



FPGA PROTOTYPING BY SYSTEMVERILOG EXAMPLES

XILINX MICROBLAZE MCS SoC EDITION



PONG P. CHU

WILEY

FPGA PROTOTYPING BY SYSTEMVERILOG EXAMPLES

FPGA PROTOTYPING BY SYSTEMVERILOG EXAMPLES

Xilinx MicroBlaze MCS SoC Edition

Pong P. Chu
Cleveland State University

WILEY

This edition first published 2018
© 2018 John Wiley & Sons, Inc.

All rights reserved. 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 or otherwise, except as permitted by law. Advice on how to obtain permission to reuse material from this title is available at <http://www.wiley.com/go/permissions>.

The right of Pong P. Chu to be identified as the author of this work has been asserted in accordance with law.

Registered Office

John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, USA

Editorial Office

111 River Street, Hoboken, NJ 07030, USA

For details of our global editorial offices, customer services, and more information about Wiley products visit us at www.wiley.com.

Wiley also publishes its books in a variety of electronic formats and by print-on-demand. Some content that appears in standard print versions of this book may not be available in other formats.

Limit of Liability/Disclaimer of Warranty

While the publisher and authors have used their best efforts in preparing this work, they make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives, written sales materials or promotional statements for this work. The fact that an organization, website, or product is referred to in this work as a citation and/or potential source of further information does not mean that the publisher and authors endorse the information or services the organization, website, or product may provide or recommendations it may make. This work is sold with the understanding that the publisher is not engaged in rendering professional services. The advice and strategies contained herein may not be suitable for your situation. You should consult with a specialist where appropriate. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Library of Congress Cataloging-in-Publication Data

Names: Chu, Pong P., 1959- author.

Title: FPGA prototyping by system VERILOG examples. Xilinx MicroBlaze MCS SoC Edition / by Pong P. Chu, Cleveland State University.

Description: Hoboken, NJ, USA : Wiley, 2018. | Includes bibliographical references and index. |

Identifiers: LCCN 2018005487 (print) | LCCN 2018006519 (ebook) | ISBN 9781119282693 (pdf) | ISBN 9781119282709 (epub) | ISBN 9781119282662 (cloth)

Subjects: LCSH: Field programmable gate arrays--Design and construction. | Prototypes, Engineering. | VHDL (Computer hardware description language)

Classification: LCC TK7895.G36 (ebook) | LCC TK7895.G36 C4835 2018 (print) | DDC 621.39/5--dc23

LC record available at <https://lccn.loc.gov/2018005487>

Cover image: Courtesy of Pong P. Chu
Cover design by Wiley

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

*To my mother, Chi-Te, my wife, Lee, and my
daughter, Patricia*

CONTENTS

Preface	xxvii
Acknowledgments	xxxiii

PART I BASIC DIGITAL CIRCUITS DEVELOPMENT

1 Gate-Level Combinational Circuit	1
1.1 Introduction	1
1.1.1 Brief history of Verilog and SystemVerilog	1
1.1.2 Book coverage	2
1.2 General description	3
1.3 Basic lexical elements and data types	4
1.3.1 Lexical elements	4
1.3.2 Data types used in the book	5
1.3.3 Number representation	6
1.3.4 Operators	7
1.4 Program skeleton	7
1.4.1 Port declaration	7
1.4.2 Signal declaration	8
1.4.3 Program body	8
1.4.4 Concurrent semantics	9
1.4.5 Another example	10
1.5 Structural description	10

1.6	Top-level signal mapping	13
1.7	Testbench	14
1.8	Bibliographic notes	16
1.9	Suggested experiments	16
1.9.1	Code for gate-level greater-than circuit	17
1.9.2	Code for gate-level binary decoder	17
2	Overview of FPGA and EDA Software	19
2.1	FPGA	19
2.1.1	Overview of a general FPGA device	19
2.1.2	Overview of the Xilinx Artix-7 devices	20
2.2	Overview of the Digilent Nexys 4 DDR board	21
2.3	Development flow	22
2.4	Xilinx Vivado Design Suite	24
2.5	Bibliographic notes	24
2.6	Suggested experiments	24
2.6.1	Gate-level greater-than circuit	24
2.6.2	Gate-level binary decoder	26
3	RT-Level Combinational Circuit	29
3.1	Operators	29
3.1.1	Arithmetic operators	31
3.1.2	Shift operators	31
3.1.3	Relational and equality operators	32
3.1.4	Bitwise, reduction, and logical operators	32
3.1.5	Concatenation and replication operators	33
3.1.6	Conditional operators	34
3.1.7	Operator precedence	35
3.1.8	Expression bit-length adjustment	35
3.1.9	Synthesis of z and x values	36
3.2	Always block for a combinational circuit	38
3.2.1	Overview of always block	39
3.2.2	Procedural assignment	40
3.2.3	Conceptual examples	40
3.3	Coding guidelines	43
3.4	If statement	43
3.4.1	Syntax	43
3.4.2	Examples	44
3.5	Case statement	45
3.5.1	Syntax	45
3.5.2	Examples	46

3.5.3	The casez and casex statements	47
3.5.4	Full case and parallel case	48
3.6	Routing structure of conditional control constructs	49
3.6.1	Priority routing network	49
3.6.2	Multiplexing network	51
3.7	Additional coding guidelines for an always block	52
3.7.1	Common errors in combinational circuit codes	52
3.7.2	Guidelines	56
3.8	Parameter and constant	56
3.8.1	Constant	56
3.8.2	Parameter	58
3.9	Replicated structure	59
3.9.1	Generate-for statement	59
3.9.2	Procedural-for statement	60
3.9.3	Example	60
3.10	Design examples	62
3.10.1	Hexadecimal digit to seven-segment LED decoder	62
3.10.2	Sign-magnitude adder	65
3.10.3	Barrel shifter	68
3.10.4	Simplified floating-point adder	69
3.11	Bibliographic notes	73
3.12	Suggested experiments	73
3.12.1	Multi-function barrel shifter	73
3.12.2	Parameterized barrel shifter	74
3.12.3	Dual-priority encoder	74
3.12.4	BCD incrementor	74
3.12.5	Floating-point greater-than circuit	74
3.12.6	Floating-point and signed integer conversion circuit	74
3.12.7	Enhanced floating-point adder	75
4	Regular Sequential Circuit	77
4.1	Introduction	77
4.1.1	D FF and register	78
4.1.2	Basic block system	78
4.1.3	Code development	79
4.1.4	Sequential circuit coding guidelines and style	79
4.2	HDL code of the FF and register	80
4.2.1	D FF	80
4.2.2	Register	85
4.3	Simple design examples	85
4.3.1	Shift register	85
4.3.2	Binary counter and variant	87

4.4	Testbench for sequential circuits	89
4.5	Case study	93
4.5.1	LED time-multiplexing circuit	93
4.5.2	Stopwatch	101
4.6	Timing and clocking	104
4.6.1	Timing of FF	104
4.6.2	Maximum operating frequency	104
4.6.3	Clock tree	107
4.6.4	GALS system and CDC	107
4.7	Bibliographic notes	108
4.8	Suggested experiments	108
4.8.1	Programmable square wave generator	108
4.8.2	PWM and LED dimmer	108
4.8.3	Rotating square circuit	109
4.8.4	Heartbeat circuit	109
4.8.5	Rotating LED banner circuit	109
4.8.6	Enhanced stopwatch	110
5	FSM	111
5.1	Introduction	111
5.1.1	Mealy and Moore outputs	112
5.1.2	FSM representation	112
5.2	FSM code development	115
5.2.1	Enumerated data type and state assignment	115
5.2.2	Multi-segment code	116
5.2.3	Two-segment code	117
5.3	Design examples	118
5.3.1	Rising-edge detector	118
5.3.2	Debouncing circuit	123
5.3.3	Testing circuit	126
5.4	Bibliographic notes	128
5.5	Suggested experiments	128
5.5.1	Dual-edge detector	128
5.5.2	Early detection debouncing circuit	128
5.5.3	Parking lot occupancy counter	129
6	FSMD	131
6.1	Introduction	131
6.1.1	Single RT operation	132
6.1.2	ASMD chart	132
6.1.3	Decision box with a register	134

6.2	Code development of an FSM	137
6.2.1	Debouncing circuit based on RT methodology	137
6.2.2	Code with explicit data path components	137
6.2.3	Code with implicit data path components	140
6.2.4	Comparison	142
6.3	Design examples	144
6.3.1	Fibonacci number circuit	144
6.3.2	Division circuit	147
6.3.3	Binary-to-BCD conversion circuit	150
6.3.4	Period counter	153
6.3.5	Accurate low-frequency counter	156
6.4	Bibliographic notes	159
6.5	Suggested experiments	159
6.5.1	Early detection debouncing circuit	159
6.5.2	BCD-to-binary conversion circuit	160
6.5.3	Fibonacci circuit with BCD I/O: design approach 1	160
6.5.4	Fibonacci circuit with BCD I/O: design approach 2	160
6.5.5	Auto-scaled low-frequency counter	161
6.5.6	Reaction timer	161
6.5.7	Babbage difference engine emulation circuit	162
7	RAM and Buffer of FPGA	165
7.1	Embedded memory of FPGA device	165
7.1.1	Memory of an Artix device	166
7.1.2	Memory available in the Nexys 4 DDR board	166
7.2	General description for a RAM-like component	167
7.2.1	Register file	167
7.2.2	Dynamic array indexing operation	169
7.2.3	Key aspects of a RAM module	170
7.2.4	Genuine ROM	171
7.3	FIFO buffer	173
7.3.1	FIFO read configuration	174
7.3.2	Circular queue implementation	175
7.4	HDL templates for memory inference	178
7.4.1	Methods to incorporate memory modules	178
7.4.2	Synchronous dual-port RAM	179
7.4.3	“Simple” synchronous dual-port RAM	180
7.4.4	Synchronous single-port RAM	181
7.4.5	Synchronous ROM	182
7.4.6	BRAM-based FIFO buffer	183
7.4.7	Design considerations	183
7.5	Overview of memory controller	184

7.6	Bibliographic notes	185
7.7	Suggested experiments	186
7.7.1	ROM-based sign-magnitude adder	186
7.7.2	ROM-based temperature conversion	186
7.7.3	FIFO with data width conversion	186
7.7.4	Standard FIFO to FWFT FIFO conversion circuit	187
7.7.5	FIFO buffer with extended status	187
7.7.6	Stack	187
8	Selected Topics of SystemVerilog	189
8.1	Timing model	189
8.1.1	Concurrent constructs	190
8.1.2	Assignment statement	190
8.1.3	Basic model	190
8.1.4	Blocking versus nonblocking assignment	192
8.2	Coding guidelines revisited	194
8.2.1	“Single variable assignment” guideline	195
8.2.2	“Blocking assignment for combinational circuit” guideline	195
8.2.3	“Nonblocking assignment for register” guideline	197
8.3	Alternative coding style	198
8.3.1	First coding style revisited	198
8.3.2	Sequential circuit with mixed blocking and nonblocking assignments	199
8.3.3	Combined coding style	201
8.3.4	Summary	206
8.4	Data types	206
8.4.1	The net and variable types	206
8.4.2	The logic data type	207
8.4.3	Limitation of the logic data type	208
8.4.4	New data types in SystemVerilog	208
8.5	Use of the signed data type	209
8.5.1	Overview	209
8.5.2	Signed number conversion	210
8.6	Bibliographic notes	211
8.7	Suggested experiments	211
8.7.1	Shift register with blocking and nonblocking assignments	211
8.7.2	Alternative coding style for the BCD counter	212
8.7.3	Alternative coding style for the FIFO buffer	212
8.7.4	Alternative coding style for the Fibonacci circuit	212
8.7.5	Dual-mode comparator	212

PART II EMBEDDED SOC I: VANILLA FPRO SYSTEM

9	Overview of Embedded SoC Systems	215
9.1	Embedded SoC	215
9.1.1	Overview of embedded systems	215
9.1.2	FPGA-based SoC	216
9.1.3	IP cores	216
9.2	Development flow of the embedded SoC	217
9.2.1	Hardware–software partition	217
9.2.2	Hardware development flow	217
9.2.3	Software development flow	219
9.2.4	Physical implementation and test	219
9.2.5	Custom IP core development	219
9.3	FPro SoC Platform	220
9.3.1	Motivations	220
9.3.2	Platform hardware organization	221
9.3.3	Platform software organization	223
9.3.4	Modified development flow	224
9.4	Adaptation on the Digilent Nexys 4 DDR board	224
9.5	Portability	226
9.5.1	Processor Module and Bridge	226
9.5.2	MMIO subsystem	227
9.5.3	Video subsystem	227
9.6	Organization	228
9.7	Bibliographic notes	228
10	Bare Metal System Software Development	231
10.1	Bare metal system development overview	231
10.1.1	Desktop-like system versus bare metal system	231
10.1.2	Basic embedded program architecture	232
10.2	Memory-mapped I/O	233
10.2.1	Overview	233
10.2.2	Memory alignment	234
10.2.3	I/O register map	234
10.2.4	I/O address space of the FPro system	234
10.3	Direct I/O Register Access	235
10.3.1	Review of C pointer	235
10.3.2	C pointer for I/O register	236
10.4	Robust I/O register access	237
10.4.1	<code>chu_io_map.h</code> and <code>chu_io_map.svh</code>	237
10.4.2	<code>inttypes.h</code>	238

10.4.3	<code>chu_io_rw.h</code>	239
10.5	Techniques for low-level I/O operations	241
10.5.1	Bit manipulation	241
10.5.2	Packing and unpacking	242
10.6	Device Drivers	243
10.6.1	Overview	243
10.6.2	GPO and GPI drivers	243
10.6.3	Timer driver	245
10.6.4	UART driver	247
10.7	FPro utility routines and directory structure	248
10.7.1	Minimal hardware requirements	248
10.7.2	Utility routines	248
10.7.3	Directory structure	251
10.8	Test program	252
10.8.1	IP core verification routine	252
10.8.2	Programming with limited memory	252
10.8.3	Test function integration	252
10.8.4	Test program for the vanilla FPro system	253
10.8.5	Implementation	254
10.9	Bibliographic notes	255
10.10	Suggested experiments	255
10.10.1	Chasing LEDs	255
10.10.2	Collision LEDs	256
10.10.3	Pulse width modulation	256
10.10.4	System time display	256
11	FPro Bus Protocol and MMIO Slot Specification	257
11.1	FPro bus	257
11.1.1	Overview of the bus	257
11.1.2	SoC interconnect	258
11.1.3	FPro bus protocol specification	259
11.2	Interface with the bus	260
11.2.1	Introduction	260
11.2.2	Write interface and decoding	261
11.2.3	Read interface and multiplexing	263
11.2.4	FIFO buffer as an I/O register	264
11.2.5	Timing consideration	265
11.3	MMIO I/O core	266
11.3.1	MMIO slot interface specification	266
11.3.2	Basic MMIO I/O core construction	268
11.3.3	GPO and GPI cores	269
11.4	Timer core development	270

11.4.1	Custom logic	270
11.4.2	Register map	271
11.4.3	Wrapping circuit for the slot interface	271
11.5	MMIO controller	272
11.5.1	<code>chu_io_map.svh</code> file	273
11.5.2	HDL code	273
11.5.3	Vanilla MMIO subsystem	275
11.6	MCS I/O bus and bridge	278
11.6.1	Overview of Xilinx MicroBlaze MCS	278
11.6.2	MicroBlaze MCS I/O bus	278
11.6.3	MCS-to-FPro bridge	279
11.7	Vanilla FPro system construction	281
11.8	Bibliographic notes	282
11.9	Suggested experiments	283
11.9.1	FPro bus with a byte-lane enable signal	283
11.9.2	Seven-segment control with a GPO core	283
11.9.3	GPIO core	283
11.9.4	Blinking-LED core	284
11.9.5	Timer core with a programmable period	284
11.9.6	Timer core with a <i>run-once</i> mode	284
12	UART Core	287
12.1	Introduction	287
12.1.1	Overview of serial communication	287
12.1.2	Overview of the UART	288
12.1.3	Oversampling procedure	288
12.2	UART construction	289
12.2.1	Conceptual design	289
12.2.2	Baud rate generator	290
12.2.3	UART receiver	291
12.2.4	UART transmitter	293
12.2.5	Top-level HDL code	295
12.3	UART core development	296
12.3.1	Register map	296
12.3.2	Wrapping circuit for the slot interface	297
12.4	UART driver	298
12.4.1	Class definition	299
12.4.2	Basic methods	300
12.4.3	ASCII code	301
12.4.4	Display methods	303
12.4.5	Test	305
12.5	Additional project ideas	305

12.5.1	Original serial port	305
12.5.2	Emulated serial port	305
12.5.3	Direct connection	306
12.5.4	USB-to-UART adaptor	306
12.5.5	Wireless adaptor	307
12.6	Bibliographic notes	308
12.7	Suggested experiments	308
12.7.1	UART-controlled chasing LEDs	308
12.7.2	Alternative read configuration	308
12.7.3	UART controller with a parity bit	308
12.7.4	UART core with an error status	309
12.7.5	Configurable UART core	309
12.7.6	UART core with automatic baud rate detection	309
12.7.7	UART core with enhanced automatic baud rate detection	310
12.7.8	UART core with an automatic baud rate and a parity detection circuit	310

PART III EMBEDDED SOC II: BASIC I/O CORES

13	Xilinx XADC Core	313
13.1	Overview of XADC	313
13.1.1	Block diagram	313
13.1.2	Configuration	314
13.2	XADC core development	315
13.2.1	XADC instantiation	315
13.2.2	Basic wrapping circuit design	316
13.2.3	Register map	318
13.2.4	HDL code	318
13.3	XADC core device driver	320
13.3.1	Class definition	320
13.3.2	Class implementation	321
13.3.3	Testing for the XADC core	322
13.4	Sampler FPro system	323
13.4.1	Testing procedure of an FPro core	323
13.4.2	System configuration	323
13.4.3	Hardware derivation	324
13.4.4	Software verification program	331
13.5	Additional project ideas	332
13.6	Bibliographic notes	333
13.7	Suggested experiments	333
13.7.1	Real-time voltage display	333

13.7.2	Potentiometer-controlled chasing LEDs	333
13.7.3	Potentiometer-controlled LED dimmer	333
13.7.4	Enhanced wrapping circuit: part I	333
13.7.5	Enhanced wrapping circuit: part II	333
14	Pulse Width Modulation Core	335
14.1	Introduction	335
14.1.1	PWM as analog output	335
14.1.2	Main characteristics	336
14.2	PWM design	336
14.2.1	Basic design	336
14.2.2	Enhanced design	337
14.3	PWM core development	339
14.3.1	Register map	339
14.3.2	Wrapped PWM circuit	340
14.4	PWM driver	341
14.4.1	Class definition	341
14.4.2	Class implementation	342
14.5	Testing	343
14.6	Project ideas	343
14.7	Suggested experiments	345
14.7.1	Police dash light	345
14.7.2	Rainbow night light	345
14.7.3	Enhanced PWM core: part I	345
14.7.4	Enhanced PWM core: part II	346
14.7.5	Enhanced GPIO core	346
14.7.6	Servo motor driver	346
15	Debouncing Core and LED-Mux Core	347
15.1	Debouncing Core	347
15.1.1	Multi-bit debouncing circuit	347
15.1.2	Register map and the slot wrapping circuit	350
15.1.3	Driver	351
15.1.4	Test	352
15.2	LED-mux core	352
15.2.1	Eight-digit seven-segment LED display multiplexing circuit	352
15.2.2	Register map and the slot wrapping circuit	354
15.2.3	Driver	355
15.2.4	Test	358
15.3	Project ideas	358
15.4	Suggested experiments	360

15.4.1	Area comparison of two debouncing circuits	360
15.4.2	Enhanced debouncing core: part I	360
15.4.3	Enhanced debouncing core: part II	360
15.4.4	Rotating square pattern revisited	360
15.4.5	Heartbeat pattern revisited	360
15.4.6	Stopwatch	360
15.4.7	Enhanced LED-mux core	361
16	SPI Core	363
16.1	Overview	363
16.1.1	Conceptual architecture	364
16.1.2	Multiple-device configuration	364
16.1.3	Basic timing	366
16.1.4	Operation modes	367
16.1.5	Undefined aspects	368
16.2	SPI controller	369
16.2.1	Basic design	369
16.2.2	FSMD construction	370
16.2.3	HDL implementation	370
16.3	SPI core development	374
16.3.1	Register map	374
16.3.2	Wrapping circuit for the slot interface	374
16.4	SPI driver	376
16.4.1	Class definition	376
16.4.2	Class implementation	377
16.5	Test	378
16.5.1	ADXL362 accelerometer	378
16.5.2	Test program	380
16.6	Project ideas	381
16.6.1	SD card	381
16.6.2	TFT LCD module	382
16.7	Bibliographic notes	382
16.8	Suggested experiments	382
16.8.1	Inclination sensing	382
16.8.2	“Tapping” detection	382
16.8.3	ADXL362 C++ class	383
16.8.4	Enhanced SPI controller: part I	383
16.8.5	Enhanced SPI controller: part II	383
16.8.6	“Automatic-read” ADXL362 wrapper: part I	383
16.8.7	“Automatic-read” ADXL362 wrapper: part II	384
16.8.8	Flash memory access	384
16.8.9	SPI slave controller: part I	384

16.8.10 SPI slave controller: part II	385
17 I²C Core	387
17.1 Overview	387
17.1.1 Electrical characteristics	388
17.1.2 Basic bus protocol	388
17.1.3 Basic timing	389
17.1.4 Additional features	390
17.2 I ² C controller	391
17.2.1 Basic design	391
17.2.2 Conceptual FSMD construction	391
17.2.3 Output control logic	394
17.2.4 I ² C bus clock generation	394
17.2.5 HDL implementation	395
17.3 I ² C core development	400
17.3.1 Register map	400
17.3.2 Wrapping circuit for the slot interface	400
17.4 I ² C driver	401
17.4.1 Class definition	401
17.4.2 Class implementation	402
17.5 Test	405
17.5.1 ADT7420 temperature sensor	405
17.5.2 Test program	406
17.6 Project idea	406
17.7 Bibliographic notes	407
17.8 Suggested experiments	407
17.8.1 Thermometer	407
17.8.2 ADT7420 C++ class	407
17.8.3 Enhanced I ² C core	408
17.8.4 “Automatic-read” ADT7420 wrapper	408
17.8.5 I ² C slave controller: part I	408
17.8.6 I ² C slave controller: part II	408
18 PS2 Core	409
18.1 Introduction	409
18.1.1 PS2-device-to-host communication protocol and timing	410
18.1.2 Host-to-PS2-device communication protocol and timing	410
18.2 PS2 controller	411
18.2.1 Conceptual design	411
18.2.2 PS2 receiving subsystem	411
18.2.3 PS2 transmitting subsystem	415

18.2.4 Complete PS2 system	419
18.3 PS2 core development	420
18.3.1 Register map	420
18.3.2 Wrapping circuit for the slot interface	421
18.4 PS2 driver	422
18.4.1 Class definition	422
18.4.2 Lower layer methods	422
18.4.3 PS2 initialization routine	423
18.4.4 Keyboard routine	425
18.4.5 Mouse routine	428
18.5 Test	430
18.6 Bibliographic notes	431
18.7 Suggested experiments	431
18.7.1 PS2 receiving subsystem with watchdog timer	431
18.7.2 Keyboard-controlled LED flashing circuit	432
18.7.3 Enhanced keyboard driver routine: part I	432
18.7.4 Enhanced keyboard driver routine: part II	432
18.7.5 Remote-mode mouse driver	432
18.7.6 Scroll-wheel mouse driver	432
 19 Sound I: DDFS Core	 433
19.1 Introduction	433
19.2 Design and implementation	434
19.2.1 Direct synthesis of a digital waveform	434
19.2.2 Direct synthesis of an unmodulated analog waveform	435
19.2.3 Direct synthesis of a modulated analog waveform	436
19.3 Fixed-point arithmetic	437
19.4 DDFS construction	438
19.5 DAC (digital-to-analog converter)	440
19.5.1 Conceptual design	440
19.5.2 HDL implementation	441
19.6 DDFS core development	442
19.6.1 Register map	442
19.6.2 Wrapping circuit for the slot interface	443
19.7 DDFS driver	444
19.7.1 Class definition	444
19.7.2 Class implementation	445
19.8 Test	447
19.9 Bibliographic notes	448
19.10 Suggested experiments	448
19.10.1 Quadrature phase carrier generation	448
19.10.2 Reduced-size phase-to-amplitude lookup table	448

19.10.3 Additive harmonic synthesis	449
19.10.4 Simple function generator	449
19.10.5 Arbitrary waveform generator	449
19.10.6 Sample-based synthesis	449
20 Sound II: ADSR Core	451
20.1 Introduction	451
20.2 ADSR envelope generator	452
20.2.1 Conceptual FSM design	453
20.2.2 ASMD chart	453
20.2.3 HDL implementation	455
20.3 ADSR core development	457
20.3.1 Register map	457
20.3.2 Wrapped ADSR circuit	458
20.4 ADSR driver	460
20.4.1 Class definition	460
20.4.2 Configuration methods	461
20.4.3 <code>calc_note_freq()</code> method	463
20.4.4 <code>play_note()</code> method	465
20.5 Test	465
20.6 Project idea	466
20.7 Bibliographic notes	467
20.8 Suggested experiments	467
20.8.1 RTTTL music player	467
20.8.2 ADSR envelope testing	467
20.8.3 Pushbutton piano	467
20.8.4 Keyboard piano	468
20.8.5 Keyboard recorder	468
20.8.6 Real-time mode ADSR generator	468
20.8.7 Real-time mode pushbutton piano	468
20.8.8 Merged DDFS and ADSR core	468
20.8.9 ADSR core with an automatic play FIFO buffer	468
20.8.10 ADSR core for frequency modulation	468
PART IV EMBEDDED SOC III: VIDEO CORES	
21 Introduction to the Video System	471
21.1 Introduction to a video display	471
21.1.1 Conceptual video display	471
21.1.2 VGA interface	472
21.2 Stream interface	473

21.2.1	Random-access interface versus stream interface	473
21.2.2	Flow control of the stream interface	473
21.3	VGA synchronization	475
21.3.1	Basic operation of a CRT monitor	475
21.3.2	Horizontal synchronization	476
21.3.3	Vertical synchronization	478
21.3.4	Pixel clock rate	479
21.3.5	VGA synchronization circuit	480
21.4	Bar test-pattern generator	483
21.5	Color-to-grayscale conversion circuit	485
21.6	Demo video system	486
21.7	Advanced video standards	488
21.8	Bibliographic notes	489
21.9	Suggested experiments	489
21.9.1	Horizontal bar test-pattern generator	489
21.9.2	Color channel selection circuit	489
21.9.3	Enhanced color-to-grayscale conversion circuit	489
21.9.4	Square test-pattern generator: part I	489
21.9.5	Square test-pattern generator: part II	489
21.9.6	Square test-pattern generator: part III	490
21.9.7	Square test-pattern generator: part IV	490
22	FPro Video Subsystem	491
22.1	Organization of the video subsystem	491
22.1.1	Overview	491
22.1.2	Video controller	493
22.1.3	HDL of the video controller	494
22.2	FPro video IP core	495
22.2.1	Basic functionality	495
22.2.2	Blending operation	496
22.2.3	Core architecture	498
22.2.4	Alternative core partition	500
22.3	Example video cores	500
22.3.1	Bar test-pattern generator core	500
22.3.2	Color-to-grayscale conversion core	503
22.3.3	“Dummy” core	504
22.4	FPro video synchronization core	504
22.4.1	Line buffer	505
22.4.2	Enhanced video synchronization circuit	508
22.4.3	HDL code	511
22.5	Daisy video subsystem	512
22.5.1	Subsystem overview	512

22.5.2	Interface to the video synchronization core	513
22.5.3	HDL code	513
22.5.4	Timing and performance considerations	517
22.6	Vanilla daisy FPro system	517
22.6.1	Clock management core	518
22.6.2	Updated <code>chu_io_map.svh</code>	519
22.6.3	HDL code	519
22.7	Video driver and test program	521
22.7.1	Updated <code>chu_io_map.h</code> and <code>chu_io_rw.h</code> files	521
22.7.2	GPV core driver	522
22.7.3	Test program	523
22.8	Bibliographic notes	524
22.9	Suggested experiments	525
22.9.1	Color channel selection core	525
22.9.2	Enhanced color-to-grayscale conversion core	525
22.9.3	Square test-pattern generator core	525
22.9.4	Alpha blending circuit	525
22.9.5	“Highlight” core	525
22.9.6	SVGA synchronization core	526
22.9.7	Configurable video synchronization core	526
22.9.8	Pipelined video subsystem	526
23	Sprite Core	527
23.1	Introduction	527
23.2	Basic design	528
23.2.1	Sprite RAM	528
23.2.2	In-region comparison circuit	529
23.3	Mouse pointer core	530
23.3.1	Pointer sprite RAM	530
23.3.2	Pixel generation circuit	531
23.3.3	Top-level design	532
23.4	“Ghost” character core	534
23.4.1	Multiple images and animation	534
23.4.2	Overview of the palette scheme	535
23.4.3	Ghost sprite RAM and the palette circuit	535
23.4.4	Animation timing circuit	537
23.4.5	Pixel generation circuit	537
23.4.6	Top-level design	540
23.5	Sprite core driver and test program	541
23.5.1	Sprite core driver	541
23.5.2	Test program	543
23.6	Bibliographic notes	544

23.7	Suggested experiments	544
23.7.1	Mouse pointer control with PS2 core	544
23.7.2	Emulated ghost core	544
23.7.3	Palette circuit for the mouse pointer sprite	544
23.7.4	Sprite scaling circuit	544
23.7.5	Portrait mode display	545
23.7.6	Multiple-object generation	545
23.7.7	Animation speed control	545
23.7.8	Imitated blinking LED: part I	545
23.7.9	Imitated blinking LED: part II	545
23.7.10	Imitated blinking LED: part III	546
24	On-Screen-Display Core	547
24.1	Introduction to tile graphics	547
24.2	Basic OSD design	549
24.2.1	Text-mode display	549
24.2.2	Font ROM	550
24.2.3	Tile RAM	550
24.2.4	Basic organization	551
24.3	OSD core	552
24.3.1	Font ROM	552
24.3.2	Pixel generation circuit	553
24.3.3	Top-level design	555
24.4	OSD core driver and test program	557
24.4.1	OSD core driver	557
24.4.2	Testing program	558
24.5	Bibliographic notes	559
24.6	Suggested experiments	559
24.6.1	Rotating banner	559
24.6.2	Text console	559
24.6.3	Underline for the cursor	559
24.6.4	Portrait-mode display	560
24.6.5	Font scaling circuit: part I	560
24.6.6	Font scaling circuit: part II	560
24.6.7	Extended font	560
24.6.8	Tile-based ghost core	560
25	VGA Frame Buffer Core	561
25.1	Overview	561
25.2	Frame buffer core	562
25.2.1	FPGA memory consideration	562

25.2.2	Video memory module	562
25.2.3	Address translation	563
25.2.4	Pixel generation circuit	564
25.2.5	Register map	566
25.2.6	Top-level HDL code	566
25.3	Driver and test program	567
25.3.1	Frame buffer core driver	567
25.3.2	Geometrical modeling	568
25.3.3	Test program	570
25.4	Project ideas	570
25.5	Bibliographic notes	572
25.6	Suggested experiments	572
25.6.1	Virtual prototyping board panel	572
25.6.2	Virtual analog wall clock	572
25.6.3	Geometrical model functions	572
25.6.4	Simulated “Etch a Sketch” toy	572
25.6.5	Frame buffer core with 3-bit color depth	573
25.6.6	Frame buffer core with 1-bit color depth	573
25.6.7	QVGA frame buffer core	573
25.6.8	Line drawing hardware accelerator	573
25.6.9	Bidirectional frame buffer access: part I	573
25.6.10	Bidirectional frame buffer access: part II	573

PART V EPILOGUE

26 What’s Next	577
-----------------------	------------

References	581
------------	-----

Appendix A: Tutorials	585
------------------------------	------------

A.1	Overview of Xilinx Vivado IDE	585
A.2	Short tutorial on Vivado hardware development	589
A.2.1	Create a design project	590
A.2.2	Add or create Xilinx IP core instances	591
A.2.3	Add or create HDL design files	591
A.2.4	Add a constraint file	592
A.2.5	Perform synthesis, implementation, and bitstream generation	593
A.2.6	Program an FPGA device	593
A.3	Short tutorial on Vivado simulation	594
A.3.1	Add or create HDL testbench	596
A.3.2	Perform initial simulation	596
A.3.3	Customize waveform display	597

A.4	Tutorial on IP instantiation	597
A.4.1	Dual-clock FIFO core via HDL templates	598
A.4.2	IP Catalog utility	599
A.4.3	Generate a MicroBlaze MCS component	600
A.4.4	XADC IP core	601
A.4.5	Clock management IP core	602
A.5	Short tutorial on FPro system development	604
A.5.1	Derive FPro system hardware	605
A.5.2	Export hardware configuration	605
A.5.3	Derive software	605
A.5.4	Embed elf file into FPGA's memory module and regenerate bitstream	608
A.5.5	Set up the terminal emulator program	610
A.5.6	Program an FPGA device	610
A.6	Bibliographic notes	611
	Topic Index	613

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 the capacity of FPGA devices continues to grow, a device can accommodate an SoC (system on a chip) design, which integrates a processor, memory modules, I/O peripherals, and custom hardware accelerators into a single chip. This book uses a “learning by doing” approach and illustrates the FPGA and HDL development and design process by a series of examples in the SoC context.

The examples start with simple gate-level circuits, progress gradually through the RT (register-transfer) level modules, and lead to a functional embedded system with custom I/O peripherals and hardware accelerators. A simple SoC framework, *FPro* (abbreviated from the book title “FPGA Prototyping”), is introduced as a platform to integrate all the design examples together. An FPro system contains a Xilinx MicroBlaze MCS soft-core processor, a video subsystem, and the MMIO (memory-mapped I/O) subsystem that can incorporate custom I/O cores. Except for the processor, all components are designed and coded from scratch. All the hardware and software examples can be synthesized, compiled, and physically tested on the prototyping board.

Focus and audience

Focus The primary focus of this book is on developing efficient and reliable digital systems and effectively using HDL as a tool to describe the intended hardware. The HDL language itself is not the main subject and its coverage is limited to a

small synthesizable subset. The book uses about a dozen proven code templates to provide the skeletal structures 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 helps us steer our effort to develop an innovative and efficient hardware architecture.

After discussing the fundamentals in Part I, the book illustrates more complicated and sophisticated designs in the SoC context. Along the way, readers will learn many system-level concepts, including the derivation of a soft-core processor and *IP* (*intellectual property*) core based system, the partition and integration of software and hardware, and the development of custom I/O peripherals and hardware accelerators.

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,” by which we mean the following:

- The same practice can be applied to large designs 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.

Audience and prerequisites The intended audience is students in an advanced digital design course as well as practicing engineers who wish to learn FPGA- and HDL-based developments. Readers need to have a basic knowledge of digital systems, usually a required course in electrical engineering and computer engineering curricula, and a working knowledge of the C/C++ language. Prior exposure to computer architecture, embedded system, and operating system is not necessary but will be helpful.

Changes for the MicroBlaze MCS SoC Edition

This book is the successor edition of *FPGA Prototyping by Verilog Examples: Xilinx Spartan 3 Version*. The *SystemVerilog* in the title reflects the fact that the book uses the new language constructs of SystemVerilog. The most significant change is that the new edition presents the hardware in the SoC context and covers many system-level concepts. Instead of treating each module as an isolated entity, the book integrates them into a single coherent SoC platform that allows readers to explore both hardware and software “programmability” and develop complex and interesting embedded system projects. The major revisions in this edition are the following:

- Add four general-purpose peripheral modules: multi-channel PWM (pulse width modulation), I²C controller, SPI controller, and XADC (Xilinx analog-to-digital converter) controller.
- Introduce a music synthesizer constructed with a DDFS (direct digital frequency synthesis) module and an ADSR (attack-decay-sustain-release) envelope generator.
- Expand the original video controller into a complete stream-based video subsystem that incorporates a video synchronization circuit, a test-pattern gen-

erator, an OSD (on-screen-display) controller, a sprite generator, and a frame buffer.

- Expand the coverage of timing model and provide an in-depth discussion of blocking and nonblocking statements.
- Introduce basic concepts of software-hardware co-design with Xilinx MicroBlaze MCS soft-core processor.
- Provide an overview of the bus interconnect and interface circuit.
- Introduce basic embedded system software development.
- Suggest additional modules and peripherals for interesting and challenging projects.

Logistics

FPGA prototyping board This book is prepared to be used with the *Nexys 4 DDR* FPGA prototyping board manufactured by Digilent Inc. It contains an Artix FPGA device and the needed I/O peripherals. All HDL codes and discussions of this book can be applied to this board directly. The less expensive *Basys 3* board can be used as well. This board incorporates fewer I/O peripherals and contains a smaller FPGA device.

Most peripherals discussed in the book are de facto industrial standards and the corresponding HDL codes can be used for other FPGA boards as long as they provide adequate analog interface circuits and connectors. Another option is to use stand-alone I/O peripheral modules or to construct the circuits on a breadboard.

Software The book uses the Xilinx *Vivado WebPack edition* for hardware development and Xilinx *SDK* for software development. Both software packages are free and can be downloaded from Xilinx's website.

PC accessories The design examples involve interfaces to several PC peripheral devices, including a USB keyboard, a USB mouse, a VGA compatible monitor, and a powered speaker. These accessories are widely available and probably can be obtained from an old PC.

Book organization

The book is divided into four 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, the basic language syntax, and the 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.
- 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 presents 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 the register-transfer (RT) methodology, in which the system operation is described by data transfers and manipulations among registers.
- Chapter 7 covers the methods to infer FPGA’s internal memory modules, which can then be used to construct buffers and lookup tables.
- Chapter 8 provides an in-depth coverage of the timing model and data types and discusses an alternate coding style. This chapter can be skipped without affecting the remaining chapters.

Part II introduces the hardware construction of an FPro system and the development of embedded software. A basic “vanilla” FPro system, which contains a timer core, a UART (universal asynchronous receiver and transmitter) core, a GPI (general-purpose input) core, and a GPO (general-purpose output) core, is used to illustrate the key concepts of the process. It consists of four chapters:

- Chapter 9 introduces the SoC development and provides an overview of the hardware organization and software structure of the FPro platform.
- Chapter 10 discusses the software development for an embedded system and the basic coding techniques to access low-level I/O cores.
- Chapter 11 covers the FPro bus protocol and the bus interface circuit and demonstrates the construction of basic GPI, GPO, and timer cores.
- Chapter 12 presents the construction of a more sophisticated UART core and the derivation of software device drivers.

Part III applies the techniques from Parts I and II to develop an array of I/O cores for the peripherals on the Nexys 4 DDR prototyping board. The I/O cores are constructed from scratch with custom hardware and device driver. Part III consists of nine chapters:

- Chapter 13 discusses the Xilinx device’s internal analog-to-digital converter (XADC) and derives an interface circuit to retrieve the analog readings.
- Chapter 14 presents the design of a multi-channel PWM core and demonstrates its application for LED brightness adjustment and servo motor control.
- Chapter 15 converts the seven-segment LED control circuit and the switch debouncing circuit of Part I into I/O cores and integrates them into an FPro system.
- Chapter 16 provides an overview of the SPI protocol, covers the design of an SPI controller core, and shows its operation with Nexys 4 DDR board’s ADXL362 three-axis accelerometer.
- Chapter 17 provides an overview of the I²C protocol, discusses the design of an I²C controller core, and demonstrates its operation with Nexys 4 DDR board’s ADT7420 temperature sensor.
- Chapter 18 covers the design of a PS2 controller core, which can be connected to a PS2 mouse or a PS2 keyboard, and discusses the device driver routines

to read and decode keyboard scan codes and to obtain and process mouse movement information and button activities.

- Chapter 19 discusses the construction of a DDFS (direct digital frequency synthesis) controller core with amplitude and frequency modulation and demonstrates its application as a music synthesizer.
- Chapter 20 augments the music synthesizer with an ADSR (attack-decay-sustain-release) envelope generator core, which can produce sound mimicking various music instruments.

Part IV discusses the development of a stream-based video subsystem. The subsystem provides a framework to generate and mix multiple video sources into a single video data stream for display. It consists of four chapters:

- Chapter 21 introduces the concept of stream data processing and constructs a basic video system with a test-pattern generator, a color-to-grayscale conversion circuit, and a frame synchronization circuit.
- Chapter 22 provides an overview of the FPro video subsystem framework and the FPro video core structure and demonstrates the stream interface with a line buffer.
- Chapter 23 presents the design of a sprite circuit, which adds an overlay of small animated objects on the screen, and applies the technique for a mouse pointer core and a “Pac-Man ghost character” core.
- Chapter 24 discusses the design of an OSD (on-screen-display) controller core, which produces an overlay of text similar to the subtitles on a TV screen.
- Chapter 25 covers the design of a frame buffer, which maintains a bitmap for one screen.

In addition to the main text chapters, the book includes an Appendix with four tutorials. The tutorials consist of the following:

- Develop, synthesize, and implement a digital circuit on the Nexys 4 DDR board with Vivado.
- Perform simulation of an HDL program with Vivado’s built-in simulator.
- Configure and instantiate Xilinx IP cores.
- Construct a basic FPro system with a Xilinx microBlaze MCS IP core and develop software with the Xilinx SDK platform.

Companion Website

On an accompanying website (http://academic.csuohio.edu/chu_p) additional information is available, including the following materials:

- Errata
- HDL and C/C++ code listings and relevant files
- Links to synthesis and simulation software
- Links to reference materials

The printed book contains a number of color figures. They are shown as grayscale in the printed version. These figures can be found in full color on the website as well.

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 website provides an updated errata sheet and a place to report errors.

P. P. CHU

Cleveland, Ohio
February 2018

ACKNOWLEDGMENTS

Part of this material is based upon work supported by the National Science Foundation under Grant No. 1504030. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

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

P. P. Chu

PART I

BASIC DIGITAL CIRCUITS DEVELOPMENT

CHAPTER 1

GATE-LEVEL COMBINATIONAL CIRCUIT

HDL (hardware description language) is used to describe and model digital systems. SystemVerilog is one of the major HDLs. In this chapter, we use a simple comparator to illustrate the skeleton of a SystemVerilog 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 remaining operators and constructs and examine the register-transfer-level combinational circuits, which are composed of intermediate-sized components, such as adders, comparators, and multiplexers.

1.1 INTRODUCTION

1.1.1 Brief history of Verilog and SystemVerilog

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 and the document is known as the *LRM (Language Reference Manual)*. The standard was ratified in 1995 (known as Verilog-1995) and significantly revised in 2001 (known as Verilog-2001). A further revision, which contains a few minor changes, was published in 2005. Unless otherwise specified, the term “Verilog” used in the book is referred to Verilog-2001.

Verilog was developed for gate-level and register-transfer-level design and modeling and it did not include advanced high-level verification features, such as assertions, functional coverage, and constrained random testing. SystemVerilog first served as an *extension of Verilog* that supports the verification features. The extension was ratified by IEEE in 2005 and formally defined by IEEE Standard 1800. It is referred to as SystemVerilog-2005.

In 2009, Verilog and SystemVerilog were combined into a single standard and defined by IEEE Standard 1800. The merged languages are called SystemVerilog and referred to as SystemVerilog-2009. The merge and name selection implies that Verilog is now part of SystemVerilog and the Verilog language has ceased to exist.

The merge and naming scheme may cause some confusion. SystemVerilog-2005 is a pure hardware verification language but the newer SystemVerilog (SystemVerilog-2009 and beyond) is a *hardware description and verification language* that incorporates both design and verification features into a single framework.

Unless otherwise specified, the term “SystemVerilog” used in the book is referred to SystemVerilog-2009, which includes hardware description portion and is a “superset” of the original Verilog.

1.1.2 Book coverage

SystemVerilog is an extremely complex language. Only a small subset of the language constructs is intended to describe gate-level and register-transfer-level systems and even a smaller subset can be recognized by the synthesis software tool and transformed into physical hardware.

The focus of this book is on hardware design rather than on the language. We introduce the key SystemVerilog synthesis constructs by examining a collection of examples. Although the syntax of SystemVerilog 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 nondeterministic behavior of SystemVerilog can lead to difficult-to-detect errors and can 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. The illustration of the covered language subset is shown in Figure 1.1. Several advanced synthesis related topics are examined further in Chapter 8 and more detailed SystemVerilog coverage may be explored through the sources listed in the bibliographic section at the end of the chapter.

Besides merging the two standards, SystemVerilog-2009 made many enhancements in the “hardware description portion” of the original Verilog-2001 standard. We use some of these new features in the book. The book occasionally includes paragraphs to explain the difference between a new SystemVerilog-2009 feature and the original Verilog-2001 construct. They are highlighted by a **Verilog FYI** side bar, as shown at left. The main purpose of these paragraphs is to help the reader understand the older Verilog codes. Note that SystemVerilog is backward-compatible with Verilog-2001 and thus these codes can be accepted by the SystemVerilog synthesis tool as well. The paragraphs with side bars can be skipped without affecting the subsequent reading.

**Verilog
FYI**

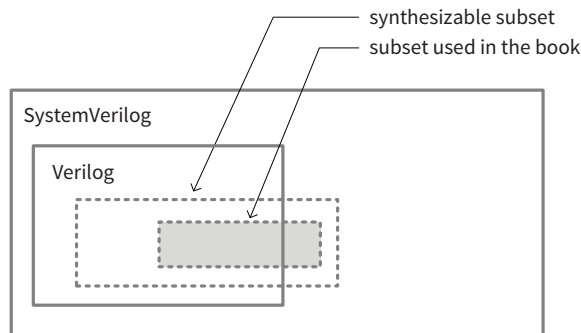


Figure 1.1 Subset covered in the book.

Table 1.1 Truth table of 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.

Suppose 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 SystemVerilog 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 logic i0, i1,
        output logic eq
    );

    // signal declaration
    logic p0, p1;

    // body
    // sum of two product terms
    assign eq = p0 | p1;

```

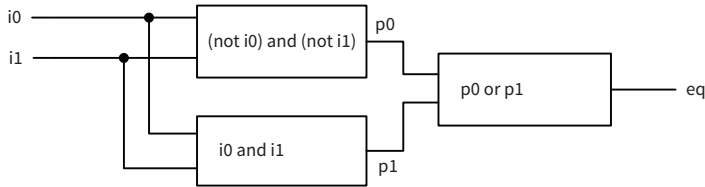


Figure 1.2 Graphical representation of a comparator program.

```
// product terms
assign p0 = ~i0 & ~i1;
assign p1 = i0 & i1;
endmodule
```

The best way to understand an HDL 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 two sections.

The graphical representation of this program is shown in Figure 1.2. The three continuous assignments constitute the three circuit parts. The connections among these parts are specified implicitly by the signal and port names. The order of the continuous statements is clearly irrelevant and the three statements can be rearranged arbitrarily.

1.3 BASIC LEXICAL ELEMENTS AND DATA TYPES

1.3.1 Lexical elements

The basic SystemVerilog lexical elements include identifiers, keyword, white space, and comment.

Identifier An *identifier* gives a unique name to an object, such as `eq`, `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.

SystemVerilog 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.

Keyword A *Keyword* is a predefined identifier that is used to describe language constructs. In this book, we use boldface type for SystemVerilog keywords, such as **module** and **logic** 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 SystemVerilog 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. SystemVerilog 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.

1.3.2 Data types used in the book

SystemVerilog supports a rich collection of data types. However, we only use a very small restricted set in the book to describe the circuit. The set consists of the following:

1. the **logic** type
2. the **integer** type
3. the **tri** type
4. the user-defined enumerate type

The **logic** type is the most commonly used data type in design. It represents the value of a one-bit signal or the content of a one-bit memory element. The **logic** type can assume a value from a *four-state set*:

- 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 tristate 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.

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

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

The one-dimensional array can be interpreted as a collection of independent bits or an unsigned binary number. 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 (most significant bit) of a binary number.

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

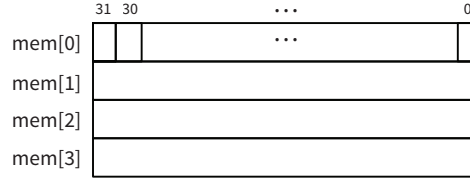


Figure 1.3 Illustration of a two-dimensional array.

```
logic [31:0] mem [0:3]; // 4-by-32 memory
```

Note that the outer dimension (i.e., [0:3]) is in ascending order, representing the memory module depicted in Figure 1.3.

The **integer** type is a special case of one-dimensional **logic** array. Its size is fixed at 32 bits and it is interpreted as a signed binary number. We use the **integer** type mainly for constants and parameters to represent threshold values, array boundaries, etc.

In our book, the **tri** type is only used to infer the tristate buffer of a bidirectional port and the user-defined enumerate type is used to represent the symbolic states of an FSM (finite state machine). These types are discussed in more detail in Sections 3.1.9 and 5.2.1.

1.3.3 Number representation

The value of a one-dimensional **logic** array is represented as a constant number. Its general format 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.

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.

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.


```

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. Since the book focuses on design description, we only use the **logic** type for the input and output ports and use the **tri** type for bidirectional port

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
    logic i0, i1;
    logic eq;

```

We do not use this format in this book.

1.4.2 Signal declaration

The declaration portion specifies the internal variables and local parameters used in the module. Since the variables frequently resemble the interconnecting wires between the circuit parts, as shown in Figure 1.2, we call them “signals” when appropriate.

The simplified syntax of signal declaration is

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

Two internal signals are declared in Listing 1.1:

```
logic p0, p1;
```

Note that an identifier does not need to be declared explicitly. The previous declaration statement is actually optional. If a declaration is omitted, the signal is assumed to be an *implicit net*. 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.

1.4.3 Program body

The program body of a synthesizable SystemVerilog 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. There are three continuous assignments in Listing 1.1 and they correspond to the three circuit parts shown in Figure 1.2.

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 a more complex circuit operation. The always block is discussed in Section 3.2.

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 pre-designed modules as subsystems of the current module. Instantiation is discussed in Section 1.5.

1.4.4 Concurrent semantics

Although the “appearance” of an HDL program is somewhat like a traditional programming language, such as C, its semantics is very different. The statements in a C programs are run on a centralized processor and executed sequentially. The statements of an HDL program are “autonomous” and executed concurrently. For example, consider the statement

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

It is executed as follows:

1. When a signal on the left-hand-side expression (i.e., p0 or p1) changes, the statement is activated.
2. The left-hand-side expression (i.e., p0 | p1) is evaluated.
3. The evaluated result is passed to the right-hand signal after a delay (an implicit delta delay or an explicitly specified delay).
4. Repeat the process continuously.

Note that the execution resembles the operation of a circuit.

The continuous assignments can be activated at the same time and run concurrently. Its behavior is totally different from a C program statement. We intentionally put the assignment

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

as the first line of the program body in Listing 1.1. The arrangement will lead to erroneous result in a traditional programming C language but has no effect on an HDL program since the order of the continuous assignments.

The execution of always block and component instantiation are more complex but can be reasoned in a similar way. In summary, the continuous assignment,

always block, and module instantiation can be treated as “concurrent building constructs.” Each construct *runs autonomously and continuously* and the overall operation of the code is executed in parallel.

1.4.5 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 1.2.

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

```

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

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

1.5 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. SystemVerilog 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 1.4.5 is to utilize previously constructed 1-bit comparators as the building blocks. The diagram is shown in Figure 1.4, 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 1.3.

Listing 1.3 Structural description of a 2-bit comparator

```

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

```

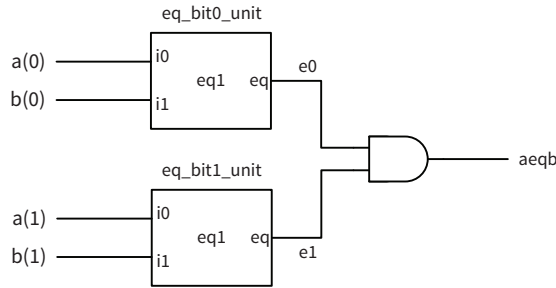


Figure 1.4 Construction of a 2-bit comparator from 1-bit comparators.

```
// internal signal declaration
logic 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 line 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 remaining 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 1.3, 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 1.4. 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.

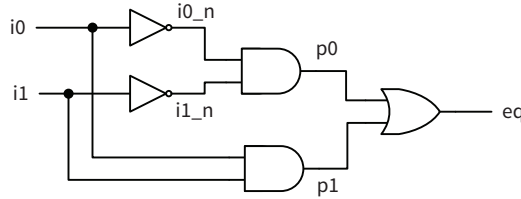


Figure 1.5 Low-level diagram of a 1-bit comparator.

The port names and signal names are sometimes identical and these mappings can be represented as “.” in SystemVerilog. For example, the instantiation statement

```
eq1 eq_unit (.i0(i0), .i1(i1), .eq(eq));
```

can be abbreviated as

```
eq1 eq_unit (.*);
```

and the instantiation statement

```
eq1 eq_unit (.i0(i0), .i1(i1), .eq(result));
```

can be abbreviated as

```
eq1 eq_unit (.*, .eq(result));
```

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 1.3 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 lead to difficult-to-find bugs. We always use the connection-by-name scheme in this book.

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 1.5. The corresponding primitive-based code is shown in Listing 1.4.

**Verilog
FYI**

**Verilog
FYI**

Listing 1.4 Implementation with Verilog primitive

```

module eq1_primitive
(
    input logic i0, i1,
    output logic eq
);

    // internal signal declaration
    logic 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 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 1.5.

**Verilog
FYI**

Listing 1.5 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

```

A UDP 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.

1.6 TOP-LEVEL SIGNAL MAPPING

When an HDL program is targeted to a physical device of a prototyping board, the design is subject to a variety of *constraints*. One constraint is the locations of the I/O pins. For example, the switches and LEDs of the board are “pre-wired” to specific I/O pins of the FPGA device and they cannot be altered. The pin assignment is defined in a *constraint file*, which is processed in conjunction with HDL files.

The designs of this book use a constraint file that specifies the pin assignment for all the I/O signals on the Nexys 4 DDR prototyping board. To use this file, the top-level HDL module must have the same predefined I/O signal names. This can be achieved by creating an HDL file to “wrap” the original design and map

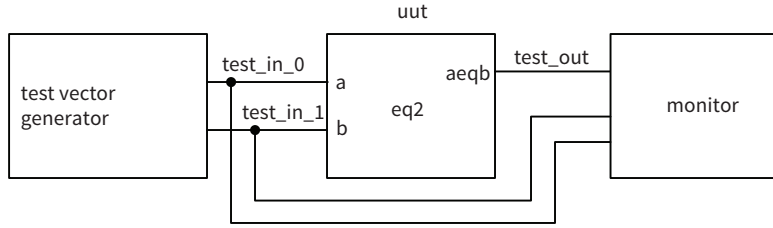


Figure 1.6 Testbench for a 2-bit comparator.

its original I/O signals to the prototyping board's I/O signals. For example, we name the I/O pins connected to the slide switches and LEDs as **sw** and **led** and specify their pin assignment in the constraint file. For a physical implementation, the **a** and **b** signals of the previous comparator circuit can be connected to the four switches and the output, **aeqb**, can be connected to an LED. The corresponding wrapping code is shown in Listing 1.6.

Listing 1.6 Top-level wrapping circuit

```

module eq2_top
(
    input logic [3:0] sw,
    output logic [0:0] led
);

    // body
    // instantiate 2-bit comparator
    eq2 eq_unit (.a(sw[3:2]), .b(sw[1:0]), .aeqb(led[0]));
endmodule

```

The code essentially maps the “logical” port names of the comparator to the physical signals on the prototyping board. Note that the output **led** signal is defined as a one-element vector to accommodate future expansion. The procedure to include the constraint file is demonstrated in Appendix A.2.

1.7 TESTBENCH

After code is developed, it can be *simulated* in a host computer to verify the correctness of the circuit operation and then *synthesized* to a physical device. Simulation is usually performed within the same language framework. We create a special program, known as a *testbench*, to mimic a physical lab bench.

The development of testbench and verification are beyond the scope of this book. We just provide several examples to illustrate the basic concepts. The templates can be used to simulate and observe inputs and outputs of a simple circuit. The sketch of a 2-bit comparator testbench is shown in Figure 1.6. The **uut** segment is the unit under test, the **test vector generator** segment generates testing input patterns, and the **monitor** segment 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
    logic [1:0] test_in0, test_in1;
    logic 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 language 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 the test vector 2:

```

test_in0 = 2'b01;
test_in1 = 2'b00;
# 200;

```

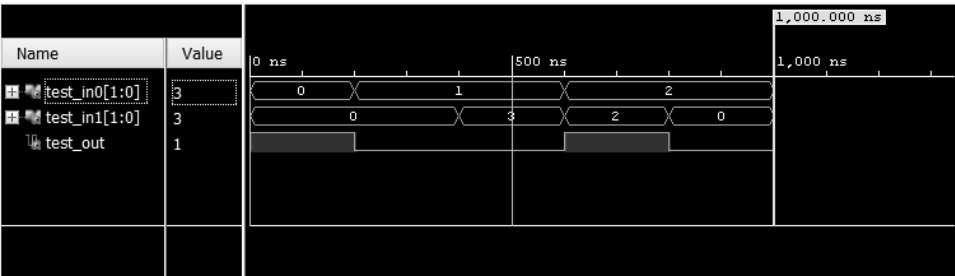


Figure 1.7 Simulated waveforms.

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 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 1.7.

Writing code for a comprehensive testbench requires detailed knowledge of SystemVerilog and is beyond the scope of this book. However, this listing can serve as a testbench template for other simple combinational circuits. We can substitute the `uut` instance and modify the test patterns according to the new circuit.

1.8 BIBLIOGRAPHIC NOTES

In this book, a short bibliographic section is included in the end of each chapter to provide the most relevant references for further exploration. A more comprehensive bibliography can be found in the end of the book.

SystemVerilog is a very complex language. The standard is specified in *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1364-2001*. *Logic Design and Verification Using SystemVerilog*, by D. Thomas highlights the usage and capability of the language. *SystemVerilog for Design, second ed.* by S. Sutherland et al. and *SystemVerilog for Verification* by T. Fitzpatrick et al. provide detailed coverage on the design and modeling portion and the verification portion of the language, respectively. Derivation of the testbench for a large digital system is a difficult task. *Writing Testbenches Using SystemVerilog* 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 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 2.6.1. 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 2.6.2. The code can be simulated and synthesized after we complete Chapter 2.

CHAPTER 2

OVERVIEW OF FPGA AND EDA SOFTWARE

An FPGA (field-programmable gate array) prototyping board is used to implement the design examples and projects of this book. We provide an overview of FPGA devices, the Nexys 4 DDR prototyping board, and the development process in this chapter.

2.1 FPGA

2.1.1 Overview of a general FPGA device

An *FPGA* (*field-programmable gate array*) 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 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*.

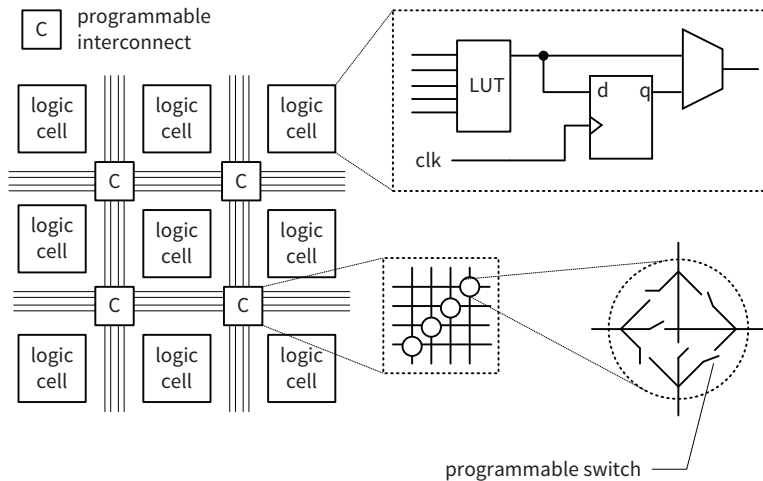


Figure 2.1 Conceptual structure of an FPGA device.

LUT-based logic cell A logic cell usually contains a small configurable combinational circuit with a D FF (D-type flip-flop). The most common method to implement a configurable combinational circuit is an *LUT* (*lookup table*). An n -input LUT can be considered as a small 2^n -by-1 memory. By properly writing the memory content, we can use the LUT to implement any n -input combinational function. The conceptual diagram of a five-input LUT-based logic cell is shown in the top right of Figure 2.1. 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.

2.1.2 Overview of the Xilinx Artix-7 devices

This book uses Xilinx Artix-7 family FPGA devices and we provide a brief overview of this device in this section.

Logic cell, slice, and CLB The most basic element of an Artix-7 device is a *logic cell* (*LC*). A logic cell contains one LUT, which can be configured either as one 6-input LUT or as two 5-input LUTs, and two FFs. 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. Some LUTs can also be configured as a small *distributed SRAM* (static random access memory) module or a *shift register*. To increase flexibility and improve performance, eight logic cells are combined together with a special internal routing structure. In Xilinx terms, four logic cells are grouped to form a *slice*, and two slices are grouped to form a *configurable logic block* (CLB).

Table 2.1 Devices in the Artix-7 family

Device	Num. of LCs	Num. of 36-Kb BRAMs	BRAM bits	Num. of DSP slices	Num. of MMCMs
XC7A15T	16,640	25	900K	45	5
XC7A35T	33,280	50	1,800K	90	5
XC7A50T	52,160	75	2,700K	120	5
XC7A75T	75,520	105	3,780K	180	6
XC7A100T	101,440	135	4,860K	240	6
XC7A200T	215,360	365	13,140K	740	10

Macro cell The Artix-7 device contains several types of macro cells. The *MMCM* (*mixed-mode clock manager*) macro cell is a clock management core that can produce a wide range of frequencies from a single oscillator input, reduce clock skew, and adjust the phase shift of a clock signal. The *BRAM* (*block random access memory*) macro cell is a 36K-bit dual-port synchronous SRAM that can be arranged in various types of configurations. The *DSP* (*digital signal processing*) macro cell is composed of a 25-by-18 binary multiplier and a 48-bit accumulator and is intended to support computation intensive DSP algorithms. An *IOB* (*input/output block*) macro cell is associated with a physical I/O pin of the FPGA device. It can be configured to support a wide variety of I/O signaling standards and high-speed serial data links. The *XADC* (*Xilinx analog-to-digital converter*) contains two 12-bit analog-to-digital converters. In addition to these, the device may include special blocks for the gigabit ethernet transceivers and the PCI express bus.

Devices in the Artix-7 family Although Artix-7 FPGA devices have similar types of logic cells and macro cells, their densities differ. The family contains an array of devices of various densities. The numbers of logic cells, 36K-bit BRAMs, DSP slices, and MMCMs of the devices are summarized in Table 2.1. The *Nexys 4 DDR* prototyping board used in the book contains an Artix XC7A100T device. The simpler *Basys 3* board contains a smaller Artix XC7A35T device

2.2 OVERVIEW OF THE DIGILENT NEXYS 4 DDR BOARD

The Digilent Nexys 4 DDR board is designed around an Artix XC7A100T device and has an array of built-in peripherals. The layouts of the board are shown in Figure 2.2.

The main components and connectors are as follows:

1. Power jack for optional external power supply
2. Shared USB JTAG and UART port
3. Artix XC7A100T FPGA device
4. Pmod port (JD)
5. Pmod port (JC)
6. Sixteen discrete LEDs
7. Sixteen slide switches
8. Temperature sensor
9. Eight-digit seven-segment LED display

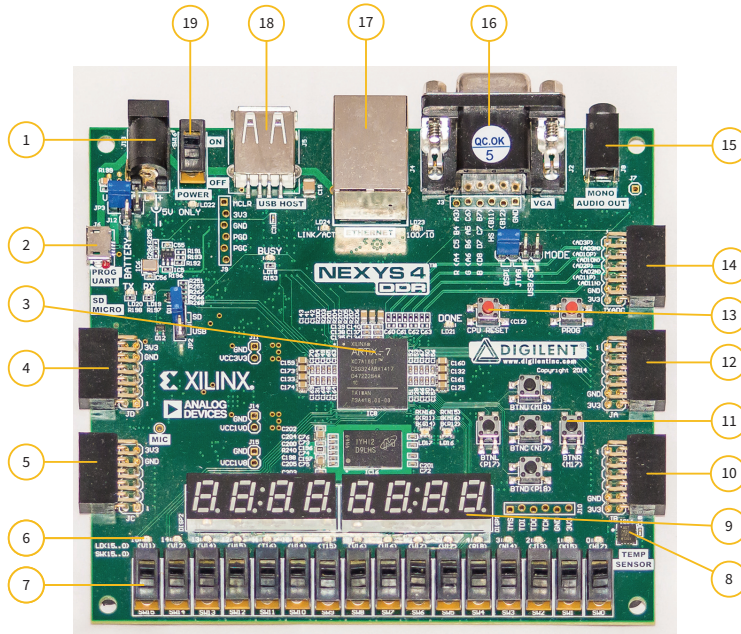


Figure 2.2 Nexys 4 DDR board.

10. Pmod port (JB)
11. Five pushbutton switches
12. Pmod port (JA)
13. Soft-core processor reset button
14. Pmod port with analog input (connected to XADC)
15. Audio jack
16. VGA port
17. Ethernet connector
18. USB host port (connected to USB mouse/keyboard)
19. Power-on switch

2.3 DEVELOPMENT FLOW

The simplified development flow of an FPGA-based system is shown in Figure 2.3. 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 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 as follows:

1. Design the system and derive the HDL file(s). We may need to add a separate constraint file to specify certain implementation constraints, such as the pin assignment and the clock frequency.

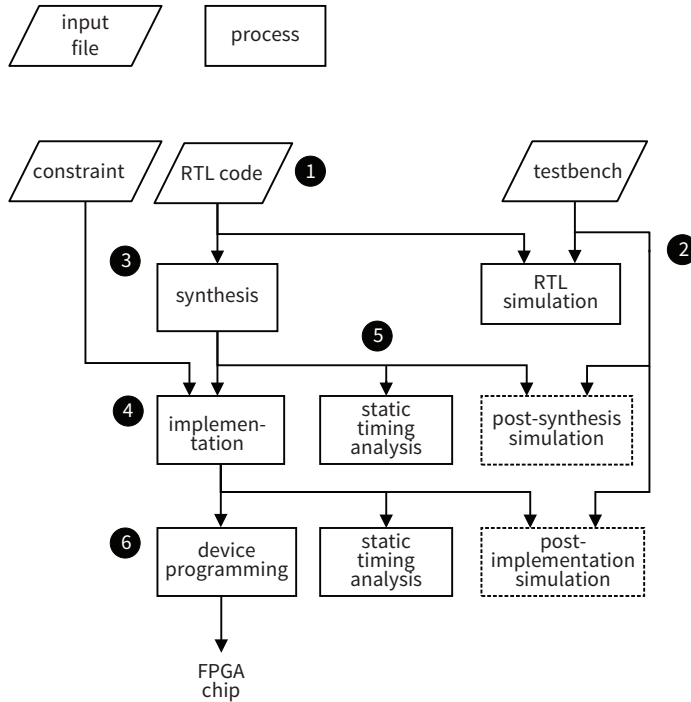


Figure 2.3 Development flow.

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*. The simulation is performed to verify code syntax and to confirm that the HDL description meets the intended specification and functions.
3. Perform *synthesis*. 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.
4. Perform *implementation*. The *implementation* process consists of three smaller subprocesses: *translate*, *technology mapping*, and *placement and routing*. The *translate* subprocess merges multiple design files to a single netlist. The *technology mapping* subprocess maps the generic gates in the netlist to FPGA's logic cells. The *placement and routing* subprocess derives the physical layout inside the FPGA chip. It places the cells in physical locations and determines the routes to connect various signals.
5. Examine the timing report. In the Xilinx flow, *static timing analysis*, which determines various timing parameters, such as the setup time slack, is performed at the end of the synthesis process and at the end of the implementation process. The latter is more accurate since the wire delays (correlated to path's length) are known and can be used for calculation.
6. Generate and download the programming file. In this process, a configuration file, which is also known as a *bit file*, is generated according to the final netlist. This file is downloaded to an FPGA device serially to configure the logic cells and programmable switches. The physical circuit can be verified accordingly.

The optional *post-synthesis simulation* can be performed after synthesis, and the optional *post-implementation simulation* can be performed after implementation. Post-synthesis simulation uses a synthesized netlist to replace the RTL description and checks the correctness of the synthesis process. Post-implementation uses the final netlist, along with detailed wire delay timing data, to perform simulation. Because of the complexity of the netlist, post-synthesis and post-implementation 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 the RTL simulation to check the correctness of the HDL code and use static timing analysis to verify the relevant timing information. Both post-synthesis and post-implementation simulations may be omitted from the development flow.

2.4 XILINX VIVADO DESIGN SUITE

We use *Vivado Design Suite* for hardware development in this book. Vivado is an integrated design environment for the Xilinx FPGA product and incorporates all the software tools discussed in Figure 2.3. A typical Vivado window is shown in Figure 2.4. A “watered-down” version, *Vivado WebPack edition*, can be downloaded for free and is adequate for the designs and projects in this book.

As FPGA capability and capacity continue to grow, the EDA (electronic design automation) tools evolve in a similar pace. The software is updated and patched almost at a quarterly basis. While the detailed description of the vendor’s tools is beyond the scope of this book, several short tutorials are provided in the Appendix. Appendix A.1 provides an overview of the Vivado Design Suite environment, Appendix A.2 illustrates the hardware development flow, and Appendix A.3 demonstrates the RTL simulation.

2.5 BIBLIOGRAPHIC NOTES

Relevant information for the Artix-7 device can be found in its data sheets. Xilinx’s *7 Series FPGAs Overview* provides a high-level overview and its user guide, *UG474 7 Series FPGAs Configurable Logic Block User Guide* gives a detailed explanation of the logic cells. *The Design Warrior’s Guide to FPGAs* by Clive Maxfield provides a comprehensive review of FPGA-related issues.

The detailed layout and relevant information of the Nexys 4 DDR board can be found in *Nexys 4 DDR FPGA Board Reference Manual*.

2.6 SUGGESTED EXPERIMENTS

2.6.1 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:

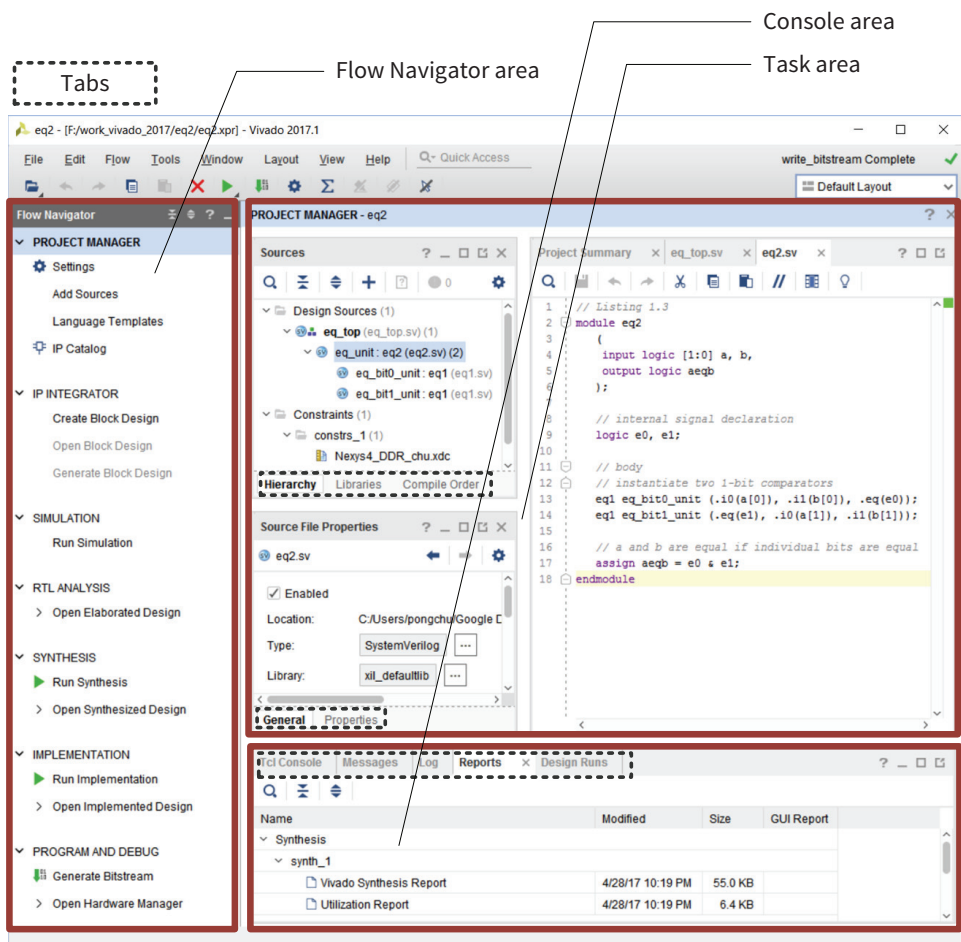


Figure 2.4 Vivado window.

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

<i>en</i>	Input		Output
	<i>a</i> (1)	<i>a</i> (0)	<i>bcode</i>
0	—	—	0000
1	0	0	0001
1	0	1	0010
1	1	0	0100
1	1	1	1000

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 obtained 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.

2.6.2 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.