boolean_expr] term is a Boolean expression and is evaluated first. If it is true, the statements in the following branch are executed. Otherwise, the statements in the else branch are executed. The else branch is optional and can be omitted. The **begin** and **end** delimiters can be omitted if there is only one procedural statement in a branch.

Multiple if statements can be "cascaded" to evaluate multiple Boolean conditions and establish priorities, as in

```
if [boolean_expr_1]
   . . .
else if [boolean_expr_2]
   . . .
else if [boolean_expr_3]
   . . .
else
   . . .
```

When synthesized, the if statements infer "priority routing" networks. This topic is discussed in Section 3.6.

3.4.2 Examples

We use two simple examples to demonstrate use of the if statement. The first example is a priority encoder. The priority encoder has four requests, r[4], r[3], r[2], and r[1], which are grouped as a single 4-bit r input, and r[4] has the highest priority. The output is the binary code of the highest-order request. The function table is shown in Table 3.6. The HDL code is shown in Listing 3.4.

Listing 3.4 Priority encoder using an if statement

```
module prio_encoder_if
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      if (r[4]==1'b1)       // can be written as (r[4])
         y = 3'b100;
      else if (r[3]==1'b1) // can be written as (r[3])
         y = 3'b011;
      else if (r[2]==1'b1) // can be written as (r[2])
```

Table 3.7 Truth table of a 2-to-4 decoder with enable

	input		output
en	a(1)	a(0)	y
0	–	–	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

```
      y = 3'b010;
   else if (r[1]==1'b1) // can be written as (r[1])
      y = 3'b001;
   else
      y = 3'b000;

endmodule
```

The code first checks the `r[4]` request and assigns 100 to `pcode` if it is asserted. It continues to check the `r[3]` request if `r[4]` is not asserted and repeats the process until all requests are examined. Note that the Boolean expression `(r[4]==1'b1)` is true when `r[4]` is 1. Since the true value is also expressed as `1'b1` in Verilog, the expression can be written as `(r[4])` as well.

The second example is a binary decoder. An n-to-2^n binary decoder asserts one bit of the 2^n-bit output according to the input combination. The functional table of a 2-to-4 decoder is shown in Table 3.7. The circuit also has a control signal, `en`, which enables the decoding function when asserted. The HDL code is shown in Listing 3.5.

Listing 3.5 Binary decoder using an if statement

```
module decoder_2_4_if
   (
    input wire [1:0] a,
    input wire en,
    output reg [3:0] y
   );

   always @*
      if (en==1'b0)          // can be written as (~en)
         y = 4'b0000;
      else if (a==2'b00)
         y = 4'b0001;
      else if (a==2'b01)
         y = 4'b0010;
      else if (a==2'b10)
         y = 4'b0100;
      else
         y = 4'b1000;

endmodule
```

The code first checks whether en is not asserted. If the condition is false (i.e., en is 1), it tests the four binary combinations in sequence. Note that the Boolean expression (en==1'b0) can be written as (~en) as well.

3.5 CASE STATEMENT

3.5.1 Syntax

The simplified syntax of a case statement is

```
case [case_expr]
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   [item]:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   . . .
   default:
      begin
         [procedural statement];
         [procedural statement];
         . . .
      end
   endcase
```

A case statement is a multiway decision statement that compares the [case_expr] expression with a number of [item] expressions. The execution jumps to the branch whose [item] matches the current value of [case_expr]. If there are multiple matched [item] expressions, execution jumps to the branch of the first match. The last item can be an optional **default** keyword. It covers all the unspecified values of the [case_expr] expression. The **begin** and **end** delimiters can be omitted if there is only one procedural statement in a branch.

3.5.2 Examples

We use the same priority encoder and decoder examples to demonstrate use of the case statement. The functional table of a 2-to-4 decoder is shown in Table 3.7. The HDL code using a case statement is shown in Listing 3.6.

Listing 3.6 Binary decoder using a case statement

```
module decoder_2_4_case
   (
    input wire [1:0] a,
    input wire en,
    output reg [3:0] y
   );

   always @*
      case({en,a})
         3'b000, 3'b001, 3'b010, 3'b011: y = 4'b0000;
         3'b100: y = 4'b0001;
         3'b101: y = 4'b0010;
         3'b110: y = 4'b0100;
         3'b111: y = 4'b1000; // default can also be used
      endcase

endmodule
```

We can group multiple values into one item expression, as in line 10, if the identical statements are used for these values. Note that all possible values of the {en,a} expression are covered by the item expressions.

The function table of the priority encoder is shown in Table 3.6. The HDL code is shown in Listing 3.7.

Listing 3.7 Priority encoder using a case statement

```
module prio_encoder_case
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      case(r)
         4'b1000, 4'b1001, 4'b1010, 4'b1011,
         4'b1100, 4'b1101, 4'b1110, 4'b1111:
            y = 3'b100;
         4'b0100, 4'b0101, 4'b0110, 4'b0111:
            y = 3'b011;
         4'b0010, 4'b0011:
            y = 3'b010;
         4'b0001:
            y = 3'b001;
         4'b0000:      // default can also be used
            y = 3'b000;
      endcase

endmodule
```

3.5.3 The casez and casex statements

There are two variations in addition to the regular **case** statement. In a **casez** statement, the z value and the ? character in the item expression are treated as don't-care (i.e., the corresponding bit does not need to be matched). In a **casex** statement, the z and x values and the ? character in the item expression are treated as don't-care. Since the z and x values may appear in simulation, the ? character is preferred.

For example, the previous priority encoder can be coded with a **casez** statement, as shown in Listing 3.8.

Listing 3.8 Priority encoder using a casez statement

```
module prio_encoder_casez
   (
    input wire [4:1] r,
    output reg [2:0] y
   );

   always @*
      casez(r)
         4'b1???: y = 3'b100;
         4'b01??: y = 3'b011;
         4'b001?: y = 3'b010;
         4'b0001: y = 3'b001;
         4'b0000: y = 3'b000; // default can also be used
      endcase

endmodule
```

3.5.4 The full case and parallel case

In Verilog, the item expressions do not need to include all values of the [case_expr] expression and some values can be matched more than once. Consider the following **casez** statement:

```
reg [2:0] s
. . .
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
endcase
```

In this statement, the value 3'b111 is matched twice in the item expressions (once in 3'b111 and once in 3'b1??). Since the first match takes effect, y gets 1'b1 if s is 3'b111. If s is 3'b001, 3'b010, or 3'b011, there are no matches and y will "keep its previous value."

When all possible binary values of the [case_expr] expression are covered by the item expressions, the statement is known as a *full case* statement. For a combinational circuit, we must use a full case statement since each input combination should have an output value. We can add the **default** item to cover all the unmatched values. For example, the previous statement can be revised either as

```
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   default: y = 1'b1; // y gets 1 for unspecified values
endcase
```

or as

```
casez (s)
   3'b111: y = 1'b1;
   3'b1??: y = 1'b0;
   3'b000: y = 1'b1;
   default: y = 1'bx; // y gets don't-care
endcase
```

When the values in the item expressions are mutually exclusive (i.e., a value appears in only one item expression), the statement is know as a *parallel case* statement. For example, the previous casez statement is not a parallel case statement since the value `3'b111` appears twice. The case statements of Listings 3.6 and 3.7 are parallel case statements.

When synthesized, a parallel case statement usually infers a multiplexing routing network and a non-parallel case statement usually infers a priority routing network. This topic is discussed in the next section.

Many synthesis software packages have "full case directive" and "parallel case directive." When they are used, all case statements are treated as full case statements and parallel case statements and synthesized accordingly. Verilog-2001 has similar attributes for this purpose. Using these directives essentially overrides original semantics of Verilog code and introduces a discrepancy between simulation and synthesis. In this book, we express these conditions in code rather than applying these directives or attributes. **FYI**

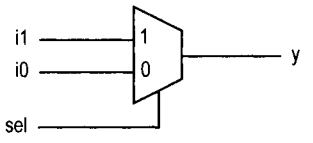
3.6 ROUTING STRUCTURE OF CONDITIONAL CONTROL CONSTRUCTS

We examine several conditional control language constructs, including the `?:` operator and the if and case statements. In the C language, these constructs are executed sequentially. There is no "sequential" control in a combinational circuit. These constructs are realized by *routing networks*. All expressions are evaluated concurrently and the routing network routes the desired result to the output. There are two types of routing structures: *priority routing network* and *multiplexing network*, which are inferred by an if-else type statement and a parallel case statement, respectively.

3.6.1 Priority routing network

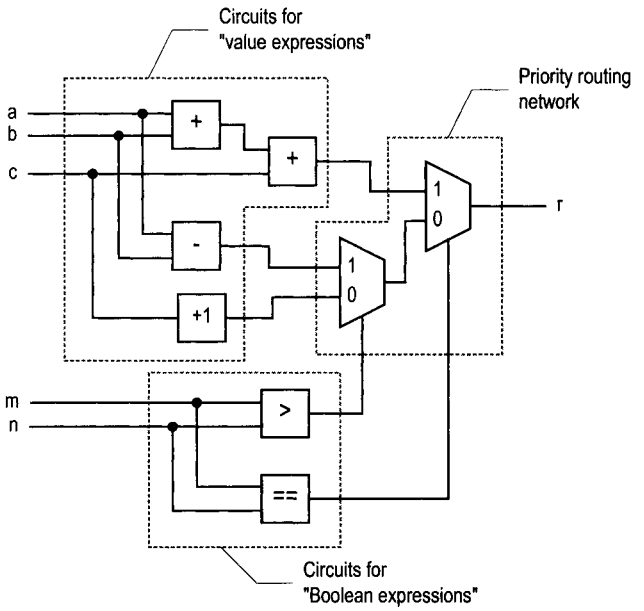
A priority routing network is implemented by a sequence of 2-to-1 multiplexers. The diagram and truth table of a 2-to-1 multiplexer are shown in Figure 3.4(a). An if-else statement implies a priority routing network. Consider the following statement:

```
if (m==n)
   r = a + b + c;
else if (m > n)
   r = a - b;
else
   r = c + 1;
```



sel	y
0 (false)	i0
1 (true)	i1

(a) Diagram of a 2-to-1 multiplexer



(b) Diagram of an if statement

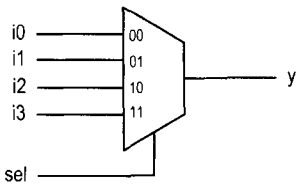
Figure 3.4 Implementation of an if statement.

The conceptual diagram of the statement is shown in Figure 3.4(b). The two 2-to-1 multiplexers form the priority routing network and other components implement various Boolean and arithmetic expressions. If the first Boolean condition (i.e., `m==n`) is true, the result of `a+b+c` is routed to `r`. Otherwise, the data connected to port 0 is passed to `r`. The next Boolean condition (i.e., `m>n`) determines whether the result of `a-b` or `c+1` is routed to the output.

Note that all the Boolean expressions and arithmetic expressions are evaluated concurrently. The outputs from the Boolean circuits set the selection signals of the multiplexers to route the desired value to `r`. The number of cascading stages increases proportionally to the number of if-else clauses. A large number of if-else clauses will lead to a long cascading chain and introduce a large propagation delay.

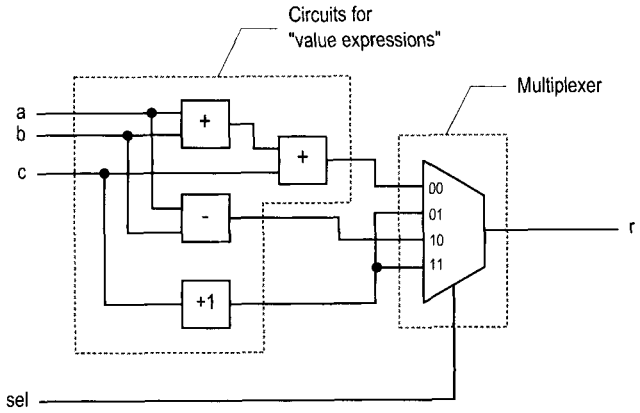
The conditional operator (`?:`) is like a simplified if-else statement and infers similar priority routing networks. A non-parallel case statement sets a preference on the first matched item and thus also infers similar priority routing networks. For example, consider the following case statement:

```
case (expr)
   item1: statement1;
   item2: statement2;
```



sel	y
00	i0
01	i1
10	i2
11	i3

(a) Diagram and functional table of a 4-to-1 multiplexer



(b) Diagram of a parallel case statement

Figure 3.5 Implementation of a parallel case statement.

```
    item3: statement3;
    default: statement4;
endcase
```

It can be translated to

```
if [expr==item1]
    statement1;
else if [expr==item2]
    statement2;
else if [expr==item3]
    statement3;
else
    statement4;
```

3.6.2 Multiplexing network

A multiplexing network is implemented by an n-to-1 multiplexer. The desired input port is specified by the selection signal and the corresponding input is routed to the output. The diagram and functional table of the 2^2-to-1 multiplexer are shown in Figure 3.5(a).

In a parallel case statement, we can map each value of the case expression to an input port of the multiplexer and connect the corresponding evaluated result to the port. The case expression is connected to the selection signal. The construction can best be explained by an example. Consider the following case statement:

```
wire [1:0] sel;
. . .
case (sel)
   2'b00: r = a + b + c;
   2'b10: r = a - b;
   default: r = c + 1; // 2'b01, 2'b11
endcase
```

The conceptual diagram of this statement is shown in Figure 3.5(b). The `sel` varaible can assume four possible values: 00, 01, 10, and 11. It implies a 2^2-to-1 multiplexer with `sel` as the selection signal. The evaluated result of `a+b+c` is routed to `r` when `sel` is 00, the result of `a-b` is routed to `r` when `sel` is 10, and the result of `c+1` is routed to `r` when `sel` is 01 or 11.

Again, note that all value expressions are evaluated concurrently. The `sel` variable is used as the selection signal to route the desired value to the output. The width (i.e., number of input ports) of the multiplexer increases geometrically with the number of bits of `sel`.

In general, the priority routing network is more effective when a preference is given under certain conditions, such as for a priority encoder, and the multiplexing network is more effective for a truth table or function table–based description, such as for a binary decoder.

3.7 GENERAL CODING GUIDELINES FOR AN ALWAYS BLOCK

Verilog is for both modeling and synthesis. While writing code for synthesis, we need to be aware of how the various language constructs are mapped to hardware. This is especially true for an always block since variables and procedural statements can be used within the block. We should remember that the purpose of the code is to infer hardware rather than describing a sequential algorithm in C. Failing to do so frequently leads to unsynthesizable codes, unnecessarily complex implementation, or discrepancy between simulation and synthesis. In this section, we review some common errors and suggest a collection of coding guidelines.

3.7.1 Common errors in combinational circuit codes

Following are common errors found in combinational circuit codes:

- Variable assigned in multiple always blocks
- Incomplete sensitivity list
- Incomplete branch and incomplete output assignment

These errors are discussed below.

Variable assigned in multiple always blocks In Verilog, variables can be assigned (i.e., appear on the left-hand side) in multiple always blocks. For example, the `y` variable is shared by two always blocks is the following code segment:

```
reg y;
reg a, b, clear;
. . .
always @*
   if (clear) y = 1'b0;
```

```
always @*
   y = a & b;
```

Although the code is syntactically correct and can be simulated, it cannot be synthesized. Recall that each always block can be interpreted as a circuit part. The code above indicates that y is the output of both circuit parts and can be updated by each part. No physical circuit exhibits this kind of behavior and thus the code cannot be synthesized. We must group the assignment statements in a single always block, as in

```
always @*
   if (clear)
      y = 1'b0;
   else
      y = a & b;
```

Incomplete sensitivity list For a combinational circuit, the output is a function of input and thus any change in an input signal should activate the circuit. This implies that all input signals should be included in the sensitivity list. For example, a two-input and gate can be written as

```
always @(a, b)    // both a and b are in sensitivity list
   y = a & b;
```

If we forget to include b, the code becomes

```
always @(a)       // a missing from sensitivity list
   y = a & b;
```

Although the code is still syntactically correct, its behavior is very different. When a changes, the always block is activated and y gets the value of a&b. When b changes, the always block remains suspended since it is not "sensitive to" b and y keeps its previous value. No physical circuit exhibits this kind of behavior. Most synthesis software will issue a warning message and infer the and gate instead. However, the simulation software still models the intended behavior and hence introduces a discrepancy between simulation and synthesis.

In Verilog-2001, a special notation, @*, is introduced to implicitly include all the relevant input signals and thus eliminates this problem. It is a good practice to use this notation for combinational circuit description.

Incomplete branch and incomplete output assignment The output of a combinational circuit is a function of input only and the circuit should not contain any *internal state* (i.e., *memory*). One common error with an always block is the inference of unintended memory in a combinational circuit. The Verilog standard specifies that a variable will *keep its previous value* if it is not assigned a value in an always block. During synthesis, this infers an internal state (via a closed feedback loop) or a memory element (such as a latch).

To prevent unintended memory in an always block, all output signals must be assigned proper values all the time. *Incomplete branch* and *incomplete output assignment* are two common errors that lead to unintended memory. To avoid these, we should observe the following rules while developing code for combinational circuit:

- Include all the branches of an if or case statement.
- Assign a value to every output signal in every branch.

Consider the following code segment, which intends to describe a circuit that generates greater-than (i.e., gt) and equal-to (i.e., eq) output signals:

```
always @*
   if (a > b)        // eq not assigned in this branch
      gt = 1'b1;
   else if (a == b) // gt not assigned in this branch
      eq = 1'b1;
                    // final else branch is omitted
```

The segment violates both rules.

Let us first examine the incomplete branch error. There is no else branch in the segment. If both the a>b and a==b expressions are false, both gt and eq are not assigned values. According to Verilog definition, they keep their previous values (i.e., the outputs depend on the internal state) and unintended latches are inferred.

The segment also has incomplete output assignment errors. For example, when the a>b expression is true, eq is not assigned a value and thus will keep its previous state. A latch will be inferred accordingly.

There are two ways to fix the errors. The first is to add the else branch and explicitly assign all output variables. The code becomes

```
always @*
   if (a > b)
      begin
         gt = 1'b1;
         eq = 1'b0;
      end
   else if (a == b)
      begin
         gt = 1'b0;
         eq = 1'b1;
      end
   else    // i.e., a < b
      begin
         gt = 1'b0;
         eq = 1'b0;
      end
```

The alternative is to assign a default value to each variable in the beginning of the always block to cover the unspecified branch and unassigned variable. The code becomes

```
always @*
begin
   gt = 1'b0;  // default value for gt
   eq = 1'b0;  // default value for eq
   if (a > b)
      gt = 1'b1;
   else if (a == b)
      eq = 1'b1;
end
```

Both gt and eq assume 0 if they are not assigned a value later.

The case statement experiences the same errors if some values of the [case_expr] expression are not covered by the item expressions (i.e., not a full-case statement). Consider the following code segment:

```
reg [1:0] s
. . .
```

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

The 2'b01 value is not covered by any branch. If s assumes this combination, y will keep its previous value and an unintended latch is inferred. To fix the error, we must ensure that y is assigned a value all the time. One way to do this is to use the **default** keyword in the end to cover all the unspecified values. We can either replace the last item expression:

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   default: y = 1'b1;   // y gets 1 for 2'b01
endcase
```

or add a new item expression with the don't-care value:

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
   default: y = 1'bx; // y gets x for 2'b01
endcase
```

Alternatively, we can assign a default value in the beginning of the always block:

```
y = 1'b0;    // can also use y = 1'bx for don't-care
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

3.7.2 Guidelines

The always block is a flexible and powerful language construct. However, it must be used with care to infer correct and efficient circuits and to avoid any discrepancy between synthesis and simulation. Following are the coding guidelines for the description of combinational circuits:

- Assign a variable only in a single always block.
- Use blocking statements for combinational circuits.
- Use @* to include all inputs automatically in the sensitivity list.
- Make sure that all branches of the if and case statements are included.
- Make sure that the outputs are assigned in all branches.
- One way to satisfy the two previous guidelines is to assign default values for outputs in the beginning of the always block.
- Describe the desired full case and parallel case in code rather than using software directives or attributes.
- Be aware of the type of routing network inferred by different control constructs.
- Think hardware, not C code.

3.8 PARAMETER AND CONSTANT

3.8.1 Constant

HDL code frequently uses constant values in expressions and array boundaries. These values are fixed within the module and cannot be modified. One good design practice is to replace the "hard literals" with symbolic constants. It makes code clear and helps future maintenance and revision. In Verilog, a constant can be declared using the **localparam** (for "local parameter") keyword. For example, we can declare the width and range of a data bus as

```
localparam DATA_WIDTH = 8,
           DATA_RANGE = 2**DATA_WIDTH - 1;
```

or define a symbolic port name:

```
localparam UART_PORT  = 4'b0001,
           LCD_PORT   = 4'b0010,
           MOUSE_PORT = 4'b0100;
```

The expression in the declaration, such as `2**DATA_WIDTH-1`, is evaluated during pre-processing and thus infers no physical circuit. In this book, we use capital letters for constants.

The use of a constant can best be explained by an example. Consider the code of an adder with the carry-out bit. One way to do it is to extend the input manually by 1 bit, perform the regular addition, and extract the MSB of the summation as the carry-out bit. The code is shown in Listing 3.9.

Listing 3.9 Adder using a hard literal

```
module adder_carry_hard_lit
   (
    input wire [3:0] a, b,
    output wire [3:0] sum,
    output wire cout  // carry-out
   );

   // signal declaration
   wire [4:0] sum_ext;

   //body
   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext[3:0];
   assign cout= sum_ext[4];

endmodule
```

The code is for a 4-bit adder. Hard literals, such as 3 and 4, are used for the ranges, as in **wire** `[4:0]` and `sum_ext[3:0]`, and the MSB, as in `sum_ext[4]`. If we want to revise the code for an 8-bit adder, these literals have to be modified manually. This will be a tedious and error-prone process if the code is complex and the literals are referred to in many places.

To improve readability, we can use a symbolic constant, `N`, to represent the number of bits of the adder. The revised code is shown in Listing 3.10.

Listing 3.10 Adder using constants

```
module adder_carry_local_par
   (
    input wire [3:0] a, b,
    output wire [3:0] sum,
    output wire cout  // carry-out
   );

   // constant declaration
   localparam N = 4,
              N1 = N-1;

   // signal declaration
   wire [N:0] sum_ext;

   //body
   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext[N1:0];
   assign cout= sum_ext[N];

endmodule
```

The constant makes the code easier to understand and maintain.

3.8.2 Parameter

A Verilog module can be instantiated as a component and becomes a part of a larger design, as discussed in Section 1.6. Verilog provides a construct, known as a *parameter*, to pass information into a module. This mechanism makes the module versatile and reusable. A parameter cannot be modified inside the module and thus functions like a constant.

In Verilog-2001, a parameter declaration section can be added in the header, before the port declaration. Its simplified syntax is

```
module [module_name]
   #(
    parameter [parameter_name]=[default_value],
              . . .
              [parameter_name]=[default_value];
   )
   (
    . . .  // I/O port declaration
   );
```

For example, the previous adder code can be modified to use the adder width as a parameter, as shown in Listing 3.11.

Listing 3.11 Adder using a parameter

```
module adder_carry_para
   #(parameter N=4)
   (
     input wire [N-1:0] a, b,
     output wire [N-1:0] sum,
     output wire cout  // carry-out
```

```
    );

    // constant declaration
    localparam N1 = N-1;

    // signal declaration
    wire [N:0] sum_ext;

    //body
    assign sum_ext = {1'b0, a} + {1'b0, b};
    assign sum = sum_ext[N1:0];
    assign cout= sum_ext[N];

endmodule
```

The N parameter is declared with a default value of 4. After N is declared, it can be used in the port declaration and module body, just like a constant.

If the adder is later used as a component in other code, we can assign a desired value to the parameter during component instantiation and override the default value. Similar to the port connection discussed in Section 1.6, parameter assignment can be done either *by name* or *by ordered list*. A potential problem of the by-ordered-list scheme is discussed in Section 1.6 and we always use the by-name scheme in this book. The default value will be used if the parameter assignment is omitted. The use of the parameter in component instantiation is demonstrated in Listing 3.12.

Listing 3.12 Adder instantiation example

```
module adder_insta
   (
     input wire [3:0] a4, b4,
     output wire [3:0] sum4,
     output wire c4,
     input wire [7:0] a8, b8,
     output wire [7:0] sum8,
     output wire c8
   );

   // instantiate 8-bit adder
   adder_carry_para #(.N(8)) unit1
      (.a(a8), .b(b8), .sum(sum8), .cout(c8));

   // instantiate 4-bit adder
   adder_carry_para unit2
      (.a(a4), .b(b4), .sum(sum4), .cout(c4));

endmodule
```

A parameter provides a mechanism to create *scalable code*, in which the "width" of a circuit can be adjusted to meet a specific need. This makes code more portable and encourages design reuse.

3.8.3 Use of parameters in Verilog-1995

The **localparam** keyword, header declaration, and assignment by name discussed earlier are all new Verilog-2001 features. In Verilog-1995, parameters are declared after the header and can only be redefined by using the by-order-list scheme or the **defparam** statement. Furthermore, constants must be declared as parameters, even though they should not be redefined. The previous adder code in Verilog-1995 syntax is shown in Listing 3.13.

FYI

Listing 3.13 Parameter use in Verilog-1995

```
module adder_carry_95 (a, b, sum, cout);
   parameter N = 4;      // parameter declared before the port
   parameter N1 = N-1;   // no localparam in Verilog-1995
   input wire [N1:0] a, b;
   output wire [N1:0] sum;
   output wire cout;

   // signal declaration
   wire [N:0] sum_ext;

   //body
   assign sum_ext = {1'b0, a} + {1'b0, b};
   assign sum = sum_ext[N1:0];
   assign cout= sum_ext[N];

endmodule
```

When a component is instantiated, the parameter can only be redefined by using the by-ordered-list scheme, as in

```
adder_carry_95 #(8,7) unit1
   (.a(a8), .b(b8), .sum(sum8), .cout(c8));
```

or by using the **defparam** statement, as in

```
defparam unit1.N=8;
defparam unit1.N1=7;
adder_carry_95 unit1
   (.a(a8), .b(b8), .sum(sum8), .cout(c8));
```

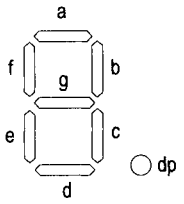
The Verilog-1995 scheme is more tedious and may introduce subtle errors and we don't use it in this book.

3.9 DESIGN EXAMPLES

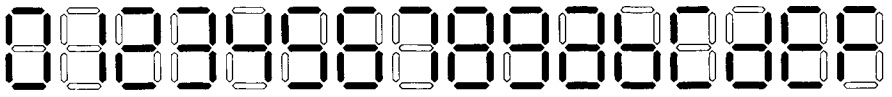
3.9.1 Hexadecimal digit to seven-segment LED decoder

The sketch of a seven-segment LED display is shown in Figure 3.6(a). It consists of seven LED bars and a single round LED decimal point. On the prototyping board, the seven-segment LED is configured as active low, which means that an LED segment is lit if the corresponding control signal is 0.

A hexadecimal digit to seven-segment LED decoder treats a 4-bit input as a hexadecimal digit and generates appropriate LED patterns, as shown in Figure 3.6(b). For completeness, we assume that there is also a 1-bit input, dp, which is connected directly to the decimal



(a) Diagram of a seven-segment LED display



(b) Hexadecimal digit patterns

Figure 3.6 Seven-segment LED display and hexadecimal patterns.

point LED. The LED control signals, dp, a, b, c, d, e, f, and g, are grouped together as a single 8-bit signal, sseg. The code is shown in Listing 3.14. It uses one case statement to list all the desired patterns for the seven LSBs of the sseg signal. The MSB is connected to dp.

Listing 3.14 Hexadecimal digit to seven-segment LED decoder

```
module hex_to_sseg
   (
    input wire [3:0] hex,
    input wire dp,
    output reg [7:0] sseg  // output active low
   );

   always @*
   begin
      case(hex)
         4'h0: sseg[6:0] = 7'b0000001;
         4'h1: sseg[6:0] = 7'b1001111;
         4'h2: sseg[6:0] = 7'b0010010;
         4'h3: sseg[6:0] = 7'b0000110;
         4'h4: sseg[6:0] = 7'b1001100;
         4'h5: sseg[6:0] = 7'b0100100;
         4'h6: sseg[6:0] = 7'b0100000;
         4'h7: sseg[6:0] = 7'b0001111;
         4'h8: sseg[6:0] = 7'b0000000;
         4'h9: sseg[6:0] = 7'b0000100;
         4'ha: sseg[6:0] = 7'b0001000;
         4'hb: sseg[6:0] = 7'b1100000;
         4'hc: sseg[6:0] = 7'b0110001;
         4'hd: sseg[6:0] = 7'b1000010;
         4'he: sseg[6:0] = 7'b0110000;
         default: sseg[6:0] = 7'b0111000;  //4'hf
      endcase
      sseg[7] = dp;
```

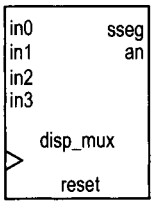
```
   end
30
endmodule
```

There are four seven-segment LED displays on the prototyping board. To save the number of FPGA chip's I/O pins, a time-multiplexing scheme is used. The block diagram of the time-multiplexing module, `disp_mux`, is shown in Figure 3.7(a). The inputs are `in0`, `in1`, `in2`, and `in3`, which correspond to four 8-bit seven-segment LED patterns, and the outputs are `an`, which is a 4-bit signal that enables the four displays individually, and `sseg`, which is the shared 8-bit signal that controls the eight LED segments. The circuit generates a properly timed enable signal and routes the four input patterns to the output alternatively. The design of this module is discussed in Chapter 4. For now, we just treat it as a black box that takes four seven-segment LED patterns, and instantiate it in the code.

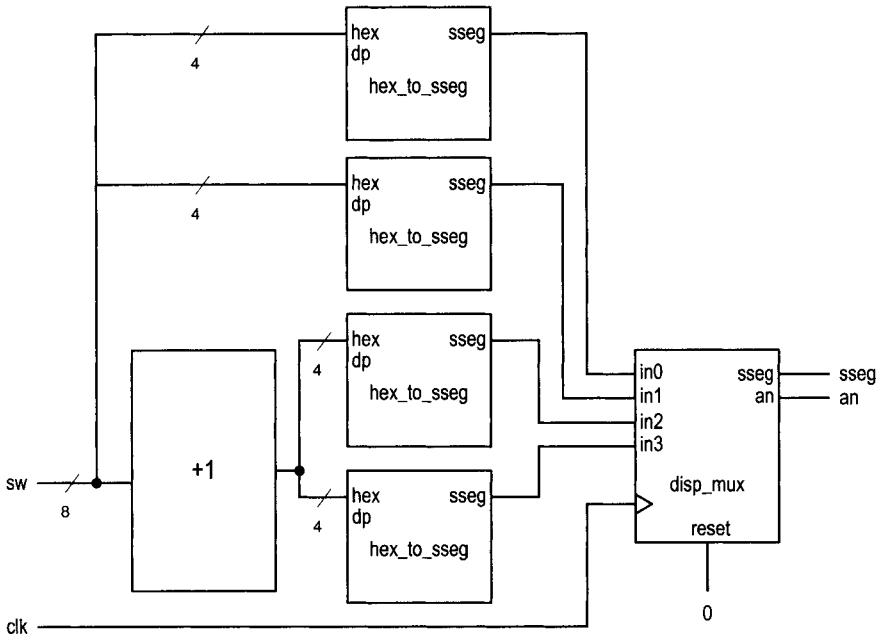
Testing circuit We use a simple 8-bit increment circuit to verify operation of the decoder. The sketch is shown in Figure 3.7(b). The `sw` input is the 8-bit switch of the prototyping board. It is fed to an incrementor to obtain `sw+1`. The original and incremented `sw` signals are then passed to four decoders to display the four hexadecimal digits on seven-segment LED displays. The code is shown in Listing 3.15.

Listing 3.15 Hex-to-LED decoder testing circuit

```
module hex_to_sseg_test
   (
     input wire clk,
     input wire [7:0] sw,
5    output wire [3:0] an,
     output wire [7:0] sseg
   );

   // signal declaration
10 wire [7:0] inc;
   wire [7:0] led0, led1, led2, led3;

   // increment input
   assign inc = sw + 1;
15
   // instantiate four instances of hex decoders
   // instance for 4 LSBs of input
   hex_to_sseg sseg_unit_0
      (.hex(sw[3:0]), .dp(1'b0), .sseg(led0));
20 // instance for 4 MSBs of input
   hex_to_sseg sseg_unit_1
      (.hex(sw[7:4]), .dp(1'b0), .sseg(led1));
   // instance for 4 LSBs of incremented value
   hex_to_sseg sseg_unit_2
25    (.hex(inc[3:0]), .dp(1'b1), .sseg(led2));
   // instance for 4 MSBs of incremented value
   hex_to_sseg sseg_unit_3
      (.hex(inc[7:4]), .dp(1'b1), .sseg(led3));

30 // instantiate 7-seg LED display time-multiplexing module
   disp_mux disp_unit
```



(a) Block diagram of an LED time-multiplexing module



(b) Block diagram of a decoder testing circuit

Figure 3.7 LED time-multiplexing module and decoder testing circuit.

```
    (.clk(clk), .reset(1'b0), .in0(led0), .in1(led1),
     .in2(led2), .in3(led3), .an(an), .sseg(sseg));

endmodule
```

We can follow the procedure in Chapter 2 to synthesize and implement the circuit on the prototyping board. Note that the disp_mux.v file, which contains the code for the time-multiplexing module, and the ucf constraint file must be included in the Xilinx ISE project during synthesis.

3.9.2 Sign-magnitude adder

An integer can be represented in *sign-magnitude* format, in which the MSB is the sign and the remaining bits form the magnitude. For example, 3 and −3 become "0011" and "1011" in 4-bit sign-magnitude format.

A sign-magnitude adder performs an addition operation in this format. The operation can be summarized as follows:

- If the two operands have the same sign, add the magnitudes and keep the sign.
- If the two operands have different signs, subtract the smaller magnitude from the larger one and keep the sign of the number that has the larger magnitude.

One possible implementation is to divide the circuit into two stages. The first stage sorts the two input numbers according to their magnitudes and routes them to the `max` and `min` signals. The second stage examines the signs and performs addition or subtraction on the magnitude accordingly. Note that since the two numbers have been sorted, the magnitude of `max` is always larger than that of `min` and the final sign is the sign of `max`.

The code is shown in Listing 3.16, which realizes the two-stage implementation scheme. For clarity, we split the input number internally and use separate sign and magnitude signals. A parameter, `N`, is used to represent the width of the adder.

Listing 3.16 Sign-magnitude adder

```
module sign_mag_add
   #(
    parameter N=4
   )
   (
    input wire [N-1:0] a, b,
    output reg [N-1:0] sum
   );

   // signal declaration
   reg [N-2:0]  mag_a, mag_b, mag_sum, max, min;
   reg sign_a, sign_b, sign_sum;

   //body
   always @*
   begin
      // separate magnitude and sign
      mag_a = a[N-2:0];
      mag_b = b[N-2:0];
      sign_a = a[N-1];
      sign_b = b[N-1];
```

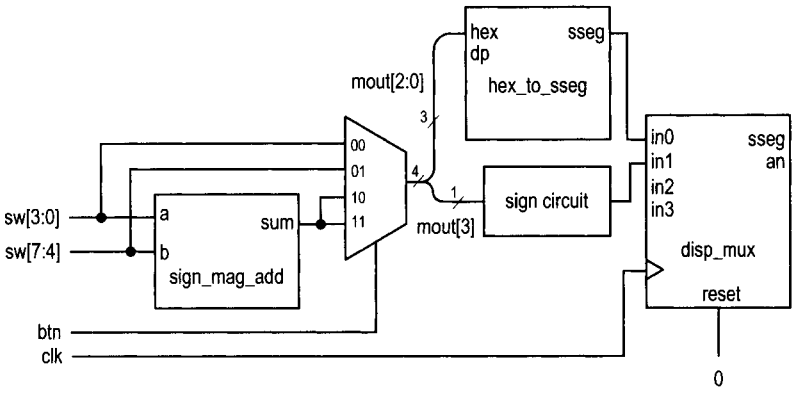


Figure 3.8 Sign-magnitude adder testing circuit.

```
        // sort according to magnitude
        if (mag_a > mag_b)
           begin
25            max = mag_a;
              min = mag_b;
              sign_sum = sign_a;
           end
        else
30         begin
              max = mag_b;
              min = mag_a;
              sign_sum = sign_b;
           end
35      // add/sub magnitude
        if (sign_a==sign_b)
           mag_sum = max + min;
        else
           mag_sum = max - min;
40      // form output
        sum = {sign_sum, mag_sum};
     end
  endmodule
```

Testing circuit We use a 4-bit sign-magnitude adder to verify circuit operation. The sketch of the testing circuit is shown in Figure 3.8. The two input numbers are connected to the 8-bit switch, and the sign and magnitude are shown on two seven-segment LED displays. Two pushbuttons are used as the selection signal of a multiplexer to route an operand or the sum to the display circuit. The rightmost even-segment LED shows the 3-bit magnitude, which is appended with a 0 in front and fed to the hexadecimal to seven-segment LED decoder. The next LED displays the sign bit, which is blank for the plus sign and is lit with a middle LED segment for the minus sign. The two LED patterns are then fed to the time-multiplexing module, `disp_mux`, as explained in Section 3.9.1. The code is shown in Listing 3.17.

Listing 3.17 Sign-magnitude adder testing circuit

```
module sm_add_test
   (
    input wire clk,
    input wire [1:0] btn,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   wire [3:0] sum, mout, oct;
   wire [7:0] led3, led2, led1, led0;

   // instantiate adder
   sign_mag_add #(.N(4)) sm_adder_unit
     (.a(sw[3:0]), .b(sw[7:4]), .sum(sum));

   // magnitude displayed on rightmost 7-seg LED
   assign mout = (btn==2'b00) ? sw[3:0] :
                 (btn==2'b01) ? sw[7:4] :
                 sum;
   assign oct = {1'b0, mout[2:0]};
   // instantiate hex decoder
   hex_to_sseg sseg_unit
      (.hex(oct), .dp(1'b0), .sseg(led0));

   // sign displayed on 2nd 7-seg LED
   // middle LED bar on for negative number
   assign led1 = mout[3] ? 8'b11111110 : 8'b11111111;
   // blank two other LEDs
   assign led2 = 8'b11111111;
   assign led3 = 8'b11111111;

   // instantiate 7-seg LED display time-multiplexing module
   disp_mux disp_unit
      (.clk(clk), .reset(1'b0), .in0(led0), .in1(led1),
       .in2(led2), .in3(led3), .an(an), .sseg(sseg));

endmodule
```

3.9.3 Barrel shifter

Although Verilog has built-in shift functions, there is no rotation operation. In this subsection, we examine an 8-bit barrel shifter that rotates an arbitrary number of bits to the right. The circuit has an 8-bit data input, a, and a 3-bit control signal, amt, which specifies the amount to be rotated. The first design uses a case statement to exhaustively list all combinations of the amt signal and the corresponding rotated results. The code is shown in Listing 3.18.

Listing 3.18 Barrel shifter using a case statement

```
module barrel_shifter_case
   (
     input wire [7:0] a,
     input wire [2:0] amt,
     output reg [7:0] y
    );

   // body
   always @*
      case(amt)
         3'o0: y = a;
         3'o1: y = {a[0], a[7:1]};
         3'o2: y = {a[1:0], a[7:2]};
         3'o3: y = {a[2:0], a[7:3]};
         3'o4: y = {a[3:0], a[7:4]};
         3'o5: y = {a[4:0], a[7:5]};
         3'o6: y = {a[5:0], a[7:6]};
         default: y = {a[6:0], a[7]};
      endcase

endmodule
```

While the code is straightforward, it will become cumbersome when the number of data bits increases. Furthermore, a large number of items in a case statement implies a wide multiplexer, which makes synthesis difficult and leads to a large propagation delay. Alternatively, we can construct the circuit by stages. In the nth stage, the input signal is either passed directly to output or rotated right by 2^n positions. The nth stage is controlled by the nth bit of the `amt` signal. Assume that the 3 bits of `amt` are $m_2m_1m_0$. The total rotated amount after three stages is $m_2 2^2 + m_1 2^1 + m_0 2^0$, which is the desired rotating amount. The code for this scheme is shown in Listing 3.19.

Listing 3.19 Barrel shifter using multistage shifts

```
module barrel_shifter_stage
   (
    input wire [7:0] a,
    input wire [2:0] amt,
    output wire [7:0] y
   );

   // signal declaration
   wire [7:0] s0, s1;

   // body
   // stage 0, shift 0 or 1 bit
   assign s0 = amt[0] ? {a[0], a[7:1]} : a;
   // stage 1, shift 0 or 2 bits
   assign s1 = amt[1] ? {s0[1:0], s0[7:2]} : s0;
   // stage 2, shift 0 or 4 bits
   assign y = amt[2] ? {s1[3:0], s1[7:4]} : s1;

endmodule
```

Testing circuit To test the circuit, we can use the 8-bit switch for the a signal, three pushbutton switches for the amt signal, and the eight discrete LEDs for output. Instead of deriving a new constraint file for pin assignment, we create a new HDL file that wraps the barrel shifter circuit and maps its signals to the prototyping board's signals. The code is shown in Listing 3.20.

Listing 3.20 Barrel shifter testing circuit

```
module shifter_test
   (
    input wire [2:0] btn,
    input wire [7:0] sw,
    output wire [7:0] led
   );

   // instantiate shifter
   barrel_shifter_stage shift_unit
     (.a(sw), .amt(btn), .y(led));

endmodule
```

3.9.4 Simplified floating-point adder

Floating point is another format to represent a number. With the same number of bits, the range in floating-point format is much larger than that in signed integer format. Although VHDL has a built-in floating-point data type, it is too complex to be synthesized automatically.

Detailed discussion of floating-point representation is beyond the scope of this book. We use a simplified 13-bit format in this example and ignore the round-off error. The representation consists of a sign bit, s, which indicates the sign of the number (1 for negative); a 4-bit exponent field, e, which represents the exponent; and an 8-bit significand field, f, which represents the significand or the fraction. In this format, the value of a floating-point number is $(-1)^s * .f * 2^e$. The $.f * 2^e$ is the magnitude of the number and $(-1)^s$ is just a formal way to state that "s equal to 1 implies a negative number." Since the sign bit is separated from the rest of the number, floating-point representation can be considered as a variation of the sign-magnitude format.

We also make the following assumptions:

- Both exponent and significand fields are in unsigned format.
- The representation has to be either normalized or zero. *Normalized representation* means that the MSB of the significand field must be 1. If the magnitude of the computation result is smaller than the smallest normalized nonzero magnitude, $0.10000000 * 2^{0000}$, it must be converted to zero.

Under these assumptions, the largest and smallest nonzero magnitudes are $0.11111111 * 2^{1111}$ and $0.10000000 * 2^{0000}$, and the range is about 2^{16} (i.e., $\frac{0.11111111*2^{1111}}{0.10000000*2^{0000}}$).

Our floating-point adder design follows the process of adding numbers manually in scientific notation. This process can best be explained by examples. We assume that the widths of the exponent and significand are 2 and 1 digits, respectively. Decimal format is used for clarity. The computations of several representative examples are shown in Figure 3.9. The computation is done in four major steps:

		sort	align	add/sub	normalize
eg. 1	+0.54E3	-0.87E4	-0.87E4	-0.87E4	-0.87E4
	-0.87E4	+0.54E3	+0.05E4	+0.05E4	+0.05E4
				-0.82E4	-0.82E4
eg. 2	+0.54E3	-0.55E3	-0.55E3	-0.55E3	-0.55E3
	-0.55E3	+0.54E3	+0.54E3	+0.54E3	+0.54E3
				-0.01E3	-0.10E2
eg. 3	+0.54E0	-0.55E0	-0.55E0	-0.55E0	-0.55E0
	-0.55E0	+0.54E0	+0.54E0	+0.54E0	+0.54E0
				-0.01E0	-0.00E0
eg. 4	+0.56E3	+0.56E3	+0.56E3	+0.56E3	+0.56E3
	+0.52E3	+0.52E3	+0.52E3	+0.52E3	+0.52E3
				+1.07E3	+0.10E4

Figure 3.9 Floating-point addition examples.

1. *Sorting*: puts the number with the larger magnitude on the top and the number with the smaller magnitude on the bottom (we call the sorted numbers "big number" and "small number").
2. *Alignment*: aligns the two numbers so that they have the same exponent. This can be done by adjusting the exponent of the small number to match the exponent of the big number. The significand of the small number has to shift to the right according to the difference in exponents.
3. *Addition/subtraction*: adds or subtracts the significands of two aligned numbers.
4. *Normalization*: adjusts the result to the normalized format. Three types of normalization procedures may be needed:
   - After a subtraction, the result may contain leading zeros in front, as in example 2.
   - After a subtraction, the result may be too small to be normalized and thus needs to be converted to zero, as in example 3.
   - After an addition, the result may generate a carry-out bit, as in example 4.

Our binary floating-point adder design uses a similar algorithm. To simplify the implementation, we ignore the rounding. During alignment and normalization, the lower bits of the significand will be discarded when shifted out. The design is divided into four stages, each corresponding to a step in the foregoing algorithm. The suffixes, 'b', 's', 'a', 'r', and 'n', used in signal names are for "big number," "small number," "aligned number," "result of addition/subtraction," and "normalized number," respectively. The code is developed according to these stages, as shown in Listing 3.21.

Listing 3.21 Simplified floating-point adder

```
module fp_adder
   (
    input wire sign1, sign2,
    input wire [3:0] exp1, exp2,
    input wire [7:0] frac1, frac2,
```

```
    output reg sign_out,
    output reg [3:0] exp_out,
    output reg [7:0] frac_out
);

// signal declaration
//suffix b, s, a, n for
//       big, small, aligned, normalized number
reg signb, signs;
reg [3:0] expb, exps, expn, exp_diff;
reg [7:0] fracb, fracs, fraca, fracn, sum_norm;
reg [8:0] sum;
reg [2:0] lead0;

// body
always @*
begin
   // 1st stage: sort to find the larger number
   if ({exp1, frac1} > {exp2, frac2})
      begin
         signb = sign1;
         signs = sign2;
         expb = exp1;
         exps = exp2;
         fracb = frac1;
         fracs = frac2;
      end
   else
      begin
         signb = sign2;
         signs = sign1;
         expb = exp2;
         exps = exp1;
         fracb = frac2;
         fracs = frac1;
      end

   // 2nd stage: align smaller number
   exp_diff = expb - exps;
   fraca = fracs >> exp_diff;

   // 3rd stage: add/substract
   if (signb==signs)
      sum = {1'b0, fracb} + {1'b0, fraca};
   else
      sum = {1'b0, fracb} - {1'b0, fraca};

   // 4th stage: normalize
   // count leading 0s
   if (sum[7])
      lead0 = 3'o0;
   else if (sum[6])
      lead0 = 3'o1;
```

```
        else if (sum[5])
           lead0 = 3'o2;
        else if (sum[4])
           lead0 = 3'o3;
        else if (sum[3])
           lead0 = 3'o4;
        else if (sum[2])
           lead0 = 3'o5;
        else if (sum[1])
           lead0 = 3'o6;
        else
           lead0 = 3'o7;
        // shift significand according to leading 0
        sum_norm = sum << lead0;
        // normalize with special conditions
        if (sum[8]) // with carry out; shift frac to right
           begin
              expn = expb + 1;
              fracn = sum[8:1];
           end
        else if (lead0 > expb) // too small to normalize
           begin
              expn = 0;          // set to 0
              fracn = 0;
           end
        else
           begin
              expn = expb - lead0;
             fracn = sum_norm;
           end

        // form output
        sign_out = signb;
        exp_out = expn;
        frac_out = fracn;
     end

endmodule
```

The circuit in the first stage compares the magnitudes and routes the big number to the `signb`, `expb`, and `fracb` signals and the smaller number to the `signs`, `exps`, and `fracs` signals. The comparison is done between `exp1&frac1` and `exp2&frac2`. It implies that the exponents are compared first, and if they are the same, the significands are compared.

The circuit in the second stage performs alignment. It first calculates the difference between the two exponents, which is `expb-exps`, and then shifts the significand, `fracs`, to the right by this amount. The aligned significand is labeled `fraca`. The circuit in the third stage performs sign-magnitude addition, similar to that in Section 3.9.2. Note that the operands are extended by 1 bit to accommodate the carry-out bit.

The circuit in the fourth stage performs normalization, which adjusts the result to make the final output conform to the normalized format. The normalization circuit is constructed in three segments. The first segment counts the number of leading zeros. It is somewhat like a priority encoder. The second segment shifts the significands to the left by the amount

specified by the leading-zero counting circuit. The last segment checks the carry-out and zero conditions and generates the final normalized number.

Testing circuit The floating-point adder has two 13-bit input operands. Since the prototyping board has only one 8-bit switch and four 1-bit pushbuttons, it cannot provide enough number of physical inputs to test the circuit. To accommodate the 26 bits of the floating-point adder, we must create a testing circuit and assign constants or duplicated switch signals to the adder's input operands. An example is shown in Listing 3.22. It assigns one operand as constant and uses duplicated switch signals for the other operand. The addition result is passed to the hexadecimal decoders and the sign circuit and is shown on the seven-segment LED display.

Listing 3.22 Floating-point adder testing circuit

```
module fp_adder_test
   (
    input wire clk,
    input wire [1:0] btn,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declarations
   wire sign1, sign2, sign_out;
   wire [3:0] exp1, exp2, exp_out;
   wire [7:0] frac1, frac2, frac_out;
   wire [7:0] led3, led2, led1, led0;

   // body
   // set up the fp adder input signals
   assign sign1 = 1'b0;
   assign exp1 = 4'b1000;
   assign frac1 = {1'b1, sw[1:0], 5'b10101};
   assign sign2 = sw[7];
   assign exp2 = btn;
   assign frac2 = {1'b1, sw[6:0]};

   // instantiate fp adder
   fp_adder fp_unit
      (.sign1(sign1), .sign2(sign2), .exp1(exp1), .exp2(exp2),
       .frac1(frac1), .frac2(frac2), .sign_out(sign_out),
       .exp_out(exp_out), .frac_out(frac_out));

   // instantiate three instances of hex decoders
   // exponent
   hex_to_sseg sseg_unit_0
      (.hex(exp_out), .dp(1'b0), .sseg(led0));
   // 4 LSBs of fraction
   hex_to_sseg sseg_unit_1
      (.hex(frac_out[3:0]), .dp(1'b1), .sseg(led1));
   // 4 MSBs of fraction
   hex_to_sseg sseg_unit_2
      (.hex(frac_out[7:4]), .dp(1'b0), .sseg(led2));
```

```
    //sign
     assign led3 = (sign_out) ? 8'b11111110 : 8'b11111111;

     // instantiate 7-seg LED display time-multiplexing module
45   disp_mux disp_unit
        (.clk(clk), .reset(1'b0), .in0(led0), .in1(led1),
         .in2(led2), .in3(led3), .an(an), .sseg(sseg));

endmodule
```

3.10 BIBLIOGRAPHIC NOTES

Verilog HDL, 2nd edition, by S. Palnitkar and *Starter's Guide to Verilog 2001* by M. D. Ciletti provide detailed coverage of Verilog's syntax and constructs. The article "The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need It" by S. Sutherland summarizes the new features. The article ""full_case parallel_case", the Evil Twins of Verilog Synthesis" by C. E. Cummings examines the caveats of the full-case and parallel-case directives, and his other article, "New Verilog-2001 Techniques for Creating Parameterized Models," discusses the advantage of Verilog-2001's new parameter passing scheme.

3.11 SUGGESTED EXPERIMENTS

3.11.1 Multifunction barrel shifter

Consider an 8-bit shifting circuit that can perform rotating right or rotating left. An additional 1-bit control signal, `lr`, specifies the desired direction.

1. Design the circuit using one rotate-right circuit, one rotate-left circuit, and one 2-to-1 multiplexer to select the desired result. Derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Synthesize the circuit, program the FPGA, and verify its operation.
4. This circuit can also be implemented by one rotate-right shifter with pre- and post-reversing circuits. The reversing circuit either passes the original input or reverses the input bitwise (e.g., if an 8-bit input is $a_7a_6a_5a_4a_3a_2a_1a_0$, the reversed result becomes $a_0a_1a_2a_3a_5a_5a_6a_7$). Repeat steps 2 and 3.
5. Check the report files and compare the number of logic cells and propagation delays of the two designs.
6. Expand the code for a 16-bit circuit and synthesize the code. Repeat steps 1 to 5.
7. Expand the code for a 32-bit circuit and synthesize the code. Repeat steps 1 to 5.

3.11.2 Dual-priority encoder

A dual-priority encoder returns the codes of the highest or second-highest priority requests. The input is a 12-bit `req` signal and the outputs are `first` and `second`, which are the 4-bit binary codes of the highest and second-highest priority requests, respectively.

1. Design the circuit and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit that displays the two output codes on the seven-segment LED display of the prototyping board, and derive the code.

4. Synthesize the circuit, program the FPGA, and verify its operation.

3.11.3 BCD incrementor

The binary-coded-decimal (BCD) format uses 4 bits to represent 10 decimal digits. For example, 259_{10} is represented as "0010 0101 1001" in BCD format. A BCD incrementor adds 1 to a number in BCD format. For example, after incrementing, "0010 0101 1001" (i.e., 259_{10}) becomes "0010 0110 0000" (i.e., 260_{10}).

1. Design a three-digit 12-bit incrementor and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit that displays three digits on the seven-segment LED display and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

3.11.4 Floating-point greater-than circuit

A floating-point greater-than circuit compares two floating-point numbers and asserts output, `gt`, when the first number is larger than the second number. Assume that the two numbers are represented in the format discussed in Section 3.9.4.

1. Design the circuit and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

3.11.5 Floating-point and signed integer conversion circuit

A number may need to be converted to different formats in a large system. Assume that we use the 13-bit format in Section 3.9.4 for the floating-point representation and the 8-bit `signed` data type for the integer representation. An integer-to-floating-point conversion circuit converts an 8-bit integer input to a normalized, 13-bit floating-point output. A floating-point-to-integer conversion circuit reverses the operation. Since the range of a floating-point number is much larger, conversion may lead to the underflow condition (i.e., the magnitude of the converted number is smaller than "00000001") or the overflow condition (i.e., the magnitude of the converted number is larger than "01111111").

1. Design an integer-to-floating-point conversion circuit and derive the code.
2. Derive a testbench and use simulation to verify operation of the code.
3. Design a testing circuit and derive the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.
5. Design a floating-point-to-integer conversion circuit. In addition to the 8-bit integer output, the design should include two status signals, `uf` and `of`, for the underflow and overflow conditions. Derive the code and repeat steps 2 to 4.

3.11.6 Enhanced floating-point adder

The floating-point adder in Section 3.9.4 discards the lower bits when they are shifted out (it is known as *round to zero*). A more accurate method is to *round to the nearest even*, as defined in the *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE Std 754). Three extra bits, known as the *guard*, *round*, and *sticky bits*, are required to implement this

method. If you learned floating-point arithmetic before, modify the floating-point adder in Section 3.9.4 to accommodate the round-to-the-nearest-even method.

CHAPTER 4

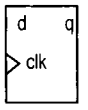
REGULAR SEQUENTIAL CIRCUIT

4.1 INTRODUCTION

A sequential circuit is a circuit with *memory*, which forms the *internal state* of the circuit. Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The *synchronous design methodology* is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. This methodology is the most important principle in developing a large, complex digital system and is the foundation of most synthesis, verification, and testing algorithms. All of the designs in the book follow this methodology.

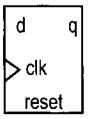
4.1.1 D FF and register

The most basic storage component in a sequential circuit is a D-type flip-flop (D FF). The symbol and function table of a positive edge-triggered D FF are shown in Figure 4.1(a). The value of the `d` signal is sampled at the rising edge of the `clk` signal and stored to FF. A D FF may contain an asynchronous reset signal to clear the FF to 0. Its symbol and function table are shown in Figure 4.1(b). Note that the reset operation is independent of the clock signal.



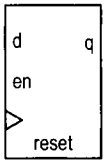
clk	q*
0	q
1	q
↓	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↓	d

(b) D FF with asynchronous reset



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↓	0	q
0	↓	1	d

(c) D FF with synchronous enable

Figure 4.1 Block diagram and functional table of a D FF.

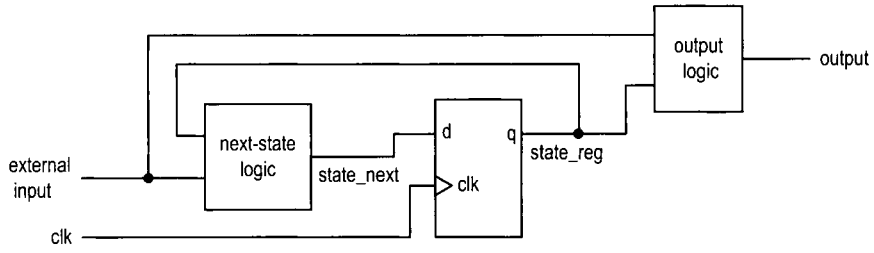


Figure 4.2 Block diagram of a synchronous system.

The three main timing parameters of a D FF are T_{cq} (clock-to-q delay), T_{setup} (setup time), and T_{hold} (hold time). T_{cq} is the time required to propagate the value of d to q at the rising edge of the clock signal. The d signal must be stable around the sampling edge to prevent the FF from entering the metastable state. T_{setup} and T_{hold} specify the time intervals before or after the sampling edge.

A D FF provides 1-bit storage. A collection of D FFs can be grouped together to store multiple bits and is known as a *register*.

4.1.2 Synchronous system

Block diagram The block diagram of a synchronous system is shown in Figure 4.2. It consists of the following parts:

- *State register*: a collection of D FFs controlled by the same clock signal

- *Next-state logic*: combinational logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the register
- *Output logic*: combinational logic that generates the output signal

Maximal operating frequency One of the most difficult design aspects of a sequential circuit is to ensure that the system timing does not violate the setup and hold time constraints. In a synchronous system, the storage components are grouped together and treated as a single register, as shown in Figure 4.2. We need to perform timing analysis on only one memory component.

The timing of a sequential circuit is characterized by f_{max}, *the maximal clock frequency*, which specifies how fast the circuit can operate. The reciprocal of f_{max} specifies T_{clock}, the minimal clock period, which can be interpreted as the interval between two sampling edges of the clock. To ensure correct operation, the next value must be generated and stabilized within this interval. Assume that the maximal propagation delay of next-state logic is T_{comb}. The minimal clock period can be obtained by adding the propagation delays and setup time constraint of the closed loop in Figure 4.2:

$$T_{clock} = T_{cq} + T_{comb} + T_{setup}$$

and the maximal clock rate is the reciprocal:

$$f_{max} = \frac{1}{T_{clock}} = \frac{1}{T_{cq} + T_{comb} + T_{setup}}$$

Timing constraint in Xilinx ISE $^{Xilinx\ specific}$ During synthesis, Xilinx software will analyze the synthesized circuit and show f_{max} in a report. We can also specify the desired operating frequency as a synthesis constraint, and the synthesis software will try to obtain a circuit to satisfy this requirement (i.e., a circuit whose f_{max} is equal to or greater than the desired operating frequency). For example, if we use the 50-MHz (i.e., 20-ns period) oscillator on the prototyping board as the clock source, f_{max} of a sequential circuit must exceed this frequency (i.e., the period must be smaller than 20 ns). The following lines can be added to the constraint file:

```
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50 %;
```

This indicates that the `clk` signal has a maximal period of 20 ns (i.e., 50 MHz) and a duty cycle of 50%.

After synthesis, we can check the relevant timing information by invoking the View Design Summary process from the ISE's Processes window. The Timing Constraints section shows whether the imposed constraints are met, and the Static Timing Report section provides more detailed timing information.

4.1.3 Code development

Our code development follows the basic block diagram in Figure 4.2. The key is to separate the memory component (i.e., the register) from the system. Once the register is isolated, the remaining portion is a pure combinational circuit, and the coding and analysis schemes discussed in previous chapters can be applied accordingly. While this approach may make the code a bit more cumbersome at times, it helps us to better visualize the circuit architecture and avoid unintended memory and subtle mistakes.

Based on the characteristics of the next-state logic, we divide sequential circuits into three categories:

- *Regular sequential circuit*. The state transitions in the circuit exhibit a "regular" pattern, as in a counter or shift register. The next-state logic is constructed primarily by a predesigned, "regular" component, such as an incrementor or shifter.
- *FSM*. The state transitions in the circuit do not exhibit a simple, repetitive pattern. The next-state logic is constructed by "random logic" and synthesized from scratch. It should be called a random sequential circuit, but is commonly known as an FSM (*finite state machine*).
- *FSMD*. The circuit consists of a regular sequential circuit and an FSM. The two parts are known as a *data path* and a *control path*, and the complete circuit is known as an FSMD (*FSM with data path*). This type of circuit is used to implement an algorithm represented by *register-transfer* (RT) methodology, which describes system operation by a sequence of data transfers and manipulations among registers.

The three types of circuits are discussed in this and two subsequent chapters.

4.2 HDL CODE OF THE FF AND REGISTER

Describing storage components in HDL is a subtle procedure, and there are many ways to do it. In fact, one common problem encountered by a new HDL user is the inference of unintended latches and buffers. Instead of covering all possible forms of syntactic descriptions, we introduce the code templates for several commonly used memory components. Since our development process separates the register and the combinational circuit, these components are sufficient for all designs in this book. The components are:

- D FF
- Register
- Register file

All code templates use always blocks. As discussed in Section 3.3.2, nonblocking assignments should be used for the memory elements, whose basic syntax is

```
[variable_name] <= [expression];
```

This type of assignment can avoid potential race condition and eliminate the discrepancy between simulation and synthesis. This topic is explained in detail in Section 7.1.

4.2.1 D FF

We consider three types of D FFs:

- D FF without asynchronous reset
- D FF with asynchronous reset
- D FF with synchronous enable

The first two are the most basic memory components and can be found in the library of any device technology. The third can be constructed from a simple D FF. We include the code since it is a frequently used memory component and can be mapped to the FF of the Spartan-3 device's logic cell.

D FF without asynchronous reset The function table of a D FF is shown in Figure 4.1(a) and the code is shown in Listing 4.1.

Listing 4.1 D FF without asynchronous reset

```
module d_ff
   (
    input wire clk,
    input wire d,
    output reg q
   );

   // body
   always @(posedge clk)
      q <= d;

endmodule
```

The rising edge is expressed by the **posedge** `clk` event in the sensitivity list. The **posedge** (for "positive edge") keyword specifies the direction of the `clk` signal changing toward 1. It indicates that the always block is activated only at the rising edge of the `clk` signal, a condition reflecting the characteristics of an edge-triggered FF. Note that the `d` signal is not included in the sensitive list. This is consistent with the fact that the `d` signal is sampled only at the rising edge of the `clk` signal, and a change in its value does not trigger any immediate response.

D FF with asynchronous reset A D FF may contain an asynchronous reset signal, as shown in the function table of Figure 4.1(b). The signal clears the D FF to 0 any time and is not controlled by the clock signal. It actually has a higher priority than the regularly sampled input. Using an asynchronous reset signal violates the synchronous design methodology and thus should be avoided in normal operation. Its major application is to perform system initialization. For example, we can generate a short reset pulse to force a system to an initial state after turning on the power. The code for a D FF with asynchronous reset is shown in Listing 4.2.

Listing 4.2 D FF with asynchronous reset

```
module d_ff_reset
   (
    input wire clk, reset,
    input wire d,
    output reg q
   );

   // body
   always @(posedge clk, posedge reset)
      if (reset)
         q <= 1'b0;
      else
         q <= d;

endmodule
```

Note that the **posedge** `reset` event is also included in the sensitivity list and its value is checked first in the if statement. The q signal is cleared to 0 if it is asserted and its operation is independent of the `clk` signal.

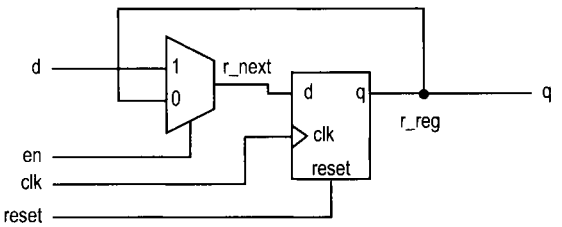


Figure 4.3 D FF with synchronous enable.

D FF with synchronous enable A D FF may include an additional control signal, en, to enable the FF to sample the input value. Its symbol and functional table are shown in Figure 4.1(c). Note that the en signal is examined only at the rising edge of the clock and thus is synchronous. If it is not asserted, the FF keeps its previous value. The code is shown in Listing 4.3.

Listing 4.3 One-segment coding style for a D FF with synchronous enable

```
module d_ff_en_1seg
   (
    input wire clk, reset,
    input wire en,
    input wire d,
    output reg q
   );

   // body
   always @(posedge clk, posedge reset)
      if (reset)
         q <= 1'b0;
      else if (en)
         q <= d;

endmodule
```

Note that there is no else branch after the second if statement. According to Verilog definition, a variable keeps its previous value if it is not assigned. If en is 0, q keeps its previous value. Thus, omission of the else branch describes the desired behavior of this FF.

The enabling feature of this D FF is useful in maintaining synchronism between a fast subsystem and a slow subsystem. For example, assume that the operation rates of a fast and a slow subsystem are 50 MHz and 1 MHz. Instead of using a derived 1-MHz clock to drive the slow subsystem, we can generate a periodic enable tick that is asserted one clock cycle every 50 clock cycles. The slow subsystem is disabled (i.e., keeps the previous state) for the remaining 49 clock cycles. The same scheme can also be applied to eliminate a gated clock signal.

Since the enable signal is synchronous, this circuit can be constructed by a regular D FF and simple next-state logic. The code is shown in Listing 4.4, and its block diagram is shown in Figure 4.3.

Listing 4.4 Two-segment coding style for a D FF with synchronous enable

```
module d_ff_en_2seg
   (
    input wire clk, reset,
    input wire en,
    input wire d,
    output reg q
   );

   // signal declaration
   reg r_reg, r_next;

   // body
   // D FF
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 1'b0;
      else
         r_reg <= r_next;

   // next-state logic
   always @*
      if (en)
         r_next = d;
      else
         r_next = r_reg;

   // output logic
   always @*
      q = r_reg;

endmodule
```

For clarity, we use suffixes _next and _reg to emphasize the next input value and the registered output of an FF. They are connected to the d and q signals of a D FF. The code in Listing 4.3 can be considered as shorthand for this more explicit description.

4.2.2 Register

A register is a collection of D FFs that are controlled by the same clock and reset signals. Like a D FF, a register can have an optional asynchronous reset signal and a synchronous enable signal. The code is identical to that of a D FF except that the array data type is needed for the relevant input and output signals. For example, an 8-bit register with asynchronous reset is shown in Listing 4.5.

Listing 4.5 Register

```
module reg_reset
   (
    input wire clk, reset,
    input wire [7:0] d,
    output reg [7:0] q
   );
```

```
   // body
   always @(posedge clk, posedge reset)
      if (reset)
         q <= 0;
      else
         q <= d;

endmodule
```

4.2.3 Register file

A register file is a collection of registers with one input port and one or more output ports. The write address signal, `w_addr`, specifies where to store data, and the read address signal, `r_addr`, specifies where to retrieve data. The register file is generally used as fast, temporary storage. The code for a parameterized 2^W-by-B register file is shown in Listing 4.6. Two parameters are defined in this design: `W` specifies the number of address bits, which implies that there are 2^W words in the file, and `B` specifies the number of bits in a word.

Listing 4.6 Parameterized register file

```
module reg_file
   #(
    parameter B = 8, // number of bits
              W = 2  // number of address bits
   )
   (
    input wire clk,
    input wire wr_en,
    input wire [W-1:0] w_addr, r_addr,
    input wire [B-1:0] w_data,
    output wire [B-1:0] r_data
   );

   // signal declaration
   reg [B-1:0] array_reg [2**W-1:0];

   // body
   // write operation
   always @(posedge clk)
      if (wr_en)
         array_reg[w_addr] <= w_data;
   // read operation
   assign r_data = array_reg[r_addr];

endmodule
```

The code includes several new features. First, a two-dimensional array data type is defined, as in

```
reg [B-1:0] array_reg [2**W-1:0];
```

It indicates that the `array_reg` variable is an array of `[2**W-1:0]` elements and each element is with the data type of **reg** `[B-1:0]`. Second, a signal is used as an index to access an

element in the array, as in `array_reg[w_addr]`. Although the description is very abstract, Xilinx software recognizes this language construct and can derive the correct implementation accordingly. The `array_reg[...] = ...` and `... = array_reg[...]` statements infer decoding and multiplexing logic, respectively.

Some applications may need to retrieve multiple data words at the same time. This can be done by adding an additional read port:

```
r_data2 = array_reg[r_addr_2];
```

4.2.4 Storage components in a Spartan-3 device$^{Xilinx\ specific}$

In a Spartan-3 device, each logic cell contains a D FF with asynchronous reset and synchronous enable. These D FFs basically constitute the register of Figure 4.2. Since a logic cell also contains a four-input LUT, it will be wasteful if the cell is used simply as 1 bit of a massive storage. The Spartan-3 device also has distributed RAM (random access memory) and block RAM modules, and they can be used for larger storage requirements. These modules can be configured for synchronous operation, and their characteristics are somewhat like a restricted version of the register file. The configuration and inference of these modules are discussed in Chapter 12.

4.3 SIMPLE DESIGN EXAMPLES

We illustrate the construction of several simple, representative sequential circuits in this section.

4.3.1 Shift register

Free-running shift register A free-running shift register shifts its content to the left or right by one position in each clock cycle. There is no other control signal. The code for an N-bit free-running shift-right register is shown in Listing 4.7.

Listing 4.7 Free-running shift register

```
module free_run_shift_reg
   #(parameter N=8)
   (
    input wire clk, reset,
    input wire s_in,
    output wire s_out
   );

   //signal declaration
   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;

   // body
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
```

```
      r_reg <= r_next;

  // next-state logic
  assign r_next = {s_in, r_reg[N-1:1]};
  // output logic
  assign s_out = r_reg[0];

endmodule
```

The next-state logic is a 1-bit shifter, which shifts `r_reg` right one position and inserts the serial input, `s_in`, to the MSB. Since the 1-bit shifter involves only reconnection of the input and output signals, no real logic is needed. Its propagation delay represents the smallest possible T_{comb}, and the corresponding f_{max} represents the highest clock rate that can be achieved for a given device technology.

Universal shift register A universal shift register can load parallel data, shift its content left or right, or remain in the same state. It can perform parallel-to-serial operation (first loading parallel input and then shifting) or serial-to-parallel operation (first shifting and then retrieving parallel output). The desired operation is specified by a 2-bit control signal, `ctrl`. The code is shown in Listing 4.8.

Listing 4.8 Universal shift register

```
module univ_shift_reg
   #(parameter N=8)
   (
    input wire clk, reset,
    input wire [1:0] ctrl,
    input wire [N-1:0] d,
    output wire [N-1:0] q
   );

   //signal declaration
   reg [N-1:0] r_reg, r_next;

   // body
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;

   // next-state logic
   always @*
     case(ctrl)
       2'b00: r_next = r_reg;                      // no op
       2'b01: r_next = {r_reg[N-2:0], d[0]};       // shift left
       2'b10: r_next = {d[N-1], r_reg[N-1:1]};     // shift right
       default: r_next = d;                        // load
     endcase
   // output logic
   assign q = r_reg;
```

```
endmodule
```

The next-state logic uses a 4-to-1 multiplexer to select the desired next value of the register. Note that the LSB and MSB of `d` (i.e., `d[0]` and `d[N-1]`) are used as serial input for the shift-left and shift-right operations.

In a Xilinx Spartan-3 device, a logic cell's 4-input LUT is implemented by a 16-by-1 SRAM. The same SRAM can also be configured as a cascading chain of sixteen 1-bit SRAM cells, which resembles a 16-bit shift register. This can be used to construct certain forms of shift register and leads to very efficient implementation. **Xilinx specific**

4.3.2 Binary counter and variant

Free-running binary counter A free-running binary counter circulates through a binary sequence repeatedly. For example, a 4-bit binary counter counts from "0000", "0001", ..., to "1111" and wraps around. The code for a parameterized N-bit free-running binary counter is shown in Listing 4.9.

Listing 4.9 Free-running binary counter

```
module free_run_bin_counter
   #(parameter N=8)
   (
    input wire clk, reset,
    output wire max_tick,
    output wire [N-1:0] q
   );

   //signal declaration
   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;

   // body
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;  // {N{1b'0}}
      else
         r_reg <= r_next;

   // next-state logic
   assign r_next = r_reg + 1;
   // output logic
   assign q = r_reg;
   assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;
      //can also use (r_reg=={N{1'b1}})

endmodule
```

The next-state logic is an incrementor, which adds 1 to the register's current value. By definition of the + operator, the addition implicitly wraps around after the `r_reg` reaches "1...1". The circuit also consists of an output status signal, `max_tick`, which is asserted when the counter reaches the maximal value, "1...1" (which is equal to $2^N - 1$).

Table 4.1 Function table of a universal binary counter

syn_clr	load	en	up	q*	Operation
1	–	–	–	00···00	synchronous clear
0	1	–	–	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	–	q	pause

The max_tick signal represents a special type of signal that is asserted for a single clock cycle. In this book, we call this type of signal a *tick* and use the suffix _tick to indicate a signal with this property. It is commonly used to interface with the enable signal of other sequential circuits.

Universal binary counter A universal binary counter is more versatile. It can count up or down, pause, be loaded with a specific value, or be synchronously cleared. Its functions are summarized in Table 4.1. Note the difference between the reset and syn_clr signals. The former is asynchronous and should only be used for system initialization. The latter is sampled at the rising edge of the clock and can be used in normal synchronous design. The code for this counter is shown in Listing 4.10.

Listing 4.10 Universal binary counter

```
module univ_bin_counter
   #(parameter N=8)
   (
    input wire clk, reset,
    input wire syn_clr, load, en, up,
    input wire [N-1:0] d,
    output wire max_tick, min_tick,
    output wire [N-1:0] q
   );

   //signal declaration
   reg [N-1:0] r_reg, r_next;

   // body
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;  //
      else
         r_reg <= r_next;

   // next-state logic
   always @*
      if (syn_clr)
         r_next = 0;
      else if (load)
         r_next = d;
      else if (en & up)
```

```
            r_next = r_reg + 1;
      else if (en & ~up)
            r_next = r_reg - 1;
      else
            r_next = r_reg;

   // output logic
   assign q = r_reg;
   assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;
   assign min_tick = (r_reg==0) ? 1'b1 : 1'b0;

endmodule
```

The next-state logic follows the functional table and is described by an always block, which contains an if statement to prioritize the desired operations.

Mod-m counter A mod-m counter counts from 0 to $m - 1$ and wraps around. A parameterized mod-m counter is shown in Listing 4.11. It has two parameters: `M`, which specifies the limit, m; and `N`, which specifies the number of bits needed and should be equal to $\lceil \log_2 \mathtt{M} \rceil$. The code is shown in Listing 4.11, and the default value is for a mod-10 counter.

Listing 4.11 Mod-m counter

```
module mod_m_counter
   #(
    parameter N=4, // number of bits in counter
              M=10 // mod-M
   )
   (
    input wire clk, reset,
    output wire max_tick,
    output wire [N-1:0] q
   );

   //signal declaration
   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;

   // body
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;

   // next-state logic
   assign r_next = (r_reg==(M-1)) ? 0 : r_reg + 1;
   // output logic
   assign q = r_reg;
   assign max_tick = (r_reg==(M-1)) ? 1'b1 : 1'b0;

endmodule
```

The next-state logic is constructed by a conditional operator. If the counter reaches M-1, the new value is cleared to 0. Otherwise, it is incremented by 1.

Inclusion of the N parameter in the code is somewhat redundant since its value depends on M. A more elegant way is to define a function that calculates N from M automatically. This scheme is discussed in Section 7.4.

4.4 TESTBENCH FOR SEQUENTIAL CIRCUITS

A testbench is a program that mimics a physical lab bench, as discussed in Section 1.7. In this section, we illustrate the construction of a simple testbench for the previous universal binary counter. It can serve as a template for other sequential circuits. Development of a more sophisticated testbench is discussed in Section 7.5. The code for the simple testbench is shown in Listing 4.12.

Listing 4.12 Testbench for a universal binary counter

```
`timescale 1 ns/10 ps

// The `timescale directive specifies that
// the simulation time unit is 1 ns and
// the simulator timestep is 10 ps

module bin_counter_tb();

   // declaration
   localparam T=20; // clock period
   reg clk, reset;
   reg syn_clr, load, en, up;
   reg [2:0] d;
   wire max_tick, min_tick;
   wire [2:0] q;

   // uut instantiation
   univ_bin_counter #(.N(3)) uut
     (.clk(clk), .reset(reset), .syn_clr(syn_clr),
      .load(load), .en(en), .up(up), .d(d),
      .max_tick(max_tick), .min_tick(min_tick), .q(q));

   // clock
   // 20 ns clock running forever
   always
   begin
      clk = 1'b1;
      #(T/2);
      clk = 1'b0;
      #(T/2);
   end

   // reset for the first half cycle
   initial
   begin
     reset = 1'b1;
```

```
      #(T/2);
      reset = 1'b0;
   end

   // other stimulus
   initial
   begin
      // ==== initial input =====
      syn_clr = 1'b0;
      load = 1'b0;
      en = 1'b0;
      up  = 1'b1; // count up
      d = 3'b000;
      @(negedge reset);  // wait reset to deassert
      @(negedge clk);    // wait for one clock
      // ==== test load =====
      load = 1'b1;
      d = 3'b011;
      @(negedge clk);    // wait for one clock
      load = 1'b0;
      repeat(2) @(negedge clk);
      // ==== test syn_clear ====
      syn_clr = 1'b1;  // assert clear
      @(negedge clk);
      syn_clr = 1'b0;
      // ==== test up counter and pause ====
      en = 1'b1; // count
      up = 1'b1;
      repeat(10) @(negedge clk);
      en = 1'b0; // pause
      repeat(2) @(negedge clk);
      en = 1'b1;
      repeat(2) @(negedge clk);
      // ==== test down counter ====
      up = 1'b0;
      repeat(10) @(negedge clk);
      // ==== wait statement ====
      // continue until q=2
      wait(q==2);
      @(negedge clk);
      up = 1'b1;
      // continue until min_tick becomes 1
      @(negedge clk);
      wait(min_tick);
      @(negedge clk);
      up = 1'b0;
      // ==== absolute delay  ====
      #(4*T);  //  wait for 80 ns
      en = 1'b0; // pause
      #(4*T);  //  wait for 80 ns
      // ==== stop simulation  ====
      // return to interactive simulation mode
      $stop;
```

```
90    end
    endmodule
```

The code consists of a component instantiation statement, which creates an instance of a 3-bit counter, and three segments, which generate a stimulus for clock, reset, and regular inputs.

The clock generation is specified by an always block:

```
always
begin
    clk = 1'b1;
    #(T/2);
    clk = 1'b0;
    #(T/2);
end
```

The T term is a constant that represents the number of time units in a clock period. It is defined as

```
localparam  T=20; // clock period
```

Note that the always block has no sensitivity list and repeats itself forever. The `clk` signal is assigned between 0 and 1 alternately, and each value lasts for half a period.

The reset stimulus is specified by an initial block:

```
initial
begin
  reset = 1'b1;
  #(T/2);
  reset = 1'b0;
end
```

An initial block is executed *once* at the beginning of a simulation. It indicates that the `reset` signal is set to 1 initially and changed to 0 after half a period. The block represents the "power-on" condition, in which the `reset` signal is asserted momentarily to clear the system to the initial state. Note that, by default, the x value (for unknown), not 0, is assigned to a variable. Using a short reset pulse is a good mechanism for performing system initialization.

The second initial block generates a stimulus for other input signals. We first test the load and clear operations and then exercise counting in both directions. The final **$stop** function forces the simulator to stop simulation.

For a synchronous system with positive edge-triggered FFs, an input signal must be stable around the rising edge of the clock signal to satisfy the setup and hold time constraints. One easy way to achieve this is to change an input signal's value during the 1-to-0 transition of the `clk` signal. We can wait for this transition edge by using

```
@(negedge clk);
```

The **negedge** `clk` event specifies the condition that the `clk` signal changes to 0 (i.e., negative edge). Note that each statement represents a new falling edge, which corresponds to the advancement of one clock cycle. In our template, we generally use this statement to specify the progress of time. For multiple clock cycles, we can use a repeat statement, as in

```
repeat(10) @(negedge clk);  // repeat 10 times
```

Several additional timing control constructs are shown at the end of the initial block. We can wait until a special condition, such as "when q is equal to 2"

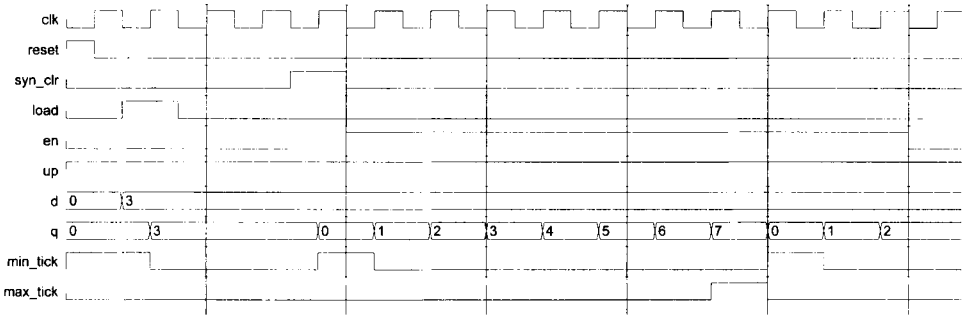


Figure 4.4 Testbench waveform.

```
wait(q==2);
```

or wait until a signal changes, such as

```
wait(min_tick);
```

or wait for an absolute time, such as

```
#(4*T);  // wait for 4T
```

If an input signal is modified after these statements, we need to make sure that the input change does not occur at the rising edge of the clock. An additional

```
@(negedge clk);
```

statement should be added when needed.

We can compile the code and perform simulation. Part of the simulated waveform is shown in Figure 4.4.

4.5 CASE STUDY

After examining several simple circuits, we discuss the design of more sophisticated examples in this section.

4.5.1 LED time-multiplexing circuit

The S3 board has four seven-segment LED displays, each containing seven bars and one small round dot. To reduce the use of FPGA's I/O pins, the S3 board uses a time-multiplexing sharing scheme. In this scheme, the four displays have their individual enable signals but share eight common signals to light the segments. All signals are active low (i.e., enabled when a signal is 0). The schematic of displaying a "3" on the rightmost LED is shown in Figure 4.5. Note that the enable signal (i.e., an) is "1110". This configuration clearly can enable only one display at a time. We can *time-multiplex* the four LED patterns by enabling the four displays in turn, as shown in the simplified timing diagram in Figure 4.6. If the refreshing rate of the enable signal is fast enough, the human eye cannot distinguish the on and off intervals of the LEDs and perceives that all four displays are lit simultaneously. This scheme reduces the number of I/O pins from 32 to 12 (i.e., eight LED segments plus four enable signals) but requires a time-multiplexing circuit. Two variations of the circuit are discussed in the following subsections.

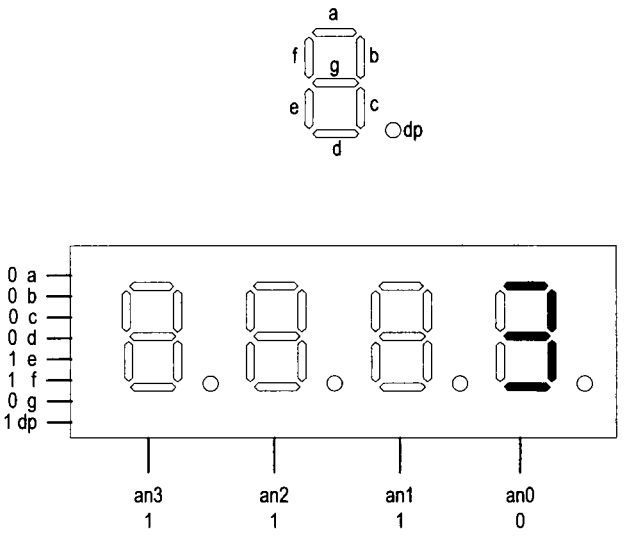


Figure 4.5 Time-multiplexed seven-segment LED display.

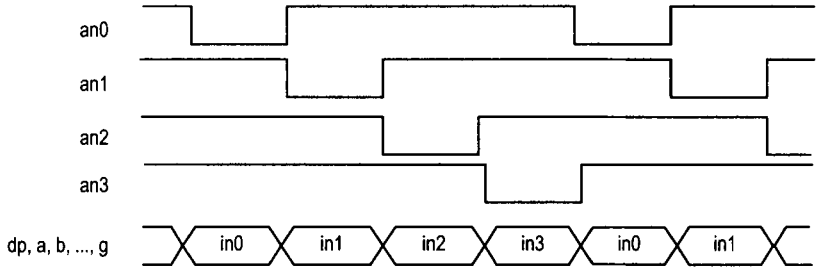


Figure 4.6 Timing diagram of a time-multiplexed seven-segment LED display.

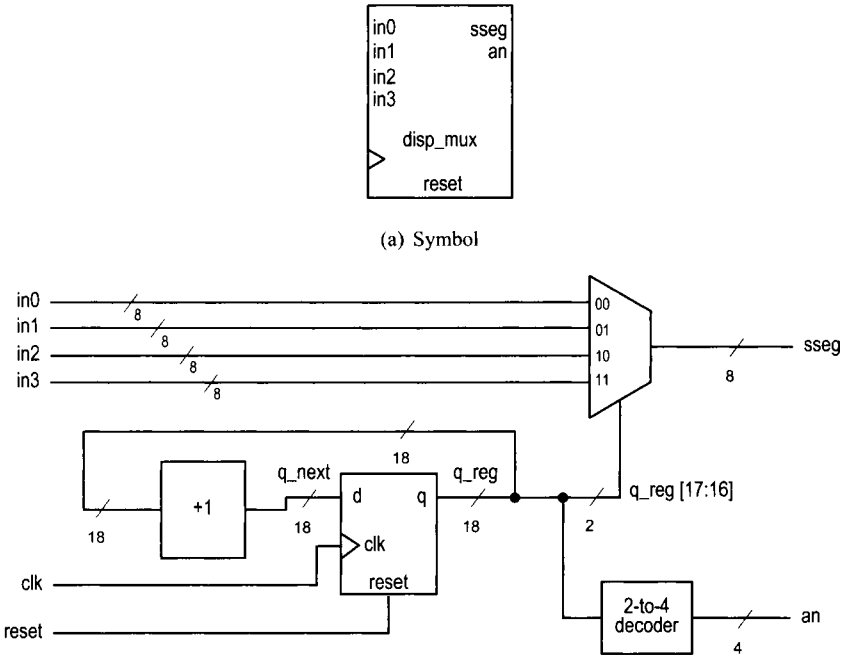


Figure 4.7 Symbol and block diagram of a time-multiplexing circuit.

Time multiplexing with LED patterns The symbol and block diagram of the time-multiplexing circuit are shown in Figure 4.7. It takes four seven-segment LED patterns, `in3`, `in2`, `in1`, and `in0`, and passes them to the output, `sseg`, in accordance with the enable signal.

The refresh rate of the enable signal has to be fast enough to fool our eyes but should be slow enough so that the LEDs can be turned on and off completely. The rate around the range 1000 Hz should work properly. In our design, we use an 18-bit binary counter for this purpose. The two MSBs are decoded to generate the enable signal and are used as the selection signal for multiplexing. The refreshing rate of an individual bit, such as `an[0]`, becomes $\frac{50M}{2^{16}}$ Hz, which is about 800 Hz. The code is shown in Listing 4.13.

Listing 4.13 LED time-multiplexing circuit with LED patterns

```
module disp_mux
   (
    input wire clk, reset,
    input [7:0] in3, in2, in1, in0,
    output reg [3:0] an,    // enable, 1-out-of-4 asserted low
    output reg [7:0] sseg   // led segments
   );

   // constant declaration
   // refreshing rate around 800 Hz (50 MHz/2^16)
   localparam N = 18;
```

```
   // signal declaration
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;

   // N-bit counter
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         q_reg <= 0;
      else
         q_reg <= q_next;

   // next-state logic
   assign q_next = q_reg + 1;

   // 2 MSBs of counter to control 4-to-1 multiplexing
   // and to generate active-low enable signal
   always @*
      case (q_reg[N-1:N-2])
         2'b00:
            begin
               an = 4'b1110;
               sseg = in0;
            end
         2'b01:
            begin
               an =  4'b1101;
               sseg = in1;
            end
         2'b10:
            begin
               an =  4'b1011;
               sseg = in2;
            end
         default:
            begin
               an =  4'b0111;
               sseg = in3;
            end
      endcase

endmodule
```

We use the testing circuit in Figure 4.8 to verify operation of the LED time-multiplexing circuit. It uses four 8-bit registers to store the LED patterns. The registers use the same 8-bit switch as input but are controlled by individual enable signals. When we press a pushbutton, the corresponding register is enabled and the switch pattern is loaded to that register. The code is shown in Listing 4.14.

Listing 4.14 Testing circuit for time multiplexing with LED patterns

```
module disp_mux_test
   (
    input wire clk,
```

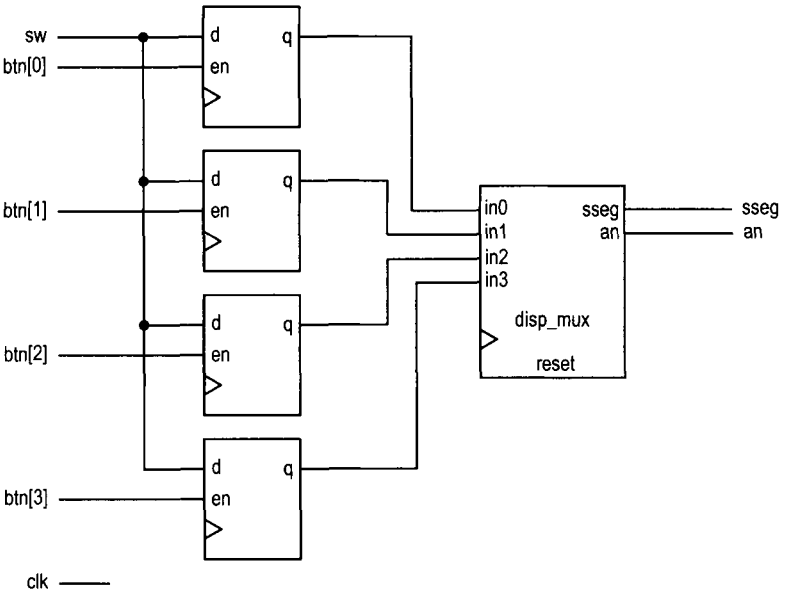


Figure 4.8 LED time-multiplexing testing circuit.

```
    input wire [3:0] btn,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
);

// signal declaration
reg [7:0] d3_reg, d2_reg, d1_reg, d0_reg;

// instantiate 7-seg LED display time-multiplexing module
disp_mux disp_unit
   (.clk(clk), .reset(1'b0), .in0(d0_reg), .in1(d1_reg),
    .in2(d2_reg), .in3(d3_reg), .an(an), .sseg(sseg));

// registers for 4 led patterns
always @(posedge clk)
begin
   if (btn[3])
      d3_reg <= sw;
   if (btn[2])
      d2_reg <= sw;
   if (btn[1])
      d1_reg <= sw;
   if (btn[0])
      d0_reg <= sw;
 end

endmodule
```

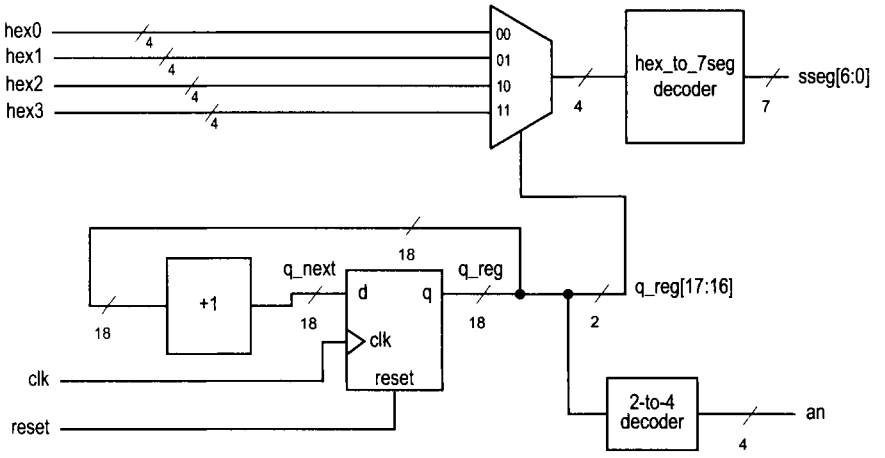


Figure 4.9 Block diagram of a hexadecimal time-multiplexing circuit.

Time multiplexing with hexadecimal digits The most common application of a seven-segment LED is to display a hexadecimal digit. The decoding circuit is discussed in Section 3.9.1. To display four hexadecimal digits with the previous time-multiplexing circuit, four decoding circuits are needed. A better alternative is first to multiplex the hexadecimal digits and then decode the result, as shown in Figure 4.9.

This scheme requires only one decoding circuit and reduces the width of the 4-to-1 multiplexer from 8 bits to 5 bits (i.e., 4 bits for the hexadecimal digit and 1 bit for the decimal point). The code is shown in Listing 4.15. In addition to clock and reset, the input consists of four 4-bit hexadecimal digits, `hex3`, `hex2`, `hex1`, and `hex0`, and four decimal points, which are grouped as one signal, `dp_in`.

Listing 4.15 LED time-multiplexing circuit with hexadecimal digits

```
module disp_hex_mux
   (
    input wire clk, reset,
    input wire [3:0] hex3, hex2, hex1, hex0,  // hex digits
    input wire [3:0] dp_in,                   // 4 decimal points
    output reg [3:0] an,  // enable 1-out-of-4 asserted low
    output reg [7:0] sseg // led segments
   );

   // constant declaration
   // refreshing rate around 800 Hz (50 MHz/2^16)
   localparam N = 18;
   // internal signal declaration
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   reg [3:0] hex_in;
   reg dp;

   // N-bit counter
   // register
   always @(posedge clk, posedge reset)
```

```
      if (reset)
         q_reg <= 0;
      else
         q_reg <= q_next;

   // next-state logic
   assign q_next = q_reg + 1;

   // 2 MSBs of counter to control 4-to-1 multiplexing
   // and to generate active-low enable signal
   always @*
      case (q_reg[N-1:N-2])
         2'b00:
            begin
               an =  4'b1110;
               hex_in = hex0;
               dp = dp_in[0];
            end
         2'b01:
            begin
               an =  4'b1101;
               hex_in = hex1;
               dp = dp_in[1];
            end
         2'b10:
            begin
               an =  4'b1011;
               hex_in = hex2;
               dp = dp_in[2];
            end
         default:
            begin
               an =  4'b0111;
               hex_in = hex3;
               dp = dp_in[3];
            end
      endcase

   // hex to seven-segment led display
   always @*
   begin
      case(hex_in)
         4'h0: sseg[6:0] = 7'b0000001;
         4'h1: sseg[6:0] = 7'b1001111;
         4'h2: sseg[6:0] = 7'b0010010;
         4'h3: sseg[6:0] = 7'b0000110;
         4'h4: sseg[6:0] = 7'b1001100;
         4'h5: sseg[6:0] = 7'b0100100;
         4'h6: sseg[6:0] = 7'b0100000;
         4'h7: sseg[6:0] = 7'b0001111;
         4'h8: sseg[6:0] = 7'b0000000;
         4'h9: sseg[6:0] = 7'b0000100;
         4'ha: sseg[6:0] = 7'b0001000;
```

```
        4'hb: sseg[6:0] = 7'b1100000;
        4'hc: sseg[6:0] = 7'b0110001;
        4'hd: sseg[6:0] = 7'b1000010;
        4'he: sseg[6:0] = 7'b0110000;
        default: sseg[6:0] = 7'b0111000;   //4'hf
    endcase
    sseg[7] = dp;
end

endmodule
```

To verify operation of this circuit, we define the 8-bit switch as two 4-bit unsigned numbers, add the two numbers, and show the two numbers and their sum on the four-digit seven-segment LED display. The code is shown in Listing 4.16.

Listing 4.16 Testing circuit for time multiplexing with hexadecimal digits

```
module hex_mux_test
   (
    input wire clk,
    input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   wire [3:0] a, b;
   wire [7:0] sum;

   // instantiate 7-seg LED display module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(1'b0),
       .hex3(sum[7:4]), .hex2(sum[3:0]), .hex1(b), .hex0(a),
       .dp_in(4'b1011), .an(an), .sseg(sseg));

   // adder
   assign a = sw[3:0];
   assign b = sw[7:4];
   assign sum = {4'b0,a} + {4'b0,b};

endmodule
```

Simulation consideration Many sequential circuit examples in the book operate at a relatively slow rate, as does the enable pulse of the LED time-multiplexing circuit. This can be done by generating a single-clock enable tick from a counter. An 18-bit counter is used in this circuit:

```
localparam N = 18;
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
. . .
assign q_next = q_reg + 1;
```

Because of the counter's size, simulating this type of circuit consumes a significant amount of computation time (i.e., 2^{18} clock cycles for one iteration). Since our main interest is in

the multiplexing part of the code, most simulation time is wasted. It is more efficient to use a smaller counter in simulation. We can do this by modifying the constant statement

```
localparam N = 4;
```

when constructing the testbench. This requires only 2^4 clock cycles for one iteration and allows us to better exercise and observe the key operations.

Instead of using a constant and modifying code between simulation and synthesis, an alternative is to define N as a parameter. During instantiation, we can assign different values for simulation and synthesis.

4.5.2 Stopwatch

We consider the design of a stopwatch in this subsection. The watch displays the time in three decimal digits, and counts from 00.0 to 99.9 seconds and wraps around. It contains a synchronous clear signal, clr, which returns the count to 00.0, and an enable signal, go, which enables and suspends the counting. This design is basically a BCD (binary-coded decimal) counter, which counts in BCD format. In this format, a decimal number is represented by a sequence of 4-bit BCD digits. For example, 139_{10} is represented as "0001 0011 1001" and the next number in sequence is 140_{10}, which is represented as "0001 0100 0000".

Since the S3 board has a 50-MHz clock, we first need a mod-5,000,000 counter that generates a one-clock-cycle tick every 0.1 second. The tick is then used to enable counting of the three-digit BCD counter.

Design I Our first design of the BCD counter uses a cascading structure of three decade (i.e., mod-10) counters, representing counts of 0.1, 1, and 10 seconds, respectively. The decade counter has an enable signal and generates a one-clock-cycle tick when it reaches 9. We can use these signals to "hook" the three counters. For example, the 10-second counter is enabled only when the enable tick of the mod-5,000,000 counter is asserted and both the 0.1- and 1-second counters are 9. The code is shown in Listing 4.17.

Listing 4.17 Cascading description for a stopwatch

```
module stop_watch_cascade
   (
    input wire clk,
    input wire go, clr,
    output wire [3:0] d2, d1, d0
   );

   // declaration
   localparam  DVSR = 5000000;
   reg [22:0] ms_reg;
   wire [22:0] ms_next;
   reg [3:0] d2_reg, d1_reg, d0_reg;
   wire [3:0] d2_next, d1_next, d0_next;
   wire d1_en, d2_en, d0_en;
   wire ms_tick, d0_tick, d1_tick;

   // body
   // register
   always @(posedge clk)
```

```
20    begin
         ms_reg <= ms_next;
         d2_reg <= d2_next;
         d1_reg <= d1_next;
         d0_reg <= d0_next;
25    end

      // next-state logic
      // 0.1 sec tick generator: mod-5000000
      assign ms_next = (clr || (ms_reg==DVSR && go)) ? 4'b0 :
30                     (go) ? ms_reg + 1 :
                              ms_reg;
      assign ms_tick = (ms_reg==DVSR) ? 1'b1 : 1'b0;
      // 0.1 sec counter
      assign d0_en = ms_tick;
35    assign d0_next = (clr || (d0_en && d0_reg==9)) ? 4'b0 :
                       (d0_en) ? d0_reg + 1 :
                                 d0_reg;
      assign d0_tick = (d0_reg==9) ? 1'b1 : 1'b0;
      // 1 sec counter
40    assign d1_en = ms_tick & d0_tick;
      assign d1_next = (clr || (d1_en && d0_reg==9)) ? 4'b0 :
                       (d1_en) ? d1_reg + 1 :
                                 d1_reg;
      assign d1_tick = (d1_reg==9) ? 1'b1 : 1'b0;
45
      // 10 sec counter
      assign d2_en = ms_tick & d0_tick & d1_tick;
      assign d2_next = (clr || (d2_en && d2_reg==9)) ? 4'b0 :
                       (d2_en) ? d2_reg + 1 :
50                               d2_reg;

      // output logic
      assign d0 = d0_reg;
      assign d1 = d1_reg;
55    assign d2 = d2_reg;

   endmodule
```

Note that all registers are controlled by the same clock signal. This example illustrates how to use a one-clock-cycle enable tick to maintain synchronicity. An inferior approach is to use the output of the lower counter as the clock signal for the next stage. Although it may appear to be simpler, it violates the synchronous design principle and is a very poor practice.

Design II An alternative for the three-digit BCD counter is to describe the entire structure in a nested if statement. The nested conditions indicate that the counter reaches .9, 9.9, and 99.9 seconds. The code is shown in Listing 4.18.

Listing 4.18 Nested if-statement description for a stopwatch

```
module stop_watch_if
   (
    input wire clk,
```

```
    input wire go, clr,
    output wire [3:0] d2, d1, d0
);

// declaration
localparam  DVSR = 5000000;
reg [22:0] ms_reg;
wire [22:0] ms_next;
reg [3:0] d2_reg, d1_reg, d0_reg;
reg [3:0] d2_next, d1_next, d0_next;
wire ms_tick;

// body
// register
always @(posedge clk)
begin
   ms_reg <= ms_next;
   d2_reg <= d2_next;
   d1_reg <= d1_next;
   d0_reg <= d0_next;
end

// next-state logic
// 0.1 sec tick generator: mod-5000000
assign ms_next = (clr || (ms_reg==DVSR && go)) ? 4'b0 :
                 (go) ? ms_reg + 1 :
                        ms_reg;
assign ms_tick = (ms_reg==DVSR) ? 1'b1 : 1'b0;
// 3-digit bcd counter
always @*
begin
   // default: keep the previous value
   d0_next = d0_reg;
   d1_next = d1_reg;
   d2_next = d2_reg;
   if (clr)
      begin
         d0_next = 4'b0;
         d1_next = 4'b0;
         d2_next = 4'b0;
      end
   else if (ms_tick)
      if (d0_reg != 9)
         d0_next = d0_reg + 1;
      else                  // reach XX9
         begin
            d0_next = 4'b0;
            if (d1_reg != 9)
               d1_next = d1_reg + 1;
            else        // reach X99
               begin
                  d1_next = 4'b0;
                  if (d2_reg != 9)
```

```
                        d2_next = d2_reg + 1;
                     else // reach 999
                        d2_next = 4'b0;
60                end
            end
    end

    // output logic
65  assign d0 = d0_reg;
    assign d1 = d1_reg;
    assign d2 = d2_reg;

endmodule
```

Verification circuit To verify operation of the stopwatch, we can combine it with the previous hexadecimal LED time-multiplexing circuit to display the output of the watch. The code is shown in Listing 4.19. Note that the first digit of the LED is assigned to 0 and the go and `clr` signals are mapped to two pushbuttons of the S3 board.

Listing 4.19 Testing circuit for a stopwatch

```
module stop_watch_test
   (
    input wire clk,
    input wire [1:0] btn,
5   output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
10 wire [3:0]  d2, d1, d0;

   // instantiate 7-seg LED display module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(1'b0),
15     .hex3(4'b0), .hex2(d2), .hex1(d1), .hex0(d0),
       .dp_in(4'b1101), .an(an), .sseg(sseg));

   // instantiate stopwatch
   stop_watch_if counter_unit
20    (.clk(clk), .go(btn[1]), .clr(btn[0]),
       .d2(d2), .d1(d1), .d0(d0) );

endmodule
```

4.5.3 FIFO buffer

A FIFO (first-in-first-out) buffer is an "elastic" storage between two subsystems, as shown in the conceptual diagram of Figure 4.10. It has two control signals, `wr` and `rd`, for write and read operations. When `wr` is asserted, the input data is written into the buffer. The read operation is somewhat misleading. The head of the FIFO buffer is normally always available and thus can be read at any time. The `rd` signal actually acts like a "remove"

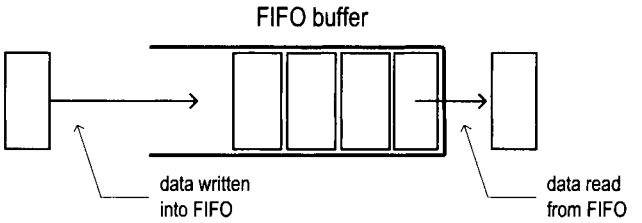


Figure 4.10 Conceptual diagram of a FIFO buffer.

signal. When it is asserted, the first item (i.e., head) of the FIFO buffer is removed and the next item becomes available.

FIFO buffer is a critical component in many applications and the optimized implementation can be quite complex. In this subsection, we introduce a simple, genuine circular-queue-based design. More efficient, device-specific implementation can be found in the Xilinx literature.

Circular-queue-based implementation One way to implement a FIFO buffer is to add a control circuit to a register file. The registers in the register file are arranged as a circular queue with two pointers. The *write pointer* points to the head of the queue, and the *read pointer* points to the tail of the queue. The pointer advances one position for each write or read operation. The operation of an eight-word circular queue is shown in Figure 4.11.

A FIFO buffer usually contains two status signals, `full` and `empty`, to indicate that the FIFO is full (i.e., cannot be written) and empty (i.e., cannot be read), respectively. One of the two conditions occurs when the read pointer is equal to the write pointer, as shown in Figure 4.11(a), (f), and (i). The most difficult design task of the controller is to derive a mechanism to distinguish the two conditions. One scheme is to use two FFs to keep track of the empty and full statuses. The FFs are set to 1 and 0 during system initialization and then modified in each clock cycle according to the values of the `wr` and `rd` signals. The code is shown in Listing 4.20.

Listing 4.20 FIFO buffer

```
module fifo
   #(
    parameter B=8, // number of bits in a word
              W=4  // number of address bits
   )
   (
    input wire clk, reset,
    input wire rd, wr,
    input wire [B-1:0] w_data,
    output wire empty, full,
    output wire [B-1:0] r_data
   );

   //signal declaration
   reg [B-1:0] array_reg [2**W-1:0];  // register array
   reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
   reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
   reg full_reg, empty_reg, full_next, empty_next;
```

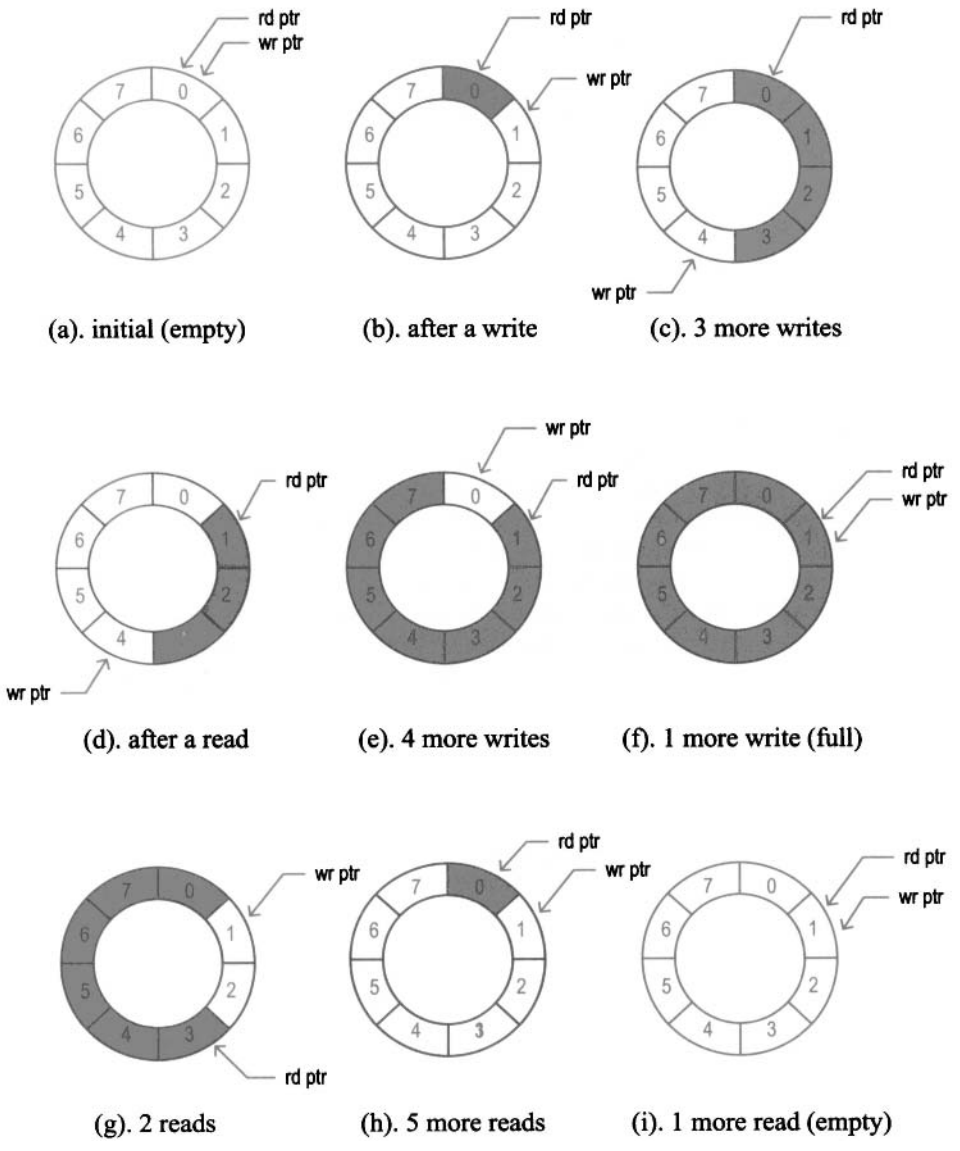


Figure 4.11 FIFO buffer based on a circular queue.

```
   wire wr_en;
20
   // body
   // register file write operation
   always @(posedge clk)
      if (wr_en)
25       array_reg[w_ptr_reg] <= w_data;
   // register file read operation
   assign r_data = array_reg[r_ptr_reg];
   // write enabled only when FIFO is not full
   assign wr_en = wr & ~full_reg;
30
   // fifo control logic
   // register for read and write pointers
   always @(posedge clk, posedge reset)
      if (reset)
35       begin
            w_ptr_reg <= 0;
            r_ptr_reg <= 0;
            full_reg <= 1'b0;
            empty_reg <= 1'b1;
40       end
      else
         begin
            w_ptr_reg <= w_ptr_next;
            r_ptr_reg <= r_ptr_next;
45          full_reg <= full_next;
            empty_reg <= empty_next;
         end

   // next-state logic for read and write pointers
50 always @*
   begin
      // successive pointer values
      w_ptr_succ = w_ptr_reg + 1;
      r_ptr_succ = r_ptr_reg + 1;
55    // default: keep old values
      w_ptr_next = w_ptr_reg;
      r_ptr_next = r_ptr_reg;
      full_next = full_reg;
      empty_next = empty_reg;
60    case ({wr, rd})
         // 2'b00:  no op
         2'b01: // read
            if (~empty_reg) // not empty
               begin
65                r_ptr_next = r_ptr_succ;
                  full_next = 1'b0;
                  if (r_ptr_succ==w_ptr_reg)
                     empty_next = 1'b1;
               end
70       2'b10: // write
            if (~full_reg) // not full
```

```
                begin
                   w_ptr_next = w_ptr_succ;
                   empty_next = 1'b0;
                   if (w_ptr_succ==r_ptr_reg)
                      full_next = 1'b1;
                end
          2'b11: // write and read
             begin
                w_ptr_next = w_ptr_succ;
                r_ptr_next = r_ptr_succ;
             end
       endcase
    end

    // output
    assign full = full_reg;
    assign empty = empty_reg;

endmodule
```

The code is divided into a register file and a FIFO controller. The controller consists of two pointers and two status FFs. Its next-state logic examines the `wr` and `rd` signals and takes actions accordingly. For example, let us consider the "10" case, which implies that only a write operation occurs. The status FF is checked first to ensure that the buffer is not full. If this condition is met, we advance the write pointer by one position and clear the empty status FF. Storing one extra word to the buffer may make it full. This happens if the new write pointer "catches" the read pointer, which is expressed by the `w_ptr_succ==r_ptr_reg` expression.

Verification circuit The verification circuit examines the operation of a 2^4-by-3 FIFO buffer. We use three switches to generate the input data and use two buttons for the `wr` and `rd` signals. The 3-bit readout and the `full` and `empty` status signals are displayed in five discrete LEDs. Because of bounces of the mechanical contact, a debouncing circuit is needed to generate a clean one-clock-cycle tick. The debouncing module, named `debounce`, is discussed in Section 6.2.1 but for now can be treated as a predesigned module. The original button inputs are `btn[0]` and `btn[1]`, and the debounced signals are `db_btn[0]` and `db_btn[1]`. The code is shown in Listing 4.21.

Listing 4.21 Testing circuit for a FIFO buffer

```
module fifo_test
   (
    input wire clk, reset,
    input wire [1:0] btn,
    input wire [2:0] sw,
    output wire [7:0] led
   );

   // signal declaration
   wire [1:0] db_btn;

   // debounce circuit for btn[0]
   debounce btn_db_unit0
```

```
      (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(db_btn[0]));
   // debounce circuit for btn[1]
   debounce btn_db_unit1
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(db_btn[1]));
   // instantiate a 2^2-by-3 fifo
   fifo #(.B(3), .W(2)) fifo_unit
      (.clk(clk), .reset(reset),
      .rd(db_btn[0]), .wr(db_btn[1]), .w_data(sw),
      .r_data(led[2:0]), .full(led[7]), .empty(led[6]));
   // disable unused leds
   assign led[5:3] = 3'b000;

endmodule
```

4.6 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 3.

4.7 SUGGESTED EXPERIMENTS

4.7.1 Programmable square-wave generator

A programmable square-wave generator is a circuit that can generate a square wave with variable on (i.e., logic 1) and off (i.e., logic 0) intervals. The durations of the intervals are specified by two 4-bit control signals, `m` and `n`, which are interpreted as unsigned integers. The on and off intervals are `m`$*100$ ns and `n`$*100$ ns, respectively (recall that the period of the S3 onboard oscillator is 20 ns). Design a programmable square-wave generator circuit. The circuit should be completely synchronous. We need a logic analyzer or oscilloscope to verify its operation.

4.7.2 PWM and LED dimmer

The duty cycle of a square wave is defined as the percentage of the on interval (i.e., logic 1) in a period. A PWM (pulse width modulation) circuit can generate an output with variable duty cycles. For a PWM with 4-bit resolution, a 4-bit control signal, `w`, specifies the duty cycle. The `w` signal is interpreted as an unsigned integer and the duty cycle is $\frac{w}{16}$.

1. Design a PWM circuit with 4-bit resolution and verify its operation using a logic analyzer or oscilloscope.
2. Modify the LED time-multiplexing circuit to include the PWM circuit for the an signal. The PWM circuit specifies the percentage of time that the LED display is on. We can control the perceived brightness by changing the duty cycle. Verify the circuit's operation by observing 1 bit of an on a logic analyzer or oscilloscope.
3. Replace the LED time-multiplexing circuit of Listing 4.19 with the new design and use the lower 4 bits of the 8-bit switch to control the duty cycle. Verify operation of the circuit. It may be necessary to go to a dark area to see the effect of dimming.

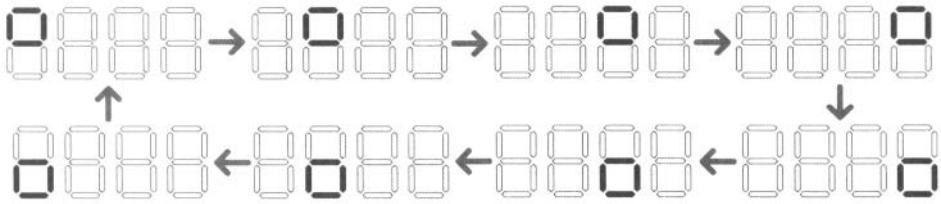


Figure 4.12 Pattern for Experiment 4.7.3.

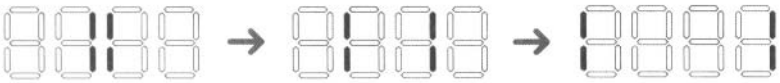


Figure 4.13 Pattern for Experiment 4.7.4.

4.7.3 Rotating square circuit

In a seven-segment LED display, a square pattern can be created by enabling the a, b, f, and g segments or the c, d, e, and g segments. We want to design a circuit that circulates the square patterns in the four-digit seven-segment LED display. The clockwise circulating pattern is shown in Figure 4.12. The circuit should have an input, `en`, which enables or pauses the circulation, and an input, `cw`, which specifies the direction (i.e., clockwise or counterclockwise) of the circulation.

Design the circuit and verify its operation on the prototyping board. Make sure that the circulation rate is slow enough for visual inspection.

4.7.4 Heartbeat circuit

We want to create a "heartbeat" for the prototyping board. It repeats the simple pattern in the four-digit seven-segment display, as shown in Figure 4.13, at a rate of 72 Hz. Design the circuit and verify its operation on the prototyping board.

4.7.5 Rotating LED banner circuit

The prototyping board has a four-digit seven-segment LED display, and thus only four symbols can be displayed at a time. We can show more information if the data is rotated and moved continuously. For example, assume that the message is 10 digits (i.e., "0123456789"). The display can show the message as "0123", "1234", "2345", ... , "6789", "7890", ... , "0123". The circuit should have an input, `en`, which enables or pauses the rotation, and an input, `dir`, which specifies the direction (i.e., rotate left or right).

Design the circuit and verify its operation on the prototyping board. Make sure that the rotation rate is slow enough for visual inspection.

4.7.6 Enhanced stopwatch

Modify the stopwatch with the following extensions:

- Add an additional signal, up, to control the direction of counting. The stopwatch counts up when the up signal is asserted and counts down otherwise.
- Add a minute digit to the display. The LED display format should be like `M.SS.D`, where `D` represents 0.1 second and its range is between 0 and 9, `SS` represents seconds and its range is between 00 and 59, and `M` represents minutes and its range is between 0 and 9.

Design the new stopwatch and verify its operation with a testing circuit.

4.7.7 Stack

A stack is a last-in-first-out buffer in which the last stored data is retrieved first. Storing a data word to a stack is known as a *push* operation, and retrieving a data word from a stack is known as a *pop* operation. The I/O signals of a stack are similar to those of a FIFO buffer except that we generally use the `push` and `pop` signals in place of the `wr` and `rd` signals. Design a stack using a register file and verify its operation with a testing circuit similar to the one in Listing 4.21.

CHAPTER 5

FSM

5.1 INTRODUCTION

An FSM (finite state machine) is used to model a system that transits among a finite number of internal states. The transitions depend on the current state and external input. Unlike a regular sequential circuit, the state transitions of an FSM do not exhibit a simple, repetitive pattern. Its next-state logic is usually constructed from scratch and is sometimes known as "random" logic. This is different from the next-state logic of a regular sequential circuit, which is composed mostly of "structured" components, such as incrementors and shifters.

In this chapter, we provide an overview of the basic characteristics and representation of FSMs and discuss the derivation of HDL codes. In practice, the main application of an FSM is to act as the controller of a large digital system, which examines the external commands and status and activates proper control signals to control operation of a *data path*, which is usually composed of regular sequential components. This is known as an FSMD (finite state machine with data path) and is discussed in Chapter 6.

5.1.1 Mealy and Moore outputs

The basic block diagram of an FSM is the same as that of a regular sequential circuit and is repeated in Figure 5.1. It consists of a state register, next-state logic, and output logic. An FSM is known as a *Moore machine* if the output is only a function of state, and is known as a *Mealy machine* if the output is a function of state and external input. Both types of output may exist in a complex FSM, and we simply refer to it as containing a Moore

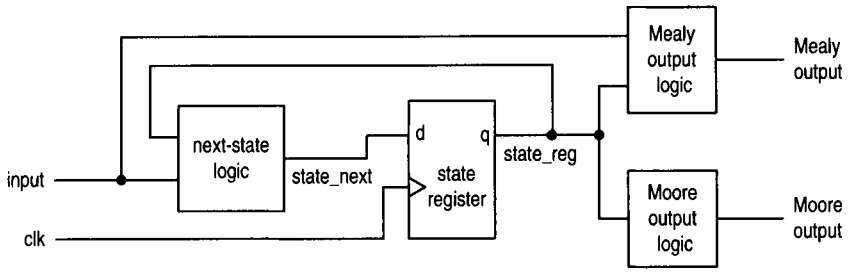


Figure 5.1 Block diagram of a synchronous FSM.

output and a Mealy output. The Moore and Mealy outputs are similar but not identical. Understanding their subtle differences is the key for controller design. The example in Section 5.3.1 illustrates the behaviors and constructions of the two types of outputs.

5.1.2 FSM representation

An FSM is usually specified by an abstract *state diagram* or *ASM chart* (algorithmic state machine chart), both capturing the FSM's input, output, states, and transitions in a graphical representation. The two representations provide the same information. The FSM representation is more compact and better for simple applications. The ASM chart representation is somewhat like a flowchart and is more descriptive for applications with complex transition conditions and actions.

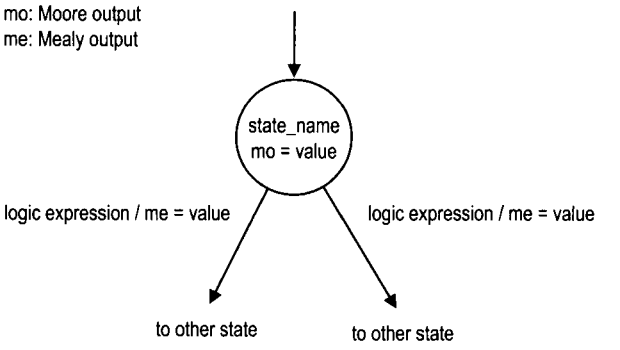
State diagram A state diagram is composed of *nodes*, which represent states and are drawn as circles, and annotated *transitional arcs*. A single node and its transition arcs are shown in Figure 5.2(a). A logic expression expressed in terms of input signals is associated with each transition arc and represents a specific condition. The arc is taken when the corresponding expression is evaluated `true`.

The Moore output values are placed inside the circle since they depend only on the current state. The Mealy output values are associated with the conditions of transition arcs since they depend on the current state and external input. To reduce clutter in the diagram, only asserted output values are listed. The output signal takes the default (i.e., unasserted) value otherwise.

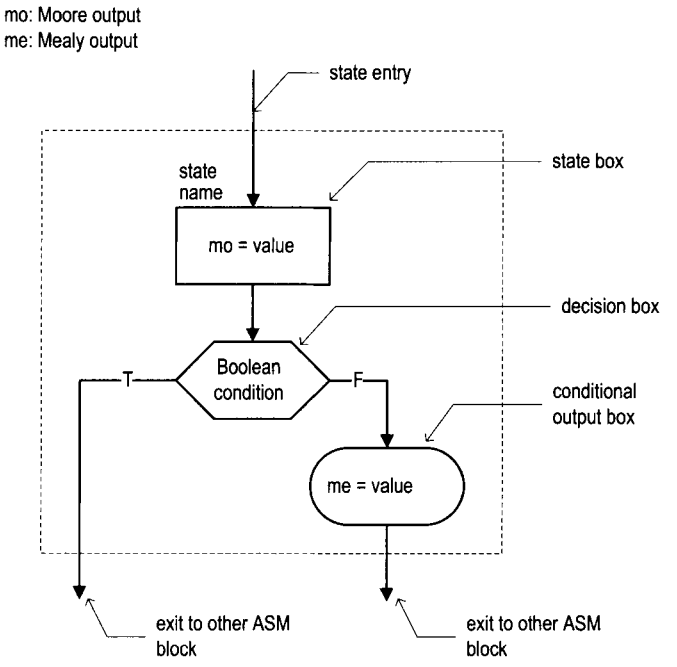
A representative state diagram is shown in Figure 5.3(a). The FSM has three states, two external input signals (i.e., `a` and `b`), one Moore output signal (i.e., `y1`), and one Mealy output signal (i.e., `y0`). The `y1` signal is asserted when the FSM is in the `s0` or `s1` state. The `y0` signal is asserted when the FSM is in the `s0` state and the `a` and `b` signals are "11".

ASM chart An ASM chart is composed of a network of ASM blocks. An *ASM block* consists of one *state box* and an optional network of *decision boxes* and *conditional output boxes*. A representative ASM block is shown in Figure 5.2(b).

A state box represents a state in an FSM, and the asserted Moore output values are listed inside the box. Note that it has only one exit path. A decision box tests the input condition and determines which exit path to take. It has two exit paths, labeled `T` and `F`, which correspond to the true and false values of the condition. A conditional output box lists asserted Mealy output values and is usually placed after a decision box. It indicates that the listed output signal can be activated only when the corresponding condition in the decision box is met.

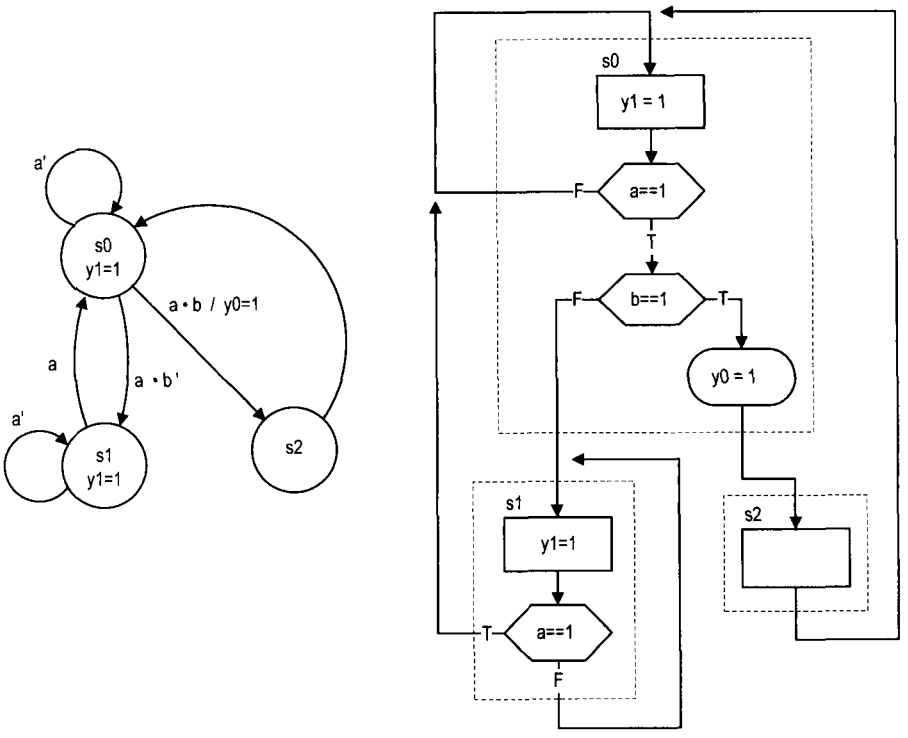


(a) Node



(b) ASM block

Figure 5.2 Symbol of a state.



(a) State diagram (b) ASM chart

Figure 5.3 Example of an FSM.

A state diagram can easily be converted to an ASM chart, and vice versa. The corresponding ASM chart of the previous FSM state diagram is shown in Figure 5.3(b).

5.2 FSM CODE DEVELOPMENT

The procedure of developing code for an FSM is similar to that of a regular sequential circuit. We first separate the state register and then derive the code for the combinational next-state logic and output logic. The main difference is the next-state logic. For an FSM, the code for the next-state logic follows the flow of a state diagram or ASM chart.

For clarity and flexibility, we use symbolic constants to represent the FSM's states. For examples, the three states in Figure 5.3 can be defined as

```
localparam [1:0] s0 = 2'b00,
                 s1 = 2'b01,
                 s2 = 2'b10;
```

During synthesis, software usually can recognize the FSM structure and may map these symbolic constants to different binary representations (e.g., one-hot codes), a process known as *state assignment*.

The complete code of the FSM is shown in Listing 5.1. It consists of segments for the state register, next-state logic, Moore output logic, and Mealy output logic.

Listing 5.1 FSM example

```
module fsm_eg_mult_seg
   (
    input wire  clk, reset,
    input wire  a, b,
    output wire y0, y1
   );

   // symbolic state declaration
   localparam [1:0] s0 = 2'b00,
                    s1 = 2'b01,
                    s2 = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= s0;
       else
          state_reg <= state_next;

   // next-state logic
   always @*
      case (state_reg)
         s0: if (a)
               if (b)
                  state_next = s2;
               else
                  state_next = s1;
             else
                state_next = s0;
         s1: if (a)
               state_next = s0;
             else
               state_next = s1;
         s2: state_next = s0;
         default: state_next = s0;
      endcase

   // Moore output logic
   assign y1 = (state_reg==s0) || (state_reg==s1);

   // Mealy output logic
   assign y0 = (state_reg==s0) & a & b;

endmodule
```

The key part is the next-state logic. It uses a case statement with the `state_reg` signal as the selection expression. The next state (i.e., `state_next` signal) is determined by the current state (i.e., `state_reg`) and external input. The code for each state basically follows the activities inside each ASM block of Figure 5.3(b).

An alternative code is to merge next-state logic and output logic into a single combinational block, as shown in Listing 5.2.

Listing 5.2 FSM with merged combinational logic

```
module fsm_eg_2_seg
   (
    input wire  clk, reset,
    input wire  a, b,
    output reg y0, y1
   );

   // symbolic state declaration
   localparam [1:0] s0 = 2'b00,
                    s1 = 2'b01,
                    s2 = 2'b10;
   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= s0;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
   begin
      state_next = state_reg; // default next state: the same
      y1 = 1'b0;              // default output: 0
      y0 = 1'b0;              // default output: 0
      case (state_reg)
         s0: begin
                y1 = 1'b1;
                if (a)
                   if (b)
                      begin
                         state_next = s2;
                         y0 = 1'b1;
                      end
                   else
                      state_next = s1;
             end
         s1: begin
               y1 = 1'b1;
               if (a)
                 state_next = s0;
             end
         s2: state_next = s0;
         default: state_next = s0;
      endcase
   end
endmodule
```

Note that the default output values are listed at the beginning of the code.

The code for the next-state logic and output logic follows the ASM chart closely. Once a detailed state diagram or ASM chart is derived, converting an FSM to HDL code is almost a mechanical procedure. Listings 5.1 and 5.2 can serve as templates for this purpose.

Xilinx ISE includes a utility program called *StateCAD*, which allows a user to draw a state diagram in graphical format. The program then converts the state diagram to HDL code. It is a good idea to try it first with a few simple examples to see whether the generated code and its style are satisfactory, particularly for the output signals. **Xilinx specific**

5.3 DESIGN EXAMPLES

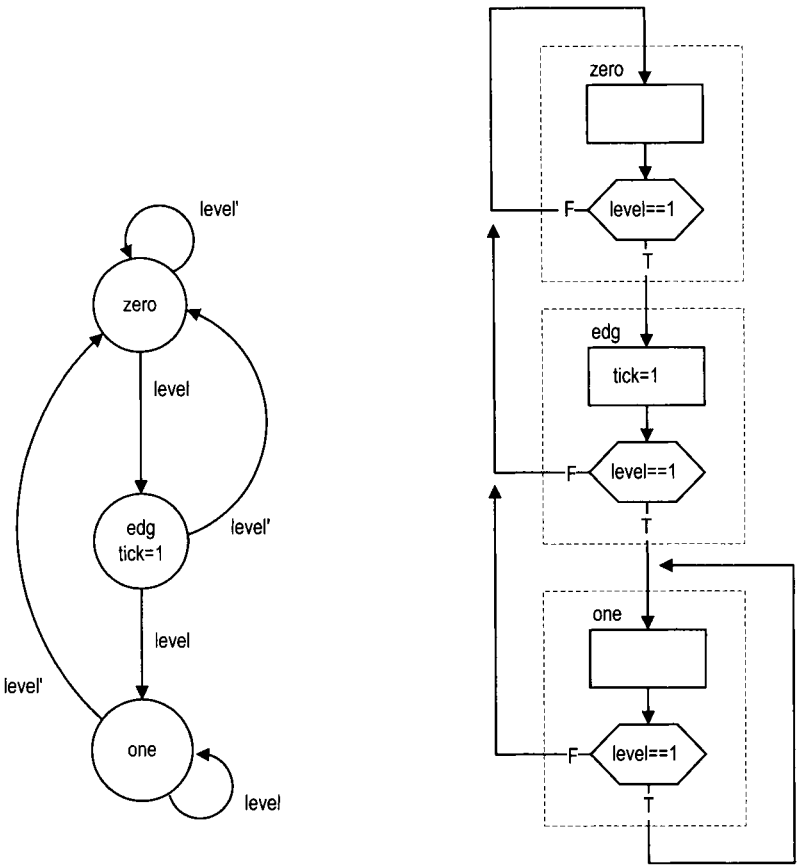
5.3.1 Rising-edge detector

The rising-edge detector is a circuit that generates a short one-clock-cycle tick when the input signal changes from 0 to 1. It is usually used to indicate the onset of a slow time-varying input signal. We design the circuit using both Moore and Mealy machines, and compare their differences.

Moore-based design The state diagram and ASM chart of a Moore machine–based edge detector are shown in Figure 5.4. The `zero` and `one` states indicate that the input signal has been 0 and 1 for a while. The rising edge occurs when the input changes to 1 in the `zero` state. The FSM moves to the `edg` state and the output, `tick`, is asserted in this state. A representative timing diagram is shown at the middle of Figure 5.5. The code is shown in Listing 5.3.

Listing 5.3 Moore machine–based edge detector

```
module edge_detect_moore
   (
    input wire clk, reset,
    input wire level,
    output reg tick
   );

   // symbolic state declaration
   localparam [1:0]
      zero = 2'b00,
      edg = 2'b01,
      one = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;

   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
```



(a) State diagram (b) ASM chart

Figure 5.4 Edge detector based on a Moore machine.

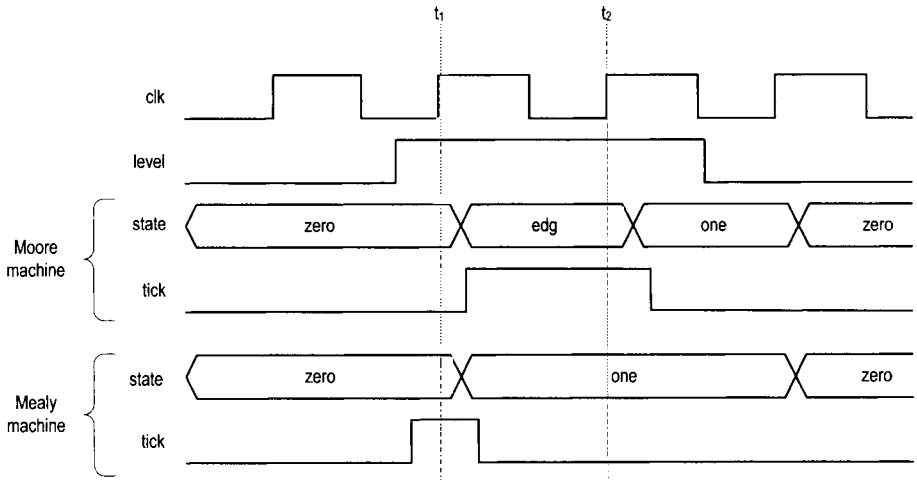


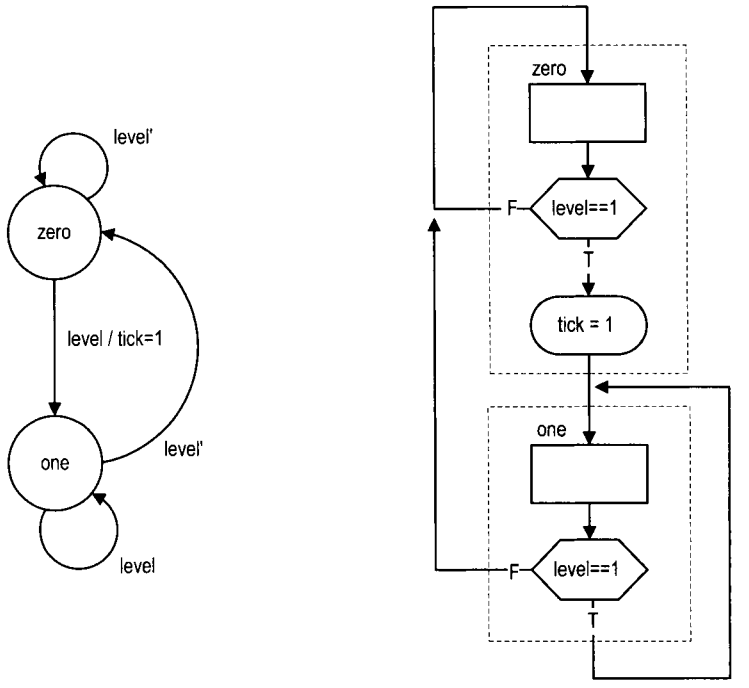
Figure 5.5 Timing diagram of two edge detectors.

```
   begin
      state_next = state_reg;   // default state: the same
      tick = 1'b0;              // default output: 0
      case (state_reg)
30       zero:
            if (level)
               state_next = edg;
         edg:
            begin
35             tick = 1'b1;
               if (level)
                  state_next = one;
               else
                  state_next = zero;
40          end
         one:
            if (~level)
               state_next = zero;
         default: state_next = zero;
45    endcase
   end
endmodule
```

Mealy-based design The state diagram and ASM chart of a Mealy machine–based edge detector are shown in Figure 5.6. The zero and one states have a similar meaning. When the FSM is in the zero state and the input changes to 1, the output is asserted immediately. The FSM moves to the one state at the rising edge of the next clock and the output is deasserted. A representative timing diagram is shown at the bottom of Figure 5.5. Note that due to the propagation delay, the output signal is still asserted at the rising edge of the next clock (i.e., at t_1). The code is shown in Listing 5.4.

Listing 5.4 Mealy machine–based edge detector

```
module edge_detect_mealy
   (
    input wire  clk, reset,
    input wire  level,
5   output reg tick
   );

   // symbolic state declaration
   localparam zero = 1'b0,
10            one = 1'b1;

   // signal declaration
   reg state_reg, state_next;

15 // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
20        state_reg <= state_next;
```



(a) State diagram (b) ASM chart

Figure 5.6 Edge detector based on a Mealy machine.

```
    // next-state logic and output logic
    always @*
    begin
25      state_next = state_reg;   // default state: the same
        tick = 1'b0;              // default output: 0
        case (state_reg)
           zero:
              if (level)
30               begin
                    tick = 1'b1;
                    state_next = one;
                 end
           one:
35            if (~level)
                 state_next = zero;
           default: state_next = zero;
        endcase
    end
40
    endmodule
```

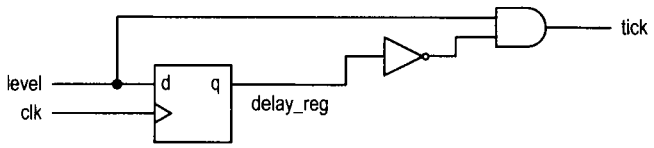


Figure 5.7 Gate-level implementation of an edge detector.

Direct implementation Since the transitions of the edge detector circuit are very simple, it can be implemented without using an FSM. We include this implementation for comparison purposes. The circuit diagram is shown in Figure 5.7. It can be interpreted that the output is asserted only when the current input is 1 and the previous input, which is stored in the register, is 0. The corresponding code is shown in Listing 5.5.

Listing 5.5 Gate-level implementation of an edge detector

```
module edge_detect_gate
   (
    input wire  clk, reset,
    input wire  level,
    output wire tick
   );

   // signal declaration
   reg delay_reg;

   // delay register
    always @(posedge clk, posedge reset)
       if (reset)
          delay_reg <= 1'b0;
       else
          delay_reg <= level;

   // decoding logic
   assign tick = ~delay_reg & level;

endmodule
```

Although the descriptions in Listings 5.4 and 5.5 appear to be very different, they describe the same circuit. The circuit diagram can be derived from the FSM if we assign 0 and 1 to the zero and one states.

Comparison Whereas both Moore machine– and Mealy machine–based designs can generate a short tick at the rising edge of the input signal, there are several subtle differences. The Mealy machine–based design requires fewer states and responds faster, but the width of its output may vary and input glitches may be passed to the output.

The choice between the two designs depends on the subsystem that uses the output signal. Most of the time the subsystem is a synchronous system that shares the same clock signal. Since the FSM's output is sampled only at the rising edge of the clock, the width and glitches do not matter as long as the output signal is stable around the edge. Note that the Mealy output signal is available for sampling at t_1, which is one clock cycle faster than

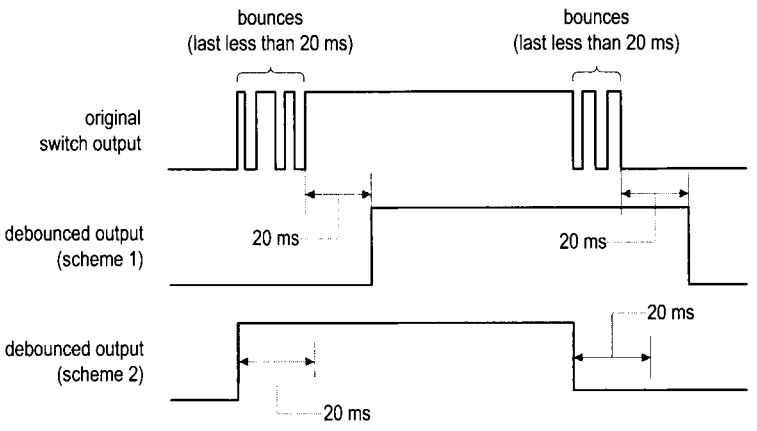


Figure 5.8 Original and debounced waveforms.

the Moore output, which is available at t_2. Therefore, the Mealy machine–based circuit is preferred for this type of application.

5.3.2 Debouncing circuit

The slide and pushbutton switches on the prototyping board are mechanical devices. When pressed, the switch may bounce back and forth a few times before settling down. The bounces lead to glitches in the signal, as shown at the top of Figure 5.8. The bounces usually settle within 20 ms. The purpose of a debouncing circuit is to filter out the glitches associated with switch transitions. The debounced output signals from two FSM-based design schemes are shown in the two bottom parts of Figure 5.8. The first design scheme is discussed in this subsection and the second scheme is left as an exercise in Experiment 5.5.2. A better alternative FSMD-based scheme is discussed in Section 6.2.1.

An FSM-based design uses a free-running 10-ms timer and an FSM. The timer generates a one-clock-cycle enable tick (the `m_tick` signal) every 10 ms and the FSM uses this information to keep track of whether the input value is stabilized. In the first design scheme, the FSM ignores the short bounces and changes the value of the debounced output only after the input is stabilized for 20 ms. The output timing diagram is shown at the middle of Figure 5.8. The state diagram of this FSM is shown in Figure 5.9. The `zero` and `one` states indicate that the switch input signal, `sw`, has been stabilized with 0 and 1 values. Assume that the FSM is initially in the `zero` state. It moves to the `wait1_1` state when `sw` changes to 1. At the `wait1_1` state, the FSM waits for the assertion of `m_tick`. If `sw` becomes 0 in this state, it implies that the width of the 1 value does not last long enough and the FSM returns to the `zero` state. This action repeats two more times for the `wait1_2` and `wait1_3` states. The operation from the `one` state is similar except that the `sw` signal must be 0.

Since the 10-ms timer is free-running and the `m_tick` tick can be asserted at any time, the FSM checks the assertion three times to ensure that the `sw` signal is stabilized for at least 20 ms (it is actually between 20 and 30 ms). The code is shown in Listing 5.6. It includes a 10-ms timer and the FSM.

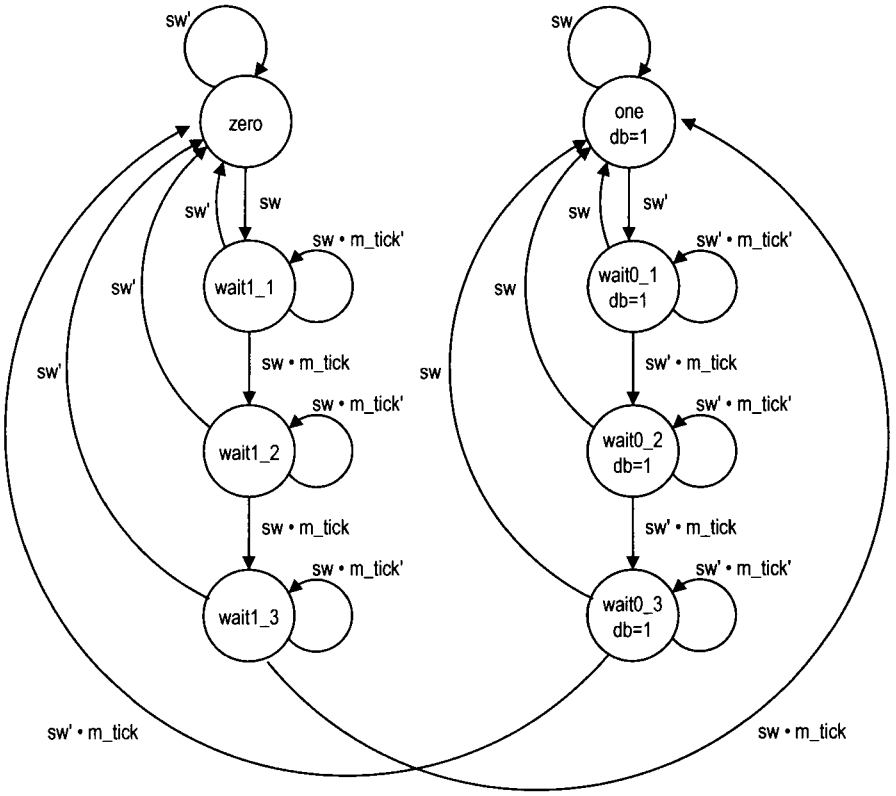


Figure 5.9 State diagram of a debouncing circuit.

Listing 5.6 FSM implementation of a debouncing circuit

```
module db_fsm
   (
    input wire clk, reset,
    input wire sw,
    output reg db
   );

   // symbolic state declaration
   localparam  [2:0]
               zero    = 3'b000,
               wait1_1 = 3'b001,
               wait1_2 = 3'b010,
               wait1_3 = 3'b011,
               one     = 3'b100,
               wait0_1 = 3'b101,
               wait0_2 = 3'b110,
               wait0_3 = 3'b111;

   // number of counter bits (2^N * 20ns = 10ms tick)
   localparam N =19;
```

```
   // signal declaration
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   wire m_tick;
   reg [2:0] state_reg, state_next;

   // body

   //=============================================
   // counter to generate 10 ms tick
   //=============================================
   always @(posedge clk)
      q_reg <= q_next;
   // next-state logic
   assign q_next = q_reg + 1;
   // output tick
   assign m_tick = (q_reg==0) ? 1'b1 : 1'b0;

   //=============================================
   // debouncing FSM
   //=============================================
   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= zero;
       else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
   begin
      state_next = state_reg;  // default state: the same
      db = 1'b0;               // default output: 0
      case (state_reg)
         zero:
            if (sw)
               state_next = wait1_1;
         wait1_1:
            if (~sw)
               state_next = zero;
            else
               if (m_tick)
                  state_next = wait1_2;
         wait1_2:
            if (~sw)
               state_next = zero;
            else
               if (m_tick)
                  state_next = wait1_3;
         wait1_3:
            if (~sw)
               state_next = zero;
```

```
            else
75              if (m_tick)
                   state_next = one;
         one:
            begin
              db = 1'b1;
80            if (~sw)
                 state_next = wait0_1;
            end
         wait0_1:
            begin
85             db = 1'b1;
               if (sw)
                  state_next = one;
               else
                 if (m_tick)
90                  state_next = wait0_2;
            end
         wait0_2:
            begin
               db = 1'b1;
95             if (sw)
                  state_next = one;
               else
                 if (m_tick)
                    state_next = wait0_3;
100         end
         wait0_3:
            begin
               db = 1'b1;
               if (sw)
105               state_next = one;
               else
                 if (m_tick)
                    state_next = zero;
            end
110      default: state_next = zero;
      endcase
   end

endmodule
```

5.3.3 Testing circuit

We use a bounce counting circuit to verify operation of the rising-edge detector and the debouncing circuit. The block diagram is shown in Figure 5.10. The input of the verification circuit is from a pushbutton switch. In the lower part, the signal is first fed to the debouncing circuit and then to the rising-edge detector. Therefore, a one-clock-cycle tick is generated each time the button is pressed and released. The tick in turn controls the enable input of an 8-bit counter, whose content is passed to the LED time-multiplexing circuit and shown on the left two digits of the prototyping board's seven-segment LED display. In the upper

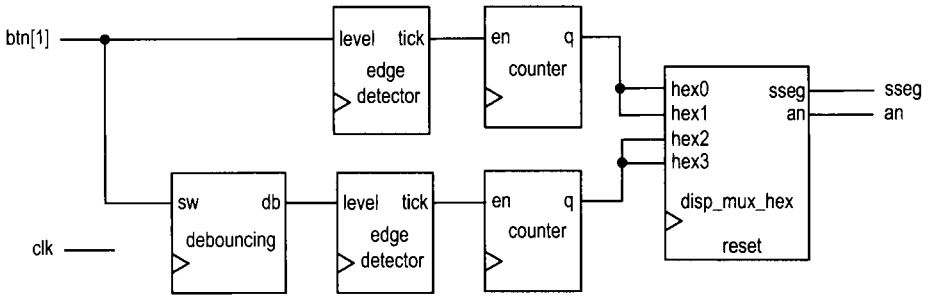


Figure 5.10 Debouncing testing circuit.

part, the input signal is fed directly to the edge detector without the debouncing circuit, and the number is shown on the right two digits of the prototyping board's seven-segment LED display. The bottom counter thus counts one desired 0-to-1 transition as well as the bounces.

The code is shown in Listing 5.7. It basically uses component instantiation to realize the block diagram.

Listing 5.7 Verification circuit for a debouncing circuit and rising-edge detector

```
module debounce_test
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   reg [7:0]  b_reg, d_reg;
   wire [7:0] b_next, d_next;
   reg  btn_reg, db_reg;
   wire db_level, db_tick, btn_tick, clr;

   // instantiate 7-seg LED display time-multiplexing module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(reset),
       .hex3(b_reg[7:4]), .hex2(b_reg[3:0]),
       .hex1(d_reg[7:4]), .hex0(d_reg[3:0]),
       .dp_in(4'b1011), .an(an), .sseg(sseg));

   // instantiate debouncing circuit
   db_fsm db_unit
      (.clk(clk), .reset(reset), .sw(btn[1]), .db(db_level));

   // edge detection circuits
   always @(posedge clk)
      begin
         btn_reg <= btn[1];
         db_reg <= db_level;
```

```
   end
assign btn_tick = ~btn_reg & btn[1];
assign db_tick = ~db_reg & db_level;

// two counters
assign clr = btn[0];
always @(posedge clk)
   begin
      b_reg <= b_next;
      d_reg <= d_next;
   end
assign b_next = (clr)      ? 8'b0 :
                (btn_tick) ? b_reg + 1 : b_reg;
assign d_next = (clr)      ? 8'b0 :
                (db_tick)  ? d_reg + 1 : d_reg;

endmodule
```

The seven-segment display shows the accumulated numbers of 0-to-1 edges of bounced and debounced switch input. After pressing and releasing the pushbutton switch several times, we can determine the average number of bounces for each transition.

5.4 BIBLIOGRAPHIC NOTES

The article "Coding and Scripting Techniques for FSM Designs with Synthesis-Optimized, Glitch-Free Outputs" by C. E. Cummings provides a detailed discussion on various coding styles of FSM.

5.5 SUGGESTED EXPERIMENTS

5.5.1 Dual-edge detector

A dual-edge detector is similar to a rising-edge detector except that the output is asserted for one clock cycle when the input changes from 0 to 1 (i.e., rising edge) and 1 to 0 (i.e., falling edge).

1. Design a circuit based on the Moore machine and draw the state diagram and ASM chart.
2. Derive the HDL code based on the state diagram of the ASM chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Replace the rising detectors in Section 5.3.3 with dual-edge detectors and verify their operations.
5. Repeat steps 1 to 4 for a Mealy machine–based design.

5.5.2 Alternative debouncing circuit

One problem with the debouncing design in Section 5.3.2 is the delayed response of the onset of a switch transition. An alternative is to react to the first edge in the transition and then wait for a small amount of time (at least 20 ms) to have the input signal settled. The output timing diagram is shown at the bottom of Figure 5.8. When the input changes from

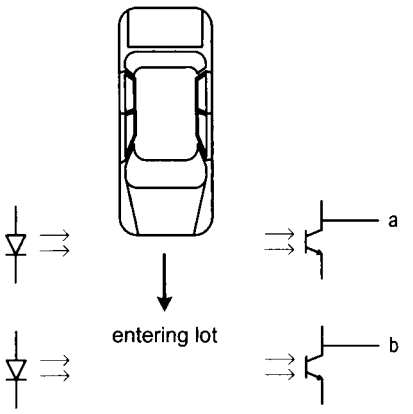


Figure 5.11 Conceptual diagram of gate sensors.

0 to 1, the FSM responds immediately. The FSM then ignores the input for about 20 ms to avoid glitches. After this amount of time, the FSM starts to check the input for the falling edge. Follow the design procedure in Section 5.3.2 to design the alternative circuit.

1. Derive the state diagram and ASM chart for the circuit.
2. Derive the HDL code.
3. Derive the HDL code based on the state diagram and ASM chart.
4. Derive a testbench and use simulation to verify operation of the code.
5. Replace the debouncing circuit in Section 5.3.3 with the alternative design and verify its operation.

5.5.3 Parking lot occupancy counter

Consider a parking lot with a single entry and exit gate. Two pairs of photo sensors are used to monitor the activity of cars, as shown in Figure 5.11. When an object is between the photo transmitter and the photo receiver, the light is blocked and the corresponding output is asserted to 1. By monitoring the events of two sensors, we can determine whether a car is entering or exiting or a pedestrian is passing through. For example, the following sequence indicates that a car enters the lot:

- Initially, both sensors are unblocked (i.e., the `a` and `b` signals are "00").
- Sensor a is blocked (i.e., the `a` and `b` signals are "10").
- Both sensors are blocked (i.e., the `a` and `b` signals are "11").
- Sensor a is unblocked (i.e., the `a` and `b` signals are "01").
- Both sensors becomes unblocked (i.e., the `a` and `b` signals are "00").

Design a parking lot occupancy counter as follows:

1. Design an FSM with two input signals, `a` and `b`, and two output signals, `enter` and `exit`. The `enter` and `exit` signals assert one clock cycle when a car enters and one clock cycle when a car exits the lot, respectively.
2. Derive the HDL code for the FSM.
3. Design a counter with two control signals, `inc` and `dec`, which increment and decrement the counter when asserted. Derive the HDL code.

4. Combine the counter and the FSM and LED multiplexing circuit. Use two debounced pushbuttons to mimic operation of the two sensor outputs. Verify operation of the occupancy counter.

CHAPTER 6

FSMD

6.1 INTRODUCTION

An FSMD (finite state machine with data path) combines an FSM and regular sequential circuits. The FSM, which is sometimes known as a *control path*, examines the external commands and status and generates control signals to specify operation of the regular sequential circuits, which are known collectively as a *data path*. The FSMD is used to implement systems described by *RT (register transfer) methodology*, in which the operations are specified as data manipulation and transfer among a collection of registers.

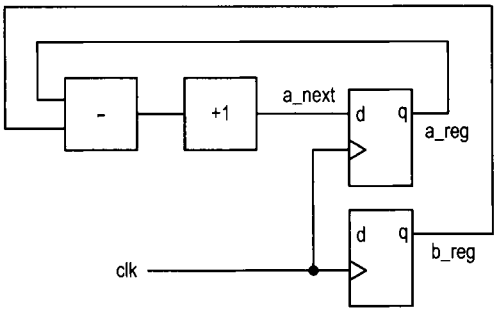
6.1.1 Single RT operation

An RT operation specifies data manipulation and transfer for a single destination register. It is represented by the notation

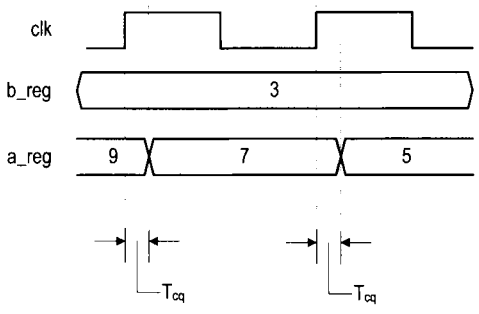
$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

where r_{dest} is the destination register, r_{src1}, r_{src2}, and r_{srcn} are the source registers, and $f(\cdot)$ specifies the operation to be performed. The notation indicates that the contents of the source registers are fed to the $f(\cdot)$ function, which is realized by a combinational circuit, and the result is passed to the input of the destination register and stored in the destination register at the next rising edge of the clock. Following are several representative RT operations:

- `r1` $\leftarrow$ 0. A constant 0 is stored in the `r1` register.
- `r1` $\leftarrow$ `r1`. The content of the `r1` register is written back to itself.



(a) Block diagram



(b) Timing diagram

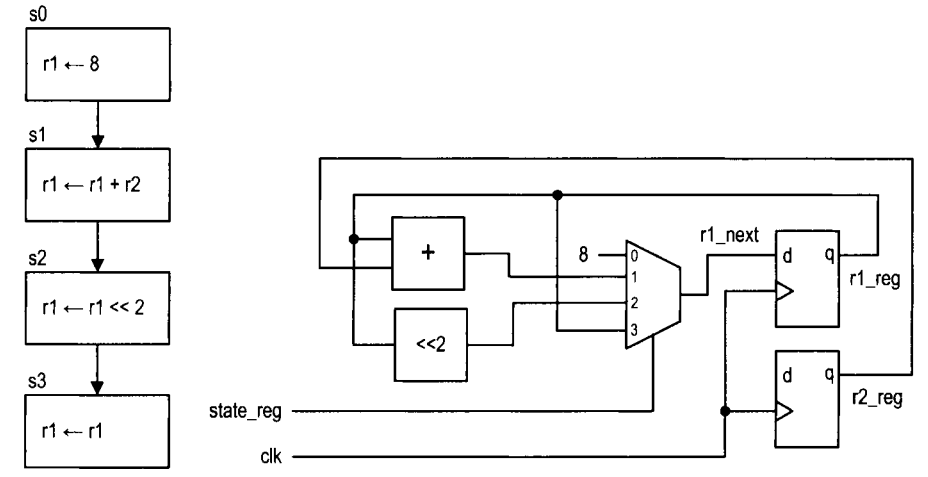
Figure 6.1 Block and timing diagrams of an RT operation.

- `r2` ← `r2 >> 3`. The `r2` register is shifted right three positions and then written back to itself.
- `r2` ← `r1`. The content of the `r1` register is transferred to the `r2` register.
- `i` ← `i + 1`. The content of the `i` register is incremented by 1 and the result is written back to itself.
- `d` ← `s1 + s2 + s3`. The summation of the `s1`, `s2`, and `s3` registers is written to the `d` register.
- `y` ← `a*a`. The `a` squared is written to the `y` register.

A single RT operation can be implemented by constructing a combinational circuit for the $f(\cdot)$ function and connecting the input and output of the registers. For example, consider the `a` ← `a-b+1` operation. The $f(\cdot)$ function involves a subtractor and an incrementor. The block diagram is shown in Figure 6.1(a). For clarity, we use the `_reg` and `_next` suffixes to represent the input and output of a register. Note that an RT operation is synchronized by an embedded clock. The result from the $f(\cdot)$ function is not stored to the destination register until the next rising edge of the clock. The timing diagram of the previous RT operation is shown in Figure 6.1(b).

6.1.2 ASMD chart

A circuit based on the RT methodology specifies which RT operations should be executed in each step. Since an RT operation is done on a clock-by-clock basis, its timing is similar to a state transition of an FSM. Thus, an FSM is a natural choice to specify the sequencing



(a) ASMD segment (b) Block diagram

Figure 6.2 Realization of an ASMD segment.

of an RT algorithm. We extend the ASM chart to incorporate RT operations and call it an *ASMD* (ASM with data path) chart. The RT operations are treated as another type of activity and can be placed where the output signals are used.

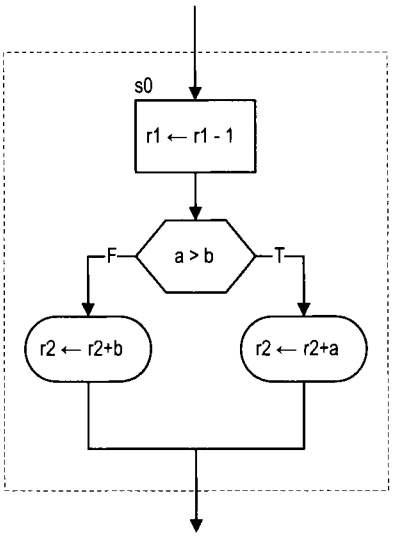
A segment of an ASMD chart is shown in Figure 6.2(a). It contains one destination register, `r1`, which is initialized with 8, added with content of the `r2` register, and then shifted left two positions. Note that the `r1` register must be specified in each state. When `r1` is not changed, the `r1` ← `r1` operation should be used to maintain its current content, as in the `s3` state. In future discussion, we assume that `r` ← `r` is the default RT operation for the `r` register and do not include it in the ASMD chart. Implementing the RT operations of an ASMD chart involves a multiplexing circuit to route the desired next value to the destination register. For example, the previous segment can be implemented by a 4-to-1 multiplexer, as shown in Figure 6.2(b). The current state (i.e., the output of the state register) of the FSM controls the selection signal of the multiplexer and thus chooses the result of the desired RT operation.

An RT operation can also be specified in a conditional output box, as the `r2` register shown in Figure 6.3(a). Depending on the `a>b` condition, the FSMD performs either `r2` ← `r2+a` or `r2` ← `r2+b`. Note that all operations are done in parallel inside an ASMD block. We need to realize the `a>b`, `r2+a`, and `r2+b` operations and use a multiplexer to route the desired value to `r2`. The block diagram is shown in Figure 6.3(b).

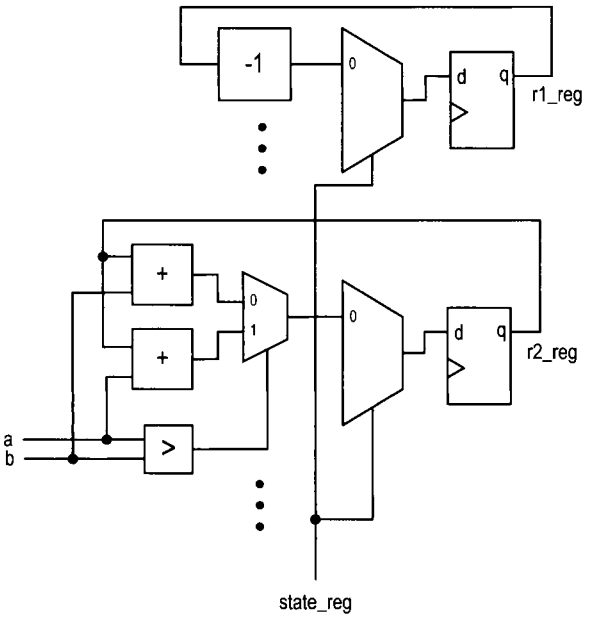
6.1.3 Decision box with a register

The appearance of an ASMD chart is similar to that of a normal flowchart. The main difference is that the RT operation in an ASMD chart is controlled by an embedded clock signal and the destination register is updated *when the FSMD exits the current ASMD block*, but not within the block. The `r` ← `r-1` operation actually means that:

- `r_next = r_reg - 1;`
- `r_reg <= r_next` at the rising edge of the clock (i.e., when the FSMD exits the current block).

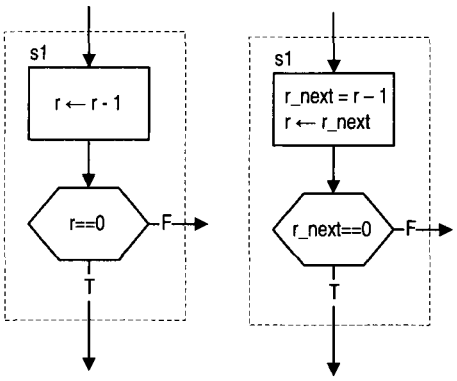


(a) ASM block



(b) Block diagram

Figure 6.3 Realization of an RT operation in a conditional output box.



(a) Use old value of r (b) Use new value of r

Figure 6.4 ASM block affected by a delayed store.

This "delayed store" may introduce subtle errors when a register is used in a decision box. Consider the FSMD segment in Figure 6.4(a). The r register is decremented in the state box and used in the decision box. Since the r register is not updated until the FSMD exits the block, the old content of r is used for comparison in the decision box. If the new value of r is desired, we should use the output of the combinational logic (i.e., r_next) in the decision box (i.e., replace the r==0 expression with r_next==0), as shown in Figure 6.4(b).

Block diagram of an FSMD The conceptual block diagram of an FSMD is divided into a data path and a control path, as shown in Figure 6.5. The data path performs the required RT operations. It consists of:

- *Data registers*: store the intermediate computation results
- *Functional units*: perform the functions specified by the RT operations
- *Routing network*: routes data between the storage registers and the functional units

The data path follows the control signal to perform the desired RT operations and generates the internal status signal.

The control path is an FSM. As a regular FSM, it contains a state register, next-state logic, and output logic. It uses the external command signal and the data path's status signal as the input and generates the control signal to control the data path operation. The FSM also generates the external status signal to indicate the status of the FSMD operation.

Note that although an FSMD consists of two types of sequential circuits, both circuits are controlled by the same clock, and thus the FSMD is still a synchronous system.

6.2 CODE DEVELOPMENT OF AN FSMD

We use an improved debouncing circuit to demonstrate derivation of the FSMD code. Although the debouncing circuit in Section 5.3.2 uses an FSM and a timer (which is a regular sequential circuit), it is not based on the RT methodology because the two units are running independently and the FSM has no control over the timer. Since the 10-ms enable tick can be asserted at any time, the FSM does not know how much time has elapsed when the first tick is detected in the wait1_1 or wait0_1 state. Thus, the waiting period in this

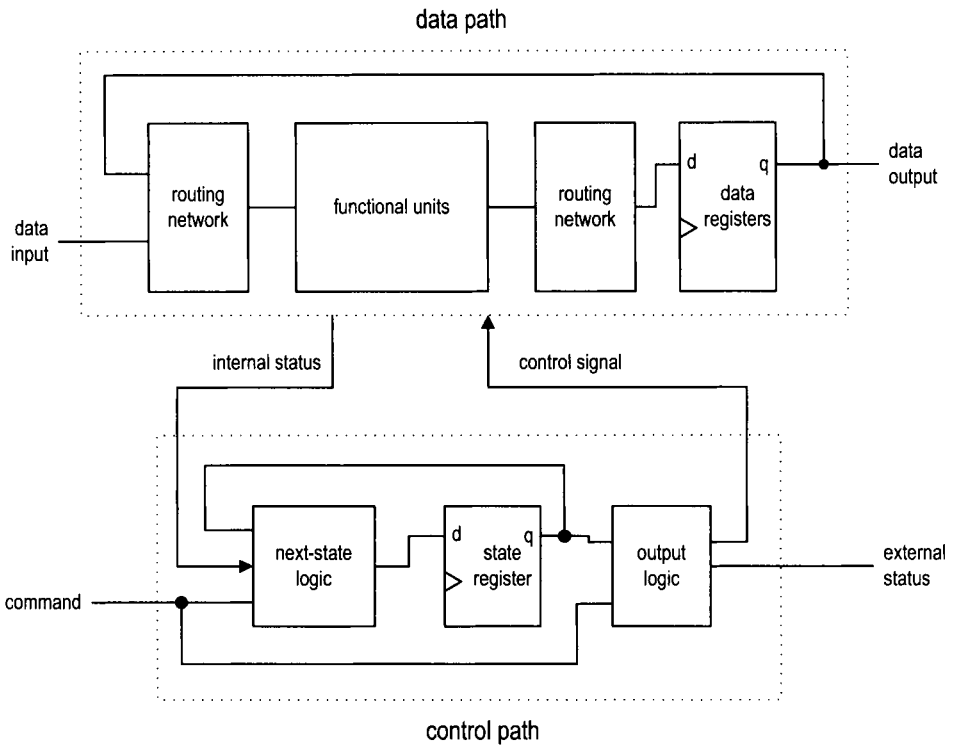


Figure 6.5 Block diagram of an FSMD.

design is between 20 and 30 ms but is not an exact interval. This deficiency can be overcome by applying the RT methodology. In this section, we use this improved debouncing circuit to illustrate FSMD code development.

6.2.1 Debouncing circuit based on RT methodology

With the RT methodology, we can use an FSM to control the initiation of the timer to obtain the exact interval. The ASMD chart is shown in Figure 6.6. The circuit is expanded to include two output signals: `db_level`, which is the debounced output, and `db_tick`, which is a one-clock-cycle enable pulse asserted at the zero-to-one transition. The `zero` and `one` states mean that the `sw` input has been stabilized for 0 and 1, respectively. The `wait1` and `wait0` states are used to filter out short glitches. The `sw` signal must be stable for a certain amount of time or the transition will be treated as a glitch. The data path contains one register, q, which is 21 bits wide. Assume that the FSMD is originally in the `zero` state. When the `sw` input signal becomes 1, the FSMD moves to the `wait1` state and initializes q to "1...1". In the `wait1` state, the q decrements in each clock cycle. If `sw` remains as 1, the FSMD returns to this state repeatedly until q reaches "0...0" and then moves to the `one` state.

Recall that the 50-MHz (i.e., 20-ns period) system clock is used on the prototyping board. Since the FSMD stays in the `wait1` state for 2^{21} clock cycles, it is about 40 ms

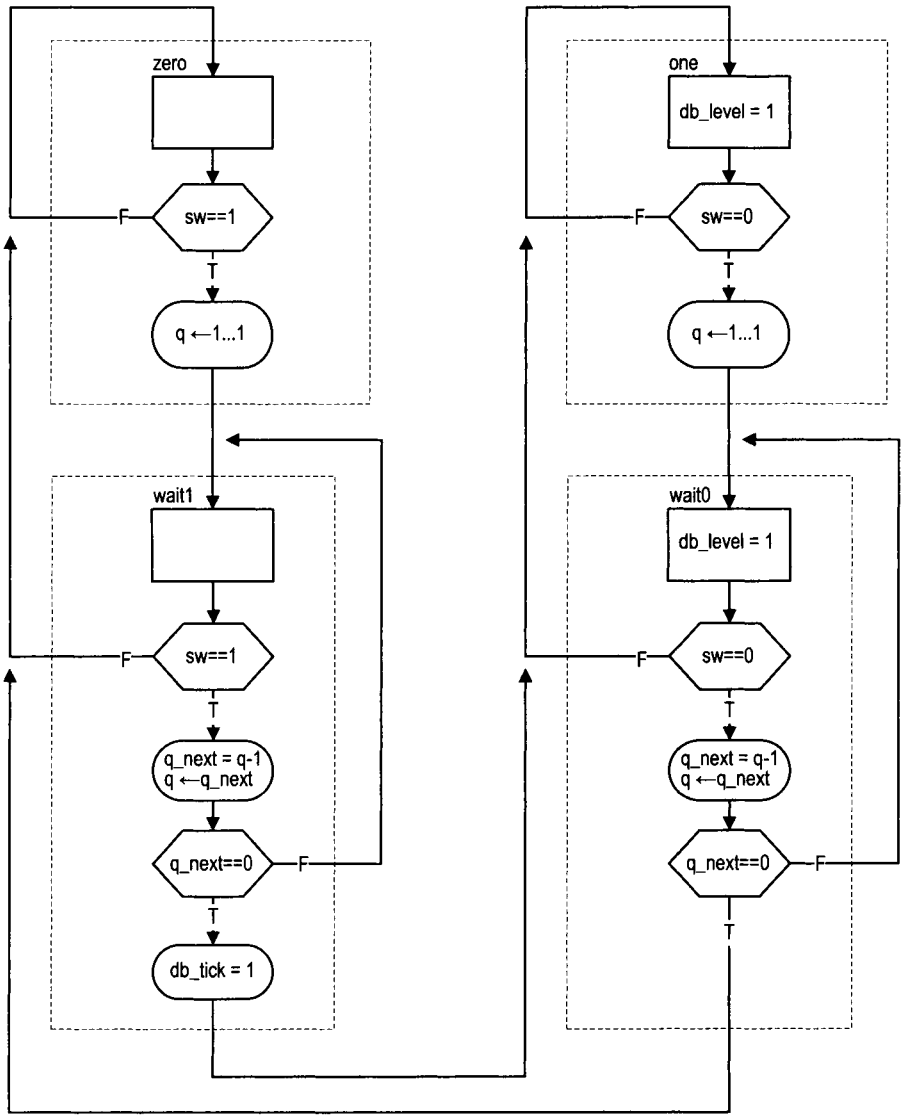


Figure 6.6 ASMD chart of a debouncing circuit.

(i.e., $2^{21} * 20$ ns). We can modify the initial value of the q register to obtain the desired wait interval.

There are two ways to derive the HDL code: one with *explicit description* of the data path components and the other with *implicit description* of the data path components.

6.2.2 Code with explicit data path components

The first approach to FSMD code development is to separate the control FSM and the key data path components. From an ASMD chart, we first identify the key components in the data path and the associated control signals and then describe these components in individual code segments.

The key data path component of the debouncing circuit ASMD chart is a custom 21-bit decrement counter that can:

- Be initialized with a specific value
- Count downward or pause
- Assert a status signal when the counter reaches 0

We can create a binary counter with a `q_load` signal to load the initial value and a `q_dec` signal to enable the counting. The counter also generates a `q_zero` status signal, which is asserted when the counter reaches zero. The complete data path is composed of the q register and the next-state logic of the custom decrement counter. A comparison circuit is included to generate the `q_zero` status signal. The control path consists of an FSM, which takes the `sw` input and the `q_zero` status and asserts the control signals, `q_load` and `q_dec`, according to the desired action in the ASMD chart. The HDL code follows the data path specification and the ASMD chart, and is shown in Listing 6.1.

Listing 6.1 Debouncing circuit with an explicit data path component

```
module debounce_explicit
   (
    input wire clk, reset,
    input wire sw,
    output reg db_level, db_tick
   );

   // symbolic state declaration
   localparam  [1:0]
               zero  = 2'b00,
               wait0 = 2'b01,
               one   = 2'b10,
               wait1 = 2'b11;

   // number of counter bits (2^N * 20ns = 40ms)
   localparam N=21;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [N-1:0] q_reg;
   wire [N-1:0] q_next;
   wire q_zero;
   reg q_load, q_dec;

   // body
```

```
// fsmd state & data registers
always @(posedge clk, posedge reset)
   if (reset)
      begin
         state_reg <= zero;
         q_reg <= 0;
      end
   else
      begin
         state_reg <= state_next;
         q_reg <= q_next;
      end

// FSMD data path (counter) next-state logic
assign q_next = (q_load) ? {N{1'b1}} :   // load 1..1
                (q_dec)  ? q_reg - 1 :   // decrement
                           q_reg;
// status signal
assign q_zero = (q_next==0);

// FSMD control path next-state logic
always @*
begin
   state_next = state_reg;  // default state: the same
   q_load = 1'b0;           // default output: 0
   q_dec = 1'b0;            // default output: 0
   db_tick = 1'b0;          // default output: 0
   case (state_reg)
      zero:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  state_next = wait1;
                  q_load = 1'b1;
               end
         end
      wait1:
         begin
            db_level = 1'b0;
            if (sw)
               begin
                  q_dec = 1'b1;
                  if (q_zero)
                     begin
                        state_next = one;
                        db_tick = 1'b1;
                     end
               end
            else // sw==0
               state_next = zero;
         end
      one:
```

```
            begin
80             db_level = 1'b1;
               if (~sw)
                  begin
                     state_next = wait0;
                     q_load = 1'b1;
85                end
            end
         wait0:
            begin
               db_level = 1'b1;
90             if (~sw)
                  begin
                     q_dec = 1'b1;
                     if (q_zero)
                        state_next = zero;
95                end
               else // sw==1
                  state_next = one;
            end
         default: state_next = zero;
100   endcase
   end

endmodule
```

6.2.3 Code with implicit data path components

An alternative coding style is to embed the RT operations within the FSM control path. Instead of explicitly defining the data path components, we just list RT operations with the corresponding FSM state. The code of the debouncing circuit is shown in Listing 6.2.

Listing 6.2 Debouncing circuit with an implicit data path component

```
module debounce
   (
    input wire clk, reset,
    input wire sw,
5   output reg db_level, db_tick
   );

   // symbolic state declaration
   localparam  [1:0]
10             zero  = 2'b00,
               wait0 = 2'b01,
               one   = 2'b10,
               wait1 = 2'b11;

15 // number of counter bits (2^N * 20ns = 40ms)
   localparam N=21;

   // signal declaration
   reg [N-1:0] q_reg, q_next;
```

```
20  reg [1:0] state_reg, state_next;

    // body
    // fsmd state & data registers
     always @(posedge clk, posedge reset)
25      if (reset)
           begin
              state_reg <= zero;
              q_reg <= 0;
           end
30      else
           begin
              state_reg <= state_next;
              q_reg <= q_next;
           end
35
    // next-state logic & data path functional units/routing
    always @*
    begin
       state_next = state_reg;    // default state: the same
40     q_next = q_reg;            // default q: unchnaged
       db_tick = 1'b0;            // default output: 0
       case (state_reg)
          zero:
             begin
45              db_level = 1'b0;
                if (sw)
                   begin
                      state_next = wait1;
                      q_next = {N{1'b1}}; // load 1..1
50                 end
             end
          wait1:
             begin
                db_level = 1'b0;
55              if (sw)
                   begin
                      q_next = q_reg - 1;
                      if (q_next==0)
                         begin
60                          state_next = one;
                            db_tick = 1'b1;
                         end
                   end
                else // sw==0
65                 state_next = zero;
             end
          one:
             begin
                db_level = 1'b1;
70              if (~sw)
                   begin
                      state_next = wait0;
```

```
                        q_next = {N{1'b1}}; // load 1..1
                     end
75             end
            wait0:
               begin
                  db_level = 1'b1;
                  if (~sw)
80                   begin
                        q_next = q_reg - 1;
                        if (q_next==0)
                           state_next = zero;
                     end
85                   else // sw==1
                        state_next = one;

               end
            default: state_next = zero;
90       endcase
      end

   endmodule
```

The code consists of a memory segment and a combinational logic segment. The former contains the state register of the FSM and the data register of the data path. The latter basically specifies the next-state logic of the control path FSM. Instead of generating control signals, the next data register values are specified in individual states. The next-state logic of the data path, which consists of functional units and a routing network, is created accordingly.

6.2.4 Comparison

Code with implicit data path components essentially follows the ASMD chart. We just convert the chart to an HDL description. Although this approach is simpler and more descriptive, we rely on synthesis software for data path construction and have less control. This can best be explained by an example. Consider the ASMD segment in Figure 6.7. The implicit description becomes

```
case (state_reg)
   s1:
     begin
       d1_next = a * b;
       ...
     end
   s2:
     begin
       d2_next = b * c;
       ...
     end
   s3:
     begin
       d3_next = a * c;
       ...
     end
```

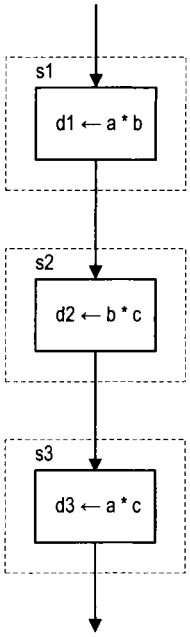


Figure 6.7 ASMD segment with sharing opportunity.

```
    . . .
endcase
```

The synthesis software may infer three multipliers. Since a combinational multiplier is a complex circuit, it is more efficient to share the circuit. We can use explicit description to isolate the multiplier:

```
case (state_reg)
   s1:
     begin
       in1 = a;
       in2 = b;
       d1_next = m_out;
       ...
     end
   s2:
     begin
       in1 = b;
       in2 = c;
       d2_next = m_out;
       ...
     end
   s3:
     begin
       in1 = a;
       in2 = c;
       d3_next = m_out;
       ...
```

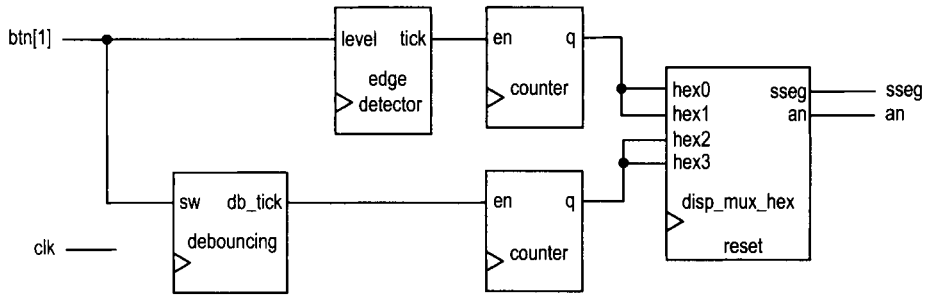


Figure 6.8 Debouncing testing circuit.

```
      end
    . . .
endcase

. . .

// explicit description of a single multiplier
// outside the always block
assign m_out = in1 * in2;
```

The code ensures that only one multiplier is inferred during synthesis. The implicit and explicit descriptions can be mixed for a complex FSMD design. We frequently isolate and extract complex data path components for code clarity and efficiency.

6.2.5 Testing circuit

The debouncing testing circuit discussed in Section 5.3.3 can be used to verify operation of the new design. Since the revised debouncing circuit's outputs include a one-clock-cycle tick signal, no edge detector is needed after the debouncing circuit. The revised block diagram is shown in Figure 6.8, and the corresponding code is shown in Listing 6.3.

Listing 6.3 Verification circuit for a debouncing circuit

```
module debounce_fsmd_test
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   reg [7:0]  b_reg, d_reg;
   wire [7:0] b_next, d_next;
   reg  btn_reg;
   wire db_tick, btn_tick, clr;

   // instantiate 7-seg LED display time-multiplexing module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(reset),
```

```
        .hex3(b_reg[7:4]), .hex2(b_reg[3:0]),
        .hex1(d_reg[7:4]), .hex0(d_reg[3:0]),
20      .dp_in(4'b1011), .an(an), .sseg(sseg));

    // instantiate debouncing circuit
    debounce db_unit
       (.clk(clk), .reset(reset), .sw(btn[1]),
25      .db_level(), .db_tick(db_tick));

    // edge detection circuit for un-debounced input
    always @(posedge clk)
       btn_reg <= btn[1];
30  assign btn_tick = ~btn_reg & btn[1];

    // two counters
    assign clr = btn[0];
    always @(posedge clk)
35     begin
          d_reg <= d_next;
          b_reg <= b_next;
       end
    //next-state logic for the counter
40  assign b_next = (clr )     ? 0 :
                    (btn_tick) ? b_reg + 1 :
                                 b_reg;
    assign d_next = (clr )     ? 0 :
                    (db_tick)  ? d_reg + 1 :
45                               d_reg;

endmodule
```

6.3 DESIGN EXAMPLES

6.3.1 Fibonacci number circuit

The Fibonacci numbers constitute a sequence defined as

$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i > 1 \end{cases}$$

One way to calculate $fib(i)$ is to construct the function iteratively, from 0 to the desired i. This approach requires two temporary registers to store the two most recently calculated values [i.e., $fib(i-1)$ and $fib(i-2)$] and one index register to keep track of the number of iterations. The ASMD chart is shown in Figure 6.9, in which `t1` and `t0` are temporary storage registers and `n` is the index register. In addition to the regular data input and output signals, `i` and `f`, we include a command signal, `start`, which signals the beginning of operation, and two status signals: `ready`, which indicates that the circuit is idle and ready to take new input, and `done_tick`, which is asserted for one clock cycle when the operation is completed. Since this circuit, like many other FSMD designs, is probably a part of a larger system, these signals are needed to interface with other subsystems.

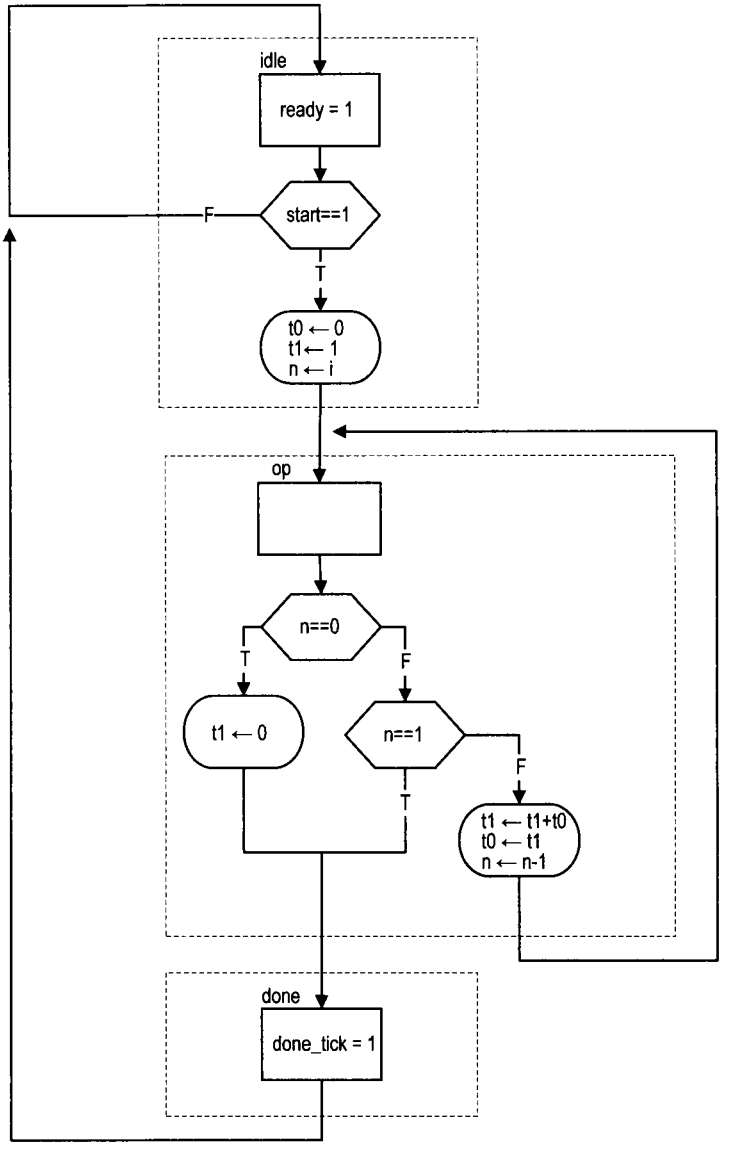


Figure 6.9 ASMD chart of a Fibonacci circuit.

The ASMD chart has three states. The `idle` state indicates that the circuit is currently idle. When `start` is asserted, the FSMD moves to the `op` state and loads initial values to three registers. The `t0` and `t1` registers are loaded with 0 and 1, which represent $fib(0)$ and $fib(1)$, respectively. The `n` register is loaded with `i`, the desired number of iterations.

The main computation is iterated through the `op` state by three RT operations:

- `t1` ← `t1 + t0`
- `t0` ← `t1`
- `n` ← `n - 1`

The first two RT operations obtain a new value and store the two most recently calculated values in `t1` and `t0`. The third RT operation decrements the iteration index. The iteration ended when `n` reaches 1 or its initial value is 0 [i.e., $fib(0)$]. Unlike a regular flowchart, the operations in an ASMD block can be performed concurrently in the same clock cycle. We put all comparison and RT operations in the `op` state to reduce the computation time. Note that the new values of the `t1` and `t0` registers are loaded at the same time when the FSMD exits the `op` state (i.e., at the next rising edge of the clock). Thus, the original value of `t1`, not `t1+t0`, is stored to `t0`. The purpose of the `done` state is to generate the one-clock-cycle `done_tick` signal to indicate completion of the computation. This state can be omitted if this status signal is not needed.

The code follows the ASMD chart and is shown in Listing 6.4. Note that the Fibonacci function grows rapidly and the output signal should be wide enough to accommodate the desired result.

Listing 6.4 Fibonacci number circuit

```
module fib
   (
    input wire clk, reset,
    input wire start,
    input wire [4:0] i,
    output reg ready, done_tick,
    output wire [19:0] f
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      done = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
   reg [4:0] n_reg, n_next;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            t0_reg <= 0;
            t1_reg <= 0;
```

```
            n_reg <= 0;
        end
    else
        begin
            state_reg <= state_next;
            t0_reg <= t0_next;
            t1_reg <= t1_next;
            n_reg <= n_next;
        end
// FSMD next-state logic
always @*
begin
    state_next = state_reg;
    ready = 1'b0;
    done_tick = 1'b0;
    t0_next = t0_reg;
    t1_next = t1_reg;
    n_next = n_reg;
    case (state_reg)
        idle:
            begin
                ready = 1'b1;
                if (start)
                    begin
                        t0_next = 0;
                        t1_next = 20'd1;
                        n_next = i;
                        state_next = op;
                    end
            end
        op:
            if (n_reg==0)
                begin
                    t1_next = 0;
                    state_next = done;
                end
            else if (n_reg==1)
                state_next = done;
            else
                begin
                    t1_next = t1_reg + t0_reg;
                    t0_next = t1_reg;
                    n_next = n_reg - 1;
                end
        done:
            begin
                done_tick = 1'b1;
                state_next = idle;
            end
        default: state_next = idle;
    endcase
end
// output
```

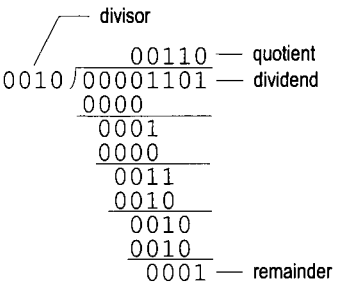


Figure 6.10 Long division of two 4-bit unsigned integers.

```
    assign f = t1_reg;

endmodule
```

6.3.2 Division circuit

Because of complexity, the division operator cannot be synthesized automatically. We use an FSMD to implement the long-division algorithm in this subsection. The algorithm is illustrated by the division of two 4-bit unsigned integers in Figure 6.10. The algorithm can be summarized as follows:

1. Double the dividend width by appending 0's in front and align the divisor to the leftmost bit of the extended dividend.
2. If the corresponding dividend bits are greater than or equal to the divisor, subtract the divisor from the dividend bits and make the corresponding quotient bit 1. Otherwise, keep the original dividend bits and make the quotient bit 0.
3. Append one additional dividend bit to the previous result and shift the divisor to the right one position.
4. Repeat steps 2 and 3 until all dividend bits are used.

The sketch of the data path is shown in Figure 6.11. Initially, the divisor is stored in the `d` register and the extended dividend is stored in the `rh` and `rl` registers. In each iteration, the `rh` and `rl` registers are shifted to the left one position. This corresponds to shifting the divisor to the right of the preceding algorithm. We can then compare `rh` and `d` and perform subtraction if `rh` is greater than or equal to `d`. When `rh` and `rl` are shifted to the left, the rightmost bit of `rl` becomes available. It can be used to store the current quotient bit. After we iterate through all dividend bits, the result of the last subtraction is stored in `rh` and becomes the remainder of the division, and all quotients are shifted into `rl`.

The ASMD chart of the division circuit is somewhat similar to that of the previous Fibonacci circuit. The FSMD consists of four states: `idle`, `op`, `last`, and `done`. To make the code clear, we extract the *compare and subtract* circuit to separate code segments. The main computation is performed in the `op` state, in which the dividend bits and divisor are compared and subtracted and then shifted left 1 bit. Note that the remainder should not be shifted in the last iteration. We create a separate state, `last`, to accommodate this special requirement. As in the preceding example, the purpose of the `done` state is to generate a one-clock-cycle `done_tick` signal to indicate completion of the computation. The code is shown in Listing 6.5.

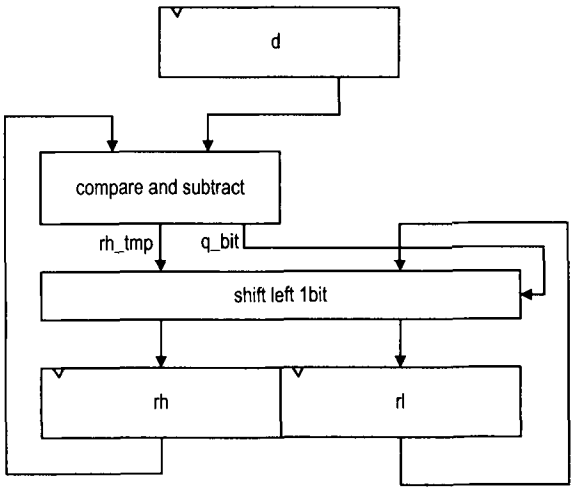


Figure 6.11 Sketch of division circuit's data path.

Listing 6.5 Division circuit

```
module div
   #(
     parameter W = 8,
               CBIT = 4    // CBIT=log2(W)+1
    )
   (
    input wire clk, reset,
    input wire start,
    input wire [W-1:0] dvsr, dvnd,
    output reg ready, done_tick,
    output wire [W-1:0] quo, rmd
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      last = 2'b10,
      done = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [W-1:0] rh_reg, rh_next, rl_reg, rl_next, rh_tmp;
   reg [W-1:0] d_reg, d_next;
   reg [CBIT-1:0] n_reg, n_next;
   reg q_bit;

   // body
   // FSMD state & data registers
```

```
30    always @(posedge clk, posedge reset)
         if (reset)
            begin
               state_reg <= idle;
               rh_reg <= 0;
35             rl_reg <= 0;
               d_reg <= 0;
               n_reg <= 0;
            end
         else
40          begin
               state_reg <= state_next;
               rh_reg <= rh_next;
               rl_reg <= rl_next;
               d_reg <= d_next;
45             n_reg <= n_next;
            end

      // FSMD next-state logic
      always @*
50    begin
         state_next = state_reg;
         ready = 1'b0;
         done_tick = 1'b0;
         rh_next = rh_reg;
55       rl_next = rl_reg;
         d_next = d_reg;
         n_next = n_reg;
         case (state_reg)
            idle:
60             begin
                  ready = 1'b1;
                  if (start)
                     begin
                        rh_next = 0;
65                      rl_next = dvnd;   // dividend
                        d_next = dvsr;    // divisor
                        n_next = CBIT;    // index
                        state_next = op;
                     end
70          end
            op:
               begin
                  // shift rh and rl left
                  rl_next = {rl_reg[W-2:0], q_bit};
75                rh_next = {rh_tmp[W-2:0], rl_reg[W-1]};
                  // decrease index
                  n_next = n_reg - 1;
                  if (n_next==1)
                     state_next = last;
80          end
            last: // last iteration
               begin
```

```
                rl_next = {rl_reg[W-2:0], q_bit};
                rh_next = rh_tmp;
                state_next = done;
             end
          done:
             begin
                done_tick = 1'b1;
                state_next = idle;
             end
          default: state_next = idle;
       endcase
    end

    // compare and subtract circuit
    always @*
       if (rh_reg >= d_reg)
          begin
             rh_tmp = rh_reg - d_reg;
             q_bit = 1'b1;
          end
       else
          begin
             rh_tmp = rh_reg;
             q_bit = 1'b0;
          end

    //output
    assign quo = rl_reg;
    assign rmd = rh_reg;

endmodule
```

6.3.3 Binary-to-BCD conversion circuit

We discussed the BCD format in Section 4.5.2. In this format, a decimal number is represented as a sequence of 4-bit BCD digits. A binary-to-BCD conversion circuit converts a binary number to the BCD format. For example, the binary number "0010 0000 0000" becomes "0101 0001 0010" (i.e., 512_{10}) after conversion.

The binary-to-BCD conversion can be processed by a special BCD shift register, which is divided into 4-bit groups internally, each representing a BCD digit. Shifting a BCD sequence to the left requires adjustment if a BCD digit is greater than 9_{10} after shifting. For example, if a BCD sequence is "0001 0111" (i.e., 17_{10}), it should become "0011 0100" (i.e., 34_{10}) rather than "0010 1110". The adjustment requires subtracting 10_{10} (i.e., "1010") from the right BCD digit and adding 1 (which can be considered as a carry-out) to the next BCD digit. Note that subtracting 10_{10} is equivalent to adding 6_{10} for a 4-bit binary number. Thus, the foregoing adjustment can also be achieved by adding 6_{10} to the right BCD digit. The carry-out bit is generated automatically in this process.

In the actual implementation, it is more efficient to first perform the necessary adjustment on a BCD digit and then shift. We can check whether a BCD digit is greater than 4_{10} and, if this is the case, add 3_{10} to the digit. After all the BCD digits are corrected, we can then shift the entire register to the left one position. A binary-to-BCD conversion circuit can

Table 6.1 Binary-to-BCD conversion example

Operation		Special BCD shift register			Binary input
		BCD digit 2	BCD digit 1	BCD digit 0	
Initial					111 1111
Bit 6	no adjustment shift left 1 bit			1 (1_{10})	11 1111
Bit 5	no adjustment shift left 1 bit			11 (3_{10})	1 1111
Bit 4	no adjustment shift left 1 bit			111 (7_{10})	1111
Bit 3	BCD digit 0 adjustment shift left 1 bit		1 (1_{10})	1010 0101 (5_{10})	111
Bit 2	BCD digit 0 adjustment shift left 1 bit		1 11 (3_{10})	1000 0001 (1_{10})	11
Bit 1	no adjustment shift left 1 bit		110 (6_{10})	0011 (3_{10})	1
Bit 0	BCD digit 1 adjustment shift left 1 bit	1 (1_{10})	1001 0010 (2_{10})	0011 0111 (7_{10})	

be constructed by shifting the binary input to a BCD shift register bit by bit, from MSB to LSB. Its operation can be summarized as follows:

1. For each 4-bit BCD digit in a BCD shift register, check whether the digit is greater than 4. If this is the case, add 3_{10} to the digit.
2. Shift the entire BCD register left one position and shift in the MSB of the input binary sequence to the LSB of the BCD register.
3. Repeat steps 1 and 2 until all input bits are used.

The conversion process of a 7-bit binary input, "111 1111" (i.e., 127_{10}), is demonstrated in Table 6.1.

The code of a 13-bit conversion circuit is shown in Listing 6.6. It uses a simple FSMD to control the overall operation. When the `start` signal is asserted, the binary input is stored to the p2s register. The FSM then iterates through the 13 bits, similar to the process described in previous examples. Four adjustment circuits are used to correct the four BCD digits. For clarity, they are isolated from the next-state logic and described in a separate code segment.

Listing 6.6 Binary-to-BCD conversion circuit

```
module bin2bcd
   (
    input wire clk, reset,
    input wire start,
    input wire [12:0] bin,
    output reg ready, done_tick,
    output wire [3:0] bcd3, bcd2, bcd1, bcd0
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      done = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [12:0] p2s_reg, p2s_next;
   reg [3:0] n_reg, n_next;
   reg [3:0] bcd3_reg, bcd2_reg, bcd1_reg, bcd0_reg;
   reg [3:0] bcd3_next, bcd2_next, bcd1_next, bcd0_next;
   wire [3:0] bcd3_tmp, bcd2_tmp, bcd1_tmp, bcd0_tmp;


   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            p2s_reg <= 0;
            n_reg <= 0;
            bcd3_reg <= 0;
            bcd2_reg <= 0;
            bcd1_reg <= 0;
            bcd0_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            p2s_reg <= p2s_next;
            n_reg <= n_next;
            bcd3_reg <= bcd3_next;
            bcd2_reg <= bcd2_next;
            bcd1_reg <= bcd1_next;
            bcd0_reg <= bcd0_next;
         end


   // FSMD next-state logic
   always @*
   begin
```

```
      state_next = state_reg;
      ready = 1'b0;
      done_tick = 1'b0;
      p2s_next = p2s_reg;
      bcd0_next = bcd0_reg;
      bcd1_next = bcd1_reg;
      bcd2_next = bcd2_reg;
      bcd3_next = bcd3_reg;
      n_next = n_reg;
      case (state_reg)
         idle:
            begin
               ready = 1'b1;
               if (start)
                  begin
                     state_next = op;
                     bcd3_next = 0;
                     bcd2_next = 0;
                     bcd1_next = 0;
                     bcd0_next = 0;
                     n_next = 4'b1101; // index
                     p2s_next = bin;   // shift register
                     state_next = op;
                  end
            end
         op:
            begin
               // shift in binary bit
               p2s_next = p2s_reg << 1;
               // shift 4 BCD digits
               //{bcd3_next, bcd2_next, bcd1_next, bcd0_next}=
               //{bcd3_tmp[2:0], bcd2_tmp, bcd1_tmp, bcd0_tmp,
               // p2s_reg[12]}

               bcd0_next = {bcd0_tmp[2:0], p2s_reg[12]};
               bcd1_next = {bcd1_tmp[2:0], bcd0_tmp[3]};
               bcd2_next = {bcd2_tmp[2:0], bcd1_tmp[3]};
               bcd3_next = {bcd3_tmp[2:0], bcd2_tmp[3]};
               n_next = n_reg - 1;
               if (n_next==0)
                   state_next = done;
            end
         done:
            begin
               done_tick = 1'b1;
               state_next = idle;
            end
         default: state_next = idle;
      endcase
   end

   // data path function units
   assign bcd0_tmp = (bcd0_reg > 4) ? bcd0_reg+3 : bcd0_reg;
```

```
   assign bcd1_tmp = (bcd1_reg > 4) ? bcd1_reg+3 : bcd1_reg;
   assign bcd2_tmp = (bcd2_reg > 4) ? bcd2_reg+3 : bcd2_reg;
   assign bcd3_tmp = (bcd3_reg > 4) ? bcd3_reg+3 : bcd3_reg;

   // output
   assign bcd0 = bcd0_reg;
   assign bcd1 = bcd1_reg;
   assign bcd2 = bcd2_reg;
   assign bcd3 = bcd3_reg;

endmodule
```

6.3.4 Period counter

A period counter measures the period of a periodic input waveform. One way to construct the circuit is to count the number of clock cycles between two rising edges of the input signal. Since the frequency of the system clock is known, the period of the input signal can be derived accordingly. For example, if the frequency of the system clock is f and the number of clock cycles between two rising edges is N, the period of the input signal is $N * \frac{1}{f}$.

The design in this subsection measures the period in milliseconds. Its ASMD chart is shown in Figure 6.12. The period counter takes a measurement when the `start` signal is asserted. We use a rising-edge detection circuit to generate a one-clock-cycle tick, edge, to indicate the rising edge of the input waveform. After `start` is asserted, the FSMD moves to the `waite` state to wait for the first rising edge of the input. It then moves to the `count` state when the next rising edge of the input is detected. In the `count` state, we use two registers to keep track of the time. The `t` register counts for 50,000 clock cycles, from 0 to 49,999, and then wraps around. Since the period of the system clock is 20 ns, the `t` register takes 1 ms to circulate through 50,000 cycles. The p register counts in terms of milliseconds. It is incremented once when the `t` register reaches 49,999. When the FSMD exits the `count` state, the period of the input waveform is stored in the p register and its unit is milliseconds. The FSMD asserts the `done_tick` signal in the `done` state, as in previous examples.

The code follows the ASMD chart and is shown in Listing 6.7. We use a constant, `CLK_MS_COUNT`, for the boundary of the millisecond counter. It can be replaced if a different measurement unit is desired.

Listing 6.7 Period counter

```
module period_counter
   (
    input wire clk, reset,
    input wire start, si,
    output reg ready, done_tick,
    output wire [9:0] prd
   );

   // symbolic state declaration
   localparam [1:0]
      idle  = 2'b00,
      waite = 2'b01,
      count = 2'b10,
```

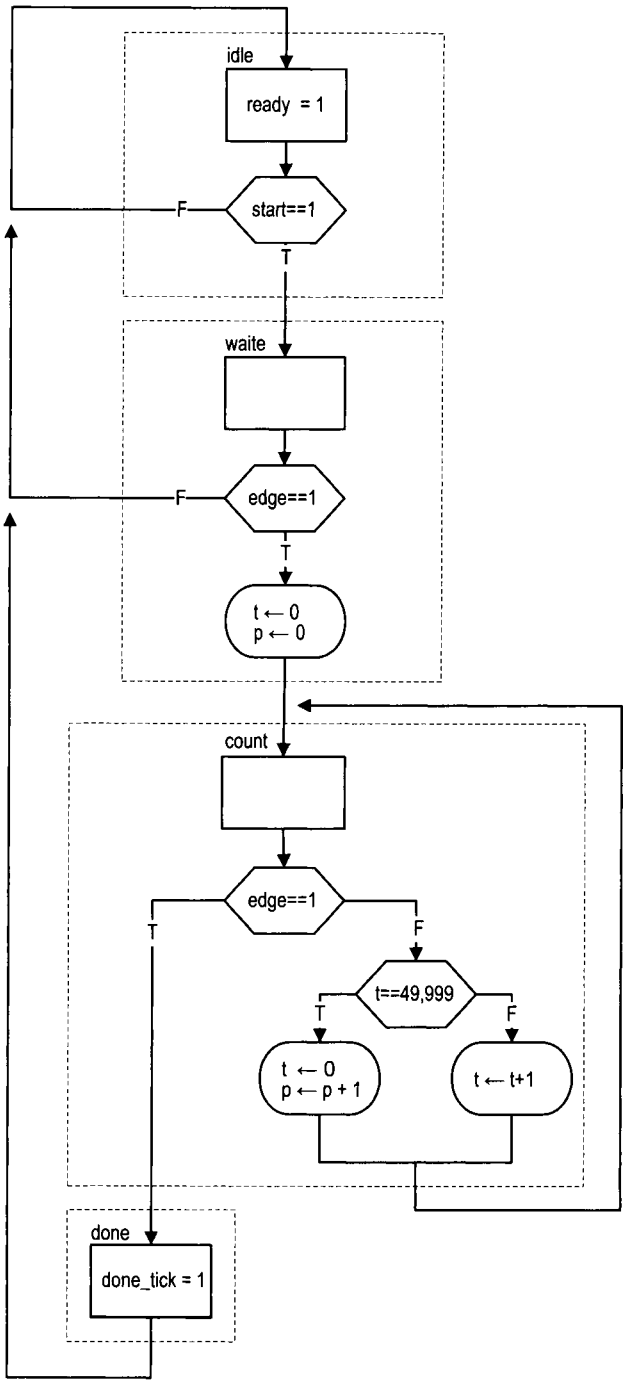


Figure 6.12 ASMD chart of a period counter.

```
   done  = 2'b11;

// constant declaration
localparam CLK_MS_COUNT= 50000; // 1 ms tick

// signal declaration
reg [1:0] state_reg, state_next;
reg [15:0] t_reg, t_next;   // up to 50000
reg [9:0] p_reg, p_next;    // up to 1 sec
reg delay_reg;
wire edg;

// body
// FSMD state & data registers
always @(posedge clk, posedge reset)
   if (reset)
      begin
         state_reg <= idle;
         t_reg <= 0;
         p_reg <= 0;
         delay_reg <= 0;
      end
   else
      begin
         state_reg <= state_next;
         t_reg <= t_next;
         p_reg <= p_next;
         delay_reg <= si;
      end

// rising-edge tick
assign edg = ~delay_reg & si;

// FSMD next-state logic
always @*
begin
   state_next = state_reg;
   ready = 1'b0;
   done_tick = 1'b0;
   p_next = p_reg;
   t_next = t_reg;
   case (state_reg)
      idle:
          begin
             ready = 1'b1;
             if (start)
                state_next = waite;
          end
      waite: // wait for the first edge
         if (edg)
            begin
               state_next = count;
               t_next = 0;
```

```
                p_next = 0;
            end
        count:
            if (edg)  // 2nd edge arrived
                state_next = done;
            else      // otherwise count
                if (t_reg == CLK_MS_COUNT-1) // 1 ms tick
                    begin
                        t_next = 0;
                        p_next = p_reg + 1;
                    end
                else
                    t_next = t_reg + 1;
        done:
            begin
                done_tick = 1'b1;
                state_next = idle;
            end
        default: state_next = idle;
    endcase
end

//ouput
assign prd = p_reg;

endmodule
```

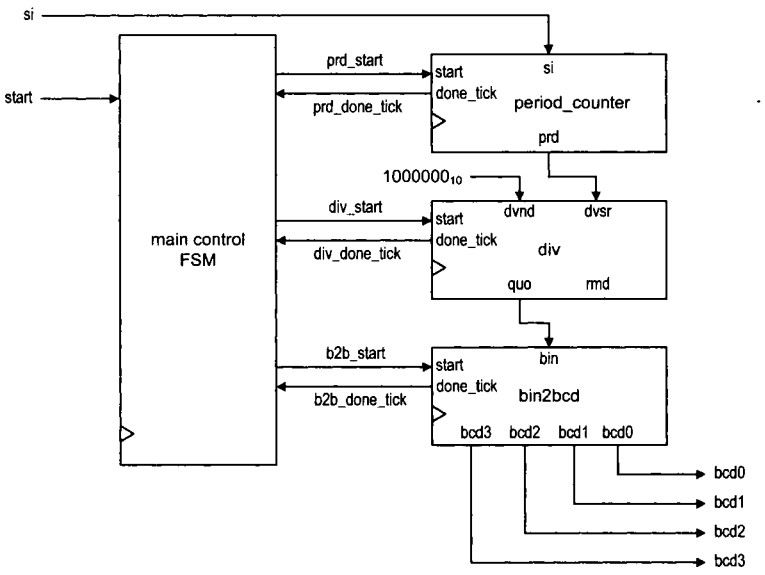
6.3.5 Accurate low-frequency counter

A frequency counter measures the frequency of a periodic input waveform. The common way to construct a frequency counter is to count the number of input pulses in a fixed amount of time, say, 1 second. Although this approach is fine for high-frequency input, it cannot measure a low-frequency signal accurately. For example, if the input is around 2 Hz, the measurement cannot tell whether it is 2.123 Hz or 2.567 Hz. Recall that the frequency is the reciprocal of the period (i.e., $frequency = \frac{1}{period}$). An alternative approach is to measure the period of the signal and then take the reciprocal to find the frequency. We use this approach to implement a low-frequency counter in this subsection.

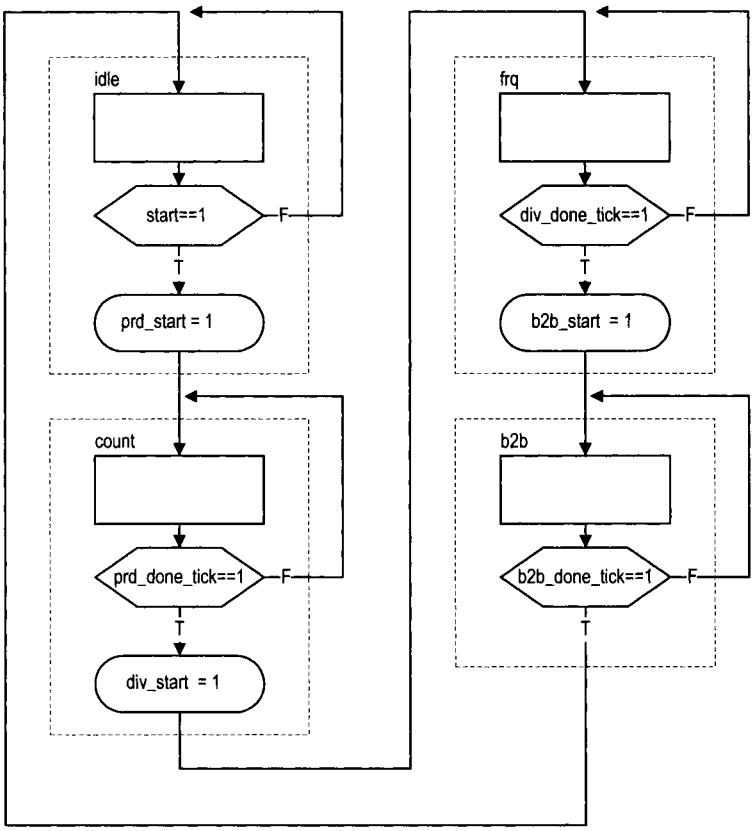
This design example demonstrates how to use the previously designed parts to construct a large system. For simplicity, we assume that the frequency of the input is between 1 and 10 Hz (i.e., the period is between 100 and 1000 ms). The operation of this circuit includes three tasks:

1. Measure the period.
2. Find the frequency by performing a division operation.
3. Convert the binary number to BCD format.

We can use the period counter, division circuit, and binary-to-BCD converter to perform the three tasks and create another FSM as the master control to sequence and coordinate the operation of the three circuits. The block diagram is shown in Figure 6.13(a), and the ASM chart of the master control is shown in Figure 6.13(b). The FSM uses the `start` and `done_tick` signals of these circuits to initialize each task and to detect completion of the task. The code is shown in Listing 6.8.



(a) Top-level block diagram



(b) ASM chart of main control

Figure 6.13 Accurate low-frequency counter.

Listing 6.8 Low-frequency counter

```
module low_freq_counter
   (
    input wire clk, reset,
    input wire start, si,
    output wire [3:0] bcd3, bcd2, bcd1, bcd0
   );

   // symbolic state declaration
   localparam  [1:0]
               idle  = 2'b00,
               count = 2'b01,
               frq   = 2'b10,
               b2b   = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   wire [9:0] prd;
   wire [19:0] dvsr, dvnd, quo;
   reg prd_start, div_start, b2b_start;
   wire prd_done_tick, div_done_tick, b2b_done_tick;

   //=================================================
   // component instantiation
   //=================================================
   // instantiate period counter
   period_counter prd_count_unit
      (.clk(clk), .reset(reset), .start(prd_start), .si(si),
       .ready(), .done_tick(prd_done_tick), .prd(prd));
   // instantiate division circuit
   div #(.W(20), .CBIT(5)) div_unit
      (.clk(clk), .reset(reset), .start(div_start),
       .dvsr(dvsr), .dvnd(dvnd), .quo(quo), .rmd(),
       .ready(), .done_tick(div_done_tick));
   // instantiate binary-to-BCD convertor
   bin2bcd b2b_unit
      (.clk(clk), .reset(reset), .start(b2b_start),
       .bin(quo[12:0]), .ready(), .done_tick(b2b_done_tick),
       .bcd3(bcd3), .bcd2(bcd2), .bcd1(bcd1), .bcd0(bcd0));
   // signal width extension
   assign dvnd = 20'd1000000;
   assign dvsr = {10'b0, prd};

   //=================================================
   // master FSM
   //=================================================
   always @(posedge clk, posedge reset)
     if (reset)
        state_reg <= idle;
     else
        state_reg <= state_next;

   always @*
```

```
   begin
      state_next = state_reg;
      prd_start = 1'b0;
      div_start = 1'b0;
      b2b_start = 1'b0;
      case (state_reg)
         idle:
            if (start)
               begin
                  prd_start = 1'b1;
                  state_next = count;
               end
         count:
            if (prd_done_tick)
               begin
                  div_start = 1'b1;
                  state_next = frq;
               end
         frq:
            if (div_done_tick)
               begin
                  b2b_start = 1'b1;
                  state_next = b2b;
               end
         b2b:
            if (b2b_done_tick)
               state_next = idle;
      endcase
   end

endmodule
```

6.4 BIBLIOGRAPHIC NOTES

FSMD is usually discussed in the context of *high-level synthesis*. *Principles of Digital Design* by D. D. Gajski contains a comprehensive chapter discussing relevant issues and algorithms of FSMD design and implementation.

6.5 SUGGESTED EXPERIMENTS

6.5.1 Alternative debouncing circuit

Consider the alternative debouncing circuit in Experiment 5.5.2. Redesign the circuit using the RT methodology:

1. Derive the ASMD chart for the circuit.
2. Derive the HDL code based on the ASMD chart.
3. Replace the debouncing circuit in Section 6.2.5 with the alternative design and verify its operation.

6.5.2 BCD-to-binary conversion circuit

A BCD-to-binary conversion converts a BCD number to the equivalent binary representation. Assume that the input is an 8-bit signal in BCD format (i.e., two BCD digits) and the output is a 7-bit signal in binary representation. Follow the procedure in Section 6.3.3 to design a BCD-to-binary conversion circuit:

1. Derive the conversion algorithm and ASMD chart.
2. Derive the HDL code based on the ASMD chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.

6.5.3 Fibonacci circuit with BCD I/O: design approach 1

To make the Fibonacci circuit more user friendly, we can modify the circuit to use the BCD format for the input and output. Assume that the input is an 8-bit signal in BCD format (i.e., two BCD digits) and the output is displayed as four BCD digits on the seven-segment LED display. Furthermore, the LED will display "9999" if the resulting Fibonacci number is larger than 9999 (i.e., overflow). The operation can be done in three steps: convert input to the binary format, compute the Fibonacci number, and convert the result back to BCD format.

The first design approach is to follow the procedure outlined in Section 6.3.5. We first construct three smaller subsystems, which are the BCD-to-binary conversion circuit, Fibonacci circuit, and binary-to-BCD conversion circuit, and then use a master FSM to control the overall operation. Design the circuit as follows:

1. Implement the BCD-to-binary conversion circuit in Experiment 6.5.2.
2. Modify the Fibonacci number circuit in Section 6.3.1 to include an output signal to indicate the overflow condition.
3. Derive the top-level block diagram and the master control FSM state diagram.
4. Derive the HDL code.
5. Derive a testbench and use simulation to verify operation of the code.
6. Synthesize the circuit, program the FPGA, and verify its operation.

6.5.4 Fibonacci circuit with BCD I/O: design approach 2

An alternative to the "subsystem approach" in Experiment 6.5.3 is to integrate the three subsystems into a single system and derive a customized FSMD for this particular application. The approach eliminates the overhead of the control FSM and provides opportunities to share registers among the three tasks. Design the circuit as follows:

1. Redesign the circuit of Experiment 6.5.3 using one FSMD. The design should eliminate all unnecessary circuits and states, such as the various `done_tick` signals and the `done` states, and exploit the opportunity to share and reuse the registers in different steps.
2. Derive the ASMD chart.
3. Derive the HDL code based on the ASMD chart.
4. Derive a testbench and use simulation to verify operation of the code.
5. Synthesize the circuit, program the FPGA, and verify its operation.
6. Check the synthesis report and compare the number of LEs used in the two approaches.
7. Calculate the number of clock cycles required to complete the operation in the two approaches.

6.5.5 Auto-scaled low-frequency counter

The operation of the low-frequency counter in Section 6.3.5 is very restricted. The frequency range of the input signal is limited between 1 and 10 Hz. It loses accuracy when the frequency is beyond this range. Recall that the accuracy of this frequency counter depends on the accuracy of the period counter of Section 6.3.5, which counts in terms of millisecond ticks. We can modify the `t` counter to generate a microsecond tick (i.e., counting from 0 to 49) and increase the accuracy 1000-fold. This allows the range of the frequency counter to increase to 9999 Hz and still maintain at least four-digit accuracy.

Using a microsecond tick introduces more than four accuracy digits for low-frequency input, and the number must be shifted and truncated to be displayed on the seven-segment LED. An auto-scaled low-frequency counter performs the adjustment automatically, displays the four most significant digits, and places a decimal point in the proper place. For example, according to their range, the frequency measurements will be shown as "1.234", "12.34", "123.4", or "1234".

The auto-scaled low-frequency counter needs an additional BCD adjustment circuit. It first checks whether the most significant BCD digit (i.e., the four MSBs) of a BCD sequence is zero. If this is the case, the circuit shifts the BCD sequence to the left four positions and increments the decimal point counter. The operation is repeated until the most significant BCD digit is not "0000".

The complete auto-scaled low-frequency counter can be implemented as follows:

1. Modify the period counter to use the microsecond tick.
2. Extend the size of the binary-to-BCD conversion circuit.
3. Derive the ASMD chart for the BCD adjustment circuit and the HDL code.
4. Modify the control FSM to include the BCD adjustment in the last step.
5. Design a simple decoding circuit that uses the decimal-point counter's output to activate the desired decimal point of the seven-segment LED display.
6. Derive a testbench and use simulation to verify operation of the code.
7. Synthesize the circuit, program the FPGA, and verify its operation.

6.5.6 Reaction timer

Eye–hand coordination is the ability of the eyes and hands to work together to perform a task. A reaction timer circuit measures how fast a human hand can respond after a person sees a visual stimulus. This circuit operates as follows:

1. The circuit has three input pushbuttons, corresponding to the `clear`, `start`, and `stop` signals. It uses a single discrete LED as the visual stimulus and displays relevant information on the seven-segment LED display.
2. A user pushes the `clear` button to force the circuit to return to the initial state, in which the seven-segment LED shows a welcome message, "HI," and the stimulus LED is off.
3. When ready, the user pushes the `start` button to initiate the test. The seven-segment LED goes off.
4. After a random interval between 2 and 15 seconds, the stimulus LED goes on and the timer starts to count upward. The timer increases every millisecond and its value is displayed in the format of "0.000" second on the seven-segment LED.
5. After the stimulus LED goes on, the user should try to push the `stop` button as soon as possible. The timer pauses counting once the `stop` button is asserted. The seven-

segment LED shows the reaction time. It should be around 0.15 to 0.30 second for most people.

6. If the `stop` button is not pushed, the timer stops after 1 second and displays "1.000".
7. If the `stop` button is pushed before the stimulus LED goes on, the circuit displays "9.999" on the seven-segment LED and stops.

Design the circuit as follows:

1. Derive the ASMD chart.
2. Derive the HDL code based on the ASMD chart.
3. Synthesize the circuit, program the FPGA, and verify its operation.

6.5.7 Babbage difference engine emulation circuit

The Babbage difference engine is a mechanical digital computation device designed to tabulate a polynomial function. It was proposed by Charles Babbage, an English mathematician, in the nineteenth century. The engine is based on Newton's method of differences and avoids the need for multiplication. For example, consider a second-order polynomial $f(n) = 2n^2 + 3n + 5$. We can find the difference between $f(n)$ and $f(n-1)$:

$$f(n) - f(n-1) = 4n + 1$$

Assume that n is an integer and $n \geq 0$. The $f(n)$ can be defined recursively as

$$f(n) = \begin{cases} 5 & \text{if } n = 0 \\ f(n-1) + 4n + 1 & \text{if } n > 0 \end{cases}$$

This process can be repeated for the $4n + 1$ expression. Let $g(n) = 4n + 1$. We can find the difference between $g(n)$ and $g(n-1)$:

$$g(n) - g(n-1) = 4$$

The $g(n)$ can be defined recursively as

$$g(n) = \begin{cases} 5 & \text{if } n = 1 \\ g(n-1) + 4 & \text{if } n > 1 \end{cases}$$

and $f(n)$ can be rewritten as

$$f(n) = \begin{cases} 5 & \text{if } n = 0 \\ f(n-1) + g(n) & \text{if } n > 0 \end{cases}$$

Note that only additions are involved in the recursive definitions of $f(n)$ and $g(n)$.

Based on the definition of the last two recursive equations, we can derive an algorithm to compute $f(n)$. Two temporary registers are needed to keep track of the most recently calculated $f(n)$ and $g(n)$, and two additions are needed to update $f(n)$ and $g(n)$. Assume that n is a 6-bit input and interpreted as an unsigned integer. Design this circuit using the RT methodology:

1. Derive the ASMD chart.
2. Derive the HDL code based on the ASMD chart.
3. Derive a testbench and use simulation to verify operation of the code.
4. Synthesize the circuit, program the FPGA, and verify its operation.
5. Let $h(n) = n^3 + 2n^2 + 2n + 1$. Use the method above to find the recursive representation of $h(n)$ (note that three levels of recursive equations are needed for a three-order polynomial). Repeat steps 1 to 4.

CHAPTER 7

SELECTED TOPICS OF VERILOG

Since the main focus of this book is on digital design, we just introduce the minimal subset of Verilog and rely on some simple guidelines and templates. In this chapter, we examine several selected Verilog topics in more detail. Except for the last section, which provides an overview of simulation-related constructs, these topics are related to synthesis and help us to develop more sophisticated codes. This chapter can be skipped without affecting the remaining chapters.

7.1 BLOCKING VERSUS NONBLOCKING ASSIGNMENT

There are two kinds of assignments that can be used in an always block: *blocking assignment* and *nonblocking assignment*. Three simple guidelines were given in the earlier chapters:

- Separate the circuit into registers and combinational circuits.
- Select a proper template for the registers, which use nonblocking assignments inside.
- Use blocking assignments to describe the combinational circuits.

We examine the two kinds of assignments and explain the rationale behind the guidelines in this section, and introduce an alternative coding style in the next section.

7.1.1 Overview

Blocking assignment The basic syntax of a blocking assignment is

```
[var] = [expression];
```

When the assignment is executed, the right-hand-side expression is evaluated and assigned to the left-hand-side variable without interruption from any other statements. Thus, it "blocks" the other assignments until execution of the current assignment is completed. The behavior of the blocking assignment is similar to the variable assignment in the C language.

Nonblocking assignment The basic syntax of a nonblocking assignment is

```
[var] <= [expression];
```

The behavior of a nonblocking assignment is more subtle and can best be explained from a hardware's perspective. Recall that an always block can be thought of as an abstract hardware part. Timing control constructs can be added to the block to model the propagation delays. When there is no explicit timing control, as in our synthesizable codes, an implicit hypothetical time step is used to model the delay. When an always block is activated, the right-hand-side expressions of nonblocking assignments are evaluated at the beginning of the time step. When the execution reaches the end of the always block (i.e., at the end of the time step), the evaluated values are assigned to the left-hand-side variables of the nonblocking assignment. The assignment is known as "nonblocking" since other statements can be executed between the evaluation and the assignment.

Let x be the variable assigned in a nonblocking assignment. While the actual scheduling in the Verilog model is quite complex, the behavior of a nonblocking assignment can be interpreted as follows:

- The value of x is assigned to x_{entry} in the beginning of the always block.
- x_{exit} replaces x in left-hand-side variable.
- x_{entry} replaces x in right-hand-side expressions.
- The value of x_{exit} is assigned to x at the end of the always block.

An interpretation is shown in the comments of the following code segment:

```
always @*
begin              // x_entry = x
. . .
y <= x & ...       // y = x_entry & ...
x <= ...           // x_exit = ...
. . .
end                // x = x_exit
```

Example To understand the difference between the blocking and nonblocking assignments, let us reconsider the three-input and circuit discussed in Section 3.3.4. The code is repeated in Listing 7.1. It uses blocking assignments and the inferred circuit is shown in Figure 3.3(a).

Listing 7.1 And circuit using blocking assignments

```
module and_block
  (
  input wire a, b, c,
  output reg y
);

  always @*
  begin
     y = a;
```

```
10      y = y & b;
        y = y & c;
    end

endmodule
```

The behavior of the assignments is similar to the sequential statements in the C language and y gets the values of a & b & c in the end. Note that the code is just for demonstration purposes. It is a poor practice to describe hardware using sequential semantics.

If we replace the blocking assignments with nonblocking assignments, the revised code is shown in Listing 7.2. The interpretation of the use of y is shown as comments.

Listing 7.2 And circuit using nonblocking assignments

```
module and_nonblock
  (
  input wire a, b, c,
  output reg y
5 );

  always @*
  begin                  // y_entry = y
     y <= a;             // y_exit = a
10   y <= y & b;         // y_exit = y_entry & b
     y <= y & c;         // y_exit = y_entry & c
  end                    // y = y_exit

endmodule
```

Note that the first two assignments have no effect and the code is the same as

```
always @*
   y <= y & c;
```

The corresponding circuit diagram is shown in Figure 3.3(b) and it is not the desired circuit.

7.1.2 Combinational circuit

The example of the previous subsection is an extreme case. Except for the default value, most codes for combinational circuits do not assign the same variable multiple times. Both blocking and nonblocking assignments can be used to describe the same circuit. However, there are subtle differences. The following example explains the differences. Let us consider the 1-bit equality circuit discussed in Section 1.2. The revised code using blocking assignments is shown in Listing 7.3. We explicitly list the variables in the sensitivity list.

Listing 7.3 Equality circuit using blocking assignments

```
module eq1_block
  (
  input wire i0, i1,
  output reg eq
5 );

  reg p0, p1;
```

```
    always @(i0,i1) // only i0 and i1 in sensitivity list
10  // the order of statements is important
    begin
       p0 = ~i0 & ~i1;
       p1 = i0 & i1;
       eq = p0 | p1;
15  end

    endmodule
```

Note that the sensitivity list consists of only i0 and i1. When one of them changes, the always block is activated, p0, p1, and eq are evaluated sequentially, and eq is updated at the end of the first time step.

The order of the statements is important. Assume that we move the last statement to the beginning:

```
always @(i0,i1)
begin
   eq = p0 | p1;
   p0 = ~i0 & ~i1;
   p1 = i0 & i1;
end
```

In the first statement, since p0 and p1 have not yet been assigned new values, the values from the *previous activation* will be used. The previous values infer latches and thus the code is not correct.

We can replace the blocking assignments with nonblocking assignments, as shown in Listing 7.4. The interpretations of these assignments are shown as comments.

Listing 7.4 Equality circuit using nonblocking assignments

```
module eq1_non_block
  (
  input wire i0, i1,
  output reg eq
5 );

  reg p0, p1;

  always @(i0,i1,p0,p1)  // p0, p1 also in sensitivity list
10 // the order of statements is not important
  begin                  // p0_entry = p0; p1_entry = p1;
     p0 <= ~i0 & ~i1;    // p0_exit = i0 & i1;
     p1 <= i0 & i1;      // p1_exit = i0 & i1
     eq <= p0 | p1;      // eq_exit = p0_entry | p1_entry
15 end                   // eq = eq_exit; p0 = p0_exit; p1 = p1_exit;

  endmodule
```

Note that p0 and p1 are also included in the sensitivity list. When i0 or i1 changes, the always block is activated and the new values are assigned to p0 and p1 in the end of the first time step. Since eq is based on the old values of p0 and p1 (i.e., $\mathrm{p0}_{entry}$ and $\mathrm{p1}_{entry}$), it remains the same. After completion of the execution of the current time step, the always block is activated again because p0 and p1 change (and this is the reason that p0 and p1 are

included in the sensitivity list). The eq variable is updated with the new values of p0 and p1 at the end of the second time step. Note that the result will be the same if we change the order of these statements.

While both codes describe the same circuits, it takes more time to simulate the code with nonblocking assignments. Because of this, the guideline recommends using blocking assignments to describe combinational circuits.

7.1.3 Memory element

In the memory element templates in Section 4.2, nonblocking assignments are used to infer memory. For example, the code for a D FF is

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

It is possible to infer a memory element using a blocking assignment, as in

```
always @(posedge clk)
   q = d;
```

Although the code works properly for an isolated FF, there are some subtle problems when multiple registers interact with each other.

Consider two registers that switch data in every clock cycle. With blocking assignments, the code becomes

```
always @(posedge clk)
   a = b;

always @(posedge clk)
   b = a;
```

At the rising edge of clk, both always blocks are activated and operated in parallel. The two operations should be completed in a time step. According to the Verilog standard, the execution of the two always blocks can be scheduled in any order. If the first always block is executed first, a gets the value of b immediately because of the blocking assignment. When the second always block is executed, b gets the updated value of a, which is its original value and thus its value remains the same. Similarly, a gets its original value if the second always block is executed first. This is known as a *race condition* in Verilog. From Verilog's point of view, both results are valid.

Now let us revise the code with nonblocking assignments (the **begin** and **end** delimiters are added to accommodate the comments):

```
always @(posedge clk)
begin          // b_entry = b
   a <= b;     // a_exit = b_entry
end            // a = a_exit

always @(posedge clk)
begin          // a_entry = a
   b <= a;     // b_exit = a_entry
end            // b = b_exit
```

The interpretation of blocking assignment is shown in the comments. Since the original entry values are used in assignments, both a and b get the correct values regardless of the order of execution.

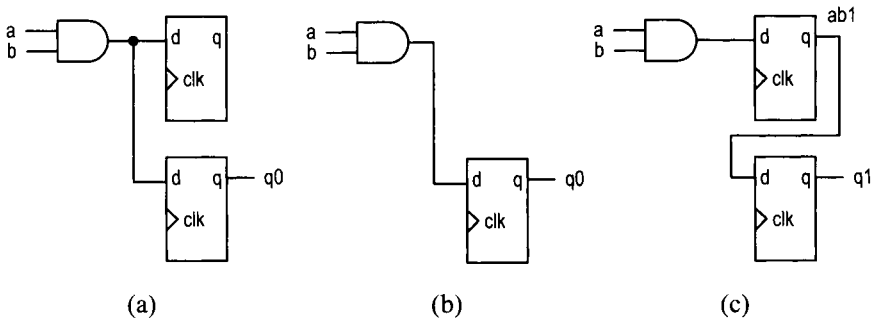


Figure 7.1 Circuits inferred by mixed assignment.

Because the blocking assignments model the desired behavior and avoid the race condition, the templates in Section 4.2 always use nonblocking assignments to infer FFs and registers.

7.1.4 Sequential circuit with mixed blocking and nonblocking assignments

The memory element templates discussed in Section 4.2 are the simplest sequential codes. It is possible to put multiple assignments, including both blocking and nonblocking assignments, in the same always block. We use a simple example to explain the behaviors of various combinations and to better understand the assignments.

Consider the circuit in Figure 7.1(b). It performs the and operation over a and b and stores the result to a D FF at the rising edge of the clock. Based on our previous approach, we can separate memory and the combinational circuit and derive the two-segment code, as shown in Listing 7.5.

Listing 7.5 Two-segment implementation

```
module ab_ff_2seg
   (
    input wire clk,
    input wire a, b,
    output reg q
   );

   reg q_next;

   // D FF
   always @(posedge clk)
      q <= q_next;

   // combinational circuit
   always @*
      q_next = a & b;

endmodule
```

Alternatively, we can combine the two segments and describe the circuit in a single always block. Six attempts, with various combinations of blocking and nonblocking assignments, are made in Listing 7.6.

Listing 7.6 Mixed assignment example

```
module ab_ff_all
   (
    input wire clk,
    input wire a, b,
    output reg q0, q1, q2, q3, q4, q5
   );

   reg ab0, ab1, ab2, ab3, ab4, ab5;

   // attempt 0
   always @(posedge clk)
   begin
      ab0 = a & b;
      q0 <= ab0;
   end

   // attempt 1
   always @(posedge clk)
   begin                  // ab1_entry = ab1; q1_entry = q1;
      ab1 <= a & b;       // ab1_exit = a & b
      q1 <= ab1;          // q1_exit = ab1_entry
   end                    // ab1 = ab1_exit; q1 = q1_exit

   // attempt 2
   always @(posedge clk)
   begin
      ab2 = a & b;
      q2 = ab2;

   end

   // attempt 3 (switch the order of attempt 0)
   always @(posedge clk)
   begin
      q3 <= ab3;
      ab3 = a & b;
   end

   // attempt 4 (switch the order of attempt 1)
   always @(posedge clk)
   begin                  // ab4_entry = ab4; q4_entry = q4;
      q4 <= ab4;          // q4_exit = ab4_entry
      ab4 <= a & b;       // ab4_exit = a & b
   end                    // ab4 = ab4_exit; q4 = q4_exit

   // attempt 5 (switch the order of attempt 2)
   always @(posedge clk)
   begin
```

```
        q5 = ab5;
50      ab5 = a & b;
    end

endmodule
```

In attempt 0, assignments to ab0 and q0 infer two registers initially, one to store the registered ab0 and one to store the registered q0. Since ab0 is updated immediately by the blocking assignment, q0 gets the value of a & b. The corresponding circuit diagram is shown in Figure 7.1(a). Since ab0 is not used outside the always block, the registered ab0 output is not needed and thus the corresponding register can be removed. The resulting diagram is shown in Figure 7.1(b), which is the desired circuit.

In attempt 1, a blocking assignment is used for ab1. The corresponding interpretation is shown in the comments. Note that q1 gets $\texttt{ab1}_{entry}$, not $\texttt{ab1}_{exit}$. The $\texttt{ab1}_{entry}$ is the previous stored value of ab1 and corresponds to the registered output. The corresponding diagram is shown in Figure 7.1(c). An unintended input buffer is inferred and the storage of a & b is delayed by one clock cycle.

In attempt 2, blocking assignments are used for both ab2 and q2. The circuit inferred is identical to that in attempt 0, as shown in Figure 7.1(a) and (b). Since using blocking assignments to infer FFs may introduce a race condition, as discussed in Section 7.1.3, this type of code is not recommended.

For demonstration purposes, let us examine what happens after switching the order of the assignments of attempts 0, 1, and 2. The results are shown in attempts 3, 4, and 5. In attempt 3, ab3 is used before it is assigned a new value. Thus, q3 gets the "previous value" from the earlier activation. The value is stored in a register and corresponds to the registered a & b. The inferred circuit corresponds to the diagram in Figure 7.1(c). In attempt 4, switching the order has no effect on the code, as explained by the interpretation in the comments. It is identical to the code in attempt 1. In attempt 5, ab5 is used before it is assigned a new value and thus q5 gets the registered a & b. It infers a circuit identical to that in attempt 3.

In summary, only the code in attempt 0 describes the desired circuit correctly and reliably.

7.2 ALTERNATIVE CODING STYLE FOR SEQUENTIAL CIRCUIT

Our sequential code template follows the block diagram in Figure 4.2 and separates the register to an individual code segment. With an understanding of blocking and nonblocking assignments, we can merge the register and the next-state logic into a single always block. This style of coding tends to be more compact. The code should follow the approach of attempt 1 in Section 7.1.4:

- Use blocking assignments to obtain intermediate results of the next-state logic. These assignments should be sequenced in proper order.
- Use nonblocking assignments to assign the intermediate results to registers.

In the following subsections, we use several examples to illustrate this style.

7.2.1 Binary counter

The free-running counter is discussed in Section 4.3.2. We can revise the code in Listing 4.9 to combine the next-state logic and the register, as shown in Listing 7.7.

Listing 7.7 Free-running binary counter with merged register and next-state logic

```
module bin_counter_merge
   #(parameter N=8)
   (
    input wire clk, reset,
    output wire max_tick,
    output wire [N-1:0] q
   );

   //signal declaration
   reg [N-1:0] r_next, r_reg;

   // body
   // register and next-state logic
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;  // {N{1b'0}}
      else
         begin
            // next-state logic
            r_next = r_reg + 1;
            // register
            r_reg <= r_next;
         end
   // output logic
   assign q = r_reg;
   assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;

endmodule
```

Note that the output logic description

```
assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;
```

must be placed outside the always block. If it is within the block, an extra FF is inferred for `max_tick` and introduces a delay of one clock cycle.

Since `r_next` is not used in another place, we can merge the two statements

```
r_next = r_reg + 1;
r_reg <= r_next;
```

into

```
r_reg <= r_reg + 1;
```

After we replace `r_reg` with `q`, the code can be simplified further, as shown in Listing 7.8.

Listing 7.8 Free-running binary counter with compact code

```
module bin_counter_terse
   #(parameter N=8)
   (
    input wire clk, reset,
    output wire max_tick,
    output reg [N-1:0] q
   );
```

```
   // body
   always @(posedge clk, posedge reset)
      if (reset)
         q <= 0;
      else
         q <= q + 1;
   // output logic
   assign max_tick = (q==2**N-1) ? 1'b1 : 1'b0;

endmodule
```

In this code, q in the right-hand-side expression is the output of the register and q on the left-hand side is the new value, which is stored to the register at the rising edge of the next clock.

The universal binary counter in Listing 4.10 can be modified in a similar way and the code is shown in Listing 7.9.

Listing 7.9 Universal binary counter with merged register and next-state logic

```
module univ_bin_counter_merged
   #(parameter N=8)
   (
    input wire clk, reset,
    input wire syn_clr, load, en, up,
    input wire [N-1:0] d,
    output wire max_tick, min_tick,
    output reg [N-1:0] q
   );

   // body
   // register and next-state logic
   always @(posedge clk, posedge reset)
      if (reset)
         q <= 0;  //
      else if (syn_clr)
         q <= 0;
      else if (load)
         q <= d;
      else if (en & up )
         q <= q + 1;
      else if (en & ~up )
         q <= q - 1;
      // no else branch since q <= q is implicitly implied

   // output logic
   assign max_tick = (q==2**N-1) ? 1'b1 : 1'b0;
   assign min_tick = (q==0) ? 1'b1 : 1'b0;

endmodule
```

Note that the last else branch is omitted. It implies that q gets its previous value, i.e.,

```
q <= q;
```

This is exactly the desired behavior.

7.2.2 FSM

The state register and next-state logic of an FSM can be merged in a similar way. For example, consider the FSM in Listing 5.1. The revised code is shown in Listing 7.10.

Listing 7.10 FSM with merged register and next-state logic

```
module fsm_eg_merged
   (
     input wire  clk, reset,
     input wire  a, b,
     output wire y0, y1
   );

   // symbolic state declaration
   parameter [1:0] s0 = 2'b00,
                   s1 = 2'b01,
                   s2 = 2'b10;

   // signal declaration
   reg [1:0] state_reg;

   // state register and next-state logic
   always @(posedge clk, posedge reset)
      if (reset)
         state_reg <= s0;
      else
         case (state_reg)
           s0: if (a)
                 if (b)
                    state_reg <= s2;
                 else
                    state_reg <= s1;
               else
                 state_reg <= s0;
           s1: if (a)
                 state_reg <= s0;
               else
                 state_reg <= s1;
           s2: state_reg <= s0;
           default state_reg <= s0;
         endcase

   // Moore output logic
   assign y1 = (state_reg==s0) || (state_reg==s1);

   // Mealy output logic
   assign y0 = (state_reg==s0) & a & b;

endmodule
```

Since the outputs are not registered, the corresponding statements must be placed outside the always block.

7.2.3 FSMD

We can apply the same approach to an FSMD as well. Consider the division FSMD example in Listing 6.5. The revised code is shown in Listing 7.11.

Listing 7.11 Division FSMD with merged register and combinational circuit

```
module div_combined
   #(
     parameter W = 8,
               CBIT = 4    // CBIT=log2(W)+1
    )
   (
    input wire clk, reset,
    input wire start,
    input wire [W-1:0] dvsr, dvnd,
    output wire ready, done_tick,
    output wire [W-1:0] quo, rmd
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      op   = 2'b01,
      last = 2'b10,
      done = 2'b11;

   // signal declaration
   reg [1:0] state_reg;
   reg [W-1:0] rh_reg, rl_reg,  rh_tmp, d_reg;
   reg [CBIT-1:0] n_reg, n_next;
   reg q_bit;

   // fsmd registers and next-state logic
   always @(posedge clk, posedge reset)
   begin
      if (reset)
         begin
            state_reg <= idle;
            rh_reg <= 0;
            rl_reg <= 0;
            d_reg <= 0;
            n_reg <= 0;
         end
      else
         begin
            //=============================================
            // data path functional units
            // to get intermediate results
            //=============================================
            // compare and subtract circuit
```

```
            if (rh_reg >= d_reg)
               begin
                  rh_tmp = rh_reg - d_reg;
                  q_bit = 1'b1;
               end
            else
               begin
                  rh_tmp = rh_reg;
                  q_bit = 1'b0;
               end
            // index decrement circuit
            n_next = n_reg - 1;

            //=============================================
            // state and data registers and next-state logic
            //=============================================
            case (state_reg)
               idle:
                  begin
                     if (start)
                        begin
                           rh_reg <= 0;
                           rl_reg <= dvnd;    // dividend
                           d_reg <= dvsr;     // divisor
                           n_reg <= CBIT;     // index
                           state_reg <= op;
                        end
                  end
               op:
                  begin
                     // shift rh and rl left
                     rl_reg <= {rl_reg[W-2:0], q_bit};
                     rh_reg <= {rh_tmp[W-2:0], rl_reg[W-1]};
                     // decrease index
                     n_reg <= n_next;
                     if (n_next==1)
                        state_reg <= last;
                  end
               last: // last iteration
                  begin
                     rl_reg <= {rl_reg[W-2:0], q_bit};
                     rh_reg <= rh_tmp;
                     state_reg <= done;
                  end
               done:
                  state_reg <= idle;
               default: state_reg <= idle;
            endcase
         end
      end

   // output
   assign quo = rl_reg;
```

```
    assign rmd = rh_reg;
    // unregistered output
100 assign ready = (state_reg==idle);
    assign done_tick = (state_reg==done);

endmodule
```

The code is more complex and includes a section for data path functional units, which generates the intermediate results. Note that some intermediate variables, such as `n_next`, are used in multiple places later.

7.2.4 Summary

In summary, it is possible to merge the next-state logic and register in one always block. This style tends to be more compact and requires fewer variables. However, the code must be crafted carefully to avoid unintended registers. It is recommended only after we have a good comprehension of blocking and nonblocking assignments.

7.3 USE OF THE SIGNED DATA TYPE

7.3.1 Overview

Depending on the nature of an application, we can use an unsigned integer, which consists of zero and the positive numbers, or a signed integer, which consists of zero and both negative and positive numbers, in a digital system. We may even need to use both types in a complex system.

The signed integer is usually represented in 2's-complement format. A 4-bit "binary wheel" is shown in Figure 7.2, which lists the binary representations and the corresponding unsigned and signed numbers. Close observation shows that the addition and subtraction operations are identical for the two types of numbers. The addition and subtraction of a positive amount corresponds to moving clockwise and counterclockwise along the wheel. For example, "1001"+"0100" means to move four positions clockwise from "1001" and the result is "1101". In the unsigned integer format, it is interpreted as (+9) + (+4) = +13, and in the signed integer format, it is interpreted as (−7) + (+4) = −3. The overflow in addition corresponds to a move over the "threshold" of the binary wheel. Note that the thresholds are different for the unsigned and signed interpretations. It is between "1111" and "0000" for the unsigned integer and between "0111" and "1000" for the signed integer.

The behavior of a physical adder or subtractor is just like the movement in the binary wheel. The same circuit can be applied to both unsigned and signed formats as long as *all operands and the result have the same bit length.* For example, let `a`, `b`, and `sum` be three 8-bit signals. The statement

```
sum = a + b;
```

infers the same hardware and uses the same binary representations regardless of whether these signals are interpreted as unsigned or signed format. This observation is also correct in other arithmetic operations (however, it cannot be applied for nonarithmetic operations, such as relational operations or overflow status generation).

On the other hand, we need to distinguish the format when the operands or the result have different bit lengths. This is due to the different requirements in width extension. The 0's

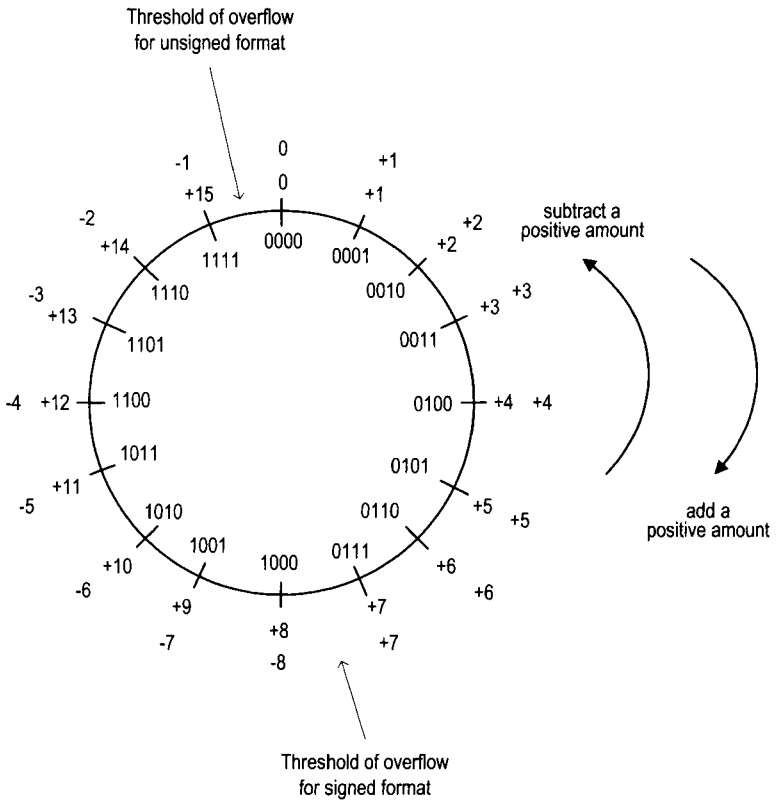


Figure 7.2 Four-bit binary wheel.

are appended to the front for the unsigned format, which is known as *zero extension*, but the sign bits are appended to the front for the signed format, which is known as *sign extension*. For example, the 4-bit representation of -5 is "1011". It becomes "1111_1011", not "0000_1011" when extended to 8 bits.

For example, let a and sum be two 8-bit signals and b be a 4-bit signal, $b_3b_2b_1b_0$. The statement

```
sum = a + b;
```

requires b to be extended to 8 bits. The extended b becomes $0000b_3b_2b_1b_0$ if it is in the unsigned format but becomes $b_3b_3b_3b_3b_3b_2b_1b_0$ if it is in the signed format. The inferred hardware for this statement consists of the width extension circuit and an adder. Since the extension circuit is different for the unsigned and signed formats, the statement infers different hardware implementations for the unsigned and signed formats.

7.3.2 Signed number in Verilog-1995

In Verilog-1995, only the **integer** data type is interpreted as a signed number, and the **reg** and **wire** data types are interpreted as unsigned numbers. Since the **integer** data type has a fixed size (usually 32 bits), it is not flexible. To achieve the signed operation, we frequently

need to manipulate the code manually. The signed and unsigned operations are illustrated in the following code segment:

```
reg [7:0] a, b;
reg [3:0] c;
reg [7:0] sum1, sum2, sum3, sum4;
. . .
// same width, can be applied to signed and unsigned
sum1 = a + b;
// automatic 0 extension
sum2 = a + c;
// manual 0 extension
sum3 = a + {4{1'b0}, c};
// manual sign extension
sum4 = a + {4{c[3]}, c};
```

In the first statement, a, b, and sum1 have identical width and thus infer the same adder circuit regardless of whether they are interpreted as unsigned or signed numbers.

In the second statement, c is only 4 bits wide. Its bit length is adjusted according to the rules discussed in Section 3.2.8. Since the **reg** type is treated as an unsigned number, zero extension is performed and four zeros are appended in front of c.

In the third statement, we manually append four zeros in front of c and achieve the same effect as in the previous statement.

In the fourth statement, we interpret the variables as signed numbers. To achieve the desired behavior, c must be sign-extended to 8 bits. This can only be done manually. In the code, we replicate the MSB of c four times (i.e., 4{c[3]}) to create the sign-extended 8-bit number.

7.3.3 Signed number in Verilog-2001

In Verilog-2001, the signed format is extended to the **reg** and **wire** data types. This is done by adding the keyword, **signed**, in declaration, as in

```
reg signed [7:0] a, b;
```

With the signed data type, the previous code segment can be revised as

```
reg signed [7:0] a, b;
reg signed [3:0] c;
reg signed [7:0] sum1, sum4;
. . .
// same width, can be applied to signed and unsigned
sum1 = a + b;
// automatic sign extension
sum4 = a + c;
```

The first statement infers a regular adder since a, b, and sum1 have identical bit length. The **signed** data type just helps us to be aware of the interpretation of the binary representation.

In the second statement, all variables in the right-hand-side expression are with the **signed** data type and c is sign-extended to 8 bits automatically. Thus, we don't need to pad the variable manually.

In a small digital system, we usually use either unsigned or signed format. However, a larger system may contain subsystems of different formats. Verilog is a loosely typed language and the unsigned and signed variables can be mixed in the same expression.

According to the Verilog standard, the sign extension is performed only if *all* variables in the right-hand-side expression are with the **signed** data type. Otherwise, zero extension is performed for *all* variables. Consider the code segment

```
reg signed [7:0] a, sum;
reg signed [3:0] b;
reg [3:0] c;
. . .
sum = a + b + c;
```

Since c is not with the **signed** data type, the variables in the right-hand-side expression, b and c, are zero extended.

Verilog consists of two system functions, **$signed( )** and **$unsigned( )**, which convert the enclosed expression to the **signed** and **unsigned** data types, respectively. For example, we can convert the data type of c in the preceding statement:

```
sum = a + b + $signed(c);
```

Now all three variables in the right-hand-side expression are with the **signed** data type and thus b and c are sign extended.

Mixed **signed** and **unsigned** data types in a complex expression can introduce subtle errors and should be avoided. If it is really necessary, the expression should be kept simple and the conversion functions should be used to ensure the consistency of the data type.

7.4 USE OF FUNCTION IN SYNTHESIS

7.4.1 Overview

In a Verilog module, some expressions may occur at many places. Instead of repeating the code, the commonly used part should be abstracted into a routine. This can be achieved by defining *functions* within a module. A Verilog function takes one or more input arguments and returns a single value. During synthesis, the functions are expanded and "flattened" and mapped to hardware. Thus, for synthesis purposes, functions should be kept simple and treated as shorthand for a complex expression. The basic syntax of a function is

```
module ...
. . .
// function defined within module
function [result_type] [func_id] ([input_arg]);
   begin
      [statements];
   end
endfunction
. . .
endmodule
```

A function is defined within the **function** and **endfunction** delimiters. The optional `[result_type]` specifies the data type of the returned result, which is usually **reg** with range or **integer**. The input arguments are declared in `[input_arg]` and the name of the function is specified by `[func_id]`. A function is described by the statements and the result is returned by a statement like

```
[func_id] = ... ;
```

7.4.2 Examples

Consider the binary-to-BCD conversion circuit in Listing 6.6. During the conversion, each BCD digit needs to be incremented in a specific way. To make the FSMD portion clear, we use a separate segment in code:

```
module ...
   . . .
   assign bcd0_tmp = (bcd0_reg > 4) ? bcd0_reg+3 : bcd0_reg;
   assign bcd1_tmp = (bcd1_reg > 4) ? bcd1_reg+3 : bcd1_reg;
   assign bcd2_tmp = (bcd2_reg > 4) ? bcd2_reg+3 : bcd2_reg;
   assign bcd3_tmp = (bcd3_reg > 4) ? bcd3_reg+3 : bcd3_reg;
   . . .
endmodule
```

Instead of repeating the same expression four times, we can define a function, `ba()`, for this purpose. The revised code segment becomes

```
module ...
   . . .
   assign bcd0_tmp = ba(bcd0_reg);
   assign bcd1_tmp = ba(bcd1_reg);
   assign bcd2_tmp = ba(bcd2_reg);
   assign bcd3_tmp = ba(bcd3_reg);
   . . .
   // function definition (ba: bcd adjust)
   function [3:1] ba(reg [3:0] bcd_in);
   begin
      ba = (bcd_in > 4) ? bcd_in + 3 : bcd_in;
   end
   endfunction
   . . .
endmodule
```

The function `ba()` (for BCD adjust) is defined in the end. It takes a 4-bit argument and returns a 4-bit result. We can use this function to replace the previous expression. In fact, we can use `bc(bcd0_reg)` to substitute `bcd0_tmp` directly and eliminate these variables from the code.

Another common application of a function is to calculate the constants whose values depend on other parameters. Consider the mod-m counter discussed in Listing 4.11. There are two parameters: `M`, which specifies the m value, and, `N`, which specifies the number of bits needed in the counter. The value of `N` is $\lceil \log_2 \mathtt{M} \rceil$ and should not be an independent parameter. A better approach is to specify `N` as a local constant and calculate its value inside the module. This can be achieved by using a function. The modified code is shown in Listing 7.12.

Listing 7.12 Mod-m counter with function

```
module mod_m_counter_fc
   #(parameter M=10) // mod-M
   (
    input wire clk, reset,
    output wire max_tick,
    output wire [log2(M)-1:0] q
   );
```

```
   //signal declaration
10 localparam N = log2(M);    // number of bits for M
   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;

   // body
15 // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
20       r_reg <= r_next;

   // next-state logic
   assign r_next = (r_reg==(M-1)) ? 0 : r_reg + 1;
   // output logic
25 assign q = r_reg;
   assign max_tick = (r_reg==(M-1)) ? 1'b1 : 1'b0;

   // log2 constant function
   function integer log2(input integer n);
30    integer i;
   begin
      log2 = 1;
      for (i = 0; 2**i < n; i = i + 1)
         log2 = i + 1;
35 end
   endfunction

endmodule
```

A function, `log2()`, which computes $\lceil \log_2(x) \rceil$, is defined inside the module and used to obtain the local parameter `N`. Since the computation is performed when the code is elaborated, the value is determined before synthesis and no physical circuit will be inferred for this function.

7.5 ADDITIONAL CONSTRUCTS FOR TESTBENCH DEVELOPMENT

Since our focus is mainly on hardware development, we examine only a small synthesizable subset of Verilog and use two basic testbench templates for verification. Although detailed coverage of the Verilog language and testbench is beyond the scope of this book, in this section we provide a brief overview of several language constructs that help us to develop a more sophisticated testbench.

Unlike the synthesizable code, the testbench code is fed to a simulator and executed on a host computer. We can include complex language constructs and sequential algorithms in the code. Many of Verilog constructs resemble those in the C language and can be used in a similar way.

7.5.1 Always block and initial block

Verilog has two types of procedural blocks: always block and initial block. An *always block* contains procedural statements inside and models an abstract circuit part. We examine one special type of always block in Section 3.3. It is intended for synthesis. The block has a sensitivity list but contains no other explicit timing control constructs. Activation and execution of the always block are trigged by the designated events of the sensitivity list.

For modeling purposes, an always block can contain timing constructs to specify the relevant propagation delays of various constructs or to wait for a specific event. The sensitivity list can sometimes be omitted. For example, we can use the following segment to model a clock signal, which alternates between 0 and 1 every 20 time units and runs forever.

```
always
begin
   clk = 1'b1;
   #20;
   clk = 1'b0;
   #20;
end
```

An *initial block* also contains procedural statements inside. However, it is executed only *once* at the beginning of simulation. The simplified syntax is

```
initial
begin
   [procedural statements]
end
```

An initial block is frequently used to set the initial values of variables. In Listing 1.7, it is used to generate the entire testing sequence. The "run-once" behavior of an initial block usually cannot be synthesized.

7.5.2 Procedural statements

Procedural statements are used within initial blocks, always blocks, functions, and tasks. Commonly used procedural statements are

- Blocking assignment
- Nonblocking assignment
- If statement
- Various case statements
- Various loop statements

We discuss the blocking and nonblocking assignments in Section 7.1 and the if and case statements in Sections 3.4 and 3.5.

Verilog supports four loop constructs: **for**, **while**, **repeat**, and **forever**. The simplified syntax of the **for** loop is

```
for ([initial_assignment]; [end_condition]; [step_assignment])
begin
   [procedural_statements;]
end
```

For example, we can clear the content of a 16-word register file:

```
integer i;
```

```
. . .
for (i=0; i<16; i=i+1)
  reg_file[i] = 0;
```

Note that the **begin** and **end** delimiters can be omitted if there is only one statement inside the body.

The simplified syntax of the **while** loop is

```
while([end_condition])
begin
   [procedural_statements;]
end
```

The statements in the loop body are repeated continuously until the condition specified by the [end_condition] expression is met. For example, the previous clearing register operation can also be done with a while loop:

```
integer i;
. . .
i=0;
while (i<16)
begin
  reg_file[i] = 0;
  i = i + 1;
end
```

The simplified syntax of the **repeat** loop is

```
repeat([number])
begin
   [procedural_statements;]
end
```

The statements in the loop body are repeated a specific number of times, which is specified by [number]. For example, the previous operation can also be done with a repeat loop:

```
integer i;
. . .
i=0;
repeat(16)
begin
  reg_file[i] = 0;
  i = i + 1;
end
```

The simplified syntax of the **forever** loop is

```
forever
begin
   [procedural_statements;]
end
```

The **forever** loop, as its name shows, repeats its body until the end of the simulation. The loop body usually contains certain timing control constructs and thus is suspended periodically. For example, the following segment is another way to describe a clock signal, which toggles its value every 10 time units and runs forever.

```
initial begin
  clk = 1'b0;
  forever
     #10 clk = ~clk;
end
```

7.5.3 Timing control

In a testbench, we must specify the time that various signals are activated and deactivated or wait for certain events or conditions. There are three timing control constructs:

- Delay control: `#[delay_time]`
- Event control: `@([event], [event], ...)`
- Wait statement: **wait** `([boolean_expression])`

In addition, a compiler directive, `'timescale`, is also related to the timing specification.

7.5.4 Delay control

Delay control is indicated by the # symbol, followed by the amount of the time unit to be delayed. It delays execution of a procedural statement by the amount specified.

If the delay control is placed on the left-hand side, execution of the *entire statement* is delayed. For example, consider the segment

```
. . .
#10 a = 1'b0;
#5  y = a | b;
. . .
```

Assume that the current simulation time is t. The statements mean that a gets 0 at $t + 10$ and after another 5 time units (i.e., at $t + 15$) the a | b expression is evaluated and the result is assigned to y.

If the delay control is placed on the right-hand side, the expression is executed immediately but the assignment to the left-hand-side variable is delayed. Consider the segment

```
. . .
#10 a = 1'b0;
    y = #5 a | b;
. . .
```

Again, a gets 0 at $t + 10$. The a | b expression is evaluated immediately (i.e., at $t + 10$) but the result is assigned to y at $t + 15$.

Instead of modeling the propagation delay, we generally use the delay control to generate a stimulus in the testbench. The following format makes the code more intuitive:

```
. . .
a = 1'b0;   // a gets 0
#10;        // the 0 value lasts 10 time units
a = 1'b1;   // a changes to 1
#5          // the 1 value lasts 5 time units
a = 1'b0;   // a changes to 0
#20         // the 0 value lasts 20 time units
. . .
```

7.5.5 Event control

Event control is indicated by the @ symbol, followed by the sensitivity list, which specified the desired events. The event control is similar to that used in an always block. An event is the occasion that a signal in the sensitivity list changes its value (i.e., a signal transition). The **posedge** and **negedge** keywords can be added to specify the desired transition edge (i.e., rising edge or falling edge). In a testbench, the execution is suspended until one of the specified events occurs. One common application of event control is to synchronize the stimulus generation with a clock signal. For example, the following segment activated the enable signal, en, for one clock cycle:

```
localparam delta=1;
. . .
@(posedge clk) // wait for the rising edge of clk
#delta;        // wait for delta to avoid hold-time violation
en = 1'b1;     // assert en to 1
@(posedge clk) // wait for the next rising edge of clk
#delta;        // wait for delta to avoid hold-time violation
en = 1'b0;     // deassert en to 0
. . .
```

Alternatively, we can also assert and deassert en at the falling edge of the clock signal:

```
. . .
@(negedge clk) // wait for the falling edge of clk
en = 1'b1;     // assert en to 1
@(negedge clk) // wait for the next falling edge of clk
en = 1'b0;     // deassert en to 0
. . .
```

7.5.6 Wait statement

The **wait** statement waits for a specific condition. The simplified syntax is

```
wait [boolean_expression]
```

Execution of the subsequent statements is suspended until the condition specified by the [boolean_expression] term is evaluated to be true. For example, we can write code like

```
wait(state==READ && mem_ready==1'b1) [statement_to_get_data];
```

We can also use the wait statement to suspend the execution. For example, we can wait for a counter to reach 15 and then activate certain signals:

```
. . .
wait(counter==4'b1111);  // wait unit counter is 15
. . .                    // continue
```

The wait statement is somewhat similar to the event control. The latter waits for the transition edges of certain signals and the former waits for a specific condition and is sometimes known as *level-sensitive*.

7.5.7 Timescale directive

Compiler directives are used to control the compiling and processing of Verilog code. They are preceded by the grave accent mark (`), which is usually located in the top-left corner of the keyboard. A timing-related directive is the `timescale directive, whose syntax is

```
'timescale [time_unit] / [time_precision]
```

The [time_unit] term specifies the unit of measurement for time and delays and the [time_precision] term specifies the "resolution" of simulation.

For example, the directive

```
'timescale 10 ns / 1 ns
```

indicates that he simulation unit is 10 ns and the resolution is 1 ns. When a delay is specified in the code, as in

```
#5 y = a & b;
```

it indicates that the actual delay is 50 ns (i.e., 5 * 10 ns).

The delay specification can be a fraction of a unit, as in

```
#5.12345 y = a & b;
```

which indicates that the actual delay is 51.2345 ns. Since the precision is 1 ns, the number is rounded to 51 ns in simulation. Finer precision can increase the accuracy of the simulation but may reduce the simulation speed.

The number portion of the [time_unit] and [time_precision] terms can be 1, 10, or 100, and the time units can be **s** (second), **ms** (millisecond), **us** (microsecond), **ns** (nanosecond), **ps** (picosecond), or **fs** (femtosecond).

7.5.8 System functions and tasks

Verilog has a set of predefined system functions and tasks. They perform system-related operations, such as simulation control and file access. Their names begin with a dollar sign (**$**). We examine several commonly used functions and tasks in this subsection.

Data type conversion functions The **$unsigned** and **$signed** functions perform the conversion between the unsigned and signed data types. Their use is discussed in Section 7.3.

Simulation time functions Simulation time functions return the current simulation time. The **$time**, **$stime**, and **$realtime** functions return the time as a 64-bit integer, a 32-bit integer, and a real number, respectively.

Simulation control tasks There are two simulation control tasks: **$finish** and **$stop**. The **$finish** task terminates the simulation and exits the simulation program. The **$stop** task suspends simulation. In ModelSim, it returns simulation to the interactive mode. In our development flow, we usually stay within the ModelSim environment to do further editing or to examine the waveform, and thus **$stop** is used in the code.

Display tasks The development flow discussed in Section 2.4 resembles doing an experiment at a lab bench. The simulated result is shown in waveform format in ModelSim, which emulates a logic analyzer used at a lab bench. An alternative is to display the results in textual format. The four main display system tasks are **$display**, **$write**, **$strobe**, and **$monitor**. They have similar syntax and display the text during simulation. In ModelSim, the text is shown in the console panel.

The format of **$display** is similar to the print function in the C language. Its simplified syntax is

```
$display([format_string], [argument], [argument], ...);
```

The [format_string] term contains regular character and "escape sequences" to specify the format of the corresponding arguments. When the string is displayed, the values of the corresponding arguments are substituted into the string and shown in the designated format. For example, in the the statement

```
$display("at %d; signal x = %b", $time, x);
```

%d and %b are escape sequences and specify that current simulation time and x are to be displayed in the decimal and binary formats, respectively. The rustling display looks like

```
at 5100; signal x = 00110001
```

The commonly used escape sequences in our simulation include %d, %b, %o, %h, %c, %s, and %g, which are for decimal, binary, octal, hexadecimal, character, string, and real number, respectively.

The **$write** task is almost identical to the **$display** task except that **$write** does not add a newline character in the end. The output of the display-related task continues from the current position. The newline character, \n, must be added to the string manually to create a line break.

Verilog incorporates the concept of a time step to model the propagation delay, as discussed in Section 7.5.7. Many activities can take place within a time step. The **$strobe** task is similar to the **$display** task. Instead of being executed immediately, the **$strobe** task is executed at the end of the current simulation time step. It avoids mismatched data display due to the race condition.

The **$monitor** task is a very versatile command. Whereas the **$display**, **$write**, or **$strobe** task displays the text once every time it is executed, the **$monitor** task displays text when an argument changes its value. The **$monitor** task provides a simple and flexible way to keep track of the simulation. For example, we can add the following segment to the testbench in Listing 1.7:

```
initial
begin
   $display("time   test_in0   test_in1   test_out");
   $monitor("%d       %b          %b         %b",
            $time, test_in0, test_in1, test_out);
end
```

The textual simulation result is displayed in the control console panel:

```
time    test_in0  test_in1  test_out
   0       00        00        1
 200       01        00        0
 400       01        11        0
 600       10        10        1
 800       10        00        0
1000       11        11        1
1200       11        01        0
```

File I/O system functions and tasks Verilog provides a set of functions and tasks to access external data files. A file can be opened and closed by the **$fopen** and **$fclose** functions. The simplified syntax of using **$fopen** is

```
[mcd_name] = $fopen("[file_name]");
```

The **$fopen** function returns a 32-bit *multichannel descriptor* associated with the file. The descriptor can be thought of as a 32-bit flag, in which each bit represents a file (i.e., a channel). The LSB is reserved for the standard output (i.e., the console). When the function is called and the file is opened successfully, it returns a descriptor value with one bit asserted. For example, 0...0010 is returned for the first opened file, 0...0100 is returned for the second opened file, and so on. The function returns all 0's if the open operation fails.

Once a file is opened, we can write data to the file with four modified display system tasks: **$fdisplay**, **$fwrite**, **$fstrobe**, and **$fmonitor**. These tasks are similar to the original ones except that a multichannel descriptor is used as the first argument, as in

```
$fdisplay([mcd_name], [format_string], ...);
```

A simple example segment is shown in Listing 7.13.

Listing 7.13 File write example

```
integer log_file, both_file;
localparam con_file=16'h0000_0001;        // console

initial
begin
   log_file = $fopen("my_log");
   if (log_file==0)
      $display("Fail to open log file"); // write console
   both_file = log_file | con_file;

   // write to both console and log file
   $fdisplay(both_file,"Simulation started");
   . . .
   // write to log file only
   $fdisplay(log_file, ...);
   . . .
   // write to both console and log file
   $fdisplay(both_file,"Simulation ended");
   $fclose(log_file);
end
```

Note that we can create a descriptor by performing a bitwise or operation over the multichannel descriptors, as for the `both_file` variable. When `both_file` is used, the text will be written to the console and the log file.

There are two simple system tasks to retrieve data from an external file: **$readmemb** and **$readmemh**. These tasks assume that the external file stores the content of a memory array and reads the content into a variable. The **$readmemb** and **$readmemh** tasks further assume that the content is in the binary and hexadecimal formats, respectively. The simplified syntax is

```
$readmemb("[file_name]",[mem_variable]);
$readmemh("[file_name]",[mem_variable]);
```

The following code segment illustrates the retrieval of an 8-by-4 memory array:

```
reg [3:0] v_mem [0:7];
. . .
$readmemb("vector.txt", v_mem);
. . .
```

The file should contain eight 4-bit binary data separated by white spaces.

With the file operation functions and tasks, it is possible to use external files to specify the test patterns and to record the simulation result. Consider the testbench in Listing 1.7. We can modify it using file operations, as shown in Listing 7.14.

Listing 7.14 Testbench based on file operation

```
`timescale 1 ns/10 ps

module eq2_file_tb;
   // signal declaration
   reg  [1:0] test_in0, test_in1;
   wire  test_out;
   integer log_file, console_file, out_file;
   reg [3:0] v_mem [0:7];
   integer i;

   // instantiate the circuit under test
   eq2_sop uut
      (.a(test_in0), .b(test_in1), .aeqb(test_out));

   initial
   begin
      // setup output file
      log_file=$fopen("eqlog.txt");
      if (!log_file)
         $display("Cannot open log file");
      console_file = 32'h0000_0001;
      out_file = log_file | console_file;

      // read test vector
      $readmemb("vector.txt", v_mem);

      // test generator iterating through 8 patterns
      for(i=0; i<8; i=i+1)
        begin
           {test_in0, test_in1} = v_mem[i];
           #200;
        end

      // stop simulation
      $fclose(log_file);
      $stop;
   end

  // text display
   initial
   begin
      $fdisplay(out_file, "      time  test_in0  test_in1  test_out");
      $fdisplay(out_file, "              (a)       (b)      (aeqb) ");
      $fmonitor(out_file, "%10d     %b         %b         %b",
                          $time, test_in0, test_in1, test_out);
   end
```

```
endmodule
```

The test patterns are specified in 4-bit binary format and are stored in the `vector.txt` file. The content of the file is

```
00_00
01_00
01_11
10_10
10_00
11_11
11_01
00_10
```

The file is read into the two-dimensional `v_mem` variable. The test pattern generator uses a for loop to iterate through the eight patterns. The simulated result is written to the console and the log file, `eqlog.txt`. The content of the log file is

```
time  test_in0  test_in1  test_out
        (a)       (b)     (aeqb)
   0     00        00        1
 200     01        00        0
 400     01        11        0
 600     10        10        1
 800     10        00        0
1000     11        11        1
1200     11        01        0
1400     00        10        0
```

The log file is a regular text file and can be examined later by any text editor.

7.5.9 User-defined functions and tasks

A comprehensive testbench can be lengthy and involved. One way to manage the complexity is to divide the code into smaller portions. The functions and tasks can help us to achieve this. We discuss Verilog functions in Section 7.4. A function takes input arguments and returns a single value. When called, a function is executed immediately and thus no timing control construct is allowed within the function.

A *task* is more flexible and versatile. It can have input, output, and bidirectional arguments and can incorporate timing control constructs. Multiple values can be returned via the output and bidirectional arguments. As with a function, a task must be declared within a module. The basic syntax of a task is

```
task [task_id] ([arg]);
   begin
      [statements];
   end
endtask
```

The `[arg]` term is the argument declaration. Its format is similar to the port declaration of a module except that the default data type is **reg** and the **wire** data type cannot be used. The example in Listing 7.15 shows the modeling of a 2-bit equality comparator using a task.

Listing 7.15 Two-bit comparator using a task

```
module eq2_task
  (
  input  wire [1:0] a, b,
  output reg aeqb
  );

  reg e0, e1;

  always @*
  begin
     equ_tsk(2, a[0], b[0], e0);
     equ_tsk(2, a[1], b[1], e1);
     aeqb = e0 & e1;
  end

// task definition
task equ_tsk
   (
    input integer delay,
    input i0, i1,
    output eq1
   );
    begin
       #delay eq1 = (~i0 & ~i1) | (i0 & i1);
    end
endtask

endmodule
```

Note that the propagation delay of the operation is specified by `#delay` and its value is passed into the task via the `delay` argument.

For comparison purposes, we rewrite the code using a function, as shown in Listing 7.16.

Listing 7.16 Two-bit comparator using a function

```
module eq2_function
  (
  input  wire [1:0] a, b,
  output reg aeqb
  );

  reg e0, e1;

  always @*
  begin
     #2 e0 = equ_fnc(a[0], b[0]);
     #2 e1 = equ_fnc(a[1], b[1]);
     aeqb = e0 & e1;
  end

// function definition
function equ_fnc(input i0, i1);
   begin
```

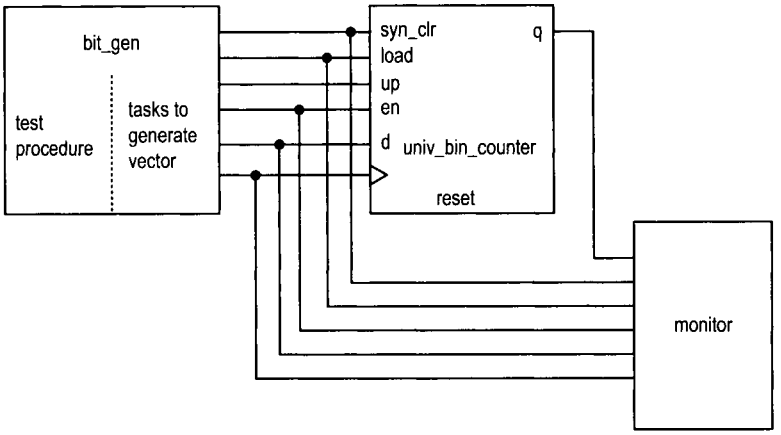


Figure 7.3 Block diagram of a comprehensive testbench.

```
        equ_fnc = (~i0 & ~i1) | (i0 & i1);
20    end
  endfunction

  endmodule
```

Note that a function cannot incorporate timing control. To achieve the same effect, the delay must only be specified in the always block.

7.5.10 Example of a comprehensive testbench

After learning additional language constructs, we can develop a more sophisticated testbench. Let us consider the testbench again for the universal binary counter in Listing 4.10. The conceptual block diagram of a new testbench is shown in Figure 7.3. There are three modules. In addition to the counter, the `bin_gen` module generates the testing vector and the `monitor` module monitors the input stimulus and the output responses.

Test vector generator module Generating test vectors directly, as in Listing 4.12, is a lengthy and tedious process. A better alternative is to develop a set of abstract procedures that correspond to various operations. This makes the code better organized and easier to comprehend. An individual procedure can be done by a task. For example, in the preceding testbench, we can define a task to perform the counter's data load operation:

```
task load_data(input wire [N-1:0] data_in);
begin
   @(negedge clk);  // wait for failing edge
   load = 1'b1;
   d = data_in;
   @(negedge clk);
   load = 1'b0;
end
endtask
```

In the task, `load` is asserted for one clock cycle between two falling edges and the data, `data_in`, is placed on `d`.

Several other tasks are defined in a similar way:

- `clr_counter_async`: clear the counter asynchronously by generating a short `reset` pulse.
- `clr_counter_sync`: clear the counter synchronously by activating the `syn_clr` signal for one clock cycle.
- `count`: enable the counter to count up or down for a certain number of cycles.
- `initialize`: set up the initial values for simulation and generate a `reset` pulse.

With these procedures, we can generate the test vector in a more abstract way:

```
initial
begin
   initialize();
   count(12, 1);          // count up 12 cycles
   count(6, 0);           // count down 6 cycles
   load_data(3'b011);     // load 011
   count(2, 1);           // count up 2 cycles
   clr_counter_sync();    // clear counter synchronously
   count(3, 1);           // count up 3 cycles
   clr_counter_async();   // clear counter asynchronously
   count(5, 1);           // count up 5 cycles
   $stop;     // stop simulation
end
```

The complete code is shown in Listing 7.17.

Listing 7.17 Test vector generator

```
module bin_gen
   #(parameter N=8, T=20)
   (
    output reg clk, reset,
    output reg syn_clr, load, en, up,
    output reg [N-1:0] d
   );


   // clock
   // clock running forever
   always
   begin
      clk = 1'b1;
      #(T/2);
      clk = 1'b0;
      #(T/2);
   end


   // test procedure
   initial
   begin
      initialize();
      count(12, 1);          // count up 12 cycles
      count(6, 0);           // count down 6 cycles
      load_data(3'b011);     // count down 6 cycles
```

```
      count(2, 1);          // count up 2 cycles
      clr_counter_sync();
      count(3, 1);          // count up 3 cycles
      clr_counter_async();
      count(5, 1);          // count up 3 cycles
      $stop;                // stop simulation
   end

   //=====================================
   // task definitions
   //=====================================
   // assert reset between clock edges
   task clr_counter_async();
   begin
      @(negedge clk);    // wait for failing edge
      reset = 1'b1;
      #(T/4);            // assert 4/T
      reset = 1'b0;
   end
   endtask

   task initialize();
   // system initialization
   begin
      en = 0;
      up = 0;
      load = 0;
      syn_clr = 0;
      d = 3'b000;
      clr_counter_async();
   end
   endtask

   // asset syn_clr one clock cycle
   task clr_counter_sync();
   begin
      @(negedge clk);  // wait for failing edge
      syn_clr = 1'b1;  // assert clear
      @(negedge clk);
      syn_clr = 1'b0;
   end
   endtask

   // load register
   task load_data(input wire [N-1:0] data_in);
   begin
      @(negedge clk);  // wait for failing edge
      load = 1'b1;
      d = data_in;
      @(negedge clk);
      load = 1'b0;
   end
   endtask
```

```
   // count up or down for C cycles
   task count(input integer C, input integer UP_DOWN);
   begin
85    @(negedge clk);  // wait for failing edge
      en = 1'b1;
      if (UP_DOWN==1)  // count up if up_down is 1
         up = 1'b1;
      repeat(C) @(negedge clk);
90    en = 1'b0;
      up = 1'b0;
   end
   endtask

95 endmodule
```

Monitor module The monitor module monitors and records the activities of the counter and verifies its operation. The complete code is shown in Listing 7.18.

Listing 7.18 Monitor

```
module bin_monitor
   #(parameter N=3)
   (
    input wire clk, reset,
5   input wire syn_clr, load, en, up,
    input wire [N-1:0] d,
    input wire max_tick, min_tick,
    input wire [N-1:0] q
   );
10
   reg [N-1:0] q_old, d_old, gold;
   reg syn_clr_old, en_old, load_old, up_old;
   reg [39:0] err_msg; // 5-letter message

15 initial   // head
      $display("time  syn_clr/load/en/up  q\n");

   always @(posedge clk)
   begin
20    // _old: the value sampled at the previous clock edge
      syn_clr_old <= syn_clr;
      en_old <= en;
      load_old <= load;
      up_old <= up;
25    q_old <= q;
      d_old <= d;

      // calculate the desired "gold" value
      if (syn_clr_old)
30       gold = 0;
      else if (load_old)
         gold = d_old;
```

```
      else if (en_old & up_old)
         gold = q_old + 1;
      else if (en_old & ~up_old)
         gold = q_old - 1;
      else
         gold = q_old;

      // error message
      if (q==gold)
         err_msg = "      ";  // result passes
      else
         err_msg = "ERROR";  // result fails
      //
      $display("%5d,  %b%b%b%b  %d  %s",
               $time, syn_clr, load, en, up, q, err_msg);
   end

endmodule
```

Since the counter is a synchronous sequential circuit, the monitor module focuses on the activities at the rising edge of the clock signal. The key is to check the correctness of the counter operation. Since the circuit under test is a simple counter, we can record the sampled input values and counter state from the previous sampling edge and determine the new counter state. For example, if the previous sampled value of `syn_clr` is 1, the counter is cleared and becomes 0 in the next rising edge of the clock.

The main part of the code is an always block, which is activated at the rising edge of the clock. There are three segments. The first segment uses nonblocking statements to infer registers, which are designated with the `_old` suffix and store the values sampled from the previous sampling edge. The second segment uses these values to calculate the expected counter output, `gold`. The last segment compares the expected counter output with the actual output and displays the values of the sampled input signals and the counter output. If a mismatch occurs, an ERROR message will be generated. Note that in Verilog a character is treated as an 8-bit number and thus the five-character message, `err_msg`, is declared as `reg [39:0]`.

Top-level module The code of the top-level testbench module is shown in Listing 7.19, which follows the block diagram in Figure 7.3.

Listing 7.19 Top-level module of testbench

```
`timescale 1 ns/10 ps

module bin_counter_tb3();

   // declaration
   localparam T=20; // clock period
   wire clk, reset;
   wire syn_clr, load, en, up;
   wire [2:0] d;
   wire max_tick, min_tick;
   wire [2:0] q;

   // uut instantiation
```

```
    univ_bin_counter #(.N(3)) uut
15    (.clk(clk), .reset(reset), .syn_clr(syn_clr),
       .load(load), .en(en), .up(up), .d(d),
       .max_tick(max_tick), .min_tick(min_tick), .q(q));

    // test vector generator
20  bin_gen #(.N(3),.T(20)) gen_unit
      (.clk(clk), .reset(reset), .syn_clr(syn_clr),
       .load(load), .en(en), .up(up), .d(d));

    // bin_monitor instantiation
25  bin_monitor #(.N(3)) mon_unit
      (.clk(clk), .reset(reset), .syn_clr(syn_clr),
       .load(load), .en(en), .up(up), .d(d),
       .max_tick(max_tick), .min_tick(min_tick), .q(q));

30 endmodule
```

In addition to the waveform, the testbench also generates textual output on the console panel:

```
time  syn_clr/load/en/up  q

   0  0000  x  ERROR
  20  0000  0  ERROR
  40  0011  0
  60  0011  1
  80  0011  2
 100  0011  3
 120  0011  4
 140  0011  5
 160  0011  6
 180  0011  7
 200  0011  0
 220  0011  1
 240  0011  2
 260  0011  3
 280  0000  4
 300  0010  4
 320  0010  3
 340  0010  2
 360  0010  1
 380  0010  0
 400  0010  7
 420  0000  6
 440  0100  6
 460  0000  3
 480  0011  3
 500  0011  4
 520  0000  5
 540  1000  5
```

```
560  0000  0
580  0011  0
600  0011  1
620  0011  2
640  0000  3
660  0000  0  ERROR
680  0011  0
700  0011  1
720  0011  2
740  0011  3
760  0011  4
```

There are three ERROR messages. The messages at times 0 and 20 occur during system initialization and are not real errors. The message at time 660 is due to the `clr_counter_async()` operation, which generates a short asynchronous pulse between the sampling edges of 640 and 660. Since the testbench monitors only synchronous activities, it misses the asynchronous reset and reports it as an error.

7.6 BIBLIOGRAPHIC NOTES

Verilog HDL, 2nd edition, by S. Palnitkar and *Starter's Guide to Verilog 2001* by M. D. Ciletti covers Verilog's syntax and constructs. *IEEE Standard Verilog Hardware Description Language, IEEE Std 1364-2001,* gives the rules regarding adjustment of an expression with mixed signed and unsigned data types. *Writing Testbenches: Functional Verification of HDL Models, 2nd edition,* by J. Bergeron, provides detailed discussion of testbench development. The article "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!" by C. E. Cummings gives guidelines for proper use of blocking and nonblocking assignments.

7.7 SUGGESTED EXPERIMENTS

7.7.1 Shift register with blocking and nonblocking assignments

The codes shown in Listing 7.20 are three attempts to describe a shift register. Derive the inferred circuits by the three attempts and determine whether they infer a shift register.

Listing 7.20 Code for Experiment 7.7.1

```
module exp1
   (
    input wire clk,
    input wire x0, y0, z0,
    output reg x3, y3, z3
   );

   reg x1, x2, y1, y2, z1, z2;
   // attempt 1
   always @(posedge clk)
   begin
      x1 <= x0;
```

```
      x2 <= x1;
      x3 <= x2;
15  end

    // attempt 2
    always @(posedge clk)
    begin
20    y1 = y0;
      y2 = y1;
      y3 = y2;
    end

25  // attempt 3
    always @(posedge clk)
    begin
      z1 = z0;
      z3 = z2;
30    z2 = z1;
    end

endmodule
```

7.7.2 Alternative coding style for BCD counter

Rewrite the BCD counter in Listing 4.18 using the coding style discussed in Section 7.2. Resynthesize the circuit and verify its operation.

7.7.3 Alternative coding style for FIFO buffer

Rewrite the FIFO buffer in Listing 4.20 using the coding style discussed in Section 7.2. Resynthesize the circuit and verify its operation.

7.7.4 Alternative coding style for Fibonacci circuit

Repeat the Fibonacci circuit discussed in Section 6.3.1 using the coding style discussed in Section 7.2.

7.7.5 Dual-mode comparator

A dual-mode comparator takes the two 8-bit data inputs, `a` and `b`, as unsigned or signed integers. A control signal, `mode`, indicates the desired mode. The circuit has one output, `agtb`, which is asserted when the interpreted value of `a` is greater than the interpreted value of `b`.

1. Assume that the **signed** data type is allowed. Design the circuit and derive the code.
2. Synthesize the circuit and verify its operation.
3. Assume that the **signed** data type is not allowed in the code. Repeat steps 1 and 2.

7.7.6 Enhanced binary counter monitor

The `monitor` module in Section 7.5.10 is intended to monitor a synchronous system and only checks the activities at the rising edges of the clock signal. The asynchronous reset operation is reported as an error. Modify the monitor circuit to take the asynchronous operation into consideration. Recreate the testbench and perform simulation to verify its operation.

7.7.7 Testbench for FIFO buffer

Follow the example in Section 7.5.10 to design a compressive testbench to verify operation of the FIFO buffer discussed in Section 4.5.3. The test vector generator module should generate various combinations of write and read operations and introduce the full and empty conditions. The monitor module should continuously watch data written into and retrieved from the buffer and check the correctness of the operations.

PART II

I/O MODULES

CHAPTER 8

UART

8.1 INTRODUCTION

A *universal asynchronous receiver and transmitter* (UART) is a circuit that sends parallel data through a serial line. UARTs are frequently used in conjunction with the EIA (Electronic Industries Alliance) RS-232 standard, which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment. Because the voltage level defined in RS-232 is different from that of FPGA I/O, a voltage converter chip is needed between a serial port and an FPGA's I/O pins.

The S3 board has an RS-232 port with a standard nine-pin connector. The board contains the necessary voltage converter chip and configures the various RS-232's control signals to automatically generate acknowledgment for the PC's serial port. A standard straight-through serial cable can be used to connect the S3 board and PC's serial port. The S3 board basically handles the RS-232 standard and we only need to concentrate on design of the UART circuit.

A UART includes a transmitter and a receiver. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and then reassembles the data. The serial line is 1 when it is idle. The transmission starts with a *start bit*, which is 0, followed by *data bits* and an optional *parity bit*, and ends with *stop bits*, which are 1. The number of data bits can be 6, 7, or 8. The optional parity bit is used for error detection. For odd parity, it is set to 0 when the data bits have an odd number of 1's. For even parity, it is set to 0 when the data bits have an even number of 1's. The number of stop bits can be 1, 1.5,

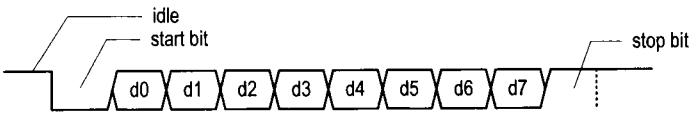


Figure 8.1 Transmission of a byte.

or 2. Transmission with 8 data bits, no parity, and 1 stop bit is shown in Figure 8.1. Note that the LSB of the data word is transmitted first.

No clock information is conveyed through the serial line. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate (i.e., number of bits per second), the number of data bits and stop bits, and use of the parity bit. The commonly used baud rates are 2400, 4800, 9600, and 19,200 bauds.

We illustrate the design of the receiving and transmitting subsystems in the following sections. The design is customized for a UART with a 19,200 baud rate, 8 data bits, 1 stop bit, and no parity bit.

8.2 UART RECEIVING SUBSYSTEM

Since no clock information is conveyed from the transmitted signal, the receiver can retrieve the data bits only by using the predetermined parameters. We use an *oversampling scheme* to estimate the middle points of transmitted bits and then retrieve them at these points accordingly.

8.2.1 Oversampling procedure

The most commonly used sampling rate is 16 times the baud rate, which means that each serial bit is sampled 16 times. Assume that the communication uses N data bits and M stop bits. The oversampling scheme works as follows:

1. Wait until the incoming signal becomes 0, the beginning of the start bit, and then start the sampling tick counter.
2. When the counter reaches 7, the incoming signal reaches the middle point of the start bit. Clear the counter to 0 and restart.
3. When the counter reaches 15, the incoming signal progresses for one bit and reaches the middle of the first data bit. Retrieve its value, shift it into a register, and restart the counter.
4. Repeat step 3 $N-1$ more times to retrieve the remaining data bits.
5. If the optional parity bit is used, repeat step 3 one time to obtain the parity bit.
6. Repeat step 3 M more times to obtain the stop bits.

The oversampling scheme basically performs the function of a clock signal. Instead of using the rising edge to indicate when the input signal is valid, it utilizes sampling ticks to estimate the middle point of each bit. While the receiver has no information about the exact onset time of the start bit, the estimation can be off by at most $\frac{1}{16}$. The subsequent data bit retrievals are off by at most $\frac{1}{16}$ from the middle point as well. Because of the oversampling, the baud rate can be only a small fraction of the system clock rate, and thus this scheme is not appropriate for a high data rate.

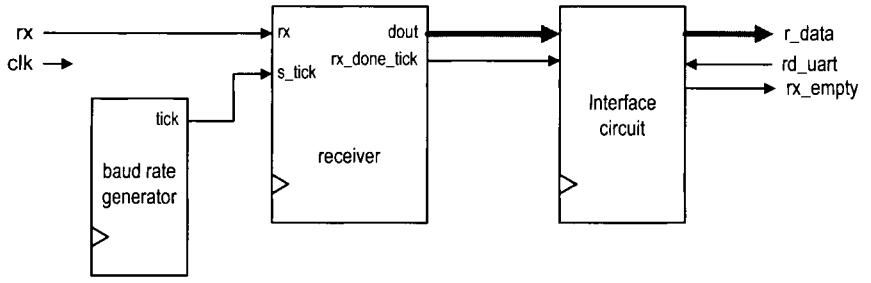


Figure 8.2 Conceptual block diagram of a UART receiving subsystem.

The conceptual block diagram of a UART receiving subsystem is shown in Figure 8.2. It consists of three major components:

- *UART receiver*: the circuit to obtain the data word via oversampling
- *Baud rate generator*: the circuit to generate the sampling ticks
- *Interface circuit*: the circuit that provides a buffer and status between the UART receiver and the system that uses the UART

8.2.2 Baud rate generator

The baud rate generator generates a sampling signal whose frequency is exactly 16 times the UART's designated baud rate. To avoid creating a new clock domain and violating the synchronous design principle, the sampling signal should function as enable ticks rather than the clock signal to the UART receiver, as discussed in Section 4.3.2.

For the 19,200 baud rate, the sampling rate has to be 307,200 (i.e., 19,200*16) ticks per second. Since the system clock rate is 50 MHz, the baud rate generator needs a mod-163 (i.e., $\frac{50*10^6}{307200}$) counter, in which a one-clock-cycle tick is asserted once every 163 clock cycles. The parameterized mod-m counter discussed in Section 4.3.2 can be used for this purpose by setting the `M` parameter to 163.

8.2.3 UART receiver

With an understanding of the oversampling procedure, we can derive the ASMD chart accordingly, as shown in Figure 8.3. To accommodate future modification, two constants are used in the description. The `D_BIT` constant indicates the number of data bits, and the `SB_TICK` constant indicates the number of ticks needed for the stop bits, which is 16, 24, and 32 for 1, 1.5, and 2 stop bits, respectively. `D_BIT` and `SB_TICK` are assigned to 8 and 16 in this design.

The chart follows the steps discussed in Section 8.2.1 and includes three major states, `start`, `data`, and `stop`, which represent the processing of the start bit, data bits, and stop bit. The `s_tick` signal is the enable tick from the baud rate generator and there are 16 ticks in a bit interval. Note that the FSMD stays in the same state unless the `s_tick` signal is asserted. There are two counters, represented by the `s` and `n` registers. The `s` register keeps track of the number of sampling ticks and counts to 7 in the `start` state, to 15 in the `data` state, and to `SB_TICK` in the `stop` state. The `n` register keeps track of the number of data bits received in the `data` state. The retrieved bits are shifted into and reassembled in the `b`

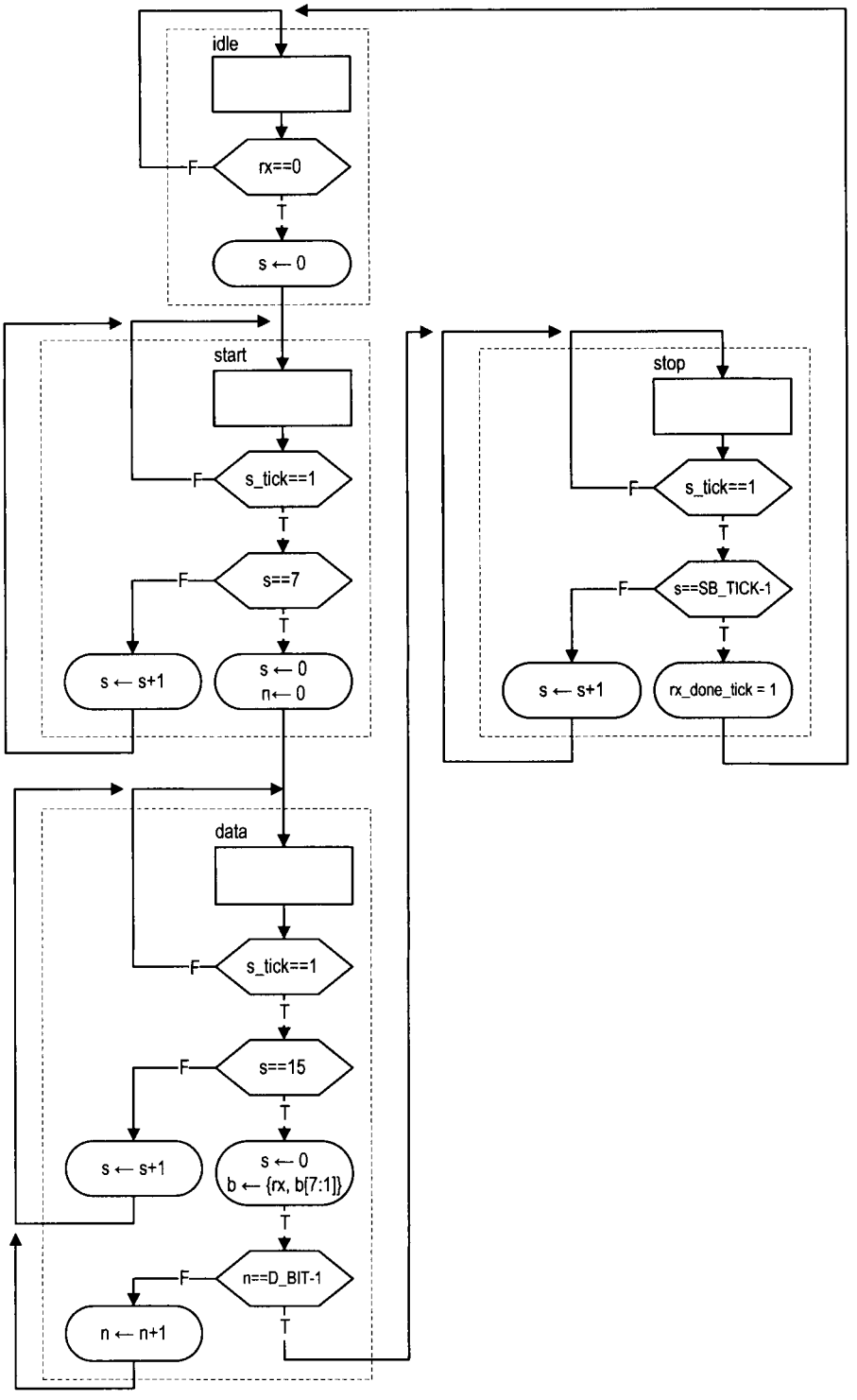


Figure 8.3 ASMD chart of a UART receiver.

register. A status signal, rx_done_tick, is included. It is asserted for one clock cycle after the receiving process is completed. The corresponding code is shown in Listing 8.1.

Listing 8.1 UART receiver

```
module uart_rx
   #(
     parameter DBIT = 8,      // # data bits
               SB_TICK = 16   // # ticks for stop bits
   )
   (
    input wire clk, reset,
    input wire rx, s_tick,
    output reg rx_done_tick,
    output wire [7:0] dout
   );

   // symbolic state declaration
   localparam [1:0]
      idle  = 2'b00,
      start = 2'b01,
      data  = 2'b10,
      stop  = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [3:0] s_reg, s_next;
   reg [2:0] n_reg, n_next;
   reg [7:0] b_reg, b_next;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            s_reg <= 0;
            n_reg <= 0;
            b_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            s_reg <= s_next;
            n_reg <= n_next;
            b_reg <= b_next;
         end

   // FSMD next-state logic
   always @*
   begin
      state_next = state_reg;
      rx_done_tick = 1'b0;
      s_next = s_reg;
```

```
      n_next = n_reg;
      b_next = b_reg;
      case (state_reg)
         idle:
            if (~rx)
               begin
                  state_next = start;
                  s_next = 0;
               end
         start:
            if (s_tick)
               if (s_reg==7)
                  begin
                     state_next = data;
                     s_next = 0;
                     n_next = 0;
                  end
               else
                  s_next = s_reg + 1;
         data:
            if (s_tick)
               if (s_reg==15)
                  begin
                     s_next = 0;
                     b_next = {rx, b_reg[7:1]};
                     if (n_reg==(DBIT-1))
                        state_next = stop ;
                      else
                        n_next = n_reg + 1;
                   end
               else
                  s_next = s_reg + 1;
         stop:
            if (s_tick)
               if (s_reg==(SB_TICK-1))
                  begin
                     state_next = idle;
                     rx_done_tick =1'b1;
                  end
               else
                  s_next = s_reg + 1;
      endcase
   end
   // output
   assign dout = b_reg;

endmodule
```

8.2.4 Interface circuit

In a large system, a UART is usually a peripheral circuit for serial data transfer. The main system checks its status periodically to retrieve and process the received word. The

receiver's interface circuit has two functions. First, it provides a mechanism to signal the availability of a *new* word and to prevent the received word from being retrieved multiple times. Second, it can provide buffer space between the receiver and the main system. There are three commonly used schemes:

- A flag FF
- A flag FF and a one-word buffer
- A FIFO buffer

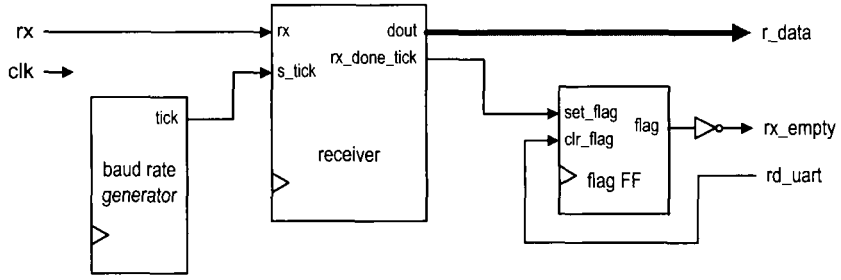
Note that the UART receiver asserts the `rx_ready_tick` signal one clock cycle after a data word is received.

The first scheme uses a *flag* FF to keep track of whether a new data word is available. The FF has two input signals. One is `set_flag`, which sets the flag FF to 1, and the other is `clr_flag`, which clears the flag FF to 0. The `rx_ready_tick` signal is connected to the `set_flag` signal and sets the flag when a new data word arrives. The main system checks the output of the flag FF to see whether a new data word is available. It asserts the `clr_flag` signal one clock cycle after retrieving the word. The top-level block diagram is shown in Figure 8.4(a). To be consistent with other schemes, the flag FF's output is inverted to generate the final `rx_empty` signal, which indicates that no new word is available. In this scheme, the main system retrieves the data word directly from the shift register of the UART receiver and does not provide any additional buffer space. If the remote system initiates a new transmission before the main system consumes the old data word (i.e., the flag FF is still asserted), the old word will be overwritten, an error known as *data overrun*.

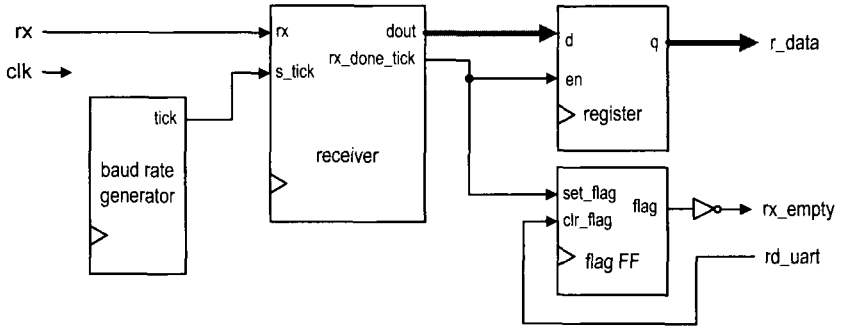
To provide some cushion, a one-word buffer can be added, as shown in Figure 8.4(b). When the `rx_ready_tick` signal is asserted, the received word is loaded to the buffer and the flag FF is set as well. The receiver can continue the operation without destroying the content of the last received word. Data overrun will not occur as long as the main system retrieves the word before a new word arrives. The code for this scheme is shown in Listing 8.2.

Listing 8.2 Interface with a flag FF and buffer

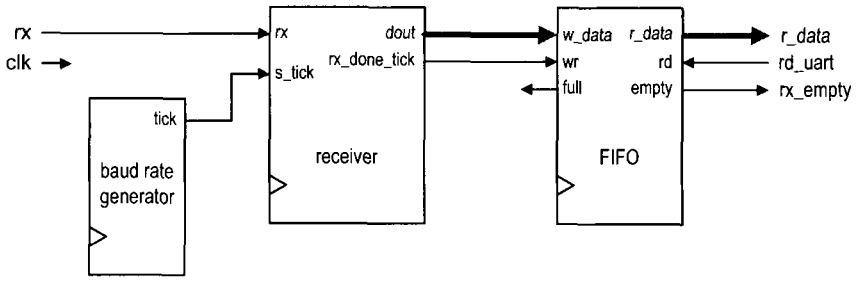
```
module flag_buf
   #(parameter W = 8) // # buffer bits
   (
    input wire clk, reset,
    input wire clr_flag, set_flag,
    input wire [W-1:0] din,
    output wire flag,
    output wire [W-1:0] dout
   );

   // signal declaration
   reg [W-1:0] buf_reg, buf_next;
   reg flag_reg, flag_next;


   // body
   // FF & register
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            buf_reg <= 0;
```



(a) Flag FF



(b) Flag FF and one-word buffer



(c) FIFO buffer

Figure 8.4 Interface circuit of a UART receiving subsystem.

```
            flag_reg <= 1'b0;
         end
      else
         begin
            buf_reg <= buf_next;
            flag_reg <= flag_next;
         end

   // next-state logic
   always @*
   begin
      buf_next = buf_reg;
      flag_next = flag_reg;
      if (set_flag)
         begin
            buf_next = din;
            flag_next = 1'b1;
         end
      else if (clr_flag)
         flag_next = 1'b0;
   end
   // output logic
   assign dout = buf_reg;
   assign flag = flag_reg;

endmodule
```

The third scheme uses a FIFO buffer discussed in Section 4.5.3. The FIFO buffer provides more buffering space and further reduces the chance of data overrun. We can adjust the desired number of words in FIFO to accommodate the processing need of the main system. The detailed block diagram is shown in Figure 8.4(c).

The `rx_ready_tick` signal is connected to the `wr` signal of the FIFO. When a new data word is received, the `wr` signal is asserted one clock cycle and the corresponding data is written to the FIFO. The main system obtains the data from FIFO's read port. After retrieving a word, it asserts the `rd` signal of the FIFO one clock cycle to remove the corresponding item. The `empty` signal of the FIFO can be used to indicate whether any received data word is available. A data-overrun error occurs when a new data word arrives and the FIFO is full.

8.3 UART TRANSMITTING SUBSYSTEM

The organization of a UART transmitting subsystem is similar to that of the receiving subsystem. It consists of a UART transmitter, baud rate generator, and interface circuit. The interface circuit is similar to that of the receiving subsystem except that the main system sets the flag FF or writes the FIFO buffer, and the UART transmitter clears the flag FF or reads the FIFO buffer.

The UART transmitter is essentially a shift register that shifts out data bits at a specific rate. The rate can be controlled by one-clock-cycle enable ticks generated by the baud rate generator. Because no oversampling is involved, the frequency of the ticks is 16 times slower than that of the UART receiver. Instead of introducing a new counter, the UART transmitter usually shares the baud rate generator of the UART receiver and uses an internal

counter to keep track of the number of enable ticks. A bit is shifted out every 16 enable ticks.

The ASMD chart of the UART transmitter is similar to that of the UART receiver. After assertion of the `tx_start` signal, the FSMD loads the data word and then gradually progresses through the `start`, `data`, and `stop` states to shift out the corresponding bits. It signals completion by asserting the `tx_done_tick` signal for one clock cycle. A 1-bit buffer, `tx_reg`, is used to filter out any potential glitch. The corresponding code is shown in Listing 8.3.

Listing 8.3 UART transmitter

```
module uart_tx
   #(
     parameter DBIT = 8,      // # data bits
               SB_TICK = 16   // # ticks for stop bits
   )
   (
    input wire clk, reset,
    input wire tx_start, s_tick,
    input wire [7:0] din,
    output reg tx_done_tick,
    output wire tx
   );

   // symbolic state declaration
   localparam [1:0]
      idle  = 2'b00,
      start = 2'b01,
      data  = 2'b10,
      stop  = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [3:0] s_reg, s_next;
   reg [2:0] n_reg, n_next;
   reg [7:0] b_reg, b_next;
   reg tx_reg, tx_next;

   // body
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            s_reg <= 0;
            n_reg <= 0;
            b_reg <= 0;
            tx_reg <= 1'b1;
         end
      else
         begin
            state_reg <= state_next;
            s_reg <= s_next;
```

```
            n_reg <= n_next;
            b_reg <= b_next;
            tx_reg <= tx_next;
         end

   // FSMD next-state logic & functional units
   always @*
   begin
      state_next = state_reg;
      tx_done_tick = 1'b0;
      s_next = s_reg;
      n_next = n_reg;
      b_next = b_reg;
      tx_next = tx_reg ;
      case (state_reg)
         idle:
            begin
               tx_next = 1'b1;
               if (tx_start)
                  begin
                     state_next = start;
                     s_next = 0;
                     b_next = din;
                  end
            end
         start:
            begin
               tx_next = 1'b0;
               if (s_tick)
                  if (s_reg==15)
                     begin
                        state_next = data;
                        s_next = 0;
                        n_next = 0;
                     end
                  else
                     s_next = s_reg + 1;
            end
         data:
            begin
               tx_next = b_reg[0];
               if (s_tick)
                  if (s_reg==15)
                     begin
                        s_next = 0;
                        b_next = b_reg >> 1;
                        if (n_reg==(DBIT-1))
                           state_next = stop ;
                        else
                           n_next = n_reg + 1;
                     end
                  else
                     s_next = s_reg + 1;
```

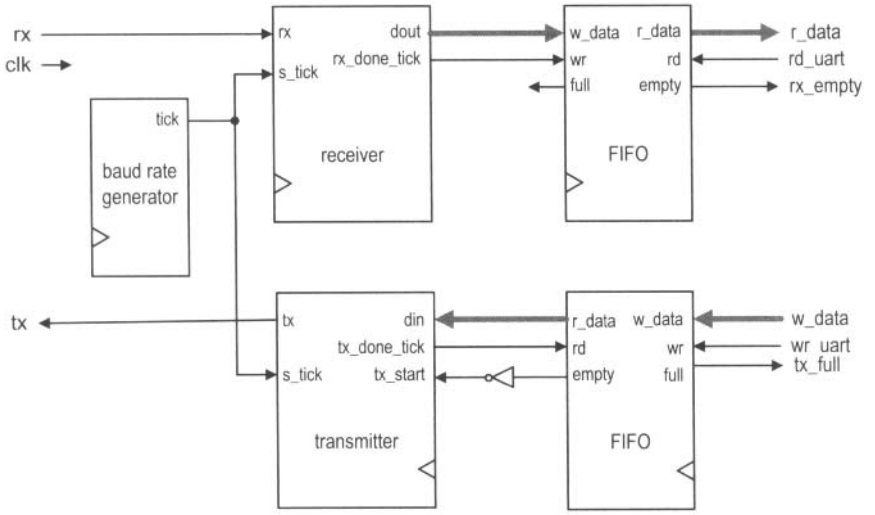


Figure 8.5 Block diagram of a complete UART.

```
            end
         stop:
            begin
               tx_next = 1'b1;
100            if (s_tick)
                  if (s_reg==(SB_TICK-1))
                     begin
                        state_next = idle;
                        tx_done_tick = 1'b1;
105                  end
                  else
                     s_next = s_reg + 1;
            end
      endcase
110 end
   // output
   assign tx = tx_reg;

endmodule
```

8.4 OVERALL UART SYSTEM

8.4.1 Complete UART core

By combining the receiving and transmitting subsystems, we can construct the complete UART core. The top-level diagram is shown in Figure 8.5. The block diagram can be described by component instantiation, and the corresponding code is shown in Listing 8.4.

Listing 8.4 UART top-level description

```
module uart
   #( // Default setting:
      // 19,200 baud, 8 data bits, 1 stop bit, 2^2 FIFO
      parameter DBIT = 8,      // # data bits
                SB_TICK = 16,  // # ticks for stop bits,
                               // 16/24/32 for 1/1.5/2 bits
                DVSR = 163,    // baud rate divisor
                               // DVSR = 50M/(16*baud rate)
                DVSR_BIT = 8,  // # bits of DVSR
                FIFO_W = 2     // # addr bits of FIFO
                               // # words in FIFO=2^FIFO_W
   )
   (
    input wire clk, reset,
    input wire rd_uart, wr_uart, rx,
    input wire [7:0] w_data,
    output wire tx_full, rx_empty, tx,
    output wire [7:0] r_data
   );

   // signal declaration
   wire tick, rx_done_tick, tx_done_tick;
   wire tx_empty, tx_fifo_not_empty;
   wire [7:0] tx_fifo_out, rx_data_out;

   //body
   mod_m_counter #(.M(DVSR), .N(DVSR_BIT)) baud_gen_unit
      (.clk(clk), .reset(reset), .q(), .max_tick(tick));

   uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_rx_unit
      (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
       .rx_done_tick(rx_done_tick), .dout(rx_data_out));

   fifo #(.B(DBIT), .W(FIFO_W)) fifo_rx_unit
      (.clk(clk), .reset(reset), .rd(rd_uart),
       .wr(rx_done_tick), .w_data(rx_data_out),
       .empty(rx_empty), .full(), .r_data(r_data));

   fifo #(.B(DBIT), .W(FIFO_W)) fifo_tx_unit
      (.clk(clk), .reset(reset), .rd(tx_done_tick),
       .wr(wr_uart), .w_data(w_data), .empty(tx_empty),
       .full(tx_full), .r_data(tx_fifo_out));

   uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_tx_unit
      (.clk(clk), .reset(reset), .tx_start(tx_fifo_not_empty),
       .s_tick(tick), .din(tx_fifo_out),
       .tx_done_tick(tx_done_tick), .tx(tx));

   assign tx_fifo_not_empty = ~tx_empty;

endmodule
```

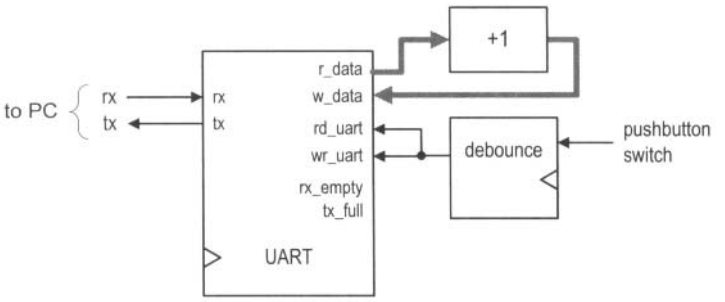


Figure 8.6 Block diagram of a UART verification circuit.

Xilinx specific

In the picoBlaze source file (discussed in Chapter 15), Xilinx supplies a customized UART module with similar functionality. Unlike our implementation, the module is described using low-level Xilinx primitives. It can be considered as a gate-level description that utilizes Xilinx-specific components. Since the designer has the expert knowledge of Xilinx devices and takes advantage of its architecture, its implementation is more efficient than the generic RT-level device-independent description of this chapter. It is instructive to compare the code complexity and the circuit size of the two descriptions.

8.4.2 UART verification configuration

Verification circuit We use a loop-back circuit and a PC to verify the UART's operation. The block diagram is shown in Figure 8.6. In the circuit, the serial port of the S3 board is connected to the serial port of a PC. When we send a character from the PC, the received data word is stored in the UART receiver's four-word FIFO buffer. When retrieved (via the `r_data` port), the data word is incremented by 1 and then sent back to the transmitter (via the `w_data` port). The debounced pushbutton switch produces a single one-clock-cycle tick when pressed and it is connected to the `rd_uart` and `wr_uart` signals. When the tick is generated, it removes one word from the receiver's FIFO and writes the incremented word to the transmitter's FIFO for transmission. For example, we can first type `HAL` in the PC and the three data words are stored in the FIFO buffer of the UART receiver. We can then push the button on the S3 board three times. The three successive characters, `IBM`, will be transmitted back and displayed. The UART's `r_data` port is also connected to the eight LEDs of the S3 board, and its `tx_full` and `rx_empty` signals are connected to the two horizontal bars of the rightmost digit of the seven-segment display. The code is shown in Listing 8.5.

Listing 8.5 UART verification circuit

```
module uart_test
   (
    input wire clk, reset,
    input wire rx,
    input wire [2:0] btn,
    output wire tx,
    output wire [3:0] an,
    output wire [7:0] sseg, led
   );
```

```
10
    // signal declaration
    wire tx_full, rx_empty, btn_tick;
    wire [7:0] rec_data, rec_data1;

15  // body
    // instantiate uart
    uart uart_unit
       (.clk(clk), .reset(reset), .rd_uart(btn_tick),
        .wr_uart(btn_tick), .rx(rx), .w_data(rec_data1),
20      .tx_full(tx_full), .rx_empty(rx_empty),
        .r_data(rec_data), .tx(tx));
    // instantiate debounce circuit
    debounce btn_db_unit
       (.clk(clk), .reset(reset), .sw(btn[0]),
25      .db_level(), .db_tick(btn_tick));
    // incremented data loops back
    assign rec_data1 = rec_data + 1;
    // LED display
    assign led = rec_data;
30  assign an = 4'b1110;
    assign sseg = {1'b1, ~tx_full, 2'b11, ~rx_empty, 3'b111};

endmodule
```

HyperTerminal of Windows On the PC side, Windows' HyperTerminal program can be used as a virtual terminal to interact with the S3 board. To be compatible with our customized UART, it has to be configured as 19,200 baud, 8 data bits, 1 stop bit, and no parity bit. The basic procedure is:

1. Select Start ≻ Programs ≻ Accessories ≻ Communications ≻ HyperTerminal. The HyperTerminal dialog appears.
2. Type a name for this connection, say `fpga_192`. Click OK. This connection can be saved and invoked later.
3. A Connect_to dialog appears. Press the Connecting Using field and select the desired serial port (e.g., COM1). Click OK.
4. The Port Setting dialog appears. Configure the port as follows:
   - Bits per second: 19200
   - Data bits: 8
   - Parity: None
   - Stop bits: 1
   - Flow control: None

   Click OK.
5. Select File ≻ Properties ≻ Setting. Click ASCII Setup and check the Echo typed characters locally box. Click OK twice. This will allow the typed characters to be shown on the screen.

The HyperTerminal program is set up now and ready to communicate with the S3 board. We can type a few keys and observe the LEDs of the S3 board. Note that the received words are stored in the FIFO buffer and only the first received data word is displayed. After we press the pushbutton, the first data word will be removed from the FIFO and the incremented word will be looped back to the PC's serial port and displayed in the

HyperTerminal window. The full and empty status of the respective FIFO buffers can be tested by consecutively receiving and transmitting more than four data words.

ASCII code In HyperTerminal, characters are sent in ASCII code, which is 7 bits and consists of 128 code words, including regular alphabets, digits, punctuation symbols, and nonprintable control characters. The characters and their code words (in hexadecimal format) are shown in Table 8.1. The nonprintable characters are shown enclosed in parentheses, such as (del). Several nonprintable characters may introduce special action when received:

- (nul): null byte, which is the all-zero pattern
- (bel): generate a bell sound, if supported
- (bs): backspace
- (ht): horizontal tab
- (nl): new line
- (vt): vertical tab
- (np): new page
- (cr): carriage return
- (esc): escape
- (sp): space
- (del): delete, which is also the all-one pattern

Since we use the PC's serial port to communicate with the S3 board in many experiments and projects, the following observations help us to manipulate and process the ASCII code:

- When the first hex digit in a code word is 0_{16} or 1_{16}, the corresponding character is a control character.
- When the first hex digit in a code word is 2_{16} or 3_{16}, the corresponding character is a digit or punctuation.
- When the first hex digit in a code word is 4_{16} or 5_{16}, the corresponding character is generally an uppercase letter.
- When the first hex digit in a code word is 6_{16} or 7_{16}, the corresponding character is generally a lowercase letter.
- If the first hex digit in a code word is 3_{16}, the lower hex digit represents the corresponding decimal digit.
- The upper- and lowercase letters differ in a single bit and can be converted to each other by adding or subtracting 20_{16} or inverting the sixth bit.

Note that the ASCII code uses only 7 bits, but a data word is normally composed of 8 bits (i.e., a byte). The PC uses an extended set in which the MSB is 1 and the characters are special graphics symbols. This code, however, is not part of the ASCII standard.

8.5 CUSTOMIZING A UART

The UART discussed in previous sections is customized for a particular configuration. The design and code can easily be modified to accommodate other required features:

- *Baud rate.* The baud rate is controlled by the frequency of the sampling ticks of the baud rate generator. The frequency can be changed by revising the `M` parameter of the mod-m counter, which is represented as the `DVSR` constant in code.
- *Number of data bits.* The number of data bits can be changed by modifying the upper limit of the `n_reg` register, which is specified as the `DBIT` constant in code.
- *Parity bit.* A parity bit can be included by introducing a new state between the `data` and `stop` states in the ASMD chart in Figure 8.3.

Table 8.1 ASCII codes

Code	Char	Code	Char	Code	Char	Code	Char
00	(nul)	20	(sp)	40	@	60	‘
01	(soh)	21	!	41	A	61	a
02	(stx)	22	"	42	B	62	b
03	(etx)	23	#	43	C	63	c
04	(eot)	24	$	44	D	64	d
05	(enq)	25	%	45	E	65	e
06	(ack)	26	&	46	F	66	f
07	(bel)	27	’	47	G	67	g
08	(bs)	28	(	48	H	68	h
09	(ht)	29	)	49	I	69	i
0a	(nl)	2a	*	4a	J	6a	j
0b	(vt)	2b	+	4b	K	6b	k
0c	(np)	2c	,	4c	L	6c	l
0d	(cr)	2d	-	4d	M	6d	m
0e	(so)	2e	.	4e	N	6e	n
0f	(si)	2f	/	4f	O	6f	o
10	(dle)	30	0	50	P	70	p
11	(dc1)	31	1	51	Q	71	q
12	(dc2)	32	2	52	R	72	r
13	(dc3)	33	3	53	S	73	s
14	(dc4)	34	4	54	T	74	t
15	(nak)	35	5	55	U	75	u
16	(syn)	36	6	56	V	76	v
17	(etb)	37	7	57	W	77	w
18	(can)	38	8	58	X	78	x
19	(em)	39	9	59	Y	79	y
1a	(sub)	3a	:	5a	Z	7a	z
1b	(esc)	3b	;	5b	[	7b	{
1c	(fs)	3c	<	5c	\	7c	|
1d	(gs)	3d	=	5d	]	7d	}
1e	(rs)	3e	>	5e	^	7e	~
1f	(us)	3f	?	5f	_	7f	(del)

- *Number of stop bits.* The number of stop bits can be changed by modifying the upper limit of the `s_reg` register in the `stop` state of the ASMD chart. The `SB_TICK` constant is used for this purpose. It can be 16, 24, or 32, which is for 1, 1.5, or 2 stop bits, respectively.
- *Error checking.* Three types of errors can be detected in the UART receiving subsystem:
  - *Parity error.* If the parity bit is included, the receiver can check the correctness of the received parity bit.
  - *Frame error.* The receiver can check the received value in the `stop` state. If the value is not 1, a frame error occurs.
  - *Buffer overrun error.* This happens when the main system does not retrieve the received words in a timely manner. The UART receiver can check the value of the buffer's `flag_reg` signal or FIFO's `full` signal when the received word is ready to be stored (i.e., when the `rx_done_tick` signal is generated). Data overrun occurs if the `flag_reg` or `full` signal is still asserted.

8.6 BIBLIOGRAPHIC NOTES

Although the RS-232 standard is very old, it still provides a simple and reliable low-speed communication link between two devices. The *Wikipedia* Web site has a good overview article and several useful links on the subject (search with the keyword RS232). *Serial Port Complete* by Jan Axelson provides information on interfacing hardware devices to a PC's serial port.

8.7 SUGGESTED EXPERIMENTS

8.7.1 Full-featured UART

The alternative to the customized UART is to include all features in design and to dynamically configure the UART as needed. Consider a full-featured UART that uses additional input signals to specify the baud rate, type of parity bit, and the numbers of data bits and stop bits. The UART also includes an error signal. In addition to the I/O signals of the `uart_top` design in Listing 8.4, the following signals are required:

- `bd_rate`: 2-bit input signal specifying the baud rate, which can be 1200, 2400, 4800, or 9600 baud
- `d_num`: 1-bit input signal specifying the number of data bits, which can be 7 or 8
- `s_num`: 1-bit input signal specifying the number of stop bits, which can be 1 or 2
- `par`: 2-bit input signal specifying the desired parity scheme, which can be no parity, even parity, or odd parity
- `err`: 3-bit output signal in which the bits indicate the existence of the parity error, frame error, and data overrun error

Derive this circuit as follows:

1. Modify the ASMD chart in Figure 8.3 to accommodate the required extensions.
2. Revise the UART receiver code according to the ASMD chart.
3. Revise the UART transmitter code to accommodate the required extensions.

4. Revise the top-level UART code and the verification circuit. Use the onboard switches for the additional input signals and three LEDs for the error signals. Synthesize the verification circuit.
5. Create different configurations in HyperTerminal and verify operation of the UART circuit.

8.7.2 UART with an automatic baud rate detection circuit

The most commonly used number of data bits of a serial connection is eight, which corresponds to a byte. When a regular ASCII code is used in communication (as we type in the HyperTerminal window), only seven LSBs are used and the MSB is 0. If the UART is configured as 8 data bits, 1 stop bit, and no parity bit, the received word is in the form of 0_dddd_ddd0_1, in which d is a data bit and can be 0 or 1. Assume that there is sufficient time between the first word and subsequent transmissions. We can determine the baud rate by measuring the time interval between the first 0 and last 0. Based on this observation, we can derive a UART with an automatic baud rate detection circuit. In this scheme, the transmitting system first sends an ASCII code for rate detection and then resumes normal operation afterward. The receiving subsystem uses the first word to determine a baud rate and then uses this rate for the baud rate generator for the remaining transmission.

Assume that the UART configuration is 8 data bits, 1 stop bit, and no parity bit, and the baud rate can be 4800, 9600, or 19,200 baud. The revised UART receiver should have two operation modes. It is initially in the "detection mode" and waits for the first word. After the word is received and the baud rate is determined, the receiver enters "normal mode" and the UART operates in a regular fashion. Derive the UART as follows:

1. Draw the ASMD chart for the automatic baud rate detector circuit.
2. Derive the VHDL code for the ASMD chart. Use three LEDs on the S3 board to indicate the baud rate of the incoming signal.
3. Modify the UART to include three different baud rates: 4800, 9600, and 19,200. This can be achieved by using a register for the divisor of the baud rate generator and loading the value according to the desired baud rate.
4. Create a top-level FSMD to keep track of the mode and to control and coordinate operation of the baud rate detection circuit and the regular UART receiver. Use a pushbutton switch on the S3 board to force the UART into the detection mode.
5. Revise the top-level UART code and the verification circuit. Synthesize the verification circuit.
6. Create different configurations in HyperTerminal and verify operation of the UART.

8.7.3 UART with an automatic baud rate and parity detection circuit

In addition to the baud rate, we assume that the parity scheme also needs to be determined automatically, which can be no parity, even parity, or odd parity. Expand the previous automatic baud rate detection circuit to detect the parity configuration and repeat Experiment 8.7.2.

8.7.4 UART-controlled stopwatch

Consider the enhanced stopwatch in Experiment 4.7.6. Operation of the stopwatch is controlled by three switches on the S3 board. With the UART, we can use PC's HyperTerminal to send commands to and retrieve time from the stopwatch:

- When a `c` or `C` (for "clear") ASCII code is received, the stopwatch aborts current counting, is cleared to zero, and sets the counting direction to "up."
- When a `g` or `G` (for "go") ASCII code is received, the stopwatch starts to count.
- When a `p` or `P` (for "pause") ASCII code is received, counting pauses.
- When a `u` or `U` (for "up-down") ASCII code is received, the stopwatch reverses the direction of counting.
- When a `r` or `R` (for "receive") ASCII code is received, the stopwatch transmits the current time to the PC. The time should be displayed as "`DD.D`", where `D` is a decimal digit.
- All other codes will be ignored.

Design the new stopwatch, synthesize the circuit, connect it to a PC, and use HyperTerminal to verify its operation.

8.7.5 UART-controlled rotating LED banner

Consider the rotating LED banner circuit in Experiment 4.7.5. With the UART, we can use a PC's HyperTerminal to control its operation and dynamically modify the digits in the banner:

- When a `g` or `G` (for "go") ASCII code is received, the LED banner rotates.
- When a `p` or `P` (for "pause") ASCII code is received, the LED banner pauses.
- When a `d` or `D` (for "direction") ASCII code is received, the LED banner reverses the direction of rotation.
- When a decimal-digit (i.e., 0, 1, . . . , 9) ASCII code is received, the banner will be modified. The banner can be treated as a 10-word FIFO buffer. The new digit will be inserted at the beginning (i.e., the leftmost position) of the banner and the rightmost digit will be shifted out and discarded.
- All other codes will be ignored.

Design the new rotating LED banner, synthesize the circuit, connect it to a PC, and use HyperTerminal to verify its operation.

CHAPTER 9

PS2 KEYBOARD

9.1 INTRODUCTION

The PS2 port was introduced in IBM's Personal System/2 personnel computers. It is a widely supported interface for a keyboard and mouse to communicate with the host. The PS2 port contains two wires for communication purposes. One wire is for data, which is transmitted in a serial stream. The other wire is for the clock information, which specifies when the data is valid and can be retrieved. The information is transmitted as an 11-bit "packet" that contains a start bit, 8 data bits, an odd parity bit, and a stop bit. Whereas the basic format of the packet is identical for a keyboard and a mouse, the interpretation for the data bits is different. The FPGA prototyping board has a PS2 port and acts as a host. We discuss the keyboard interface in this chapter and cover the mouse interface in Chapter 10.

The communication of the PS2 port is bidirectional and the host can send a command to the keyboard or mouse to set certain parameters. For our purposes, the bidirectional communication is hardly required for the PS2 keyboard, and thus our discussion is limited to one direction, from the keyboard to the prototyping board. Bidirectional design is examined in the mouse interface in Chapter 10.

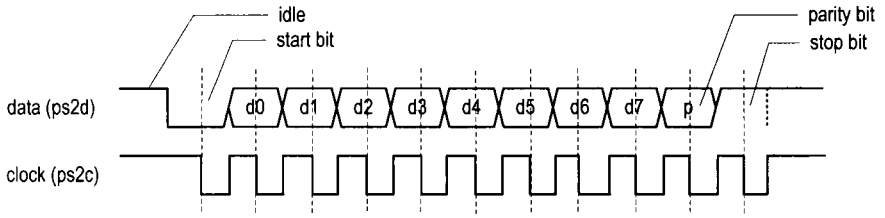


Figure 9.1 Timing diagram of a PS2 port.

9.2 PS2 RECEIVING SUBSYSTEM

9.2.1 Physical interface of a PS2 port

In addition to data and clock lines, the PS2 port includes connections for power (i.e., V_{cc}) and ground. The power is supplied by the host. In the original PS2 port, V_{cc} is 5 V and the outputs of the data and clock lines are open-collector. However, most current keyboards and mice can work well with 3.3 V. For an older keyboard and mouse, the 5-V supply can be obtained by switching the J2 jumper on the S3 board. The FPGA should still function properly since its I/O pins can tolerate a 5-V input.

9.2.2 Device-to-host communication protocol

A PS2 device and its host communicate via packets. The basic timing diagram of transmitting a packet from a PS2 device to a host is shown in Figure 9.1, in which the data and clock signals are labeled `ps2d` and `ps2c`, respectively.

The data is transmitted in a serial stream, and its format is similar to that of a UART. Transmission begins with a start bit, followed by 8 data bits and an odd parity bit, and ends with a stop bit. Unlike a UART, the clock information is carried in a separate clock signal, `ps2c`. The falling edge of the `ps2c` signal indicates that the corresponding bit in the `ps2d` line is valid and can be retrieved. The clock period of the `ps2c` signal is between 60 and 100 μs (i.e., 10 kHz to 16.7 kHz), and the `ps2d` signal is stable at least 5 μs before and after the falling edge of the `ps2c` signal.

9.2.3 Design and code

The design of the PS2 port receiving subsystem is somewhat similar to that of a UART receiver. Instead of using the oversampling scheme, the falling edge of the `ps2c` signal is used as the reference point to retrieve data. The subsystem includes a falling-edge detection circuit, which generates a one-clock-cycle tick at the falling edge of the `ps2c` signal, and the receiver, which shifts in and assembles the serial bits.

The edge detection circuit discussed in Section 5.3.1 can be used to detect the falling edge and generate an enable tick. However, because of the potential noise and slow transition, a simple filtering circuit is added to eliminate glitches. Its code is

```
always @(posedge clk, posedge reset)
. . .
   filter_reg <= filter_next;
. . .
```

```
// 1-bit shifter
assign filter_next = {ps2c, filter_reg[7:1]};
// "filter"
assign f_ps2c_next = (filter_reg==8'b11111111) ? 1'b1 :
                     (filter_reg==8'b00000000) ? 1'b0 :
                      f_ps2c_reg;
```

The circuit is composed of an 8-bit shift register and returns a 1 or 0 when eight consecutive 1's or 0's are received. Any glitches shorter than eight clock cycles will be ignored (i.e., filtered out). The filtered output signal is then fed to the regular falling-edge detection circuit.

The ASMD chart of the receiver is shown in Figure 9.2. The receiver is initially in the `idle` state. It includes an additional control signal, `rx_en`, which is used to enable or disable the receiving operation. The purpose of the signal is to coordinate the bidirectional operation. It can be set to 1 for the keyboard interface.

After the first falling-edge tick and the `rx_en` signal are asserted, the FSMD shifts in the start bit and moves to the `dps` state. Since the received data is in fixed format, we shift in the remaining 10 bits in a single state rather than using separate `data`, `parity`, and `stop` states. The FSMD then moves to the `load` state, in which one extra clock cycle is provided to complete the shifting of the stop bit, and the `psrx_done_tick` signal is asserted for one clock cycle. The HDL code consists of the filtering circuit and an FSMD, which follows the ASMD chart. It is shown in Listing 9.1.

Listing 9.1 PS2 port receiver

```
module ps2_rx
   (
    input wire clk, reset,
    input wire ps2d, ps2c, rx_en,
    output reg rx_done_tick,
    output wire [7:0] dout
   );

   // symbolic state declaration
   localparam [1:0]
      idle = 2'b00,
      dps  = 2'b01,
      load = 2'b10;

   // signal declaration
   reg [1:0] state_reg, state_next;
   reg [7:0] filter_reg;
   wire [7:0] filter_next;
   reg f_ps2c_reg;
   wire f_ps2c_next;
   reg [3:0] n_reg, n_next;
   reg [10:0] b_reg, b_next;
   wire fall_edge;

   // body
   //=============================================
   // filter and falling-edge tick generation for ps2c
   //=============================================
```

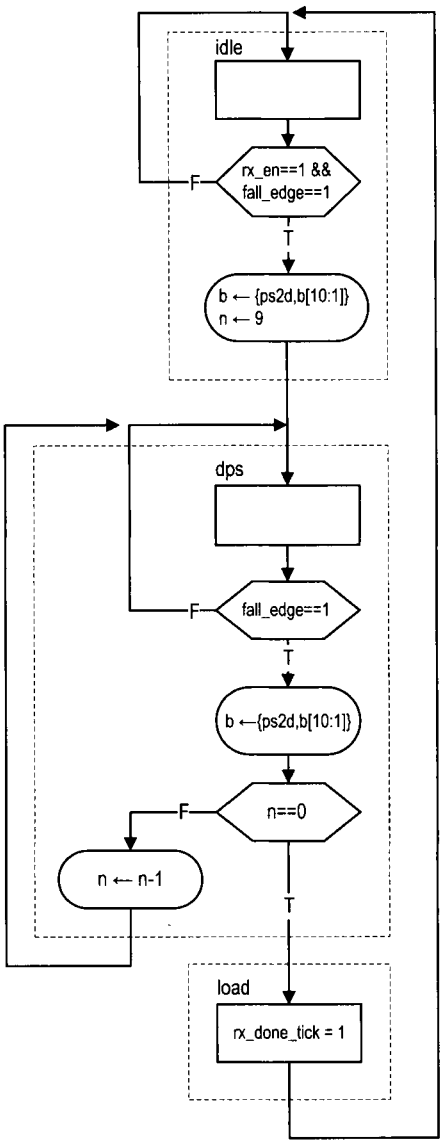


Figure 9.2 ASMD chart of PS2 port receiver.

```
   always @(posedge clk, posedge reset)
30 if (reset)
      begin
         filter_reg <= 0;
         f_ps2c_reg <= 0;
      end
35 else
      begin
         filter_reg <= filter_next;
         f_ps2c_reg <= f_ps2c_next;
      end
40
   assign filter_next = {ps2c, filter_reg[7:1]};
   assign f_ps2c_next = (filter_reg==8'b11111111) ? 1'b1 :
                        (filter_reg==8'b00000000) ? 1'b0 :
                         f_ps2c_reg;
45 assign fall_edge = f_ps2c_reg & ~f_ps2c_next;

   //=============================================
   // FSMD
   //=============================================
50 // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
55          n_reg <= 0;
            b_reg <= 0;
         end
      else
         begin
60          state_reg <= state_next;
            n_reg <= n_next;
            b_reg <= b_next;
         end
   // FSMD next-state logic
65 always @*
   begin
      state_next = state_reg;
      rx_done_tick = 1'b0;
      n_next = n_reg;
70    b_next = b_reg;
      case (state_reg)
         idle:
            if (fall_edge & rx_en)
               begin
75                // shift in start bit
                  b_next = {ps2d, b_reg[10:1]};
                  n_next = 4'b1001;
                  state_next = dps;
               end
80       dps: // 8 data + 1 parity + 1 stop
            if (fall_edge)
```

```
                begin
                   b_next = {ps2d, b_reg[10:1]};
                   if (n_reg==0)
                      state_next = load;
                   else
                      n_next = n_reg - 1;
                end
          load: // 1 extra clock to complete the last shift
             begin
                state_next = idle;
                rx_done_tick = 1'b1;
             end
       endcase
    end
    // output
    assign dout = b_reg[8:1]; // data bits

endmodule
```

There is no error detection circuit in the description. A more robust design should check the correctness of the start, parity, and stop bits and include a watchdog timer to prevent the keyboard from being locked in an incorrect state. This is left as an experiment at the end of the chapter.

9.3 PS2 KEYBOARD SCAN CODE

9.3.1 Overview of the scan code

A keyboard consists of a matrix of keys and an embedded microcontroller that monitors (i.e., scans) the activities of the keys and sends *scan code* accordingly. Three types of key activities are observed:

- When a key is pressed, the *make code* of the key is transmitted.
- When a key is held down continuously, a condition known as *typematic*, the make code is transmitted repeatedly at a specific rate. By default, a PS2 keyboard transmits the make code about every 100 ms after a key has been held down for 0.5 second.
- When a key is released, the *break code* of the key is transmitted.

The make code of the main part of a PS2 keyboard is shown in Figure 9.3. It is normally 1 byte wide and represented by two hexadecimal numbers. For example, the make code of the A key is 1C. This code can be conveyed by one packet when transmitted. The make codes of a handful of special-purpose keys, which are known as the *extended keys*, can have 2 to 4 bytes. A few of these keys are shown in Figure 9.3. For example, the make code of the upper arrow on the right is E0 75. Multiple packets are needed for the transmission. The break codes of the regular keys consist of F0 followed by the make code of the key. For example, the break code of the A key is F0 1C.

The PS2 keyboard transmits a sequence of codes according to the key activities. For example, when we press and release the A key, the keyboard first transmits its make code and then the break code:

```
1C F0 1C
```

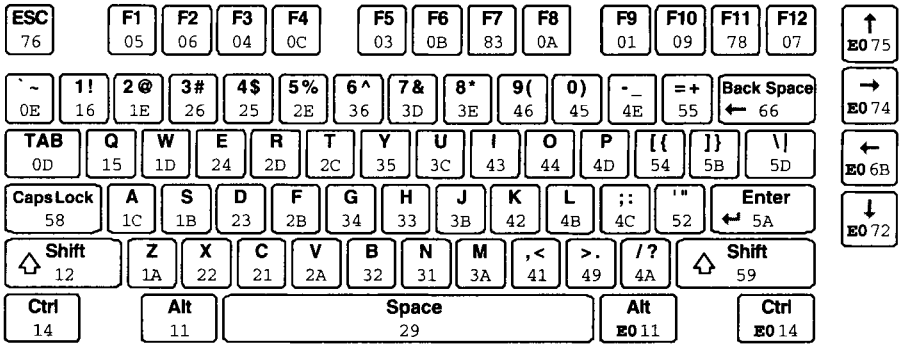


Figure 9.3 Scan code of the PS2 keyboard. (Courtesy of Xilinx, Inc. © Xilinx, Inc. 1994–2007. All rights reserved.)

If we hold the key down for awhile before releasing it, the make code will be transmitted multiple times:

```
1C 1C 1C ... 1C F0 1C
```

Multiple keys can be pressed at the same time. For example, we can first press the `shift` key (whose make code is 12) and then the `A` key, and release the `A` key and then release the `shift` key. The transmitted code sequence follows the make and break codes of the two keys:

```
12 1C F0 1C F0 12
```

The preceding sequence is how we normally obtain an uppercase `A`. Note that there is no special code to distinguish the lower- and uppercase keys. It is the responsibility of the host device to keep track of whether the shift key is pressed and to determine the case accordingly.

9.3.2 Scan code monitor circuit

The scan code monitor circuit monitors the arrival of the received packets and displays the scan codes on a PC's HyperTerminal window. The basic design approach is to first split the received scan code into two 4-bit parts and treat them as two hexadecimal digits, and then convert the two digits to ASCII code words and send the words to a PC via the UART. The received scan codes should be displayed similar to the previous example sequences. The program is shown in Listing 9.2.

Listing 9.2 PS2 keyboard scan code monitor circuit

```
module kb_monitor
   (
    input wire clk, reset,
    input wire ps2d, ps2c,
    output wire tx
   );

   // constant declaration
   localparam SP=8'h20; // space in ASCII
```

```
10
    // symbolic state declaration
    localparam [1:0]
       idle  = 2'b00,
       send1 = 2'b01,
15     send0 = 2'b10,
       sendb = 2'b11;

    // signal declaration
    reg [1:0] state_reg, state_next;
20  reg [7:0] w_data, ascii_code;
    wire [7:0] scan_data;
    reg wr_uart;
    wire scan_done_tick;
    wire [3:0] hex_in;
25
    // body
    //=================================================
    // instantiation
    //=================================================
30  // instantiate ps2 receiver
    ps2_rx ps2_rx_unit
       (.clk(clk), .reset(reset), .rx_en(1'b1),
        .ps2d(ps2d), .ps2c(ps2c),
        .rx_done_tick(scan_done_tick), .dout(scan_data));
35
    // instantiate UART
    uart uart_unit
       (.clk(clk), .reset(reset), .rd_uart(1'b0),
        .wr_uart(wr_uart), .rx(1'b1), .w_data(w_data),
40      .tx_full(), .rx_empty(), .r_data(), .tx(tx));

    //=================================================
    // FSM to send 3 ASCII characters
    //=================================================
45  // state registers
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= idle;
       else
50        state_reg <= state_next;

    // next-state logic
    always @*
    begin
55     wr_uart = 1'b0;
       w_data = SP;
       state_next = state_reg;
       case (state_reg)
          idle:
60           if (scan_done_tick) // a scan code received
                state_next = send1;
          send1: // send higher hex char
```

```
            begin
               w_data = ascii_code;
               wr_uart = 1'b1;
               state_next = send0;
            end
         send0: // send lower hex char
            begin
               w_data = ascii_code;
               wr_uart = 1'b1;
               state_next = sendb;
            end
         sendb:  // send blank char
            begin
               w_data = SP;
               wr_uart = 1'b1;
               state_next = idle;
            end
      endcase
   end

   //=================================================
   // scan code to ASCII display
   //=================================================
   // split the scan code into two 4-bit hex
   assign hex_in = (state_reg==send1)? scan_data[7:4] :
                                       scan_data[3:0];
   // hex digit to ASCII code
   always @*
    case (hex_in)
       4'h0: ascii_code = 8'h30;
       4'h1: ascii_code = 8'h31;
       4'h2: ascii_code = 8'h32;
       4'h3: ascii_code = 8'h33;
       4'h4: ascii_code = 8'h34;
       4'h5: ascii_code = 8'h35;
       4'h6: ascii_code = 8'h36;
       4'h7: ascii_code = 8'h37;
       4'h8: ascii_code = 8'h38;
       4'h9: ascii_code = 8'h39;
       4'ha: ascii_code = 8'h41;
       4'hb: ascii_code = 8'h42;
       4'hc: ascii_code = 8'h43;
       4'hd: ascii_code = 8'h44;
       4'he: ascii_code = 8'h45;
       default: ascii_code = 8'h46;
    endcase

endmodule
```

An FSM is used to control the overall operation. The UART operation is initiated when a new scan code is received (as indicated by the assertion of `scan_done_tick`). The FSM circulates through the `send1`, `send0`, and `sendb` states, in which the ASCII codes of the upper hexadecimal digit, lower hexadecimal digit, and blank space are written to the UART.

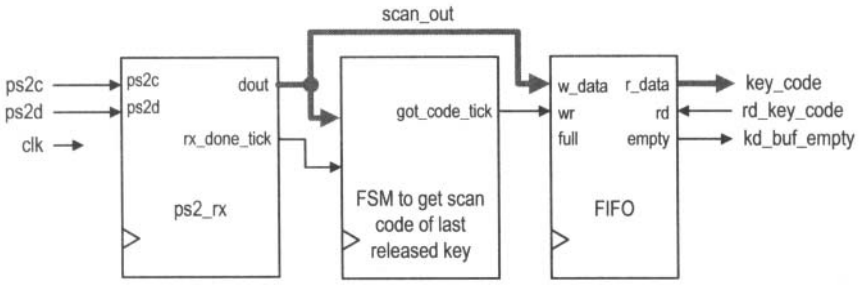


Figure 9.4 Block diagram of a last-released key circuit.

Recall that the UART has a FIFO of four words, and thus no overflow will occur. Note that the UART receiver is not used and the corresponding ports are mapped to constants or left blank.

9.4 PS2 KEYBOARD INTERFACE CIRCUIT

As discussed in Section 9.3.1, a sequence of packets is transmitted even for simple keyboard activities. It will be quite involved if we want to cover all possible combinations. In this section, we assume that only one regular key is pressed and released at a time and design a circuit that returns the make code of this key. This design provides a simple way to send a character or digit to the prototyping board and should be satisfactory for our purposes.

9.4.1 Basic design and HDL code

The keyboard circuit, as a UART, is a peripheral circuit of a large system and needs a mechanism to communicate with the main system. The flagging and buffering schemes discussed in Section 8.2.4 can be applied for the keyboard circuit as well. We use a four-word FIFO buffer as the interface in this design.

The top-level conceptual diagram is shown in Figure 9.4. It consists of the PS2 receiver, a FIFO buffer, and a control FSM. The basic idea is to use the FSM to keep track of the F0 packet of the break code. After it is received, the next packet should be the make code of this key and is written into the FIFO buffer. Note that this scheme cannot be applied to the extended keys since their make codes involve multiple packets. The corresponding HDL code is shown in Listing 9.3.

Listing 9.3 PS2 keyboard last-released key circuit

```
module kb_code
   #(parameter W_SIZE = 2)  // 2^W_SIZE words in FIFO
   (
    input wire clk, reset,
    input wire ps2d, ps2c, rd_key_code,
    output wire [7:0] key_code,
    output wire kb_buf_empty
   );

   // constant declaration
   localparam BRK = 8'hf0; // break code
```

```
   // symbolic state declaration
   localparam
      wait_brk = 1'b0,
      get_code = 1'b1;

   // signal declaration
   reg state_reg, state_next;
   wire [7:0] scan_out;
   reg got_code_tick;
   wire scan_done_tick;

   // body
   //=============================================================
   // instantiation
   //=============================================================
   // instantiate ps2 receiver
   ps2_rx ps2_rx_unit
      (.clk(clk), .reset(reset), .rx_en(1'b1),
       .ps2d(ps2d), .ps2c(ps2c),
       .rx_done_tick(scan_done_tick), .dout(scan_out));

   // instantiate fifo buffer
   fifo #(.B(8), .W(W_SIZE)) fifo_key_unit
     (.clk(clk), .reset(reset), .rd(rd_key_code),
      .wr(got_code_tick), .w_data(scan_out),
      .empty(kb_buf_empty), .full(),
      .r_data(key_code));

   //=============================================================
   // FSM to get the scan code after F0 received
   //=============================================================
   // state registers
   always @(posedge clk, posedge reset)
      if (reset)
         state_reg <= wait_brk;
      else
         state_reg <= state_next;

   // next-state logic
   always @*
   begin
      got_code_tick = 1'b0;
      state_next = state_reg;
      case (state_reg)
         wait_brk:  // wait for F0 of break code
            if (scan_done_tick==1'b1 && scan_out==BRK)
               state_next = get_code;
         get_code:  // get the following scan code
            if (scan_done_tick)
               begin
                  got_code_tick =1'b1;
                  state_next = wait_brk;
```

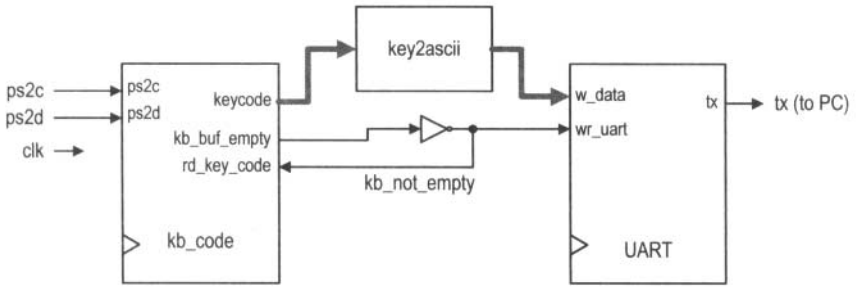


Figure 9.5 Block diagram of a keyboard verification circuit.

```
                end
        endcase
    end

endmodule
```

The main part of the code is the FSM, which screens for the break code and coordinates the operation of two other modules. It checks the received packets in the `wait_brk` state continuously. When the `F0` packet is detected, it moves to the `get_code` state and waits for the next packet, which is the make code of the key. The FSM then asserts the `code_done_tick` signal for one clock cycle and returns to the `wait_brk` state.

9.4.2 Verification circuit

We design a simple serial interface and decoding circuit to verify operation of the PS2 keyboard interface. The top-level block diagram is shown in Figure 9.5. The circuit converts a key's make code to the corresponding ASCII code and then sends the ASCII code to the UART. The corresponding character or digits can be displayed in the HyperTerminal window. The HDL code for the conversion circuit is shown in Listing 9.4.

Listing 9.4 Keyboard make code to ASCII code

```
module key2ascii
   (
    input wire [7:0] key_code,
    output reg [7:0] ascii_code
   );

always @*
   case(key_code)
       8'h45: ascii_code = 8'h30;    // 0
       8'h16: ascii_code = 8'h31;    // 1
       8'h1e: ascii_code = 8'h32;    // 2
       8'h26: ascii_code = 8'h33;    // 3
       8'h25: ascii_code = 8'h34;    // 4
       8'h2e: ascii_code = 8'h35;    // 5
       8'h36: ascii_code = 8'h36;    // 6
       8'h3d: ascii_code = 8'h37;    // 7
       8'h3e: ascii_code = 8'h38;    // 8
```

```
        8'h46: ascii_code = 8'h39;    // 9

        8'h1c: ascii_code = 8'h41;    // A
        8'h32: ascii_code = 8'h42;    // B
        8'h21: ascii_code = 8'h43;    // C
        8'h23: ascii_code = 8'h44;    // D
        8'h24: ascii_code = 8'h45;    // E
        8'h2b: ascii_code = 8'h46;    // F
        8'h34: ascii_code = 8'h47;    // G
        8'h33: ascii_code = 8'h48;    // H
        8'h43: ascii_code = 8'h49;    // I
        8'h3b: ascii_code = 8'h4a;    // J
        8'h42: ascii_code = 8'h4b;    // K
        8'h4b: ascii_code = 8'h4c;    // L
        8'h3a: ascii_code = 8'h4d;    // M
        8'h31: ascii_code = 8'h4e;    // N
        8'h44: ascii_code = 8'h4f;    // O
        8'h4d: ascii_code = 8'h50;    // P
        8'h15: ascii_code = 8'h51;    // Q
        8'h2d: ascii_code = 8'h52;    // R
        8'h1b: ascii_code = 8'h53;    // S
        8'h2c: ascii_code = 8'h54;    // T
        8'h3c: ascii_code = 8'h55;    // U
        8'h2a: ascii_code = 8'h56;    // V
        8'h1d: ascii_code = 8'h57;    // W
        8'h22: ascii_code = 8'h58;    // X
        8'h35: ascii_code = 8'h59;    // Y
        8'h1a: ascii_code = 8'h5a;    // Z

        8'h0e: ascii_code = 8'h60;    // `
        8'h4e: ascii_code = 8'h2d;    // -
        8'h55: ascii_code = 8'h3d;    // =
        8'h54: ascii_code = 8'h5b;    // [
        8'h5b: ascii_code = 8'h5d;    // ]
        8'h5d: ascii_code = 8'h5c;    // \
        8'h4c: ascii_code = 8'h3b;    // ;
        8'h52: ascii_code = 8'h27;    // '
        8'h41: ascii_code = 8'h2c;    // ,
        8'h49: ascii_code = 8'h2e;    // .
        8'h4a: ascii_code = 8'h2f;    // /

        8'h29: ascii_code = 8'h20;    // (space)
        8'h5a: ascii_code = 8'h0d;    // (enter, cr)
        8'h66: ascii_code = 8'h08;    // (backspace)
        default: ascii_code = 8'h2a; // *
    endcase

endmodule
```

The complete code for the verification circuit follows the block diagram and is shown in Listing 9.5.

Listing 9.5 Keyboard verification circuit

```
module kb_test
   (
    input wire clk, reset,
    input wire ps2d, ps2c,
    output wire tx
   );

   // signal declaration
   wire [7:0] key_code, ascii_code;
   wire kb_not_empty, kb_buf_empty;

   // body
   // instantiate keyboard scan code circuit
   kb_code kb_code_unit
      (.clk(clk), .reset(reset), .ps2d(ps2d), .ps2c(ps2c),
       .rd_key_code(kb_not_empty), .key_code(key_code),
       .kb_buf_empty(kb_buf_empty));

   // instantiate UART
   uart uart_unit
      (.clk(clk), .reset(reset), .rd_uart(1'b0),
       .wr_uart(kb_not_empty), .rx(1'b1), .w_data(ascii_code),
       .tx_full(), .rx_empty(), .r_data(), .tx(tx));

   // instantiate key-to-ascii code conversion circuit
   key2ascii k2a_unit
      (.key_code(key_code), .ascii_code(ascii_code));

   assign kb_not_empty = ~kb_buf_empty;

endmodule
```

9.5 BIBLIOGRAPHIC NOTES

Three articles, "PS/2 Mouse/Keyboard Protocol," "PS/2 Keyboard Interface," and "PS/2 Mouse Interface," by Adam Chapweske, provide detailed information on the PS2 keyboard and mouse interface. They can be found at the http://www.computer-engineering.org site. *Rapid Prototyping of Digital Systems: Quartus® II Edition* by James O. Hamblen et al. also contains a chapter on the PS2 port and the keyboard and mouse protocols.

9.6 SUGGESTED EXPERIMENTS

9.6.1 Alternative keyboard interface I

The interface circuit in Section 9.4 returns the make code of the last released key and thus ignores the typematic condition. An alternative approach is to consider the typematic condition. The keyboard interface circuit should return a key's make code repeatedly when it is held down and ignore the final break code. For simplicity, we assume that the extended

keys are not used. Design the new interface circuit, resynthesize the verification circuit, and verify operation of the new interface circuit.

9.6.2 Alternative keyboard interface II

We can expand the interface circuit to distinguish whether the shift key is pressed so that both lower- and uppercase characters can be entered. The expanded circuit can be modified as follows:

- The `keycode` output should be extended from 8 bits to 9 bits. The extra bit indicates whether the shift key is held down.
- The FSM should add a special branch to process the make and break codes of the shift key and set the value of the corresponding bit accordingly.
- The width of the FIFO buffer should be extended to 9 bits.

Design the expanded interface circuit, modify the `key2ascii` circuit to handle both lower- and uppercase characters, resynthesize the verification circuit, and verify operation of the expanded interface circuit.

9.6.3 PS2 receiving subsystem with watchdog timer

There is no error-handling capability in the PS2 receiving subsystem in Section 9.2. The potential noise and glitches in the `ps2c` signal may cause the FSMD to be stuck in an incorrect state. One way to deal with this problem is to add a watchdog timer. The timer is initiated every time the `fall_edge_tick` signal is asserted in the `get_bit` state. The `time_out` signal is asserted if no subsequently falling edge arrives in the next 20 μs, and the FSMD returns to the `idle` state. Design the modified receiving subsystem, derive a testbench, and use simulation to verify its operation.

9.6.4 Keyboard-controlled stopwatch

Consider the enhanced stopwatch in Experiment 4.7.6. Operation of the stopwatch is controlled by three switches on the prototyping board. We can use the keyboard to send commands to the stopwatch:

- When the `C` (for "clear") key is pressed, the stopwatch aborts the current counting, is cleared to zero, and sets the counting direction to "up."
- When the `G` (for "go") key is pressed, the stopwatch starts to count.
- When the `P` (for "pause") key is pressed, the counting pauses.
- When the `U` (for "up-down") key is pressed, the stopwatch reverses the direction of counting.
- All other keys will be ignored.

Design the new stopwatch, synthesize the circuit, and verify its operation.

9.6.5 Keyboard-controlled rotating LED banner

Consider the rotating LED banner circuit in Experiment 4.7.5. We can use a keyboard to control its operation and dynamically modify the digits in the banner:

- When the `G` (for "go") key is pressed, the LED banner rotates.
- When the `P` (for "pause") key is pressed, the LED banner pauses.

- When the D (for "direction") key is pressed, the LED banner reverses the direction of rotation.
- When a decimal digit (i.e., 0, 1, . . . , 9) key is pressed, the banner will be modified. The banner can be treated as a 10-word FIFO buffer. The new digit will be inserted at the beginning (i.e., the leftmost position) of the banner, and the rightmost digit will be shifted out and discarded.
- All other keys will be ignored.

Design the new rotating LED banner, synthesize the circuit, and verify its operation.

CHAPTER 10

PS2 MOUSE

10.1 INTRODUCTION

A computer mouse is designed mainly to detect two-dimensional motion on a surface. Its internal circuit measures the relative distance of movement and checks the status of the buttons. For a mouse with a PS2 interface, this information is packed in three packets and sent to the host through the PS2 port. In the *stream mode*, a PS2 mouse sends the packets continuously in a predesignated sampling rate.

Communication of the PS2 port is bidirectional and the host can send a command to the keyboard or mouse to set certain parameters. For our purposes, this functionality is hardly required for a keyboard, and thus the keyboard interface in Chapter 9 is limited to one direction, from the keyboard to the FPGA host. However, unlike the keyboard, a mouse is set to be in the nonstreaming mode after power-up and does not send any data. The host must first send a command to the mouse to initialize the mouse and enable the stream mode. Thus, bidirectional communication of the PS2 port is needed for the PS2 mouse interface, and we must design a transmitting subsystem (i.e., from FPGA board to mouse) for the PS2 interface.

In this chapter, we provide a short overview of the PS2 mouse protocol, design a bidirectional PS interface, and derive a simple mouse interface.

Table 10.1 Mouse data packet format

byte 1	y_v	x_v	y_8	x_8	1	m	r	l
byte 2	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
byte 3	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0

10.2 PS2 MOUSE PROTOCOL

10.2.1 Basic operation

A standard PS2 mouse reports the x-axis (right/left) and y-axis (up/down) movement and the status of the left button, middle button, and right button. The amount of each movement is recorded in a mouse's internal counter. When the data is transmitted to the host, the counter is cleared to zero and restarts the counting. The content of the counter represents a 9-bit signed integer in which a positive number indicates the right or up movement, and a negative number indicates the left or down movement.

The relationship between the physical distances is defined by the mouse's *resolution* parameter. The default value of resolution is four counts per millimeter. When a mouse moves continuously, the data is transmitted at a regular rate. The rate is defined by the mouse's *sampling rate* parameter. The default value of the sampling rate is 100 samples per second. If a mouse moves too fast, the amount of the movement during the sampling period may exceed the maximal range of the counter. The counter is set to the maximum magnitude in the appropriate direction. Two overflow bits are used to indicate the conditions.

The mouse reports the movement and button activities in 3 bytes, which are embedded in three PS2 packets. The detailed format of the 3-byte data is shown in Table 10.1. It contains the following information:

- $x_8, \ldots, x_0$: x-axis movement in 2's-complement format
- x_v: x-axis movement overflow
- $y_8, \ldots, y_0$: y-axis movement in 2's-complement format
- y_v: y-axis movement overflow
- l: left button status, which is 1 when the left button is pressed
- r: right button status, which is 1 when the right button is pressed
- m: optional middle button status, which is 1 when the middle button is pressed

During transmission, the byte 1 packet is sent first and the byte 3 packet is sent last.

10.2.2 Basic initialization procedure

The operation of a mouse is more complex than that of a keyboard. It has different operation modes. The most commonly used one is the *stream mode*, in which a mouse sends the movement data when it detects movement or button activity. If the movement is continuous, the data is generated at the designated sample rate.

During the operation, a host can send commands to a mouse to modify the default values of various parameters and set the operation mode, and a mouse may generate the status and send an acknowledgment. For our purposes, the default values are adequate, and the only task is to set the mouse to the stream mode.

The basic interaction sequence between a PS2 mouse and the FPGA host consists of the following:

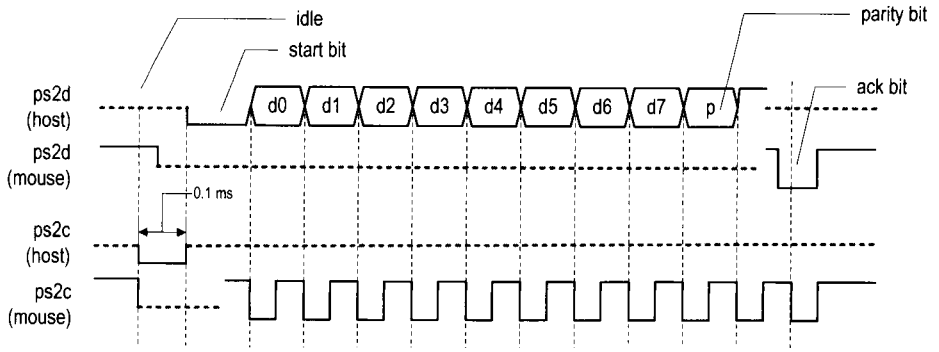


Figure 10.1 Host-to-device timing diagram of a PS2 port.

1. At power-on, a mouse performs a power-on test internally. The mouse sends 1-byte data AA, which indicates that the test is passed, and then 1-byte data 00, which is the id of a standard PS2 mouse.
2. The FPGA host sends the command, F4, to enable the stream mode. The mouse will respond with FE to acknowledge acceptance of the command.
3. The mouse now enters the stream mode and sends normal data packets.

If a mouse is plugged into the FPGA prototyping board in advance, it performs the power-on test when the power of the board is turned on and sends the AA 00 data immediately. The FPGA chip is not configured at this point and will not receive this data. Thus, we can usually ignore the power-on message in step 1. A minimal mouse interface circuit only needs to send the F4 command, check the FE acknowledge, and enter the normal operation mode to process the mouse's regular data packet.

We can force the mouse to return to the initial state by sending the reset command:

1. The FPGA host sends the command, FF, to reset the mouse. The mouse will respond with FE to acknowledge acceptance of the command.
2. The mouse performs a power-on test internally and then sends AA 00. The stream mode will be disabled during the process.

Newer mouses add more functionality, such as a scrolling wheel and additional buttons, and thus send more information. Additional bytes are appended to the original 3-byte data to accommodate these new features.

10.3 PS2 TRANSMITTING SUBSYSTEM

10.3.1 Host-to-PS2-device communication protocol

Host-to-PS2-device communication protocol involves bidirectional data exchange. The mouse's data and clock lines actually are *open-collector* circuits. For our design purposes, we treat them as tri-state lines. The basic timing diagram of transmitting a packet from a host to a PS2 device is shown in Figure 10.1, in which the data and clock signals are labeled ps2d and ps2c. For clarity, the diagram is split into two parts to show which activities are generated by the host (i.e., the FPGA chip) and which activities are generated by the device (i.e., mouse). The basic operation sequence is as follows:

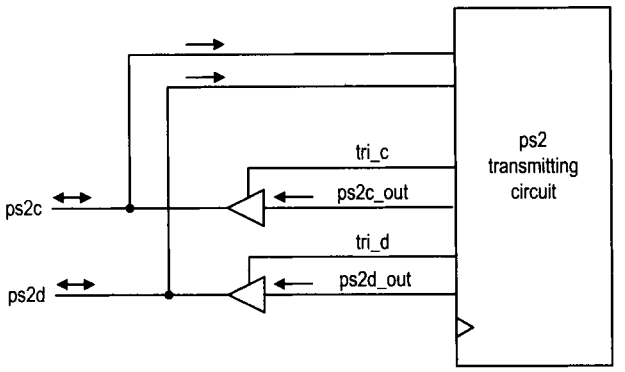


Figure 10.2 Tri-state buffers of the PS2 transmission subsystem.

1. The host forces the ps2c line to be 0 for at least 100 μs to inhibit any mouse activity. It can be considered that the host requests to send a packet.
2. The host forces the ps2d line to be 0 and disables the ps2c line (i.e., makes it high impedance). This step can be interpreted as the host sending a start bit.
3. The PS2 device now takes over the ps2c line and is responsible for future PS2 clock signal generation. After sensing the starting bit, the PS2 device generates a 1-to-0 transition.
4. Once detecting the transition, the host shifts out the least significant data bit over the ps2d line. It holds this value until the PS2 device generates a 1-to-0 transition in the ps2c line, which essentially acknowledges retrieval of the data bit.
5. Repeat step 4 for the remaining 7 data bits and 1 parity bit.
6. After sending the parity bit, the host disables the ps2d line (i.e., makes it high impedance). The PS2 device now takes over the ps2d line and acknowledges completion of the transmission by asserting the ps2d line to 0. If desired, the host can check this value at the last 1-to-0 transition in the ps2c line to verify that the packet has been transmitted successfully.

10.3.2 Design and code

Unlike the receiving subsystem, the ps2c and ps2d signals communicate in both directions. A tri-state buffer is needed for each signal. The tri-state interface is shown in Figure 10.2. The `tri_c` and `tri_d` signals are enable signals that control the tri-state buffers. When they are asserted, the corresponding `ps2c_out` and `ps2d_out` signals will be routed to the output ports.

To design the transmitting subsystem, we can follow the sequence of the preceding protocol to create an ASMD chart, as shown in Figure 10.3. The FSMD is initially in the `idle` state. To start the transmission, the host asserts the `wr_ps2` signal and places the data on the `din` bus. The FSMD loads `din`, along with the parity bit, `par`, to the `shift_reg` register, loads the "1...1" to `c_reg`, and moves to the `rts` (for "request to send") state. In this state, the `ps2c_out` is set to 0 and the corresponding `tri_c` is asserted to enable the corresponding tri-state buffer. The `c_reg` is used as a 13-bit counter to generate a 164-μs delay. The FSMD then moves to the `start` state, in which the PS2 clock line is disabled and the data line is set to 1. The PS2 device (i.e., mouse) now takes over and generates a

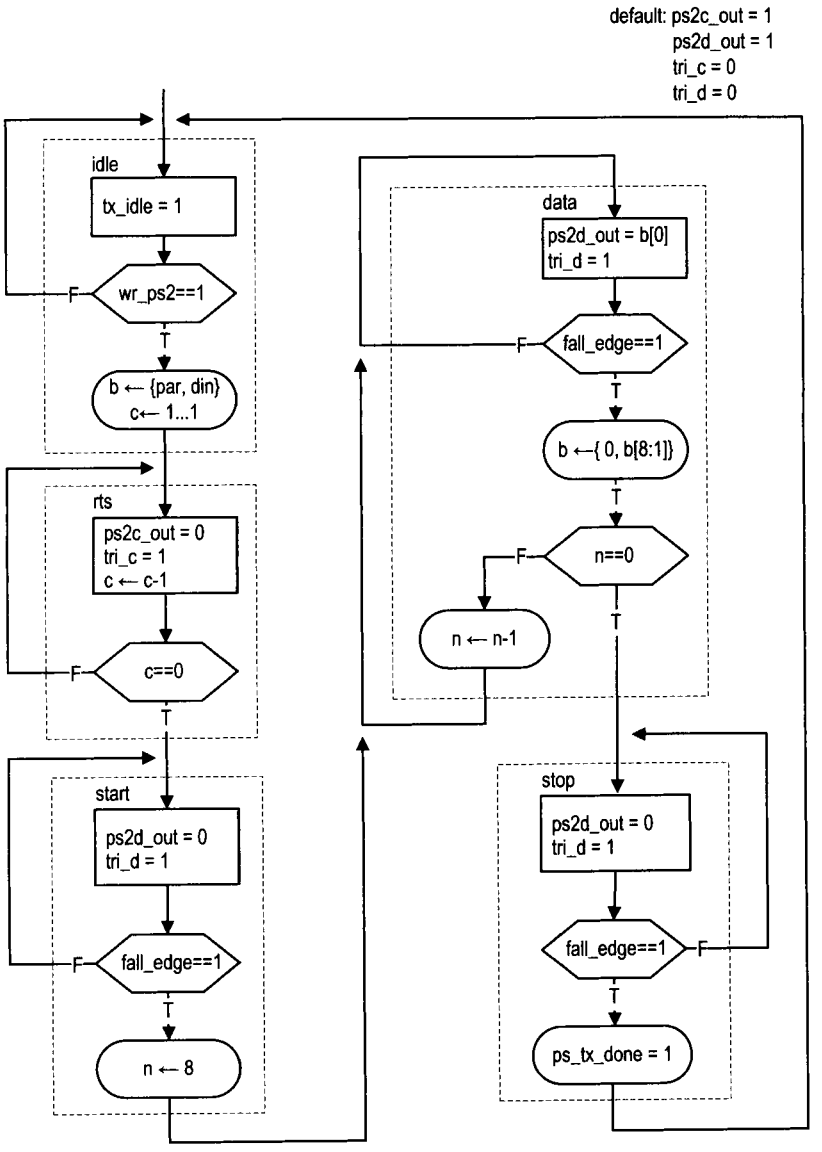


Figure 10.3 ASMD chart of the PS2 transmitting subsystem.

clock signal over the ps2c line. After detecting the falling edge of the ps2c signal through the fall_edge signal, the FSMD goes to the data state and shifts 8 data bits and 1 parity bit. The n register is used to keep track of the number of bits shifted. The FSMD then moves to the stop state, in which the data line is disabled. It returns to the idle state after sensing the last falling edge.

The FSMD also includes a tx_idle signal to indicate whether a transmission is in progress. This signal can be used to coordinate operation between the receiving and transmitting subsystems. The code follows the ASMD chart and is shown in Listing 10.1. A filtering circuit similar to that of Section 9.2 is used to generate the fall_edge signal.

Listing 10.1 PS2 port transmitter

```
module ps2_tx
   (
    input wire clk, reset,
    input wire wr_ps2,
    input wire [7:0] din,
    inout wire ps2d, ps2c,
    output reg tx_idle, tx_done_tick
   );

   // symbolic state declaration
   localparam [2:0]
      idle  = 3'b000,
      rts   = 3'b001,
      start = 3'b010,
      data  = 3'b011,
      stop  = 3'b100;

   // signal declaration
   reg [2:0] state_reg, state_next;
   reg [7:0] filter_reg;
   wire [7:0] filter_next;
   reg f_ps2c_reg;
   wire f_ps2c_next;
   reg [3:0] n_reg, n_next;
   reg [8:0] b_reg, b_next;
   reg [12:0] c_reg, c_next;
   wire par, fall_edge;
   reg ps2c_out, ps2d_out;
   reg tri_c, tri_d;

   // body
   //=============================================
   // filter and falling-edge tick generation for ps2c
   //=============================================
   always @(posedge clk, posedge reset)
   if (reset)
      begin
         filter_reg <= 0;
         f_ps2c_reg <= 0;
      end
   else
```

```
      begin
         filter_reg <= filter_next;
         f_ps2c_reg <= f_ps2c_next;
      end

   assign filter_next = {ps2c, filter_reg[7:1]};
   assign f_ps2c_next = (filter_reg==8'b11111111) ? 1'b1 :
                        (filter_reg==8'b00000000) ? 1'b0 :
                         f_ps2c_reg;
   assign fall_edge = f_ps2c_reg & ~f_ps2c_next;

   //=================================================
   // FSMD
   //=================================================
   // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= idle;
            c_reg <= 0;
            n_reg <= 0;
            b_reg <= 0;
         end
      else
         begin
            state_reg <= state_next;
            c_reg <= c_next;
            n_reg <= n_next;
            b_reg <= b_next;
         end

   // odd parity bit
   assign par = ~(^din);

   // FSMD next-state logic
   always @*
   begin
      state_next = state_reg;
      c_next = c_reg;
      n_next = n_reg;
      b_next = b_reg;
      tx_done_tick = 1'b0;
      ps2c_out = 1'b1;
      ps2d_out = 1'b1;
      tri_c = 1'b0;
      tri_d = 1'b0;
      tx_idle = 1'b0;
      case (state_reg)
         idle:
            begin
               tx_idle = 1'b1;
               if (wr_ps2)
                  begin
```

```
                    b_next = {par, din};
                    c_next = 13'h1fff; // 2^13-1
                    state_next = rts;
                end
            end
        rts:   // request to send
            begin
                ps2c_out = 1'b0;
                tri_c = 1'b1;
                c_next = c_reg - 1;
                if (c_reg==0)
                    state_next = start;
            end
        start:  // assert start bit
            begin
                ps2d_out = 1'b0;
                tri_d = 1'b1;
                if (fall_edge)
                    begin
                        n_next = 4'h8;
                        state_next = data;
                    end
            end
        data:   //  8 data + 1 parity
            begin
                ps2d_out = b_reg[0];
                tri_d = 1'b1;
                if (fall_edge)
                    begin
                        b_next = {1'b0, b_reg[8:1]};
                        if (n_reg == 0)
                            state_next = stop;
                        else
                            n_next = n_reg - 1;
                    end
            end
        stop:   // assume floating high for ps2d
            if (fall_edge)
                begin
                    state_next = idle;
                    tx_done_tick = 1'b1;
                end
    endcase
end

// tri-state buffers
assign ps2c = (tri_c) ? ps2c_out : 1'bz;
assign ps2d = (tri_d) ? ps2d_out : 1'bz;

endmodule
```

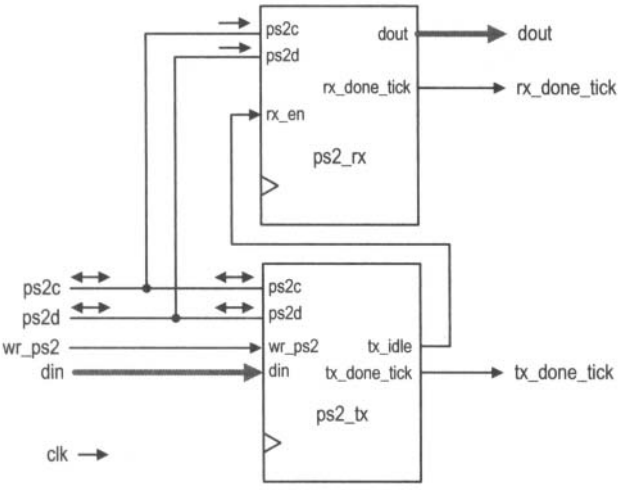


Figure 10.4 Top-level block diagram of a bidirectional PS2 interface.

There is no error detection circuit in this code. A more robust design should check the correctness of the parity and acknowledgment bits and include a watchdog timer to prevent the mouse from being locked in an incorrect state.

10.4 BIDIRECTIONAL PS2 INTERFACE

10.4.1 Basic design and code

We can combine the receiving and transmitting subsystems to form a bidirectional PS2 interface. The top-level diagram is shown in Figure 10.4. We use the `tx_idle` and `rx_en` signals to coordinate the transmitting and receiving operations. Priority is given to the transmitting operation. When the transmitting subsystem is in operation, the `tx_idle` signal is deasserted, which, in turn, disables the receiving subsystem. The receiving subsystem can process input only when the transmitting subsystem is idle. The corresponding HDL code is shown in Listing 10.2.

Listing 10.2 Bidirectional PS2 interface

```
module ps2_rxtx
   (
    input wire clk, reset,
    input wire wr_ps2,
    inout wire ps2d, ps2c,
    input wire [7:0] din,
    output wire rx_done_tick, tx_done_tick,
    output wire [7:0] dout
   );

   // signal declaration
   wire tx_idle;
```

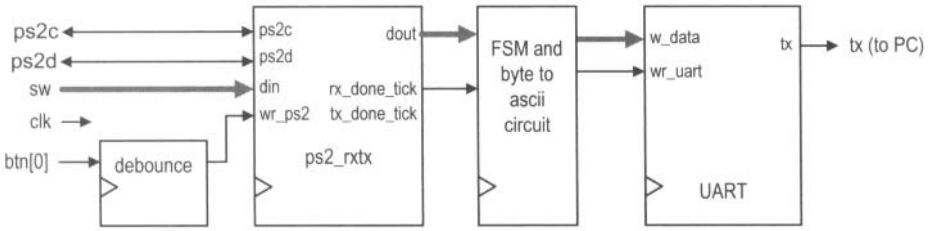


Figure 10.5 Block diagram of a mouse monitor circuit.

```
    // body
    // instantiate ps2 receiver
    ps2_rx ps2_rx_unit
       (.clk(clk), .reset(reset), .rx_en(tx_idle),
        .ps2d(ps2d), .ps2c(ps2c),
        .rx_done_tick(rx_done_tick), .dout(dout));
    // instantiate ps2 transmitter
    ps2_tx ps2_tx_unit
       (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
        .din(din), .ps2d(ps2d), .ps2c(ps2c),
        .tx_idle(tx_idle), .tx_done_tick(tx_done_tick));

endmodule
```

10.4.2 Verification circuit

We create a testing circuit to verify and monitor operation of the bidirectional interface. The block diagram is shown in Figure 10.5. A command is transmitted manually. We use the 8-bit switch to specify the data (i.e., the command from the host) and use a pushbutton to generate a one-clock-cycle tick to transmit the packet. The received packet data is first passed to the `byte-to-ascii` circuit, which converts the data into two ASCII characters plus a blank space. The characters are then transmitted via the UART and displayed in Windows HyperTerminal. The HDL code is shown in Listing 10.3.

Listing 10.3 Bidirectional PS2 interface monitor circuit

```
module ps2_monitor
   (
    input wire clk, reset,
    input wire [7:0] sw,
    input wire [2:0] btn,
    inout wire ps2d, ps2c,
    output wire tx
   );

   // constant declaration
   localparam SP=8'h20; // space in ASCII

  // symbolic state declaration
   localparam [1:0]
```

```
15       idle  = 2'b00,
         send1 = 2'b01,
         send0 = 2'b10,
         sendb = 2'b11;

20    // signal declaration
      reg [1:0] state_reg, state_next;
      wire [7:0] rx_data;
      reg [7:0] w_data, ascii_code;
      wire psrx_done_tick, wr_ps2;
25    reg wr_uart;
      wire [3:0] hex_in;

      // body
      //=================================================
30    // instantiation
      //=================================================
      // instantiate ps2 transmitter/receiver
      ps2_rxtx ps2_rxtx_unit
         (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
35        .din(sw), .dout(rx_data), .ps2d(ps2d), .ps2c(ps2c),
          .rx_done_tick(psrx_done_tick), .tx_done_tick());

      // instantiate UART (only use the UART transmitter)
      uart uart_unit
40       (.clk(clk), .reset(reset), .rd_uart(1'b0),
          .wr_uart(wr_uart), .rx(1'b1), .w_data(w_data),
          .tx_full(), .rx_empty(), .r_data(), .tx(tx));

      // instantiate debounce circuit
45    debounce btn_db_unit
         (.clk(clk), .reset(reset), .sw(btn[0]),
          .db_level(), .db_tick(wr_ps2));

      //=================================================
50    // FSM to send 3 ASCII characters
      //=================================================
      // state registers
      always @(posedge clk, posedge reset)
         if (reset)
55          state_reg <= idle;
         else
            state_reg <= state_next;

      // next-state logic
60    always @*
      begin
         wr_uart = 1'b0;
         w_data = SP;
         state_next = state_reg;
65       case (state_reg)
            idle:
               if (psrx_done_tick) // a scan code received
```

```
                state_next = send1;
            send1: // send higher hex char
                begin
                    w_data = ascii_code;
                    wr_uart = 1'b1;
                    state_next = send0;
                end
            send0: // send lower hex char
                begin
                    w_data = ascii_code;
                    wr_uart = 1'b1;
                    state_next = sendb;
                end
            sendb:  // send blank char
                begin
                    w_data = SP;
                    wr_uart = 1'b1;
                    state_next = idle;
                end
        endcase
    end

    //=============================================================
    // scan code to ASCII display
    //=============================================================
    // split the scan code into two 4-bit hex
    assign hex_in = (state_reg==send1)? rx_data[7:4] :
                                        rx_data[3:0];
    // hex digit to ASCII code
    always @*
     case (hex_in)
        4'h0: ascii_code = 8'h30;
        4'h1: ascii_code = 8'h31;
        4'h2: ascii_code = 8'h32;
        4'h3: ascii_code = 8'h33;
        4'h4: ascii_code = 8'h34;
        4'h5: ascii_code = 8'h35;
        4'h6: ascii_code = 8'h36;
        4'h7: ascii_code = 8'h37;
        4'h8: ascii_code = 8'h38;
        4'h9: ascii_code = 8'h39;
        4'ha: ascii_code = 8'h41;
        4'hb: ascii_code = 8'h42;
        4'hc: ascii_code = 8'h43;
        4'hd: ascii_code = 8'h44;
        4'he: ascii_code = 8'h45;
        default: ascii_code = 8'h46;
     endcase

endmodule
```

If a mouse is connected to the PS2 circuit, we can first issue the FF command to reset the mouse and then issue the F4 command to enable the stream mode. Windows HyperTerminal will show the mouse's acknowledge packets and subsequent mouse movement packets.

10.5 PS2 MOUSE INTERFACE

10.5.1 Basic design

The basic PS2 mouse interface creates another layer over the bidirectional PS2 circuit. Its two basic functions are to enable the stream mode and to reassemble the 3 data bytes. The output of the circuit consists of `xm` and `ym`, which are two 9-bit x- and y-axis movement signals; `btm`, which is the 3-bit button status signal; and `m_done_tick`, which is a one-clock-cycle status signal and is asserted when the assembled data is available.

The HDL code is shown in Listing 10.4. It is implemented by an FSMD with seven states. The `init1`, `init2`, and `init3` states are executed once after the `reset` signal is asserted. In these states, the FSMD issues the F4 command, waits for completion of the transmission, and then waits for the acknowledgment packet. The mouse is in the stream mode now. The FSMD then obtains and assembles the next three packets in the `pack1`, `pack2`, and `pack3` states, and activates the `m_done_tick` signal in the done state. The FSMD circulates these four states afterward.

Listing 10.4 Basic mouse interface circuit

```
module mouse
   (
    input wire clk, reset,
    inout wire ps2d, ps2c,
    output wire [8:0] xm, ym,
    output wire [2:0] btnm,
    output reg  m_done_tick
   );

   // constant declaration
   localparam STRM=8'hf4; // stream command F4

   // symbolic state declaration
   localparam [2:0]
      init1 = 3'b000,
      init2 = 3'b001,
      init3 = 3'b010,
      pack1 = 3'b011,
      pack2 = 3'b100,
      pack3 = 3'b101,
      done  = 3'b110;

   // signal declaration
   reg [2:0] state_reg, state_next;
   wire [7:0] rx_data;
   reg wr_ps2;
   wire rx_done_tick, tx_done_tick;
   reg [8:0] x_reg, y_reg, x_next, y_next;
   reg [2:0] btn_reg, btn_next;
```

```
30

    // body
    // instantiation
    ps2_rxtx ps2_unit
35     (.clk(clk), .reset(reset), .wr_ps2(wr_ps2),
        .din(STRM), .dout(rx_data), .ps2d(ps2d), .ps2c(ps2c),
        .rx_done_tick(rx_done_tick),
        .tx_done_tick(tx_done_tick));

40  // body
    // FSMD state and data registers
    always @(posedge clk, posedge reset)
       if (reset)
          begin
45           state_reg <= init1;
             x_reg <= 0;
             y_reg <= 0;
             btn_reg <= 0;
          end
50     else
          begin
             state_reg <= state_next;
             x_reg <= x_next;
             y_reg <= y_next;
55           btn_reg <= btn_next;
          end

    // FSMD next-state logic
    always @*
60  begin
       state_next = state_reg;
       wr_ps2 = 1'b0;
       m_done_tick = 1'b0;
       x_next = x_reg;
65     y_next = y_reg;
       btn_next = btn_reg;
       case (state_reg)
          init1:
             begin
70              wr_ps2 = 1'b1;
                state_next = init2;
             end
          init2:  // wait for send to complete
             if (tx_done_tick)
75              state_next = init3;
          init3:  // wait for acknowledge packet
             if (rx_done_tick)
                state_next = pack1;
          pack1:  // wait for 1st data packet
80           if (rx_done_tick)
                begin
                   state_next = pack2;
```

```
                  y_next[8] = rx_data[5];
                  x_next[8] = rx_data[4];
                  btn_next = rx_data[2:0];
               end
         pack2: // wait for 2nd data packet
            if (rx_done_tick)
               begin
                  state_next = pack3;
                  x_next[7:0] = rx_data;
               end
         pack3: // wait for 3rd data packet
            if (rx_done_tick)
               begin
                  state_next = done;
                  y_next[7:0] = rx_data;
               end
         done:
            begin
               m_done_tick = 1'b1;
               state_next = pack1;
            end
      endcase
   end
   // output
   assign xm = x_reg;
   assign ym = y_reg;
   assign btnm = btn_reg;

endmodule
```

This design provides only minimal functionalities. A more sophisticated circuit should have a robust method to initiate the stream mode and add an additional buffer, similar to that in Section 8.2.4, to interact better with the external system.

10.5.2 Testing circuit

We use a simple testing circuit to demonstrate use of the PS2 interface. The circuit uses a mouse to control the eight discrete LEDs of the prototyping board. Only one of the eight LEDs is lit and the position of that LED follows the x-axis movement of the mouse. Pressing the left or right button places the lit LED to the leftmost or rightmost position.

The HDL code is shown in Listing 10.5. It uses a 10-bit counter to keep track of the current x-axis position. The counter is updated when a new data item is available (i.e., when the `m_done_tick` signal is asserted). The counter is set to 0 or maximum when the left or right mouse button is pressed. Otherwise, it adds the amount of the signed-extended x-axis movement. A decoding circuit uses the three MSBs of the counter to activate one of the LEDs.

Listing 10.5 Mouse-controlled LED circuit

```
module mouse_led
   (
    input wire clk, reset,
    inout wire ps2d, ps2c,
```

```
    output reg [7:0] led
);

// signal declaration
reg [9:0] p_reg;
wire [9:0] p_next;
wire [8:0] xm;
wire [2:0] btnm;
wire m_done_tick;

// body
// instantiation
mouse mouse_unit
   (.clk(clk), .reset(reset), .ps2d(ps2d), .ps2c(ps2c),
    .xm(xm), .ym(), .btnm(btnm),
    .m_done_tick(m_done_tick));

// counter
always @(posedge clk, posedge reset)
   if (reset)
      p_reg <= 0;
   else
      p_reg <= p_next;

assign p_next = (~m_done_tick) ? p_reg  : // no activity
                (btnm[0])      ? 10'b0  : // left button
                (btnm[1])      ? 10'h3ff : // right button
                p_reg + {xm[8], xm};      // x movement

always @*
   case (p_reg[9:7])
      3'b000: led = 8'b10000000;
      3'b001: led = 8'b01000000;
      3'b010: led = 8'b00100000;
      3'b011: led = 8'b00010000;
      3'b100: led = 8'b00001000;
      3'b101: led = 8'b00000100;
      3'b110: led = 8'b00000010;
      default: led = 8'b00000001;
   endcase

endmodule
```

10.6 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 9.

10.7 SUGGESTED EXPERIMENTS

The mouse is used mainly with a graphic video interface, which is discussed in Chapters 13 and 14. Many additional mouse-related experiments can be found in these chapters.

10.7.1 Keyboard control circuit

A host can issue a command to set certain parameters for a PS2 keyboard as well. For example, we can control the three LEDs of the keyboard by sending `ED 0X`. The `X` is a hexadecimal number with a format of "*0snc*", where s, n, and c are 1-bit values that control the Scroll, Num, and Caps Lock LEDs, respectively. We can incorporate this feature into the keyboard interface circuit of Section 9.4.1 and use a 3-bit switch to control the three keyboard LEDs. Design the expanded interface circuit, resynthesize the circuit, and verify its operation.

10.7.2 Enhanced mouse interface

For the mouse interface discussed in Section 10.5, we can alter the design to manually enable or disable the steam mode. This can be done by using two pushbuttons of the FPGA prototyping board. One button issues the reset command, `FF`, which disables the stream mode during operation, and the other button issues the `F4` command to enable the steam mode. Modify the original interface to incorporate this feature, and resynthesize the LED testing circuit to verify its operation.

10.7.3 Mouse-controlled seven-segment LED display

We can use the mouse to enter four decimal digits on the four-digit seven-segment LED display. The circuit functions as follows:

- Only one of the four decimal points of the LED display is lit. The lit decimal point indicates the location of the selected digit.
- The location of the selected digit follows the x-axis movement of the mouse.
- The content of the select seven-segment LED display is a decimal digit (i.e., 0, . . . , 9) and changes with the y-axis movement of the mouse.

Design and synthesize this circuit and verify its operation.

CHAPTER 11

EXTERNAL SRAM

11.1 INTRODUCTION

Random access memory (RAM) is used for massive storage in a digital system since a RAM cell is much simpler than an FF cell. A commonly used type of RAM is the asynchronous static RAM (SRAM). Unlike a register, in which the data is sampled and stored at an edge of a clock signal, accessing data from an asynchronous SRAM is more complicated. A read or write operation requires that the data, address, and control signals be asserted in a specific order, and these signals must be stable for a certain amount of time during the operation.

It is difficult for a synchronous system to access an SRAM directly. We usually use a *memory controller* as the interface, which takes commands from the main system synchronously and then generates properly timed signals to access the SRAM. The controller shields the main system from the detailed timing and makes the memory access appear like a synchronous operation. The performance of a memory controller is measured by the number of memory accesses that can be completed in a given period. While designing a simple memory controller is straightforward, achieving optimal performance involves many timing issues and is quite difficult.

The S3 board has two 256K-by-16 asynchronous SRAM devices, which total 1M bytes. In this chapter, we demonstrate the construction of a memory controller for these devices. Since the timing characteristics of each RAM device are different, the controller is applicable only to this particular device. However, the same design principle can be used for similar

SRAM devices. The Xilinx Spartan-3 device also contains smaller embedded memory blocks. Use of this memory is discussed in Chapter 12.

11.2 SPECIFICATION OF THE IS61LV25616AL SRAM

11.2.1 Block diagram and I/O signals

The S3 board has two IS61LV25616AL devices, which are 256K-by-16 SRAM manufactured by Integrated Silicon Solution, Inc. (ISSI). A simplified block diagram is shown in Figure 11.1(a). This device has an 18-bit address bus, `ad`, a bidirectional 16-bit data bus, `dio`, and five control signals. The data bus is divided into upper and lower bytes, which can be accessed individually. The five control signals are:

- `ce_n` (chip enable): disables or enables the chip
- `we_n` (write enable): disables or enables the write operation
- `oe_n` (output enable): disables or enables the output
- `lb_n` (lower byte enable): disables or enables the lower byte of the data bus
- `ub_n` (upper byte enable): disables or enables the upper byte of the data bus

All these signals are active low and the `_n` suffix is used to emphasize this property. The functional table is shown in Figure 11.1(b). The `ce_n` signal can be used to accommodate memory expansion, and the `we_n` and `oe_n` signals are used for write and read operations. The `lb_n` and `ub_n` signals are used to facilitate the byte-oriented configuration.

In the remainder of the chapter, we illustrate the design and timing issues of a memory controller. For clarity, we use one SRAM device and access the SRAM in 16-bit word format. This means that the `ce_n`, `lb_n`, and `ub_n` signals should always be activated (i.e., tied to 0). The simplified functional table is shown in Figure 11.1(c).

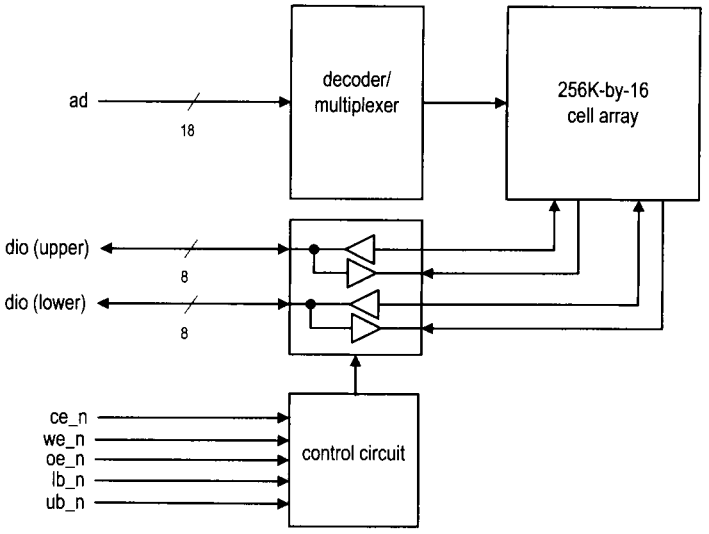
11.2.2 Timing parameters

The timing characteristics of an asynchronous SRAM are quite complex and involve more than two dozen parameters. We concentrate on only a few key parameters that are relevant to our design.

The simplified timing diagrams for two types of read operations are shown in Figure 11.2(a) and (b). The relevant timing parameters are:

- t_{RC}: read cycle time, the minimal elapsed time between two read operations. It is about the same as t_{AA} for SRAM.
- t_{AA}: address access time, the time required to obtain stable output data after an address change.
- t_{OHA}: output hold time, the time that the output data remains valid after the address changes. This should not be confused with the hold time of an edge-triggered FF, which is a constraint for the `d` input.
- t_{DOE}: output enable access time, the time required to obtain valid data after `oe_n` is activated.
- t_{HZOE}: output enable to high-Z time, the time for the tri-state buffer to enter the high-impedance state after `oe_n` is deactivated.
- t_{LZOE}: output enable to low-Z time, the time for the tri-state buffer to leave the high-impedance state after `oe_n` is activated. Note that even when the output is no longer in the high-impedance state, the data is still invalid.

Values of these parameters for the IS61LV25616AL device are shown in Figure 11.2(c).



(a) Block diagram

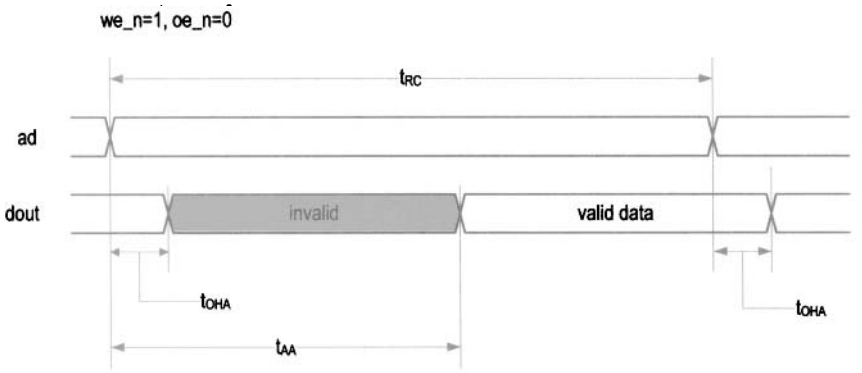
Operation	ce_n	we_n	oe_n	lb_n	ub_n	dio (lower)	dio (upper)
disabled	1	-	-	-	-	Z	Z
	0	1	1	-	-	Z	Z
	0	-	-	1	1	Z	Z
read	0	1	0	0	1	data out	Z
	0	1	0	1	0	Z	data out
	0	1	0	0	0	data out	data out
write	0	0	-	0	1	data in	Z
	0	0	-	1	0	Z	data in
	0	0	-	0	0	data in	data in

(b) Functional table

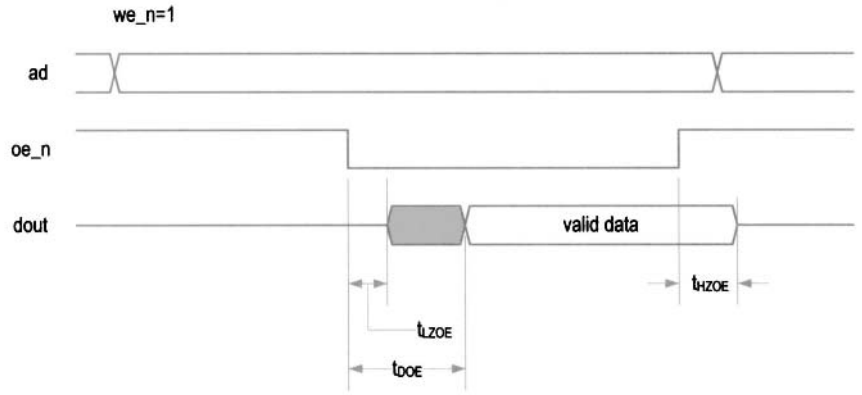
Operation	we_n	oe_n	dio (16 bits)
output disabled	1	1	Z
read 16-bit word	1	0	data out
write 16-bit word	0	-	data in

(c) Simplified functional table

Figure 11.1 Block diagram and functional table of the ISSI 256K-by-16 SRAM.



(a) Timing diagram of an address-controlled read cycle

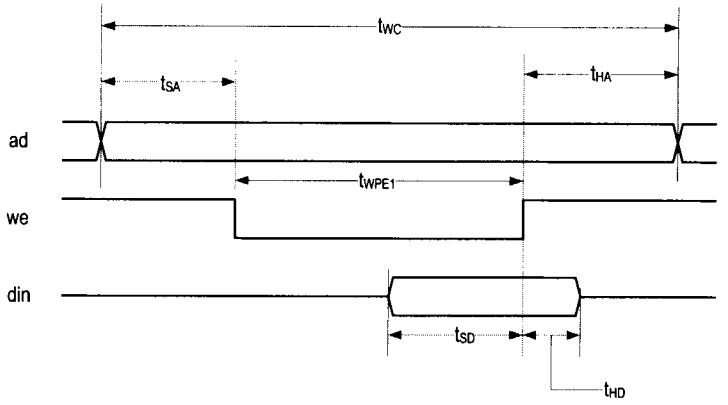


(b) Timing diagram of an oe_n-controlled read cycle

parameter		min	max
t_{RC}	read cycle time	10	–
t_{AA}	address access time	–	10
t_{OHA}	output hold time	2	–
t_{DOE}	output enable access time	–	4
t_{HZOE}	output enable to high-Z time	–	4
t_{LZOE}	output enable to low-Z time	0	–

(c) Timing parameters (in ns)

Figure 11.2 Timing diagrams and parameters of a read operation.



(a) Timing diagram of a write cycle

parameter		min	max
t_{WC}	write cycle time	10	–
t_{SA}	address setup time	0	–
t_{HA}	address hold time	0	–
t_{PWE1}	we_n pulse width	8	–
t_{SD}	data setup time	6	–
t_{HD}	data hold time	0	–

(b) Timing parameter (in ns)

Figure 11.3 Timing diagram and parameters of a write operation.

The simplified timing diagram for a we_n-controlled write operation is shown in Figure 11.3(a). The relevant timing parameters are:

- t_{WC}: write cycle time, the minimal elapsed time between two write operations.
- t_{SA}: address setup time, the minimal time that the address must be stable before we_n is activated.
- t_{HA}: address hold time, the minimal time that the address must be stable after we_n is deactivated.
- t_{PWE1}: we_n pulse width, the minimal time that we_n must be asserted.
- t_{SD}: data setup time, the minimal time that data must be stable before the latching edge (the edge in which we_n moves from 0 to 1).
- t_{HD}: data hold time, the minimal time that data must be stable after the latching edge.

The values of these parameters for the IS61LV25616AL device are shown in Figure 11.3(b). The complete timing information can be found in the data sheet of the IS61LV25616AL device.

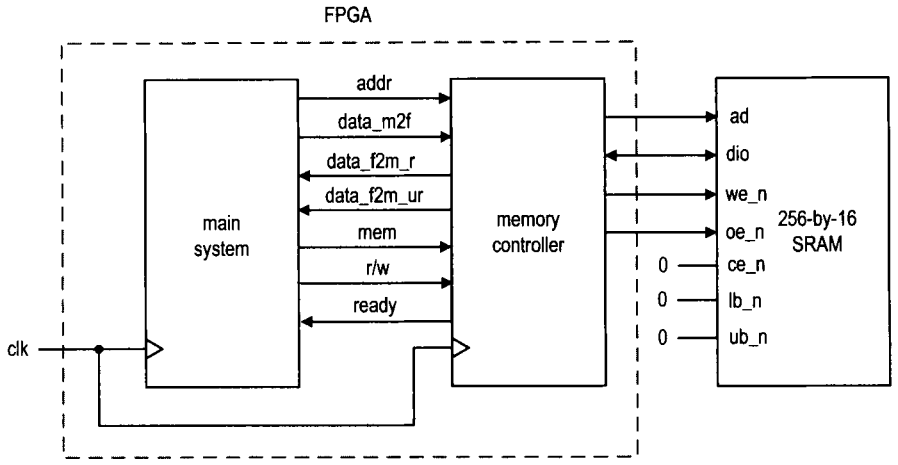


Figure 11.4 Role of an SRAM memory controller.

11.3 BASIC MEMORY CONTROLLER

11.3.1 Block diagram

The role of a memory controller and its I/O signals are shown in Figure 11.4. The signals to the SRAM side are discussed in Section 11.2.1. The signals to the main system side are:

- `mem`: is asserted to 1 to initiate a memory operation.
- `rw`: specifies whether the operation is a read (1) or write (0) operation.
- `addr`: is the 18-bit address.
- `data_f2s`: is the 16-bit data to be written to the SRAM (the `_f2s` suffix stands for FPGA to SRAM).
- `data_s2f_r`: is the 16-bit registered data retrieved from the SRAM (the `_s2f` suffix stands for SRAM to FPGA).
- `data_s2f_ur`: is the 16-bit unregistered data retrieved from SRAM.
- `ready`: is a status signal indicating whether the controller is ready to accept a new command. This signal is needed since a memory operation may take more than one clock cycle.

The memory controller basically provides a "synchronous wrap" around the SRAM. When the main system wants to access the memory, it places the address and data (for a write operation) on the bus and activates the command (i.e., the `mem` and `rw` signals). At the rising edge of the clock, all signals are sampled by the memory controller and the desired operation is performed accordingly. For a read operation, the data becomes available after one or two clock cycles.

The block diagram of a memory controller is shown in Figure 11.5. Its data path contains one address register, which stores the address, and two data registers, which store the data from each direction. Since the data bus, `dio`, is a bidirectional signal, a tri-state buffer is needed. The control path is an FSM, which follows the timing diagrams and specifications in Figures 11.2 and 11.3 to generate a proper control sequence.

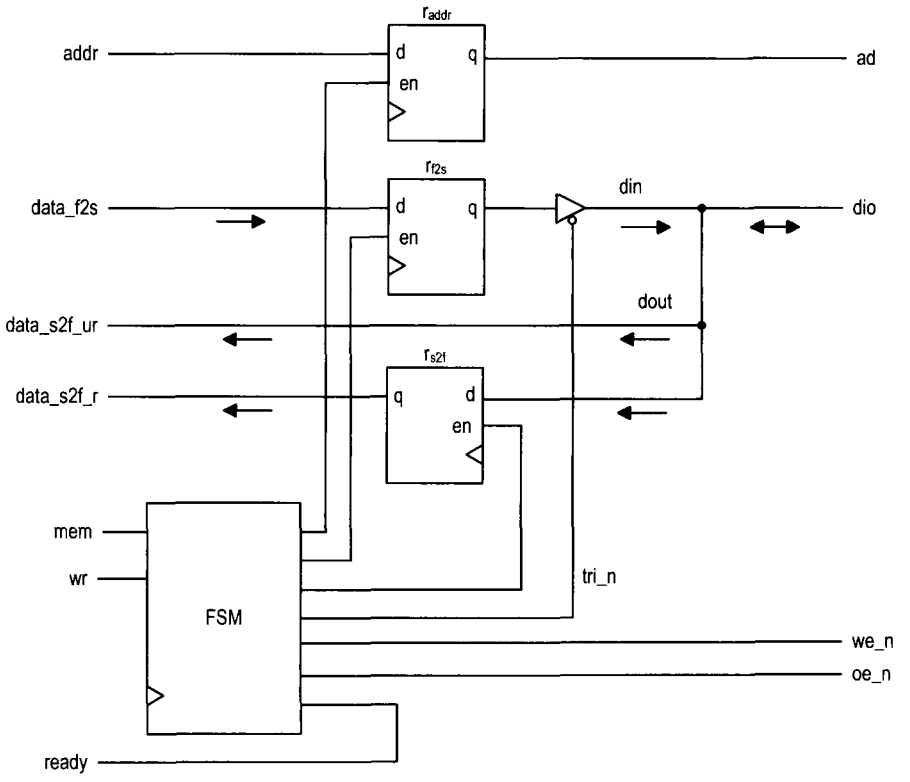


Figure 11.5 Block diagram of a memory controller.

11.3.2 Timing requirement

Although the timing diagrams appear to be complicated at first glance, the control sequences are fairly simple. Let us first consider a read cycle. The `we_n` should be deactivated during the entire operation. Its basic operation sequence is:

1. Place the address on the `ad` bus and activate the `oe_n` signal. These two signals must be stable for the entire operation.
2. Wait for at least t_{AA}. The data from the SRAM becomes available after this interval.
3. Retrieve the data from `dio` and deactivate the `oe_n` signal.

We use the `we_n`-controlled write cycle in our design, as shown in Figure 11.3(a). The basic operation sequence is:

1. Place the address on the `ad` bus and data on the `dio` bus and activate the `we_n` signal. These signals must be stable for the entire operation.
2. Wait for at least t_{PWE1}.
3. Deactivate the `we_n` signal. The data is latched to the SRAM at the 0-to-1 transition edge.
4. Remove the data from the `dio` bus.

Note that t_{HD} (data hold time after write ends) is 0 ns for this SRAM, which implies that it is theoretically possible to remove the data and deactivate `we_n` simultaneously. However, because of the variations in propagation delays, this condition cannot be guaranteed in a

real circuit. To achieve proper latching, we need to ensure that the we_n signal is always deactivated first.

11.3.3 Register file versus SRAM

We discuss the design of a register file in Section 4.2.3. Its basic storage elements are D FFs and thus it is completely synchronous. Although a memory controller wraps the SRAM in a synchronous interface, there are several differences:

- A register file usually has one write port and multiple read ports.
- The read and write ports of a register file can be accessed at the same time (i.e., the read and write operations can be done at the same time).
- Writing to a register takes only one clock cycle.
- Data from a register's read ports is always available and the read operation involves no clock or additional control signals.

In summary, a register file is faster and more flexible. However, due to the circuit size of an FF, a register file is feasible only for small storage.

11.4 A SAFE DESIGN

With the block diagram of Figure 11.5, the remaining task is to derive the controller. Our first scheme uses a "safe" design, which means that the design provides large timing margins and does not impose any stringent timing constraints. The control signals are generated directly from the FSM. The controller uses two clock cycles (i.e., 40 ns) to complete memory access and requires three clock cycles (i.e., 60 ns) for back-to-back operations.

11.4.1 ASMD chart

The ASMD chart for this controller is shown in Figure 11.6. The FSM has five states and is initially in the idle state. It starts the memory operation when the mem signal is activated. The rw signal determines whether it is a read or write operation.

For a read operation, the FSM moves to the rd1 state. The memory address, addr, is sampled and stored in the addr_reg register at the transition. The oe_n signal is activated in the rd1 and rd2 states. At the end of the read cycle, the FSM returns to the idle state. The retrieved data is stored in the data_s2f_reg register at the transition, and the oe_n signal is deactivated afterward. Note that the block diagram of Figure 11.5 has two read ports. The data_s2f_r signal is a registered output and becomes available *after* the FSM exits the r2 state. The data remains unchanged until the end of the next read cycle. The data_s2f_ur signal is connected directly to the SRAM's dio bus. Its data should become valid at the end of the rd2 state but will be removed after the FSM enters the idle state. In some applications, the main system samples and stores the memory readout in its own register, and the unregistered output allows this action to be completed one clock cycle earlier.

For a write operation, the FSM moves to the wr1 state. The memory address, addr, and data, data_f2s, are sampled and stored in the addr_reg and data_f2s_reg registers at the transition. The we_n and tri_n signals are both activated in the wr1 state. The latter enables the tri-state buffer to put the data over the SRAM's dio bus. When the FSM moves to the wr2 state, we_n is deactivated but tri_n remains asserted. This ensures that the data is properly latched to the SRAM when we_n changes from 0 to 1. At the end of the write

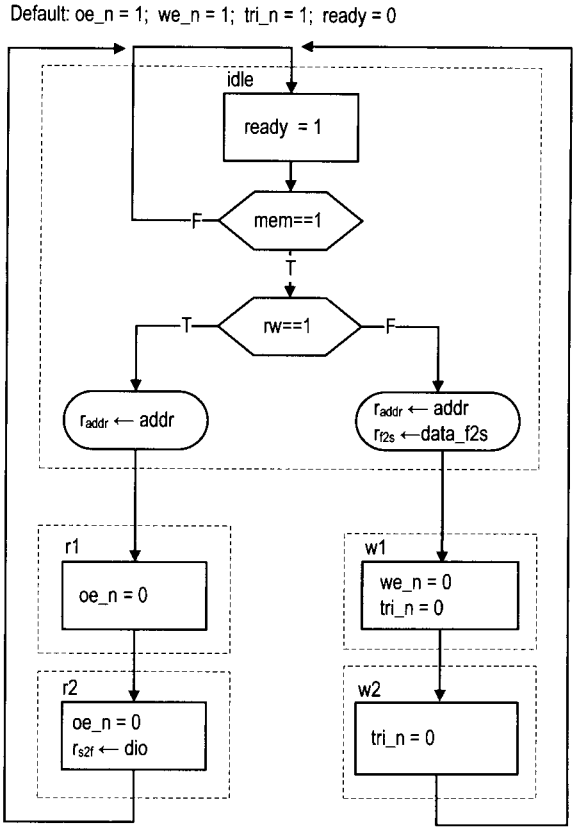


Figure 11.6 ASMD chart of a safe SRAM controller.

cycle, the FSM returns to the `idle` state and `tri_n` is deactivated to remove data from the `dio` bus.

11.4.2 Timing analysis

To ensure correct operation of a memory controller, we must verify that the design meets various timing requirements. Recall that the FSM is controlled by a 50-MHz clock signal and thus stays in each state for 20 ns.

During the read cycle, `oe_n` is asserted for two states, totaling 40 ns, which provides a 30-ns margin over the 10-ns t_{AA}. Although it appears that `oe_n` can be deasserted in the `rd2` state, this imposes a more stringent timing constraint. This issue is explained in Section 11.5.3. The data is stored in the `data_s2f` register when the FSM moves from the `rd2` state to the `idle` state. Although `oe_n` is deasserted at the transition, the data remains valid for a small interval because of the FPGA's pad delay and the t_{HZOE} delay of the SRAM chip. It can be sampled properly by the clock edge.

During the write cycle, `we_n` is asserted in the `wr1` state, and the 20-ns interval exceeds the 8-ns t_{PWE1} requirement. The `tri_n` signal remains asserted in the `wr2` state and thus ensures that the data is still stable during the 0-to-1 transition edge of the `we_n` signal.

In terms of performance, both read and write operations take two clock cycles to complete. During the read operation, the unregistered data (i.e., `data_s2f_ur`) is available at the end of the second clock cycle (i.e., just before the rising edge of the second clock cycle) and the registered data (i.e., `data_s2f_r`) is available right after the rising edge of the second clock cycle. Although a memory operation can be done in two clocks, the main system cannot access memory at this rate. Both read and write operations must return to the `idle` state after completion. The main system must wait for another clock cycle to issue a new memory operation, and thus the back-to-back memory access takes three clock cycles.

11.4.3 HDL implementation

The HDL code can be derived by following the block diagram in Figure 11.5 and the ASMD chart in Figure 11.6. The memory controller must generate fast, glitch-free control signals. One method is to modify the output logic to include *look-ahead output buffers* for the Moore output signals. This scheme adds a buffer (i.e., D FF) for each output signal to remove glitches and reduce clock-to-output delay. To compensate the one clock cycle delay introduced by the buffer, we "look ahead" at the state's future value (i.e., the `state_next` signal) and use it to replace the state's current value (i.e., the `state_reg` signal) in the FSM's output logic.

The complete HDL code is shown in Listing 11.1. To facilitate future expansion, we label the S3 board's two SRAM chips as `a` and `b` and add an `_a` suffix to the SRAM's I/O signals in port declaration. Note that tri-state buffers are required for the bidirectional data signal `dio_a`.

Listing 11.1 SRAM controller with three-cycle back-to-back operation

```
module sram_ctrl
   (
    input wire clk, reset,
    // to/from main system
    input wire mem, rw,
    input wire [17:0] addr,
    input wire [15:0] data_f2s,
    output reg ready,
    output wire [15:0] data_s2f_r, data_s2f_ur,
    // to/from sram chip
    output wire [17:0] ad,
    output wire we_n, oe_n,
    // sram chip a
    inout wire [15:0] dio_a,
    output wire  ce_a_n, ub_a_n, lb_a_n
   );

   // symbolic state declaration
   localparam [2:0]
      idle = 3'b000,
      rd1  = 3'b001,
      rd2  = 3'b010,
      wr1  = 3'b011,
      wr2  = 3'b100;

   // signal declaration
```

```
reg [2:0] state_reg, state_next;
reg [15:0] data_f2s_reg, data_f2s_next;
reg [15:0] data_s2f_reg, data_s2f_next;
reg [17:0] addr_reg, addr_next;
reg we_buf, oe_buf, tri_buf;
reg we_reg, oe_reg, tri_reg;

// body
// FSMD state & data registers
always @(posedge clk, posedge reset)
   if (reset)
      begin
         state_reg <= idle;
         addr_reg <= 0;
         data_f2s_reg <= 0;
         data_s2f_reg <= 0;
         tri_reg <= 1'b1;
         we_reg <= 1'b1;
         oe_reg <= 1'b1;
      end
   else
      begin
         state_reg <= state_next;
         addr_reg <= addr_next;
         data_f2s_reg <= data_f2s_next;
         data_s2f_reg <= data_s2f_next;
         tri_reg <= tri_buf;
         we_reg <= we_buf;
         oe_reg <= oe_buf;
      end

// FSMD next-state logic
always @*
begin
   addr_next = addr_reg;
   data_f2s_next = data_f2s_reg;
   data_s2f_next = data_s2f_reg;
   ready = 1'b0;
   case (state_reg)
      idle:
         begin
            if (~mem)
               state_next = idle;
            else
               begin
                  addr_next = addr;
                  if (~rw)  // write
                     begin
                        state_next = wr1;
                        data_f2s_next = data_f2s;
                     end
                  else  // read
                     state_next = rd1;
```

```
                end
              ready = 1'b1;
           end
        wr1:
           state_next = wr2;
        wr2:
           state_next = idle;
        rd1:
           state_next = rd2;
        rd2:
           begin
              data_s2f_next = dio_a;
              state_next = idle;
           end
        default:
           state_next = idle;
     endcase
  end

  // look-ahead output logic
  always @*
  begin
     tri_buf = 1'b1;  // signals are active low
     we_buf = 1'b1;
     oe_buf = 1'b1;
     case (state_next)
        idle:
           oe_buf = 1'b1;
        wr1:
           begin
              tri_buf = 1'b0;
              we_buf = 1'b0;
           end
        wr2:
           tri_buf = 1'b0;
        rd1:
           oe_buf = 1'b0;
        rd2:
           oe_buf = 1'b0;
     endcase
  end

  // to main system
  assign data_s2f_r = data_s2f_reg;
  assign data_s2f_ur = dio_a;
  // to sram
  assign we_n = we_reg;
  assign oe_n = oe_reg;
  assign ad = addr_reg;
  // i/o for sram chip a
  assign ce_a_n = 1'b0;
  assign ub_a_n = 1'b0;
  assign lb_a_n = 1'b0;
```

```
    assign dio_a = (~tri_reg) ? data_f2s_reg : 16'bz;

135 endmodule
```

To minimize the off-chip pad delay (discussed in Section 11.5.1), the corresponding FPGA's I/O pins should be configured properly. This can be done by adding additional information in the constraint file. A typical line is

```
NET "ad<17>"  LOC = "L3" | IOSTANDARD = LVCMOS33 | SLEW=FAST ;
```

11.4.4 Basic testing circuit

We use two circuits to verify operation of the SRAM controller. The first one is a basic testing circuit that allows us manually to perform a single read or write operation. In addition to the SRAM chip I/O signals, the circuit has the following signals:

- `sw`. It is 8 bits wide and used as data or address input.
- `led`. It is 8 bits wide and used to display the retrieved data.
- `btn[0]`. When it is asserted, the current value of `sw` is loaded to a data register. The output of the register is used as the data input for the write operation.
- `btn[1]`. When it is asserted, the controller uses the value of `sw` as a memory address and performs a write operation.
- `btn[2]`. When it is asserted, the controller uses the value of `sw` as a memory address and performs a read operation. The readout is routed to the `led` signal.

During a write operation, we first specify the data value and load it to the internal register and then specify the address and initiate the write operation. During a read operation, we specify the address and initiate the read operation. The retrieved data is displayed in eight discrete LEDs. The complete HDL code is shown in Listing 11.2.

Listing 11.2 Basic SRAM testing circuit

```
module ram_ctrl_test
   (
    input wire clk, reset,
    input wire [7:0] sw,
5   input wire [2:0] btn,
    output wire [7:0] led,
    output wire [17:0] ad,
    output wire we_n, oe_n,
    inout wire [15:0] dio_a,
10  output wire ce_a_n, ub_a_n, lb_a_n
   );

   // signal declaration
   wire [17:0] addr;
15 wire [15:0] data_s2f;
   reg [15:0] data_f2s;
   reg mem, rw;
   reg [7:0] data_reg;
   wire [2:0] db_btn;
20
   // body
   // instantiation
```

```
    sram_ctrl ctrl_unit
       (.clk(clk), .reset(reset), .mem(mem), .rw(rw),
        .addr(addr), .data_f2s(data_f2s), .ready(),
        .data_s2f_r(data_s2f), .data_s2f_ur(), .ad(ad),
        .we_n(we_n), .oe_n(oe_n), .dio_a(dio_a),
        .ce_a_n(ce_a_n), .ub_a_n(ub_a_n), .lb_a_n(lb_a_n));

    debounce deb_unit0
       (.clk(clk), .reset(reset), .sw(btn[0]),
        .db_level(), .db_tick(db_btn[0]));

    debounce deb_unit1
       (.clk(clk), .reset(reset), .sw(btn[1]),
        .db_level(), .db_tick(db_btn[1]));

    debounce deb_unit2
       (.clk(clk), .reset(reset), .sw(btn[2]),
        .db_level(), .db_tick(db_btn[2]));

    // data registers
    always @(posedge clk)
       if (db_btn[0])
          data_reg <= sw;

    // address
    assign addr = {10'b0, sw};

    //
    always @*
    begin
      data_f2s = 0;
      if (db_btn[1])  // write
         begin
            mem = 1'b1;
            rw = 1'b0;
            data_f2s = {8'b0, data_reg};
         end
      else if (db_btn[2]) // read
         begin
            mem = 1'b1;
            rw = 1'b1;
         end
      else
         begin
            mem = 1'b0;
            rw = 1'b1;
         end
    end
    // output
    assign led = data_s2f[7:0];

endmodule
```

11.4.5 Comprehensive SRAM testing circuit

The second circuit performs comprehensive testing. It verifies operation of the SRAM controller and checks the integrity of the SRAM chip as well. This circuit has three functions:

- Write testing data patterns to the entire SRAM at the maximal rate.
- Read the entire SRAM at the maximal rate, check the retrieved data against the original patterns, and record the number of erroneous readouts.
- Inject erroneous data.

These functions can be initiated by three debounced pushbuttons.

The ASMD chart is shown in Figure 11.7. It contains three branches, corresponding to three functions. The middle branch writes the test patterns to the SRAM. The `wr_clk1`, `wr_clk2`, and `wr_clk3` states correspond to the `idle`, `wr1`, and `wr2` states of the SRAM controller. The FSMD uses the 18-bit `c` register as a counter to loop through this branch 2^{18} times. The content of the `c` register is used as an address and the reversed 16 LSBs are used as data during a write operation. The FSMD writes all memory locations while looping through this branch. The left branch reads data from the SRAM. The three states correspond to the `idle`, `rd1`, and `rd2` states of the SRAM controller. The FSMD again loops through the branch 2^{18} times. The retrieved data is compared with the original test patterns, and the `err` register is used to keep track of the number of mismatches. The right branch performs a single write operation. It uses the 8-bit switch to form a memory address and writes an erroneous pattern to that address. The `inj` counter is used to keep track of the number of injected errors. The complete HDL code is shown in Listing 11.3.

Listing 11.3 Comprehensive SRAM testing circuit

```
module sram_test
   (
    input wire clk, reset,
    input wire [7:0] sw,
    input wire [2:0] btn,
    output wire [3:0] an,
    output wire [7:0] led, sseg,
    output wire [17:0] ad,
    output wire we_n, oe_n,
    inout wire [15:0] dio_a,
    output wire ce_a_n, ub_a_n, lb_a_n
   );

   // symbolic state declaration
   localparam [2:0]
      test_init = 3'b000,
      rd_clk1   = 3'b001,
      rd_clk2   = 3'b010,
      rd_clk3   = 3'b011,
      wr_err    = 3'b100,
      wr_clk1   = 3'b101,
      wr_clk2   = 3'b110,
      wr_clk3   = 3'b111;

   // signal declaration
   reg [2:0] state_reg, state_next;
   reg [17:0] addr;
```

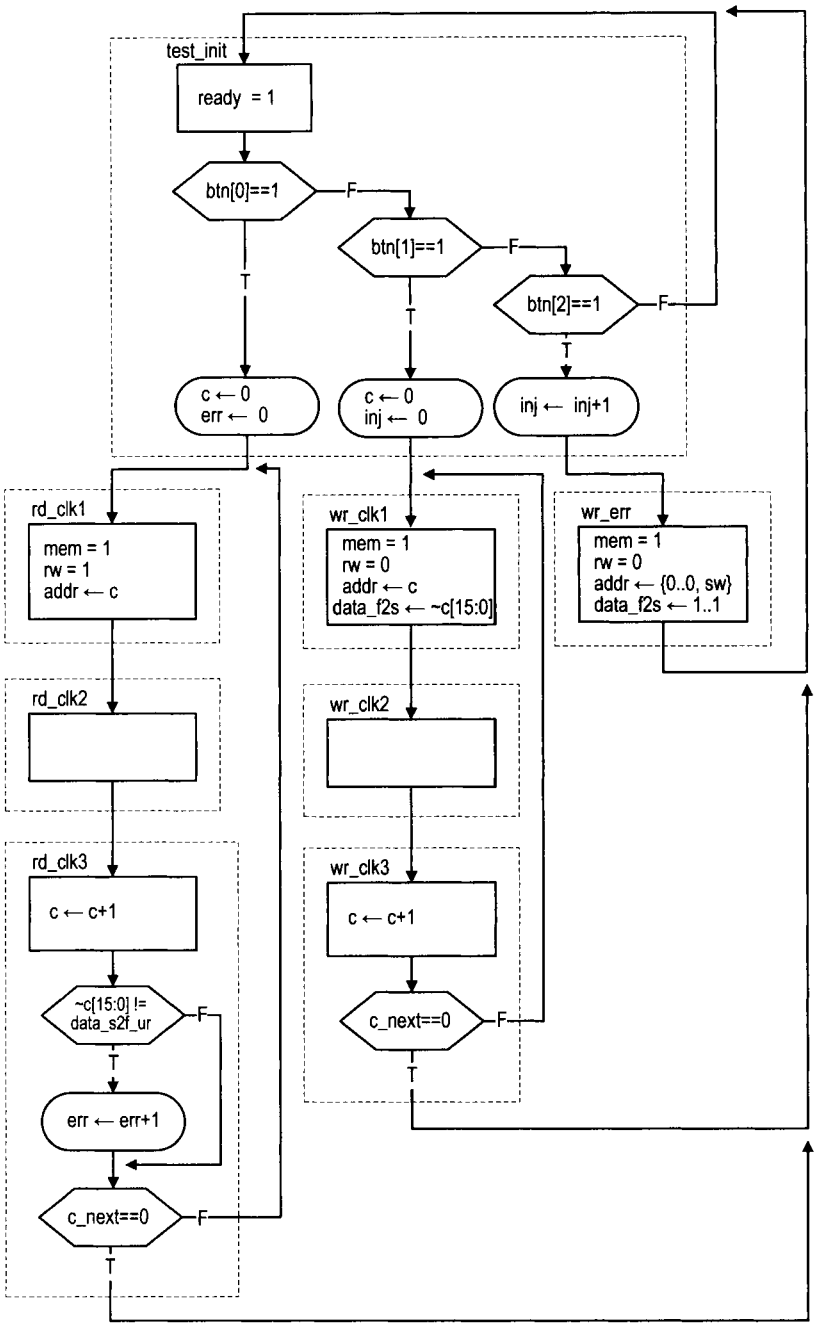


Figure 11.7 ASMD chart of a comprehensive SRAM testing circuit.

```
   wire [15:0] data_s2f;
   reg [15:0] data_f2s;
30 reg mem, rw;
   wire [2:0] db_btn;
   reg [17:0] c_next, c_reg;
   reg [7:0] inj_next, inj_reg;
   reg [15:0] err_next, err_reg;
35
   // body
   //=============================================
   // component instantiation
   //=============================================
40 // instantiation
   sram_ctrl ctrl_unit
      (.clk(clk), .reset(reset), .mem(mem), .rw(rw),
       .addr(addr), .data_f2s(data_f2s), .ready(),
       .data_s2f_r(), .data_s2f_ur(data_s2f), .ad(ad),
45     .we_n(we_n), .oe_n(oe_n), .dio_a(dio_a),
       .ce_a_n(ce_a_n), .ub_a_n(ub_a_n),
       .lb_a_n(lb_a_n));

   debounce deb_unit0
50    (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(db_btn[0]));

   debounce deb_unit1
      (.clk(clk), .reset(reset), .sw(btn[1]),
55     .db_level(), .db_tick(db_btn[1]));

   debounce deb_unit2
      (.clk(clk), .reset(reset), .sw(btn[2]),
       .db_level(), .db_tick(db_btn[2]));
60
   disp_hex_mux disp_unit
      (.clk(clk), .reset(1'b0), .dp_in(4'b1111),
       .hex3(err_reg[15:12]), .hex2(err_reg[11:8]),
       .hex1(err_reg[7:4]), .hex0(err_reg[3:0]),
65     .an(an), .sseg(sseg));

   //=============================================
   //   FSMD
   //=============================================
70 // FSMD state & data registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            state_reg <= test_init;
75          c_reg <= 0;
            inj_reg <= 0;
            err_reg <= 0;
         end
      else
80       begin
```

```
            state_reg <= state_next;
            c_reg <= c_next;
            inj_reg <= inj_next;
            err_reg <= err_next;
        end
    // FSMD next-state logic
    always @*
    begin
        c_next = c_reg;
        inj_next = inj_reg;
        err_next = err_reg;
        addr = 0;
        rw = 1'b1;
        mem = 1'b0;
        data_f2s = 0;
        case (state_reg)
            test_init:
                if (db_btn[0])
                    begin
                        state_next = rd_clk1;
                        c_next = 0;
                        err_next = 0;
                    end
                else if (db_btn[1])
                    begin
                        state_next = wr_clk1;
                        c_next = 0;
                        inj_next = 0;
                    end
                else if (db_btn[2])
                    begin
                        state_next = wr_err;
                        inj_next = inj_reg + 1;
                    end
                else
                    state_next = test_init;
            wr_err: // write 1 error; done in next 2 clocks
                begin
                    state_next = test_init;
                    mem = 1'b1;
                    rw = 1'b0;
                    addr = {10'b0, sw};
                    data_f2s = 16'hffff;
                end
            wr_clk1: // in idle state of sram_ctrl
                begin
                    state_next = wr_clk2;
                    mem = 1'b1;
                    rw = 1'b0;
                    addr = c_reg;
                    data_f2s = ~c_reg[15:0];
                end
            wr_clk2: // in wr1 state of sram_ctrl
```

```
            state_next = wr_clk3;
        wr_clk3: // in wr2 state of sram_ctrl
            begin
                c_next = c_reg + 1;
                if (c_next==0)
                    state_next = test_init;
                else
                    state_next = wr_clk1;
            end
        rd_clk1: // in idle state of sram_ctrl
            begin
                state_next = rd_clk2;
                mem = 1'b1;
                rw = 1'b1;
                addr = c_reg;
            end
        rd_clk2: // in rd1 state of sram_ctrl
            state_next = rd_clk3;
        rd_clk3: // in rd2 state of sram_ctrl
            begin
                // compare readout; must use unregistered output
                if (~c_reg[15:0] != data_s2f)
                    err_next = err_reg + 1;
                c_next = c_reg + 1;
                if (c_next==0)
                    state_next = test_init;
                else
                    state_next = rd_clk1;
            end
    endcase
  end
  // output
  assign led = inj_reg;

endmodule
```

Note that the number of write–read mismatches is connected to the seven-segment LED display and shown as a four-digit hexadecimal number, and the number of injected errors is connected to the eight discrete LEDs.

We can use this circuit as follows:

- Perform the read function. Since the SRAM is not written yet, it is in the initial "power-on" state. The seven-segment LED display should show a large number of mismatches.
- Perform the write function.
- Perform the read function. The number of mismatches should be zero if both the SRAM controller and the SRAM device work properly.
- Inject error data a few times (to different memory locations).
- Perform the read function again. The number of mismatches should be the same as the number of injected errors.

11.5 MORE AGGRESSIVE DESIGN

Although the previous memory controller functions properly, it does not have optimal performance. While both the read and write cycles are 10 ns of the SRAM device, the back-to-back memory access of this controller takes 60 ns (i.e., three clock cycles). In this section, we study the timing issue in more detail, examine several more aggressive designs and their potential problems, and discuss some FPGA features that help to remedy the problems.

11.5.1 Timing issues

Timing issues on asynchronous SRAM There are two subtle timing issues in designing a high-performance asynchronous SRAM controller. The first issue is deactivation of the `we_n` signal. The 0-to-1 transition of `we_n` functions somewhat like a clock edge of an FF, in which the data is latched and stored to the internal memory element. Note that the data hold time (t_{HD}) is zero for this SRAM. Although it appears that it is fine to deactivate `we_n` and remove data at the same time, this approach is not reliable because of the variations in propagation delays. We must ensure that `we_n` is deactivated *before* data is removed from the bus.

The second issue is the potential conflict on the data bus, `dio`. Recall that the data bus is a bidirectional bus. The controller places data on the bus during a write operation, and the SRAM places data on the bus during a read operation. A condition known as *fighting* occurs if the controller and SRAM place data on the bus at the same time. This condition should be avoided to ensure reliable operation.

Estimation of propagation delay Designing a good memory controller requires having a good understanding about the propagation delays of various signals. However, it is a difficult task. First, during synthesis, an RT-level description is optimized and mapped to logic cells and wire interconnects. The final implementation may not resemble the block diagram depicted by the initial description, and thus it is difficult to estimate the propagation delay from the initial description.

Second, a memory operation involves *off-chip* data access. Additional propagation delay is introduced when a signal propagates through the FPGA's I/O pads. The delay, sometimes known as *pad delay*, is usually much larger than the internal wiring delay and its exact value depends on a variety of factors, including the type of FPGA device, the location of the output register (in LE or IOB), the I/O standards, the slew rate, the driver strength, and external loading.

It requires intimate knowledge of the FPGA device and the synthesis software to perform a good timing analysis and to estimate the propagation delays of various signals.

11.5.2 Alternative design I

The first alternative design is targeted to reduce the back-to-back operation overhead. Instead of always returning to the `idle` state, the memory controller can check the `mem` signal at the end of current memory operation (i.e., in the `rd2` or `wr2` state) and determine what to do next. It initiates a new memory operation immediately if there is a pending request.

The revised ASMD chart for this controller is shown in Figure 11.8. In the `rd2` and `wr2` states, the `mem` and `rw` signals are examined and the FSMD may move directly to the `rd1` or `wr1` state if another memory operation is required.

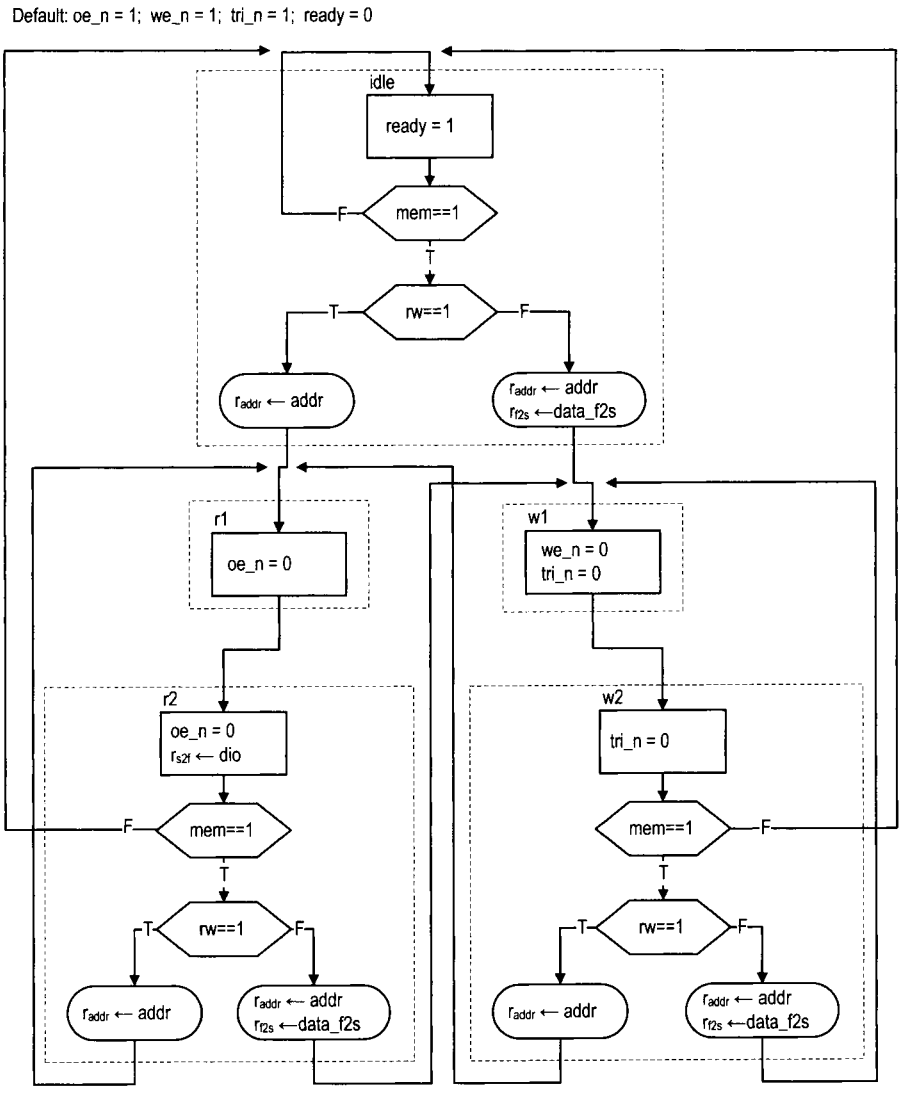


Figure 11.8 ASMD chart of SRAM controller design I.

Timing analysis Most of the original timing analysis in Section 11.4.2 can still be applied to this design. However, skipping the `idle` state introduces subtle new complications when different types of back-to-back memory operations are performed. The issue is the potential fighting on the data bus.

Let us consider a write operation performed immediately after a read operation. During the read operation, the signal flows from the SRAM to the FPGA. To facilitate this operation, the tri-state buffer of the SRAM should be "turned on" (i.e., passing signal) and the tri-state buffer of the FPGA should be "turned off" (i.e., high impedance). During the write operation, the signal flows from the FPGA to the SRAM, and the roles of the two tri-state buffers are reversed. Note that a small delay is required to turn on or off a tri-state buffer. In the SRAM chip, these delays are specified by t_{HZOE} (`oe_n` to high-impedance time) and t_{LZOE} (`oe_n` to low-impedance time) in Figure 11.2.

In the original SRAM controller, both tri-state buffers are turned off in the `idle` state. The state provides enough time for the data bus to settle to the high-impedance condition. The new design requires the two tristate buffers to reverse directions simultaneously during back-to-back operations. For example, when moving from the `rd2` state to the `wr1` state, the FSMD generates signals to turn off the SRAM's tri-state buffer and to turn on the FPGA's tri-state buffer. A problem may occur in this transition if the SRAM's tri-state buffer is turned off too slowly or the FPGA's tri-state buffer is turned on too quickly. In a small interval, both buffers may allow data to be placed on the bus and fighting occurs. Similarly, fighting may occur when a read operation is performed immediately after a write operation.

Since the interval tends to be very small, the fighting should not cause severe damage to the devices but may introduce a large transient current which makes the design less reliable. We must do a detailed timing analysis to examine whether fighting occurs and may even need to fine-tune the timing to fix the problem. As discussed in Section 11.5.1, it is a difficult task.

11.5.3 Alternative design II

Timing analysis in Section 11.4.2 shows that the initial design provides a large safety margin. In this controller, a memory operation takes two clock cycles, which amount to 40 ns. Since the read and write cycles of the SRAM are each 10 ns, we naturally wonder whether it is possible to reduce the operation time to a single 20-ns clock cycle. This can be done by eliminating the `rd2` and `wr2` states in the ASMD chart. The second alternative design uses this approach. The revised ASMD chart is shown in Figure 11.9. It takes one clock cycle to complete the memory access and requires two clock cycles to complete the back-to-back operations.

Timing analysis Reducing a state from the original controller imposes much tighter timing constraints for both read and write operations. Let us first consider the read operation. During operation, the address signal first propagates through the FPGA's I/O pads to the SRAM's address bus, and the retrieved data then propagates back through the I/O pads to FPGA's internal logic. All of this must be completed within a 20-ns clock cycle. In addition to the 10-ns SRAM address access time (i.e., t_{AA}), the cycle must accommodate two pad delays. The pad delay of a Spartan-3 device can range from 4 ns to more than 10 ns. Therefore, we need to "fine-tune" the synthesis to achieve this margin.

Unlike the read operation, a write operation is "one-way" and only needs to propagate the address, data, and control signals to the SRAM chip. If we assume that the signals experience similar pad delays, the absolute value of the delay is a lesser issue. Instead, the

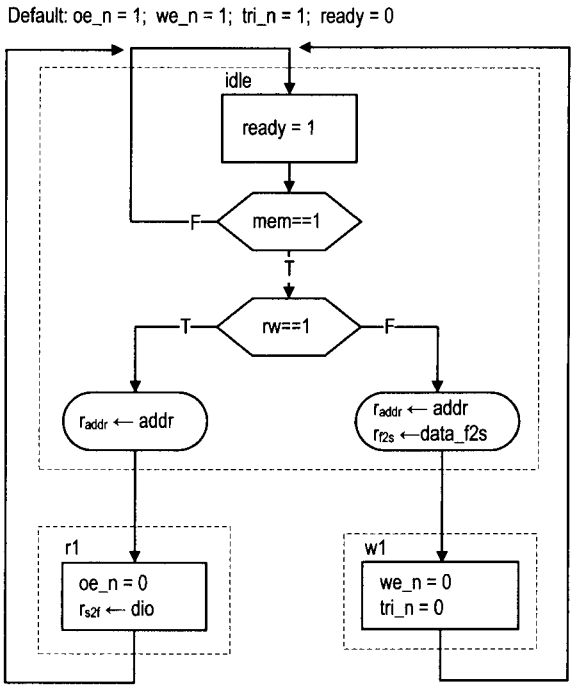


Figure 11.9 ASMD chart of SRAM controller design II.

key is the *order* of signals being activated and deactivated. As discussed in Section 11.5.1, `we_n` must be deactivated before data to latch the data properly to the SRAM. In the original design, this is achieved by including the second state in the write operation, `wr2`, in which `we_n` is deactivated but the data is still available (i.e., `tri_n` is still active). In the revised controller, the `we_n` and `tri_n` signals are deactivated simultaneously at the end of the `wr1` state. Due to the variations in the internal logic and pad delays, normal synthesis cannot guarantee that `we_n` is deactivated before the data is removed from the external data bus. Again, for a reliable design, we need to fine-tune the synthesis to satisfy this goal.

11.5.4 Alternative design III

We can combine the features from the two preceding revisions to derive the third alternative design. This new controller eliminates the second clock cycle in the read and write operations and allows back-to-back operation without first returning to the `idle` state. This is the most aggressive design. The revised ASMD chart is shown in Figure 11.10. It combines the modifications from the previous two ASMD charts. The revised design takes one clock cycle to complete the memory access and one clock cycle to complete back-to-back operations.

Note that the `we_n` signal must be asserted for a fraction of the clock period and cannot be shown in the ASMD chart. We use the `we_tmp` in the `wr1` state and later derive `we_n` from this signal.

Timing analysis Since the new design combines the features of the two previous designs, all the timing issues discussed in the two preceding subsections must be considered

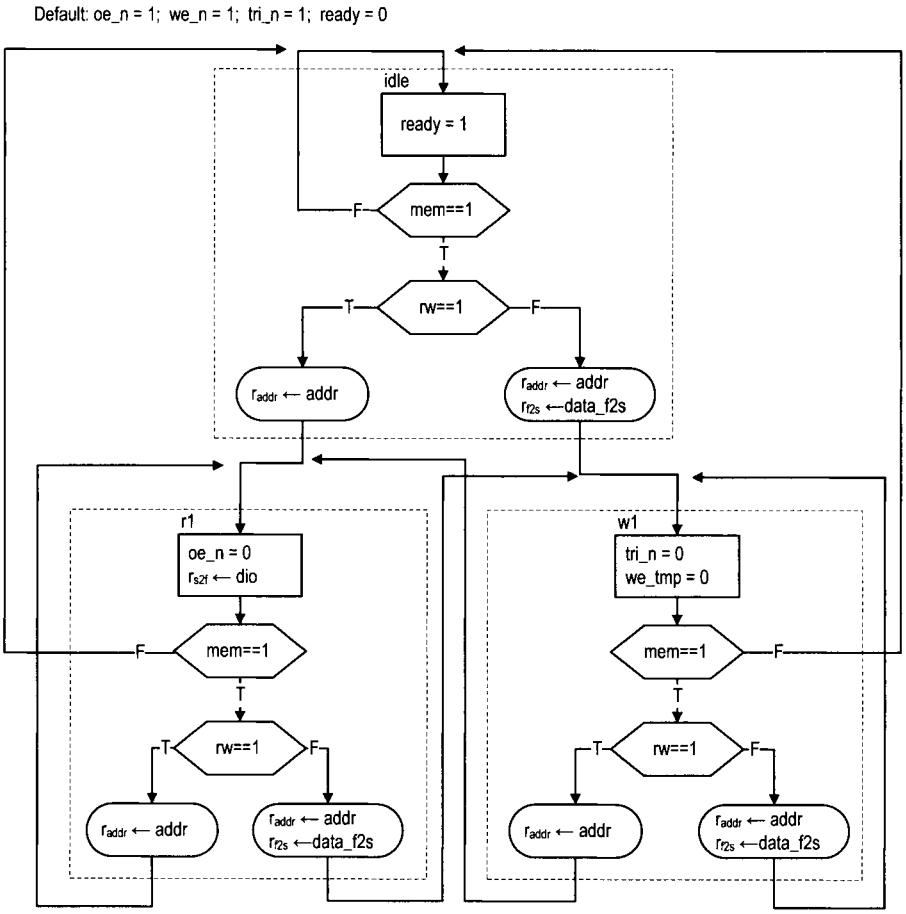


Figure 11.10 ASMD chart of SRAM controller design III.

for this design as well. One additional issue is generation of the we_n signal. During back-to-back write operations, the ASMD stays on the wr1 state. In the original design, the we_n signal is a Moore output. It will be asserted to 0 continuously in this case. The controller does not function properly since the data is latched to the SRAM at the 0-to-1 transition of the we_n signal. To solve the problem, the we_n signal must be asserted in only a fraction of the clock period.

One possible way to solve the problem is to assert the signal only at the first half of the clock, which is 10 ns and can satisfy the t_{WPE1} requirement in theory. Intuitively, we are tempted to do this by gating the we_tmp signal with the clock signal, clk:

```
assign we_n = we_tmp | ~clk;
```

However, this is not a reliable solution because of the potential glitches and delay variation. A better alternative is discussed in the next subsection.

11.5.5 Advanced FPGA features$^{Xilinx\ specific}$

The memory controller examples in this section illustrate the limitations of the FSM-based controller and synchronous design methodology. Basically, an FSM cannot generate a control sequence that is "finer" than the period of its clock signal. The operation of these alternative designs relies on factors that cannot be specified by an RT-level HDL description. Due to the variations in propagation delays, the synthesized circuits are not reliable and may or may not work.

There are some ad hoc features to obtain better control. These features are usually device and software dependent. For example, the digital clock manager (DCM) circuit and input/output block (IOB) of the Spartan-3 device can help to remedy some of the previously discussed problems. Detailed discussion of DCM and IOB is beyond the scope of this book. In this subsection, we sketch a few ideas and illustrate how to apply these features to obtain a more reliable controller.

DCM A Spartan-3 FPGA device contains up to eight *digital clock managers* (DCMs). As its name indicates, a DCM is a circuit that manipulates the system clock signal. It can multiply or divide the frequency or shift the phase of the incoming clock signal to generate new clock signals.

One way to obtain a "finer" control sequence is to use a faster clock. Since implementation of a memory controller is fairly simple, the circuit itself can operate at a faster clock rate. For example, we can isolate the memory controller and drive it with a DCM-generated 200-MHz clock signal, whose period is only 5 ns. Consider the write operation of the ASMD chart in Figure 11.6. In the new controller, each state lasts only 5 ns. To satisfy the 10-ns we_n requirement, we need to expand the wr1 state to two states and assert the we_n signal in these states. The complete write operation now requires four states. However, because of the faster clock rate, the four clock cycles amount to only 20 ns, which is much better than the original 60-ns design.

A simple application of clock phase shift is discussed in the next subsection.

IOB An *input/output block* (IOB) of a Spartan-3 FPGA device provides a programmable interface between an I/O pin and the device's internal logic. It contains several storage registers and tri-state buffers as well as analog driver circuits that can be configured to provide different slew rates and driver strength and to support a variety of I/O standards.

To minimize the off-chip pad delay discussed in Section 11.5.3, we can put the output registers of the memory controller to the FFs inside the IOBs and configure the driver with

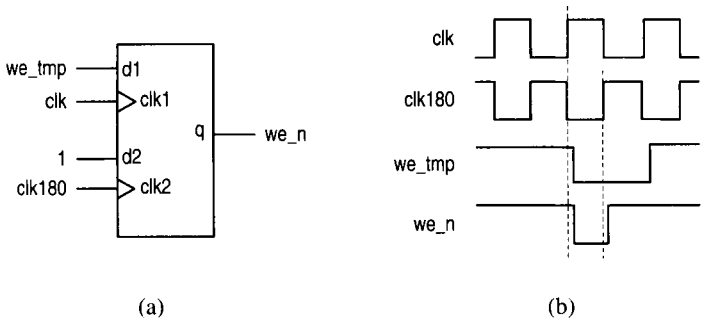


Figure 11.11 Generating a half-cycle signal with DDR.

the proper slew rate and strength. This can be done by specifying the desired condition and configuration in the constraint file.

An IOB also contains a *double data rate* (DDR) register, which has two clocks and two inputs. Conceptually, we can think that the two inputs are sampled independently by the two clocks and the sampled values are stored in the same register. The DDR register and DCM can be combined to generate a control signal whose width is a fraction of a clock signal, as the `we_n` signal discussed in Section 11.5.4. The block diagram is shown in Figure 11.11(a). The regular output register is replaced with a DDR register. The top portion of the DDR consists of the *we_tmp* signal and the original clock signals, `clk`. The bottom input of the DDR is tied to 1 and the clock is connected to the out-of-phase clock signal, `clk180`, which is generated by a DCM. The 1 is always loaded at the rising edge of the `clk180` signal, which corresponds to the falling edge of the `clk` signal. It essentially deactivates the second half of the *we_n* signal. The timing diagram is shown in Figure 11.11(b). This approach generates a clean half-cycle signal and is far more reliable than the clock gating scheme discussed in Section 11.5.4.

11.6 BIBLIOGRAPHIC NOTES

The data sheet published by ISSI provides detailed information for the IS61LV25616AL SRAM device. The Xilinx application note, *XAPP462 Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*, discusses the use of DCM, and the data sheet, *DS099 Spartan-3 FPGA Family: Complete Data Sheet*, explains the architecture and configuration of the IOB and the DDR register.

11.7 SUGGESTED EXPERIMENTS

11.7.1 Memory with a 512K-by-16 configuration

There are two 256K-by-16 SRAM chips, and their I/O connections are shown in the manual of the S3 board. We can expand them to form a 512K-by-16 SRAM.

1. Derive a scheme to combine the two chips.
2. Follow the procedure in Section 11.4 to design a memory controller for the 512K-by-16 SRAM. Derive the HDL description.

3. Modify the testing circuit in Section 11.4.5 for the new controller and derive the HDL description.
4. Synthesize the testing circuit and verify operation of the controller and SRAM chips.

11.7.2 Memory with a 1M-by-8 configuration

Repeat Experiment 11.7.1 but configure the two chips as a 1M-by-8 SRAM. The `lb_n` and `ub_n` signals can be used for this purpose.

11.7.3 Memory with an 8M-by-1 configuration

A single bit of the 256K-by-16 SRAM can be written as follows:

- Read a 16-bit word.
- Modify the designated bit in the word.
- Write the 16-bit word back.

Repeat Experiment 11.7.1 but configure the two chips as an 8M-by-1 SRAM.

11.7.4 Expanded memory testing circuit

The memory testing circuit in Section 11.4.5 conducts exhaustive back-to-back read and back-to-back write tests. We can expand the circuit to include an exhaustive "read-after-write" test, in which the testing circuit issues write and read operations alternately for the entire memory space. To make the test more effective, the writing and reading addresses should be different. For example, we can make the read operation retrieve the data written 16 positions earlier (i.e., if the current writing address is `c`, the reading address will be `c-16`). Create a modified ASMD chart, derive an HDL description, synthesize the circuit, and verify its operation.

11.7.5 Memory controller and testing circuit for alternative design I

Derive the HDL code for alternative design I in Section 11.5.2 and create an expanded testing circuit similar to the one in Experiment 11.7.4. Synthesize the testing circuit and examine whether any error occurs during operation.

11.7.6 Memory controller and testing circuit for alternative design II

Repeat the process in Experiment 11.7.5 for alternative design II discussed in Section 11.5.3.

11.7.7 Memory controller and testing circuit for alternative design III

Repeat the process in Experiment 11.7.5 for alternative design III discussed in Section 11.5.4.

11.7.8 Memory controller with DCM

Study the application note on DCM and follow the discussion in Section 11.5.5 to drive the safe memory controller discussed in Section 11.4 with a higher clock rate (150 MHz or even 200 MHz). Derive an ASMD chart and HDL code, and create a new testing circuit. Synthesize the circuit and verify operation of the memory controller and the SRAM.

11.7.9 High-performance memory controller

Study the documentation of the DCM and the IOB and apply these features to reconstruct alternative design III discussed in Section 11.5.4. Create a new testing circuit. Synthesize the circuit and verify operation of the memory controller and the SRAM.

CHAPTER 12

XILINX SPARTAN-3 SPECIFIC MEMORY

12.1 INTRODUCTION

A digital system frequently requires memory for storage. To facilitate this need, most FPGA devices contain dedicated embedded memory modules. While these modules cannot replace the massive external memory devices, they are useful for applications that require small or intermediate-sized memory.

Although the basic internal structure of memory modules is similar, there are many subtle differences in their I/O interfaces. It is usually difficult for synthesis software to extract the desired features from the code and to infer a matching memory module from the underlying device library. In Xilinx ISE, we can use *HDL instantiation*, the Core Generator program, or the *behavioral HDL inference template* to incorporate an embedded memory module into a design. The third one is semi-device independent and we use this method in this book. In this chapter, we briefly examine Spartan-3 memory modules and the first two methods and provide detailed descriptions of several key behavioral HDL templates.

12.2 EMBEDDED MEMORY OF SPARTAN-3 DEVICE

12.2.1 Overview

There are two types of embedded memory in a Spartan-3 device: distributed RAM and block RAM. A *distributed RAM* is constructed from the logic cell's look-up table (LUT). The LUT can be configured as a 16-by-1 synchronous RAM, and multiple LUTs can be

cascaded to form a wider and deeper memory module. The Spartan-3 XC3S200 device of the S3 board can provide up to 30K bits of distributed memory, which is small compared to a block RAM or external memory. Furthermore, since the distributed RAM uses the logic cells, it competes for resources with the normal logic. Thus, it is feasible only for applications that require relatively small storage.

A *block RAM* is a special memory module embedded in an FPGA device and is separated from the regular logic cells. It can be thought of as a fast SRAM wrapped by a synchronous, configurable interface. Each block RAM consists of 16K (2^{14}) data bits plus optional 2K parity bits. It can be organized in different widths, from 16K by 1 (i.e., 2^{14} by 2^0) to 512 by 32 (i.e., 2^9 by 2^5). The Spartan-3 XC3S200 device has 12 block RAMs, totaling 172K data bits. These block RAMs can be used for intermediate-sized applications, such as a FIFO, a large look-up table, or an intermediate-sized local memory. In comparison, the external SRAM chips of the S3 board have a capacity of 8M bits.

Both the distributed RAM and block RAM are already "wrapped" with a synchronous interface, and thus no additional memory controller circuit is needed. They are very flexible and can be configured to perform single- and dual-port access and to support various types of buffering and clocking schemes. Detailed discussion is beyond the scope of this book. We only examine several commonly used configurations, including a synchronous single-port RAM, a synchronous dual-port RAM, and a ROM in Section 12.4.

12.2.2 Comparison

The Spartan-3 device and the S3 board provide several options for storage elements. It is a good idea to keep in mind the relative capacities of these options:

- *XC3S200's FFs* (for registers): about 4.5K bits, embedded in logic cells and I/O buffers
- *XC3S200's distributed RAM*: 30K bits, constructed from the logic cells
- *XC3S200's block RAM*: 172K bits, configured as twelve 16K-bit modules
- *External SRAM*: 8M bits, configured as two 256K-by-16 SRAM chips

This helps us to decide which option is most suitable for an application at hand.

12.3 METHOD TO INCORPORATE MEMORY MODULES

Although memory modules have similar internal structure, there are many subtle differences in their interfaces, such as the numbers of read and write ports, clocking scheme, data and address buffering, enable and reset signals, and initial values. Although it is possible to describe the desired module behaviors in HDL code, the synthesis software may or may not recognize the designer's intention. Therefore, the HDL code cannot always infer the proper memory module and is normally not portable. In Xilinx ISE, there are three methods to incorporate an embedded memory module into a design:

- HDL instantiation
- The Core Generator program
- The behavioral HDL inference template

The first two are specific for Xilinx devices and the third is a semi-device-independent behavioral description. Because of the clarity of the behavioral description, we use the third method in this book. We provide a brief overview of the three methods in this section.

12.3.1 Memory module via HDL component instantiation

We have used HDL component instantiation in many earlier design examples to include predesigned modules or to create a hierarchy. Instantiating a Xilinx memory module is similar except that there is no HDL description for the architecture body. We must check the manual to find the exact module name and the associated parameters and I/O port definitions. This is a tedious process and is particularly error-prone for memory modules because of the large number of configurations and options.

The instantiation code for many Xilinx components can be obtained directly from ISE by selecting Edit ≻ Language Templates. The following are segments of a 16K-by-1 dual-port RAM:

```
// RAMB16_S1_S1: Virtex-II/II-Pro,
// Spartan-3/3E 16k x 1 Dual-Port RAM
// Xilinx HDL Language Template version 8.1i

RAMB16_S1_S1 #(
   .INIT_A(1'b0),
   .INIT_B(1'b0),
   .SRVAL_A(1'b0),
   .SRVAL_B(1'b0),
   .WRITE_MODE_A("WRITE_FIRST"),
   .WRITE_MODE_B("WRITE_FIRST"),
   .SIM_COLLISION_CHECK("ALL"),
   .INIT_00(256'h0 ... 0),
   ...
   .INIT_3F(256'h0 ... 0)
) RAMB16_S1_S1_inst (
   .DOA(DOA),       // Port A 1-bit Data Output
   .DOB(DOB),       // Port B 1-bit Data Output
   .ADDRA(ADDRA),   // Port A 14-bit Address Input
   .ADDRB(ADDRB),   // Port B 14-bit Address Input
   .CLKA(CLKA),     // Port A Clock
   .CLKB(CLKB),     // Port B Clock
   .DIA(DIA),       // Port A 1-bit Data Input
   .DIB(DIB),       // Port B 1-bit Data Input
   .ENA(ENA),       // Port A RAM Enable Input
   .ENB(ENB),       // PortB RAM Enable Input
   .SSRA(SSRA),     // Port A Synchronous Set/Reset Input
   .SSRB(SSRB),     // Port B Synchronous Set/Reset Input
   .WEA(WEA),       // Port A Write Enable Input
   .WEB(WEB)        // Port B Write Enable Input
);
```

Although the code is readily available, we must study the manual carefully to find the right component and proper configuration parameters.

12.3.2 Memory module via Core Generator

To simplify the instantiation process, Xilinx provides a utility program, known as Core Generator (Coregen), to generate Xilinx-specific components. This utility can be invoked from the ISE environment by selecting Project ≻ New Source. After the New Source Wizard dialog appears, we select IP (Coregen & Architecture Wizard) to invoke the Coregen

program. The program guides the users through a series of questions and then generates several files. The file with the .xco extension is a text file that contains the information necessary to construct the desired memory component. The file with the .v extension contains the "wrapper" code for simulation purpose. This file cannot be used to instantiate the desired component and is ignored during the synthesis process.

Although using the Coregen program is more convenient than direct HDL instantiation, it is not within the HDL framework and can lead to a compatibility problem when a design is not done in the Xilinx ISE environment.

12.3.3 Memory module via HDL inference

Although it is not possible to develop a device-independent HDL description, the synthesis program of ISE, known as XST, provides a collection of behavioral HDL templates to infer memory modules from Xilinx FPGA devices. These templates are done by behavioral descriptions and contain no device-specific component instantiation. They are easy to understand and can be simulated without an additional HDL library. However, while the description does not explicitly refer to any Xilinx component, the code may not be recognized by other third-party synthesis software, and the desired memory module cannot always be inferred. Thus, these templates can best be described as "semi-portable" and "semi-device-independent" behavioral descriptions. Templates for commonly used memory modules are discussed in Section 12.4.

On the downside, the template approach is based on the ability of the XST software to recognize the template and infer the proper memory module accordingly. The software may change during upgrade or misinterpret some code. It is a good idea to check the XST synthesis report to ensure that the desired memory module is inferred correctly.

12.4 HDL TEMPLATES FOR MEMORY INFERENCE

To use behavioral HDL description to infer the Xilinx memory module, the XST's templates should be followed closely. To avoid misinterpretation, we should refrain from creating our own "innovative" code. The codes in the following subsections are all based on templates of the *XST v8.1i Manual*. They are the same as the original templates except that the Verilog-2001 style of port declaration is used and parameters are added for the width of address bits and the width of data bits. It is a good practice to confine the memory description in a separate HDL module so that the module can easily be identified and replaced when needed. In this section, we discuss the behavioral HDL templates for six configurations, including two for single-port RAMs, two for dual-port RAMs, and two for ROMs.

12.4.1 Single-port RAM

The embedded memory of a Spartan-3 device is already wrapped with a synchronous interface similar to that in Section 11.3. Its write operation is always synchronous. At the rising edge of the clock, the address, input data, and relevant control signals, such as we (i.e., write enable), are sampled. If we is asserted, a write operation is performed (i.e., the input data is stored into the memory location designated by the address signal).

The read operation can be asynchronous or synchronous. For *asynchronous read*, the address signal is used directly to access the RAM array. After the address signal changes, the data becomes available after a short delay. For *synchronous read*, the address signal is

sampled at the rising edge of the clock and stored in a register. The registered address is then used to access the RAM array. Because of the register, the availability of data is delayed and is synchronized by the clock signal. Due to the internal structure, an asynchronous read operation can be realized only by the distributed RAM.

Single-port RAM with asynchronous read The template for the single-port RAM with asynchronous read is shown in Listing 12.1. It is modified after the `rams_04` module of the *XST Manual*.

Listing 12.1 Template for a single-port RAM with asynchronous read

```
// Single-port RAM with asynchronous read
// Modified from XST 8.1i v_rams_04

module xilinx_one_port_ram_async
  #(
    parameter ADDR_WIDTH = 8,
              DATA_WIDTH = 1
  )
  (
   input wire clk,
   input wire we,
   input wire [ADDR_WIDTH-1:0] addr,
   input wire [DATA_WIDTH-1:0] din,
   output wire [DATA_WIDTH-1:0] dout
  );

  // signal declaration
  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];

  // body
  always @(posedge clk)
     if (we)  // write operation
        ram[addr] <= din;
  // read operation
  assign dout = ram[addr];

endmodule
```

The code is very similar to the register file discussed in Section 4.2.3 except that the read and write operations use the same address. It contains a two-dimensional array data type for storage and uses dynamic indexing to access the element in the array. The code shows that the write operation is controlled by the clock signal and the read operation depends only on the address. Since an asynchronous read can be realized only by the distributed RAM, this configuration is recommended only for applications that require small storage.

Single-port RAM with synchronous read The template for the single-port RAM with synchronous read is shown in Listing 12.2. It is modified after the `rams_07` module of the *XST Manual*.

Listing 12.2 Template for a single-port RAM with synchronous read

```
// Single-port RAM with synchronous read
// Modified from XST 8.1i v_rams_07
```

```
module xilinx_one_port_ram_sync
  #(
    parameter ADDR_WIDTH = 12,
              DATA_WIDTH = 8
  )
  (
   input wire clk,
   input wire we,
   input wire [ADDR_WIDTH-1:0] addr,
   input wire [DATA_WIDTH-1:0] din,
   output wire [DATA_WIDTH-1:0] dout
  );

  // signal declaration
  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];
  reg [ADDR_WIDTH-1:0] addr_reg;

  // body
  always @(posedge clk)
  begin
     if (we)  // write operation
       ram[addr] <= din;
     addr_reg <= addr;
  end
  // read operation
  assign dout = ram[addr_reg];

endmodule
```

Note that the addr signal is now sampled and stored to the addr_reg register at the rising edge of the clock, and the memory array (the ram signal) is accessed via the addr_reg signal. The data is available only after the addr_reg is updated and thus implicitly synchronized to the clk signal.

Synthesis report During synthesis, a proper RAM module should be inferred from the code template. We can check the synthesis report to confirm the inference of the RAM module. For example, consider the instantiation of a 4K-by-8 RAM (2^{12}-by-2^3) with synchronous read:

```
xilinx_one_port_ram_sync
   #(.ADDR_WIDTH(12), .DATA_WIDTH(8)) ram_unit_4k_by_8
   (.clk(clk), .we(we), .addr(addr), .din(din), .dout(dout));
```

The inference of RAM should be indicated in the HDL Synthesis section of the synthesis report:

```
=========================================================
*                    HDL Synthesis                      *
=========================================================
. . .
Found 4096x8-bit single-port block RAM for signal <ram>.
---------------------------------------------------------
```

```
| mode           | write-first                    |       |
| aspect ratio   | 4096-word x 8-bit              |       |
| clock          | connected to signal <clk>      | rise  |
| write enable   | connected to signal <we>       | high  |
| address        | connected to signal <addr>     |       |
| data in        | connected to signal <din>      |       |
| data out       | connected to signal <dout>     |       |
| ram_style      | Auto                           |       |
-----------------------------------------------------------
Summary:
inferred   1 RAM(s).
```

The number of block RAMs used should be reported in the `Final Report` section of the synthesis report:

```
Device utilization summary:
Selected Device : 3s200ft256-5
 . . .
 Number of BRAMs:    2  out of     12    16%
 . . .
```

As we expected, a 4K-by-8 single-port block RAM is inferred and two block RAMs are used to realize the circuit.

12.4.2 Dual-port RAM

A dual-port RAM includes a second port for memory access. Ideally, the second port should be able to conduct read or write operation independently and have its own set of address, data input and output, and control signals. To be compatible with older versions of XST, we consider a configuration with the second port that can conduct a read operation only. In this book, the main application of the dual-port configuration is for video memory, which requires one write port and one read port. Thus, this configuration does not impose a serious limitation for our purposes. As in a single-port RAM, the read operation of a dual-port RAM can be asynchronous or synchronous.

Dual-port RAM with asynchronous read The template for the dual-port RAM with asynchronous read is shown in Listing 12.3. It is modified after the `rams_09` module of the *XST Manual*.

Listing 12.3 Template for a dual-port RAM with asynchronous read

```
// Dual-port RAM with asynchronous read
// Modified from XST 8.1i v_rams_09

module xilinx_dual_port_ram_async
    #(
      parameter ADDR_WIDTH = 6,
                DATA_WIDTH = 8
    )
    (
```

```
    input wire clk,
    input wire we,
    input wire [ADDR_WIDTH-1:0] addr_a, addr_b,
    input wire [DATA_WIDTH-1:0] din_a,
    output wire [DATA_WIDTH-1:0] dout_a, dout_b
  );

  // signal declaration
  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];

  // body
  always @(posedge clk)
     if (we)  // write operation
        ram[addr_a] <= din_a;
  // two read operations
  assign dout_a = ram[addr_a];
  assign dout_b = ram[addr_b];

endmodule
```

The write operation is similar to that of the single-port RAM, but the code includes a second output port, `dout_b`, which retrieves data from the second address, `addr_b`. As in a single-port RAM with asynchronous read, the dual-port version can be realized only by distributed RAM, and thus its size is limited. Note that if we ignore the `dout_a` port, it is the same as the single-read-port register file of Listing 4.6.

Dual-port RAM with synchronous read The template for the dual-port RAM with synchronous read is shown in Listing 12.4. It is modified after the `rams_11` module of the *XST Manual*.

Listing 12.4 Template for a dual-port RAM with synchronous read

```
// Dual-port RAM with synchronous read
// Modified from XST 8.1i v_rams_11

module xilinx_dual_port_ram_sync
  #(
    parameter ADDR_WIDTH = 6,
              DATA_WIDTH = 8
  )
  (
    input wire clk,
    input wire we,
    input wire [ADDR_WIDTH-1:0] addr_a, addr_b,
    input wire [DATA_WIDTH-1:0] din_a,
    output wire [DATA_WIDTH-1:0] dout_a, dout_b
  );

  // signal declaration
  reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0];
  reg [ADDR_WIDTH-1:0] addr_a_reg, addr_b_reg;

  // body
  always @(posedge clk)
```

```
   begin
      if (we)  // write operation
        ram[addr_a] <= din_a;
      addr_a_reg <= addr_a;
      addr_b_reg <= addr_b;
   end
   // two read operations
   assign dout_a = ram[addr_a_reg];
   assign dout_b = ram[addr_b_reg];

endmodule
```

The code is similar to Listing 12.3 except that the two addresses are first stored in two registers and the registered outputs are used to access memory.

12.4.3 ROM

Despite its name, a ROM (read-only memory) is a combinational circuit and has no internal state. Its output depends only on its input (i.e., address). There is no real embedded ROM in a Spartan-3 device, but it can be emulated by a combinational circuit or a single-port RAM with the write operation disabled. The content of the ROM can be expressed as a case statement in the HDL code and the values are loaded to the RAM when the device is programmed. Since the ROM is based in a RAM, the read operation can be asynchronous or synchronous.

ROM with asynchronous read A real ROM is a combinational circuit and thus should not have a buffer or a clock signal. To be consistent with the terms used in this section, we call it a *ROM with asynchronous read.* This type of ROM can be described by a single case statement and the template is shown by an example in Listing 12.5. The code simply renames the input and output ports of the hex-to-seven-segment LED decoder in Listing 3.14. The address of the ROM functions as the selection expression of the case statement and the corresponding content is assigned to the `data` signal.

Listing 12.5 Template for a ROM with asynchronous read

```
module rom_template
   (
    input wire [3:0] addr,
    output reg [7:0] data
   );

   // body
   always @*
      case (addr)
         4'h0: data = 7'b0000001;
         4'h1: data = 7'b1001111;
         4'h2: data = 7'b0010010;
         4'h3: data = 7'b0000110;
         4'h4: data = 7'b1001100;
         4'h5: data = 7'b0100100;
         4'h6: data = 7'b0100000;
         4'h7: data = 7'b0001111;
         4'h8: data = 7'b0000000;
```

```
            4'h9: data = 7'b0000100;
            4'ha: data = 7'b0001000;
            4'hb: data = 7'b1100000;
            4'hc: data = 7'b0110001;
            4'hd: data = 7'b1000010;
            4'he: data = 7'b0110000;
            4'hf: data = 7'b0111000;
         endcase

   endmodule
```

Since there is no address or data buffer in this circuit, the ROM cannot be realized by a block RAM. It is actually synthesized as a combinational circuit with the logic cells and thus this type of ROM is feasible only for a small table.

ROM with synchronous read For a large table, it is better to utilize a block RAM to realize the ROM. Since the read operation of a block RAM is controlled and synchronized by a clock signal, the ROM requires a clock signal as well. The template for the ROM with synchronous read is shown in Listing 12.6. It is modified after the `rams_21c` module of the *XST Manual*, and the hex-to-seven-segment LED decoder is used for demonstration.

Listing 12.6 Template for a ROM with synchronous read

```
module xilinx_rom_sync_template
   (
    input wire clk,
    input wire [3:0] addr,
    output reg [7:0] data
   );

   // signal declaration
   reg [3:0] addr_reg;

   // body
   always @(posedge clk)
      addr_reg <= addr;

   always @*
      case (addr_reg)
         4'h0: data = 7'b0000001;
         4'h1: data = 7'b1001111;
         4'h2: data = 7'b0010010;
         4'h3: data = 7'b0000110;
         4'h4: data = 7'b1001100;
         4'h5: data = 7'b0100100;
         4'h6: data = 7'b0100000;
         4'h7: data = 7'b0001111;
         4'h8: data = 7'b0000000;
         4'h9: data = 7'b0000100;
         4'ha: data = 7'b0001000;
         4'hb: data = 7'b1100000;
         4'hc: data = 7'b0110001;
         4'hd: data = 7'b1000010;
         4'he: data = 7'b0110000;
```

```
            4'hf: data = 7'b0111000;
        endcase

35  endmodule
```

The code is similar to that of the single-port RAM with synchronous read but with an additional case statement. Note that operation of this ROM depends on the clock signal, and its timing is different from that of a normal ROM. Artificial inclusion of the clock signal is necessary to infer a block RAM for the ROM implementation. During synthesis, the software automatically determines whether to use regular logic cells or block RAMs to realize this circuit.

12.5 BIBLIOGRAPHIC NOTES

Two Xilinx application notes, *XAPP464 Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs* and *XAPP463 Using Block RAM in Spartan-3 Generation FPGAs*, provide detailed information on the distributed RAM and block RAM. Chapter 2 of the *XST User Guide v8.1i*, titled *HDL Coding Techniques*, includes about two dozen HDL code templates to infer various memory configurations.

The comprehensive ISE tutorial, *ISE In-Depth Tutorial*, includes a section on the Core Generator program. Although the program is simple, we need to know the module's basic functionalities and its relevant parameters to create a proper instance.

12.6 SUGGESTED EXPERIMENTS

12.6.1 Block-RAM-based FIFO

In Section 4.5.3, we design a FIFO buffer that uses a register file for storage. To increase its capacity, we can replace the register file with a block RAM-based dual-port RAM module. Derive the HDL code for the new design. Synthesize the verification circuit discussed in Section 4.5.3 with the new FIFO buffer and verify its operation. Note that due to the synchronous read, the behavior of the new FIFO is not completely identical to that of the original FIFO.

12.6.2 Block-RAM-based stack

We discuss the function of a stack in Experiment 4.7.7. To increase its capacity, we can replace the register file with a block RAM-based dual-port RAM module. Repeat the experiment.

12.6.3 ROM-based sign-magnitude adder

We can implement any n-input, m-output function with a 2^n-by-m ROM. Consider the sign-magnitude adder discussed in Section 3.9.2 and assume that `a` and `b` are 4-bit input signals. Design this circuit as follows:

1. Write a program in a conventional programming language, such as C or Java, to generate a case statement that incorporates the 2^8-by-4 truth table of this circuit.
2. Follow the ROM template in Listing 12.5 to derive the HDL code.

3. Synthesize the circuit and verify its operation.
4. Check the synthesis report and compare the sizes (in terms of the number of logic cells) of the original implementation and the ROM-based implementation.
5. Expand a and b to 8-bit input signals and repeat steps 1 to 4.

12.6.4 ROM-based $\sin(x)$ function

One way to implement a sinusoidal function, $\sin(x)$, is to use a look-up table. Assume that the desired implementation requires 10-bit input resolution [i.e., there are 1024 (2^{10}) points between the input range of 0 and 2π] and 8-bit output resolution [i.e., there are 256 (2^8) points between the output range of -1 and $+1$]. Let the input and output be the 10-bit x signal and the 8-bit y signal. The relationship between x and y is

$$\frac{y}{2^7} = \sin\left(2\pi\frac{x}{2^{10}}\right)$$

Because of the symmetry of the sin function, we only need to construct a 2^8-by-7 table for the first quadrant (i.e., between 0 and $\frac{\pi}{2}$) and use simple pre- and postprocessing circuits to obtain the values in other quadrants. Design this circuit as follows:

1. Write a program in a conventional programming language to generate a case statement that incorporates the 2^8-by-7 look-up table for the first quadrant.
2. Follow the ROM template in Listing 12.6 to derive the HDL code for the look-up table.
3. Derive a testbench to generate the sinusoidal output for three complete periods. This can be done by using a 10-bit counter to generate the 10-bit ROM address for $3 * 2^{10}$ clock cycles. In ModelSim, we can display the y signal in Analog format to emulate the effect of a digital-to-analog converter.

12.6.5 ROM-based $\sin(x)$ and $\cos(x)$ functions

In many communication modulation schemes, the $\sin(x)$ and $\cos(x)$ functions are needed at the same time. Assume that the format of the input and output is similar to that in Experiment 12.6.4. The new circuit has two outputs, y_s and y_c:

$$\frac{y_s}{2^7} = \sin\left(2\pi\frac{x}{2^{10}}\right)$$
$$\frac{y_c}{2^7} = \cos\left(2\pi\frac{x}{2^{10}}\right)$$

Although we can follow the previous procedure and create a new ROM for the $\cos(x)$ function, a better alternative is to share the same ROM for both $\sin(x)$ and $\cos(x)$ functions. This is based on the observations that $\cos(x)$ is only a phase shift of $\sin(x)$ and that the FPGA's block RAM can provide dual-port access.

Note that this circuit requires essentially a "dual-port ROM." No HDL behaviorial template is given for this type of memory. We need to experiment with HDL codes and to check the synthesis report to ensure that only one block RAM is inferred. It may be necessary to use the Core Generator program or direct HDL component instantiation to achieve this goal.

Construct this special ROM and derive the HDL code for the pre- and postprocessing circuits. Use a testbench similar to that in Experiment 12.6.4 to verify the circuit's operation.

CHAPTER 13

VGA CONTROLLER I: GRAPHIC

13.1 INTRODUCTION

VGA (video graphics array) is a video display standard introduced in the late 1980s in IBM PCs and is widely supported by PC graphics hardware and monitors. We discuss the design of a basic eight-color 640-by-480 resolution interface for CRT (cathode ray tube) monitors in this book. CRT synchronization and basic graphic processing are examined in this chapter, and text generation is discussed in Chapter 14.

13.1.1 Basic operation of a CRT

The conceptual sketch of a monochrome CRT monitor is shown in Figure 13.1. The electron gun (cathode) generates a focused electron beam, which traverses a vacuum tube and eventually hits the phosphorescent screen. Light is emitted at the instant that electrons hit a phosphor dot on the screen. The intensity of the electron beam and the brightness of the dot are determined by the voltage level of the external video input signal, labeled `mono` in Figure 13.1. The `mono` signal is an analog signal whose voltage level is between 0 and 0.7 V.

A vertical deflection coil and a horizontal deflection coil outside the tube produce magnetic fields to control how the electron beam travels and to determine where on the screen the electrons hit. In today's monitors, the electron beam traverses (i.e., scans) the screen systematically in a fixed pattern, from left to right and from top to bottom, as shown in Figure 13.2.

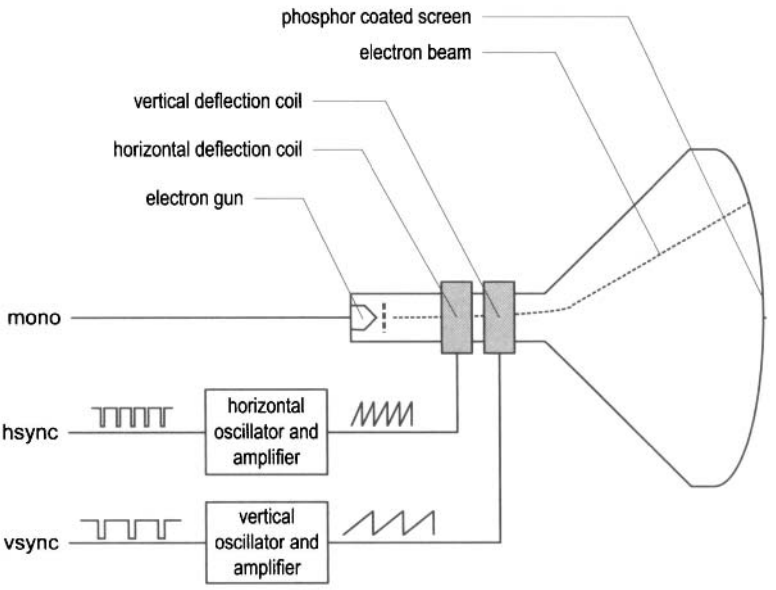


Figure 13.1 Conceptual diagram of a CRT monitor.

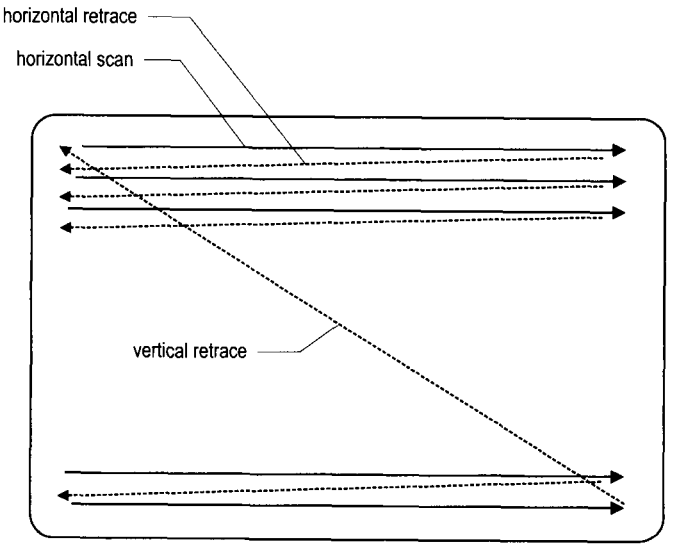


Figure 13.2 CRT scanning pattern.

Table 13.1 Three-bit VGA color combinations

Red (R)	Green (G)	Blue (B)	Resulting color
0	0	0	black
0	0	1	blue
0	1	0	green
0	1	1	cyan
1	0	0	red
1	0	1	magenta
1	1	0	yellow
1	1	1	white

The monitor's internal oscillators and amplifiers generate sawtooth waveforms to control the two deflection coils. For example, the electron beam moves from the left edge to the right edge as the voltage applied to the horizontal deflection coil gradually increases. After reaching the right edge, the beam returns rapidly to the left edge (i.e., *retraces*) when the voltage changes to 0. The relationship between the sawtooth waveform and the scan is shown in Figure 13.4. Two external synchronization signals, `hsync` and `vsync`, control generation of the sawtooth waveforms. These signals are digital signals. The relationship between the `hsync` signal and the horizontal sawtooth is also shown in Figure 13.4. Note that the "1" and "0" periods of the `hsync` signal correspond to the rising and falling ramps of the sawtooth waveform.

The basic operation of a color CRT is similar except that it has three electron beams, which are projected to the red, green, and blue phosphor dots on the screen. The three dots are combined to form a pixel. We can adjust the voltage levels of the three video input signals to obtain the desired pixel color.

13.1.2 VGA port of the S3 board

The VGA port has five active signals, including the horizontal and vertical synchronization signals, `hsync` and `vsync`, and three video signals for the red, green, and blue beams. It is physically connected to a 15-pin D-subminiature connector. A video signal is an analog signal and the video controller uses a digital-to-analog converter to convert the digital output to the desired analog level. If a video signal is represented by an N-bit word, it can be converted to 2^N analog levels. The three video signals can generate 2^{3N} different colors. This is also known as *3N-bit color* since a color is defined by $3N$ bits. In the S3 board, a 1-bit word is used for each video signal, and this leads to only eight (i.e., 2^3) possible colors. The possible color combinations are shown in Table 13.1. If we use the same 1-bit signal to drive the video signals, they become either "000" or "111" and the monitor functions as a black-and-white monochrome monitor.

13.1.3 Video controller

A video controller generates the synchronization signals and outputs data pixels serially. A simplified block diagram of a VGA controller is shown in Figure 13.3. It contains a synchronization circuit, labeled `vga_sync`, and a pixel generation circuit.

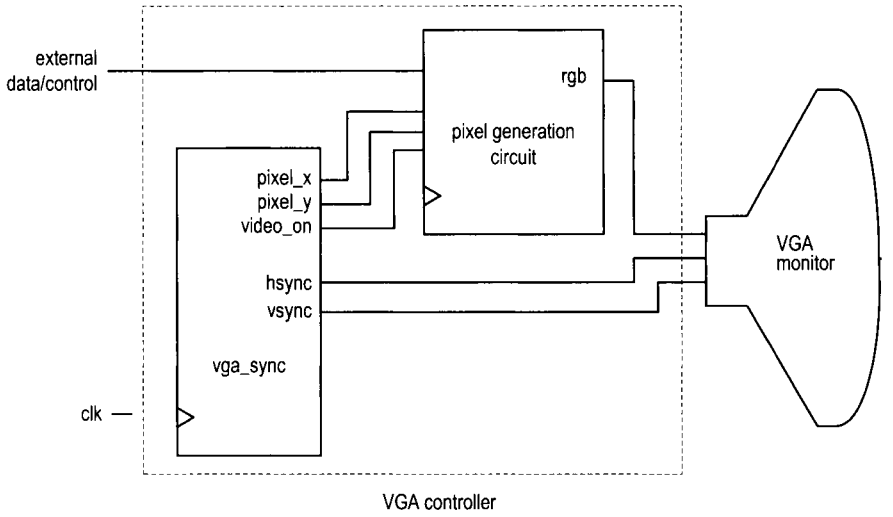


Figure 13.3 Simplified block diagram of a VGA controller.

The vga_sync circuit generates timing and synchronization signals. The hsync and vsync signals are connected to the VGA port to control the horizontal and vertical scans of the monitor. The two signals are decoded from the internal counters, whose outputs are the pixel_x and pixel_y signals. The pixel_x and pixel_y signals indicate the relative positions of the scans and essentially specify the location of the current pixel. The vga_sync circuit also generates the video_on signal to indicate whether to enable or disable the display. The design of this circuit is discussed in Section 13.2.

The pixel generation circuit generates the three video signals, which are collectively referred to as the rgb signal. A color value is obtained according to the current coordinates of the pixel (the pixel_x and pixel_y signals) and the external control and data signals. This circuit is more involved and is discussed in the second half of this chapter and Chapter 14.

13.2 VGA SYNCHRONIZATION

The video synchronization circuit generates the hsync signal, which specifies the required time to traverse (scan) a row, and the vsync signal, which specifies the required time to traverse (scan) the entire screen. Subsequent discussions are based on a 640-by-480 VGA screen with a 25-MHz *pixel rate*, which means that 25M pixels are processed in a second. Note that this resolution is also know as the *VGA mode*.

The screen of a CRT monitor usually includes a small black border, as shown at the top of Figure 13.4. The middle rectangle is the visible portion. Note that the coordinate of the vertical axis increases downward. The coordinates of the top-left and bottom-right corners are (0,0) and (639,479), respectively.

13.2.1 Horizontal synchronization

A detailed timing diagram of one horizontal scan is shown in Figure 13.4. A period of the hsync signal contains 800 pixels and can be divided into four regions:

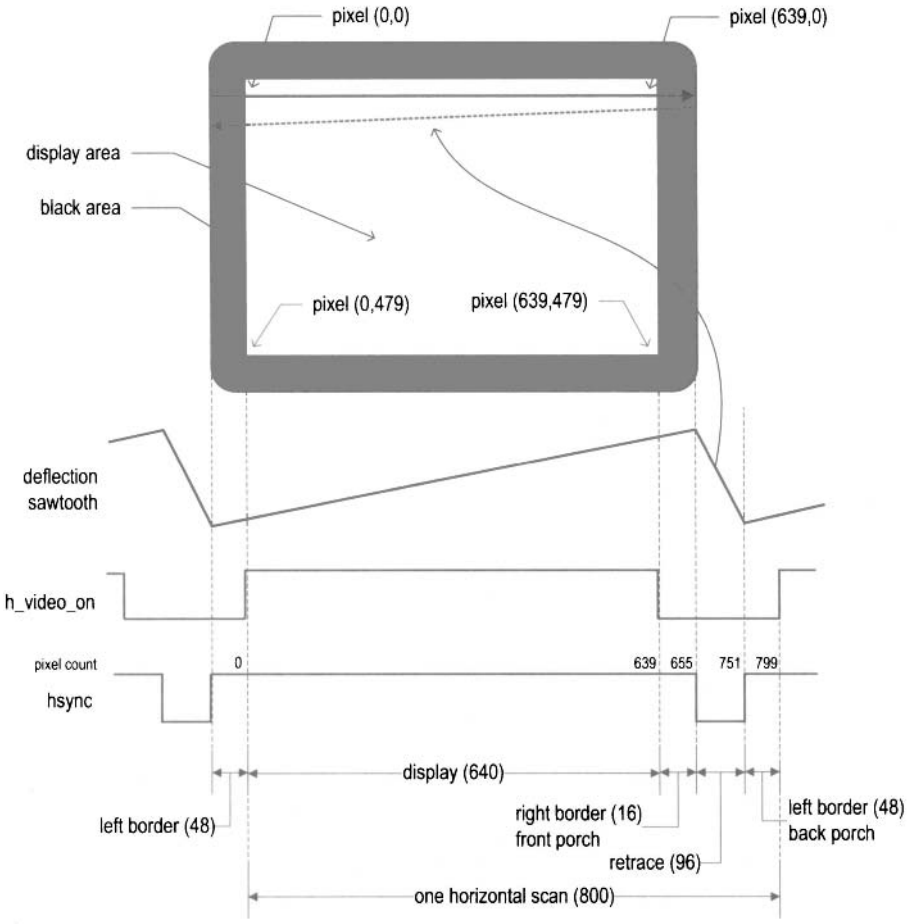


Figure 13.4 Timing diagram of a horizontal scan.

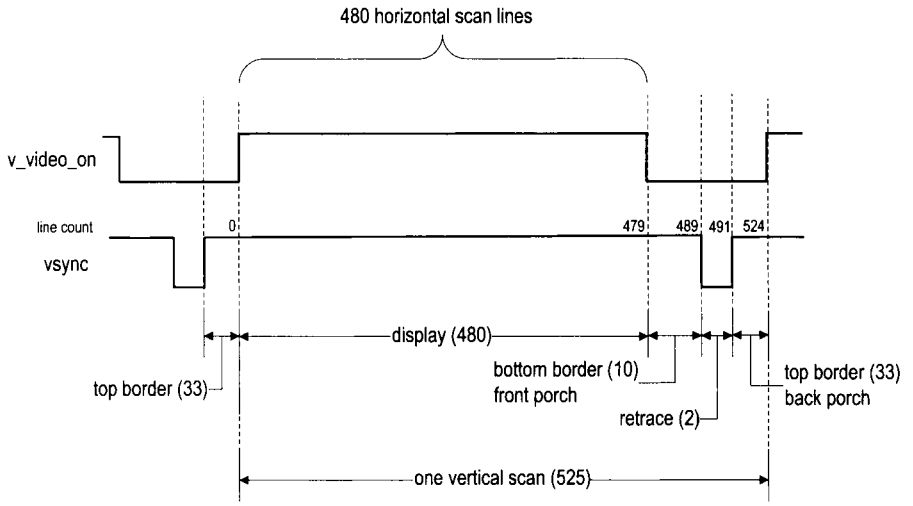


Figure 13.5 Timing diagram of a vertical scan.

- *Display*: region where the pixels are actually displayed on the screen. The length of this region is 640 pixels.
- *Retrace*: region in which the electron beams return to the left edge. The video signal should be disabled (i.e., black), and the length of this region is 96 pixels.
- *Right border*: region that forms the right border of the display region. It is also know as the *front porch* (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 16 pixels.
- *Left border*: region that forms the left border of the display region. It is also know as the *back porch* (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 48 pixels.

Note that the lengths of the right and left borders may vary for different brands of monitors.

The `hsync` signal can be obtained by a special mod-800 counter and a decoding circuit. The counts are marked on the top of the `hsync` signal in Figure 13.4. We intentionally start the counting from the beginning of the display region. This allows us to use the counter output as the horizontal (x-axis) coordinate. This output constitutes the `pixel_x` signal. The `hsync` signal goes low when the counter's output is between 656 and 751.

Note that the CRT monitor should be black in the right and left borders and during retrace. We use the `h_video_on` signal to indicate whether the current horizontal coordinate is in the displayable region. It is asserted only when the pixel count is smaller than 640.

13.2.2 Vertical synchronization

During the vertical scan, the electron beams move gradually from top to bottom and then return to the top. This corresponds to the time required to refresh the entire screen. The format of the `vsync` signal is similar to that of the `hsync` signal, as shown in Figure 13.5. The time unit of the movement is represented in terms of horizontal scan lines. A period of the `vsync` signal is 525 lines and can be divided into four regions:

- *Display*: region where the horizontal lines are actually displayed on the screen. The length of this region is 480 lines.

- *Retrace*: region that the electron beams return to the top of the screen. The video signal should be disabled, and the length of this region is 2 lines.
- *Bottom border*: region that forms the bottom border of the display region. It is also know as the *front porch* (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 10 lines.
- *Top border*: region that forms the top border of the display region. It is also know as the *back porch* (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 33 lines.

As in the horizontal scan, the lengths of the top and bottom borders may vary for different brands of monitors.

The `vsync` signal can be obtained by a special mod-525 counter and a decoding circuit. Again, we intentionally start counting from the beginning of the display region. This allows us to use the counter output as the vertical (y-axis) coordinate. This output constitutes the `pixel_y` signal. The `vsync` signal goes low when the line count is 490 or 491.

As in the horizontal scan, we use the `v_video_on` signal to indicate whether the current vertical coordinate is in the displayable region. It is asserted only when the line count is smaller than 480.

13.2.3 Timing calculation of VGA synchronization signals

As mentioned earlier, we assume that the pixel rate is 25 MHz. It is determined by three parameters:

- p: the number of pixels in a horizontal scan line. For 640-by-480 resolution, it is

$$p = 800 \frac{pixels}{line}$$

- l: the number of lines in a screen (i.e., a vertical scan). For 640-by-480 resolution, it is

$$l = 525 \frac{lines}{screen}$$

- s: the number of screens per second. For flickering-free operation, we can set it to

$$s = 60 \frac{screens}{second}$$

The s parameter specifies how fast the screen should be refreshed. For a human eye, the refresh rate must be at least 30 screens per second to make the motion appear to be continuous. To reduce flickering, the monitor usually has a much higher rate, such as the 60 screens per second specification above. The pixel rate can be calculated by the three parameters:

$$\text{pixel rate} = p * l * s \approx 25M \frac{pixels}{second}$$

The pixel rate for other resolutions and refresh rates can be calculated in a similar fashion. Clearly, the rate increases as the resolution and refresh rate grow.

13.2.4 HDL implementation

The function of the `vga_sync` circuit is discussed in Section 13.1.3. If the frequency of the system clock is 25 MHz, the circuit can be implemented by two special counters: a

mod-800 counter to keep track of the horizontal scan and a mod-525 counter to keep track of the vertical scan.

Since our designs generally use the 50-MHz oscillator of the prototyping board, the system clock rate is twice the pixel rate. Instead of creating a separate 25-MHz clock domain and violating the synchronous design methodology, we can generate a 25-MHz enable tick to enable or pause the counting. The tick is also routed to the p_tick port as an output signal to coordinate operation of the pixel generation circuit.

The HDL code is shown in Listing 13.1. It consists of a mod-2 counter to generate the 25-MHz enable tick and two counters for the horizontal and vertical scans. We use two status signals, h_end and v_end, to indicate completion of the horizontal and vertical scans. The values of various regions of the horizontal and vertical scans are defined as constants. They can easily be modified if a different resolution or refresh rate is used. To remove potential glitches, output buffers are inserted for the hsync and vsync signals. This leads to a one-clock-cycle delay. We should add a similar buffer for the rgb signal in the pixel generation circuit to compensate for the delay.

Listing 13.1 VGA synchronization circuit

```
module vga_sync
   (
    input wire clk, reset,
    output wire hsync, vsync, video_on, p_tick,
    output wire [9:0] pixel_x, pixel_y
   );

   // constant declaration
   // VGA 640-by-480 sync parameters
   localparam HD = 640; // horizontal display area
   localparam HF = 48 ; // h. front (left) border
   localparam HB = 16 ; // h. back (right) border
   localparam HR = 96 ; // h. retrace
   localparam VD = 480; // vertical display area
   localparam VF = 10;  // v. front (top) border
   localparam VB = 33;  // v. back (bottom) border
   localparam VR = 2;   // v. retrace

   // mod-2 counter
   reg mod2_reg;
   wire mod2_next;
   // sync counters
   reg [9:0] h_count_reg, h_count_next;
   reg [9:0] v_count_reg, v_count_next;
   // output buffer
   reg v_sync_reg, h_sync_reg;
   wire v_sync_next, h_sync_next;
   // status signal
   wire h_end, v_end, pixel_tick;

   // body
   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
```

```
            mod2_reg <= 1'b0;
            v_count_reg <= 0;
            h_count_reg <= 0;
            v_sync_reg <= 1'b0;
            h_sync_reg <= 1'b0;
         end
      else
         begin
            mod2_reg <= mod2_next;
            v_count_reg <= v_count_next;
            h_count_reg <= h_count_next;
            v_sync_reg <= v_sync_next;
            h_sync_reg <= h_sync_next;
         end

   // mod-2 circuit to generate 25 MHz enable tick
   assign mod2_next = ~mod2_reg;
   assign pixel_tick = mod2_reg;

   // status signals
   // end of horizontal counter (799)
   assign h_end = (h_count_reg==(HD+HF+HB+HR-1));
   // end of vertical counter (524)
   assign v_end = (v_count_reg==(VD+VF+VB+VR-1));

   // next-state logic of mod-800 horizontal sync counter
   always @*
      if (pixel_tick)  // 25 MHz pulse
         if (h_end)
            h_count_next = 0;
         else
            h_count_next = h_count_reg + 1;
      else
         h_count_next = h_count_reg;

   // next-state logic of mod-525 vertical sync counter
   always @*
      if (pixel_tick & h_end)
         if (v_end)
            v_count_next = 0;
         else
            v_count_next = v_count_reg + 1;
      else
         v_count_next = v_count_reg;

   // horizontal and vertical sync, buffered to avoid glitch
   // h_sync_next asserted between 656 and 751
   assign h_sync_next = (h_count_reg>=(HD+HB) &&
                         h_count_reg<=(HD+HB+HR-1));
   // vh_sync_next asserted between 490 and 491
   assign v_sync_next = (v_count_reg>=(VD+VB) &&
                         v_count_reg<=(VD+VB+VR-1));
```

```
   // video on/off
90 assign video_on = (h_count_reg<HD) && (v_count_reg<VD);

   // output
   assign hsync = h_sync_reg;
   assign vsync = v_sync_reg;
95 assign pixel_x = h_count_reg;
   assign pixel_y = v_count_reg;
   assign p_tick = pixel_tick;

endmodule
```

13.2.5 Testing circuit

To verify operation of the synchronization circuit, we can connect the rgb signal to three switches. The entire visible region should be turned on with a single color. We can go through the eight possible combinations and check the colors defined in Table 13.1. The HDL code is shown in Listing 13.2. As mentioned in Section 13.2.4, an output buffer is added for the rgb signal.

Listing 13.2 VGA synchronization testing circuit

```
module vga_test
   (
    input wire clk, reset,
    input wire [2:0] sw,
5   output wire hsync, vsync,
    output wire [2:0] rgb
   );

   //signal declaration
10 reg [2:0] rgb_reg;
   wire video_on;

   // instantiate vga sync circuit
   vga_sync vsync_unit
15    (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(), .pixel_x(), .pixel_y());
   // rgb buffer
   always @(posedge clk, posedge reset)
      if (reset)
20       rgb_reg <= 0;
      else
         rgb_reg <= sw;
   // output
   assign rgb = (video_on) ? rgb_reg : 3'b0;
25
endmodule
```

13.3 OVERVIEW OF THE PIXEL GENERATION CIRCUIT

The pixel generation circuit generates the 3-bit `rgb` signal for the VGA port. The external control and data signals specify the content of the screen, and the `pixel_x` and `pixel_y` signals from the `vga_sync` circuit provide the current coordinates of the pixel. For our discussion purposes, we divided this circuit into three broad categories:

- Bit-mapped scheme
- Tile-mapped scheme
- Object-mapped scheme

In a *bit-mapped scheme*, a *video memory* is used to store the data to be displayed on the screen. Each pixel of the screen is mapped directly to a memory word, and the `pixel_x` and `pixel_y` signals form the address. A graphics processing circuit continuously updates the screen and writes relevant data to the video memory. A retrieval circuit continuously reads the video memory and routes the data to the `rgb` signal. This is the scheme used in today's high-performance video controller. For 640-by-480 resolution, there are about 310k (i.e., 640*480) pixels on a screen. This translates to 310k memory bits for a monochrome display and 930k memory bits (i.e., 3 bits per pixel) for a 3-bit color display. A bit-mapped example is discussed in Section 13.5.

To reduce the memory requirement, one alternative is to use a *tile-mapped scheme*. In this scheme, we group a collection of bits to form a *tile* and treat each tile as a display unit. For example, we can define an 8-by-8 square of pixels (i.e., 64 pixels) as a tile. The 640-by-480 pixel-oriented screen becomes an 80-by-60 tile-oriented screen. Only 4800 (i.e., 80*60) words are needed for the *tile memory*. The number of bits in a word depends on the number of tile patterns. For example, if there are 32 tile patterns, each word should contain 5 bits, and the size of the tile memory is about 24k bits (i.e., 5*4800). The tile-mapped scheme usually requires a ROM to store the tile patterns. We call it *pattern memory*. Assume that monochrome patterns are used in the previous example. Each 8-by-8 tile pattern requires 64 bits, and the entire 32 patterns need 2K (i.e., 8*8*32) bits. The overall memory requirement is about 26k bits, which is much smaller than the 310k bits of the bit-mapped scheme. The text display discussed in Chapter 14 is based on this scheme.

For some applications, the video display can be very simple and contains only a few objects. Instead of wasting memory to store a mostly blank screen, we can generate these objects using simple object generation circuits. We call this approach an *object-mapped scheme*. An object-mapped example is discussed in Section 13.4.

The three schemes can be mixed together to generate a full screen. For example, we can use a bit-mapped scheme to generate the background and use an object-mapped scheme to produce the main objects. We can also use a bit-mapped scheme for one portion of a screen and tile-mapped text for another part of the screen.

13.4 GRAPHIC GENERATION WITH AN OBJECT-MAPPED SCHEME

The conceptual diagram of an object-mapped pixel generation circuit that contains three objects is shown in Figure 13.6. The diagram consists of three object generation circuits and a special selecting and routing circuit, labeled `rgb mux`. An object generation circuit performs the following tasks:

- It keeps the coordinates of the current object and compares them with the current scan location provided by the `pixel_x` and `pixel_y` signals.

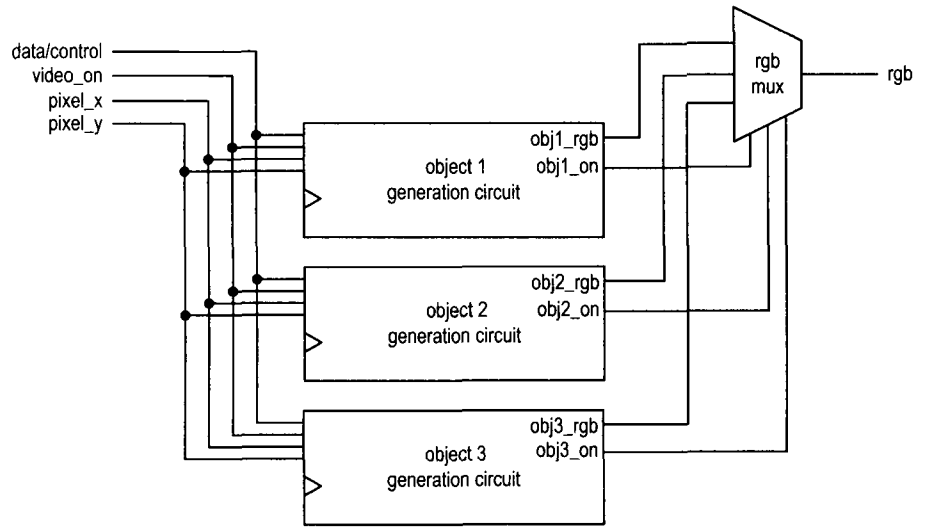


Figure 13.6 Conceptual diagram of object-mapped pixel generation.

- If the current scan location falls within the region, it asserts the `obj_i_on` signal to indicate that the current scan location is within the region of the ith object and the object should be "turned on."
- It specifies the desired color in the `obj_i_rgb` signal.

The `rgb mux` circuit performs multiplexing according to an internal prioritizing scheme. It examines various `obj_i_on` signals and determines which `obj_i_rgb` signal is to be routed to the `rgb` output. The prioritizing scheme prioritizes the order of the displays when multiple `obj_i_on` signals are asserted at the same time. It corresponds to selecting an object for the foreground.

We use a simplified ping-pong-like game to illustrate the various graphic generation schemes. The design is constructed as follows:

1. Create a simple still screen with rectangular objects.
2. Add a round object.
3. Introduce animation.
4. Add text for scores and information.
5. Create a top-level control circuit.

The first three steps are discussed in this section, and the last two steps are discussed in Chapter 14.

13.4.1 Rectangular objects

A rectangular object can be described by its boundary coordinates on the screen. The still screen of the game is shown in Figure 13.7. It has three objects: a wall, which is shown as a narrow stripe on the left; a paddle, which is shown as a short vertical bar on the right; and a square ball. The coordinates of the displayable area of the screen are also shown. Note that the y-axis increases downward.

Let us first examine generation of the wall stripe. For clarity, we define constants for the relevant boundaries and sizes in code. The code segment for the wall is

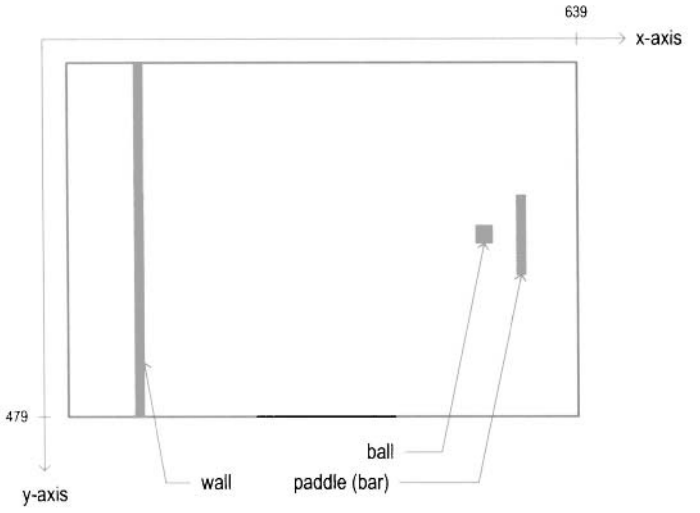


Figure 13.7 Still screen of the pong game.

```
// wall left, right boundary
localparam WALL_X_L = 32;
localparam WALL_X_R = 35;
...
// pixel within wall
assign wall_on = (WALL_X_L<=pix_x) && (pix_x<=WALL_X_R);
// wall rgb output
assign wall_rgb = 3'b001; // blue
```

The wall is a four-pixel-wide vertical stripe between columns 32 and 35, which as defined as `WALL_X_L` and `WALL_X_R`, representing the left and right x-coordinates of the wall, respectively. The object has two output signals, `wall_on` and `wall_rgb`. The `wall_on` signal, which indicates that the wall object should be turned on, is asserted when the current horizontal scan is within its region. Since the stripe covers the entire vertical column, there is no need for the y-axis boundaries. The `wall_rgb` signal indicates that the color of the wall is "001" (blue).

The code segment for the bar (paddle) is

```
// bar left, right boundary
localparam BAR_X_L = 600;
localparam BAR_X_R = 603;
// bar top, bottom boundary
localparam BAR_Y_SIZE = 72;
localparam BAR_Y_T = MAX_Y/2-BAR_Y_SIZE/2; //204
localparam BAR_Y_B = BAR_Y_T+BAR_Y_SIZE-1;
...
// pixel within bar
assign bar_on = (BAR_X_L<=pix_x) && (pix_x<=BAR_X_R) &&
                (BAR_Y_T<=pix_y) && (pix_y<=BAR_Y_B);
// bar rgb output
assign bar_rgb = 3'b010; // green
```

The code is similar to that of the wall segment except that it includes the y-axis boundaries. The desired vertical length of the bar is 72 pixels, which is defined by BAR_Y_SIZE. Since we wish to place the bar in the middle, the top boundary of the bar, which is BAR_Y_T, is one half of the maximal y-value (i.e., 480/2) minus one half of the bar length. The bottom boundary of the bar is the top boundary plus the bar length. Generation of the bar_on signal is similar to that of the wall_on signal except that the vertical scan must be within the bar's y-axis boundaries as well.

The code for the ball can be constructed in a similar fashion. The final code segment is the selection and multiplexing circuit, which examines the on signals of three objects and routes the corresponding rgb signal to output. The code is

```
always @*
   if (~video_on)
      graph_rgb = 3'b000; // blank
   else
      if (wall_on)
         graph_rgb = wall_rgb;
      else if (bar_on)
         graph_rgb = bar_rgb;
      else if (sq_ball_on)
         graph_rgb = ball_rgb;
      else
         graph_rgb = 3'b110; // yellow background
```

The circuit first checks whether the video_on is asserted, and if this is the case, examines the three on signals in turn. When an on signal is asserted, it indicates that the scan is within its region, and the corresponding rgb signal is passed to the output. If no signal is asserted, the scan is in the "background" and the output is assigned to be "110" (yellow).

The complete HDL code is shown in Listing 13.3.

Listing 13.3 Pixel-generation circuit for the pong game screen

```
module pong_graph_st
   (
    input wire video_on,
    input wire [9:0] pix_x, pix_y,
    output reg [2:0] graph_rgb
   );

   // constant and signal declaration
   // x, y coordinates (0,0) to (639,479)
   localparam MAX_X = 640;
   localparam MAX_Y = 480;
   //-----------------------------------------------
   // vertical stripe as a wall
   //-----------------------------------------------
   // wall left, right boundary
   localparam WALL_X_L = 32;
   localparam WALL_X_R = 35;
   //-----------------------------------------------
   // right vertical bar
   //-----------------------------------------------
   // bar left, right boundary
   localparam BAR_X_L = 600;
```

```
   localparam BAR_X_R = 603;
   // bar top, bottom boundary
25 localparam BAR_Y_SIZE = 72;
   localparam BAR_Y_T = MAX_Y/2-BAR_Y_SIZE/2; //204
   localparam BAR_Y_B = BAR_Y_T+BAR_Y_SIZE-1;
   //-----------------------------------------------
   // square ball
30 //-----------------------------------------------
   localparam BALL_SIZE = 8;
   // ball left, right boundary
   localparam BALL_X_L = 580;
   localparam BALL_X_R = BALL_X_L+BALL_SIZE-1;
35 // ball top, bottom boundary
   localparam BALL_Y_T = 238;
   localparam BALL_Y_B = BALL_Y_T+BALL_SIZE-1;
   //-----------------------------------------------
   // object output signals
40 //-----------------------------------------------
   wire wall_on, bar_on, sq_ball_on;
   wire [2:0] wall_rgb, bar_rgb, ball_rgb;

   // body
45 //-----------------------------------------------
   // (wall) left vertical strip
   //-----------------------------------------------
   // pixel within wall
   assign wall_on = (WALL_X_L<=pix_x) && (pix_x<=WALL_X_R);
50 // wall rgb output
   assign wall_rgb = 3'b001; // blue
   //-----------------------------------------------
   // right vertical bar
   //-----------------------------------------------
55 // pixel within bar
   assign bar_on = (BAR_X_L<=pix_x) && (pix_x<=BAR_X_R) &&
                   (BAR_Y_T<=pix_y) && (pix_y<=BAR_Y_B);
   // bar rgb output
   assign bar_rgb = 3'b010; // green
60 //-----------------------------------------------
   // square ball
   //-----------------------------------------------
   // pixel within squared ball
   assign sq_ball_on =
65           (BALL_X_L<=pix_x) && (pix_x<=BALL_X_R) &&
             (BALL_Y_T<=pix_y) && (pix_y<=BALL_Y_B);
   assign ball_rgb = 3'b100;    // red
   //-----------------------------------------------
   // rgb multiplexing circuit
70 //-----------------------------------------------
   always @*
      if (~video_on)
         graph_rgb = 3'b000; // blank
      else
75       if (wall_on)
```

```
            graph_rgb = wall_rgb;
         else if (bar_on)
            graph_rgb = bar_rgb;
         else if (sq_ball_on)
            graph_rgb = ball_rgb;
         else
            graph_rgb = 3'b110; // yellow background

endmodule
```

After deriving the pixel generation circuit, we can combine it with the VGA synchronization circuit to construct the complete video interface. The top-level HDL code is shown in Listing 13.4. Note that the graph_rgb signal is routed to output through an output buffer. It is loaded when the pixel_tick signal is asserted. This synchronizes the rgb output with the buffered hsync and vsync signals.

Listing 13.4 Complete circuit for a still pong game screen

```
module pong_top_st
   (
    input wire clk, reset,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;

   // body
   // instantiate vga sync circuit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));
   // instantiate graphic generator
   pong_graph_st pong_grf_unit
      (.video_on(video_on), .pix_x(pixel_x), .pix_y(pixel_y),
       .graph_rgb(rgb_next));
   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;

endmodule
```

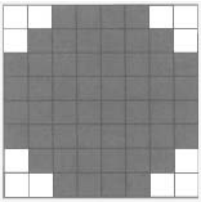


Figure 13.8 Bit map of a circle.

13.4.2 Non-rectangular object

Direct checking of the boundaries of a non-rectangular object is very difficult. An alternative is to specify the object pattern in a bit map and generate the `rgb` and `on` signals according to the map. This can best be explained by an example. Assume that we want to have a round ball in the pong game screen. The bit map of a circle within an 8-by-8 pixel square is shown in Figure 13.8. The circle object can be generated as follows:

- Check whether the scan coordinates are within the 8-by-8 pixel square.
- If this is the case, obtain the corresponding pixel from the bit map.
- Use the retrieved bit to generate the `rgb` and `on` signals for the circle object.

To implement this scheme, we need to include a *pattern ROM* to store the bit map and an address mapping circuit to convert the scan coordinates to the ROM's row and column.

To accommodate the change, the ball portion from Listing 13.3 must be modified. First, we define a pattern ROM for the circle using a case statement, as in the ROM template of Listing 12.5:

```
wire [2:0] rom_addr;
reg [7:0] rom_data;
...
// round ball image ROM
always @*
case (rom_addr)
   3'h0: rom_data = 8'b00111100; //   ****
   3'h1: rom_data = 8'b01111110; //  ******
   3'h2: rom_data = 8'b11111111; // ********
   3'h3: rom_data = 8'b11111111; // ********
   3'h4: rom_data = 8'b11111111; // ********
   3'h5: rom_data = 8'b11111111; // ********
   3'h6: rom_data = 8'b01111110; //  ******
   3'h7: rom_data = 8'b00111100; //   ****
endcase
```

Second, we expand the ball generation segment to include the mapping of the circle bit map. To facilitate future animation, we also use signals to replace constants for the square ball boundaries. The revised code becomes

```
// pixel within ball
assign sq_ball_on =
         (ball_x_l<=pix_x) && (pix_x<=ball_x_r) &&
         (ball_y_t<=pix_y) && (pix_y<=ball_y_b);
// map current pixel location to ROM addr/col
assign rom_addr = pix_y[2:0] - ball_y_t[2:0];
```

```
assign rom_col = pix_x[2:0] - ball_x_l[2:0];
assign rom_bit = rom_data[rom_col];
// pixel within ball
assign rd_ball_on = sq_ball_on & rom_bit;
// ball rgb output
assign ball_rgb = 3'b100;    // red
```

The first statement checks whether the current scan coordinates are within the square ball region and asserts the `sq_ball_on` signal accordingly. This part is the same as Listing 13.3 except that signals are used for boundaries. The second part obtains the corresponding ROM bit according to the current scan coordinates. If the scan coordinates are within the square ball region, subtracting the three LSBs from the top boundary (i.e., `ball_y_t`) provides the corresponding ROM row (i.e., `rom_addr`), and subtracting the three LSBs from the left boundary (i.e., `ball_x_l`) provides the corresponding ROM column (i.e., `rom_col`). The final bit is retrieved by an indexing operation. It is then combined with the `sq_ball_on` signal to generate the `rd_ball_on` signal. This design just assigns a monochrome color (i.e., 100, red) for the round ball region. We can duplicate the pattern ROM three times to store the `rgb` value for each pixel and generate a multiple-color ball.

Finally, we need to make a minor modification in the multiplexing circuit to substitute the `sq_ball_on` signal with the `rd_ball_on` signal:

```
...
else if (rd_ball_on)
   graph_rgb = ball_rgb;
...
```

These modifications are incorporated into the animated graph in the next subsection.

13.4.3 Animated object

When an object changes its location gradually in each scan, it creates the illusion of motion and becomes *animated*. To achieve this, we can use registers to store the boundaries of an object and update its value in each scan. In the pong game, the paddle is controlled by two pushbuttons and can move up and down, and the ball can move and bounce in all directions. We illustrate how to create animation for these two objects in this subsection.

While the VGA controller is driven by a 25-MHz pixel rate, the screen of the VGA monitor is refreshed only 60 times per second. The boundary registers only need to be updated at this rate. We create a 60-Hz enable tick, `refr_tick`, which is asserted one clock cycle every $\frac{1}{60}$ second.

Let us first examine the design of the paddle. To accommodate the changing y-axis coordinates, we replace the constants with two signals, `bar_y_t` and `bar_y_b`, to represent the top and bottom boundaries, and create a register, `bar_y_reg`, to store the current y-axis location of the top boundary. If one of the pushbuttons is pressed, `bar_y_reg` either increases or decreases a fixed amount when the `refr_tick` signal is asserted. The amount is defined by a constant, `BAR_V`, which stands for the bar velocity. We assume that assertion of the `btn[1]` and `btn[0]` signals causes the paddle to move up and down, respectively, and that the paddle stops moving when it reaches the top or the bottom of the screen. The code segment for updating `bar_y_reg` is

```
// new bar y-position
always @*
begin
```

```
      bar_y_next = bar_y_reg; // no move
      if (refr_tick)
         if (btn[1] & (bar_y_b < (MAX_Y-1-BAR_V)))
            bar_y_next = bar_y_reg + BAR_V; // move down
         else if (btn[0] & (bar_y_t > BAR_V))
            bar_y_next = bar_y_reg - BAR_V; // move up
   end
```

The design of the ball is more involved. We have to replace the four boundary constants with four signals and create two registers, `ball_x_reg` and `ball_y_reg`, to store the current x- and y-axis coordinates of the left and top boundaries. The ball usually moves at a constant velocity (i.e., at a constant speed and in the same direction). It may change direction when hitting the wall, the paddle, or the bottom or top of the screen. We decompose the velocity into an x-component and a y-component, whose values can be either a positive constant value, `BALL_V_P`, or a negative constant value, `BALL_V_N`. The current values of the two components are stored in the `x_delta_reg` and `y_delta_reg` registers. The code segment for updating `ball_x_reg` and `ball_y_reg` is

```
// new ball position
assign ball_x_next = (refr_tick) ? ball_x_reg+x_delta_reg :
                     ball_x_reg ;
assign ball_y_next = (refr_tick) ? ball_y_reg+y_delta_reg :
                     ball_y_reg ;
```

and the code segment for updating `x_delta_reg` and `y_delta_reg` is

```
// new ball velocity
always @*
begin
   x_delta_next = x_delta_reg;
   y_delta_next = y_delta_reg;
   if (ball_y_t < 1) // reach top
      y_delta_next = BALL_V_P;
   else if (ball_y_b > (MAX_Y-1)) // reach bottom
      y_delta_next = BALL_V_N;
   else if (ball_x_l <= WALL_X_R) // reach wall
      x_delta_next = BALL_V_P;    // bounce back
   else if ((BAR_X_L<=ball_x_r) && (ball_x_r<=BAR_X_R) &&
            (bar_y_t<=ball_y_b) && (ball_y_t<=bar_y_b))
      // reach x of right bar and hit, ball bounce back
      x_delta_next = BALL_V_N;
end
```

Note that if the paddle bar misses the ball, the ball continues moving to the right and eventually wraps around.

The complete code is shown in Listing 13.5.

Listing 13.5 Pixel-generation circuit for the animated pong game

```
module pong_graph_animate
   (
    input wire clk, reset,
    input wire video_on,
    input wire [1:0] btn,
    input wire [9:0] pix_x, pix_y,
```

```
    output reg [2:0] graph_rgb
   );

   // constant and signal declaration
   // x, y coordinates (0,0) to (639,479)
   localparam MAX_X = 640;
   localparam MAX_Y = 480;
   wire refr_tick;
   //-----------------------------------------------
   // vertical stripe as a wall
   //-----------------------------------------------
   // wall left, right boundary
   localparam WALL_X_L = 32;
   localparam WALL_X_R = 35;
   //-----------------------------------------------
   // right vertical bar
   //-----------------------------------------------
   // bar left, right boundary
   localparam BAR_X_L = 600;
   localparam BAR_X_R = 603;
   // bar top, bottom boundary
   wire [9:0] bar_y_t, bar_y_b;
   localparam BAR_Y_SIZE = 72;
   // register to track top boundary  (x position is fixed)
   reg [9:0] bar_y_reg, bar_y_next;
   // bar moving velocity when a button is pressed
   localparam BAR_V = 4;
   //-----------------------------------------------
   // square ball
   //-----------------------------------------------
   localparam BALL_SIZE = 8;
   // ball left, right boundary
   wire [9:0] ball_x_l, ball_x_r;
   // ball top, bottom boundary
   wire [9:0] ball_y_t, ball_y_b;
   // reg to track left, top position
   reg [9:0] ball_x_reg, ball_y_reg;
   wire [9:0] ball_x_next, ball_y_next;
   // reg to track ball speed
   reg [9:0] x_delta_reg, x_delta_next;
   reg [9:0] y_delta_reg, y_delta_next;
   // ball velocity can be pos or neg)
   localparam BALL_V_P = 2;
   localparam BALL_V_N = -2;
   //-----------------------------------------------
   // round ball
   //-----------------------------------------------
   wire [2:0] rom_addr, rom_col;
   reg [7:0] rom_data;
   wire rom_bit;
   //-----------------------------------------------
   // object output signals
   //-----------------------------------------------
```

```
   wire wall_on, bar_on, sq_ball_on, rd_ball_on;
   wire [2:0] wall_rgb, bar_rgb, ball_rgb;

   // body
   //-----------------------------------------------------------
   // round ball image ROM
   //-----------------------------------------------------------
   always @*
   case (rom_addr)
      3'h0: rom_data = 8'b00111100; //   ****
      3'h1: rom_data = 8'b01111110; //  ******
      3'h2: rom_data = 8'b11111111; // ********
      3'h3: rom_data = 8'b11111111; // ********
      3'h4: rom_data = 8'b11111111; // ********
      3'h5: rom_data = 8'b11111111; // ********
      3'h6: rom_data = 8'b01111110; //  ******
      3'h7: rom_data = 8'b00111100; //   ****
   endcase

   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            bar_y_reg <= 0;
            ball_x_reg <= 0;
            ball_y_reg <= 0;
            x_delta_reg <= 10'h004;
            y_delta_reg <= 10'h004;
         end
      else
         begin
            bar_y_reg <= bar_y_next;
            ball_x_reg <= ball_x_next;
            ball_y_reg <= ball_y_next;
            x_delta_reg <= x_delta_next;
            y_delta_reg <= y_delta_next;
         end

   // refr_tick: 1-clock tick asserted at start of v-sync
   //            i.e., when the screen is refreshed (60 Hz)
   assign refr_tick = (pix_y==481) && (pix_x==0);

   //-----------------------------------------------------------
   // (wall) left vertical strip
   //-----------------------------------------------------------
   // pixel within wall
   assign wall_on = (WALL_X_L<=pix_x) && (pix_x<=WALL_X_R);
   // wall rgb output
   assign wall_rgb = 3'b001; // blue
   //-----------------------------------------------------------
   // right vertical bar
   //-----------------------------------------------------------
   // boundary
```

```
   assign bar_y_t = bar_y_reg;
   assign bar_y_b = bar_y_t + BAR_Y_SIZE - 1;
   // pixel within bar
   assign bar_on = (BAR_X_L<=pix_x) && (pix_x<=BAR_X_R) &&
                   (bar_y_t<=pix_y) && (pix_y<=bar_y_b);
   // bar rgb output
   assign bar_rgb = 3'b010; // green
   // new bar y-position
   always @*
   begin
      bar_y_next = bar_y_reg; // no move
      if (refr_tick)
         if (btn[1] & (bar_y_b < (MAX_Y-1-BAR_V)))
            bar_y_next = bar_y_reg + BAR_V; // move down
         else if (btn[0] & (bar_y_t > BAR_V))
            bar_y_next = bar_y_reg - BAR_V; // move up
   end

   //-----------------------------------------------
   // square ball
   //-----------------------------------------------
   // boundary
   assign ball_x_l = ball_x_reg;
   assign ball_y_t = ball_y_reg;
   assign ball_x_r = ball_x_l + BALL_SIZE - 1;
   assign ball_y_b = ball_y_t + BALL_SIZE - 1;
   // pixel within ball
   assign sq_ball_on =
            (ball_x_l<=pix_x) && (pix_x<=ball_x_r) &&
            (ball_y_t<=pix_y) && (pix_y<=ball_y_b);
   // map current pixel location to ROM addr/col
   assign rom_addr = pix_y[2:0] - ball_y_t[2:0];
   assign rom_col = pix_x[2:0] - ball_x_l[2:0];
   assign rom_bit = rom_data[rom_col];
   // pixel within ball
   assign rd_ball_on = sq_ball_on & rom_bit;
   // ball rgb output
   assign ball_rgb = 3'b100;   // red
   // new ball position
   assign ball_x_next = (refr_tick) ? ball_x_reg+x_delta_reg :
                        ball_x_reg ;
   assign ball_y_next = (refr_tick) ? ball_y_reg+y_delta_reg :
                        ball_y_reg ;
   // new ball velocity
   always @*
   begin
      x_delta_next = x_delta_reg;
      y_delta_next = y_delta_reg;
      if (ball_y_t < 1) // reach top
         y_delta_next = BALL_V_P;
      else if (ball_y_b > (MAX_Y-1)) // reach bottom
         y_delta_next = BALL_V_N;
      else if (ball_x_l <= WALL_X_R) // reach wall
```

```
            x_delta_next = BALL_V_P;     // bounce back
         else if ((BAR_X_L<=ball_x_r) && (ball_x_r<=BAR_X_R) &&
                  (bar_y_t<=ball_y_b) && (ball_y_t<=bar_y_b))
            // reach x of right bar and hit, ball bounce back
            x_delta_next = BALL_V_N;
      end
      //-----------------------------------------------------
      // rgb multiplexing circuit
      //-----------------------------------------------------
      always @*
         if (~video_on)
            graph_rgb = 3'b000; // blank
         else
            if (wall_on)
               graph_rgb = wall_rgb;
            else if (bar_on)
               graph_rgb = bar_rgb;
            else if (rd_ball_on)
               graph_rgb = ball_rgb;
            else
               graph_rgb = 3'b110; // yellow background

   endmodule
```

As in the still screen, we can combine the synchronization circuit and create the top-level description. The HDL code is shown in Listing 13.6.

Listing 13.6 Complete circuit for the animated pong game screen

```
module pong_top_an
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;

   // body
   // instantiate vga sync circuit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));

   // instantiate graphic generator
   pong_graph_animate pong_graph_an_unit
      (.clk(clk), .reset(reset), .btn(btn),
       .video_on(video_on), .pix_x(pixel_x),
```

```
      .pix_y(pixel_y), .graph_rgb(rgb_next));

   // rgb buffer
   always @(posedge clk)
30    if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;

35 endmodule
```

Note that there is no other control mechanism is this code. The ball simply moves and bounces continuously. A top-level control circuit is discussed in Chapter 14.

13.5 GRAPHIC GENERATION WITH A BIT-MAPPED SCHEME

The bit-mapped scheme maps each pixel to a word in video memory. There are about 310k pixels in a 640-by-480 screen. This translates to 310k and 930k bits for monochrome and color displays, respectively. The actual size of the video memory can be much larger since the memory address must be properly aligned for fast access. For example, to map the pixel's current coordinates to a memory location, we can concatenate the pixel's x-coordinate, which is 10 bits (i.e., $\lceil \log_2(640) \rceil$), and the pixel's y-coordinate, which is 9 bits (i.e., $\lceil \log_2(480) \rceil$). This approach requires no additional circuit to translate the pixel's coordinates to a memory address but introduces some unused "holes" in memory. The memory size is increased from 310k words to 512K (i.e., 2^{10+9}) words.

For the S3 board, memory is available from the external SRAM chips and FPGA's embedded block RAMs, as discussed in Chapters 11 and 12. Recall that the total capacity of the Spartan 3S200 device's block RAM is only about 192K bits. It is not large enough for a full-screen bit-mapped display. We must use the external SRAM, which is 8M bits, for this purpose.

In this section, we use a small 128-by-128 (2^7-by-2^7) area of the screen to illustrate the design of the bit-mapped scheme. The screen has 16K (2^{14}) pixels in this area and requires a 16K-by-3 video memory for color display. This can be implemented by three embedded block RAMs. The small area is at the top-left corner of the screen and displays the trace of a bouncing one-pixel dot, as shown in Figure 13.9. The circuit uses a 3-bit switch to specify the color of the trace and a pushbutton switch to randomly select the origin of the trace. When the pushbutton switch is pressed, the dot starts to move, like the bouncing ball in Section 13.4.3. The trace forms a rectangle after the dot hits the four sides of the small area. A new trace is generated each time the pushbutton switch is pressed.

13.5.1 Dual-port RAM implementation

A conceptual block diagram of this circuit is shown in Figure 13.10. The video memory is a synchronous 16K-by-3 (i.e., 2^{14}-by-3) dual-port RAM. The dual-port module discussed in Listing 12.4 can be used for this purpose. The seven LSBs of the pixel's y-coordinate form the seven MSBs of the memory address, and the seven LSBs of the pixel's x-coordinate form the seven LSBs of the memory address. The `dot_xy` circuit keeps track of the current location of the dot and generates its current y- and x-coordinates, which are concatenated as the write address. The 3-bit external switch input, `sw`, is the `rgb` value, which is connected

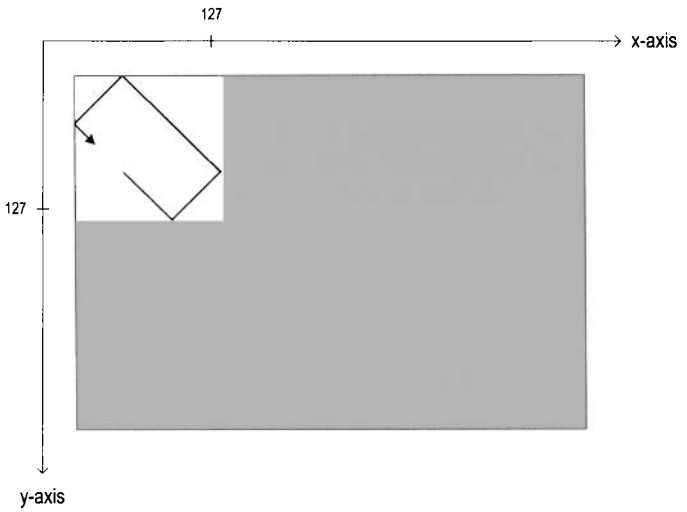


Figure 13.9 Dot trace shown in a 128-by-128 bit map.

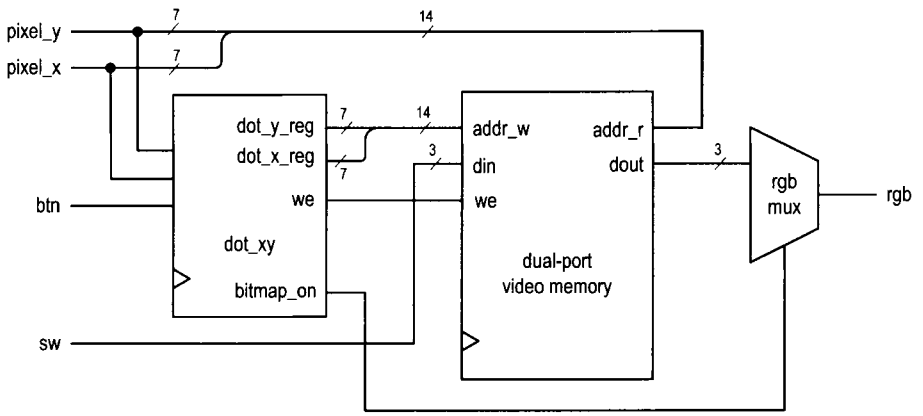


Figure 13.10 Conceptual block diagram of a dot trace circuit.

to the memory's din_a port. The seven LSBs of pixel_y and the seven LSBs of pixel_x form the read address. The data is retrieved continuously and the corresponding readout is routed to the rgb multiplexing circuit.

The complete code of the dot trace pixel generation circuit is shown in Listing 13.7. We use two registers, dot_x_reg and dot_y_reg, to keep track of the dot's current x- and y-coordinates and use two registers, v_x_reg and v_y_reg, to keep track of the current horizontal and vertical velocities. Computation of the dot's coordinates and velocities is similar to that of the bouncing ball in Section 13.4.3. In addition to regular updates, the dot_x_next and dot_y_next signals obtain the values of the seven LSBs of pix_x and pix_y when the pushbutton switch is pressed. Since these signals change much faster than a human's perception, the new origin appears to be random.

Listing 13.7 Pixel-generation circuit for a 128-by-128 bit map

```
module bitmap_gen
   (
    input wire clk, reset,
    input wire video_on,
    input [1:0] btn,
    input [2:0] sw,
    input wire [9:0] pix_x, pix_y,
    output reg [2:0] bit_rgb
   );

   // constant and signal declaration
   wire refr_tick, load_tick;
   //-----------------------------------------------
   // video sram
   //-----------------------------------------------
   wire we;
   wire [13:0] addr_r, addr_w;
   wire [2:0] din, dout;
   //-----------------------------------------------
   // dot location and velocity
   //-----------------------------------------------
   localparam MAX_X = 128;
   localparam MAX_Y = 128;
   // dot velocity can be pos or neg
   localparam DOT_V_P = 1;
   localparam DOT_V_N = -1;
   // reg to keep track of dot location
   reg [6:0] dot_x_reg, dot_y_reg;
   wire [6:0] dot_x_next, dot_y_next;
   // reg to keep track of dot velocity
   reg [6:0] v_x_reg, v_y_reg;
   wire [6:0] v_x_next, v_y_next;
   //-----------------------------------------------
   // object output signals
   //-----------------------------------------------
   wire bitmap_on;
   wire [2:0] bitmap_rgb;

   // body
```

```
40    // instantiate debounce circuit for a button
      debounce deb_unit
         (.clk(clk), .reset(reset), .sw(btn[0]),
          .db_level(), .db_tick(load_tick));
      // instantiate dual-port video RAM (2^12-by-7)
45    xilinx_dual_port_ram_sync
         #(.ADDR_WIDTH(14), .DATA_WIDTH(3)) video_ram
         (.clk(clk), .we(we), .addr_a(addr_w), .addr_b(addr_r),
          .din_a(din), .dout_a(), .dout_b(dout));
      // video ram interface
50    assign addr_w = {dot_y_reg, dot_x_reg};
      assign addr_r = {pix_y[6:0], pix_x[6:0]};
      assign we = 1'b1;
      assign din = sw;
      assign bitmap_rgb = dout;
55    // registers
      always @(posedge clk, posedge reset)
         if (reset)
            begin
               dot_x_reg <= 0;
60             dot_y_reg <= 0;
               v_x_reg <= DOT_V_P;
               v_y_reg <= DOT_V_P;
            end
         else
65          begin
               dot_x_reg <= dot_x_next;
               dot_y_reg <= dot_y_next;
               v_x_reg <= v_x_next;
               v_y_reg <= v_y_next;
70          end

      // refr_tick: 1-clock tick asserted at start of v-sync
      assign refr_tick = (pix_y==481) && (pix_x==0);

75    // pixel within bit map area
      assign bitmap_on =(pix_x<=127) & (pix_y<=127);
      // dot position
      // "randomly" load dot location when btn[0] pressed
      assign dot_x_next = (load_tick) ? pix_x[6:0] :
80                        (refr_tick) ? dot_x_reg + v_x_reg :
                          dot_x_reg ;
      assign dot_y_next = (load_tick) ? pix_y[6:0] :
                          (refr_tick) ? dot_y_reg + v_y_reg :
                          dot_y_reg ;
85    // dot x velocity
      assign v_x_next =
           (dot_x_reg==1) ? DOT_V_P :          // reach left
           (dot_x_reg==(MAX_X-2)) ? DOT_V_N : // reach right
            v_x_reg;
90    // dot y velocity
      assign v_y_next =
           (dot_y_reg==1) ? DOT_V_P :          // reach top
```

```
      (dot_y_reg==(MAX_Y-2)) ? DOT_V_N : // reach bottom
       v_y_reg;
//-----------------------------------------------
// rgb multiplexing circuit
//-----------------------------------------------
always @*
   if (~video_on)
      bit_rgb = 3'b000; // blank
   else
      if (bitmap_on)
         bit_rgb = bitmap_rgb;
      else
         bit_rgb = 3'b110; // yellow background

endmodule
```

The HDL code for the top-level system is shown in Listing 13.8.

Listing 13.8 Complete circuit for a bit-mapped screen

```
module dot_top
   (
    input wire clk, reset,
    input wire [1:0] btn,
    input wire [2:0] sw,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;

   // body
   // instantiate VGA sync circuit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));

   // instantiate graphic generator
   bitmap_gen bitmap_unit
      (.clk(clk), .reset(reset), .btn(btn), .sw(sw),
       .video_on(video_on), .pix_x(pixel_x),
       .pix_y(pixel_y), .bit_rgb(rgb_next));

   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;
```

```
35
endmodule
```

13.5.2 Single-port RAM implementation

Although a dual-port memory is ideal, it is not always available. Using regular single-port memory, such as the S3 board's external SRAM, for the video memory requires careful coordination between the write and read operations to avoid interruption in data retrieval. For demonstration purposes, we configure the embedded block RAM as a single-port synchronous SRAM and redesign the previous dot trace circuit.

In the dot trace circuit, the dot's coordinates are updated once every screen scan. Thus, the video memory can be written at this rate as well. We can do this during the vertical retrace since the video is off in this period and writing video memory does not interfere with screen data retrieval. Note that the `refr_tick` signal is asserted when `pixel_y` is 481. The video is off in this location, and writing video memory will not interfere with the screen data retrieval. We use this signal as the write enable signal, `we`, for the single-port RAM. The single-port RAM module discussed in Listing 12.2 can be used for this purpose. The memory portion of Listing 13.7 now becomes

```
// instantiate dual-port video RAM (2^12-by-7)
xilinx_one_port_ram_sync
   #(.ADDR_WIDTH(14), .DATA_WIDTH(3)) video_ram
   (.clk(clk), .we(we), .addr(addr),
    .din(din), .dout(dout));
// video ram interface
assign addr_w = {dot_y_reg, dot_x_reg};
assign addr_r = {pix_y[6:0], pix_x[6:0]};
assign addr  = (refr_tick) ? addr_w : addr_r;
assign we = refr_tick;
assign din = sw;
assign bitmap_rgb = dout;
```

The dot trace circuit updates one pixel in a screen scan. The required memory bandwidth for writing is 60*3 bits per second, which is rather low. Thus, the previous design is fairly straightforward. The design of memory interface becomes much more difficult when a large memory bandwidth is required (i.e., when a large portion of the screen is updated at a rapid rate).

13.6 BIBLIOGRAPHIC NOTES

Rapid Prototyping of Digital Systems by James O. Hamblen et al. contains timing information for monitors with different resolutions and refresh rates.

13.7 SUGGESTED EXPERIMENTS

13.7.1 VGA test pattern generator

A VGA test pattern generator produces two simple patterns to verify operation of a VGA monitor. The first pattern divides the screen evenly into eight vertical stripes, each displaying

a unique color. The second pattern is similar but the screen is divided into eight horizontal stripes. A 1-bit switch is used to select the pattern.

Design a pixel-generating circuit for this pattern generator and then combine it with the synchronization circuit in a top-level module. Synthesize and verify operation of the circuit.

13.7.2 SVGA mode synchronization circuit

The specification for the super VGA (SVGA) mode with 72-Hz refresh rate is

- *resolution*: 800-by-600 pixels
- *pixel rate*: 50 MHz
- *horizontal display region*: 800 pixels
- *horizontal right border*: 64 pixels
- *horizontal left border*: 56 pixels
- *horizontal retrace*: 120 pixels
- *vertical display region*: 600 lines
- *vertical bottom border*: 23 lines
- *vertical top border*: 37 lines
- *vertical retrace*: 6 lines

We wish to create a dual-mode synchronization circuit that can support both VGA and SVGA modes. The mode can be selected by a switch. Construct the circuit as follows:

1. Modify the horizontal and vertical synchronization counters of Listing 13.1 to accommodate both modes.
2. Design a pixel-generating circuit that draws a 100-pixel grid on the screen (i.e., draw a vertical line every 100 pixels and draw a horizontal line every 100 pixels).
3. Derive a top-level module. Synthesize and verify operation of the two modes.

13.7.3 Visible screen adjustment circuit

Due to the internal timing error of a monitor, the visible portion of the screen may not always be centered. We can adjust the location of the visible portion by slightly modifying the widths surrounding black border areas. In a horizontal scan line, there are 64 pixels for the right and left border regions. To move the visible portion horizontally, we can add a certain number of pixels to one border region and subtract the same number from the opposite border region. We can adjust the visible portion vertically in a similar fashion. Design a screen adjustment circuit as follows:

1. Expand the VGA synchronization circuit to include this feature. Use a switch to select the vertical or horizontal mode, and use two pushbuttons to move the visible screen to left/up and right/down.
2. Modify the testing circuit in Section 13.2.5 to incorporate the new synchronization circuit.
3. Synthesize and verify operation of the circuit.

13.7.4 Ball-in-a-box circuit

The ball-in-a-box circuit displays a bouncing ball inside a square box. The square box is centered on the screen and its size is 256-by-256 pixels. The ball is an 8-by-8 round ball. When the ball hits the wall, the ball bounces back and the wall flashes (i.e., changes color briefly). The ball can travel at four different speeds, which are selected by two slide

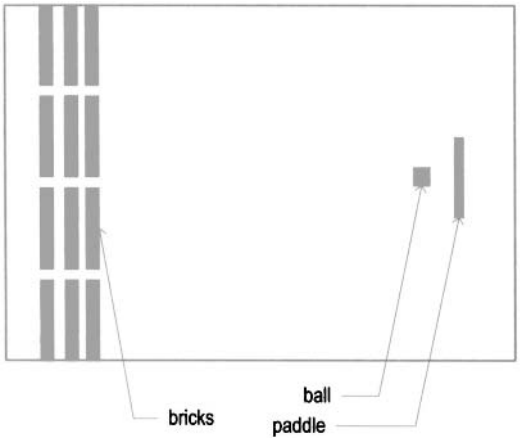


Figure 13.11 Screen of the breakout game.

switches, and its direction changes randomly when a pushbutton switch is pressed. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.5 Two-balls-in-a-box circuit

We can expand the circuit in Experiment 13.7.4 to include two balls inside the box. When two balls collide, the new directions of the two balls should follow the laws of physics. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.6 Two-player pong game

The two-player pong game replaces the left wall with another paddle, which is controlled by the second player. To better accommodate two players, we can use the keyboard interface of Section 9.4 as the input device. Four keys can be defined to control vertical movements of the two paddles. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.7 Breakout game

The breakout game is somewhat like the pong game. In this game, the left wall is replaced by several layers of "bricks." When the ball hits a brick, the ball bounces back and the brick disappears. The basic screen is shown in Figure 13.11. As in the code of Listing 13.5, we assume that the game runs continuously. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.8 Full-screen dot trace

We can implement the full-screen dot trace circuit of Section 13.5 using the external SRAM chip as follows:

1. Modify the SRAM controller in Chapter 11 to configure the SRAM chip as a 2^{19}-by-8 memory.

2. Follow the discussion in Section 13.5.2 to incorporate the new memory module in the circuit. Note that accessing the external memory requires two clock cycles.
3. Synthesize and verify operation of the circuit.

13.7.9 Mouse pointer circuit

The mouse interface is discussed in Section 10.5. The mouse pointer circuit uses a mouse to control the movement of a small 16-by-16 square on the screen. It functions as follows:

- The square moves according to the movement of the mouse.
- The pointer wraps around when it reaches a border.
- The pointer changes color when the left button of the mouse is pressed. It circulates through the eight colors defined in Table 13.1.

Synthesize and verify operation of the circuit.

13.7.10 Small-screen mouse scribble circuit

Mouse scribble circuit keeps track of the trace of the mouse movement in a 128-by-128 screen, somewhat similar to the dot trace circuit discussed in Section 13.5. Its specification is as follows:

- The 3-bit switch determines the color of the trace.
- Clicking the left button of the mouse turns on and off the trace alternately.
- Clicking the right button of the mouse clears the screen.

Synthesize and verify operation of the circuit.

13.7.11 Full-screen mouse scribble circuit

Repeat Experiment 13.7.10, but use the full screen. An external SRAM module similar to that in Experiment 13.7.8 is needed for this circuit.

CHAPTER 14

VGA CONTROLLER II: TEXT

14.1 INTRODUCTION

A tile-mapped pixel generation scheme is discussed in Section 13.3. A tile can be considered as a "super pixel." Whereas a pixel is defined by a 3-bit word in a bit-mapped scheme, a tile is mapped to a predesigned pattern. One method of constructing a text display is to treat the characters as tiles and design the pixel generation circuit with the tile-mapped scheme. We discuss this method in this chapter and apply it to add scores and rules to the pong game.

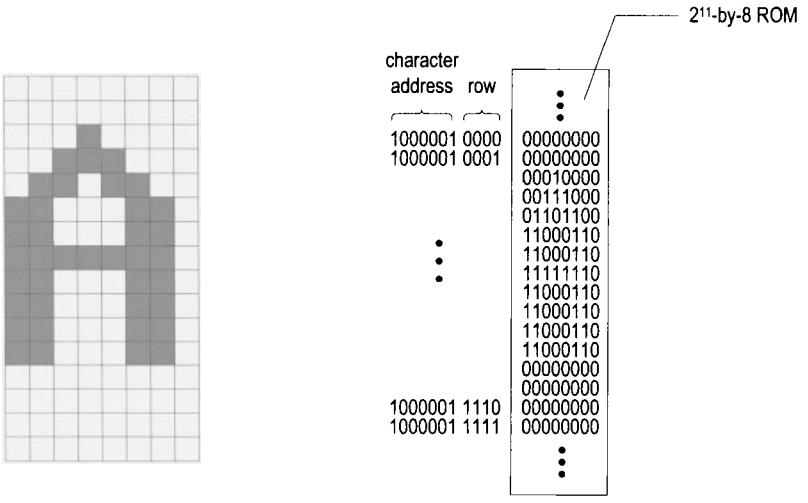
14.2 TEXT GENERATION

14.2.1 Character as a tile

When applying a tile-mapped scheme, we treat each character as a tile. In a bit-mapped scheme, the value of a pixel represents a 3-bit color. On the other hand, the value of a tile represents the code of a specific pattern. For the text display, we use the 7-bit ASCII code for the character tiles.

The patterns of the tiles constitute the *font* of the character set. A variety of fonts are available. We choose an 8-by-16 (i.e., 8-column-by-16-row) font similar to the one used in early IBM PCs. In this font, each character is represented as an 8-by-16 pixel pattern. The pattern for the letter "A" is shown in Figure 14.1(a).

The character patterns are stored in a ROM and each pattern requires $2^4 * 8$ bits. The pattern memory is known as *font ROM*. The original font set consists of 256 patterns,



(a) Pixel pattern

(b) ROM content

Figure 14.1 Font pattern for the letter A.

including digits, upper- and lowercase letters, punctuation symbols, and many special-purpose graphic symbols. We implement only the first half [i.e., 128 (2^7)] of the patterns and exclude most graphic symbols. To accommodate this set, $2^7 * 2^4 * 8$ ROM bits are needed. It is usually configured as a 2^{11}-by-8 ROM.

When we use these 8-by-16 characters (i.e., tiles) in a 640-by-480 resolution screen, 80 (i.e., $\frac{640}{8}$) tiles can be fitted into a horizontal line and 30 (i.e., $\frac{480}{16}$) tiles can be fitted into a vertical line. In other words, the screen can be treated as an 80-by-25 tile screen. We can put characters on the screen using these scaled coordinates.

14.2.2 Font ROM

Our font set implements the 128 characters of the ASCII code, listed in Table 8.1. The 128 (2^7) character patterns can be accommodated by a 2^{11}-by-8 font ROM. In this ROM, the seven MSBs of the 11-bit address are used to identify the character, and the four LSBs of the address are used to identify the row within a character pattern. The address and ROM content for the letter "A" are shown in Figure 14.1(b).

In the ASCII table, the first column (ASCII codes 00_{16} to $1F_{16}$) consists of nonprintable control characters. The font ROM uses these codes to implement special graphic symbols. For example, the 06_{16} code will generate a spade pattern, ♠, on the screen. Note that the 00_{16} code is reserved for a blank tile.

The 2^{11}-by-8 font ROM can fit neatly into a single block RAM of the Spartan-3 device. We use the ROM template of Listing 12.6 to ensure that a block RAM will be inferred during synthesis. Part of the HDL code is shown in Listing 14.1. The complete code has 2^{11} rows in constant definition and the file can be downloaded from the companion Web site.

Listing 14.1 Partial code of the font ROM

```
module font_rom
   (
    input wire clk,
    input wire [10:0] addr,
    output reg [7:0] data
   );

   // signal declaration
   reg [10:0] addr_reg;

   // body
   always @(posedge clk)
      addr_reg <= addr;

   always @*
      case (addr_reg)
         //code x00 blank
         11'h000: data = 8'b00000000; //
         11'h001: data = 8'b00000000; //
         11'h002: data = 8'b00000000; //
         11'h003: data = 8'b00000000; //
         11'h004: data = 8'b00000000; //
         11'h005: data = 8'b00000000; //
         11'h006: data = 8'b00000000; //
         11'h007: data = 8'b00000000; //
         11'h008: data = 8'b00000000; //
         11'h009: data = 8'b00000000; //
         11'h00a: data = 8'b00000000; //
         11'h00b: data = 8'b00000000; //
         11'h00c: data = 8'b00000000; //
         11'h00d: data = 8'b00000000; //
         11'h00e: data = 8'b00000000; //
         11'h00f: data = 8'b00000000; //
         //code x01 smiley face
         11'h010: data = 8'b00000000; //
         11'h011: data = 8'b00000000; //
         11'h012: data = 8'b01111110; //  ******
         11'h013: data = 8'b10000001; // *      *
         11'h014: data = 8'b10100101; // * *  * *
         11'h015: data = 8'b10000001; // *      *
         11'h016: data = 8'b10000001; // *      *
         11'h017: data = 8'b10111101; // * **** *
         11'h018: data = 8'b10011001; // *  **  *
         11'h019: data = 8'b10000001; // *      *
         11'h01a: data = 8'b10000001; // *      *
         11'h01b: data = 8'b01111110; //  ******
         11'h01c: data = 8'b00000000; //
         11'h01d: data = 8'b00000000; //
         11'h01e: data = 8'b00000000; //
         11'h01f: data = 8'b00000000; //
         . . .
         //code x7f
```

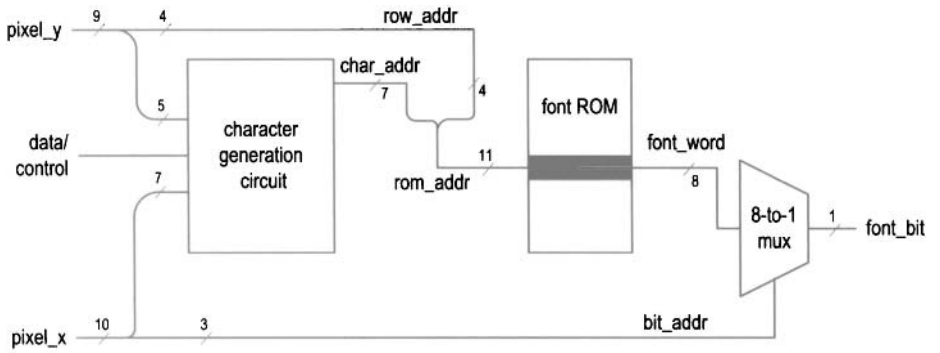


Figure 14.2 Two-stage text generation circuit.

```
            11'h7f0: data = 8'b00000000; //
            11'h7f1: data = 8'b00000000; //
            11'h7f2: data = 8'b00000000; //
            11'h7f3: data = 8'b00000000; //
            11'h7f4: data = 8'b00010000; //    *
            11'h7f5: data = 8'b00111000; //   ***
            11'h7f6: data = 8'b01101100; //  ** **
            11'h7f7: data = 8'b11000110; // **   **
            11'h7f8: data = 8'b11000110; // **   **
            11'h7f9: data = 8'b11000110; // **   **
            11'h7fa: data = 8'b11111110; // *******
            11'h7fb: data = 8'b00000000; //
            11'h7fc: data = 8'b00000000; //
            11'h7fd: data = 8'b00000000; //
            11'h7fe: data = 8'b00000000; //
            11'h7ff: data = 8'b00000000; //
        endcase

endmodule
```

Note that the block RAM-based ROM implementation introduces a one-clock-cycle delay, as discussed in Section 12.4.3.

14.2.3 Basic text generation circuit

The pixel generation circuit generates pixel values according to the current pixel coordinates (provided by the `pixel_x` and `pixel_y` signals) and the external data and control signals. Pixel generation based on a tile-mapped scheme involves two stages. The first stage uses the upper bits of the `pixel_x` and `pixel_y` signals to generate a tile's code, and the second stage uses this code and lower bits to generate the pixel's value.

The text generation circuit follows this method, and the basic diagram is shown in Figure 14.2. The screen is treated as a grid of 80-by-30 tiles, each containing an 8-by-16 font pattern. In the first stage, the `pixel_x[9:3]` and `pixel_y[8:4]` signals provide the x- and y-coordinates of the current tile location. The character generation circuit uses these coordinates, combined with other external data, to generate the value of this tile (labeled `char_addr`), which corresponds to a character's ASCII code. In the second stage, the ASCII

code becomes the seven MSBs of the address of the font ROM and specifies the location of the current pattern. It is concatenated with the four LSBs of the screen's y-coordinate (i.e., `pixel_y[3:0]`, labeled `row_addr`) to form the complete address (labeled `rom_addr`) of the font ROM. The output of the font ROM (labeled `font_word`) corresponds to an 8-bit row in the pattern. The three LSBs of the screen's x-coordinate (i.e., `pixel_x[2:0]`, labeled `bit_addr`) specify the desired pixel location, and an 8-to-1 multiplexer routes the pixel to the output.

14.2.4 Font display circuit

We use a simple font display circuit to verify operation of the font ROM and display all font patterns on the screen. The 128 patterns are arranged in four rows, which correspond to the four columns of the ASCII table in Table 8.1. We can obtain each pattern by using the proper x- and y-coordinates to generate the desired ASCII code, which is labeled the `char_addr` signal. The code segment is

```
assign char_addr = {pixel_y[5:4], pixel_x[7:3]};
```

The `pixel_x[7:3]` signal forms the five LSBs of the ASCII code, and thus 32 (2^5) consecutive font patterns will be displayed in a row. The `pixel_y[5:4]` signal forms the two MSBs of the ASCII code, and thus four consecutive rows will be displayed. Since the upper bits of the `pixel_x` and `pixel_y` signals are left unspecified, the 32-by-4 region will be displayed repetitively on the screen. An additional code segment is included to turn on the display for the top-left portion of the screen only. The complete code is shown in Listing 14.2.

Listing 14.2 Pixel generation of a font display circuit

```
module font_test_gen
   (
    input wire clk,
    input wire video_on,
    input wire [9:0] pixel_x, pixel_y,
    output reg [2:0] rgb_text
   );

   // signal declaration
   wire [10:0] rom_addr;
   wire [6:0] char_addr;
   wire [3:0] row_addr;
   wire [2:0] bit_addr;
   wire [7:0] font_word;
   wire font_bit, text_bit_on;

   // body
   // instantiate font ROM
   font_rom font_unit
      (.clk(clk), .addr(rom_addr), .data(font_word));
   // font ROM interface
   assign char_addr = {pixel_y[5:4], pixel_x[7:3]};
   assign row_addr = pixel_y[3:0];
   assign rom_addr = {char_addr, row_addr};
   assign bit_addr = pixel_x[2:0];
```

```
   assign font_bit = font_word[~bit_addr];
   // "on" region limited to top-left corner
   assign text_bit_on = (pixel_x[9:8]==0 && pixel_y[9:6]==0) ?
                        font_bit : 1'b0;
   // rgb multiplexing circuit
   always @*
      if (~video_on)
         rgb_text = 3'b000; // blank
      else
         if (text_bit_on)
            rgb_text = 3'b010;  // green
          else
            rgb_text = 3'b000;  // black

endmodule
```

The key part of the code is the font ROM interface. For clarity, we define the following signals for the font ROM, as shown in Figure 14.2:

- `char_addr`: 7 bits, the ASCII code of the character
- `row_addr`: 4 bits, the row number in a particular font pattern
- `rom_addr`: 11 bits, the address of the font ROM; the concatenation of `char_addr` and `row_addr`
- `bit_addr`: 3 bits, the column number in a particular font pattern
- `font_word`: 8 bits, a row of pixels of the font pattern specified by `rom_addr`
- `font_bit`: 1 bit, one pixel of `font_word` specified by `bit_addr`

The connection of these signals follows the diagram in Figure 14.2. The routing of the `font_bit` signal is done by a multiplexer, coded as an array with a dynamic index:

```
assign font_bit = font_word[~bit_addr];
```

Note that a row (i.e., a word) in the font ROM is defined in descending order (i.e., `[7:0]`). Since the screen's x-coordinate is defined in ascending fashion, in which the number increases from left to right, the order of the retrieved bits must be reversed. This is achieved by the `~` operator in the expression.

We need to combine the synchronization circuit and create the top-level description. The HDL code is shown in Listing 14.3.

Listing 14.3 Top-level description of a font display circuit

```
module font_test_top
   (
    input wire clk, reset,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;

   // body
   // instantiate vga sync circuit
```

```
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));
   // font generation circuit
   font_test_gen font_gen_unit
      (.clk(clk), .video_on(video_on), .pixel_x(pixel_x),
       .pixel_y(pixel_y), .rgb_text(rgb_next));
   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;

endmodule
```

There is subtle timing issue in this circuit. Because of the block RAM implementation, the font ROM's output suffers a one-clock-cycle delay. However, since the `pixel_tick` signal is asserted every two clock cycles, the `pixel_x` signal remains unchanged within this interval and the corresponding bit (i.e., `font_bit`) can be retrieved properly. The rgb multiplexing circuit can use this data, and the desired value is stored to the `rgb_reg` register in a timely manner.

14.2.5 Font scaling

In the tile-mapped scheme, we can scale a tile pattern to larger sizes by "enlarging" the screen pixels. For example, we can scale the 8-by-16 font to a 16-by-32 font by enlarging the original pixel four times (i.e., expanding one pixel to four pixels). To perform the scaling, we just need to shift pixel coordinates to the right 1 bit and discard the LSBs of the `pixel_x` and `pixel_y` signals. This can best be explained by an example. Let us repeat the previous font displaying circuit with enlarged 16-by-32 fonts. The screen can now be treated as a grid of 40-by-15 tiles. The new font addresses become

```
assign row_addr = pixel_y[4:1];
assign bit_addr = pixel_x[3:1];
assign char_addr = {pixel_y[6:5], pixel_x[8:4]};
```

The first two statements imply that the same `font_bit` value will be obtained when `pixel_x[0]` and `pixel_y[0]` are "00", "01", "10", and "11", and this effectively enlarges the original pixel to four pixels. The `text_bit_on` condition also needs to be modified to accommodate a larger region:

```
assign text_bit_on = (pixel_x[9]==0 && pixel_y[9:7]==0) ?
                     font_bit : 1'b0;
```

We can apply this scheme to scale up the font even further. Note that the enlarged fonts may appear jagged because they simply magnify the original pattern and introduce no new detail.

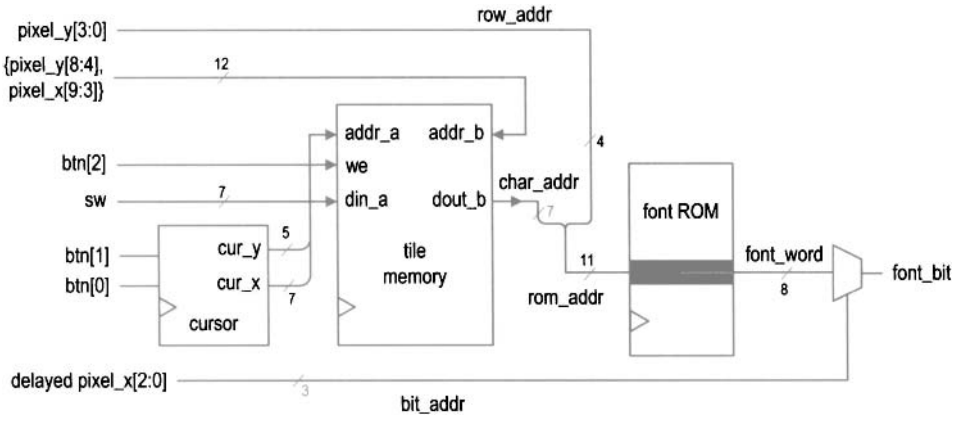


Figure 14.3 Text generation circuit with tile memory.

14.3 FULL-SCREEN TEXT DISPLAY

A full-screen text display, as the name indicates, uses the entire screen to display text characters. The character generation circuit now contains a *tile memory* that stores the ASCII code of each tile. The design of the tile memory is similar to the video memory of the bit-mapped circuit in Section 13.5. For easy memory access, we can concatenate the x- and y-coordinates of a tile to form the address. This translates to 12 bits for the 80-by-30 (i.e., 2^7-by-2^5) tile screen. Since each tile contains a 7-bit ASCII code, a 2^{12}-by-7 memory module is required. A synchronous dual-port RAM can be used for this purpose. A circuit with tile memory is shown in Figure 14.3.

Because accessing tile memory requires another clock cycle, retrieving a font pattern is now increased to two clock cycles. This prolonged delay introduces a subtle timing problem. Because the `pixel_x` signal is updated every two clock cycles, its value has incremented when the `font_word` value becomes available. Thus, when the bit is retrieved by the statements

```
assign bit_addr = pix_x2_reg[2:0];
assign font_bit = font_word[~bit_addr];
```

the incremented `bit_addr` is used and an incorrect font bit will be selected and routed to the output. One way to overcome the problem is to pass the `pixel_x` signal through two buffers and use this delayed signal in place of the `pixel_x` signal.

We use a simple circuit to demonstrate the design of the full-screen tile-mapped scheme. The circuit reads an ASCII code from a 7-bit switch and places it in the marked location of the 80-by-30 tile screen. The conceptual diagram is shown in Figure 14.3. A cursor is included to mark the current location of entry, where the color is reversed. The `cursor` block keeps track of the current location of the cursor. The circuit uses three pushbutton switches for control. Two buttons move the cursor right and down, respectively. The third button is for the write operation. When it is pressed, the current value of the 7-bit switch is written to the tile memory. The HDL code is shown in Listing 14.4.

Listing 14.4 Pixel generation of a full-screen text display

```
module text_screen_gen
   (
    input wire clk, reset,
    input wire video_on,
    input wire [2:0] btn,
    input wire [6:0] sw,
    input wire [9:0] pixel_x, pixel_y,
    output reg [2:0] text_rgb
   );

   // signal declaration
   // font ROM
   wire [10:0] rom_addr;
   wire [6:0] char_addr;
   wire [3:0] row_addr;
   wire [2:0] bit_addr;
   wire [7:0] font_word;
   wire font_bit;
   // tile RAM
   wire we;
   wire [11:0] addr_r, addr_w;
   wire [6:0] din, dout;
   // 80-by-30 tile map
   localparam MAX_X = 80;
   localparam MAX_Y = 30;
   // cursor
   reg [6:0] cur_x_reg;
   wire [6:0] cur_x_next;
   reg [4:0] cur_y_reg;
   wire [4:0]cur_y_next;
   wire move_x_tick, move_y_tick, cursor_on;
   // delayed pixel count
   reg [9:0] pix_x1_reg, pix_y1_reg;
   reg [9:0] pix_x2_reg, pix_y2_reg;
   // object output signals
   wire [2:0] font_rgb, font_rev_rgb;

   // body
   // instantiate debounce circuit for two buttons
   debounce deb_unit0
      (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(move_x_tick));
   debounce deb_unit1
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(move_y_tick));
   // instantiate font ROM
   font_rom font_unit
      (.clk(clk), .addr(rom_addr), .data(font_word));
   // instantiate dual-port video RAM (2^12-by-7)
   xilinx_dual_port_ram_sync
      #(.ADDR_WIDTH(12), .DATA_WIDTH(7)) video_ram
      (.clk(clk), .we(we), .addr_a(addr_w), .addr_b(addr_r),
```

```
      .din_a(din), .dout_a(), .dout_b(dout));

   // registers
   always @(posedge clk)
      begin
         cur_x_reg <= cur_x_next;
         cur_y_reg <= cur_y_next;
         pix_x1_reg <= pixel_x;
         pix_x2_reg <= pix_x1_reg;
         pix_y1_reg <= pixel_y;
         pix_y2_reg <= pix_y1_reg;
      end
   // tile RAM write
   assign addr_w = {cur_y_reg, cur_x_reg};
   assign we = btn[2];
   assign din = sw;
   // tile RAM read
   // use nondelayed coordinates to form tile RAM address
   assign addr_r = {pixel_y[8:4], pixel_x[9:3]};
   assign char_addr = dout;
   // font ROM
   assign row_addr = pixel_y[3:0];
   assign rom_addr = {char_addr, row_addr};
   // use delayed coordinate to select a bit
   assign bit_addr = pix_x2_reg[2:0];
   assign font_bit = font_word[~bit_addr];
   // new cursor position
   assign cur_x_next =
      (move_x_tick && (cur_x_reg==MAX_X-1)) ? 0 : // wrap
      (move_x_tick) ? cur_x_reg + 1 :
                      cur_x_reg;
   assign cur_y_next =
      (move_y_tick && (cur_x_reg==MAX_Y-1)) ? 0 : // wrap
      (move_y_tick) ? cur_y_reg + 1 :
                      cur_y_reg;
   // object signals
   // green over black and reversed video for cursor
   assign font_rgb = (font_bit) ? 3'b010 : 3'b000;
   assign font_rev_rgb = (font_bit) ? 3'b000 : 3'b010;
   // use delayed coordinates for comparison
   assign cursor_on = (pix_y2_reg[8:4]==cur_y_reg) &&
                      (pix_x2_reg[9:3]==cur_x_reg);
   // rgb multiplexing circuit
   always @*
      if (~video_on)
         text_rgb = 3'b000; // blank
      else
         if (cursor_on)
            text_rgb = font_rev_rgb;
          else
            text_rgb = font_rgb;
endmodule
```

The font ROM interface signals are similar to those in Listing 14.2 except that the char_addr is obtained from the read port of the tile memory. To facilitate the font ROM access delay, we create two delayed signals, pix_x2_reg and pix_y2_reg, from the current x- and y-coordinates, pixel_x and pixel_y. Note that the undelayed signals, pixel_x and pixel_y, are used to form the address to access the font ROM, but the delayed signal, pix_x2_reg, is used to obtain the font bit. The instantiation and interface of the dual-port tile RAM are similar to those of the video RAM in Listing 13.7.

The cursor_on signal is used to identify the current cursor location. The colors of the font pattern are reversed in this location. Because the font bits are delayed by two clocks, we use the delayed coordinates, pix_x2_reg and pix_y2_reg, for comparison.

The delayed font bits also introduce one pixel delay for the final rgb signal. This implies that the overall visible portion of the VGA monitor is shifted to the right by one pixel. To correct the problem, we should revise the vga_sync circuit and use the delayed pix_x2_reg and pix_y2_reg signals to generate the hsync and vsync signals. Since the shift has little effect on the overall video quality, we do not make this modification.

The top-level code combines the text pixel generation circuit and the synchronization circuit and is shown in Listing 14.5.

Listing 14.5 Top-level system of a full-screen text display

```
module text_screen_top
   (
    input wire clk, reset,
    input wire [2:0] btn,
    input wire [6:0] sw,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;
   // body
   // instantiate vga sync circuit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));
   // font generation circuit
   text_screen_gen text_gen_unit
      (.clk(clk), .reset(reset), .video_on(video_on),
       .btn(btn), .sw(sw), .pixel_x(pixel_x),
       .pixel_y(pixel_y), .text_rgb(rgb_next));
   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;
endmodule
```

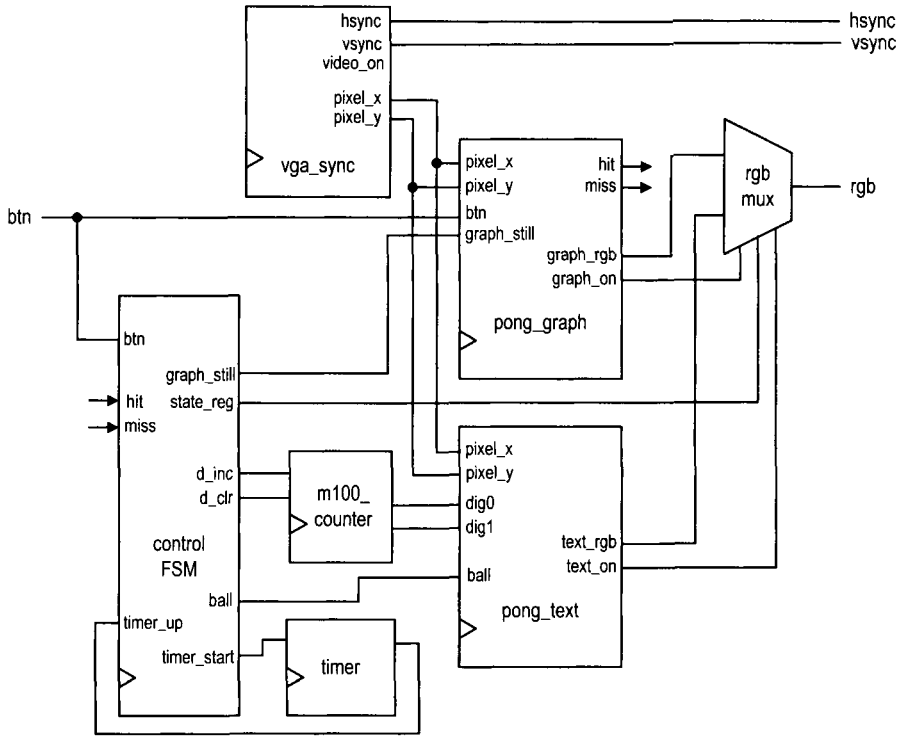


Figure 14.4 Top-level block diagram of the complete pong game.

14.4 THE COMPLETE PONG GAME

We create a free-running graphic circuit for the pong game in Section 13.4.3. In this section, we add a text interface to display scores and messages, and design a top-level control FSM that integrates the graphic and text subsystems and coordinates the overall circuit operation. The rules and operations of the complete game are:

- When the game starts, it displays the text of the rule.
- After a player presses a button, the game starts.
- The player scores a point each time hitting the ball with the paddle.
- When the player misses the ball, the game pauses and a new ball is provided. Three balls are provided in each session.
- The score and the number of remaining balls are displayed on the top of the screen.
- After three misses, the game is ended and displays the end-of-game message.

In the following subsections, we first discuss the text subsystem, graphic subsystem, and auxiliary counters, and then derive a top-level FSM to coordinate and control the overall operation. The conceptual diagram is shown in Figure 14.4.

14.4.1 Text subsystem

The text subsystem of the pong game consists of four text messages:

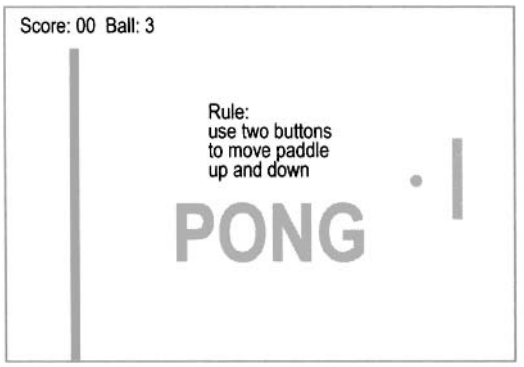


Figure 14.5 Text of the pong game.

- Display the score as "`Scores: DD`" and the number of remaining balls as"`Ball: D`" in the 16-by-32 font on top of the screen.
- Display the rule message "`Rules: Use two buttons to move paddle up or down.`" in the regular font at the beginning of the game.
- Display the "`PONG`" logo in the 64-by-128 font on the background.
- Display the end-of-game message "`Game Over`" in the 32-by-64 font at the end of the game.

A sketch of the first three messages is shown in Figure 14.5. The end-of-game message is overlapped with the rule message and not included.

Since these messages use different font sizes and are displayed at different occasions, they cannot be treated as a single screen. We treat each text message as an individual object and generate the on status signal and the font ROM address. For example, the logo message segment is

```
assign logo_on = (pix_y[9:7]==2) &&
                 (3<=pix_x[9:6]) && (pix_x[9:6]<=6);
assign row_addr_l = pix_y[6:3];
assign bit_addr_l = pix_x[5:3];
always @*
   case (pix_x[8:6])
      3'o3: char_addr_l = 7'h50;     // P
      3'o4: char_addr_l = 7'h4f;     // O
      3'o5: char_addr_l = 7'h4e;     // N
      default: char_addr_l = 7'h47; // G
   endcase
```

The `logo_on` signal indicates that the current scan is in the logo region and the corresponding text should be "turned on." The other statements specify the message content and the font ROM connections to generate the scaled 32-by-64 characters. The other three segments are similar. A separate multiplexing circuit examines various on signals and routes one set of addresses to the font ROM.

The text subsystem receives the score and the number of remaining balls via the `ball`, `dig0`, and `dig1` ports. It outputs the `rgb` information via the `rgb_text` port and outputs the on status information via the 4-bit `text_on` port, which is the concatenation of four individual on signals. The complete code is shown in Listing 14.6.

Listing 14.6 Text subsystem for the pong game

```
module pong_text
   (
    input wire clk,
    input wire [1:0] ball,
    input wire [3:0] dig0, dig1,
    input wire [9:0] pix_x, pix_y,
    output wire [3:0] text_on,
    output reg [2:0] text_rgb
   );

   // signal declaration
   wire [10:0] rom_addr;
   reg [6:0] char_addr, char_addr_s, char_addr_l,
             char_addr_r, char_addr_o;
   reg [3:0] row_addr;
   wire [3:0] row_addr_s, row_addr_l, row_addr_r, row_addr_o;
   reg [2:0] bit_addr;
   wire [2:0] bit_addr_s, bit_addr_l,bit_addr_r, bit_addr_o;
   wire [7:0] font_word;
   wire font_bit, score_on, logo_on, rule_on, over_on;
   wire [7:0] rule_rom_addr;

   // instantiate font ROM
   font_rom font_unit
      (.clk(clk), .addr(rom_addr), .data(font_word));

   //-----------------------------------------------------
   // score region
   //  - display two-digit score, ball on top left
   //  - scale to 16-by-32 font
   //  - line 1, 16 chars: "Score:DD Ball:D"
   //-----------------------------------------------------
   assign score_on = (pix_y[9:5]==0) && (pix_x[9:4]<16);
   assign row_addr_s = pix_y[4:1];
   assign bit_addr_s = pix_x[3:1];
   always @*
      case (pix_x[7:4])
         4'h0: char_addr_s = 7'h53; // S
         4'h1: char_addr_s = 7'h63; // c
         4'h2: char_addr_s = 7'h6f; // o
         4'h3: char_addr_s = 7'h72; // r
         4'h4: char_addr_s = 7'h65; // e
         4'h5: char_addr_s = 7'h3a; // :
          4'h6: char_addr_s = {3'b011, dig1}; // digit 10
          4'h7: char_addr_s = {3'b011, dig0}; // digit 1
          4'h8: char_addr_s = 7'h00; //
          4'h9: char_addr_s = 7'h00; //
          4'ha: char_addr_s = 7'h42; // B
          4'hb: char_addr_s = 7'h61; // a
          4'hc: char_addr_s = 7'h6c; // l
          4'hd: char_addr_s = 7'h6c; // l
          4'he: char_addr_s = 7'h3a; // :
```

```
            4'hf: char_addr_s = {5'b01100, ball};
         endcase
   //-------------------------------------------------
   // logo region:
   //   - display logo "PONG" at top center
   //   - used as background
   //   - scale to 64-by-128 font
   //-------------------------------------------------
   assign logo_on = (pix_y[9:7]==2) &&
                    (3<=pix_x[9:6]) && (pix_x[9:6]<=6);
   assign row_addr_l = pix_y[6:3];
   assign bit_addr_l = pix_x[5:3];
   always @*
      case (pix_x[8:6])
         3'o3: char_addr_l = 7'h50; // P
         3'o4: char_addr_l = 7'h4f; // O
         3'o5: char_addr_l = 7'h4e; // N
         default: char_addr_l = 7'h47; // G
      endcase
   //-------------------------------------------------
   // rule region
   //   - display rule (4-by-16 tiles)on center
   //   - rule text:
   //       Rule:
   //         Use two buttons
   //         to move paddle
   //         up and down
   //-------------------------------------------------
   assign rule_on = (pix_x[9:7]==2) && (pix_y[9:6]==2);
   assign row_addr_r = pix_y[3:0];
   assign bit_addr_r = pix_x[2:0];
   assign rule_rom_addr = {pix_y[5:4], pix_x[6:3]};
   always @*
      case (rule_rom_addr)
         // row 1
         6'h00: char_addr_r = 7'h52; // R
         6'h01: char_addr_r = 7'h55; // U
         6'h02: char_addr_r = 7'h4c; // L
         6'h03: char_addr_r = 7'h45; // E
         6'h04: char_addr_r = 7'h3a; // :
         6'h05: char_addr_r = 7'h00; //
         6'h06: char_addr_r = 7'h00; //
         6'h07: char_addr_r = 7'h00; //
         6'h08: char_addr_r = 7'h00; //
         6'h09: char_addr_r = 7'h00; //
         6'h0a: char_addr_r = 7'h00; //
         6'h0b: char_addr_r = 7'h00; //
         6'h0c: char_addr_r = 7'h00; //
         6'h0d: char_addr_r = 7'h00; //
         6'h0e: char_addr_r = 7'h00; //
         6'h0f: char_addr_r = 7'h00; //
         // row 2
         6'h10: char_addr_r = 7'h55; // U
```

```
            6'h11: char_addr_r = 7'h73; // s
            6'h12: char_addr_r = 7'h65; // e
            6'h13: char_addr_r = 7'h00; //
            6'h14: char_addr_r = 7'h74; // t
            6'h15: char_addr_r = 7'h77; // w
            6'h16: char_addr_r = 7'h6f; // o
            6'h17: char_addr_r = 7'h00; //
            6'h18: char_addr_r = 7'h62; // b
            6'h19: char_addr_r = 7'h75; // u
            6'h1a: char_addr_r = 7'h74; // t
            6'h1b: char_addr_r = 7'h74; // t
            6'h1c: char_addr_r = 7'h6f; // o
            6'h1d: char_addr_r = 7'h6e; // n
            6'h1e: char_addr_r = 7'h73; // s
            6'h1f: char_addr_r = 7'h00; //
            // row 3
            6'h20: char_addr_r = 7'h74; // t
            6'h21: char_addr_r = 7'h6f; // o
            6'h22: char_addr_r = 7'h00; //
            6'h23: char_addr_r = 7'h6d; // m
            6'h24: char_addr_r = 7'h6f; // o
            6'h25: char_addr_r = 7'h76; // v
            6'h26: char_addr_r = 7'h65; // e
            6'h27: char_addr_r = 7'h00; //
            6'h28: char_addr_r = 7'h70; // p
            6'h29: char_addr_r = 7'h61; // a
            6'h2a: char_addr_r = 7'h64; // d
            6'h2b: char_addr_r = 7'h64; // d
            6'h2c: char_addr_r = 7'h6c; // l
            6'h2d: char_addr_r = 7'h65; // e
            6'h2e: char_addr_r = 7'h00; //
            6'h2f: char_addr_r = 7'h00; //
            // row 4
            6'h30: char_addr_r = 7'h75; // u
            6'h31: char_addr_r = 7'h70; // p
            6'h32: char_addr_r = 7'h00; //
            6'h33: char_addr_r = 7'h61; // a
            6'h34: char_addr_r = 7'h6e; // n
            6'h35: char_addr_r = 7'h64; // d
            6'h36: char_addr_r = 7'h00; //
            6'h37: char_addr_r = 7'h64; // d
            6'h38: char_addr_r = 7'h6f; // o
            6'h39: char_addr_r = 7'h77; // w
            6'h3a: char_addr_r = 7'h6e; // n
            6'h3b: char_addr_r = 7'h2e; // .
            6'h3c: char_addr_r = 7'h00; //
            6'h3d: char_addr_r = 7'h00; //
            6'h3e: char_addr_r = 7'h00; //
            6'h3f: char_addr_r = 7'h00; //
        endcase
    //----------------------------------------------------------------
    // game over region
    //  - display "Game Over" at center
```

```
   // - scale to 32-by-64 fonts
160 //------------------------------------------------------------
   assign over_on = (pix_y[9:6]==3) &&
                    (5<=pix_x[9:5]) && (pix_x[9:5]<=13);
   assign row_addr_o = pix_y[5:2];
   assign bit_addr_o = pix_x[4:2];
165 always @*
      case(pix_x[8:5])
         4'h5: char_addr_o = 7'h47; // G
         4'h6: char_addr_o = 7'h61; // a
         4'h7: char_addr_o = 7'h6d; // m
170      4'h8: char_addr_o = 7'h65; // e
         4'h9: char_addr_o = 7'h00; //
         4'ha: char_addr_o = 7'h4f; // O
         4'hb: char_addr_o = 7'h76; // v
         4'hc: char_addr_o = 7'h65; // e
175      default: char_addr_o = 7'h72; // r
      endcase
   //------------------------------------------------------------
   // mux for font ROM addresses and rgb
   //------------------------------------------------------------
180 always @*
   begin
      text_rgb = 3'b110;  // background, yellow
      if (score_on)
         begin
185         char_addr = char_addr_s;
            row_addr = row_addr_s;
            bit_addr = bit_addr_s;
            if (font_bit)
               text_rgb = 3'b001;
190      end
      else if (rule_on)
         begin
            char_addr = char_addr_r;
            row_addr = row_addr_r;
195         bit_addr = bit_addr_r;
            if (font_bit)
               text_rgb = 3'b001;
         end
      else if (logo_on)
200      begin
            char_addr = char_addr_l;
            row_addr = row_addr_l;
            bit_addr = bit_addr_l;
            if (font_bit)
205            text_rgb = 3'b011;
         end
      else // game over
         begin
            char_addr = char_addr_o;
210         row_addr = row_addr_o;
            bit_addr = bit_addr_o;
```

```
            if (font_bit)
               text_rgb = 3'b001;
         end
   end

   assign text_on = {score_on, logo_on, rule_on, over_on};
   //-----------------------------------------------------
   // font rom interface
   //-----------------------------------------------------
   assign rom_addr = {char_addr, row_addr};
   assign font_bit = font_word[~bit_addr];

endmodule
```

The structure of each segment is similar. Because the messages are short, they are coded with the regular ROM template. Since no clock signal is used, a distributed RAM or combinational logic should be inferred. Generation of the two-digit score depends on the two 4-bit external signals, `dig0` and `dig1`. Note that the ASCII codes for the digits 0, 1, ..., 9, are 30_{16}, 31_{16}, ..., 39_{16}. We can generate the `char_addr` signal simply by concatenating `"011"` in front of `dig0` and `dig1`.

14.4.2 Modified graphic subsystem

To accommodate the new top-level controller, the graphic circuit in Section 13.4.3 requires several modifications:

- Add a `gra_still` (for "still graphics") control signal. When it is asserted, the vertical bar is placed in the middle and the ball is placed at the center of the screen without movement.
- Add the `hit` and `miss` status signals. The `hit` signal is asserted for one clock cycle when the paddle hits the ball. The `miss` signal is asserted when the paddle misses the ball and the ball reaches the right border.
- Add a `graph_on` signal to indicate the on status of the graph subsystem.

The modified portion of the code is shown in Listing 14.7.

Listing 14.7 Modified portion of a graph subsystem for the pong game

```
   . . .
   // new ball position
   assign ball_x_next = (gra_still) ? MAX_X/2 :
                        (refr_tick) ? ball_x_reg+x_delta_reg :
                        ball_x_reg ;
   assign ball_y_next = (gra_still) ? MAX_Y/2 :
                        (refr_tick) ? ball_y_reg+y_delta_reg :
                        ball_y_reg ;
   // new ball velocity
   always @*
   begin
      hit = 1'b0;
      miss = 1'b0;
      x_delta_next = x_delta_reg;
      y_delta_next = y_delta_reg;
      if (gra_still)     // initial velocity
```

```
            begin
               x_delta_next = BALL_V_N;
               y_delta_next = BALL_V_P;
            end
         else if (ball_y_t < 1) // reach top
            y_delta_next = BALL_V_P;
         else if (ball_y_b > (MAX_Y-1)) // reach bottom
            y_delta_next = BALL_V_N;
         else if (ball_x_l <= WALL_X_R) // reach wall
            x_delta_next = BALL_V_P;     // bounce back
         else if ((BAR_X_L<=ball_x_r) && (ball_x_r<=BAR_X_R) &&
                  (bar_y_t<=ball_y_b) && (ball_y_t<=bar_y_b))
            begin
               // reach x of right bar and hit, ball bounce back
               x_delta_next = BALL_V_N;
               hit = 1'b1;
            end
         else if (ball_x_r>MAX_X)    // reach right border
            miss = 1'b1;             // a miss
      end
      . . .
      assign graph_on = wall_on | bar_on | rd_ball_on;
      . . .
```

14.4.3 Auxiliary counters

The top-level design requires two small utility modules, `m100_counter` and `timer`, to facilitate the counting. The `m100_counter` module is a two-digit decade counter that counts from 00 to 99 and is used to keep track of the scores of the game. Two control signals, `d_inc` and `d_clr`, increment and clear the counter, respectively. The code is shown in Listing 14.8.

Listing 14.8 Two-digit decade counter

```
module m100_counter
   (
    input wire clk, reset,
    input wire d_inc, d_clr,
    output wire [3:0] dig0, dig1
   );

   // signal declaration
   reg [3:0] dig0_reg, dig1_reg, dig0_next, dig1_next;

   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            dig1_reg <= 0;
            dig0_reg <= 0;
         end
      else
         begin
```

```
            dig1_reg <= dig1_next;
            dig0_reg <= dig0_next;
         end

   // next-state logic
   always @*
   begin
      dig0_next = dig0_reg;
      dig1_next = dig1_reg;
      if (d_clr)
         begin
            dig0_next = 0;
            dig1_next = 0;
         end
      else if (d_inc)
         if (dig0_reg==9)
            begin
               dig0_next = 0;
               if (dig1_reg==9)
                  dig1_next = 0;
               else
                  dig1_next = dig1_reg + 1;
             end
         else  // dig0 not 9
            dig0_next = dig0_reg + 1;
   end
   // output
   assign dig0 = dig0_reg;
   assign dig1 = dig1_reg;

endmodule
```

The `timer` module uses the 60-Hz tick, `timer_tick`, to generate a 2-second interval. Its purpose is to pause the video for a small interval between transitions of the screens. It starts counting when the `timer_start` signal is asserted and activates the `timer_up` signal when the 2-second interval is up. The code is shown in Listing 14.9.

Listing 14.9 Two-second timer

```
module timer
   (
    input wire clk, reset,
    input wire timer_start, timer_tick,
    output wire timer_up
   );

   // signal declaration
   reg [6:0] timer_reg, timer_next;

   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         timer_reg <= 7'b1111111;
      else
```

```
      timer_reg <= timer_next;

   // next-state logic
   always @*
      if (timer_start)
         timer_next = 7'b1111111;
      else if ((timer_tick) && (timer_reg != 0))
         timer_next = timer_reg - 1;
      else
         timer_next = timer_reg;
   // output
   assign timer_up = (timer_reg==0);

endmodule
```

14.4.4 Top-level system

The top-level system of the pong game consists of the previously designed modules, including a video synchronization circuit, graphic subsystem, text subsystem, and utility counters, as well as a control FSM and an rgb multiplexing circuit. The block diagram is shown in Figure 14.4.

The control FSM monitors overall system operation and coordinates the activities of the text and graphic subsystems. Its ASMD chart is shown in Figure 14.6. The FSM has four states and operates as follows:

- Initially, the FSM is in the newgame state. The game starts when a button is pressed and the FSM moves to the play state.
- In the play state, the FSM checks the hit and miss signals continuously. When the hit signal is activated, the d_inc signal is asserted for one clock cycle to increment the score counter. When the miss signal is asserted, the FSM activates the 2-second timer, decrements the number of the balls by 1, and examines the number of remaining balls. If it is zero, the game is ended and the FSM moves to the over state. Otherwise, the FSM moves to the newball state.
- The FSM waits in the newball state until the 2-second interval is up (i.e., when the timer_up signal is asserted) and a button is pressed. It then moves to the play state to continue the game.
- The FSM stays in the over state until the 2-second interval is up. It then moves to the newgame state for a new game.

The rgb multiplexing circuit routes the text_rgb or graph_rgb signals to output according to the text_on and graphic_on signals. The key segment is

```
always @*
   if (~video_on)
      rgb_next = "000"; // blank the edge/retrace
   else
      // display score, rule, or game over
      if ( text_on[3] ||
           ((state_reg==newgame) && text_on[1]) ||
           ((state_reg==over) && text_on[0]) )
         rgb_next = text_rgb;
      else if (graph_on)  // display graph
```

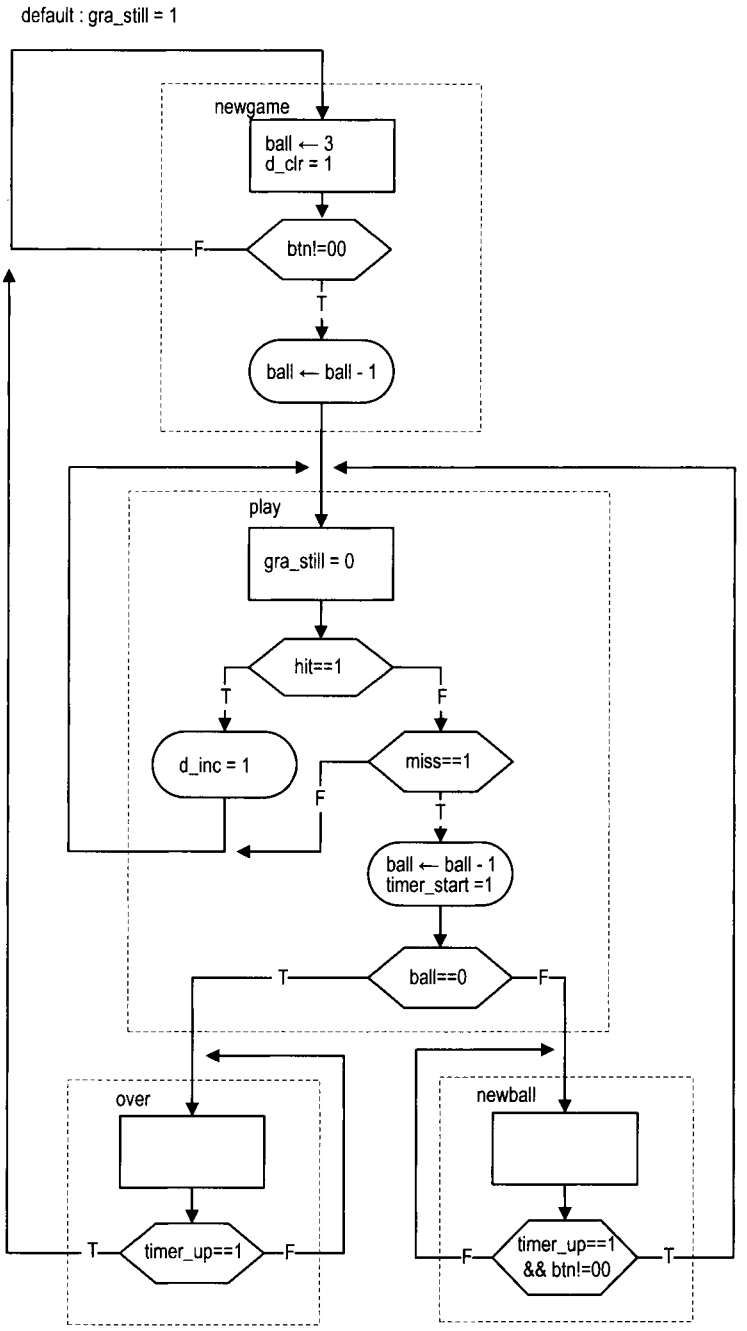


Figure 14.6 ASMD chart of the pong controller.

```
            rgb_next = graph_rgb;
         else if (text_on[2]) // display logo
            rgb_next = text_rgb;
         else
            rgb_next = 3'b110; // yellow background
   // output
   assign rgb = rgb_reg;
```

The text_on[3] signal is for the scores, which is always displayed. The text_on[1] signal is for the rule, which is displayed only when the FSM is in the newgame state. Similarly, the end-of-game message, whose status is indicated by the text_on[0] signal, is displayed only when the FSM is in the over state. The logo, whose status is indicated by the text_on[2] signal, is used as part of the background and is displayed only when no other on signal is asserted.

The complete code is shown in Listing 14.10.

Listing 14.10 Top-level system for the pong game

```
module pong_top
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // symbolic state declaration
   localparam  [1:0]
      newgame = 2'b00,
      play    = 2'b01,
      newball = 2'b10,
      over    = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick, graph_on, hit, miss;
   wire [3:0] text_on;
   wire [2:0] graph_rgb, text_rgb;
   reg [2:0] rgb_reg, rgb_next;
   wire [3:0] dig0, dig1;
   reg gra_still, d_inc, d_clr, timer_start;
   wire timer_tick, timer_up;
   reg [1:0] ball_reg, ball_next;

   //=============================================================
   // instantiation
   //=============================================================
   // instantiate video synchronization unit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));
   // instantiate text module
```

```
pong_text text_unit
   (.clk(clk),
    .pix_x(pixel_x), .pix_y(pixel_y),
    .dig0(dig0), .dig1(dig1), .ball(ball_reg),
    .text_on(text_on), .text_rgb(text_rgb));
// instantiate graph module
pong_graph graph_unit
   (.clk(clk), .reset(reset), .btn(btn),
    .pix_x(pixel_x), .pix_y(pixel_y),
    .gra_still(gra_still), .hit(hit), .miss(miss),
    .graph_on(graph_on), .graph_rgb(graph_rgb));
// instantiate 2 sec timer
// 60 Hz tick
assign timer_tick = (pixel_x==0) && (pixel_y==0);
timer timer_unit
   (.clk(clk), .reset(reset), .timer_tick(timer_tick),
    .timer_start(timer_start), .timer_up(timer_up));
// instantiate 2-digit decade counter
m100_counter counter_unit
   (.clk(clk), .reset(reset), .d_inc(d_inc), .d_clr(d_clr),
    .dig0(dig0), .dig1(dig1));
//=============================================================
// FSMD
//=============================================================
// FSMD state & data registers
 always @(posedge clk, posedge reset)
    if (reset)
       begin
          state_reg <= newgame;
          ball_reg <= 0;
          rgb_reg <= 0;
       end
    else
       begin
         state_reg <= state_next;
         ball_reg <= ball_next;
         if (pixel_tick)
            rgb_reg <= rgb_next;
       end
// FSMD next-state logic
always @*
begin
   gra_still = 1'b1;
   timer_start = 1'b0;
   d_inc = 1'b0;
   d_clr = 1'b0;
   state_next = state_reg;
   ball_next = ball_reg;
   case (state_reg)
      newgame:
         begin
            ball_next = 2'b11; // three balls
            d_clr = 1'b1;      // clear score
```

```
            if (btn != 2'b00)  // button pressed
               begin
                  state_next = play;
                  ball_next = ball_reg - 1;
               end
         end
      play:
         begin
            gra_still = 1'b0;  // animated screen
            if (hit)
               d_inc = 1'b1;   // increment score
            else if (miss)
               begin
                  if (ball_reg==0)
                     state_next = over;
                  else
                     state_next = newball;
                  timer_start = 1'b1;  // 2 sec timer
                  ball_next = ball_reg - 1;
               end
         end
      newball:
         // wait for 2 sec and until button pressed
         if (timer_up && (btn != 2'b00))
            state_next = play;
      over:
         // wait for 2 sec to display game over
         if (timer_up)
            state_next = newgame;
    endcase
  end
//=============================================================
// rgb multiplexing circuit
//=============================================================
always @*
   if (~video_on)
      rgb_next = "000"; // blank the edge/retrace
   else
      // display score, rule, or game over
      if (text_on[3] ||
            ((state_reg==newgame) && text_on[1]) || // rule
            ((state_reg==over) && text_on[0]))
         rgb_next = text_rgb;
      else if (graph_on)  // display graph
        rgb_next = graph_rgb;
      else if (text_on[2]) // display logo
        rgb_next = text_rgb;
      else
        rgb_next = 3'b110; // yellow background
// output
assign rgb = rgb_reg;
endmodule
```

14.5 BIBLIOGRAPHIC NOTES

Several other character fonts are available. *Rapid Prototyping of Digital Systems* by James O. Hamblen et al. uses a compact 64-character 8-by-8 font set. The tile-mapped scheme is not limited to the text display. It is widely used in the early video game. The article "Computer Graphics During the 8-Bit Computer Game Era" by Steven Collins (*ACM SIGGRAPH*, May 1998) provides a comprehensive review of the history and design techniques of the tile-based game.

14.6 SUGGESTED EXPERIMENTS

14.6.1 Rotating banner

A rotating banner on the monitor screen moves a line from right to left and then wraps around. It is similar to the Window's Marquee screen saver. Let the text on the banner be "Hello, FPGA World." The banner should be displayed in four different font sizes and can travel at four different speeds. The font size and speed are controlled by four switches. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.2 Underline for the cursor

The full-screen text display circuit in Section 14.3 uses reversed color to indicate the current cursor location. Modify the design to use an underline to indicate the cursor location. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.3 Dual-mode text display

It is sometimes better for text to be displayed on a "vertical" screen. This can be done by turning the monitor 90 degrees and resting it on its side. Design this circuit as follows:

1. Modify the full-screen text display circuit in Section 14.3 for a vertical screen.
2. Merge the normal and vertical designs to create a "dual-mode" text display. Use a switch to select the desired mode.
3. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.4 Keyboard text entry

Instead of switches and buttons, it is more natural to use a keyboard to enter text. We can use the four arrow keys to move the cursor and use the regular keys to enter the characters. Use the keyboard interface discussed in Section 9.4 to design the new circuit. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.5 UART terminal

The UART terminal receives input from the UART port and displays the received characters on a monitor. When connected to the PC's serial port, it should echo the text on Window's HypterTerminal. The detailed specifications are:

- A cursor is used to indicate the current location.
- The screen starts a new line when a "carriage return" code ($0d_{16}$) is received.

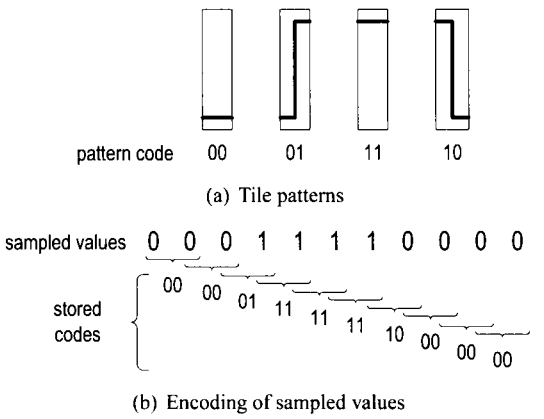


Figure 14.7 Tile patterns and encoding of a square wave.

- A line wraps around (i.e., starts a new line) after 80 characters.
- When the cursor reaches the bottom of the screen (i.e., the last line), the first line will be discarded and all other lines move up (i.e., scroll up) one position.

Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.6 Square-wave display

We can draw a square wave by using the four simple tile patterns shown in Figure 14.7(a). Follow the procedure of a full-screen text display in Section 14.3 to design a full-screen wave editor:

1. Let the tile size be 8 columns by 64 rows. Create a pattern ROM for the four patterns.
2. Calculate the number of tiles on a 640-by-480 resolution screen and derive the proper configuration for the tile memory.
3. Use three pushbuttons for control and a 2-bit switch to enter the pattern.
4. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.7 Simple four-trace logic analyzer

A logic analyzer displays the waveforms of a collection of digital signals. We want to design a simple logic analyzer that captures the waveforms of four input signals in "free-running" mode. Instead of using a trigger pattern, data capture is initiated with activation of a pushbutton switch. For simplicity, we assume that the frequencies of the input waveform are between 10 kHz and 100 kHz. The circuit can be designed as follows:

1. Use a sampling tick to sample the four input signals. Make sure to select a proper rate so that the desired input frequency range can be displayed properly on the screen.
2. For a point in the sampled signal, its value can be encoded as a tile pattern by including the value of the previous point. For example, if the sampled sequence of one signal is "00001111000", the tile patterns become "00 00 00 01 11 11 11 10 00 00", as shown in Figure 14.7(b).
3. Follow the procedure of the preceding square-wave experiment to design the tile memory and video interface to display the four waveforms being stored .
4. Derive the HDL description and then synthesize the circuit.

To verify operation of the circuit, we can connect four external signals via headers around the prototyping board. Alternatively, we can create a top-level test module that includes a 4-bit counter (say, a mod-10 counter around 50 kHz) and the logic analyzer, resynthesize the circuit, and verify its operation.

14.6.8 Complete two-player pong game

The free-running two-player pong game is described in Experiment 13.7.6. Follow the procedure of the pong game in Section 14.4 to derive the complete system. This should include the design of a new text display subsystem and the design of a top-level FSM controller. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.9 Complete breakout game

The free-running breakout game is described in Experiment 13.7.7. Follow the procedure of the pong game in Section 14.4 to derive the complete system. This should include the design of a new text display subsystem and the design of a top-level FSM controller. Derive the HDL description and then synthesize and verify operation of the circuit.

PART III

PICOBLAZE MICROCONTROLLER *XILINX SPECIFIC*

CHAPTER 15

PICOBLAZE OVERVIEW

15.1 INTRODUCTION

The *PicoBlaze* processor is a compact 8-bit microcontroller core for Xilinx FPGA devices. It is provided as a cell-level HDL description (which is known as *soft core*) and can be synthesized along with other logic. PicoBlaze is optimized for efficiency and occupies only about 200 logic cells, which amount to less than 5% of the resources of a 3S200 device. While not intended as a high-performance processor, it is compact and flexible and can be used for simple data processing and control, particularly for non-time-critical "housekeeping" and I/O operations. The PicoBlaze processor can easily be integrated into a larger system and adds another dimension of flexibility in an FPGA-based design.

Although the detailed coverage of assembly language programming and microcontrollers is beyond the scope of this book, this part provides a comprehensive overview of PicoBlaze's organization and instruction set, and illustrates the general assembly program development and I/O interface through a set of examples. We review PicoBlaze's organization and instruction set in this chapter, introduce assembly language programming in Chapter 16, and discuss the general I/O interface and interrupt interface in Chapters 17 and 18.

FPGA Prototyping by Verilog Examples. By Pong P. Chu

15.2 CUSTOMIZED HARDWARE AND CUSTOMIZED SOFTWARE

15.2.1 From special-purpose FSMD to general-purpose microcontroller

The RT-level design and FSMD discussed in Chapter 6 provide a general methodology to convert a sequential algorithm to customized hardware. The rearranged block diagram is shown in Figure 15.1(a). In an FSMD, all components, including the number of registers, the routing of registers' input and output, the number and types of functional units, and the control FSM, are tailored to the target application. The data path may contain multiple function units and multiple routing paths, as shown in the diagram.

An alternative is to keep the same hardware but use *customized software* for different applications. The transformation can be done as follows. First, we can replace the customized data path with a fixed configuration, as shown at the top of Figure 15.1(b). The data registers and customized routing networks are replaced by a register file, which has a fixed number of registers and contains only two read ports and one write port. The customized function units are replaced with an *ALU* (arithmetic and logic unit), which can only perform a set of predefined functions. The data path can now perform RT operations in the following format only:

$$\texttt{rd} \leftarrow \texttt{r1 op r2}$$

where `r1`, `r2`, and `rd` are the addresses of two source registers and one destination register, and `op` is one of the available ALU functions.

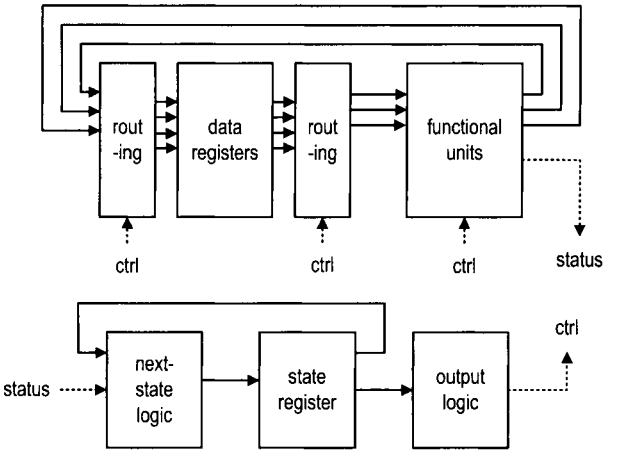
Second, we can replace the customized FSM with a *programmable state machine*, as shown at the bottom of Figure 15.1(b). Recall that operation of an FSM consists of three parts:

- The state register keeps track of the current state.
- The output logic activates certain output signals according to the current state.
- The next-state logic determines the new state.

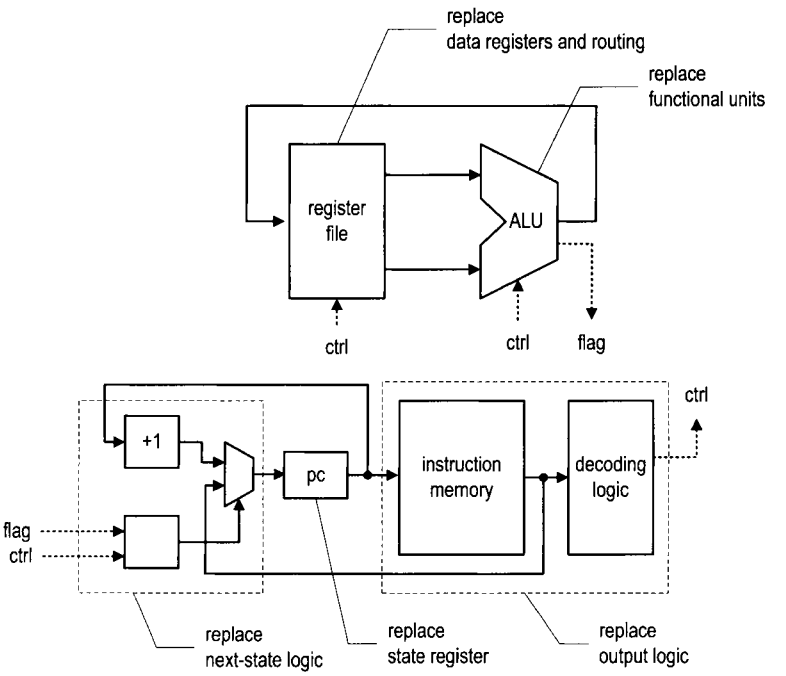
The programmable state machine modifies these operations as follows:

- It replaces the state register with the *program counter*. The content of the program counter represents the current state of the control path.
- In an FSM, each state activates certain output signals to control operation of the data path. The programmable state machine encodes these output patterns into *instructions* and stores them in a memory module, known as *program memory* or *instruction memory*. A memory address corresponds to a state (i.e., a value) of the program counter. During execution, the instruction pointed to by the program counter is retrieved from memory and decoded to generate the control signals. The instruction memory and decoding logic function as a sophisticated output logic circuit.
- In an FSM, there is no limitation on where to go next. From a given state, the FSM can check the input condition and move to one of many possible next states. In a programmable state machine, the next state is usually the value of the current state plus 1 (i.e., the program counter is incremented by 1), which reflects the nature of the sequential execution. The sequential execution may be altered only by several special instructions, such as a *jump* instruction, in which the program counter is loaded with a different value. The incrementor and associated multiplexing logic function as a simple next-state logic circuit.

After we replace the data path with a register file and an ALU and replace the dedicated FSM with a programmable state machine, customizing the system corresponds to developing a new sequence of instructions (i.e., developing a *software program*) and loads the



(a) Block diagram of an FSMD



(b) Simplified block diagram of a microcontroller

Figure 15.1 Diagrams of an FSMD and a microcontroller.

instructions to the instruction memory. The organization of the FSMD is now the same for different applications and becomes a *general-purpose* hardware platform. The platform constitutes the basic skeleton of the PicoBlaze microcontroller.

15.2.2 Application of microcontroller

In a customized FSMD, the data path can be created to accommodate an individual application's needs. It may contain multiple customized functional units and parallel routing paths, and can complete complex computation in a single state (i.e., one clock cycle). On the other hand, the PicoBlaze microcontroller can perform only one predefined RT operation (i.e., an instruction) at a time. It may need many instructions to perform the same task and thus require much more time.

Many tasks can be carried out using either a customized FSMD or a microcontroller. The trade-off is between the hardware complexity, performance, and ease of development. There is no exact rule on which one to choose. Because developing software is usually easier than creating customized hardware, the microcontroller option is generally preferable for non-time-critical applications. We can determine the feasibility of this option by examining the computation complexity. PicoBlaze requires two clock cycles to complete an instruction. If the system clock is 50 MHz, 25 million instructions can be performed in 1 second. For a task (or a collection of tasks), we can examine how frequently a request is issued and how fast the task must be completed, and then estimate the number of available instructions. For example, assume that a keyboard interface generates new input data every 1 ms and the data must be processed within this interval. Within the 1-ms period, PicoBlaze can complete 25,000 instructions. The PicoBlaze controller will be a viable option if the required processing can be carried out by using fewer than 25,000 instructions. In general, the microcontroller is suitable for many non-time-critical I/O-interface or housekeeping tasks.

15.3 OVERVIEW OF PICOBLAZE

15.3.1 Basic organization

PicoBlaze is a compact 8-bit microcontroller with the following characteristics:

- 8-bit data width
- 8-bit ALU with carry and zero flags
- 16 8-bit general-purpose registers
- 64-byte data memory
- 18-bit instruction width
- 10-bit instruction address, which supports a program up to 1024 instructions
- 31-word call/return stack
- 256 input ports and 256 output ports
- 2 clock cycles per instruction
- 5 clock cycles for interrupt handling

PicoBlaze is based on the skeleton described in Figure 15.1(b) and adds several enhancements to make it more versatile. The expanded diagram is shown in Figure 15.2. To reduce clutter, only the main data flow is shown. The sizes of main storage components are listed in round brackets. The processor makes several enhancements over the original skeleton:

- *Add a 64-word data memory.* This is known as *scratch RAM* in the Xilinx literature, but we call it *data RAM.* The data RAM can be considered as a reservoir to store

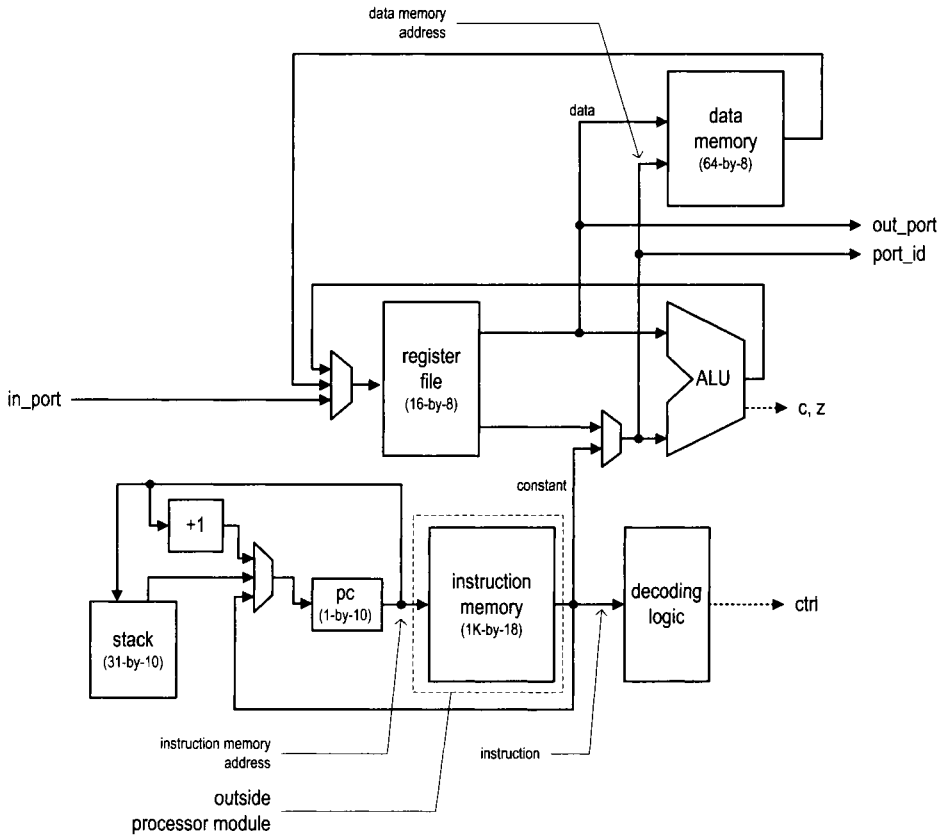


Figure 15.2 Block diagram of PicoBlaze.

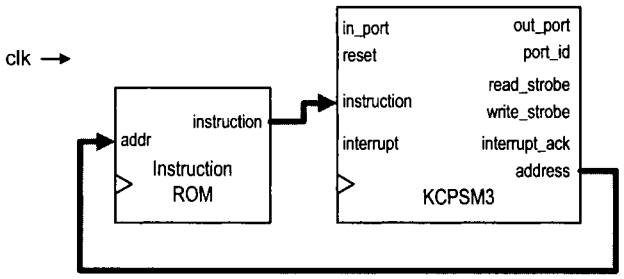


Figure 15.3 Top-level diagram of PicoBlaze.

additional data. Note that there is no direct path between the data RAM and the ALU. Data must be fetched to a register for processing and then stored back to the data RAM.

- *Add an immediate constant field in some instructions.* This allows a constant, rather than the content of a register, to be used in ALU and other operations. The two-to-one multiplexer before the ALU's bottom input is used to select the register output or the constant field.
- *Add a 31-word stack to support a function call.* We discuss the call and return procedure in more detail in Section 15.5.8.
- *Add paths to input and output external data.* An 8-bit `port_id` signal is used to identify a port and thus up to 256 input ports and 256 output ports can be supported. The I/O interface is discussed in detail in Chapter 17.
- *Add an interrupt-handling circuit* (not shown in the diagram). The interrupt mechanism is discussed in detail in Chapter 18.

15.3.2 Top-level HDL modules

During synthesis, a PicoBlaze system is organized as two top-level HDL modules, as shown in Figure 15.3. The `KCPSM3` module is the PicoBlaze processor. `KCPSM3`, which stands for *constant (K) coded programmable state machine*, reflects the original name of the PicoBlaze processor. It has the following input and output signals:

- `clk` (input, 1 bit): system clock signal
- `reset` (input, 1 bit): reset signal
- `address` (output, 10 bits): address of the instruction memory, which specifies the location of the instruction to be retrieved
- `instruction` (input, 18 bits): fetched instruction
- `port_id` (output, 8 bits): address of the input or output port
- `in_port` (input, 8 bits): input data from I/O peripherals
- `read_storbe` (output, 1 bit): strobe associated with the input operation
- `out_port` (output, 8 bits): output data to I/O peripherals
- `write_storbe` (output, 1 bit): strobe associated with the output operation
- `interrupt` (input, 1 bit): interrupt request from I/O peripherals
- `interrupt_ack` (output, 1 bit): interrupt acknowledgment to I/O peripherals

The second module is for the instruction memory. During the development, we usually store the compiled assembly code to memory in advance and configure it as a ROM in HDL code. It is thus known as an *instruction ROM*.

15.4 DEVELOPMENT FLOW

While developing a system based on a conventional microcontroller, we examine the required functionalities and select a processor with the proper computation capability and adequate I/O interface. Additional chips are frequently needed to perform special functions. One advantage of using a soft-core microcontroller is that we can have both a customized circuit and a microcontroller developed and implemented in the same FPGA device. A large application usually includes many different tasks. In an FPGA platform, we can implement the time-critical tasks in a customized circuit (i.e., "hardware") for performance and realize the remaining housekeeping and low-speed I/O functions in a microcontroller (i.e., "software").

The basic PicoBlaze-based development flow is shown in Figure 15.4. It consists of the following steps:

1. Determine the software–hardware partition.
2. Develop the assembly program for the software portion.
3. Compile the assembly program to generate an instruction ROM. The ROM is an HDL file.
4. Perform instruction-set-level simulation.
5. Derive HDL code for the hardware portion. The hardware includes customized circuits to perform special I/O and time-critical functions and customized circuits to interface with PicoBlaze.
6. Create top-level HDL code that combines the codes for the PicoBlaze core, the instruction ROM, and customized hardware.
7. Develop a testbench and perform HDL simulation for the entire system.
8. Synthesize and implement the HDL code and program the FPGA chip on the prototyping board.

We explain these steps in detail in subsequent chapters.

Step 9, shown in the dotted line, is not a part of the normal development flow. It reloads the instruction memory after the entire system is synthesized. This step is discussed in Section 16.5.3.

15.5 INSTRUCTION SET

PicoBlaze has 57 instructions. The instructions have five general formats. We organize the instructions according to the nature of their operations and divide them into the following categories:

- Logical instructions
- Arithmetic instructions
- Compare and test instructions
- Shift and rotate instructions
- Data movement instructions
- Program flow control instructions
- Interrupt related instructions

In this section, we first examine the program model and instruction format and then list and explain each instruction.

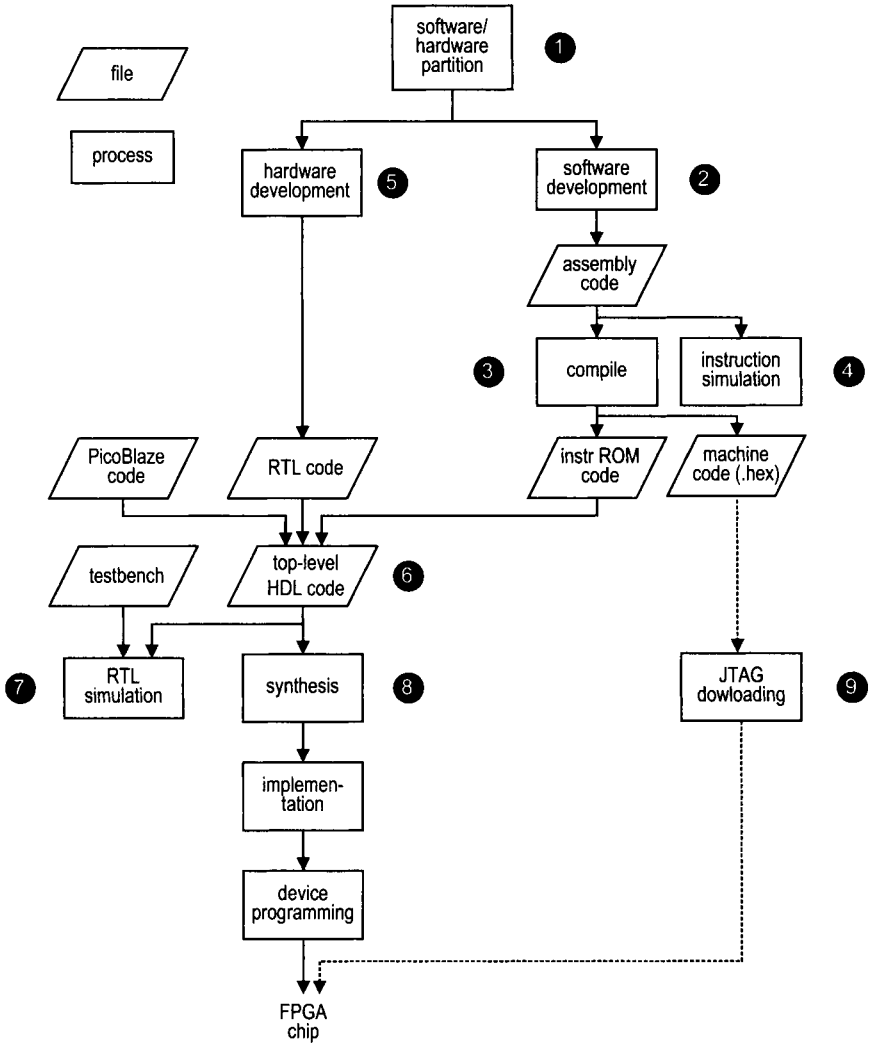


Figure 15.4 Development flow of a system with PicoBlaze.

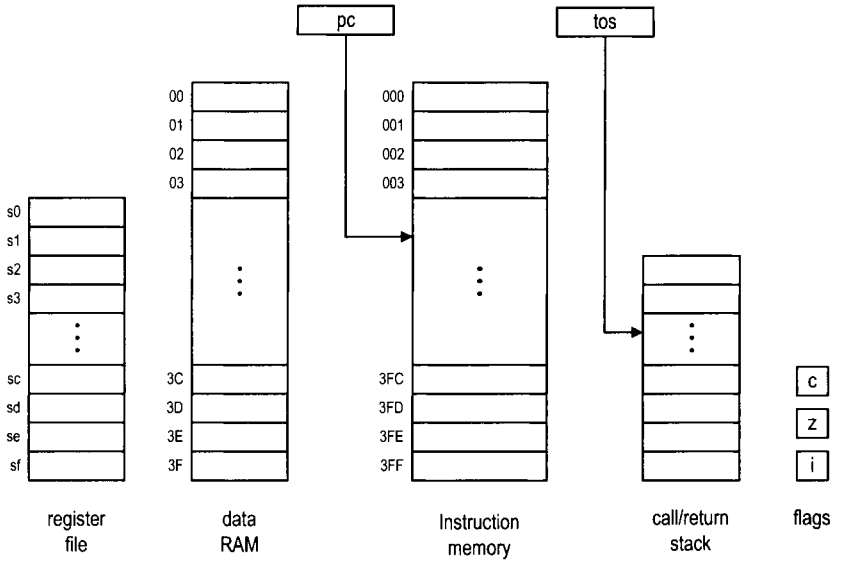


Figure 15.5 PicoBlaze programming model.

15.5.1 Programming model

From an assembly programming point of view, PicoBlaze contains sixteen 8-bit registers, a 64-byte data RAM, three flags (for zero, carry, and interrupt), the program counter, and the top-of-stack pointer. The model, sometimes known as the *instruction set architecture*, is shown in Figure 15.5. After an instruction is executed, the contents of these components are modified explicitly or implicitly. The operations associated with each instruction are discussed in Section 15.5.3.

We use the following notations for these memory components and some constant definitions:

- `sX`, `sY`: each representing one of the 16 general-purpose registers, where `X` and `Y` take on hexadecimal values from `0` to `f`
- `pc`: program counter
- `tos`: top-of-stack pointer of the call/return stack
- `c`, `z`, `i`: carry, zero, and interrupt flags
- `KK`: 8-bit constant value or port id, which is usually expressed as two hexadecimal digits
- `SS`: 6-bit constant data memory address, which is usually expressed as two hexadecimal digits
- `AAA`: 10-bit constant instruction memory address, which is usually expressed as three hexadecimal digits

15.5.2 Instruction format

In an assembly program, we generally follow the conventions used in our HDL code, in which a keyword (an instruction mnemonic) is in boldface type and a constant is in capital letters. PicoBalze's instructions have five formats:

- **op** sX, sY: *register-register format.* The **op** term specifies the operation. The sX and sY terms are the two operands and sX also serves as the destination register. It performs the sX ← sX **op** sY operation.
- **op** sX, KK: *register-constant format.* This format is similar to the register-register format except that the second operand is replaced by an immediate constant. It performs the sX ← sX **op** KK operation.
- **op** sX: *single-register format.* This format is used in shift and rotate instructions, which involve only one operand. It performs the sX ← **op** sX operation.
- **op** AAA: *single-address format.* This format is used in jump and call instructions. The AAA term is an address of the instruction memory. If the specified condition is met, AAA is loaded into the program counter.
- **op**: *zero-operand format.* This format is used in some miscellaneous instructions that do not involve any operand.

There are two assembler programs for PicoBlaze: *KCPSM3* from Xilinx and *PBlazeIDE* from Mediatronix. The two programs use different mnemonics for several instructions. In the following subsections, the alternative mnemonics used in PBlazeIDE are shown in round brackets.

15.5.3 Logical instructions

There are six logical instructions, which support the and, or, and xor operations. An instruction performs bitwise logical operation between two registers or between one register and a constant. The carry flag, c, is always cleared. The zero flag, z, reflects the result of the operation. The mnemonics, brief descriptions, and pseudo operations of these instructions are:

- **and** sX, sY
  - bitwise and operation
  - pseudo operation:

```
sX ← sX & sY;
c ← 0;
```

- **and** sX, KK
  - bitwise and operation
  - pseudo operation:

```
sX ← sX & KK;
c ← 0;
```

- **or** sX, sY
  - bitwise or operation
  - pseudo operation:

```
sX ← sX | sY;
c ← 0;
```

- **or** sX, KK
  - bitwise or operation
  - pseudo operation:

```
sX ← sX | KK;
c ← 0;
```

- **xor** sX, sY
  - bitwise xor operation
  - pseudo operation:

```
sX ← sX ^ sY;
c ← 0;
```

- **xor** sX, KK
  - bitwise xor operation
  - pseudo operation:

```
sX ← sX ^ KK;
c ← 0;
```

15.5.4 Arithmetic instructions

There are eight arithmetic instructions, which support addition and subtraction with or without the carry flag. The carry flag, c, and the zero flag, z, reflect the result of operation. The mnemonics, brief descriptions, and pseudo operations of these instructions are:

- **add** sX, sY
  - add without the carry flag
  - pseudo operation:

```
sX ← sX + sY;
```

- **add** sX, KK
  - add without the carry flag
  - pseudo operation:

```
sX ← sX + KK;
```

- **addcy** sX, sY (**addc** sX, sY)
  - add with the carry flag
  - pseudo operation:

```
sX ← sX + sY + c;
```

- **addcy** sX, KK (**addc** sX, KK)
  - add with the carry flag
  - pseudo operation:

```
sX ← sX + KK + c;
```

- **sub** sX, sY
  - subtract without the carry flag
  - pseudo operation:

```
sX ← sX - sY;
```

- **sub** sX, KK
  - subtract without the carry flag
  - pseudo operation:

```
sX ← sX - KK;
```

- **subcy** sX, sY (**subc** sX, sY)
  - subtract with the carry flag (flag functioning as a borrow bit)
  - pseudo operation:

```
sX ← sX - sY - c;
```

- **subcy** sX, KK (**subc** sX, KK)
  - subtract with the carry flag (flag functioning as a borrow bit)
  - pseudo operation:

```
sX ← sX - KK - c;
```

15.5.5 Compare and test instructions

The compare and test instructions examine two registers or one register and a constant, and set the carry and zero flags accordingly. The contents of the registers remain intact. These instructions are usually used in conjunction with a conditional jump or call instruction, whose operation is based on the values of the flags.

A compare instruction performs subtraction operation. The result is used to set the carry and zero flags and not stored to any register. The mnemonics, brief descriptions, and pseudo operations of the two instructions are:

- **compare** sX, sY (**comp** sX, sY)
  - compare two registers and set the flags
  - pseudo operation:

```
if sX==sY then z ← 1 else z ← 0;
if sY>sX then c ← 1 else c ← 0;
```

- **compare** sX, KK (**comp** sX, KK)
  - compare a register and a constant and set the flags
  - pseudo operation:

```
if sX==KK then z ← 1 else z ← 0;
if KK>sX then c ← 1 else c ← 0;
```

A test instruction performs an and operation. The result is used to set the flags and is not stored in any register. If the result is 0, the zero flag is set to 1. The result is also fed to an eight-input xor circuit to obtain the odd parity. If there are an odd number of 1's in the result, the carry flag is set to 1. The mnemonics, brief descriptions, and pseudo operations of the two instructions are shown below. The t is the 8-bit temporary result and will be discarded.

- **test** sX, sY
  - test two registers and set the flags
  - pseudo operation:

```
t ← sX & sY;
if t==0 then z ← 1 else z ← 0;
c ← t[7] ^ t[6] ^ ··· ^ t[0];
```

- **test** sX, KK
  - test a register and a constant and set the flags
  - pseudo operation:

```
t ← sX & KK;
if t==0 then z ← 1 else z ← 0;
c ← t[7] ^ t[6] ^ ··· ^ t[0];
```

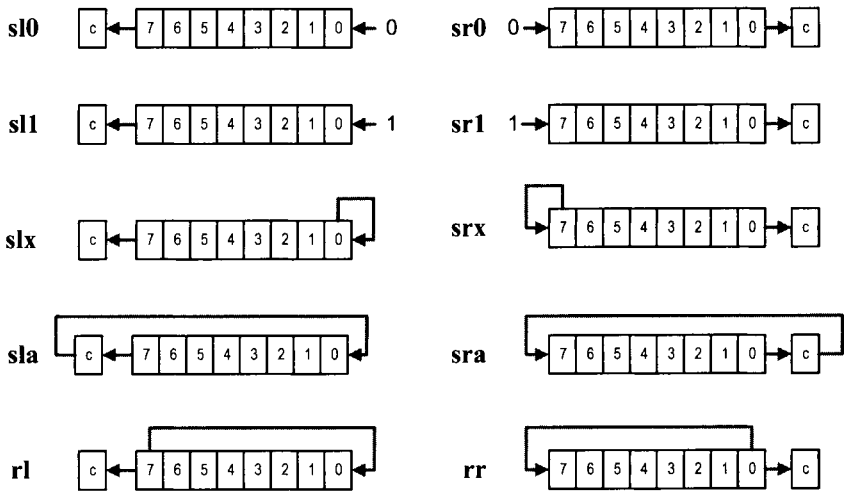


Figure 15.6 Illustration of shift and rotate instructions.

15.5.6 Shift and rotate instructions

There are four shift-left instructions, four shift-right instructions, and two rotate instructions. These instructions use the single-register format and have only one operand. The graphical representations of these instructions are shown in Figure 15.6. The mnemonics, brief descriptions, and pseudo operations of these instructions are:

- **sl0** sX
  - shift a register left 1 bit and shift 0 into the LSB
  - pseudo operation:

```
sX ← {sX[6:0], 0};
c ← sX[7];
```

- **sl1** sX
  - shift a register left 1 bit and shift 1 into the LSB
  - pseudo operation:

```
sX ← {sX[6:0], 1};
c ← sX[7];
```

- **slx** sX
  - shift a register left 1 bit and shift sX[0] into the LSB
  - pseudo operation:

```
sX ← {sX[6:0], sX[0]};
c ← sX[7];
```

- **sla** sX
  - shift a register left 1 bit and shift c into the LSB
  - pseudo operation:

```
sX ← {sX[6:0], c};
c ← sX[7];
```

- **sr0** sX
  - shift a register right 1 bit and shift 0 into the MSB
  - pseudo operation:
    ```
    sX ← {0, sX[7:1]};
    c ← sX[0];
    ```
- **sr1** sX
  - shift a register right 1 bit and shift 1 into the MSB
  - pseudo operation:
    ```
    sX ← {1, sX[7:1]};
    c ← sX[0];
    ```
- **srx** sX
  - shift a register right 1 bit and shift sX[7] into the MSB
  - pseudo operation:
    ```
    sX ← {sX[7], sX[7:1]};
    c ← sX[0];
    ```
- **sra** sX
  - shift a register right 1 bit and shift c into the MSB
  - pseudo operation:
    ```
    sX ← {c, sX[7:1]};
    c ← sX[0];
    ```
- **rl** sX
  - rotate a register left 1 bit
  - pseudo operation:
    ```
    sX ← {sX[6:0], sX[7]};
    c ← sX[7];
    ```
- **rr** sX
  - rotate a register right 1 bit
  - pseudo operation:
    ```
    sX ← {sX[0], sX[7:1]};
    c ← sX[0];
    ```

15.5.7 Data movement instructions

In PicoBlaze, the computation is done via the registers and ALU. The data RAM supplies additional storage and the I/O ports provide paths to peripherals. There are several instructions to move data between the registers, data RAM, and I/O ports. The instructions can be divided into three categories:

- *Between registers*: the **load** instruction
- *Between a register and data RAM*: the **fetch** and **store** instructions
- *Between a register and an I/O port*: the **input** and **output** instructions

The mnemonics, brief descriptions, and pseudo operations of the data movement instructions are shown below. The RAM[ ] notation represents the content of the data RAM. Note that in some instructions, the *indirect address* notation, as in (sY), is used in the mnemonic to emphasize that the content of the sY register is used.

- **load** sX, sY
  - move data between two registers
  - pseudo operation:

```
sX ← sY;
```

- **load** sX, KK
  - move a constant to a register
  - pseudo operation:

```
sX ← KK;
```

- **fetch** sX, (sY) (**fetch** sX, sY)
  - move data from the data RAM to a register
  - pseudo operation:

```
sX ← RAM[(sY)];
```

- **fetch** sX, SS
  - move data from the data RAM to a register
  - pseudo operation:

```
sX ← RAM[SS];
```

- **store** sX, (sY) (**store** sX, sY)
  - move data from a register to the data RAM
  - pseudo operation:

```
RAM[(sY)] ← sX;
```

- **store** sX, SS
  - move data from a register to the data RAM
  - pseudo operation:

```
RAM[SS] ← sX;
```

- **input** sX, (sY) (**in** sX, sY)
  - move data from the input port to a register
  - pseudo operation:

```
port_id ← sY;
sX ← in_port;
```

- **input** sX, KK (**in** sX, KK)
  - move data from the input port to a register
  - pseudo operation:

```
port_id ← KK;
sX ← in_port;
```

- **output** sX, (sY) (**out** sX, sY)
  - move data from a register to the output port
  - pseudo operation:

```
port_id ← sY;
out_port ← sX;
```

- **output** sX, KK (**out** sX, KK)
  - move data from a register to the output port
  - pseudo operation:

```
port_id ← KK;
out_port ← sX;
```

There is no explicit instruction to move data to or from instruction memory. However, many instructions include a field for an immediate constant. Since the constant is part of the instruction and stored in the instruction memory, it can be considered as data that is moved implicitly from the instruction memory to a register.

15.5.8 Program flow control instructions

In PicoBlaze, the program counter indicates where to fetch the instruction. By default, the execution proceeds to the next address in the instruction memory and the program counter is incremented implicitly (i.e., `pc ← pc + 1`). The **jump**, **call**, and **return** instructions can explicitly load a value to the program counter and modify the program flow. These instructions can be executed unconditionally or conditionally based on the values of the carry and zero flags.

A **jump** instruction loads a new value to the program counter if the corresponding condition is met. The program execution changes the regular flow and branches to the new address. The program flow continues normally after this point. The mnemonics, brief descriptions, and pseudo operations of these instructions are shown below. Recall that `AAA` is for the 10-bit instruction memory address and `pc` is for the program counter.

- **jump** `AAA`
  - unconditionally jump
  - pseudo operation:

    ```
    pc ← AAA;
    ```

- **jump c**, `AAA`
  - jump if the carry flag is set
  - pseudo operation:

    ```
    if c==1 then pc ← AAA else pc ← pc + 1;
    ```

- **jump nc**, `AAA`
  - jump if the carry flag is not set
  - pseudo operation:

    ```
    if c==0 then pc ← AAA else pc ← pc + 1;
    ```

- **jump z**, `AAA`
  - jump if the zero flag is set
  - pseudo operation:

    ```
    if z==1 then pc ← AAA else pc ← pc + 1;
    ```

- **jump nz**, `AAA`
  - jump if the zero flag is not set
  - pseudo operation:

    ```
    if z==0 then pc ← AAA else pc ← pc + 1;
    ```

The **call** and **return** instructions are used to implement a software function. When a function is *called*, the processor suspends the current execution and branches to the corresponding routine. When the routine computation is completed, the processor *returns* to the suspended point and continues the execution. Like a **jump** instruction, a **call** instruction loads a new value to the program counter if the corresponding condition is met. In addition, it also saves the current value of the program counter in a special buffer, known as the *stack*. The new address represents the starting point of a routine. The routine should include a **return** instruction in the end. The **return** instruction obtains the saved value from the

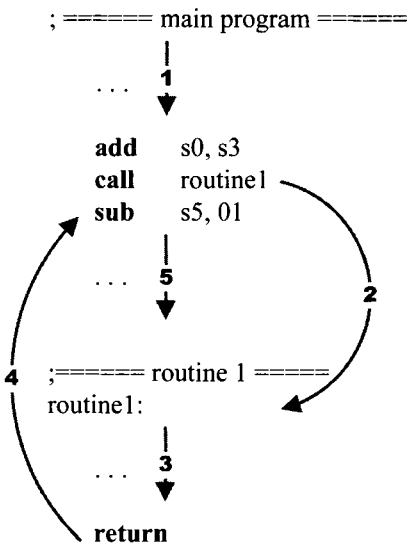


Figure 15.7 Representative flow of a subroutine call.

stack, increments the value by 1, and loads it to the program counter. This allows the execution to return to the instruction that immediately follows the original **call** instruction. A representative program flow is shown in Figure 15.7.

PicoBlaze allows nested function calls, which means that a function can be called within another function. To support this feature, a stack, which is a *last-in-first-out* buffer, is used to store the program counter's values. In this buffer, the address of the newest call is pushed to the top of the stack (i.e., the "last-in"). Assume that this routine does not contain other function call inside. It will be completed first and the saved returned address is on the top of the stack. It should be popped from the stack (i.e., "first-out") to resume the previous execution. PicoBlaze provides a 31-word stack for the nested call and return operations.

The mnemonics, brief descriptions, and pseudo operations of the **call** and **return** instructions are shown below. Recall that `tos` is for the top-of-stack pointer. The `STACK[ ]` notation represents the content of the stack.

- **call** `AAA`
  - unconditional call subroutine
  - pseudo operation:

```
tos ← tos + 1;
STACK[tos] ← pc;
pc ← AAA;
```

- **call c**, `AAA`
  - call subroutine if the carry flag is set
  - pseudo operation:

```
if c==1 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
```

```
pc ← pc + 1;
```

- **call nc**, AAA
  - call subroutine if the carry flag is not set
  - pseudo operation:

```
if c==0 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
   pc ← pc + 1;
```

- **call z**, AAA
  - call subroutine if the zero flag is set
  - pseudo operation:

```
if z==1 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
   pc ← pc + 1;
```

- **call nz**, AAA
  - call subroutine if the zero flag is not set
  - pseudo operation:

```
if z==0 then
   tos ← tos + 1;
   STACK[tos] ← pc;
   pc ← AAA;
else
   pc ← pc + 1;
```

- **return (ret)**
  - unconditional return
  - pseudo operation:

```
pc ← STACK[tos] + 1;
tos ← tos - 1;
```

- **return c (ret c)**
  - return if the carry flag is set
  - pseudo operation:

```
if c==1 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
else
   pc ← pc + 1;
```

- **return nc (ret nc)**
  - return if the carry flag is not set
  - pseudo operation:

```
if c==0 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
```

```
else
   pc ← pc + 1;
```

- **return z (ret z)**
  - return if the zero flag is set
  - pseudo operation:

```
if z==1 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
else
   pc ← pc + 1;
```

- **return nz (ret nz)**
  - return if the zero flag is not set
  - pseudo operation:

```
if z==0 then
   pc ← STACK[tos] + 1;
   tos ← tos - 1;
else
   pc ← pc + 1;
```

15.5.9 Interrupt related instructions

Interrupt is another mechanism to alter program execution and its detail is discussed in Chapter 18. Unlike the **jump** and **call** instructions, it is initiated from an external request. When the interrupt flag is enabled and the interrupt request is asserted, PicoBlaze completes execution of the current instruction, saves the address of the next instruction in the call/return stack, preserves the carry and zero flags, disables the interrupt flag, and loads the program counter with 3FF, which is the starting address of the interrupt service routine. PicoBlaze has two return-from-interrupt instructions, which resume operation from the interrupted location. It also has two instructions that enable and disable the interrupt request by setting or clearing the interrupt flag, `i`. The mnemonics, brief descriptions, and pseudo operations of these instructions are:

- **returni disable (reti disable)**
  - return from interrupt service routine and keep the interrupt flag disabled
  - pseudo operation:

```
pc ← STACK[tos];
tos ← tos - 1;
i ← 0;
c ← preserved c;
z ← preserved z;
```

- **returni enable (reti enable)**
  - return from interrupt service routine and keep the interrupt flag enabled
  - pseudo operation:

```
pc ← STACK[tos];
tos ← tos - 1;
i ← 1;
c ← preserved c;
z ← preserved z;
```

- **enable interrupt (eint)**
  - enable interrupt request
  - pseudo operation:
    ```
    i ← 1;
    ```
- **disable interrupt (dint)**
  - disable interrupt request
  - pseudo operation:
    ```
    i ← 0;
    ```

Note that the interrupt mechanism saves the address of the next instruction. When a **returni** instruction is executed, the address saved on the top of the stack (i.e., `STACK[tos]`) is restored. This is different from a regular **return** instruction, in which the incremented address (i.e., `STACK[tos]+1`) is restored.

15.6 ASSEMBLER DIRECTIVES

An *assembler directive* looks like an instruction in an assembly program. However, it is not part of the microcontroller's instruction set but is used to help program development. As its name suggests, a directive "directs" the assembler to perform a specific task, such as defining a constant or reserving data space. The KCPSM3 and PBlazeIDE assemblers have somewhat different directives and they are discussed in the following subsections.

15.6.1 The KCPSM3 directives

The mnemonics, descriptions, and examples of key directives used in the KCPSM3 assembler are:

- **address**
  - The directive specifies the subsequent code to be put to a specific address in the instruction ROM.
  - Example:
    ```
    address  3FF
    ```
- **namereg**
  - The directive gives a symbolic name for a register. It makes code more descriptive.
  - Example:
    ```
    namereg  s5, index
    ```
- **constant**
  - The directive gives a symbolic name for a constant. It makes code more descriptive.
  - Example:
    ```
    constant  max, F0
    ```

15.6.2 The PBlazeIDE directives

The mnemonics, descriptions, and examples of key directives used in the PBlazeIDE assembler are shown below. Note that a $ sign is needed for a number in hexadecimal format.

- **org**
  - The directive specifies the subsequent code to be put to a specific address in the instruction ROM (i.e., "originate" from this address).
  - Example:

```
org   $3FF
```

- **equ**
  - The directive "equates" a symbol to a value or register. It gives a symbolic name for a constant or a register.
  - Example:

```
max     equ    128/8
index   equ    s5
```

- **dsin, dsout, dsio**
  - These directives equate a symbolic name for an I/O port id. The corresponding port can be defined as input, output, or both input and output. The difference between these directives and **equ** is that PBlazeIDE generates "port indicators" for these directives on the simulation screen. The I/O activities can be displayed and simulated via these indicators.
  - Example:

```
keyboard   dsin   $0E
switch     dsin   $0F
led        dsout  $15
```

- **vhdl**
  - This directive generates instruction ROM in VHDL format. The details are discussed in Chapter 16.
  - Example:

```
vhdl "template.vhd", "target.vhd", "ROM"
```

15.7 BIBLIOGRAPHIC NOTES

The PicoBlaze manual from Xilinx, *PicoBlaze 8-Bit Embedded Microcontroller User Guide*, provides detailed information about this microcontroller, including the hardware organization, instruction set, development process, and KCPSM3 and PBlazeIDE assemblers. Ken Chapman, the designer of PicoBlaze, describes the derivation of this microcontroller in the article "Creating Embedded Microcontrollers," which is available in the *TechXclusives* section of the Xilinx Web site.

The KCPSM3 assembler, PicoBlaze HDL code, and instruction ROM HDL template can be downloaded from the Xilinx Web site. Searching with the keyword "PicoBlaze" will lead to the downloading page. The PBlazeIDE assembler can be downloaded from the Mediatronix Web site, `http://www.mediatronix.com`. The site also provides more detailed information about the software.

CHAPTER 16

PICOBLAZE ASSEMBLY CODE DEVELOPMENT

16.1 INTRODUCTION

Because of its simplicity, PicoBlaze cannot effectively support high-level programming languages and the code is generally developed in assembly language. In this chapter, we provide an overview of code development, which is illustrated in a bottom-up fashion. We first introduce the segments of frequently used data and control operations and then examine the use of a subroutine and finally outline the derivation of overall program structure.

16.2 USEFUL CODE SEGMENTS

The PicoBlaze microcontroller contains instructions for byte-oriented data manipulation and simple conditional branch. In this section, we illustrate how to construct code to perform bit and multiple-byte operations and to realize frequently used high-level language control constructs.

16.2.1 KCPSM3 conventions

The KCPSM3 assembler uses the following conventions in an assembly program:

- Use a ":" sign after a symbolic address in code, as in "`done:`".
- Use a ";" sign before a comment.
- Use `HH` for a constant, in which `H` is a hexadecimal digit.

An example of a code segment follows:

```
    ;this is a demo segment
    test s0, 82          ;compare s0 with 1000_0010
    jump z, clr_s1       ;if MSB of s0 is 0, go to clr_s1
    load s1, FF          ;no, load 1111_1111 to s1
clr_s1:
    load s1, 01          ;load 0000_0001 to s1
```

16.2.2 Bit manipulation

PicoBlaze's instruction set is primarily for byte-oriented operations. Bit-oriented operations are frequently needed to control low-level I/O activities, such as testing, setting, and clearing a 1-bit flag signal.

To manipulate a single bit, we first define a *mask* to isolate and preserve (i.e., mask) the unrelated bits and then apply the designated operation on the desired bits (i.e., unmasked bits). We can set, clear, and toggle (i.e., invert) some bits of a data byte by performing **or**, **and**, and **xor** instructions with a proper mask. The following code segment shows how to set, clear, and toggle the second LSB of the s0 register:

```
    constant SET_MASK, 02    ;mask=0000_0010
    constant CLR_MASK, FD    ;mask=1111_1101
    constant TOG_MASK, 02    ;mask=0000_0010

    or  s0, SET_MASK         ;set 2nd LSB to 1
    and s0, CLR_MASK         ;clear 2nd LSB to 0
    xor s0, TOG_MASK         ;toggle 2nd LSB
```

The toggle operation is based on the observation that for any Boolean variable x, $x \oplus 0 = x$ and $x \oplus 1 = x'$. The same principle can be applied to multiple bits. For example, we can clear the upper nibble (i.e., four MSBs) by using

```
    and s0, 0F               ;mask=0000_1111
```

We can also apply the concept of the and mask to the **test** instruction to check a single bit. For example, the following code segment tests the MSB of the s0 register and branches to a proper routine accordingly:

```
    test s0, 80          ;mask=1000_0000
    jump nz, msb_set     ;MSB is 1, branch to msb_set
    ;code for MSB not set
    jump  done
msb_set:
    ;code for MSB set
    ...
done:
    ...
```

A single bit can be extracted by applying the previous code. For example, the following code segment extracts the MSB of the s0 register and stores it in the s1 register:

```
    load s1, 00
    test s0, 80          ;mask=1000_0000, extract MSB
    jump z, done         ;yes, MSB is 0
    load s1, 01          ;no, load 1 to s1
```

```
done:
    ...
```

16.2.3 Multiple-byte manipulation

A microcontroller sometimes needs to handle wide, multiple-byte data, such as a large counter. Since the data width of PicoBlaze is 8 bits, processing this type of data requires a mechanism to propagate information between two successive instructions. PicoBlaze uses the carry flag for this purpose. For the arithmetic instructions, there are two versions for addition and subtraction, one with carry and one without carry, as in the **add** and **addcy** instructions. For the shift and rotate instructions, carry can be shifted into the MSB or LSB of a register, and vice versa.

Assume that x and y are 24-bit data and that each occupies three registers. The following code segment illustrates the use of carry in multiple-byte addition:

```
namereg s0, x0   ;least significant byte of x
namereg s1, x1   ;middle byte of x
namereg s2, x2   ;most significant byte of x
namereg s3, y0   ;least significant byte of y
namereg s4, y1   ;middle byte of y
namereg s5, y2   ;most significant byte of y

;add: {x2,x1,x0} + {y2,y1,y0}
add x0, y0       ;add least significant bytes
addcy x1, y1     ;add middle bytes with carry
addcy x2, y2     ;add most significant bytes with carry
```

The first instruction performs normal addition of the least significant bytes and stores the carry-out bit into the carry flag. The second instruction then includes the carry flag when adding the middle bytes. Similarly, the third instruction uses the carry flag from the previous addition to obtain the result for the most significant bytes.

The incrementing and subtraction of multiple bytes can be achieved in a similar fashion:

```
;increment: {x2,x1,x0} + 1
add x0, 01       ;inc least significant byte
addcy x1, 00     ;add carry to middle byte
addcy x2, 00     ;add carry to most significant byte

;subtract: {x2,x1,x0} - {y2,y1,y0}
sub x0, y0       ;sub least significant byte
subcy x1, y1     ;sub middle byte with borrow
subcy x2, y2     ;sub most significant byte with borrow
```

Multiple-byte data can be shifted by including the carry flag in the individual shift instruction. For example, the **sla** instruction shifts data left one position and shifts the carry flag into LSB. The code for shifting a 3-byte data left can be written as

```
;shift {x2,x1,x0} via carry
sl0 x0           ;0 to LSB of x0, MSB of x0 to carry
sla x1           ;carry to LSB of x1, MSB of x1 to carry
sla x2           ;carry to LSB of x2, MSB of x2 to carry
```

16.2.4 Control structure

A high-level programming language usually contains various control constructs to alter the execution sequence. These include the if-then-else, case, and for-loop statements. On the other hand, PicoBlaze provides only simple conditional and unconditional **jump** instructions. Despite its simplicity, we can use them with a **test** or **compare** instruction to implement the high-level control constructs. The following examples illustrate the construction of the if-then-else, case, and for-loop statements.

Let us first consider the if-then-else statement:

```
if (s0==s1) {
   /* then-branch statements */
}
else {
   /* else-branch statements */
}
```

The corresponding assembly code segment is

```
   compare s0, s1
   jump nz, else_branch
   ;code for then branch
   ...
   jump if_done
else_branch:
   ;code for else branch
   ...
if_done:
   ;code following if statement
   ...
```

The code uses the **compare** instruction to check the `s0==s1` condition and to set the zero flag. The following **jump** instruction examines the flag and jumps to the else branch if the flag is not set.

The case statement can be considered as a multiway jump, in which execution is transferred according to the value of the selection expression. The following statement uses the `s0` variable as the selection expression and jumps to the corresponding branch:

```
switch (s0) {
   case value1:
      /* case value1 statements */
      break;
   case value2:
      /* case value2 statements */
      break;
   case value3:
      /* case value3 statements */
      break;
   default:
      /* default statements */
}
```

The multiway jump can be implemented by a hardware feature known as "index address mode" in some processors. However, since PicoBlaze does not support this feature, the case statement has to be constructed as a sequence of if-then-else statements. In other words, the previous case statement is treated as

```
if (s0==value1) {
   /* case value1 statements */
}
else if (s0==value2) {
   /* case value2 statements */
}
else if (s0==value3) {
   /* case value3 statements */
}
else{
   /* default statements */
}
```

The corresponding assembly code segment becomes

```
constant value1, ...
constant value2, ...
constant value3, ...

compare s0, value1      ;test value1
jump nz, case_2         ;not equal to value1, jump
;code for case 1
...
jump case_done
case_2:
compare s0, value2      ;test value2
jump nz, case_3         ;not equal to value2, jump
;code for case 2
...
jump case_done
case_3:
compare s0, value3      ;test value3
jump default            ;not equal to value3, jump
;code for case 3
...
jump case_done
default:
;code for default case
...
case_done:
;code following case statement
...
```

The for-loop statement executes a segment of the code repetitively. The loop statement can be implemented by using a counter to keep track of the iteration number. For example, consider the following:

```
for(i=MAX, i=0, i-1) {
   /* loop body statements */
}
```

The assembly code segment is

```
namereg s0, i           ;loop index
constant MAX, ...       ;loop boundary
```

```
    load i, MAX             ;load loop index
loop_body:
    ;code for loop body
    ...
    sub i, 01               ;dec loop index?
    jump nz, loop_body      ;done?
    ;code following for loop
    ...
```

16.3 SUBROUTINE DEVELOPMENT

A subroutine, such as a function in C, implements a section of a larger program. It is coded to perform a specific task and can be used repetitively. Using subroutines allows us to divide a program into small, manageable parts and thus greatly improve the reliability and readability of a program. It is the base of modern programming practice and is supported by all high-level programming languages.

PicoBlaze uses the **call** and **return** instructions to implement the subroutine. The **call** instruction saves the current content of the program counter and transfers program execution to the starting address of a subroutine. A subroutine ends with a **return** instruction, which restores the saved program counter and resumes the previous execution. A representative flow is shown in Figure 15.7. Note that PicoBlaze only saves and restores the content of the program counter during a function call and return. We have to manage the register and data RAM use manually to ensure that the original system state is not altered after a subroutine call.

The following multiplication example illustrates the development of subroutines. We assume that the inputs are two 8-bit numbers in unsigned integer format and the output is a 16-bit product. The algorithm is based on a simple shift-and-add method. This method iterates through 8 bits of multiplier. In each iteration, the multiplicand is shifted left one position. If the corresponding multiplier bit is 1, the shifted multiplicand is added to the partial product. The assembly code is shown in Listing 16.1. The multiplicand and multiplier are stored in the s3 and s4 registers. The individual bit of multiplier is obtained by repetitively shifting s4 to the right, which moves the LSB to the carry flag. Note that instead of actually shifting the multiplicand to the left, we shift the partial product, which consists of 2 bytes and is stored in s5 and s6, to the right.

Listing 16.1 Software integer multiplication

```
;=====================================================
;routine: mult_soft
;  function: 8-bit unsigned multiplier using
;            shift-and-add algorithm
;  input register:
;     s3: multiplicand
;     s4: multiplier
;  output register:
;     s5: upper byte of product
;     s6: lower byte of product
;  temp register: i
;=====================================================
mult_soft:
    load s5, 00             ;clear s5
```

```
    load i, 08              ;initialize loop index
mult_loop:
    sr0 s4                  ;shift LSB to carry
    jump nc, shift_prod     ;LSB is 0
    add s5, s3              ;LSB is 1
shift_prod:
    sra s5                  ;shift upper byte right,
                            ;carry to MSB, LSB to carry
    sra s6                  ;shift lower byte right,
                            ;LSB of s5 to MSB of s6
    sub i, 01               ;dec loop index
    jump nz, mult_loop      ;repeat until i=0
    return
```

Because of the primitive nature of the assembly language, thorough documentation is instrumental. A subroutine should include a descriptive header and detailed comments. A representative header is shown in Listing 16.1. It consists of a short function description and the use of registers. The latter shows how the registers are allocated and is crucial to preventing conflict in a large program.

16.4 PROGRAM DEVELOPMENT

Developing a complete assembly program consists of the following steps:

1. Derive the pseudo code of the *main program.*
2. Identify tasks in the main program and define them as subroutines. If needed, continue refining the complex subroutines and divide them into smaller routines.
3. Determine the register and data RAM use.
4. Derive assembly code for the subroutines.

Steps 1, 2, and 4 basically follow a *divide-and-conquer* approach and are applicable for any software development. A microcontroller-based application is normally for a simple embedded system, in which the processor monitors the I/O activities continuously and responds accordingly. Its main program usually has the following structure:

```
    call initialization_routine
forever:
    call task1_routine
    call task2_routine
    ...
    call taskn_routine
    jump forever
```

Step 3 is unique for assembly code development. Unlike a high-level language program, in which the compiler allocates storage to variables automatically, we must manage the data storage manually in assembly code. PicoBlaze has 16 registers and 64 bytes of data RAM to store data. The registers can be considered as fast storage, in which the data can be manipulated directly. The data RAM, on the other hand, is "auxiliary" storage. Its data needs to be transferred to a register for processing. For example, if we want to increment a data item located in the RAM, it must first be loaded into a register, incremented there, and then stored back to the RAM.

Because of the limited space for data storage, its use has to be planned carefully in advance, particularly when the code is complex and involves nested subroutines. To assist

00	lower byte of a
01	unused
02	lower byte of b
03	unused
04	lower byte of a^2
05	upper byte of a^2
06	lower byte of b^2
07	upper byte of b^2
08	lower byte of $a^2 + b^2$
09	upper byte of $a^2 + b^2$
0A	carry of $a^2 + b^2$

Figure 16.1 Data RAM memory allocation.

coding, we can first identify the needed *global storage* or *local storage*. The former keeps data that is needed in the entire program. The latter provides space to store intermediate results, and the data will be discarded after the required computation is completed.

16.4.1 Demonstration example

The development process can best be explained by an example. Let us consider a program that uses the previous multiplication subroutine. It reads two inputs, a and b, from the switch, calculates $a^2 + b^2$, and displays the result on eight discrete LEDs. Since the I/O interface is to be discussed in Chapter 17, we limit the I/O to a single input port, the 8-bit switch, and a single output port, the 8-bit LEDs. We assume that a and b are obtained from the upper nibble (i.e., the four MSBs) and the lower nibble (i.e., the four LSBs) of the switch. The main program is

```
    call clear_data_ram
forever:
    call read_switch
    call square
    call write_led
    jump forever
```

The subroutines are defined as follows:

- `clr_data_mem`: clears data memory at system initialization
- `read_switch`: obtains the two nibbles from the switch and stores their values to the data RAM
- `square`: uses the multiplication subroutine to calculate $a^2 + b^2$
- `write_led`: writes the eight LSBs of the calculated result to the LED port

For demonstration purposes, we create two smaller routines, `get_upper_nibble` and `get_lower_nibble`, within the `read_switch` routine to obtain the upper nibble and lower nibble from a register.

The next step in development is to plan the register and data RAM use. For global storage, we introduce a global register, `sw_in`, to store the input value of switch and allocate 11 bytes of data RAM to store the inputs and result of the `square` routine. Allocation of the data RAM is shown in Figure 16.1. Note that the addresses 01 and 03 are not actually used. They are reserved to simplify the seven-segment LED display code, which is discussed in Chapter 17. All remaining registers are used as local storage. For program clarity, we

define three symbolic names, `data`, `addr`, and `i`, as temporary registers for data, port and memory address, and loop index.

The last step is to derive the assembly code for the subroutines. The complete code is shown in Listing 16.2. The `clr_data_mem` uses a loop to clear data memory. The `i` register is the loop index and is initialized with 64 (i.e., 40_{16}). The index is decremented in each loop and 0 is loaded to the corresponding data RAM address. The `write_led` routine fetches the eight LSBs of the calculated result from the data RAM and outputs them to the LED port.

The `read_switch` routine includes two smaller routines. The `get_upper_nibble` routine shifts the `data` register right four times to move the upper nibble to the four LSBs. The `get_lowe_nibble` routine clears the four MSBs of the `data` register to 0's and thus removes the upper nibble. The "glue instructions" of `read_switch` input the switch values, set up the input for the two nibble routines, and store the result in the data RAM.

The `square` routine fetches data from the data RAM, utilizes the `mult_soft` routine to calculate a^2 and b^2, performs addition, and stores the result back to the data RAM.

Listing 16.2 Square program with simple nibble input

```
;=============================================================
; square circuit with simple I/O interface
;=============================================================
;program operation:
;  - read switch to a (4 MSBs) and b (4 LSBs)
;  - calculate a*a + b*b
;  - display data on 8 leds

;=============================================================
; data constant
;=============================================================
constant UP_NIBBLE_MASK, 0F   ;00001111

;=============================================================
; data ram address alias
;=============================================================
constant a_lsb, 00
constant b_lsb, 02
constant aa_lsb, 04
constant aa_msb, 05
constant bb_lsb, 06
constant bb_msb, 07
constant aabb_lsb, 08
constant aabb_msb, 09
constant aabb_cout, 0A

;=============================================================
; register alias
;=============================================================
;commonly used local variables
namereg s0, data   ;reg for temporary data
namereg s1, addr   ;reg for temporary mem & i/o port addr
namereg s2, i      ;general-purpose loop index
;global variables
namereg sf, sw_in
```

```
;=================================================================
; port alias
;=================================================================
;--------------input port definitions-------------------------
constant sw_port, 01 ;8-bit switches
;--------------output port definitions------------------------
constant led_port, 05

;=================================================================
; main program
;=================================================================
;calling hierarchy:
;
;main
;  - clr_data_mem
;  - read_switch
;      - get_upper_nibble
;      - get_lower_nibble
;  - square
;      - mult_soft
;  - write_led
;

    call clr_data_mem
forever:
    call read_switch
    call square
    call write_led
    jump forever

;=================================================================
;routine: clr_data_mem
;  function: clear data ram
;  temp register: data, i
;=================================================================
clr_data_mem:
    load i, 40              ;unitize loop index to 64
    load data, 00
clr_mem_loop:
    store data, (i)
    sub i, 01               ;dec loop index
    jump nz, clr_mem_loop   ;repeat until i=0
    return

;=================================================================
;routine: read switch
;  function: obtain two nibbles from input
;  input register: sw_in
;  temp register: data
;=================================================================
read_switch:
    input sw_in, sw_port    ;read switch input
```

```
    load data, sw_in
    call get_lower_nibble
    store data, a_lsb       ;store a to data ram
    load data, sw_in
    call get_upper_nibble
    store data, b_lsb       ;store b to data ram

;=============================================================
;routine: get_lower_nibble
;  function: get lower 4 bits of data
;  input register: data
;  output register: data
;=============================================================
get_lower_nibble:
    and data, UP_NIBBLE_MASK ;clear upper nibble
    return

;=============================================================
;routine: get_upper_nibble
;  function: get upper 4 bits of data
;  input register: data
;  output register: data
;=============================================================
get_upper_nibble:
    sr0 data                ;right shift 4 times
    sr0 data
    sr0 data
    sr0 data
    return

;=============================================================
;routine: write_led
;  function: output 8 LSBs of result to 8 leds
;  temp register: data
;=============================================================
write_led:
    fetch data, aabb_lsb
    output data, led_port
    return

;=============================================================
;routine: square
;  function: calculate a*a + b*b
;     data/result stored in ram started w/ SQ_BASE_ADDR
;  temp register: s3, s4, s5, s6, data
;=============================================================
square:
    ;calculate a*a
    fetch s3, a_lsb         ;load a
    fetch s4, a_lsb         ;load a
    call mult_soft          ;calculate a*a
    store s6, aa_lsb        ;store lower byte of a*a
    store s5, aa_msb        ;store upper byte of a*a
```

```
    ;calculate b*b
    fetch s3, b_lsb          ;load b
    fetch s4, b_lsb          ;load b
    call mult_soft           ;calculate b*b
    store s6, bb_lsb         ;store lower byte of b*b
    store s5, 07             ;store upper byte of b*b
    ;calculate a*a+b*b
    fetch data, aa_lsb       ;get lower byte of a*a
    add data, s6             ;add lower byte of a*a+b*b
    store data, aabb_lsb     ;store lower byte of a*a+b*b
    fetch data, aa_msb       ;get upper byte of a*a
    addcy data, s5           ;add upper byte of a*a+b*b
    store data, aabb_msb     ;store upper byte of a*a+b*b
    load data, 00            ;clear data, but keep carry
    addcy data, 00           ;get carry-out from previous +
    store data, aabb_cout    ;store carry-out of a*a+b*b
    return

;=========================================================
;routine: mult_soft
;  function: 8-bit unsigned multiplier using
;            shift-and-add algorithm
;  input register:
;     s3: multiplicand
;     s4: multiplier
;  output register:
;     s5: upper byte of product
;     s6: lower byte of product
;  temp register: i
;=========================================================
mult_soft:
    load s5, 00              ;clear s5
    load i, 08               ;initialize loop index
mult_loop:
    sr0  s4                  ;shift lsb to carry
    jump nc, shift_prod      ;lsb is 0
    add s5, s3               ;lsb is 1
shift_prod:
    sra s5                   ;shift upper byte right,
                             ;carry to MSB, LSB to carry
    sra s6                   ;shift lower byte right,
                             ;lsb of s5 to MSB of s6
    sub i, 01                ;dec loop index
    jump nz, mult_loop       ;repeat until i=0
    return
```

16.4.2 Program documentation

Developing an assembly program is a tedious process. The use of symbolic names and good documentation can make the code clear and reduce many unnecessary errors. It also helps future revision and maintenance. For the KCPSM3 assembler, we can use the **constant**

directive to assign a symbolic name (alias) to a data constant, a memory address, or a port id, and use the **namereg** directive to assign a symbolic name to a register.

A representative main program header is shown in Listing 16.2. It contains the following segments:

- *General program description*: provides a general description for the purpose, operation, and I/O of the program
- *Data constants*: declares symbolic names for constants
- *Data RAM address alias*: declares symbolic names for data RAM addresses
- *Register alias*: declares symbolic names for registers
- *Port alias*: declares symbolic names for I/O ports
- *Program calling hierarchy*: illustrates the calling structure and subroutines

The aliases and directives have no effect on the final machine code. When the assembly code is processed, they are replaced with the actual constant values. However, using aliases can greatly enhance the readability of the assembly code and reduce unnecessary errors. The following code segment further illustrates the impact of the alias and documentation. The purpose of this segment is to obtain values for variables a, b, and c, and store them in proper data RAM locations. The location is specified by the UART input, which is the ASCII code of character a, b, or c. The segment with aliases and proper comments is

```
;constant alias
   constant ASCII_a, 61          ;ASCII code for a
   constant ASCII_b, 62          ;ASCII code for b
   constant ASCII_c, 63          ;ASCII code for c
;data ram address alias
   constant a_addr, 02
   constant b_addr, 04
   constant c_addr, 06
;register alias
   namereg s0, data              ;reg for temporary data
   namereg s1, addr              ;reg for temporary addr
   namereg sF, sw_in             ;switch input
;port alias
   constant sw_port, 01          ;switch input
   constant uart_rx_port, 02     ;UART input

;assembly code with alias
   ;get input
   input sw_in, sw_port          ;get switch
   input data, uart_rx_port      ;get char
   ;check received char
   compare data, ASCII_a         ;check ASCII a
   jump nz, chk_ascii_b          ;no, check next
   store  sw_in, a_addr          ;yes, store a to data ram
   jump done
chk_ascii_b:
   compare data, ASCII_b         ;check ASCII b
   jump nz, chk_ascii_c          ;no, check next
   store sw_in, b_addr           ;yes, store b to data ram
   jump done
chk_ascii_c:
   compare data, ASCII_c         ;check ASCII c
   jump nz, ascii_err            ;no, error
```

```
        store sw_in, c_addr          ;yes, store b to data ram
        jump done
    ascii_err:
        ...
    done:
        ...
```

If we use hard literals and strip the comments, the code becomes

```
    ;assembly code with no alias or comments
        input sf, 01
        input s0, 02
        compare s0, 61
        jump nz, addr1
        store sf, 02
        jump addr4
    addr1:
        compare s0, 62
        jump nz, addr2
        store sf, 04
        jump addr4
    addr2:
        compare s0, 63
        jump nz, addr3
        store sf, 06
        jump addr4
    addr3:
        ...
    addr4:
        ...
```

While the functionality of this code segment is the same, it is very difficult to comprehend, debug, or modify.

16.5 PROCESSING OF THE ASSEMBLY CODE

PicoBlaze-based development flow is reviewed in Section 15.4. After the assembly code is developed, it is then compiled (translated) to machine instructions in step 3. The instruction-set-level simulation can also be performed to verify the correctness of the code, as in step 4. The two steps and the direct downloading process (step 9) are discussed in detail in this section.

Xilinx provides an assembler known as *KCPSM3* for compiling in step 3 and downloading utility programs in step 9. The programs, HDL codes for the PicoBlaze processor, and relevant template files can be downloaded from the Xilinx Web site. A program known as *PBlazeIDE* from Mediatronix can perform the instruction-set-level simulation in step 4. It can also be used as an assembler. PBlazeIDE can be downloaded from Mediatronix's Web site.

16.5.1 Compiling with KCSPM3

Assembler is the software that translates the instruction mnemonics to machine instructions, which are represented as 0's and 1's, and substitutes the aliases and symbolic branch addresses with actual values. The machine instructions are then downloaded to the instruction

memory of a microcontroller. Since PicoBlaze is embedded inside FPGA, the instruction ROM becomes an HDL ROM module with the compiled assembly code. The ROM will be instantiated later in the top-level HDL code and synthesized along with PicoBlaze and the I/O interface circuit.

Xilinx provides the *KCPSM3* assembler for this task. It is a command-line, DOS-based program. KCPSM3 basically takes an assembly program, along with the necessary template files, and generates the HDL code for the instruction ROM. The procedure of compiling an assembly program is as follows:

1. Create a directory for the project and copy `kcpsm3.exe`, `ROM_form.vhd`, `ROM_form.v`, and `ROM_form.coe` to the directory. The latter three are code templates used by KCPSM3.
2. Create the assembly program and save it as plain text file with an extension of `.psm`. Any PC-based editor, such as Notepad, can be used for this purpose.
3. Invoke a DOS window by selecting Start ≻ Programs ≻ Accessories ≻ Command Prompt. In the DOS window, navigate to the project directory.
4. Type `kcpsm3 myfile.psm` to run the program.
5. Correct syntax errors if necessary and recompile.
6. After successful compiling, the file containing the instruction ROM, `myfile.v`, is generated.

In addition to the HDL file, KCPSM3 also generates files that are suitable for block RAM initialization and other utilities. The file with the `.hex` extension can be used for JTAG downloading, which is discussed in Section 16.5.3, and the file with the `.fmt` extension is a reformatted `.psm` file for "pretty printing."

16.5.2 Simulation by PBlazeIDE

As the name indicates, instruction-set-level simulation simulates the operation of a PicoBlaze system instruction by instruction. The *PBlazeIDE* program can be used for this purpose. PBlazeIDE is a Windows-based program with an integrated development environment, which includes a text editor, an assembler, and an instruction-set-level simulator.

PBlazeIDE uses slightly different instruction mnemonics and directives, as discussed in Section 15.5. Thus, the code written for by KCPSM3 cannot be used directly by PBlazeIDE, and vice versa. The mnemonic differences are summarized in Table 16.1, and the directive examples are shown in Table 16.2. Note that the PBlazeIDE assembler uses both decimal and hexadecimal format for constants. A hexadecimal number is started with a $ sign, as in `$1A`.

The procedure of using PBlazeIDE for KCPSM3 code is as follows:

1. Compile the assembly code with KCPSM3.
2. Launch PBlazeIDE.
3. Select Settings ≻ PicoBlaze 3. This specifies version 3 of PicoBlaze, which is used in the Spartan-3 device.
4. Select File ≻ Import and a dialog window appears. Select the corresponding `.fmt` file. The "import" function converts the KCPSM3 code to the PBlazeIDE code. The formatted program is easier for conversion. The converted file may sometimes need minor manual editing.
5. Manually specify the **dsin**, **dsout**, and **dsio** directives for I/O ports. When one of these directives is used, a port indicator will be added to the simulation screen to show the activities of the port.

Table 16.1 Mnemonic differences between KCPSM3 and PBlazeIDE

KCPSM3	PBlazeIDE
addcy	**addc**
subcy	**subc**
compare	**comp**
store sX, (sY)	**store** sX, sY
fetch sX, (sY)	**fetch** sX, sY
input sX, (sY)	**in** sX, sY
input sX, KK	**in** sX, $KK
output sX, (sY)	**out** sX, sY
output sX, KK	**out** sX, $KK
return	**ret**
returni	**reti**
enable interrupt	**eint**
disable interrupt	**dint**

Table 16.2 Directive examples of KCPSM3 and PBlazeIDE

Function	KCPSM3	PBlazeIDE
code location	**address** 3FF	**org** $3FF
constant	**constant** MAX, 3F	MAX **equ** $3F
register alias	**namereg** addr, s2	addr **equ** s2
port alias	**constant** in_port, 00	in_port **dsin** $00
	constant out_port, 10	out_port **dsout** $10
	constant bi_port, 0F	bi_port **dsio** $0F

6. Enter the simulation mode by selecting Simulate ≻ Simulate. Perform simulation.
7. If the assembly code needs to be revised, it must be done outside PBlazeIDE. Simply close the current file, invoke an external editor to edit the original .psm file, save the file, and restart from step 1. If the file is edited within PBlazeIDE, it cannot be converted back to KCPSM3 code.

A representative simulation screenshot is shown in Figure 16.2. The simulator displays the assembly code in the central window and highlights the next instruction to be executed. The instruction address, instruction code, and breakpoints are shown next to the code. The current state of PicoBlaze is shown at the left, including the status of the flags, the content of the registers, and the content of the data RAM. The values of the program counter and stack pointer as well as some execution statistics are shown in the bottom row.

The emulated I/O ports created by the **dsin**, **dsout**, and **dsio** directives are shown at the right. There are an input port, switch, and an output port, led, on this particular screen. Since PBlazeIDE has no information about I/O behavior, the input port data must be entered and modified manually during simulation.

During simulation, the assembly program can be executed continuously, by one step, by one instruction, or to pause at a specific breakpoint. The simulation action is controlled by the commands of the Simulate menu or the icons on the top:

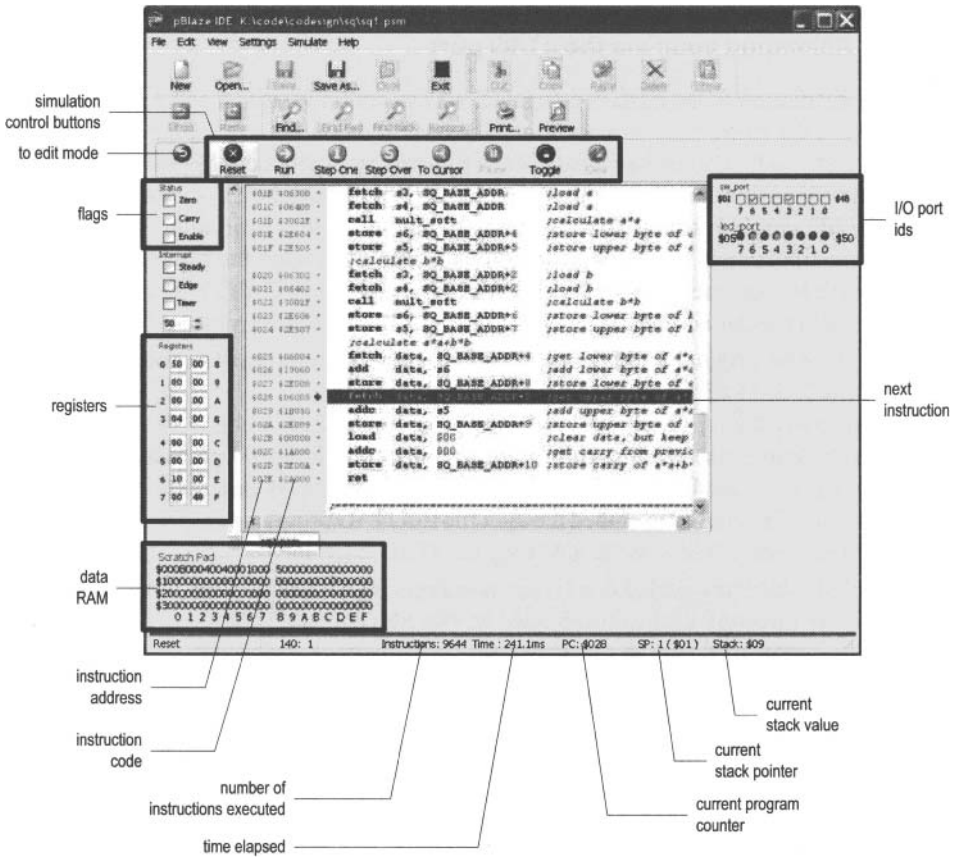


Figure 16.2 Screenshot of pBlazeIDE in simulation mode.

- Reset: clears the program counter and stack pointer
- Run: runs the program continuously until a breakpoint
- Single step: executes one instruction
- Step over: executes the entire subroutine for a **call** instruction and executes one instruction for other instructions
- Run to cursor: runs the program to the current cursor position
- Pause: pauses the simulation
- Toggle breakpoint: sets or clears a breakpoint at the current cursor position
- Remove all breakpoints: clears all breakpoints

16.5.3 Reloading code via the JTAG port

After the instruction ROM HDL is generated, we can continue steps 6 and 8 in Figure 15.4 to synthesize the entire code and download the configuration file to the FPGA chips. Note that the synthesis flow must be repeated each time the assembly code is modified.

Since synthesis is a complex process, it requires a significant amount of computation time. When the I/O configuration is fixed, resynthesizing the entire circuit after each assembly program modification is not really needed. It is possible to reload the machine code to the ROM, which is implemented by a block RAM, by using the FPGA's JTAG interface. This corresponds to the dotted line of step 9 in Figure 15.4. The basic procedure is as follows:

1. Replace the original ROM template with one that contains the JTAG interface circuit.
2. Use KCPSM3 to compile the assembly code as usual.
3. Synthesize the top-level HDL code and program the FPGA chip.
4. In subsequent assembly program modifications, compile the program as usual. Recall that a file in `hex` format (ended with the `.hex` extension) is generated.
5. Use the Xilinx utility to embed the `.hex` file to a JTAG programming file and download the file to the FPGA's block RAM via the JTAG interface.

The detailed procedure and the relevant programs and templates can be found in the `JTAG_loader` directory of the downloaded `KCPSM` file.

16.5.4 Compiling by PBlazeIDE

As discussed earlier, PBlazeIDE is an integrated program that contains an assembler and editor. PBlazeIDE can generate an instruction ROM HDL file as well. However, the file is only in VHDL format. Since Xilinx IST supports mixed-language synthesis, this file can still be incorporated into the top-level Verilog module. The detailed procedure can be found in the IST manual.

To obtain the instruction ROM file, we simply include the **vhdl** directive in the assembly code. Its syntax is

```
vhdl "ROM_form.vhd", "rom_target.vhd", "rom_entity_name"
```

The three parameters specify a VHDL template file, which is the same file as that discussed in Section 16.5.1, the name of the generated ROM VHDL file, and the desired entity name in the VHDL file. Note that since PBlazeIDE does not generate a `.hex` file, the reloading scheme discussed in Section 16.5.3 cannot be applied directly.

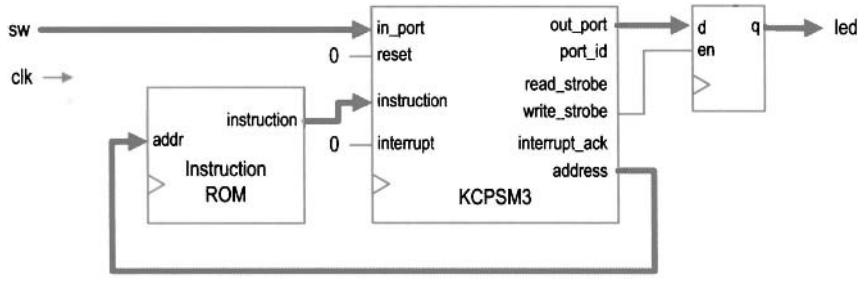


Figure 16.3 PicoBlaze with a simple I/O interface.

16.6 SYNTHESES WITH PICOBLAZE

After generating the HDL file for the instruction ROM, we can combine it with PicoBlaze to synthesize the entire system in an FPGA chip. Unlike a normal microcontroller, PicoBlaze has no built-in I/O peripherals. The I/O interface is created and customized as needed. The circuit is described in HDL code. Since the focus in this chapter is on assembly program development, we use a simple I/O configuration, which contains only one switch input port and one led output port, for synthesis. The development of a more sophisticated I/O interface is discussed in detail in Chapters 17 and 18.

The top-level block diagram of this design is shown in Figure 16.3. It contains the PicoBlaze processor, which is labeled `kcpsm3`, the instruction ROM, and a register. The register functions as a buffer for the eight LEDs. When PicoBlaze executes the **output** instruction, it places the data on `out_port` and asserts the `write_strobe` signal, which enables the register and stores the data in the register. The `sw` signal is connected to `in_port`. When PicoBlaze executes the **input** instruction, it retrieves the value of the `sw` signal and stores it in an internal register. The corresponding HDL code is shown in Listing 16.3. It consists of instantiations of the PicoBlaze processor and instruction ROM, and a segment for the output buffer. The `kcpsm3` module is the name of the PicoBlaze processor, and its code is stored in an HDL file of the same name. The `sio_rom` module is from the previously generated instruction ROM file.

Listing 16.3 PicoBlaze with a simple I/O configuration

```
module pico_sio
   (
    input wire clk, reset,
    input wire [7:0] sw,
    output wire [7:0] led
   );

   // signal declaration
   // KCPSM3/ROM signals
   wire [9:0] address;
   wire [17:0] instruction;
   wire [7:0] port_id, in_port, out_port;
   wire  write_strobe;
   // register signals
   reg [7:0] led_reg;
```

```
   //body
   // ================================================
   //  KCPSM and ROM instantiation
   // ================================================
   kcpsm3 proc_unit
      (.clk(clk), .reset(reset), .address(address),
       .instruction(instruction), .port_id(),
       .write_strobe(write_strobe), .out_port(out_port),
       .read_strobe(), .in_port(in_port), .interrupt(1'b0),
       .interrupt_ack());
   sio_rom rom_unit
      (.clk(clk), .address(address),
       .instruction(instruction));
   // ================================================
   //  output interface
   // ================================================
   always @(posedge clk)
      if (write_strobe)
         led_reg <= out_port;
   assign led = led_reg;
   // ================================================
   //  input interface
   // ================================================
   assign in_port = sw;

endmodule
```

16.7 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 15. The procedure of reloading compiled code via JTAG port is explained in the article "PicoBlaze JTAG Loader Quick User Guide" by Kris Chaplin and Ken Chapman, which appears in the `JTAG_loader` directory of the downloaded `KCPSM` file.

16.8 SUGGESTED EXPERIMENTS

16.8.1 Signed multiplication

The subroutine in Listing 16.1 assumes that the inputs are in unsigned integer format. Modify the subroutine to perform the signed multiplication, in which the two inputs and output are interpreted as signed integers, and use simulation to verify its operation.

16.8.2 Multi-byte multiplication

The subroutine in Listing 16.1 assumes that the inputs are 8 bits wide. Some application may need more precision and we want to extend the subroutine to take 16-bit unsigned inputs. An operand now requires two registers and the result needs four registers. Develop the subroutine and use simulation to verify its operation.

16.8.3 Barrel shift function

PicoBlaze can only shift or rotate a single bit. A "barrel" shifting function can perform the shift and rotate operation for multiple bits. This function has three input registers. The first register contains data to be shifted or rotated; the second register specifies the amount, which is between 0 and 7; and the third register indicates the types of operation, which can be shift left, shift right, rotate left, or rotate right. We assume that 0 will be shifted in for the two shift operations. Develop the subroutine and use simulation to verify its operation.

16.8.4 Reverse function

A reverse function reverses the bit order of an input. For example, if the input is "01010011", the output becomes "11001010". We can use the 8-bit switch as input and the 8-bit discrete LEDs as output. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

16.8.5 Binary-to-BCD conversion

Binary-to-BCD conversion is discussed in Section 6.3.3. This function can be implemented by using assembly code as well. Assume that the input is an 8-bit binary number and the output is a two-digit 8-bit BCD number. If the input exceeds 99, the output generates a special overflow pattern, "11111111". We can use the 8-bit switch as input and the 8-bit discrete LEDs as output. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

16.8.6 BCD-to-binary conversion

Repeat Experiment 16.8.5, but develop the assembly code and circuit for BCD-to-binary conversion.

16.8.7 Heartbeat circuit

A "heartbeat circuit" is discussed in Experiment 4.7.4. We can create a similar pattern using the eight discrete LEDs as well. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

16.8.8 Rotating LED circuit

We want to design a circuit that rotates a simple LED pattern to the left or right at four different speeds. The four patterns are "00000001", "00000011", "00001111", and "00001101". The pattern, direction, and rotation speed can be selected from the 8-bit switch (only 5 bits are used). The speed should be chosen properly so that all four patterns are visually observable. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

16.8.9 Discrete LED dimmer

The concept of PWM and LED dimmer are discussed in Experiment 4.7.2. In this experiment, we want to use eight discrete LEDS to show the various degrees of brightness. This

can be done by changing the "on" fraction of an LED. The "on" fraction of the eight LEDS will be $\frac{8}{8}, \frac{7}{8}, \frac{6}{8}, \ldots, \frac{1}{8}$. Derive and simulate the assembly code, obtain the instruction ROM and create the top-level HDL code, synthesize the system, and verify its operation.

CHAPTER 17

PICOBLAZE I/O INTERFACE

17.1 INTRODUCTION

To interact with the external environment, a regular microcontroller chip consists of a variety of built-in I/O peripherals, such as a UART, SPI (serial peripheral interface), timer, and so on. When starting a new development, we select a microcontroller chip according to the I/O requirements of the application and may sometimes need to use additional chips to realize less commonly used functions.

Unlike a regular microcontroller, PicoBlaze has no built-in I/O peripherals. It just provides a simple generic input and output structure for an I/O interface. I/O peripherals are constructed as needed and thus are customized to each application. PicoBlaze uses the **input** and **output** instructions to transfer data between its internal registers and I/O ports, and its interface consists of the following signals:

- `port_id`: an 8-bit signal that specifies the port id (i.e., port address) of an **input** or **output** instruction
- `in_port`: an 8-bit signal where PicoBlaze obtains input data during operation of an **input** instruction
- `out_port`: an 8-bit signal where PicoBlaze places output data during operation of an **output** instruction
- `read_strobe`: a 1-bit signal that is asserted in the second clock cycle of an **input** instruction
- `write_strobe`: a 1-bit signal that is asserted in the second clock cycle of an **output** instruction

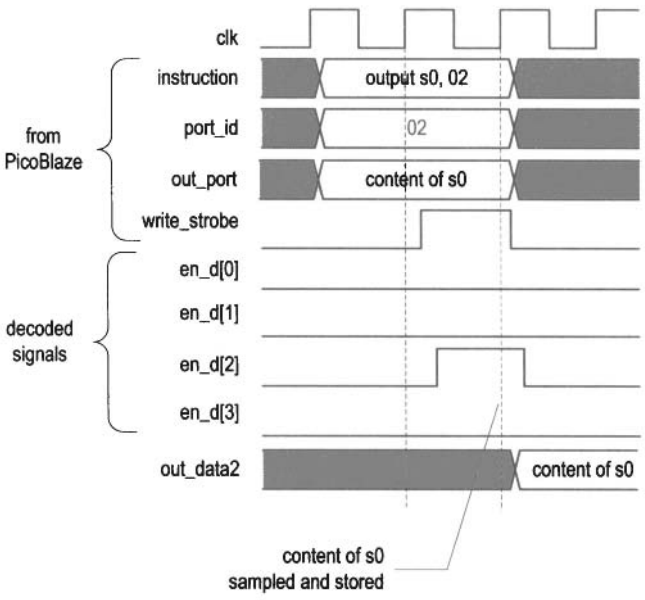


Figure 17.1 Timing diagram of an **output** instruction.

Although there are only two 8-bit ports to input and output data, the 8-bit `port_id` signal can be used to distinguish different peripherals, and thus it is said that PicoBlaze can support up to 256 (i.e., 2^8) input ports and 256 output ports.

In the remaining chapter, we examine the detailed I/O timing of PicoBlaze and illustrate the I/O interface development by adding a series of peripherals for the square circuit of Chapter 16.

17.2 OUTPUT PORT

17.2.1 Output instruction and timing

The **output** instruction writes data to the output port. It has two forms:

```
output sX, (sY)
output sX, port_name
```

In the first form, the port id is stored in the `sY` register. In the second form, the port id is specified explicitly by `port_name`, which is a two-digit hexadecimal number or a previously defined symbolic constant. The output data is always stored in the `sX` register.

The timing diagram of an **output** instruction,

```
output s0, 02
```

is shown in the top five traces of Figure 17.1. Recall that each PicoBlaze instruction takes two clock cycles. When the instruction is executed, the content of `s0` is placed on `out_port` and 02 is placed on `port_id` for two clock cycles. The `write_strobe` signal is asserted in the second clock cycle. It can be used as an enable tick to store data in an output register or to initiate the designated peripheral operation.

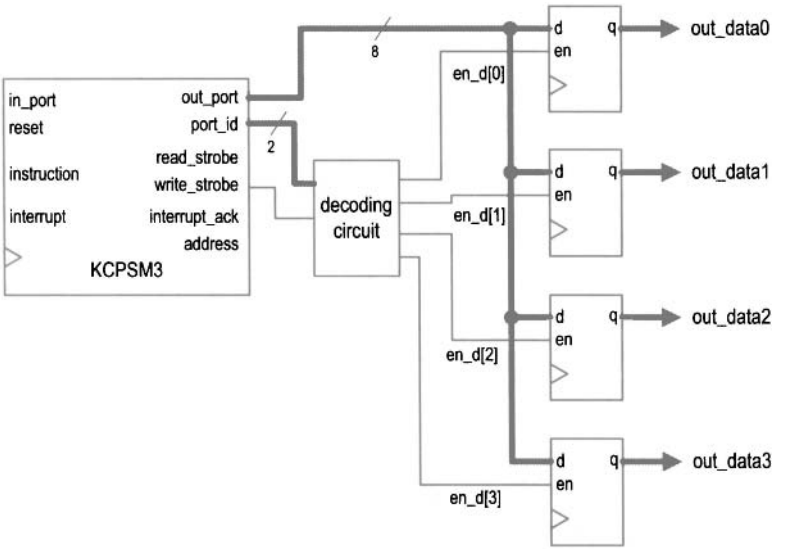


Figure 17.2 Output decoding of four output registers.

Table 17.1 Truth table of a decoding circuit

input			output
write_strobe	port_id[1]	port_id[0]	en_d
0	–	–	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

17.2.2 Output interface

The output interface between PicoBlaze and an output peripheral usually consists of a decoding circuit and necessary output buffers, which are normally an array of registers. The decoding circuit decodes the port id and generates an enable tick accordingly. After the **output** instruction, the data will be stored in the designated buffer.

To illustrate the construction, let us consider a PicoBlaze interface with four output buffers. We assign 00_{16}, 01_{16}, 02_{16}, and 03_{16} as their port ids. Note that the six MSBs of the port addresses are identical and only two LSBs are needed to distinguish a port. The block diagram is shown in Figure 17.2. The key is the decoding circuit, whose function table is shown in Table 17.1. It is a 2-to-2^2 decoder. In the second clock cycle of an **output** instruction, `write_strobe` is asserted and 1 bit of the 4-bit `en_d` signal is asserted accordingly. The one-clock-cycle enable tick activates the corresponding output register to retrieve data from the `out_port` signal. The decoding timing diagram of the instruction

```
output s0, 02
```

is shown at the bottom of Figure 17.1. During the second clock cycle of the **output** instruction, the en_d[2] signal is asserted and the data value on out_port is stored in the corresponding buffer at the rising edge of the next clock.

Once understanding the basic operation, we can derive the HDL code accordingly. The code segment is

```
always @*
   if (write_strobe)
      case (port_id[1:0])
         2'b00: en_d = 4'b0001;
         2'b01: en_d = 4'b0010;
         2'b10: en_d = 4'b0100;
         2'b11: en_d = 4'b1000;
      endcase
   else
      en_d = 4'b0000;
```

This scheme is very general and can be applied to any number of output ports.

The choice of the port address is somewhat arbitrary. We use the binary code in the previous example. If the number of the output port is smaller than eight, one-hot code can be used to simplify the decoding circuit. For example, we can define the four previous port ids as 01_{16} (i.e., 00000001_2), 02_{16} (i.e., 00000010_2), 04_{16} (i.e., 00000100_2), and 08_{16} (i.e., 00001000_2). The decoding logic can be simplified to

```
always @*
   if (write_strobe)
      en_d = port_id[3:0];
   else
      en_d = 4'b0000;
```

Note that no decoding logic is needed if there is only a single output port. The write_strobe signal can be connected to the register's enable signal, as shown in Figure 16.3.

As discussed in Section 16.4.2, it is good practice to use symbolic aliases for I/O ports and declare their binary addresses in the header. For example, the initial output port address assignment can be declared as

```
;-----------output port definitions-----------
constant out_port_a, 00
constant out_port_b, 01
constant out_port_c, 02
constant out_port_d, 04
```

If the assignment is changed, we need to modify the header but keep the remaining assembly code intact. Using a clear header also allows us easily to identify the port ids when the companion HDL code is developed.

17.3 INPUT PORT

17.3.1 Input instruction and timing

The **input** instruction reads data from the input port. Similar to the **output** instruction, it has two forms:

```
input sX, (sY)
input sX, port_name
```

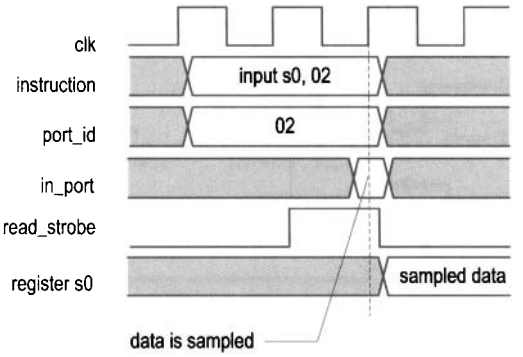


Figure 17.3 Timing diagram of an **input** instruction.

The sY register or port_name specifies the read port id. The retrieved data is stored in the sX register.

The timing diagram of an **input** instruction,

```
input s0, 02
```

is shown in Figure 17.3. When the instruction is executed, 02 is placed on port_id. After two clock cycles, in_port will be sampled at the rising edge of the clock and its value is stored in the s0 register. The external circuit must ensure that the input data is stable during the sampling edge to avoid a timing violation.

As in the **output** instruction, the read_strobe signal is asserted in the second clock cycle. The function of the read_strobe signal is less obvious and is discussed in the next subsection.

17.3.2 Input interface

The input interface between PicoBlaze and input peripherals usually consists of a multiplexing circuit, which uses port_id as the selection signal to route the desired value to in_port. Sometimes, a decoding circuit similar to the one in the output interface is also necessary to signal the completion of the data access.

For the purpose of input interface design, an input port can be classified as a *continuous-access* or *single-access port*. For a continuous-access port, the data is presented continuously, such as the switch input of Section 16.4.1. On the other hand, the availability of data of a single-access port is triggered by a single discrete event, such as receiving a character in an UART buffer. The flag FF and buffers discussed in Section 8.2.4 are in this category. After the data is retrieved, we must remove it from the buffer to prevent the same data from being processed again. This is usually done by utilizing a one-clock-cycle tick to clear the flag FF or remove a word from a FIFO buffer.

The interface for continuous-access ports involves only a multiplexing circuit. Consider an interface with four such ports. The block diagram is shown in Figure 17.4.

The interface for single-access ports needs a mechanism to remove the retrieved data from the buffer in the end of an **input** instruction. This can be done by using a decoding circuit that decodes the port_id and read_strobe signals. The circuit is identical to the decoding circuit of the output interface except that write_strobe is replaced by read_strobe. The decoded output can be considered as a "removal" signal, which is asserted for one clock

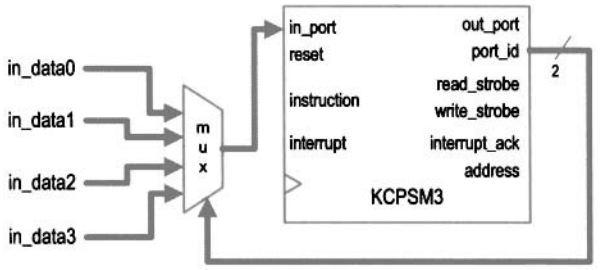


Figure 17.4 Block diagram of four continuous-access ports.

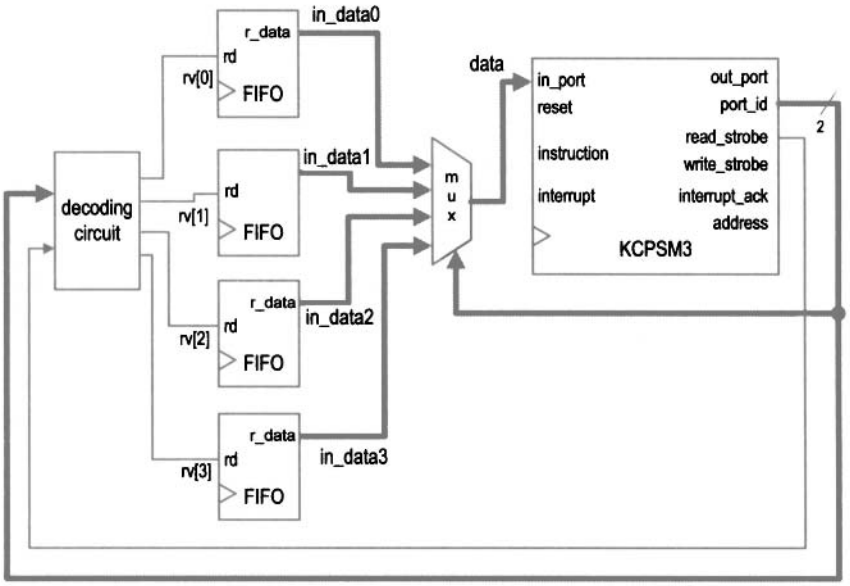


Figure 17.5 Block diagram of four single-access ports.

cycle and removes the previously retrieved data. Consider an interface with four FIFOs. The diagram of the complete decoding and multiplexing circuit is shown in Figure 17.5. The `rv` signal is the decoded removal signal. In the end of an **input** instruction, 1 bit of this 4-bit signal is asserted and the corresponding FIFO performs a read operation, in which the first word is removed from the buffer. Assume that 00_{16}, 01_{16}, 02_{16}, and 03_{16} are assigned as the port ids. The HDL code segment for the interface is

```
// multiplexing circuit
always @*
   case (port_id[1:0])
      2'b00: data = in_data0;
      2'b01: data = in_data1;
      2'b10: data = in_data2;
      2'b11: data = in_data3;
   endcase
// decoding circuit
```

```
always @*
   if (read_strobe)
      case (port_id[1:0])
         2'b00: rv = 4'b0001;
         2'b01: rv = 4'b0010;
         2'b10: rv = 4'b0100;
         2'b11: rv = 4'b1000;
      endcase
   else
      rv = 4'b0000;
```

In a real application, it is likely that the input interface contains both continuous- and single-access ports. A decoding circuit is only needed for single-access ports.

17.4 SQUARE PROGRAM WITH A SWITCH AND SEVEN-SEGMENT LED DISPLAY INTERFACE

To demonstrate the construction of the PicoBlaze I/O interface, we add more versatile input and output peripherals to the square routine of Chapter 16. Recall that the square routine calculates $a^2 + b^2$, where a and b are 8-bit unsigned integers.

We use the 8-bit switch and a pushbutton to enter the values of a and b. The pushbutton generates a one-clock-cycle tick when pressed. The tick indicates that the current value of the switch should be loaded. The values of a and b are loaded alternately; i.e., the first pressing loads a, the second pressing loads b, the third pushing loads a, and so on. A second pushbutton is also included to clear the PicoBlaze's data RAM and relevant registers.

We use four seven-segment LEDs to display the inputs and computed results. The LEDs are arranged as four hexadecimal numbers. Since the range of $a^2 + b^2$ is up to 17 bits, the decimal point of the leftmost LED is used for the MSB. The three lower bits of the switch select what to display, which can be a, b, a^2, b^2, or $a^2 + b^2$.

In summary, the interface consists of the following:

- *Switch*: provides the values of a and b and selects the content of the LED display
- *Pushbutton 0*: loads the a and b alternately when pressed
- *Pushbutton 1*: clears data RAM and relevant registers when pressed
- *Seven-segment LED*: displays the selected 17-bit value in four hexadecimal digits

17.4.1 Output interface

Recall that the four seven-segment LEDs on the prototyping board share the same input pins, and a time-multiplexing circuit is required. For a PicoBlaze-based design, the multiplexing can be done by either an external circuit or a software routine. We use the external-circuit approach, which is simpler for assembly code development, in this section and discuss the software approach in Chapter 18. The LED time-multiplexing circuit designed in Section 4.5.1 can be used for this purpose. This circuit shields the timing and appears as four independent seven-segment LEDs for an external system. The block diagram of the PicoBlaze output interface is shown in Figure 17.6. The interface consists of four 8-bit output ports, each port representing a seven-segment LED pattern.

In the assembly code, the four LED patterns are stored in PicoBlaze's data RAM with symbolic addresses of `led0`, `led1`, `led2`, and `led3`. The corresponding code segment is

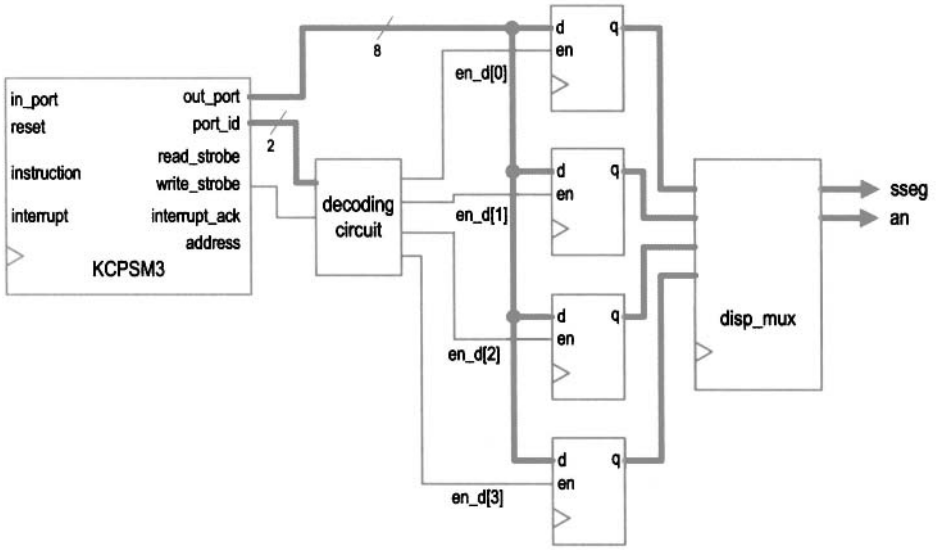


Figure 17.6 Output interface of a square circuit.

```
...
;data RAM address alias
constant led0, 10
constant led1, 11
constant led2, 12
constant led3, 13
...
;output port definitions
constant sseg0_port, 00     ;7-seg led 0
constant sseg1_port, 01     ;7-seg led 1
constant sseg2_port, 02     ;7-seg led 2
constant sseg3_port, 03     ;7-seg led 3
...
disp_led:
   fetch data, led0
   output data, sseg0_port
   fetch data, led1
   output data, sseg1_port
   fetch data, led2
   output data, sseg2_port
   fetch data, led3
   output data, sseg3_port
   return
```

17.4.2 Input interface

The input interface consists of an 8-bit switch and two 1-bit pushbuttons. The former is a continuous-access port since the value is always present. The latter is a single-access port since pressing a button leads to only a single event (e.g., loading a to the register once rather

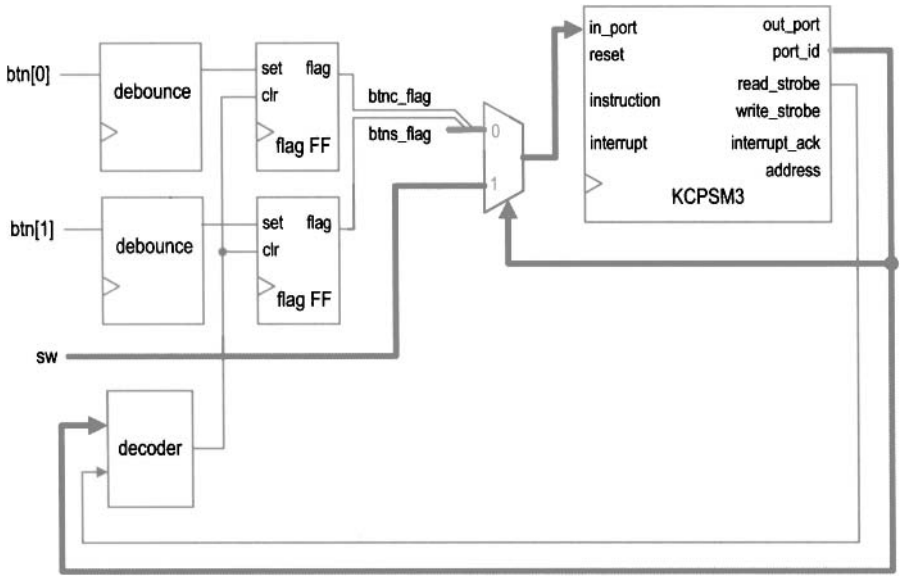


Figure 17.7 Input interface of a square circuit.

than continuously). Because of the mechanical glitches, a debouncing circuit is needed to generate a clean one-clock-cycle tick. Since PicoBlaze's port can take up 8-bit data, inputs from the two pushbuttons can be grouped together as a single input port. The block diagram of the input interface is shown in Figure 17.7. The interface consists of two debouncing circuits, a two-to-one multiplexer, a decoding circuit, and two flag FFs. The function of the two flag FFs is discussed in Section 8.2.4. They provide a mechanism to set and clear the "button-pressing event." When a button is pressed, the debouncing circuit's output sets the flag. It remains asserted until it is retrieved by the PicoBlaze's **input** instruction, which sets the selection signal of the multiplexer to route the desired value to PicoBlaze's input port, and activates the clear signal. For clarity, we name the pushbutton 1 as the s button (for setting the value) and pushbutton 0 as the c button (for clearing the data RAM).

The pseudo code to process the input is

```
;input the button flags
;if c=1 then
;   call the clearing-ram routine
;if s=1 then
;   input switch value
;   store it to data ram
;   toggle a/b address offset
```

Since the s button inputs the values of a and b alternately, we use a global register, `switch_a_b`, to keep track of which one is being read currently. The register serves as the data RAM address offset, which can be 0 or 2, and its value toggles when the s button is pressed. The corresponding assembly code subroutine is

```
;input port definitions
constant rd_flag_port, 00   ;2 flags (xxxxxxsc):
constant sw_port, 01        ;8-bit switch
```

```
...
proc_btn:
   input s3, rd_flag_port   ;get flag
   ;check and process c button
   test s3, 01              ;check c button flag
   jump z, chk_btns         ;flag not set
   call init                ;flag set, clear
   jump proc_btn_done
chk_btns:
   ;check and process s button
   test s3, 02              ;check s button flag
   jump z, proc_btn_done    ;flag not set
   input data, sw_port      ;get switch
   load addr, a_lsb         ;get addr of a
   add addr, switch_a_b     ;add offset
   store data, (addr)       ;write data to ram
   ;update current disp position
   xor switch_a_b, 02       ;toggle between 00, 02
proc_btn_done:
   return
```

17.4.3 Assembly code development

After designing the I/O interface, we can derive the assembly program. The development follows the divide-and-conquer approach discussed in Chapter 16 and partitions the main program into several subroutines. The main program is

```
   call init                ;initialization
forever:
   ;main loop body
   call proc_btn            ;check & process buttons
   call square              ;calculate square
   call load_led_pttn       ;store led patterns to ram
   call disp_led            ;output led pattern
   jump forever
```

The complete code is shown in Listing 17.1.

The `square` subroutine is from Chapter 16, and the `proc_btn` and `disp_led` subroutines are discussed in the two preceding subsections. The `init` subroutine performs system initialization. It uses a loop to load 0's to data RAM (i.e., clear the RAM) and sets the `switch_a_b` register to 0 (i.e., read a). The `load_led_pttn` subroutine reads the switch input, retrieves the desired values from the data RAM, converts the values to seven-segment LED patterns, and stores them to the corresponding locations in the data RAM. These patterns are then written to the output ports in the subsequent `disp_led` routine. The `load_led_pttn` routine consists of the `get_upper_nibble` and `get_lower_nibble` routines to extract the two hexadecimal digits and the `hex_to_led` routine to convert a hexadecimal digit to the corresponding seven-segment LED pattern.

The program requires more storage. In addition to the data RAM and registers required for the `square` subroutine, this program utilizes a new global register `switch_a_b` to keep track of whether a or b is being read, and 4 bytes in data RAM, whose addresses are labeled `led0`, `led1`, `led2`, and `led3`, to store four seven-segment LED patterns.

Listing 17.1 Square program with a switch and seven-segment LED interface

```
;=============================================================
; square circuit with 7-seg LED interface
;=============================================================
;program operation:
;  - read a and b from switch
;  - calculate a*a + b*b
;  - display data on 7-seg led

;=============================================================
; data RAM address alias
;=============================================================
constant a_lsb, 00
constant b_lsb, 02
constant aa_lsb, 04
constant aa_msb, 05
constant bb_lsb, 06
constant bb_msb, 07
constant aabb_lsb, 08
constant aabb_msb, 09
constant aabb_cout, 0A
constant led0, 10
constant led1, 11
constant led2, 12
constant led3, 13

;=============================================================
; register alias
;=============================================================
;commonly used local variables
namereg s0, data          ;reg for temporary data
namereg s1, addr          ;reg for temporary mem & i/o port addr
namereg s2, i             ;general-purpose loop index
;global variables
namereg sf, switch_a_b ;ram offset for current switch input

;=============================================================
; port alias
;=============================================================
;--------------input port definitions-------------------------
constant rd_flag_port, 00   ;2 flags (xxxxxxsc):
constant sw_port, 01        ;8-bit switch
;--------------output port definitions------------------------
constant sseg0_port, 00     ;7-seg led 0
constant sseg1_port, 01     ;7-seg led 1
constant sseg2_port, 02     ;7-seg led 2
constant sseg3_port, 03     ;7-seg led 3

;=============================================================
; main program
;=============================================================
;calling hierarchy:
;
```

```
;main
;  - init
;  - proc_btn
;       - init
;  - square
;       - mult_soft
;  - load_led_pttn
;       - get_lower_nibble
;       - get_upper_nibble
;       - hex_to_led
;  - disp_led
;
; ============================================================

    call init                ;initialization
forever:
    ;main loop body
    call proc_btn            ;check & process buttons
    call square              ;calculate square
    call load_led_pttn       ;store led patterns to ram
    call disp_led            ;output led pattern
    jump forever

;============================================================
;routine: init
;  function: perform initialization, clear register/ram
;  output register:
;    switch_a_b: cleared to 0
;  temp register: data, i
;============================================================
init:
    ;clear memory
    load i, 40               ;unitize loop index to 64
    load data, 00
clr_mem_loop:
    store data, (i)
    sub i, 01                ;dec loop index
    jump nz, clr_mem_loop    ;repeat until i=0
    ;clear register
    load switch_a_b, 00
    return

;============================================================
;routine: proc_btn
;  function: check two buttons and process the display
;  input reg:
;     switch_a_b: ram offset (0 for a and 2 for b)
;  output register:
;     s3: store input port flag
;     switch_a_b: may be toggled
;  temp register used: data, addr
;============================================================
proc_btn:
```

```
    input s3, rd_flag_port  ;get flag
    ;check and process c button
    test s3, 01             ;check c button flag
    jump z, chk_btns        ;flag not set
    call init               ;flag set, clear
    jump proc_btn_done
chk_btns:
    ;check and process s button
    test s3, 02             ;check s button flag
    jump z, proc_btn_done   ;flag not set
    input data, sw_port     ;get switch
    load addr, a_lsb        ;get addr of a
    add addr, switch_a_b    ;add offset
    store data, (addr)      ;write data to ram
    ;update current disp position
    xor switch_a_b, 02      ;toggle between 00, 02
proc_btn_done:
    return

;=============================================================
;routine: load_led_pttn
;  function: read 3 LSBs of switch input and convert the
;            desired values to four led patterns and
;            load them to ram
;            switch: 000:a; 001:b; 010:a^2; 011:b^2;
;                    others: a^2 + b^2
;  temp register used: data, addr
;    s6: data from sw input port

;=============================================================
load_led_pttn:
    input s6, sw_port       ;get switch
    sl0 s6                  ;*2 to obtain addr offset
    compare s6, 08          ;sw>100?
    jump c, sw_ok           ;no
    load s6, 08             ;yes, sw error, make default
sw_ok:
    ;process byte 0, lower nibble
    load addr, a_lsb
    add addr, s6            ;get lower addr
    fetch data, (s6)        ;get lower byte
    call get_lower_nibble   ;get lower nibble
    call hex_to_led         ;convert to led pattern
    store data, led0
    ;process byte 0, upper nibble
    fetch data, (addr)
    call get_upper_nibble
    call hex_to_led
    store data, led1
    ;process byte 1, lower nibble
    add addr, 01            ;get upper addr
    fetch data, (addr)
    call get_lower_nibble
```

```
      call hex_to_led
      store data, led2
      ;process byte 1, upper nibble
      fetch data, (addr)
      call get_upper_nibble
      call hex_to_led
      ;check for sw=100 to process carry as led dp
      compare s6, 08          ;display final result?
      jump nz, led_done       ;no
      add addr, 01            ;get carry addr
      fetch s6, (addr)        ;s6 to store carry
      test s6, 01             ;carry=1?
      jump z, led_done        ;no
      and data, 7F            ;yes, assert msb (dp) to 0
  led_done:
      store data, led3
      return

  ;=================================================
  ;routine: disp_led
  ;  function: output four led patterns
  ;  temp register used: data
  ;=================================================
  disp_led:
      fetch data, led0
      output data, sseg0_port
      fetch data, led1
      output data, sseg1_port
      fetch data, led2
      output data, sseg2_port
      fetch data, led3
      output data, sseg3_port
      return

  ;=================================================
  ;routine: hex_to_led
  ;  function: convert a hex digit to 7-seg led pattern
  ;  input register: data
  ;  output register: data
  ;=================================================
  hex_to_led:
      compare data, 00
      jump nz, comp_hex_1
      load data, 81           ;7-seg pattern 0
      jump hex_done
  comp_hex_1:
      compare   data, 01
      jump nz, comp_hex_2
      load data, CF           ;7-seg pattern 1
      jump hex_done
  comp_hex_2:
      compare data, 02
      jump nz, comp_hex_3
```

```
    load data, 92          ;7-seg pattern 2
    jump hex_done
comp_hex_3:
    compare data, 03
    jump nz, comp_hex_4
    load data, 86          ;7-seg pattern 3
    jump hex_done
comp_hex_4:
    compare data, 04
    jump nz, comp_hex_5
    load data, CC          ;7-seg pattern 4
    jump hex_done
comp_hex_5:
    compare data, 05
    jump nz, comp_hex_6
    load data, A4          ;7-seg pattern 5
    jump hex_done
comp_hex_6:
    compare data, 06
    jump nz, comp_hex_7
    load data, A0          ;7-seg pattern 6
    jump hex_done
comp_hex_7:
    compare data, 07
    jump nz, comp_hex_8
    load data, 8F          ;7-seg pattern 7
    jump hex_done
comp_hex_8:
    compare data, 08
    jump nz, comp_hex_9
    load data, 80          ;7-seg pattern 8
    jump hex_done
comp_hex_9:
    compare data, 09
    jump nz, comp_hex_a
    load data, 84          ;7-seg pattern 9
    jump hex_done
comp_hex_a:
    compare data, 0A
    jump nz, comp_hex_b
    load data, 88          ;7-seg pattern a
    jump hex_done
comp_hex_b:
    compare data, 0B
    jump nz, comp_hex_c
    load data, E0          ;7-seg pattern b
    jump hex_done
comp_hex_c:
    compare data, 0C
    jump nz, comp_hex_d
    load data, B1          ;7-seg pattern C
    jump hex_done
comp_hex_d:
```

```
    compare data, 0D
    jump nz, comp_hex_e
    load data, C2          ;7-seg pattern d
    jump hex_done
comp_hex_e:
    compare data, 0E
    jump nz, comp_hex_f
    load data, B0          ;7-seg pattern E
    jump hex_done
comp_hex_f:
    load data, B8          ;7-seg pattern F
hex_done:
    return

;=================================================================
;routine: get_lower_nibble
;  function: get lower 4 bits of data
;  input register: data
;  output register: data
;=================================================================
get_lower_nibble:
    and data, 0F           ;clear upper nibble
    return

;=================================================================
;routine: get_upper_nibble
;  function: get upper 4 bits of in_data
;  input register: data
;  output register: data
;=================================================================
get_upper_nibble:
    sr0 data               ;right shift 4 times
    sr0 data
    sr0 data
    sr0 data
    return

;=================================================================
;routine: square
;  function: calculate a*a + b*b
;      data/result stored in ram started w/ SQ_BASE_ADDR
;  temp register: s3, s4, s5, s6, data
;=================================================================
square:
    ;calculate a*a
    fetch s3, a_lsb        ;load a
    fetch s4, a_lsb        ;load a
    call mult_soft         ;calculate a*a
    store s6, aa_lsb       ;store lower byte of a*a
    store s5, aa_msb       ;store upper byte of a*a
    ;calculate b*b
    fetch s3, b_lsb        ;load b
    fetch s4, b_lsb        ;load b
```

```
        call mult_soft          ;calculate b*b
        store s6, bb_lsb        ;store lower byte of b*b
320     store s5, bb_msb        ;store upper byte of b*b
        ;calculate a*a+b*b
        fetch data, aa_lsb      ;get lower byte of a*a
        add data, s6            ;add lower byte of a*a+b*b
        store data, aabb_lsb    ;store lower byte of a*a+b*b
325     fetch data, aa_msb      ;get upper byte of a*a
        addcy data, s5          ;add upper byte of a*a+b*b
        store data, aabb_msb    ;store upper byte of a*a+b*b
        load data, 00           ;clear data, but keep carry
        addcy data, 00          ;get carry from previous +
330     store data, aabb_cout   ;store carry of a*a+b*b
        return

;=================================================================
;routine: mult_soft
335 ;   function: 8-bit unsigned multiplier using
;               shift-and-add algorithm
;   input register:
;       s3: multiplicand
;       s4: multiplier
340 ;   output register:
;       s5: upper byte of product
;       s6: lower byte of product
;   temp register: i
;=================================================================
345 mult_soft:
        load s5, 00             ;clear s5
        load i, 08              ;initialize loop index
    mult_loop:
        sr0  s4                 ;shift lsb to carry
350     jump nc, shift_prod     ;lsb is 0
        add s5, s3              ;lsb is 1
    shift_prod:
        sra s5                  ;shift upper byte right,
                                ;carry to MSB, LSB to carry
355     sra s6                  ;shift lower byte right,
                                ;lsb of s5 to MSB of s6
        sub i, 01               ;dec loop index
        jump nz, mult_loop      ;repeat until i=0
        return
```

17.4.4 HDL code development

The complete HDL code simply combines the PicoBlaze processor, instruction ROM, the input interface and peripherals shown in Figure 17.7, and the output interface and peripherals shown in Figure 17.6. It is shown in Listing 17.2.

Listing 17.2 PicoBlaze with a switch and seven-segment LED interface

```
module pico_btn
   (
    input wire clk, reset,
    input wire [7:0] sw,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   // KCPSM3/ROM signals
   wire [9:0] address;
   wire [17:0] instruction;
   wire [7:0] port_id, out_port;
   reg [7:0] in_port;
   wire  write_strobe, read_strobe;
   // I/O port signals
   // output enable
   reg [3:0] en_d;
   // four-digit seven-segment led display
   reg [7:0] ds3_reg, ds2_reg, ds1_reg, ds0_reg;
   // two pushbuttons
   reg btnc_flag_reg, btns_flag_reg;
   wire btnc_flag_next, btns_flag_next;
   wire set_btnc_flag, set_btns_flag, clr_btn_flag;

   //body
   // ================================================
   //  I/O modules
   // ================================================
   disp_mux disp_unit
      (.clk(clk), .reset(reset),
       .in3(ds3_reg), .in2(ds2_reg), .in1(ds1_reg),
       .in0(ds0_reg), .an(an), .sseg(sseg));
   debounce btnc_unit
      (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(set_btnc_flag));
   debounce btns_unit
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(set_btns_flag));
   // ================================================
   //  KCPSM and ROM instantiation
   // ================================================
   kcpsm3 proc_unit
      (.clk(clk), .reset(1'b0), .address(address),
       .instruction(instruction), .port_id(port_id),
       .write_strobe(write_strobe), .out_port(out_port),
       .read_strobe(read_strobe), .in_port(in_port),
         .interrupt(1'b0), .interrupt_ack());
   btn_rom rom_unit
      (.clk(clk), .address(address),
       .instruction(instruction));
```

```
// ============================================================
//  output interface
// ============================================================
//    outport port id:
//      0x00: ds0
//      0x01: ds1
//      0x02: ds2
//      0x03: ds3
// ============================================================
// registers
always @(posedge clk)
   begin
      if (en_d[0])
         ds0_reg <= out_port;
      if (en_d[1])
         ds1_reg <= out_port;
      if (en_d[2])
         ds2_reg <= out_port;
      if (en_d[3])
         ds3_reg <= out_port;
   end
// decoding circuit for enable signals
always @*
   if (write_strobe)
      case (port_id[1:0])
         2'b00: en_d = 4'b0001;
         2'b01: en_d = 4'b0010;
         2'b10: en_d = 4'b0100;
         2'b11: en_d = 4'b1000;
      endcase
   else
      en_d = 4'b0000;

// ============================================================
//  input interface
// ============================================================
//    input port id
//      0x00: flag
//      0x01: switch
// ============================================================
// input register (for flags)
always @(posedge clk)
   begin
      btnc_flag_reg <= btnc_flag_next;
      btns_flag_reg <= btns_flag_next;
   end
assign btnc_flag_next = (set_btnc_flag) ? 1'b1 :
                        (clr_btn_flag)  ? 1'b0 :
                         btnc_flag_reg;
assign btns_flag_next = (set_btns_flag) ? 1'b1 :
                        (clr_btn_flag)  ? 1'b0 :
                         btns_flag_reg;
// decoding circuit for clear signals
```

```
   assign clr_btn_flag = read_strobe && (port_id[0]==1'b0);
   // input multiplexing
   always @*
      case(port_id[0])
110      1'b0: in_port = {6'b0, btns_flag_reg, btnc_flag_reg};
         1'b1: in_port = sw;
      endcase

endmodule
```

17.5 SQUARE PROGRAM WITH A COMBINATIONAL MULTIPLIER AND UART CONSOLE

In this section, we add two more I/O peripherals to the previous design. One is a combinational multiplier, which accelerates the multiplication, and the other is an UART, which provides a communication link to a PC.

17.5.1 Multiplier interface

Since PicoBlaze does not contain a hardware multiplier, the multiplication is done by a software routine, `mult_soft`. It uses a shift-and-add algorithm to iterate through the 8-bit multiplier and requires about 60 instructions in the worst-case scenario. An alternative is to utilize the Spartan-3 device's built-in combinational multiplier.

Since PicoBlaze provides no mechanism to use a coprocessor, the multiplier must be configured as an I/O peripheral. We can create an 8-bit combinational multiplier that takes two 8-bit operands and returns a 16-bit product. To facilitate this peripheral, the PicoBlaze's interface requires two additional output ports and buffers for the two operands and two additional input ports for the 16-bit product. The assembly routine now only needs to pass the operands to the output ports and then retrieve the results from the input ports. The code becomes

```
;input port definitions
constant mult_prod0_port, 03 ;multiplication product 8 LSBs
constant mult_prod1_port, 04 ;multiplication product 8 MSBs
;output port definitions
constant mult_src0_port, 05  ;multiplier operand 0
constant mult_src1_port, 06  ;multiplier operand 1
...
mult_hard:
   output s3, mult_src0_port
   output s4, mult_src1_port
   input s5, mult_prod1_port
   input s6, mult_prod0_port
   return
```

Note that the combinational multiplier can complete the computation with one instruction (i.e., two clock cycles), and thus no additional timing mechanism is needed in the code. This routine can be used in place of the previous `mult_soft` routine.

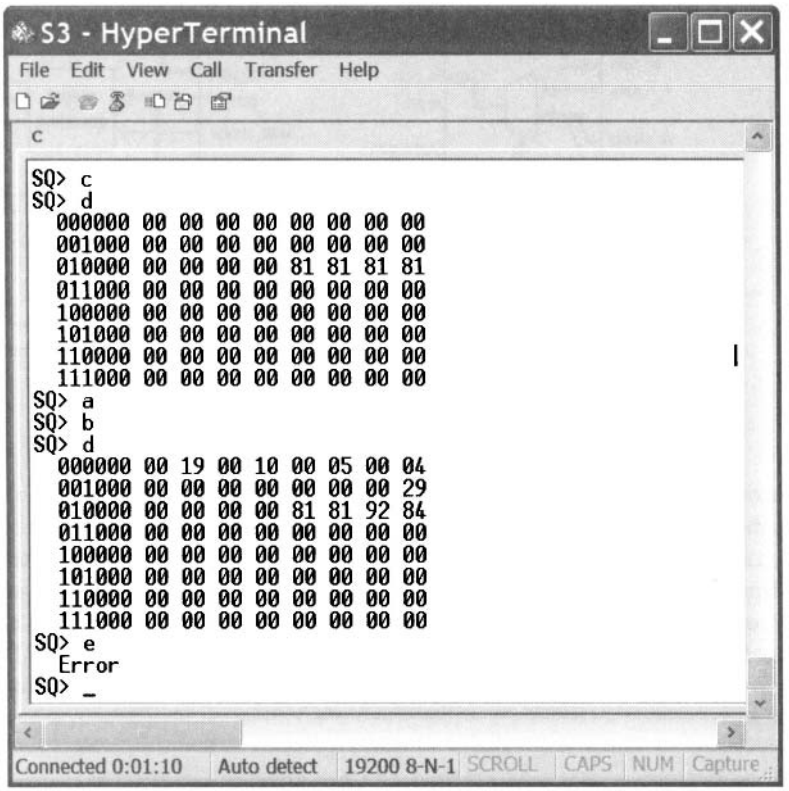


Figure 17.8 Representative console screen.

17.5.2 UART interface

With the UART interface, information can be entered and displayed in Windows HyperTerminal, which is more flexible and versatile than switches and LEDs. We use it as a simple control console for the `square` routine. A representative screen is shown in Figure 17.8. The console generates an `SQ>` prompt and a user can respond with a lowercase `a`, `b`, `c`, or `d` character. The `a` and `b` characters are used to input values for a and b of the `square` routine. When the key is pressed, the value of the 8-bit switch is read and stored into the corresponding data RAM location. The `c` character is used to clear the data RAM and reinitialize the program. Its function is identical to that of the `c` button. The `d` character leads to a "data RAM dump," in which the 64 bytes of the data RAM are displayed on screen. This allows us to observe the various values of the `square` routine and the four seven-segment LED patterns. An `Error` message is returned for all other characters.

The UART module designed in Section 8.4 can be used for this purpose. Since the transmission and receiving FIFO buffers provide a storage and flagging mechanism, no additional circuit is needed. We need only expand the decoding and multiplexing circuits to accommodate the additional I/O ports. The UART interface block diagram is sketched in Figure 17.9, in which the other I/O peripherals are omitted to reduce clutter. PicoBlaze's output port, `out_port`, is connected to `w_data` of UART. The decoded enable signal is connected to `wr_uart`, and the data is written to UART transmitting FIFO when it is asserted. Similarly, `r_data` of UART is routed to PicoBlaze's input multiplexing circuit,

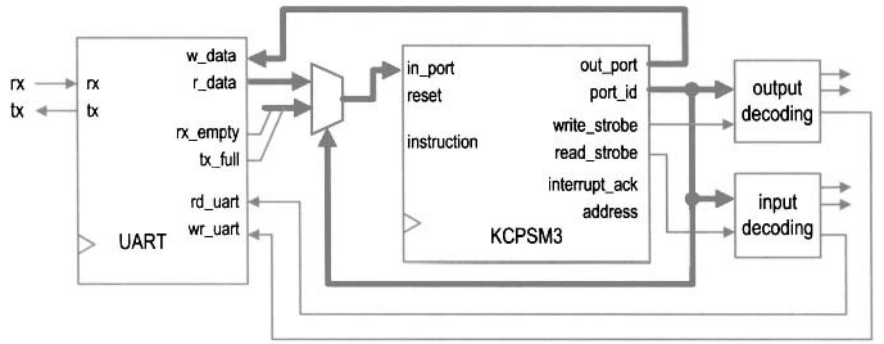


Figure 17.9 UART I/O interface.

and the decoded clear signal is connected to `rd_uart`. When the UART receiving FIFO port is specified in an **input** instruction, the receiving FIFO's output is routed to PicoBlaze's input port, `in_port`, and the decoded remove signal is asserted one clock cycle to remove one word from the receiving FIFO. The UART interface also needs to route the two status signals, `rx_empty` and `tx_full`, to PicoBlaze's input multiplexing circuit. The assembly program needs to check the status before reading or writing the UART's FIFOs. Since the signals are only 2 bits wide, they can be grouped with the previous `s` and `c` buttons in the same input port.

17.5.3 Assembly code development

Since the previous assembly code is developed in a modular fashion, we can expand the program by adding a routine, `proc_uart`, to process UART transactions. The main program becomes

```
    call init                ;initialization
forever:
    ;main loop body
    call proc_btn            ;check & process buttons
    call proc_uart           ;check & process uart rx
    call square              ;calculate square
    call load_led_pttn       ;store led patterns to ram
    call disp_led            ;output led pattern
    jump forever
```

Because of the complexity of the required console operation, the `proc_uart` is quite involved. The pseudo code of this routine is

```
;   if (no character in UART receiving FIFO) then
;      return
;   input characters from FIFO
;   if (characters is a) then
;      input switch value
;      store it to data ram
;      display prompt
;      return
;   if (characters is b) then
```

```
;       input switch value
;       store it to data ram
;       display prompt
;       return
;    if (characters is c) then
;       perform initialization
;       return
;    if (characters is d) then
;       dump data ram
;       return
;    display error message
;    return
```

We follow the modular development approach and further divide this routine into simpler routines. A key low-level routine is `tx_one_byte`, which transmits 1 byte via the UART port. Its code is

```
;input port definitions
constant rd_flag_port, 00
; 4 flags (xxxxtrsc):
;    t: uart tx full,   r: uart rx not empty
;    s: s button flag,  c: c button flag
;output port definitions
constant uart_tx_port, 04      ;uart receiver port
;register alias
namereg sd, tx_data            ;data to be tx by uart
...
tx_one_byte:
   input s6, rd_flag_port
   test s6, 08                    ;check uart_tx_full
   jump nz, tx_one_byte           ;yes, keep on waiting
   output tx_data, uart_tx_port ;no, write to uart tx fifo
   return
```

Since PicoBlaze's processing speed is much higher than the UART's transmission speed, we must prevent buffer overflow. The routine keeps on checking the status of the transmitting FIFO buffer, and writes data only when the buffer is not full.

The task of dumping data RAM requires the most work. It displays the data RAM address and contents as an 8-by-8 table, which lists the byte address first and then the 8 bytes of data in hexadecimal format, as in

```
001000 00 0F 00 09 00 04 00 03
010000 00 00 FF 1D 00 00 00 19
. . .
111000 00 00 00 00 00 FF FF FF
```

The routine consists of three major routines: `disp_ram_addr`, which sends ASCII codes to display the 5-bit base address in binary format; `disp_ram_data`, which sends ASCII codes to display 8 bytes of data; and `hex_to_ascii`, which converts a hexadecimal digit to the corresponding ASCII code.

The complete code is shown in Listing 17.3. It includes detailed comments to explain operation of the subroutines. The unmodified subroutines of Listing 17.1 are omitted.

Listing 17.3 Square program with a UART console

```
;=============================================================
; square circuit with UART and multiplier interface
;=============================================================
;program operation:
;  - read a and b from switch
;  - calculate a*a + b*b
;  - display data on HyperTerminal and 7-seg led

;=============================================================
; data constants
;=============================================================
;selected ASCII codes
constant ASCII_0,  30
constant ASCII_1,  31
constant ASCII_2,  32
constant ASCII_3,  33
constant ASCII_a,  61
constant ASCII_b,  62
constant ASCII_c,  63
constant ASCII_d,  64
constant ASCII_o,  6F
constant ASCII_r,  72
constant ASCII_E,  45
constant ASCII_S,  53
constant ASCII_Q,  51
constant ASCII_D_U,44    ; uppercase D
constant ASCII_GT, 3E    ; >
constant ASCII_SP, 20    ; space
constant ASCII_CR, 0D    ; carriage return
constant ASCII_LF, 0A    ; line feed

;=============================================================
; data RAM address alias
;=============================================================
constant a_lsb, 00
constant b_lsb, 02
constant aa_lsb, 04
constant aa_msb, 05
constant bb_lsb, 06
constant bb_msb, 07
constant aabb_lsb, 08
constant aabb_msb, 09
constant aabb_cout, 0A
constant led0, 10
constant led1, 11
constant led2, 12
constant led3, 13

;=============================================================
; register alias
;=============================================================
;commonly used local variables
```

```
namereg s0, data        ;reg for temporary data
namereg s1, addr        ;reg for temporary mem & i/o port addr
namereg s2, i           ;general-purpose loop index
;global variables
namereg sc, switch_a_b ;ram offset for current switch input
namereg sd, tx_data     ;data to be tx by uart

;=====================================================
; port alias
;=====================================================
;--------------input port definitions-------------------
constant rd_flag_port, 00
; 4 flags (xxxxtrsc):
;   t: uart tx full
;   r: uart rx not empty
;   s: s button flag
;   c: c button flag
constant sw_port,          01    ;8-bit switches
constant uart_rx_port,     02    ;uart receiver port
constant mult_prod0_port, 03    ;multiplication product 8 LSBs
constant mult_prod1_port, 04    ;multiplication product 8 MSBs
;--------------output port definitions-------------------
constant sseg0_port,       00    ;7-seg led 0
constant sseg1_port,       01    ;7-seg led 1
constant sseg2_port,       02    ;7-seg led 2
constant sseg3_port,       03    ;7-seg led 3
constant uart_tx_port,     04    ;uart receiver port
constant mult_src0_port, 05    ;multiplier operand 0
constant mult_src1_port, 06    ;multiplier operand 1

;=====================================================
; main program
;=====================================================
;calling hierarchy:
;
;main
;  - init
;      - tx_prompt
;          - tx_one_byte
;  - proc_btn
;      - init
;  - proc_uart
;      - tx_prompt
;      - init
;      - proc_uart_err
;          - tx_one_byte
;      - dump_mem
;          - tx_prompt
;          - disp_ram_addr
;              - tx_one_byte
;          - disp_ram_data
;              - tx_one_byte
;              - get_upper_nibble
```

```
;               - get_lower_nibble
;               - hex_to_ascii
;   - square
;       - mult_hard
;   - load_led_pttn
;       - get_lower_nibble
;       - get_upper_nibble
;       - hex_to_led
;   - disp_led
;
; ================================================================
    call init                   ;initialization
forever:
    ;main loop body
    call proc_btn               ;check & process buttons
    call proc_uart              ;check & process uart rx
    call square                 ;calculate square
    call load_led_pttn          ;store led patterns to ram
    call disp_led               ;output led pattern
    jump forever

;================================================================
;routine: init
;  function: perform initialization, clear register/ram
;  output register:
;    switch_a_b: cleared to 0
;  temp register: data, i
;================================================================
init:
    ;clear memory
    load i, 40                  ;unitize loop index to 64
    load data, 00
clr_mem_loop:
    store data, (i)
    sub i, 01                   ;dec loop index
    jump nz, clr_mem_loop       ;repeat until i=0
    ;clear register
    load switch_a_b, 00
    call tx_prompt
    return

;================================================================
;routine: proc_uart
;  function: read uart input char:
;    a or b: read a or b from switch;
;    c: clear;   d: dump/display data ram   other: error
;  input reg: s3 (input port flag)
;  temp register used: data
;    s4: store received uart char or 00 (no uart input)
;================================================================
proc_uart :
    test s3, 04                 ;check uart rx status
    jump z, uart_rx_done        ;go to done if rx empty
```

```
    ;process received char
    input s4, uart_rx_port   ;get char
    ;check if received char is a
    compare s4, ASCII_a      ;check ASCII a
    jump nz, chk_ascii_b     ;no, check next
    input data, sw_port      ;get switch
    store data, a_lsb        ;write a to data ram
    call tx_prompt           ;new prompt line
    jump uart_rx_done
chk_ascii_b:
    ;check if received char is b
    compare s4, ASCII_b      ;check ASCII b
    jump nz, chk_ascii_c     ;no, check next
    input data, sw_port      ;get switch
    store data, b_lsb        ;write b to data ram
    call tx_prompt           ;new prompt line
    jump uart_rx_done
chk_ascii_c:
    ;check if received char is c
    compare s4, ASCII_c      ;check ASCII c
    jump nz, chk_ascii_d     ;no check next
    call init                ;clear
    jump uart_rx_done
chk_ascii_d:
    ;check if received char is d
    compare s4, ASCII_d        ;check ASCII d
    jump nz, ascii_undefined
    call dump_mem            ;dump/display ram
    jump uart_rx_done
ascii_undefined:
    ;undefined char
    call proc_uart_error
uart_rx_done:
    return

;=============================================================
;routine: proc_uart_error
;  function: display "Error" for unknown uart char
;=============================================================
proc_uart_error:
    load tx_data, ASCII_LF
    call tx_one_byte         ;transmit LF
    load tx_data, ASCII_CR
    call tx_one_byte         ;transmit CR
    load tx_data, ASCII_SP
    call tx_one_byte         ;transmit SP
    call tx_one_byte         ;transmit SP
    load tx_data, ASCII_E
    call tx_one_byte         ;transmit E
    load tx_data, ASCII_r
    call tx_one_byte         ;transmit r
    load tx_data, ASCII_r
    call tx_one_byte         ;transmit r
```

```
    load tx_data, ASCII_o
    call tx_one_byte          ;transmit o
    load tx_data, ASCII_r
    call tx_one_byte          ;transmit r
    call tx_prompt
    return

;=================================================================
;routine: dump_mem
;  function:  when d received, dump 64 bytes of ram as
;    001000 XX XX XX XX XX XX XX
;    010000 XX XX XX XX XX XX XX
;    . . .
;    111000 XX XX XX XX XX XX XX
;  temp register used:
;    s3: as outer loop index
;    s4: ram base address
;=================================================================
dump_mem:
    load s3, 00               ;addr used as loop index
dump_loop:
    ;loop body
    load s4, s3               ;get ram base addr (xxx000)
    sl0 s4
    sl0 s4
    sl0 s4
    call disp_ram_addr
    call disp_ram_data
    add s3, 01                ;inc loop index
    compare s3, 08
    jump nz, dump_loop        ;loop not reach 8 yet
    call tx_prompt            ;new prompt
    return

;=================================================================
;routine: tx_prompt
;  function: generate prompt "SQ>"
;  temp register: tx_data
;=================================================================
tx_prompt:
    load tx_data, ASCII_LF
    call tx_one_byte          ;transmit LF
    load tx_data, ASCII_CR
    call tx_one_byte          ;transmit CR
    load tx_data, ASCII_S
    call tx_one_byte          ;transmit S
    load tx_data, ASCII_Q
    call tx_one_byte          ;transmit Q
    load tx_data, ASCII_GT
    call tx_one_byte          ;transmit >
    load tx_data, ASCII_SP
    call tx_one_byte          ;transmit SP
    return
```

```
265
   ;=================================================================
   ;routine: disp_ram_addr
   ;  function: display 6-bit ram addr
   ;     bbb000
270 ;  input register:
   ;     s4: base address
   ;  temp register:
   ;     i, s7: 1-bit mask
   ;=================================================================
275 disp_ram_addr:
      ;new line
      load tx_data, ASCII_LF
      call tx_one_byte          ;transmit LF
      load tx_data, ASCII_CR
280   call tx_one_byte          ;transmit CR
      load tx_data, ASCII_SP
      call tx_one_byte          ;transmit SP
      call tx_one_byte          ;transmit SP
      ;initialize the loop index and mask
285   load i, 06                ;addr used as loop index
      load s7, 20               ;set mask to 0010_0000
   tx_loop:
      ;loop body
      load tx_data, ASCII_1     ;load default ASCII 1
290   test s7, s4               ;check the bit
      jump nz, tx_01            ;the bit is 1
      load tx_data, ASCII_0;    ;the bit is 0, load ASCII 0
   tx_01:
      call tx_one_byte          ;transmit the ASCII 1 or 0
295   ;update loop index and mask
      sr0 s7                    ;shift mask bit
      sub  i, 01                ;dec loop index
      jump nz, tx_loop          ;loop not reach 0 yet
      ;done with loop, send ASCII space
300   load tx_data, ASCII_SP    ;load ASCII SP
      call tx_one_byte          ;transmit SP
      return

   ;=================================================================
305 ;routine: disp_ram_data
   ;  function: 8-byte data in form of
   ;     00 11 22 33 44 55 66 77 88
   ;  input register:
   ;     s4: ram base address (xxx000)
310 ;  temp register: i, addr, data
   ;=================================================================
   disp_ram_data:
      ;initialize the loop index and mask
      load i, 08                ;addr used as loop index
315 d_ram_loop:
      ;loop body
      load addr, s4
```

```
    add addr, i
    sub addr, 01              ;calculate addr offset
320 ;send upper nibble
    fetch data, (addr)
    call get_upper_nibble
    call hex_to_ascii         ;convert to ascii
    load tx_data, data
325 call tx_one_byte
    ;send lower nibble
    fetch data, (addr)
    call get_lower_nibble
    call hex_to_ascii         ;convert to ascii
330 load tx_data, data
    call tx_one_byte
    ;send a space
    load tx_data, ASCII_SP;
    call tx_one_byte          ;transmit SP
335 sub i, 01                 ;dec loop index
    jump nz, d_ram_loop       ;loop not reach 0 yet
    return

;=============================================================
340 ;routine: hex_to_ascii
;   function: convert a hex number to ascii code
;             add 30 for 0-9, add 37 for A-F
;   input register: data
;=============================================================
345 hex_to_ascii:
    compare data, 0a
    jump c, add_30            ;0 to 9, offset 30
    add data, 07              ;a to f, extra offset 07
add_30:
350 add data, 30
    return

;=============================================================
;routine: tx_one_byte
355 ;   function: wait until uart tx fifo not full;
;             then write a byte to fifo
;   input register: tx_data
;   temp register:
;      s6: read port flag
360 ;=============================================================
tx_one_byte:
    input s6, rd_flag_port
    test s6, 08                   ;check uart_tx_full
    jump nz, tx_one_byte          ;yes, keep on waiting
365 output tx_data, uart_tx_port  ;no, write to uart tx fifo
    return

;=============================================================
;routine: square
370 ;   function: calculate a*a + b*b
```

```
;       data/result stored in ram started w/ SQ_BASE_ADDR
;   temp register: s3, s4, s5, s6, data
;=============================================================
square:
    ;calculate a*a
    fetch s3, a_lsb         ;load a
    fetch s4, a_lsb         ;load a
    call mult_hard          ;calculate a*a
    store s6, aa_lsb        ;store lower byte of a*a
    store s5, aa_msb        ;store upper byte of a*a
    ;calculate b*b
    fetch s3, b_lsb         ;load b
    fetch s4, b_lsb         ;load b
    call mult_hard          ;calculate b*b
    store s6, bb_lsb        ;store lower byte of b*b
    store s5, bb_msb        ;store upper byte of b*b
    ;calculate a*a+b*b
    fetch data, aa_lsb      ;get lower byte of a*a
    add data, s6            ;add lower byte of a*a+b*b
    store data, aabb_lsb    ;store lower byte of a*a+b*b
    fetch data, aa_msb      ;get upper byte of a*a
    addcy data, s5          ;add upper byte of a*a+b*b
    store data, aabb_msb    ;store upper byte of a*a+b*b
    load data, 00           ;clear data, but keep carry
    addcy data, 00          ;get carry from previous +
    store data, aabb_cout   ;store carry of a*a+b*b
    return

;=============================================================
;routine: mult_hard
;   function: 8-bit unsigned multiplication using
;             external combinational multiplier;
;   input register:
;      s3: multiplicand
;      s4: multiplier
;   output register:
;      s5: upper byte of product
;      s6: lower byte of product
;   temp register:
;=============================================================
mult_hard:
    output s3, mult_src0_port
    output s4, mult_src1_port
    input s5, mult_prod1_port
    input s6, mult_prod0_port
    return

;=============================================================
;The following are the same as the previous listings:
;   proc_btn, load_led_pttn, disp_led
;   hex_to_led, get_lower_nibble, get_upper_nibble
;   ...
;=============================================================
```

17.5.4 HDL code development

The new square circuit adds a UART and a combinational multiplier to an I/O interface. The former is the module discussed in Section 8.4, and the latter can be inferred from the HDL's * operator. The decoding and multiplexing parts of HDL code in Listing 17.2 can be expanded to accommodate the two new peripherals. The complete HDL code is shown in Listing 17.4. The detailed I/O port address assignment can be found in the header section of Listing 17.3.

Listing 17.4 PicoBlaze with UART console and multiplier interface

```
module pico_uart
   (
    input wire clk, reset,
    input wire [7:0] sw,
    input wire rx,
    input wire [1:0] btn,
    output wire tx,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   // KCPSM3/ROM signals
   wire [9:0] address;
   wire [17:0] instruction;
   wire [7:0] port_id, out_port;
   reg [7:0] in_port;
   wire  write_strobe, read_strobe;
   // I/O port signals
   // output enable
   reg [6:0] en_d;
   // four-digit seven-segment led display
   reg [7:0] ds3_reg, ds2_reg, ds1_reg, ds0_reg;
   // two pushbuttons
   reg btnc_flag_reg, btns_flag_reg;
   wire btnc_flag_next, btns_flag_next;
   wire set_btnc_flag, set_btns_flag, clr_btn_flag;
   // uart
   wire [7:0] rx_char;
   wire rd_uart, rx_not_empty, rx_empty;
   wire wr_uart, tx_full;
   // multiplier
   reg [7:0] m_src0_reg, m_src1_reg;
   wire [15:0] prod;


   //body
   // ==================================================
   //  I/O modules
   // ==================================================
   disp_mux disp_unit
      (.clk(clk), .reset(reset),
       .in3(ds3_reg), .in2(ds2_reg), .in1(ds1_reg),
```

```
        .in0(ds0_reg), .an(an), .sseg(sseg));
45    debounce btnc_unit
         (.clk(clk), .reset(reset), .sw(btn[0]),
          .db_level(), .db_tick(set_btnc_flag));
      debounce btns_unit
         (.clk(clk), .reset(reset), .sw(btn[1]),
50        .db_level(), .db_tick(set_btns_flag));
      uart uart_unit
         (.clk(clk), .reset(reset), .rd_uart(rd_uart),
          .wr_uart(wr_uart), .rx(rx),
          .w_data(out_port), .tx_full(tx_full),
55        .rx_empty(rx_empty), .r_data(rx_char), .tx(tx));
      // combinational multiplier
      assign prod = m_src0_reg * m_src1_reg;
      // ======================================================
      //  KCPSM and ROM instantiation
60    // ======================================================
      kcpsm3 proc_unit
         (.clk(clk), .reset(1'b0), .address(address),
          .instruction(instruction), .port_id(port_id),
          .write_strobe(write_strobe), .out_port(out_port),
65        .read_strobe(read_strobe), .in_port(in_port),
          .interrupt(1'b0), .interrupt_ack());
      uart_rom rom_unit
         (.clk(clk), .address(address),
          .instruction(instruction));
70    // ======================================================
      //  output interface
      // ======================================================
      //     outport port id:
      //       0x00: ds0
75    //       0x01: ds1
      //       0x02: ds2
      //       0x03: ds3
      //       0x04: uart_tx_fifo
      //       0x05: m_src0
80    //       0x06: m_src1
      // ======================================================
      // registers
      always @(posedge clk)
         begin
85          if (en_d[0])
               ds0_reg <= out_port;
            if (en_d[1])
               ds1_reg <= out_port;
            if (en_d[2])
90             ds2_reg <= out_port;
            if (en_d[3])
               ds3_reg <= out_port;
            if (en_d[5])
               m_src0_reg <= out_port;
95          if (en_d[6])
               m_src1_reg <= out_port;
```

```
   end

// decoding circuit for enable signals
always @*
   if (write_strobe)
      case (port_id[2:0])
         3'b000: en_d = 7'b0000001;
         3'b001: en_d = 7'b0000010;
         3'b010: en_d = 7'b0000100;
         3'b011: en_d = 7'b0001000;
         3'b100: en_d = 7'b0010000;
         3'b101: en_d = 7'b0100000;
         default: en_d = 7'b1000000;
      endcase
   else
      en_d = 7'b0000000;

assign wr_uart = en_d[4];
// ================================================
//  input interface
// ================================================
//    input port id
//      0x00: flag
//      0x01: switch
//      0x02: uart_rx_fifo
//      0x03: prod lower byte
//      0x04: prod upper byte
// ================================================
// input register (for flags)
always @(posedge clk)
   begin
      btnc_flag_reg <= btnc_flag_next;
      btns_flag_reg <= btns_flag_next;
   end
assign btnc_flag_next = (set_btnc_flag) ? 1'b1 :
                        (clr_btn_flag)  ? 1'b0 :
                         btnc_flag_reg;
assign btns_flag_next = (set_btns_flag) ? 1'b1 :
                        (clr_btn_flag)  ? 1'b0 :
                         btns_flag_reg;

// decoding circuit for clear signals
assign clr_btn_flag = read_strobe && (port_id[2:0]==3'b000);
assign rd_uart = read_strobe && (port_id[2:0]==3'b010);
// input multiplexing
assign rx_not_empty = ~rx_empty;
always @*
   case(port_id[2:0])
      3'b000: in_port = {4'b0, tx_full, rx_not_empty,
                         btns_flag_reg, btnc_flag_reg};
      3'b001: in_port = sw;
      3'b010: in_port = rx_char;
      3'b011: in_port = prod[7:0];
```

```
150          default: in_port = prod[15:8];
          endcase

   endmodule
```

17.6 BIBLIOGRAPHIC NOTES

The basic bibliographic information for this chapter is similar to that for Chapter 15. The downloaded `kcpsm` file contains a comprehensive UART and timer design example. The Xilinx Web site has pages for "PicoBlaze Forum" and "PicoBlaze User Resources," where additional PicoBlaze examples are available.

17.7 SUGGESTED EXPERIMENTS

17.7.1 Low-frequency counter I

An accurate low-frequency counter is discussed in Section 6.3.5. We can treat the period counter, division circuit, and binary-to-BCD conversion circuit as three I/O modules, and replace the top-level FSM with PicoBlaze. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.2 Low-frequency counter II

We can reduce the hardware of the frequency counter of Experiment 17.7.1 by replacing the division circuit and binary-to-BCD conversion circuit with software subroutines. Redesign the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.3 Auto-scaled low-frequency counter

An auto-scaled low-frequency counter is discussed in Experiment 6.5.5. We can use PicoBlaze to perform all non-time-critical functions. Redesign the circuit with PicoBlaze and minimal external hardware. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.4 Basic reaction timer with a software timer

The reaction timer is discussed in Experiment 6.5.6. We can redesign the circuit using PicoBlaze. One task of the design is to keep track of the elapsed time interval. This can be done by a software counting routine. Recall that a 50-MHz clock is used on the prototyping board and each instruction takes two clock cycles. We can create a counting loop to record the number of instructions executed and derive the time interval accordingly. Since the interval is at least in the millisecond range, multiple registers are needed for this purpose. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.5 Basic reaction timer with a hardware timer

We can repeat Experiment 17.7.4 with a customized hardware timer. The timer should be treated as an I/O peripheral. PicoBlaze can output a command to clear, start, or pause the timer, and can input the counter's content. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.6 Enhanced reaction timer

An enhanced reaction timer keeps track of the last four response times and the fastest response time, and displays the data on Windows HyperTerminal. We can design a console similar to that of Section 17.5. There should be three commands:

- `c`: clears all data
- `f`: displays the fastest response
- `r`: displays the time of the last four responses
- All other characters: display "error"

Expand the design in Experiment 17.7.4 or 17.7.5 to include this feature. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.7 Small-screen mouse scribble circuit

A small-screen mouse scribble circuit is discussed in Experiment 13.7.10. We can use PicoBlaze to monitor the activities of the mouse and update the video memory accordingly. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.8 Full-screen mouse scribble circuit

A full-screen mouse scribble circuit is discussed in Experiment 13.7.11. We can use PicoBlaze to monitor the activities of the mouse and update the video memory accordingly. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.9 Enhanced rotating banner

A VGA rotating banner circuit is discussed in Experiment 14.6.1. Instead of a fixed message, we can enhance this circuit by using a keyboard to enter the message dynamically. Assume that the message buffer is 20 characters long and its characters are updated in a first-in-first-out fashion. Redesign the circuit with PicoBlaze. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.10 Pong game

The complete pong game is discussed in Section 14.4. Some functions of the design can be implemented by PicoBlaze:

- Top-level control FSM
- Top-level two-second timer and two-digit decade counter
- The circuit that updates the paddle position, ball position, and ball velocities in Listing 13.5

Modify the original circuit, design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

17.7.11 Text editor

A UART terminal is discussed in Experiment 14.6.5. We can use PicoBlaze to obtain data and commands from the UART and update the tile memory accordingly. Design the I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

CHAPTER 18

PICOBLAZE INTERRUPT INTERFACE

18.1 INTRODUCTION

During normal program execution, a microcontroller *polls* the I/O peripherals (i.e., checks the status signals) and determines the course of action accordingly. An I/O peripheral is passive and waits for its turn. The *interrupt* is a mechanism that allows an external I/O peripheral to initiate the operation. It, as the name shows, interrupts normal program execution and starts a service routine for the I/O peripheral. For a microcontroller, the interrupt is usually reserved for a time-critical peripheral operation, which must be processed immediately. The PicoBlaze microcontroller provides support for simple interrupt-handling capability. In this chapter, we examine the PicoBlaze's interrupt mechanism and use an example to illustrate software and interface development.

18.2 INTERRUPT HANDLING IN PICOBLAZE

Interrupt handling is a coordinated effort between hardware and software. When an external peripheral needs service through interrupt, it asserts the `interrupt` signal of PicoBlaze. If the interrupt service is enabled, PicoBlaze completes execution of the current instruction, activates the `interrupt_ack` signal to acknowledge the acceptance of the interrupt request, and then implicitly executes the **call** 3FF instruction. When the instruction is executed, the current content of the program counter is saved in a stack and the 3FF address is loaded to the programmer counter. Note that the 3FF address is the last location in the instruction

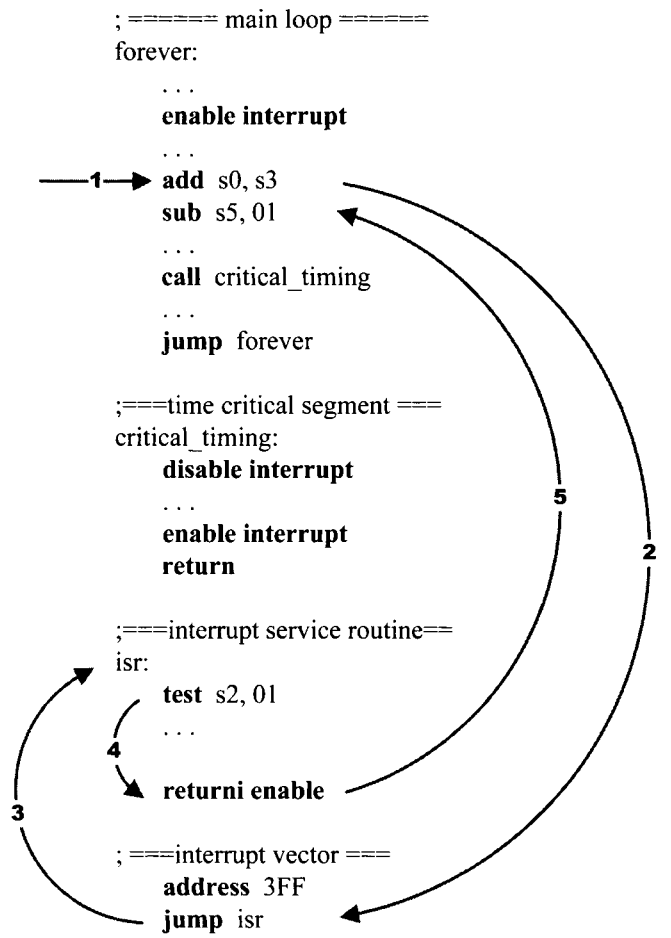


Figure 18.1 Interrupted flow.

memory and serves as the starting point of the interrupt service routine. It usually contains a **jump** instruction, which leads to the body of the service routine. The service should be ended with a **returni** instruction to return to the interrupted point and resume the previous execution.

18.2.1 Software processing

Four instructions are associated with interrupt, as discussed in Section 15.5.9. The **enable interrupt** and **disable interrupt** instructions enable and disable the interrupt request, and the two return-from-interrupt instructions, **returni enable** and **returni disable**, return execution to the interrupted point.

A typical program segment with interrupt service routine is shown in Figure 18.1. It generally consists of the following segments:

- *An initial* **enable interrupt** *instruction*: used to enable the interrupt service. This is needed since the interrupt request is disabled by default.

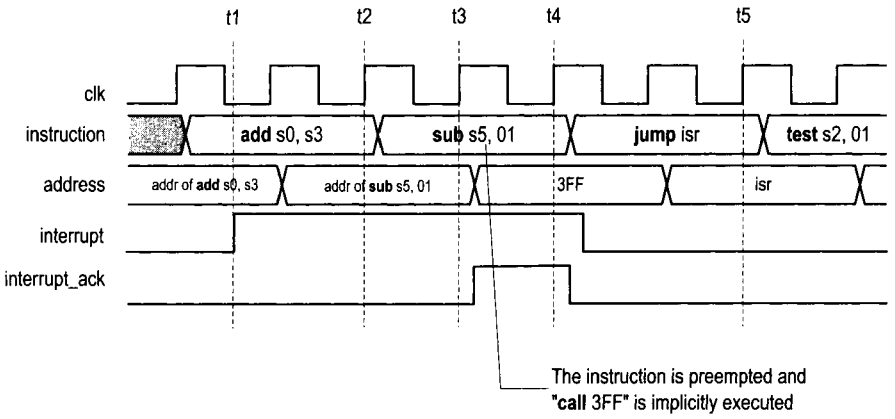


Figure 18.2 Timing diagram of an interrupt event.

- *A* **jump** *instruction in the end of the instruction memory* (i.e., 3FF): leads to the interrupt service routine.
- *Interrupt service routine*: the code that actually performs the requested service. The routine should be ended with a **returni** instruction.

A representative flow of an interrupt event is shown in Figure 18.1. We assume that the external I/O asserts the `interrupt` signal in the middle of the **add** `s0,s3` instruction. PicoBlaze performs the following steps in sequence:

1. Completes execution of the current execution.
2. Saves the content of the program counter, clears the interrupt flag, `i`, to zero, preserves the zero and carry flags, and loads the program counter with 3FF.
3. Executes the **jump** `isr` instruction in the 3FF address.
4. Performs the service routine.
5. Executes the **returni** instruction, in which the saved program counter and flags are restored.
6. Resumes the interrupted program and executes the **sub** `s5,01` instruction.

18.2.2 Timing

The detailed timing diagram of the previous interrupt event is shown in Figure 18.2. The basic sequence is:

- At `t1`: The external interrupt interface asserts the `interrupt` signal. PicoBlaze continues the normal operation to complete execution of the current **add** `s0,s3` instruction.
- At `t2`: PicoBlaze recognizes the interrupt and aborts the next instruction (**sub** `s5,01`) and implicitly executes the **call** 3FF instruction.
- At `t3`: PicoBlaze asserts the `interrupt_ack` signal. It also saves the address of the **sub** `s5,01` instruction, preserves the zero and carry flags, and clears the interrupt flag to 0.
- At `t4`: PicoBlaze loads and executes the instruction in address 3FF, **jump** `isr`. The external interrupt interface circuit acknowledges the `interrupt_ack` signal and deasserts the `interrupt` signal.

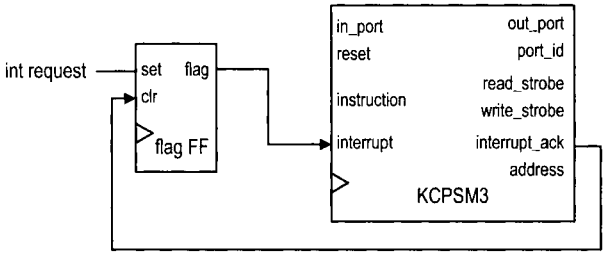


Figure 18.3 Interrupt interface with a single request.

- At t5: PicoBlaze starts the interrupt service routine.

Note that it requires up to five clock cycles from the time that the `interrupt` signal is asserted to the time that the first instruction of interrupt service routine is executed.

18.3 EXTERNAL INTERFACE

The nature of the interrupt request is similar to that of a single-access port discussed in Section 17.3.2. After the request is accepted, it must be cleared so that the same request will not be processed multiple times. The flag FF discussed in Section 8.2.4 can be used for this purpose.

18.3.1 Single interrupt request

If there is only one I/O peripheral in a PicoBlaze system that can generate an interrupt request, we just need a single flag FF in the interrupt interface circuit, as shown in Figure 18.3. When the service is required, the external I/O circuit asserted the `int request` signal for one clock cycle, which sets the flag FF to 1 and activates the `interrupt` input of PicoBlaze. If the interrupt is enabled in PicoBlaze, it acknowledges acceptance of the request by asserting the `interrupt_ack` signal for one clock cycle, which clears the flag FF to 0.

18.3.2 Multiple interrupt requests

Processing a PicoBlaze system with two or more interrupt requests is more involved. The PicoBlaze microcontroller must determine which peripheral issues the request and clear the corresponding flag FF after the request is accepted. This needs the coordination of the hardware interface and the interrupt service routine.

The interrupt interface with two requests is shown in Figure 18.4. The two individual requests, `int request0` and `int request1`, are connected to two flag FFs, and the output signals of the FFs are passed to an or gate to generate the final interrupt request signal. In addition, the two signals are also routed to the input multiplexer. If at least one request is asserted, the `interrupt` signal of PicoBlaze is asserted. When PicoBlaze senses the request, it does not know which peripheral or whether both peripherals issue the request. The interrupt service routine must first input the two request signals and check their values according to the assigned priority, and then perform the corresponding service.

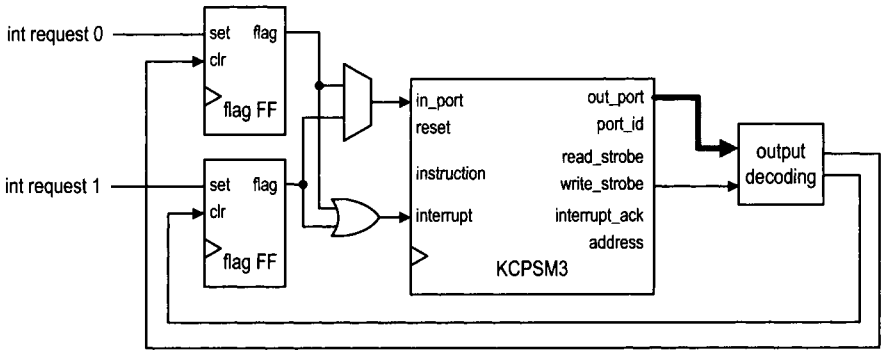


Figure 18.4 Interrupt interface with two requests.

In addition, PicoBlaze also needs to clear the corresponding flag FF. The `interrupt_ack` signal cannot be used for this purpose because it is not known which peripheral's request is accepted when the `interrupt_ack` signal is asserted. Instead, we need to use a special output decoding circuit to generate a clear tick. The `clr` signal of each flag FF is assigned to a unique port id. In the interrupt service routine, we add an **output** instruction after determining which interrupt request is accepted. The instruction does not actually output any data. It is used to generate a single-clock-cycle tick to clear the corresponding flag FF.

To reduce the software overhead and increase response speed, we can design an *interrupt controller* to facilitate the process. This approach is discussed in Experiment 18.7.5.

18.4 SOFTWARE DEVELOPMENT CONSIDERATIONS

18.4.1 Interrupt as an alternative scheduling scheme

Recall that a microcontroller-based application usually follows a simple polling program structure:

```
      call initialization_routine
forever:
      call task1_routine;
      call task2_routine;
       ...
      call taskn_routine;
      jump forever;
```

Some tasks may involve I/O operations. During execution, the microcontroller checks the I/O status in turn and takes actions accordingly. The program structure implicitly implements a *round-robin* schedule, in which each task waits in turn to be executed. This scheme can work properly if the loop interval is short enough so that each I/O request can be checked and processed in a timely manner. In some applications, there may exist one or two time-critical I/O requests that require immediate attention. The interrupt mechanism provides a way to alter the original schedule and gives certain tasks higher priorities.

Since an interrupt can occur at any time, the original loop must consider the frequency of interrupt and the required service time of each interrupt request. This can be complicated if there are multiple interrupt requests and the service routine is involved.

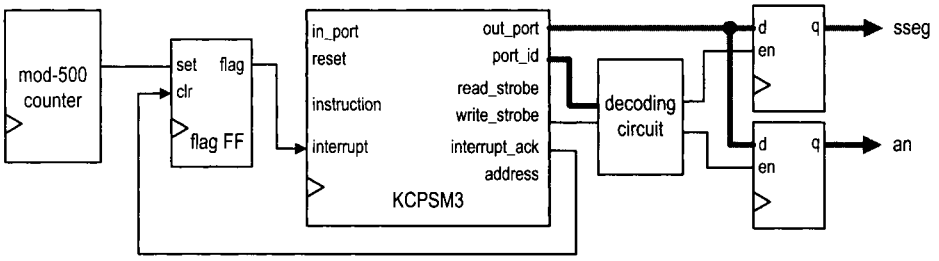


Figure 18.5 Interrupt interface with a timer.

18.4.2 Development of an interrupt service routine

The interrupt service routine is somewhat like a subroutine. It suspends normal program execution, performs an independent task, and then resumes the previous execution. However, unlike a subroutine call, an interrupt can occur any time. To resume execution later, the service routine must save the *current state* (also known as the *context*) of the PicoBlaze processor. In other words, the service routine must save all registers used in service routine computation and then restore them before returning to normal execution. This process is known as *context switching*.

Since PicoBlaze is a compact 8-bit microcontroller, the hardware support for context switching and scheduling is very limited. We should use the polling scheme in general and keep the interrupt structure simple and straightforward. Instead of worrying about context switching, we can allocate several dedicated registers to be used exclusively in the interrupt service routine.

18.5 DESIGN EXAMPLE

The square circuit of Chapter 17 uses a seven-segment LED display to show the values of input operands and result. We use the predesigned LED multiplexing module, `disp_mux`, for this purpose. The design of this module is discussed in Section 4.5.1. It consists of a large counter to generate slow enable pulses and a multiplexing circuit to route the input patterns.

To save hardware, we can implement this functionality in software and let PicoBlaze control the 4-bit enable signal, `an`, and the 8-bit LED signal, `sseg`, of the four-digit LED display directly. To generate a visually continuous pattern, the enable pulse and LED patterns must be refreshed at a constant rate, as shown in Figure 4.6. While using pure software to keep track of time is possible, the code is tedious and error-prone. We use a dedicated hardware timer and PicoBlaze's interrupt facility to perform the task. The required hardware and software modifications are illustrated in the following subsections.

18.5.1 Interrupt interface

The block diagram of the timer and interrupt interface, as well as the new output buffers, is shown in Figure 18.5. The timer is a mod-500 counter and generates a single-clock-cycle tick every 500 clock cycles. Since the 50-MHz clock is used for the timer, the period of the tick is 0.01 ms. Because there is only one interrupt request, we use the flag FF scheme

discussed in Section 18.3.1 for the interrupt interface. The tick sets the flag FF and activates the `interrupt` signal of PicoBlaze.

18.5.2 Interrupt service routine development

To keep track of the elapsed time, PicoBlaze counts the number of timer ticks. As discussed in Section 18.4.2, we want to keep the interrupt service routine simple and use two dedicated registers, `count_msb` and `count_lsb`, for this task. The two registers are cascaded as a 16-bit register and are incremented each time the interrupt service routine is called. They can count to 0.6 second (i.e., $2^{16} * 0.01$ ms). The interrupt-related code segment is

```
   namereg se, count_msb   ;timer tick count 8 MSBs
   namereg sf, count_lsb   ;timer tick count 8 LSBs
   ...
;interrupt service routine
int_service_routine:
   add count_lsb, 01       ;inc 16-bit counter
   addcy count_msb, 00
   returni enable

;interrupt vector
   address 3FF
   jump int_service_routine
```

18.5.3 Assembly code development

With the timing information available, we can derive a new subroutine, `display_mux_out`, for the LED display. This routine replaces the `disp_led` routine used in Chapter 17. Two new output buffers are needed to store the an and sseg signals, as shown in Figure 18.5. The main task of the subroutine is to store the an pattern, which can be "1110", "1101", "1011", or "0111", and the corresponding seven-segment LED pattern to the registers periodically. As discussed in Section 4.5.1, the refreshing rate should be around from a few hundred to a few thousand hertz. In our code we update these registers every 2^{10} ticks, which is about 10 ms. We also use a register, `led_pos`, to keep track of the current display position (i.e., one of the four LED displays).

To incorporate the new interrupt feature into Listing 17.3, the code is modified as follows:

- Add new port and register definitions.
- Replace the original `disp_led` routine with the `display_mux_out` routine.
- Add the **enable interrupt** instruction in the `init` routine to enable interrupt handling.
- Initialize the `led_pos`, `count_msb`, and `count_lsb` registers in the `init` routine.
- Add the interrupt service routine.

The modified portion of the assembly code is shown in Listing 18.1.

Listing 18.1 Square program with interrupt interface

```
...
;register alias
namereg sb, led_pos     ;led disp position (0, 1, 2 or 3)
namereg se, count_msb   ;timer tick count 8 MSBs
namereg sf, count_lsb   ;timer tick count 8 LSBs
...
```

```
;output port definitions
constant an_port, 00
constant sseg_port, 01
...
; main program
   call init                ;initialization
forever:
   ;main loop body
   call proc_btn            ;check & process buttons
   call square              ;calculate square
   call load_led_pttn       ;store led patterns to ram
   call display_mux_out     ; multiplex led patterns
   jump forever

;=============================================================
;routine: init
;=============================================================
init:
   enable interrupt
   ...
   load led_pos, 00
   load count_msb, 00
   load count_lsb, 00
   return

;=============================================================
;routine: display_mux_out
;  function: generate enable pulse & led pattern
;            for 4-digit 7-segment led display
;  input register:
;     count_msb, count_lsb: timer count
;     led_pos: current led position
;  output register:
;     led_pos: updated led position
;  tmp register: data, addr
;=============================================================
display_mux_out:
   compare count_msb, 02  ;count=00000100_00000000
   jump c, mux_out_done
   ;clear time counter (count > 20)
   load count_lsb, 00
   load count_msb, 00
   ;update 7-segment led position
   add led_pos, 01
   compare led_pos, 04
   jump nz, gen_an_signal
   load led_pos, 00       ;led_pos wraps around
gen_an_signal:
   ;generate 4-bit anode enable signal
   load data, 0E          ;xxxx_1110
   compare led_pos, 00
   jump z, shift_an_0
   compare led_pos, 01
```

```
    jump z, shift_an_1
    compare led_pos, 02
    jump z, shift_an_2
    sll data                    ;shift 1110 3 times
shift_an_2:
    sll data                    ;shift 1110 2 times
shift_an_1:
    sll data                    ;shift 1110 1 times
shift_an_0:
    output data, an_port
    ;output 7-seg led pattern
    load addr, led0
    add addr, led_pos
    fetch data, (addr)
    output data, sseg_port
mux_out_done:
    return

;=============================================================
;routine: interrupt service routine
;  function: increment 16-bit counter
;  input register:
;     count_msb, count_lsb: timer count
;  output register:
;     count_msb, count_lsb: incremented
;=============================================================
int_service_routine:
    add count_lsb, 01       ;inc 16-bit counter
    addcy count_msb, 00
    returni enable

;=============================================================
;interrupt vector
;=============================================================
    address 3FF
    jump int_service_routine

;=============================================================
;The following are the same as the previous listings:
;  proc_btn, load_led_pttn,
;  hex_to_led, get_lower_nibble, get_upper_nibble
;  square, mult_soft
;  ...
;=============================================================
```

18.5.4 HDL code development

The I/O interface of the interrupt-based square circuit includes three parts. The input interface is similar to that in Section 17.4. The output interface consists of a decoding circuit and two output registers for the an and sseg signals, as shown on the right of Figure 18.5. The interrupt interface consists of a timer and a flag FF, as shown on the

left of Figure 18.5. The HDL code basically follows the block diagram and is shown in Listing 18.2.

Listing 18.2 PicoBlaze-based square circuit with interrupt

```
module pico_int
   (
    input wire clk, reset,
    input wire [7:0] sw,
    input wire [1:0] btn,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
   // KCPSM3/ROM signals
   wire [9:0] address;
   wire [17:0] instruction;
   wire [7:0] port_id, out_port;
   reg [7:0] in_port;
   wire  write_strobe, read_strobe;
   wire interrupt, interrupt_ack;
   // I/O port signals
   // output enable
   reg [1:0] en_d;
   // four-digit seven-segment led display
   reg [7:0] sseg_reg;
   reg [3:0] an_reg;
   // two pushbuttons
   reg btnc_flag_reg, btns_flag_reg;
   wire btnc_flag_next, btns_flag_next;
   wire set_btnc_flag, set_btns_flag, clr_btn_flag;
   // interrupt related signals
   reg [8:0] timer_reg;
   wire [8:0] timer_next;
   wire ten_us_tick;
   reg timer_flag_reg;
   wire timer_flag_next;

   //body
   // ====================================================
   //  I/O modules
   // ====================================================
   debounce btnc_unit
      (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(set_btnc_flag));
   debounce btns_unit
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(set_btns_flag));
   // ====================================================
   //  KCPSM and ROM instantiation
   // ====================================================
   kcpsm3 proc_unit
      (.clk(clk), .reset(1'b0), .address(address),
```

```
    .instruction(instruction), .port_id(port_id),
    .write_strobe(write_strobe), .out_port(out_port),
    .read_strobe(read_strobe), .in_port(in_port),
    .interrupt(interrupt), .interrupt_ack(interrupt_ack));
int_rom rom_unit
   (.clk(clk), .address(address),
    .instruction(instruction));
// ================================================
//  output interface
// ================================================
//    outport port id:
//      0x00: an
//      0x01: ssg
// ================================================
// registers
always @(posedge clk)
   begin
      if (en_d[0])
         an_reg <= out_port[3:0];
      if (en_d[1])
         sseg_reg <= out_port;
   end
assign an = an_reg;
assign sseg = sseg_reg;
// decoding circuit for enable signals
always @*
   if (write_strobe)
      case (port_id[0])
         1'b0: en_d = 2'b01;
         1'b1: en_d = 2'b10;
      endcase
   else
      en_d = 2'b00;
// ================================================
//  input interface
// ================================================
//    input port id
//      0x00: flag
//      0x01: switch
// ================================================
// input register (for flags)
always @(posedge clk)
   begin
      btnc_flag_reg <= btnc_flag_next;
      btns_flag_reg <= btns_flag_next;
   end
assign btnc_flag_next = (set_btnc_flag) ? 1'b1 :
                        (clr_btn_flag)  ? 1'b0 :
                         btnc_flag_reg;
assign btns_flag_next = (set_btns_flag) ? 1'b1 :
                        (clr_btn_flag)  ? 1'b0 :
                         btns_flag_reg;
// decoding circuit for clear signals
```

```
   assign clr_btn_flag = read_strobe && (port_id[0]==1'b0);
   // input multiplexing
   always @*
      case(port_id[0])
         1'b0: in_port = {6'b0, btns_flag_reg, btnc_flag_reg};
         1'b1: in_port = sw;
      endcase

   // ==========================================================
   //  interrupt interface
   // ==========================================================
   // 10 us counter
   always @(posedge clk)
      timer_reg <= timer_next;
   assign ten_us_tick = (timer_reg==499);
   assign timer_next = ten_us_tick ? 0 : timer_reg + 1;
   // 10 us tick flag
   always @(posedge clk)
       timer_flag_reg <= timer_flag_next;
   assign timer_flag_next = (ten_us_tick) ? 1'b1 :
                            (interrupt_ack) ? 1'b0 :
                             timer_flag_reg;
   // interrupt request
   assign interrupt = timer_flag_reg;

endmodule
```

18.6 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapters 15 to 17.

18.7 SUGGESTED EXPERIMENTS

18.7.1 Alternative timer interrupt service routine

The interrupt service routine in Listing 18.1 uses two dedicated registers to record the number of timer ticks. The two registers thus cannot be used for other computation. An alternative is to use 2 bytes of the data RAM for this purpose and use the registers only temporarily in the service routine. Since an interrupt can occur anytime, we must save and restore the corresponding registers. For example, if the `s0` and `s1` registers are used in the service routine for computation, their contents must be saved when the service routine is invoked and then restored later when the computation is completed. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

18.7.2 Programmable timer

We can replace the mod-500 counter of Section 18.5 with a general mod-m counter and thus make the timer "programmable." The new timer operates as follows:

- m is a 12-bit unsigned number.

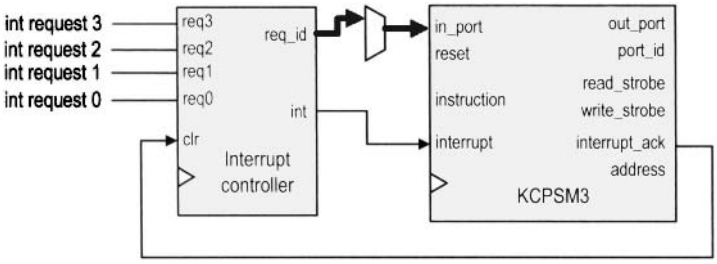


Figure 18.6 Interrupt interface with a four-request interrupt handler.

- The four LSBs of m is "1111".
- The timer has an 8-bit register to store the eight MSBs of m. The register is treated as a new output port of PicoBlaze.
- A new pushbutton controls the loading of the register. When it is pressed, PicoBlaze inputs the value from the 8-bit switch and outputs the value to the timer's register.

Design the new I/O interface, derive the assembly and HDL codes, and compile and synthesize the circuit. Load different values in the timer and observe what happens to the LED display.

18.7.3 Set-button interrupt service routine

In the square circuit discussed in Section 17.4, the `s` button is used to load the a and b operands from the 8-bit switch. Its status is polled continuously in the main loop. We can revise this portion of the code and use an interrupt mechanism to perform this task. The interrupt service routine involves several temporary registers, and they must be saved and restored properly, as discussed in Experiment 18.7.1. Design the new I/O interface, derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

18.7.4 Interrupt interface with two requests

Assume that we want to implement both the timer interrupt request of Listing 18.1 and the set-button interrupt request of Experiment 18.7.3 in a PicoBlaze system. Follow the discussion in Section 18.3.2 to design the new interrupt interface and interrupt service routine. Derive the assembly and HDL codes, compile and synthesize the circuit, and verify its operation.

18.7.5 Four-request interrupt controller

An interrupt controller helps the processor to process multiple interrupt requests. The block diagram of a four-request interrupt controller is shown in Figure 18.6. The interrupt controller should contain four flag FFs and a special priority encoding circuit. If one or more interrupt requests are activated, the controller determines which request has the highest priority, places its 2-bit code on the `req_id` port, and asserts the `int` signal. When PicoBlaze asserts the `interrupt_ack` signal, the controller clears the corresponding flag. For simplicity, we assume that `int_request_3` has the highest priority and `int_request_0` has the lowest priority.

Derive HDL code for the interrupt controller and repeat Experiment 18.7.4 using the new controller (the two unused interrupt requests can be tied to 0).

APPENDIX A

SAMPLE VERILOG TEMPLATES

A.1 NUMBERS AND OPERATORS

A.1.1 Sized and unsized numbers

number	stored value	comment
5'b11010	11010	
5'b11_010	11010	_ ignored
5'o32	11010	
5'h1a	11010	
5'd26	11010	
5'b0	00000	0 extended
5'b1	00001	0 extended
5'bz	zzzzz	z extended
5'bx	xxxxx	x extended
5'bx01	xxx01	x extended
-5'b00001	11111	2's complement of 00001
'b11010	00000000000000000000000000011010	extended to 32 bits
'hee	00000000000000000000000011101110	extended to 32 bits
1	00000000000000000000000000000001	extended to 32 bits
-1	11111111111111111111111111111111	extended to 32 bits

A.1.2 Operators

Type of operation	Operator symbol	Description	Number of operands
Arithmetic	+	addition	2
	-	subtraction	2
	*	multiplication	2
	/	division	2
	%	modulus	2
	**	exponentiation	2
Shift	>>	logical right shift	2
	<<	logical left shift	2
	>>>	arithmetic right shift	2
	<<<	logical left shift	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
Equality	==	equality	2
	!=	inequality	2
	===	case equality	2
	!==	case inequality	2
Bitwise	~	bitwise negation	1
	&	bitwise and	2
	|	bitwise or	2
	^	bitwise xor	2
Reduction	&	reduction and	1
	|	reduction or	1
	^	reduction xor	1
Logical	!	logical negation	1
	&&	logical and	2
	||	logical or	2
Concatenation	{ }	concatenation	any
	{ { } }	replication	any
Conditional	? :	conditional	3

A.2 GENERAL VERILOG CONSTRUCTS

A.2.1 Overall code structure

Listing A.1 Overall code structure

```
module bin_counter
   // optional parameter declaration
   #(parameter N=8)    // default 8
   // port declaration
   (
    input wire clk, reset,          // clock & reset
    input wire syn_clr, load, en,   // input control
    input wire [N-1:0] d,           // input data
    output wire max_tick,           // output status
    output wire [N-1:0] q           // output data
   );

   // constant declaration
   localparam MAX = 2**N - 1;
   // signal declaration
   reg [N-1:0] r_reg, r_next;

   // body
   //=============================================
   // component instantiation
   //=============================================
   // no instantiation in this code

   //=============================================
   // memory elements
   //=============================================
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;

   //=============================================
   // combinational circuits
   //=============================================
   // next-state logic
   always @*
      if (syn_clr)
         r_next = 0;
      else if (load)
         r_next = d;
      else if (en)
         r_next = r_reg + 1;
      else
         r_next = r_reg;
   // output logic
```

```
   assign q = r_reg;
   assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;

endmodule
```

A.2.2 Component instantiation

Listing A.2 Component instantiation template

```
module counter_inst
   (
    input wire clk, reset,
    input wire syn_clr16, load16, en16,
    input wire [15:0] d,
    output wire max_tick8, max_tick16,
    output wire [15:0] q
   );

   // body
   // instantiation of 16-bit counter, all ports used
   bin_counter #(.N(16)) counter_16_unit
      (.clk(clk), .reset(reset),
       .syn_clr(syn_clr16), .load(load16), .en(en16),
       .d(d), .max_tick(max_tick16), .q(q));
   // instantiation of free-running 8-bit counter
   // with only the max_tick signal
   bin_counter counter_8_unit
      (.clk(clk), .reset(reset),
       .syn_clr(1'b0), .load(1'b0), .en(1'b1),
       .d(8'h00), .max_tick(max_tick8), .q());

endmodule
```

A.3 ROUTING WITH CONDITIONAL OPERATOR AND IF AND CASE STATEMENTS

A.3.1 Conditional operator and if statement

Listing A.3 Priority encoder using conditional operator and if statement

```
   (
    input wire [4:1] r,
    output wire [2:0] y1,
    output reg [2:0] y2
   );

   // Conditional operator
   assign y1 = (r[4]) ? 3'b100 :  // can also use (r[4]==1'b1)
               (r[3]) ? 3'b011 :
               (r[2]) ? 3'b010 :
```

```
                (r[1]) ? 3'b001 :
                3'b000;

    // If statement
    //    - each branch can contain multiple statements
    //      with begin ... end delimiters
    always @*
       if (r[4])
          y2 = 3'b100;
       else if (r[3])
          y2 = 3'b011;
       else if (r[2])
          y2 = 3'b010;
       else if (r[1])
          y2 = 3'b001;
       else
          y2 = 3'b000;

endmodule
```

A.3.2 Case statement

Listing A.4 Priority encoder using case statement

```
module prio_encoder_case
   (
    input wire [4:1] r,
    output reg [2:0] y1, y2
   );

   // case statement
   //    - each branch can contain multiple statements
   //      with begin ... end delimiters
   always @*
      case(r)
         4'b1000, 4'b1001, 4'b1010, 4'b1011,
         4'b1100, 4'b1101, 4'b1110, 4'b1111:
            y1 = 3'b100;
         4'b0100, 4'b0101, 4'b0110, 4'b0111:
            y1 = 3'b011;
         4'b0010, 4'b0011:
            y1 = 3'b010;
         4'b0001:
            y1 = 3'b001;
         4'b0000:      // default can also be used
            y1 = 3'b000;
     endcase

   // casez statement
   always @*
      casez(r)
         4'b1???: y2 = 3'b100; // use ? for don't-care
```

```
            4'b01??: y2 = 3'b011;
            4'b001?: y2 = 3'b010;
            4'b0001: y2 = 3'b001;
            4'b0000: y2 = 3'b000; // default can also be used
        endcase

endmodule
```

A.4 COMBINATIONAL CIRCUIT USING AN ALWAYS BLOCK

A.4.1 Always block without default output assignment

Listing A.5 Always block template (without default output assignment)

```
module compare_no_defult
   (
    input wire a, b,
    output reg gt, eq
   );

   // - use @* to include all inputs in sensitivity list
   // - else branch cannot be omitted
   // - all outputs must be assigned in all branches
   always @*
      if (a > b)
         begin
            gt = 1'b1;
            eq = 1'b0;
         end
      else if (a == b)
         begin
            gt = 1'b0;
            eq = 1'b1;
         end
      else   // else branch cannot be omitted
         begin
            gt = 1'b0;
            eq = 1'b0;
         end

endmodule
```

A.4.2 Always block with default output assignment

Listing A.6 Always block template (with default output assignment)

```
module compare_with_default
   (
    input wire a, b,
    output reg gt, eq
```

```
    );

    // - use @* to include all inputs in sensitivity list
    // - assign each output with a default value
    always @*
    begin
       gt = 1'b0;  // default value for gt
       eq = 1'b0;  // default value for eq
       if (a > b)
          gt = 1'b1;
       else if (a == b)
          eq = 1'b1;
    end

endmodule
```

A.5 MEMORY COMPONENTS

A.5.1 Register template

Listing A.7 Register template

```
module reg_template
   (
    input wire clk, reset,
    input wire en,
    input wire [7:0] q1_next, q2_next, q3_next,
    output reg [7:0] q1_reg, q2_reg, q3_reg
   );

   //=========================================
   // register without reset
   //=========================================
   // use nonblock assignment ( <= )
   always @(posedge clk)
      q1_reg <= q1_next;

   //=========================================
   // register with asynchronous reset
   //=========================================
   // use nonblock assignment ( <= )
   always @(posedge clk, posedge reset)
      if (reset)
         q2_reg <= 8'b0;
      else
         q2_reg <= q2_next;

   //=========================================
   // register with enable and asynchronous reset
   //=========================================
   // use nonblock assignment ( <= )
```

```
30    always @(posedge clk, posedge reset)
         if (reset)
            q3_reg <= 8'b0;
         else if (en)
            q3_reg <= q3_next;
35
   endmodule
```

A.5.2 Register file

Listing A.8 Register file

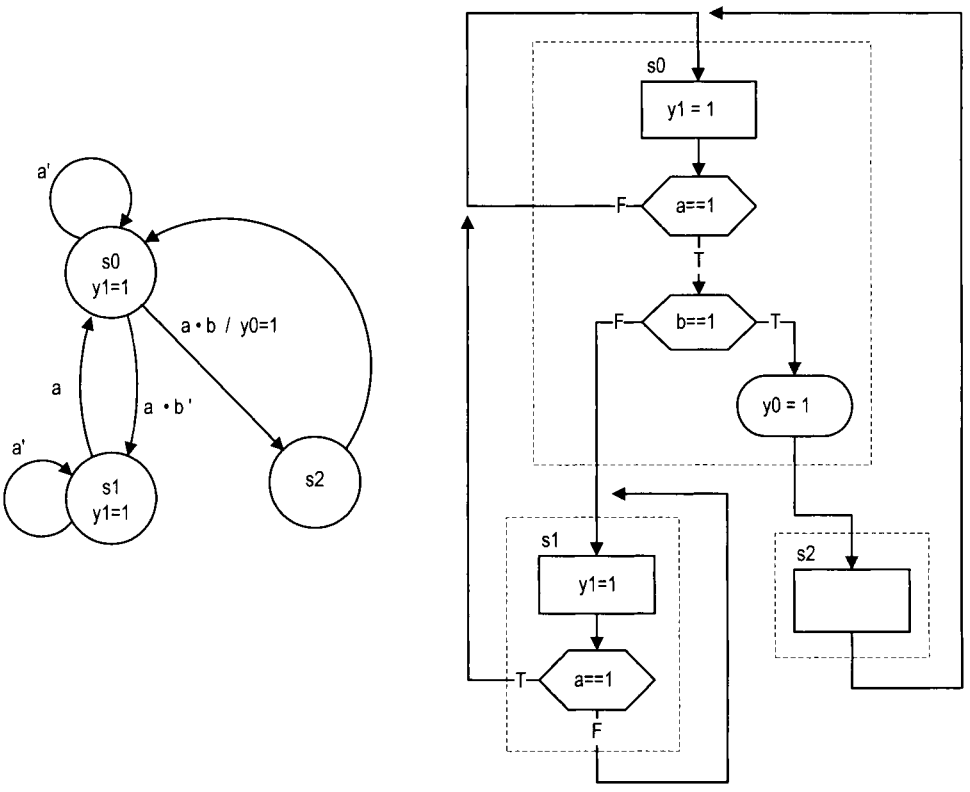
```
module reg_file
   #(
    parameter B = 8, // number of bits
              W = 2  // number of address bits
5  )
   (
    input wire clk,
    input wire wr_en,
    input wire [W-1:0] w_addr, r_addr,
10  input wire [B-1:0] w_data,
    output wire [B-1:0] r_data
   );

   // signal declaration
15 reg [B-1:0] array_reg [2**W-1:0];

   // body
   // write operation
   always @(posedge clk)
20    if (wr_en)
         array_reg[w_addr] <= w_data;
   // read operation
   assign r_data = array_reg[r_addr];

25 endmodule
```

A.6 REGULAR SEQUENTIAL CIRCUITS

Listing A.9 Sequential circuit template

```
//-------------------------------------------------------
//     Universal counter function table
//-------------------------------------------------------
// syn_clr  load  en    q*     operation
5 //-------------------------------------------------------
//     1      -     -     0     synchronous clear
//     0      1     -     d     parallel load
//     0      0     1    q+1    count up
```

```
//      0      0      0      q      pause
//-------------------------------------------------------------
module bin_counter
   #(parameter N=8)    // default 8
   (
    input wire clk, reset,          // clock & reset
    input wire syn_clr, load, en,   // input control
    input wire [N-1:0] d,           // input data
    output wire max_tick,           // output status
    output wire [N-1:0] q           // output data
   );

   // constant declaration
   localparam MAX = 2**N - 1;
   // signal declaration
   reg [N-1:0] r_reg, r_next;

   // body
   //=============================================
   // register
   //=============================================
   // register
   always @(posedge clk, posedge reset)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;
   //=============================================
   // next-state logic
   //=============================================
   always @*
      if (syn_clr)
         r_next = 0;
      else if (load)
         r_next = d;
      else if (en)
         r_next = r_reg + 1;
      else
         r_next = r_reg;
   //=============================================
   // output logic
   //=============================================
   assign q = r_reg;
   assign max_tick = (r_reg==2**N-1) ? 1'b1 : 1'b0;

endmodule
```



(a) State diagram (b) ASM chart

Figure A.1 State diagram and ASM chart of an FSM template.

A.7 FSM

Listing A.10 FSM template

```
// code for the FSM in Figure A.1
module fsm_eg_2_seg
   (
    input wire  clk, reset,
    input wire  a, b,
    output reg y0, y1
   );

   // symbolic state declaration
   localparam [1:0] s0 = 2'b00,
                    s1 = 2'b01,
                    s2 = 2'b10;
   // signal declaration
   reg [1:0] state_reg, state_next;
```

```
   // state register
    always @(posedge clk, posedge reset)
       if (reset)
          state_reg <= s0;
20     else
          state_reg <= state_next;

   // next-state logic and output logic
   always @*
25 begin
      state_next = state_reg; // default next state: the same
      y1 = 1'b0;              // default output: 0
      y0 = 1'b0;              // default output: 0
      case (state_reg)
30       s0: begin
                y1 = 1'b1;
                if (a)
                   if (b)
                      begin
35                       state_next = s2;
                         y0 = 1'b1;
                      end
                   else
                      state_next = s1;
40           end
         s1: begin
               y1 = 1'b1;
               if (a)
                 state_next = s0;
45           end
         s2: state_next = s0;
         default: state_next = s0;
      endcase
   end
50 endmodule
```

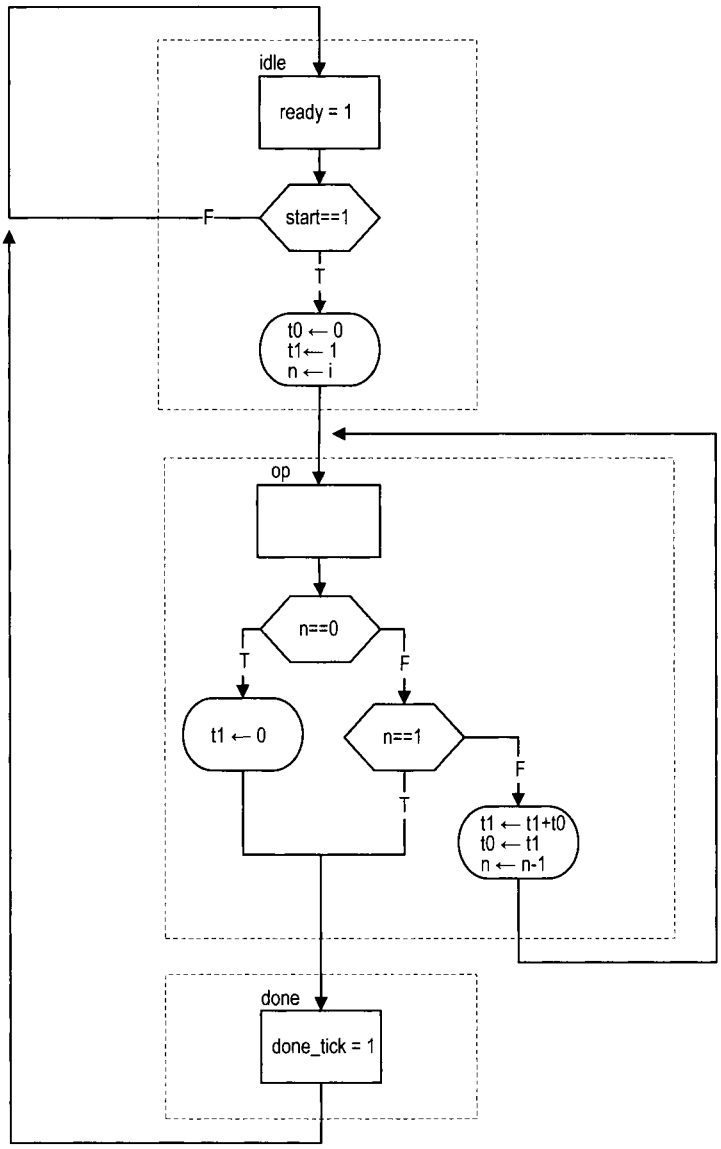


Figure A.2 ASMD chart of an FSMD template.

A.8 FSMD

Listing A.11 FSMD template

```
// code for the FSMD shown in Figure A.2
module fib
   (
    input wire clk, reset,
    input wire start,
    input wire [4:0] i,
```

```
    output reg ready, done_tick,
    output wire [19:0] f
);

// symbolic state declaration
localparam [1:0]
   idle = 2'b00,
   op   = 2'b01,
   done = 2'b10;

// signal declaration
reg [1:0] state_reg, state_next;
reg [19:0] t0_reg, t0_next, t1_reg, t1_next;
reg [4:0] n_reg, n_next;

// body
// state & data registers
always @(posedge clk, posedge reset)
   if (reset)
      begin
         state_reg <= idle;
         t0_reg <= 0;
         t1_reg <= 0;
         n_reg <= 0;
      end
   else
      begin
         state_reg <= state_next;
         t0_reg <= t0_next;
         t1_reg <= t1_next;
         n_reg <= n_next;
      end
// next-state logic and data path functional units
always @*
begin
   state_next = state_reg;  // default return to same state
   ready = 1'b0;            // default output 0
   done_tick = 1'b0;        // default output 0
   t0_next = t0_reg;        // default keep previous value
   t1_next = t1_reg;        // default keep previous value
   n_next = n_reg;          // default keep previous value
   case (state_reg)
      idle:
         begin
            ready = 1'b1;
            if (start)
               begin
                  t0_next = 0;
                  t1_next = 20'd1;
                  n_next = i;
                  state_next = op;
               end
         end
```

```
60          op:
               if (n_reg==0)
                  begin
                     t1_next = 0;
                     state_next = done;
65                end
               else if (n_reg==1)
                  state_next = done;
               else
                  begin
70                   t1_next = t1_reg + t0_reg;
                     t0_next = t1_reg;
                     n_next = n_reg - 1;
                  end
            done:
               begin
75                done_tick = 1'b1;
                  state_next = idle;
               end
            default: state_next = idle;
80       endcase
      end
      // output
      assign f = t1_reg;

85 endmodule
```

A.9 S3 BOARD CONSTRAINT FILE (S3.UCF)

```
#=========================================================
#    Pin assignment for Xilinx
#    Spartan-3 Starter board
#=========================================================

#=========================================================
# clock and reset
#=========================================================
NET "clk"    LOC = "T9" ;
NET "reset"  LOC = "L14";

#=========================================================
# buttons & switches
#=========================================================
# 4 pushbuttons
NET "btn<0>"   LOC = "M13";
NET "btn<1>"   LOC = "M14";
NET "btn<2>"   LOC = "L13";
#NET "btn<3>"  LOC = "L14";  #btn<3> also used as reset

# 8 slide switches
NET "sw<0>"  LOC = "F12";
```

```
NET "sw<1>"  LOC = "G12";
NET "sw<2>"  LOC = "H14";
NET "sw<3>"  LOC = "H13";
NET "sw<4>"  LOC = "J14";
NET "sw<5>"  LOC = "J13";
NET "sw<6>"  LOC = "K14";
NET "sw<7>"  LOC = "K13";

#=========================================================
# RS232
#=========================================================
NET "rx"  LOC = "T13" | DRIVE=8 | SLEW=SLOW;
NET "tx"  LOC = "R13" | DRIVE=8 | SLEW=SLOW;

#=========================================================
# 4-digit time-multiplexed 7-segment LED display
#=========================================================
# digit enable
NET "an<0>"  LOC = "D14";
NET "an<1>"  LOC = "G14";
NET "an<2>"  LOC = "F14";
NET "an<3>"  LOC = "E13";

# 7-segment led segments
NET "sseg<7>"  LOC = "P16"; # decimal point
NET "sseg<6>"  LOC = "E14"; # segment a
NET "sseg<5>"  LOC = "G13"; # segment b
NET "sseg<4>"  LOC = "N15"; # segment c
NET "sseg<3>"  LOC = "P15"; # segment d
NET "sseg<2>"  LOC = "R16"; # segment e
NET "sseg<1>"  LOC = "F13"; # segment f
NET "sseg<0>"  LOC = "N16"; # segment g

#=========================================================
# 8 discrete LEDs
#=========================================================
NET "led<0>"  LOC = "K12";
NET "led<1>"  LOC = "P14";
NET "led<2>"  LOC = "L12";
NET "led<3>"  LOC = "N14";
NET "led<4>"  LOC = "P13";
NET "led<5>"  LOC = "N12";
NET "led<6>"  LOC = "P12";
NET "led<7>"  LOC = "P11";

#=========================================================
# VGA outputs
#=========================================================
NET "rgb<2>"  LOC = "R12" | DRIVE=8 | SLEW=FAST;
NET "rgb<1>"  LOC = "T12" | DRIVE=8 | SLEW=FAST;
NET "rgb<0>"  LOC = "R11" | DRIVE=8 | SLEW=FAST;
NET "vsync"   LOC = "T10" | DRIVE=8 | SLEW=FAST;
NET "hsync"   LOC = "R9"  | DRIVE=8 | SLEW=FAST;
```

```
#=========================================================
# PS2 port
#=========================================================
NET "ps2c" LOC="M16" | IOSTANDARD=LVCMOS33 | DRIVE=8 |SLEW=SLOW;
NET "ps2d" LOC="M15" | IOSTANDARD=LVCMOS33 | DRIVE=8 |SLEW=SLOW;

#=========================================================
# two SRAM chips
#=========================================================
# shared 18-bit memory address
NET "ad<17>"  LOC="L3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<16>"  LOC="K5" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<15>"  LOC="K3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<14>"  LOC="J3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<13>"  LOC="J4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<12>"  LOC="H4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<11>"  LOC="H3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<10>"  LOC="G5" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<9>"   LOC="E4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<8>"   LOC="E3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<7>"   LOC="F4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<6>"   LOC="F3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<5>"   LOC="G4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<4>"   LOC="L4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<3>"   LOC="M3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<2>"   LOC="M4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<1>"   LOC="N3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "ad<0>"   LOC="L5" | IOSTANDARD = LVCMOS33 | SLEW=FAST;

# shared oe, we
NET "oe_n"    LOC="K4" | IOSTANDARD = LVCMOS33 | SLEW=FAST;
NET "we_n"    LOC="G3" | IOSTANDARD = LVCMOS33 | SLEW=FAST;

# sram chip 1 data, ce, ub, lb
NET "dio_a<15>" LOC="R1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<14>" LOC="P1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<13>" LOC="L2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<12>" LOC="J2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<11>" LOC="H1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<10>" LOC="F2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<9>"  LOC="P8" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<8>"  LOC="D3" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<7>"  LOC="B1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<6>"  LOC="C1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<5>"  LOC="C2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<4>"  LOC="R5" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<3>"  LOC="T5" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<2>"  LOC="R6" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<1>"  LOC="T8" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_a<0>"  LOC="N7" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "ce_a_n"    LOC="P7" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "ub_a_n"    LOC="T4" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
```

```
NET "lb_a_n"    LOC="P6" | IOSTANDARD=LVCMOS33 | SLEW=FAST;

# sram chip 2 data, ce, ub, lb
NET "dio_b<15>" LOC="N1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<14>" LOC="M1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<13>" LOC="K2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<12>" LOC="C3" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<11>" LOC="F5" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<10>" LOC="G1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<9>"  LOC="E2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<8>"  LOC="D2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<7>"  LOC="D1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<6>"  LOC="E1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<5>"  LOC="G2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<4>"  LOC="J1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<3>"  LOC="K1" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<2>"  LOC="M2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<1>"  LOC="N2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "dio_b<0>"  LOC="P2" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "ce_b_n"    LOC="N5" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "ub_b_n"    LOC="R4" | IOSTANDARD=LVCMOS33 | SLEW=FAST;
NET "lb_b_n"    LOC="P5" | IOSTANDARD=LVCMOS33 | SLEW=FAST;

#=========================================================
# Timing constraint of S3 50-MHz onboard oscillator
# name of the clock signal is clk
#=========================================================
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 40 ns HIGH 50 %;
```

REFERENCES

1. P. J. Ashenden, *The Designer's Guide to VHDL*, 2nd ed., Morgan Kaufmann, 2001.
2. J. Axelson, *Serial Port Complete*, 2nd ed., Lakeview Research, 2007.
3. L. Bening and H. D. Foster, *Principles of Verifiable RTL Design*, 2nd ed., Springer-Verlag, 2001.
4. J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Springer-Verlag, 2003.
5. K. Chapman, "Creating Embedded Microcontrollers," *TechXclusives* at http://www.xilinx.com.
6. A. Chapweske, "PS/2 Mouse/Keyboard Protocol," http://www.computer-engineering.org.
7. A. Chapweske, "PS/2 Keyboard Interface," http://www.computer-engineering.org.
8. A. Chapweske, "PS/2 Mouse Interface," http://www.computer-engineering.org.
9. P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*, Wiley-IEEE Press, 2006.
10. M. D. Ciletti, *Advanced Digital Design with the Verilog HDL*, Prentice Hall, 2003.
11. M. D. Ciletti, *Starter's Guide to Verilog 2001*, Prentice Hall, 2003.
12. C. E. Cummings, ""full_case parallel_case", the Evil Twins of Verilog Synthesis," SNUG (*Synopsys Users Group Conference*), Boston, 1999.
13. C. E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!" SNUG (*Synopsys Users Group Conference*), San Jose, 2000.
14. C. E. Cummings, "Coding and Scripting Techniques for FSM Designs with Synthesis-Optimized, Glitch-Free Outputs," SNUG (*Synopsys Users Group Conference*), Boston, 2000.
15. C. E. Cummings, "New Verilog-2001 Techniques for Creating Parameterized Models (or Down With 'define and Death of a defparam!)," *International HDL Conference*, 2002.
16. D. D. Gajski, *Principles of Digital Design*, Prentice Hall, 1997.

17. J. O. Hamblen et al., *Rapid Prototyping of Digital Systems: Quartus® II Edition*, Springer, 2005.

18. IEEE, *IEEE Standard for Verilog Hardware Description Language (IEEE Std 1364-2001)*, Institute of Electrical and Electronics Engineers, 2001.

19. IEEE, *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-2001)*, Institute of Electrical and Electronics Engineers, 2001.

20. M. Keating and P. Bricaud, *Methodology Manual for System-on-a-Chip Designs*, 3rd ed., Springer-Verlag, 2002.

21. C. M. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

22. Mentor Graphics, *ModelSim Tutorial*, Mentor Graphics Corporation.

23. S. Palnitkar, *Verilog HDL*, 2nd ed., Prentice Hall, 2003.

24. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed., Morgan Kaufmann, 2004.

25. J. M. Rabaey, *Digital Integrated Circuits*, 2nd ed., Prentice Hall, 2002.

26. S. Sutherland, "What's New in the IEEE 1364 Verilog-2001 Standard," *International HDL Conference*, 2000.

27. J. F. Wakerly, *Digital Design: Principles and Practices*, Prentice Hall, 2002.

28. W. Wolf, *FPGA-Based System Design*, Prentice Hall, 2004.

29. Xilinx, *DS099 Spartan-3 FPGA Family: Complete Data Sheet*, Xilinx, Inc.

30. Xilinx, *ISE 8.1i Quick Start Tutorial*, Xilinx, Inc.

31. Xilinx, *ISE In-Depth Tutorial*, Xilinx, Inc.

32. Xilinx, *PicoBlaze 8-Bit Embedded Microcontroller User Guide*, Xilinx, Inc.

33. Xilinx, *Spartan-3 Starter Kit Board User Guide*, Xilinx, Inc.

34. Xilinx, *XAPP462 Using Digital Clock Managers (DCMs) in Spartan-3 FPGAs*, Xilinx, Inc.

35. Xilinx, *XAPP463 Using Block RAM in Spartan-3 Generation FPGAs*, Xilinx, Inc.

36. Xilinx, *XAPP464 Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs*, Xilinx, Inc.

37. Xilinx, *XST User Guide v8.1i*, Xilinx, Inc.

INDEX