
FPGA PROTOTYPING BY VHDL EXAMPLES

Xilinx Spartan™-3 Version

Pong P. Chu

Cleveland State University

 **WILEY-
INTERSCIENCE**

A JOHN WILEY & SONS, INC., PUBLICATION

This Page Intentionally Left Blank

FPGA PROTOTYPING BY VHDL EXAMPLES

This Page Intentionally Left Blank

FPGA PROTOTYPING BY VHDL EXAMPLES

Xilinx Spartan™-3 Version

Pong P. Chu

Cleveland State University

 **WILEY-
INTERSCIENCE**

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2008 by John Wiley & Sons, Inc. All rights reserved.

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

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic format. For information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Chu, Pong P., 1959–

FPGA prototyping by VHDL examples / Pong P. Chu.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-470-18531-5 (cloth : alk. paper)

1. Field programmable gate arrays—Design and construction. 2. Prototypes, Engineering. 3. VHDL (Computer hardware description language) I. Title.

TK7895.G36C485 2008

621.39'5—dc22

2007029063

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

To my parents, Chia-Chi and Chi-Te, my wife, Lee, and my daughter, Patricia

This Page Intentionally Left Blank

CONTENTS

Preface	xix
Acknowledgments	xxv

PART I BASIC DIGITAL CIRCUITS

1 Gate-level combinational circuit	1
1.1 Introduction	1
1.2 General description	2
1.2.1 Basic lexical rules	2
1.2.2 Library and package	3
1.2.3 Entity declaration	3
1.2.4 Data type and operators	3
1.2.5 Architecture body	4
1.2.6 Code of a 2-bit comparator	5
1.3 Structural description	6
1.4 Testbench	8
1.5 Bibliographic notes	9
1.6 Suggested experiments	10
1.6.1 Code for gate-level greater-than circuit	10
1.6.2 Code for gate-level binary decoder	10
2 Overview of FPGA and EDA software	11

2.1	Introduction	11
2.2	FPGA	11
2.2.1	Overview of a general FPGA device	11
2.2.2	Overview of the Xilinx Spartan-3 devices	13
2.3	Overview of the Digilent S3 board	13
2.4	Development flow	15
2.5	Overview of the Xilinx ISE project navigator	17
2.6	Short tutorial on ISE project navigator	19
2.6.1	Create the design project and HDL codes	21
2.6.2	Create a testbench and perform the RTL simulation	22
2.6.3	Add a constraint file and synthesize and implement the code	22
2.6.4	Generate and download the configuration file to an FPGA device	24
2.7	Short tutorial on the ModelSim HDL simulator	27
2.8	Bibliographic notes	32
2.9	Suggested experiments	33
2.9.1	Gate-level greater-than circuit	33
2.9.2	Gate-level binary decoder	33
3	RT-level combinational circuit	35
3.1	Introduction	35
3.2	RT-level components	35
3.2.1	Relational operators	37
3.2.2	Arithmetic operators	37
3.2.3	Other synthesis-related VHDL constructs	38
3.2.4	Summary	40
3.3	Routing circuit with concurrent assignment statements	41
3.3.1	Conditional signal assignment statement	41
3.3.2	Selected signal assignment statement	44
3.4	Modeling with a process	46
3.4.1	Process	46
3.4.2	Sequential signal assignment statement	46
3.5	Routing circuit with if and case statements	47
3.5.1	If statement	47
3.5.2	Case statement	49
3.5.3	Comparison to concurrent statements	50
3.5.4	Unintended memory	52
3.6	Constants and generics	53
3.6.1	Constants	53
3.6.2	Generics	54
3.7	Design examples	56
3.7.1	Hexadecimal digit to seven-segment LED decoder	56
3.7.2	Sign-magnitude adder	59

3.7.3	Barrel shifter	62
3.7.4	Simplified floating-point adder	63
3.8	Bibliographic notes	69
3.9	Suggested experiments	69
3.9.1	Multi-function barrel shifter	69
3.9.2	Dual-priority encoder	69
3.9.3	BCD incrementor	69
3.9.4	Floating-point greater-than circuit	70
3.9.5	Floating-point and signed integer conversion circuit	70
3.9.6	Enhanced floating-point adder	70
4	Regular Sequential Circuit	71
4.1	Introduction	71
4.1.1	D FF and register	71
4.1.2	Synchronous system	72
4.1.3	Code development	73
4.2	HDL code of the FF and register	74
4.2.1	D FF	74
4.2.2	Register	77
4.2.3	Register file	78
4.2.4	Storage components in a Spartan-3 device <i>Xilinx specific</i>	79
4.3	Simple design examples	79
4.3.1	Shift register	79
4.3.2	Binary counter and variant	81
4.4	Testbench for sequential circuits	84
4.5	Case study	88
4.5.1	LED time-multiplexing circuit	88
4.5.2	Stopwatch	96
4.5.3	FIFO buffer	100
4.6	Bibliographic notes	104
4.7	Suggested experiments	105
4.7.1	Programmable square wave generator	105
4.7.2	PWM and LED dimmer	105
4.7.3	Rotating square circuit	105
4.7.4	Heartbeat circuit	106
4.7.5	Rotating LED banner circuit	106
4.7.6	Enhanced stopwatch	106
4.7.7	Stack	106
5	FSM	107
5.1	Introduction	107

5.1.1	Mealy and Moore outputs	107
5.1.2	FSM representation	108
5.2	FSM code development	111
5.3	Design examples	114
5.3.1	Rising-edge detector	114
5.3.2	Debouncing circuit	118
5.3.3	Testing circuit	122
5.4	Bibliographic notes	124
5.5	Suggested experiments	124
5.5.1	Dual-edge detector	124
5.5.2	Alternative debouncing circuit	124
5.5.3	Parking lot occupancy counter	125
6	FSMD	127
6.1	Introduction	127
6.1.1	Single RT operation	127
6.1.2	ASMD chart	128
6.1.3	Decision box with a register	129
6.2	Code development of an FSMD	131
6.2.1	Debouncing circuit based on RT methodology	132
6.2.2	Code with explicit data path components	134
6.2.3	Code with implicit data path components	136
6.2.4	Comparison	137
6.2.5	Testing circuit	138
6.3	Design examples	140
6.3.1	Fibonacci number circuit	140
6.3.2	Division circuit	143
6.3.3	Binary-to-BCD conversion circuit	147
6.3.4	Period counter	150
6.3.5	Accurate low-frequency counter	153
6.4	Bibliographic notes	156
6.5	Suggested experiments	157
6.5.1	Alternative debouncing circuit	157
6.5.2	BCD-to-binary conversion circuit	157
6.5.3	Fibonacci circuit with BCD I/O: design approach 1	157
6.5.4	Fibonacci circuit with BCD I/O: design approach 2	157
6.5.5	Auto-scaled low-frequency counter	158
6.5.6	Reaction timer	158
6.5.7	Babbage difference engine emulation circuit	159

PART II I/O MODULES

7	UART	163
7.1	Introduction	163
7.2	UART receiving subsystem	164
7.2.1	Oversampling procedure	164
7.2.2	Baud rate generator	165
7.2.3	UART receiver	165
7.2.4	Interface circuit	168
7.3	UART transmitting subsystem	171
7.4	Overall UART system	174
7.4.1	Complete UART core	174
7.4.2	UART verification configuration	176
7.5	Customizing a UART	178
7.6	Bibliographic notes	180
7.7	Suggested experiments	180
7.7.1	Full-featured UART	180
7.7.2	UART with an automatic baud rate detection circuit	181
7.7.3	UART with an automatic baud rate and parity detection circuit	181
7.7.4	UART-controlled stopwatch	181
7.7.5	UART-controlled rotating LED banner	182
8	PS2 Keyboard	183
8.1	Introduction	183
8.2	PS2 receiving subsystem	184
8.2.1	Physical interface of a PS2 port	184
8.2.2	Device-to-host communication protocol	184
8.2.3	Design and code	184
8.3	PS2 keyboard scan code	188
8.3.1	Overview of the scan code	188
8.3.2	Scan code monitor circuit	189
8.4	PS2 keyboard interface circuit	191
8.4.1	Basic design and HDL code	192
8.4.2	Verification circuit	194
8.5	Bibliographic notes	196
8.6	Suggested experiments	196
8.6.1	Alternative keyboard interface I	196
8.6.2	Alternative keyboard interface II	196
8.6.3	PS2 receiving subsystem with watchdog timer	197
8.6.4	Keyboard-controlled stopwatch	197
8.6.5	Keyboard-controlled rotating LED banner	197
9	PS2 Mouse	199

9.1	Introduction	199
9.2	PS2 mouse protocol	200
9.2.1	Basic operation	200
9.2.2	Basic initialization procedure	200
9.3	PS2 transmitting subsystem	201
9.3.1	Host-to-PS2-device communication protocol	201
9.3.2	Design and code	202
9.4	Bidirectional PS2 interface	206
9.4.1	Basic design and code	206
9.4.2	Verification circuit	208
9.5	PS2 mouse interface	210
9.5.1	Basic design	210
9.5.2	Testing circuit	212
9.6	Bibliographic notes	214
9.7	Suggested experiments	214
9.7.1	Keyboard control circuit	214
9.7.2	Enhanced mouse interface	214
9.7.3	Mouse-controlled seven-segment LED display	214
10	External SRAM	215
10.1	Introduction	215
10.2	Specification of the IS61LV25616AL SRAM	216
10.2.1	Block diagram and I/O signals	216
10.2.2	Timing parameters	216
10.3	Basic memory controller	220
10.3.1	Block diagram	220
10.3.2	Timing requirement	221
10.3.3	Register file versus SRAM	222
10.4	A safe design	222
10.4.1	ASMD chart	222
10.4.2	Timing analysis	223
10.4.3	HDL implementation	224
10.4.4	Basic testing circuit	226
10.4.5	Comprehensive SRAM testing circuit	228
10.5	More aggressive design	233
10.5.1	Timing issues	233
10.5.2	Alternative design I	234
10.5.3	Alternative design II	236
10.5.4	Alternative design III	237
10.5.5	Advanced FPGA features <i>Xilinx specific</i>	237
10.6	Bibliographic notes	240
10.7	Suggested experiments	240

10.7.1	Memory with a 512K-by-16 configuration	240
10.7.2	Memory with a 1M-by-8 configuration	240
10.7.3	Memory with an 8M-by-1 configuration	240
10.7.4	Expanded memory testing circuit	241
10.7.5	Memory controller and testing circuit for alternative design I	241
10.7.6	Memory controller and testing circuit for alternative design II	241
10.7.7	Memory controller and testing circuit for alternative design III	241
10.7.8	Memory controller with DCM	241
10.7.9	High-performance memory controller	241
11	Xilinx Spartan-3 Specific Memory	243
11.1	Introduction	243
11.2	Embedded memory of Spartan-3 device	243
11.2.1	Overview	243
11.2.2	Comparison	244
11.3	Method to incorporate memory modules	244
11.3.1	Memory module via HDL component instantiation	245
11.3.2	Memory module via Core Generator	245
11.3.3	Memory module via HDL inference	246
11.4	HDL templates for memory inference	246
11.4.1	Single-port RAM	246
11.4.2	Dual-port RAM	249
11.4.3	ROM	251
11.5	Bibliographic notes	254
11.6	Suggested experiments	254
11.6.1	Block-RAM-based FIFO	254
11.6.2	Block-RAM-based stack	254
11.6.3	ROM-based sign-magnitude adder	255
11.6.4	ROM based $\sin(x)$ function	255
11.6.5	ROM-based $\sin(x)$ and $\cos(x)$ functions	255
12	VGA controller I: graphic	257
12.1	Introduction	257
12.1.1	Basic operation of a CRT	257
12.1.2	VGA port of the S3 board	259
12.1.3	Video controller	259
12.2	VGA synchronization	260
12.2.1	Horizontal synchronization	260
12.2.2	Vertical synchronization	262
12.2.3	Timing calculation of VGA synchronization signals	263
12.2.4	HDL implementation	263

12.2.5	Testing circuit	266
12.3	Overview of the pixel generation circuit	267
12.4	Graphic generation with an object-mapped scheme	268
12.4.1	Rectangular objects	269
12.4.2	Non-rectangular object	273
12.4.3	Animated object	275
12.5	Graphic generation with a bit-mapped scheme	282
12.5.1	Dual-port RAM implementation	282
12.5.2	Single-port RAM implementation	287
12.6	Bibliographic notes	287
12.7	Suggested experiments	287
12.7.1	VGA test pattern generator	287
12.7.2	SVGA mode synchronization circuit	288
12.7.3	Visible screen adjustment circuit	288
12.7.4	Ball-in-a-box circuit	288
12.7.5	Two-balls-in-a-box circuit	289
12.7.6	Two-player pong game	289
12.7.7	Breakout game	289
12.7.8	Full-screen dot trace	289
12.7.9	Mouse pointer circuit	290
12.7.10	Small-screen mouse scribble circuit	290
12.7.11	Full-screen mouse scribble circuit	290

13 VGA controller II: text **291**

13.1	Introduction	291
13.2	Text generation	291
13.2.1	Character as a tile	291
13.2.2	Font ROM	292
13.2.3	Basic text generation circuit	294
13.2.4	Font display circuit	295
13.2.5	Font scaling	297
13.3	Full-screen text display	298
13.4	The complete pong game	302
13.4.1	Text subsystem	302
13.4.2	Modified graphic subsystem	309
13.4.3	Auxiliary counters	310
13.4.4	Top-level system	312
13.5	Bibliographic notes	317
13.6	Suggested experiments	317
13.6.1	Rotating banner	317
13.6.2	Underline for the cursor	317
13.6.3	Dual-mode text display	317

13.6.4	Keyboard text entry	317
13.6.5	UART terminal	317
13.6.6	Square wave display	318
13.6.7	Simple four-trace logic analyzer	318
13.6.8	Complete two-player pong game	319
13.6.9	Complete breakout game	319

PART III PICOBLAZE MICROCONTROLLER^{*XILINX SPECIFIC*}

14	PicoBlaze Overview	323
14.1	Introduction	323
14.2	Customized hardware and customized software	324
14.2.1	From special-purpose FSM to general-purpose microcontroller	324
14.2.2	Application of microcontroller	326
14.3	Overview of PicoBlaze	326
14.3.1	Basic organization	326
14.3.2	Top-level HDL modules	328
14.4	Development flow	329
14.5	Instruction set	329
14.5.1	Programming model	331
14.5.2	Instruction format	332
14.5.3	Logical instructions	332
14.5.4	Arithmetic instructions	333
14.5.5	Compare and test instructions	334
14.5.6	Shift and rotate instructions	335
14.5.7	Data movement instructions	336
14.5.8	Program flow control instructions	338
14.5.9	Interrupt related instructions	341
14.6	Assembler directives	342
14.6.1	The KCPSM3 directives	342
14.6.2	The PBlazeIDE directives	342
14.7	Bibliographic notes	343
15	PicoBlaze Assembly Code Development	345
15.1	Introduction	345
15.2	Useful code segments	345
15.2.1	KCPSM3 conventions	345
15.2.2	Bit manipulation	346
15.2.3	Multiple-byte manipulation	347
15.2.4	Control structure	348
15.3	Subroutine development	350
15.4	Program development	351

15.4.1	Demonstration example	352
15.4.2	Program documentation	356
15.5	Processing of the assembly code	358
15.5.1	Compiling with KCSPM3	358
15.5.2	Simulation by PBlazeIDE	359
15.5.3	Reloading code via the JTAG port	362
15.5.4	Compiling by PBlazeIDE	362
15.6	Syntheses with PicoBlaze	363
15.7	Bibliographic notes	364
15.8	Suggested experiments	365
15.8.1	Signed multiplication	365
15.8.2	Multi-byte multiplication	365
15.8.3	Barrel shift function	365
15.8.4	Reverse function	365
15.8.5	Binary-to-BCD conversion	365
15.8.6	BCD-to-binary conversion	365
15.8.7	Heartbeat circuit	365
15.8.8	Rotating LED circuit	366
15.8.9	Discrete LED dimmer	366
16	PicoBlaze I/O Interface	367
16.1	Introduction	367
16.2	Output port	368
16.2.1	Output instruction and timing	368
16.2.2	Output interface	369
16.3	Input port	371
16.3.1	Input instruction and timing	371
16.3.2	Input interface	371
16.4	Square program with a switch and seven-segment LED display interface	373
16.4.1	Output interface	374
16.4.2	Input interface	375
16.4.3	Assembly code development	376
16.4.4	VHDL code development	384
16.5	Square program with a combinational multiplier and UART console	386
16.5.1	Multiplier interface	387
16.5.2	UART interface	387
16.5.3	Assembly code development	389
16.5.4	VHDL code development	398
16.6	Bibliographic notes	402
16.7	Suggested experiments	402
16.7.1	Low-frequency counter I	402
16.7.2	Low-frequency counter II	402

16.7.3	Auto-scaled low-frequency counter	402
16.7.4	Basic reaction timer with a software timer	403
16.7.5	Basic reaction timer with a hardware timer	403
16.7.6	Enhanced reaction timer	403
16.7.7	Small-screen mouse scribble circuit	403
16.7.8	Full-screen mouse scribble circuit	403
16.7.9	Enhanced rotating banner	403
16.7.10	Pong game	404
16.7.11	Text editor	404
17	PicoBlaze Interrupt Interface	405
17.1	Introduction	405
17.2	Interrupt handling in PicoBlaze	405
17.2.1	Software processing	406
17.2.2	Timing	407
17.3	External interface	408
17.3.1	Single interrupt request	408
17.3.2	Multiple interrupt requests	408
17.4	Software development considerations	409
17.4.1	Interrupt as an alternative scheduling scheme	409
17.4.2	Development of an interrupt service routine	410
17.5	Design example	410
17.5.1	Interrupt interface	410
17.5.2	Interrupt service routine development	411
17.5.3	Assembly code development	411
17.5.4	VHDL code development	413
17.6	Bibliographic notes	417
17.7	Suggested experiments	417
17.7.1	Alternative timer interrupt service routine	417
17.7.2	Programmable timer	417
17.7.3	Set-button interrupt service routine	417
17.7.4	Interrupt interface with two requests	417
17.7.5	Four-request interrupt controller	418
Appendix A:	Sample VHDL templates	419
A.1	General VHDL constructs	419
A.1.1	Overall code structure	419
A.1.2	Component instantiation	420
A.2	Combinational circuits	421
A.2.1	Arithmetic operations	421
A.2.2	Fixed-amount shift operations	422

A.2.3	Routing with concurrent statements	422
A.2.4	Routing with if and case statements	423
A.2.5	Combinational circuit using process	424
A.3	Memory Components	425
A.3.1	Register template	425
A.3.2	Register file	426
A.4	Regular sequential circuits	427
A.5	FSM	428
A.6	FSMD	430
A.7	S3 board constraint file (s3.ucf)	433
References		437
Topic Index		439

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 is limited to 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 VHDL, and a separate book with a similar title uses Verilog. 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 compatibility*.

Audience and prerequisites 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 need. 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 in other boards as well. The applicability of the HDL codes is summarized below.

- **Spartan-3 Starter 3 (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 in the S3 board. There are two differences. First, the “color depth” of its VGA interface is expanded from 3 bits to 8 bits. The the output of the VGA interface circuits discussed in Chapters 12 and 13 needs to be modified accordingly. Second, it contains a more sophisticated external memory device. Although the device can be configured as an asynchronous SRAM, the timing characteristics is different from that of the S3 board’s memory device, and thus the HDL codes for the memory controller in Chapter 10 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 7, 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 10 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.

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 7 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 8 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 9 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 10 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 11 discusses the inference and application of Spartan-3 device-specific components. The focus is on the FPGA’s internal memory blocks and the digital clock management (DCM) circuit.
- Chapter 12 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 interface. The monitor is connected to the prototyping board’s VGA port.
- Chapter 13 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 14 provides an overview of the organization and instruction set of PicoBlaze.
- Chapter 15 introduces the basic assembly programming and provides an overview of the development process.
- Chapter 16 discusses PicoBlaze’s I/O feature and illustrates the procedure to derive customized circuits to interface other I/O peripherals.
- Chapter 17 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* While the examples of this 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 device-independent and software-neutral as much 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, such as the one shown on the outer edge, to indicate that the discussion in the 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**

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 in Part III can be used as 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

October 2007

This Page Intentionally Left Blank

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, and 6.5 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 8.3 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

This Page Intentionally Left Blank

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

1.1 INTRODUCTION

VHDL stands for “VHSIC (very high-speed integrated circuit) hardware description language.” It was originally sponsored by the U.S. Department of Defense and later transferred to the IEEE (Institute of Electrical and Electronics Engineers). The language is formally defined by IEEE Standard 1076. The standard was ratified in 1987 (referred to as VHDL 87), and revised several times. This book mainly follows the revision in 1993 (referred to as VHDL 93).

VHDL 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 VHDL, we introduce the key VHDL synthesis constructs by examining a collection of examples. Detailed VHDL coverage may be explored through the sources listed in the Bibliography.

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

Table 1.1 Truth table of a 1-bit equality comparator

input		output
<i>i0</i>	<i>i1</i>	<i>eq</i>
0	0	1
0	1	0
1	0	0
1	1	1

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 corresponding VHDL 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

```

library ieee;
use ieee.std_logic_1164.all;
entity eq1 is
    port(
5      i0, i1: in std_logic;
      eq: out std_logic
    );
end eq1;

10 architecture sop_arch of eq1 is
    signal p0, p1: std_logic;
    begin
        -- sum of two product terms
        eq <= p0 or p1;
15    -- product terms
        p0 <= (not i0) and (not i1);
        p1 <= i0 and i1;
    end sop_arch;

```

1.2.1 Basic lexical rules

VHDL is case insensitive, which means that upper- and lowercase letters can be used interchangeably, and free formatting, which means that spaces and blank lines can be inserted freely. It is good practice to add proper spaces to make the code clear and to associate special meaning with cases. In this book, we reserve uppercase letters for constants.

An *identifier* is the name of an object and is composed of 26 letters, digits, and the underscore (`_`), as in `i0`, `i1`, and `data_bus1_enable`. The identifier must start with a letter.

The comments start with `--` and the text after it is ignored. In this book, the VHDL keywords are shown in boldface type, as in **entity**, and the comments are shown in italics type, as in

```
-- this is a comment
```

1.2.2 Library and package

The first two lines,

```
library ieee;
use ieee.std_logic_1164.all;
```

invoke the `std_logic_1164` package from the `ieee` library. The package and library allow us to add additional types, operators, functions, etc. to VHDL. The two statements are needed because a special data type is used in the code.

1.2.3 Entity declaration

The entity declaration

```
entity eq1 is
  port (
    i0, i1: in std_logic;
    eq: out std_logic
  );
end eq1;
```

essentially outlines the I/O signals of the circuit. The first line indicates that the name of the circuit is `eq1`, and the port section specifies the I/O signals. The basic format for an I/O port declaration is

```
signal_name1, signal_name2, ... : mode data_type;
```

The mode term can be **in** or **out**, which indicates that the corresponding signals flow “into” or “out of” of the circuit. It can also be **inout**, for bidirectional signals.

1.2.4 Data type and operators

VHDL is a *strongly typed language*, which means that an object must have a data type and only the defined values and operations can be applied to the object. Although VHDL is rich in data types, our discussion is limited to a small set of predefined types that are suitable for synthesis, mainly the `std_logic` type and its variants.

std_logic type The `std_logic` type is defined in the `std_logic_1164` package and consists of nine values. Three of the values, `'0'`, `'1'`, and `'Z'`, which stand for logical 0, logical 1, and high impedance, can be synthesized. Two values, `'U'` and `'X'`, which stand for “uninitialized” and “unknown” (e.g., when signals with `'0'` and `'1'` values are tied together), may be encountered in simulation. The other four values, `'-'`, `'H'`, `'L'`, and `'W'`, are not used in this book.

A signal in a digital circuit frequently contains multiple bits. The `std_logic_vector` data type, which is defined as an array with elements of `std_logic`, can be used for this purpose. For example, let `a` be an 8-bit input port. It can be declared as

```
a: in std_logic_vector(7 downto 0);
```

We can use term like `a(7 downto 4)` to specify a desired range and term like `a(1)` to access a single element of the array. The array can also be declared in ascending order:

```
a: in std_logic_vector(0 to 7);
```

We generally avoid this format since it is more natural to associate the MSB with the leftmost position.

Logical operators Several logical operators, including **not**, **and**, **or**, and **xor**, are defined over the `std_logic_vector` and `std_logic` data type. Bit-wise operation is used when an operator is applied to an object with the `std_logic_vector` data type. Note that the **and**, **or**, and **xor** operators have the same precedence and we need to use parentheses to specify the desired order of evaluation, as in

```
(a and b) or (c and d)
```

1.2.5 Architecture body

The architecture body,

```
architecture sop_arch of eq1 is
  signal p0, p1: std_logic;
begin
  -- sum of two product terms
  eq <= p0 or p1;
  -- product terms
  p0 <= (not i0) and (not i1);
  p1 <= i0 and i1;
end sop_arch;
```

describes operation of the circuit. VHDL allows multiple bodies associated with an entity, and thus the body is identified by the name `sop_arch` (“sum-of-products architecture”).

The architecture body may include an optional declaration section, which specifies constants, internal signals, and so on. Two internal signals are declared in this program:

```
signal p0, p1: std_logic;
```

The main description, encompassed between **begin** and **end**, contains three *concurrent statements*. Unlike a program in C language, in which the statements are executed sequentially, concurrent statements are like circuit parts that operate in parallel. The signal on the left-hand side of a statement can be considered as the output of that part, and the expression specifies the circuit function and corresponding input signals. For example, consider the statement

```
eq <= p0 or 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 default propagation delay.

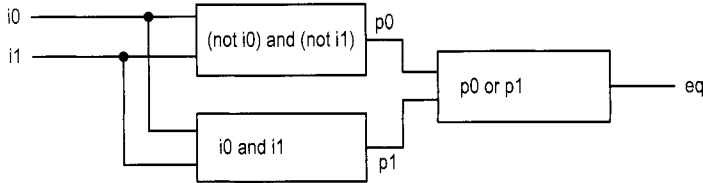


Figure 1.1 Graphical representation of a comparator program.

The graphical representation of this program is shown in Figure 1.1. The three circuit parts represent the three concurrent statements. The connections among these parts are implicitly specified by the signal and port names. The order of the concurrent statements is clearly irrelevant and the statements can be rearranged arbitrarily.

1.2.6 Code of a 2-bit comparator

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 1.2.

Listing 1.2 Gate-level implementation of a 2-bit comparator

```

library ieee;
use ieee.std_logic_1164.all;
entity eq2 is
  port(
5     a, b: in std_logic_vector(1 downto 0);
     aeqb: out std_logic
  );
end eq2;

10 architecture sop_arch of eq2 is
    signal p0,p1,p2,p3: std_logic;
  begin
    -- sum of product terms
    aeqb <= p0 or p1 or p2 or p3;
15    -- product terms
    p0 <= ((not a(1)) and (not b(1))) and
          ((not a(0)) and (not b(0)));
    p1 <= ((not a(1)) and (not b(1))) and (a(0) and b(0));
    p2 <= (a(1) and b(1)) and ((not a(0)) and (not b(0)));
20    p3 <= (a(1) and b(1)) and (a(0) and b(0));
  end sop_arch;

```

The *a* and *b* ports are now declared as a two-element `std_logic_vector`. Derivation of the architecture body is similar to that of a 1-bit comparator. The *p0*, *p1*, *p2*, and *p3* signals represent the results of the four product terms, and the final result, *aeqb*, is the logic expression in sum-of-products format.

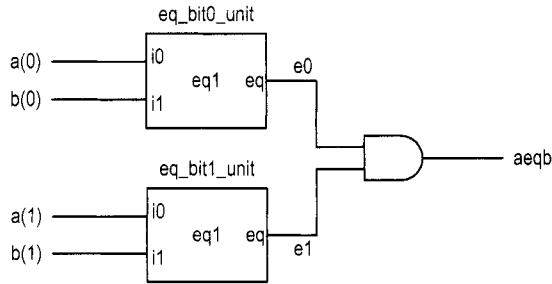


Figure 1.2 Construction of a 2-bit comparator from 1-bit comparators.

1.3 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. VHDL provides a mechanism, known as *component 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 1.2.6 is to utilize the 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 the two bits are equal.

The corresponding code is shown in Listing 1.3. Note that the entity declaration is the same and thus is not included.

Listing 1.3 Structural description of a 2-bit comparator

```

architecture struc_arch of eq2 is
    signal e0, e1: std_logic;
begin
    -- instantiate two 1-bit comparators
    eq_bit0_unit: entity work.eq1(sop_arch)
    port map(i0=>a(0), i1=>b(0), eq=>e0);
    eq_bit1_unit: entity work.eq1(sop_arch)
    port map(i0=>a(1), i1=>b(1), eq=>e1);
    -- a and b are equal if individual bits are equal
    aeqb <= e0 and e1;
end struc_arch;

```

The code includes two component instantiation statements, whose syntax is:

```

unit_label: entity lib_name.entity_name(arch_name)
    port map(
        formal_signal=>actual_signal,
        formal_signal=>actual_signal,
        . . .
    );

```

The first portion of the statement specifies which component is used. The *unit_label* term gives a unique id for an instance, the *lib_name* term indicates where (i.e., which library) the component resides, and the *entity_name* and *arch_name* terms indicate the names of the

entity and architecture. The `arch_name` term is optional. If it is omitted, the last compiled architecture body will be used. The second portion is port mapping, which indicates the connection between *formal signals*, which are I/O ports declared in a component's entity declaration, and *actual signals*, which are the signals used in the architecture body.

The first component instantiation statement is

```
eq_bit0_unit: entity work.eq1(sop_arch)
  port map(i0=>a(0), i1=>b(0), eq=>e0);
```

The `work` library is the default library in which the compiled entity and architecture units are stored, and `eq1` and `sop_arch` are the names of the entity and architecture defined in Listing 1.1. The port mapping reflects the connections shown in Figure 1.2. The component instantiation statement is also a concurrent statement and 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 a diagram, it puts all representations into a single HDL framework. 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.

**Xilinx
specific**

The component instantiation statement is added in VHDL 93. Older codes may use the mechanism in VHDL 87, in which a component must first be declared (i.e., made known) and then used. The code in this format is shown in Listing 1.4.

Listing 1.4 Structural description with VHDL-87

```
architecture vhd_87_arch of eq2 is
  -- component declaration
  component eq1
    port(
      5      i0, i1: in std_logic;
          eq: out std_logic
    );
  end component;
  signal e0, e1: std_logic;
10 begin
  -- instantiate two 1-bit comparators
  eq_bit0_unit: eq1 -- use the declared name, eq1
    port map(i0=>a(0), i1=>b(0), eq=>e0);
  eq_bit1_unit: eq1 -- use the declared name, eq1
15    port map(i0=>a(1), i1=>b(1), eq=>e1);
  -- a and b are equal if individual bits are equal
  aeqb <= e0 and e1;
end vhd_87_arch;
```

Note that the original clause,

```
eq_bit0_unit: entity work.eq1(sop_arch)
```

is replaced by a clause with the declared component name

```
eq_bit0_unit: eq1
```

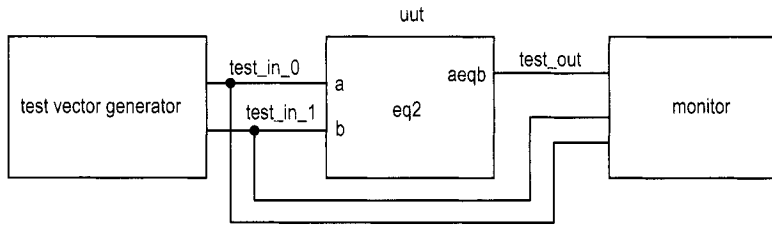


Figure 1.3 Testbench for a 2-bit comparator.

1.4 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 *testbench*, to mimic a physical lab bench. The sketch of a 2-bit comparator testbench program is shown in Figure 1.3. 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 1.5.

Listing 1.5 Testbench for a 2-bit comparator

```

library ieee;
use ieee.std_logic_1164.all;
entity eq2_testbench is
end eq2_testbench;
5
architecture tb_arch of eq2_testbench is
    signal test_in0, test_in1: std_logic_vector(1 downto 0);
    signal test_out: std_logic;
begin
10  -- instantiate the circuit under test
    uut: entity work.eq2(struct_arch)
        port map(a=>test_in0, b=>test_in1, aeqb=>test_out);
    -- test vector generator
    process
15  begin
        -- test vector 1
        test_in0 <= "00";
        test_in1 <= "00";
        wait for 200 ns;
20  -- test vector 2
        test_in0 <= "01";
        test_in1 <= "00";
        wait for 200 ns;
        -- test vector 3
25  test_in0 <= "01";
        test_in1 <= "11";
        wait for 200 ns;
        -- test vector 4

```

```

    test_in0 <= "10";
30  test_in1 <= "10";
    wait for 200 ns;
    -- test vector 5
    test_in0 <= "10";
    test_in1 <= "00";
35  wait for 200 ns;
    -- test vector 6
    test_in0 <= "11";
    test_in1 <= "11";
    wait for 200 ns;
40  -- test vector 7
    test_in0 <= "11";
    test_in1 <= "01";
    wait for 200 ns;
    end process;
45 end tb_arch;

```

The code consists of a component instantiation statement, which creates an instance of a 2-bit comparator, and a process statement, which generates a sequence of test patterns.

The process statement is a special VHDL construct in which the operations are performed sequentially. Each test pattern is generated by three statements. For example,

```

    -- test vector 2
    test_in0 <= "01";
    test_in1 <= "00";
    wait for 200 ns;

```

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 ns.

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 VHDL and is beyond the scope of this book. 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.

1.5 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.

VHDL is a complex language. *The Designer's Guide to VHDL* by P. J. Ashenden provides detailed coverage of the language's syntax and constructs. The author's *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability* provides a comprehensive discussion on developing effective, synthesizable codes. The 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.

1.6 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.

1.6.1 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.

1.6.2 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.