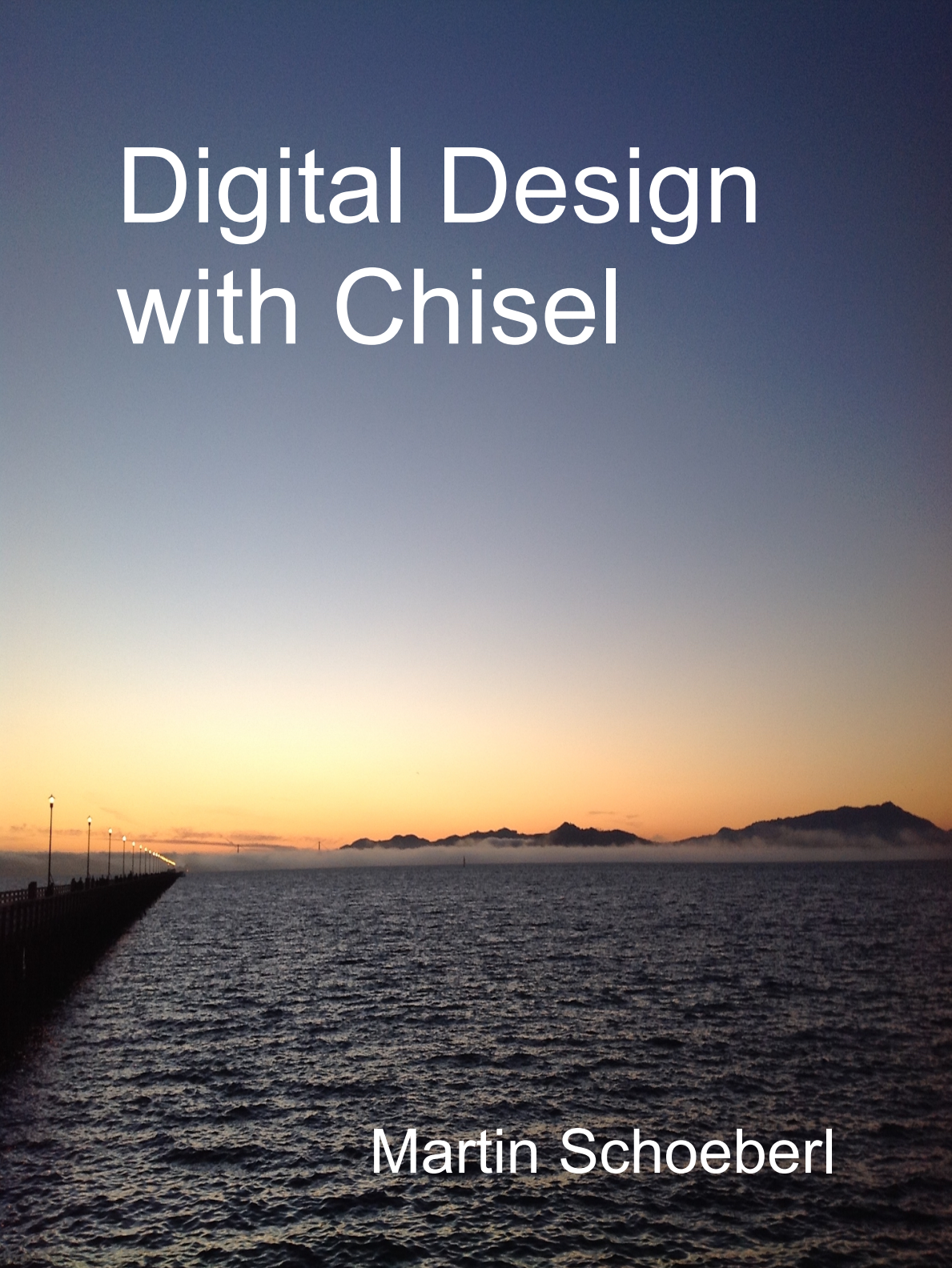
Digital Design with Chisel



Martin Schoeberl

Digital Design with Chisel

Seventh Edition

Digital Design with Chisel

Seventh Edition

Martin Schoeberl

Copyright © 2016–2026 Martin Schoeberl



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License. http://creativecommons.org/licenses/by-sa/4.0/

Email: martin@jopdesign.com
Visit the source at https://github.com/schoeberl/chisel-book

The first edition was published in 2019 by Kindle Direct Publishing,
https://kdp.amazon.com/

Library of Congress Cataloging-in-Publication Data

Schoeberl, Martin

Digital Design with Chisel
Martin Schoeberl
Includes bibliographical references and an index.
ISBN 9781689336031

Manufactured in the United States of America.
Typeset by Martin Schoeberl.

Contents

Foreword xv

Preface xvii

1 Introduction 1
- 1.1 Installing Chisel and FPGA Tools . . . 2
  - 1.1.1 macOS . . . 3
  - 1.1.2 Linux/Ubuntu . . . 3
  - 1.1.3 Windows . . . 3
  - 1.1.4 FPGA Tools . . . 3
- 1.2 Hello World . . . 4
- 1.3 Chisel Hello World . . . 4
- 1.4 An IDE for Chisel . . . 4
- 1.5 Source Access and eBook Features . . . 6
- 1.6 Further Reading . . . 7
- 1.7 Exercises . . . 8

2 Basic Components 11
- 2.1 Chisel Types and Constants . . . 11
- 2.2 Combinational Circuits . . . 13
  - 2.2.1 Multiplexer . . . 16
- 2.3 Registers . . . 16
  - 2.3.1 Counting . . . 18
- 2.4 Structure with Bundle and Vec . . . 18
  - 2.4.1 Bundle . . . 18
  - 2.4.2 Vec . . . 19
- 2.5 Wire, Reg, and IO . . . 25
- 2.6 Chisel Generates Hardware . . . 26
- 2.7 Exercises . . . 26

3 Build Process and Testing **29**
3.1 Building your Project with sbt . . . 29
3.1.1 Source Organization . . . 29
3.1.2 Running sbt . . . 31
3.1.3 Generating Verilog . . . 32
3.1.4 Tool Flow . . . 33
3.1.5 Chisel Versions . . . 33
3.1.6 Using a GitHub Template . . . 36
3.2 Testing with Chisel . . . 37
3.2.1 ScalaTest . . . 38
3.2.2 ChiselTest . . . 39
3.2.3 Waveforms . . . 43
3.2.4 printf Debugging . . . 45
3.3 Exercises . . . 46
3.3.1 A Minimal Project . . . 46
3.3.2 A Testing Exercise . . . 48

4 Components **49**
4.1 Components in Chisel are Modules . . . 49
4.2 Nested Components . . . 53
4.3 An Arithmetic Logic Unit . . . 56
4.4 Bulk Connections . . . 59

5 Combinational Building Blocks **61**
5.1 Combinational Circuits . . . 61
5.2 Decoder . . . 63
5.3 Encoder . . . 65
5.4 Arbiter . . . 67
5.5 Priority Encoder . . . 70
5.6 Comparator . . . 71
5.7 Exercise . . . 72

6 Sequential Building Blocks **73**
6.1 Registers . . . 73
6.2 Counters . . . 78
6.2.1 Counting Up and Down . . . 79
6.2.2 Generating Timing with Counters . . . 80
6.2.3 The Nerd Counter . . . 82

6.2.4 A Timer . . . 83
6.2.5 Pulse-Width Modulation . . . 83
6.3 Shift Registers . . . 87
6.3.1 Shift Register with Parallel Output . . . 87
6.3.2 Shift Register with Parallel Load . . . 88
6.4 Memory . . . 89
6.5 Exercises . . . 94

7 Input Processing 97
7.1 Asynchronous Input . . . 97
7.2 Debouncing . . . 98
7.3 Filtering of the Input Signal . . . 100
7.4 Combining the Input Processing with Functions . . . 102
7.5 Synchronizing Reset . . . 104
7.6 Exercise . . . 105

8 Finite-State Machines 107
8.1 Basic Finite-State Machine . . . 107
8.2 Faster Output with a Mealy FSM . . . 111
8.3 Moore versus Mealy . . . 115
8.4 Exercise . . . 117

9 Communicating State Machines 119
9.1 A Light Flasher Example . . . 119
9.2 State Machine with Datapath . . . 124
9.3 Ready/Valid Interface . . . 130

10 Hardware Generators 135
10.1 A Little Bit of Scala . . . 135
10.2 Lightweight Components with Functions . . . 137
10.3 Generate Combinational Logic . . . 139
10.3.1 File Reading . . . 140
10.3.2 Type Conversion . . . 142
10.4 Configuration with Parameters . . . 143
10.4.1 Simple Parameters . . . 143
10.4.2 Case Classes . . . 144
10.4.3 Functions with Type Parameters . . . 145
10.4.4 Modules with Type Parameters . . . 146

10.4.5 Parameterized Bundles 147
10.4.6 Optional Ports 148
10.5 Use Inheritance 150
10.6 Hardware Generation with Functional Programming 152
10.6.1 Minimum Search Example 154
10.6.2 An Arbitration Tree 156

11 Example Designs **161**
11.1 FIFO Buffer 161
11.2 A Serial Port 164
11.3 FIFO Design Variations 169
11.3.1 Parameterizing FIFOs 169
11.3.2 Redesigning the Bubble FIFO 172
11.3.3 Double Buffer FIFO 174
11.3.4 FIFO with Register Memory 176
11.3.5 FIFO with On-Chip Memory 180
11.4 A Multi-clock Memory 183
11.5 Exercises 184
11.5.1 Explore the Bubble FIFO 184
11.5.2 The UART 185
11.5.3 FIFO Exploration 186

12 Interconnect **187**
12.1 A Classic Microprocessor Bus 187
12.2 An On-Chip Bus 188
12.2.1 Combinational Handshake 190
12.2.2 Pipelined Handshake 191
12.2.3 Example IO Device 194
12.2.4 Memory Mapped Devices 194
12.3 Bus and Interface Standards 198
12.3.1 Wishbone 198
12.3.2 AXI 200
12.3.3 Open Core Protocol 200
12.3.4 Further Bus Specifications 201
12.4 Exercise 201

13 Debugging, Testing, and Verification **203**
13.1 Debugging 203

13.2 Testing in Chisel . . . 204
13.2.1 Use Functions . . . 205
13.2.2 Selecting Tests . . . 208
13.2.3 Accessing Internal Signals . . . 209
13.2.4 Multithreaded Testing . . . 210
13.2.5 Simulator Backends . . . 212
13.3 Assertions and Formal Verification . . . 213
13.4 Exercise . . . 215

14 Design of a Processor **217**
14.1 The Instruction Set Architecture . . . 217
14.2 The Datapath . . . 221
14.3 Start with an ALU . . . 221
14.4 Decoding Instructions . . . 226
14.5 Assembling Instructions . . . 229
14.6 The Instruction Memory . . . 231
14.7 A State Machine with Data Path Implementation . . . 231
14.8 Implementation Variations . . . 236
14.9 Exercise . . . 236

15 A RISC-V Pipeline **237**
15.1 The RISC-V Instruction Set Architecture . . . 237
15.2 Pipeline Stage Definition . . . 239
15.3 Number of Pipeline Stages . . . 239
15.4 The Wildcat Pipeline . . . 240
15.4.1 Top Level . . . 240
15.4.2 Instruction Fetch . . . 242
15.4.3 Instruction Decode and Register File Read . . . 243
15.4.4 Execute and Memory Read . . . 245
15.5 Summary and Exercise . . . 250

16 Contributing to Chisel **251**
16.1 Publishing a Chisel Library . . . 251
16.1.1 Using a Library . . . 252
16.1.2 Prerequisite . . . 252
16.1.3 Library Setup . . . 253
16.1.4 Regular Publishing . . . 254

16.2 Contributing to Chisel 254
16.2.1 Setup the Development Environment 254
16.2.2 Testing 255
16.2.3 Contribute with a Pull Request 256
16.3 Exercise 256

17 Summary **257**

A VHDL and Verilog **259**
A.1 Code Examples 259
A.1.1 Components 259
A.1.2 Using a Component 261
A.1.3 Register 263
A.1.4 Combinational Blocks 264
A.1.5 Advanced Chisel Features 267
A.2 External Modules and Integration of Legacy Code 268
A.3 Exercise 271

B Reserved Keywords **273**

C Chisel Projects **275**

D Acronyms **277**

Bibliography **281**

Index **285**

List of Figures

2.1 Logic for the expression `(a & b)| c`. The wires can be a single bit or multiple bits. The Chisel expression, and the schematics are the same. . . . 13
2.2 A basic 2:1 multiplexer. . . . 16
2.3 A D flip-flop based register with a synchronous reset to 0. . . . 17
2.4 A vector wrapped in a `Wire` is just a multiplexer. . . . 20
2.5 A vector of registers. . . . 22

3.1 Source tree of a Chisel project (using `sbt`) . . . 30
3.2 Tool flow of the Chisel ecosystem. . . . 34

4.1 An adder component. . . . 50
4.2 A register components. . . . 50
4.3 A counter built out of components. . . . 51
4.4 A design consisting of a hierarchy of components. . . . 54
4.5 An arithmetic logic unit, or ALU for short. . . . 57

5.1 A chain of multiplexers. . . . 62
5.2 A 2-bit to 4-bit decoder. . . . 64
5.3 A 4-bit to 2-bit encoder. . . . 66
5.4 A symbol for a 4-bit arbiter. . . . 68
5.5 A 4-bit arbiter. . . . 69
5.6 With an arbiter and an encoder, we can build a priority encoder. . . . 71
5.7 A simple comparator. . . . 71

6.1 A D flip-flop-based register. . . . 73
6.2 A D flip-flop based register with a synchronous reset. . . . 75
6.3 A waveform diagram for a register with a reset. . . . 75
6.4 A D flip-flop-based register with an enable signal. . . . 76
6.5 A waveform diagram for a register with an enable signal. . . . 77

6.6 An adder and a register result in counter. . . . 78
6.7 Counting events. . . . 79
6.8 A waveform diagram for generating a slow frequency tick. . . . 81
6.9 Using the slow frequency tick. . . . 82
6.10 A one-shot timer. . . . 84
6.11 Pulse-width modulation. . . . 84
6.12 A 4-stage shift register. . . . 86
6.13 A 4-bit shift register with parallel output. . . . 88
6.14 A 4-bit shift register with parallel load. . . . 88
6.15 A synchronous memory. . . . 89
6.16 A synchronous memory with forwarding for a defined read-during-write behavior. . . . 91

7.1 Input synchronizer. . . . 98
7.2 Debouncing an input signal. . . . 99
7.3 Majority voting on the sampled input signal. . . . 101

8.1 A finite-state machine (Moore type). . . . 107
8.2 The state diagram of an alarm FSM. . . . 108
8.3 A rising edge detector (Mealy type FSM). . . . 112
8.4 A Mealy type finite-state machine. . . . 112
8.5 The state diagram of the rising edge detector as Mealy FSM. . . . 113
8.6 The state diagram of the rising edge detector as Moore FSM. . . . 115
8.7 Mealy and a Moore FSM waveform for rising edge detection. . . . 115

9.1 The light flasher split into a Master FSM and a Timer FSM. . . . 120
9.2 The light flasher split into a Master FSM, a Timer FSM, and a Counter FSM. . . . 122
9.3 A state machine with a datapath. . . . 125
9.4 State diagram for the popcount FSM. . . . 126
9.5 Datapath for the popcount circuit. . . . 126
9.6 The ready/valid flow control. . . . 130
9.7 Data transfer with a ready/valid interface, early ready. . . . 131
9.8 Data transfer with a ready/valid interface, late ready. . . . 131
9.9 Single cycle ready/valid and back-to-back transfers. . . . 132

11.1 A writer, a FIFO buffer, and a reader. . . . 161
11.2 One byte transmitted by a UART. . . . 165

12.1 A classic computer consisting of a processor (CPU), memory, and I/O; connected via address, data, and control buses. 188
12.2 The translation of the off-chip bus concept to an on-chip "bus". . . . 189
12.3 A read transaction with a combinational acknowledge. 190
12.4 Read transaction with a pipelined acknowledgement. 191
12.5 Wishbone asynchronous read followed by an asynchronous write. . 199
12.6 Wishbone synchronous read followed by a synchronous write. . . . 199

14.1 The Leros datapath. 222

15.1 The 3-stage Wildcat processor pipeline (simplified, omitting control and decoded signals). 241

List of Tables

2.1 Chisel defined hardware operators. 15
2.2 Chisel defined hardware functions, invoked on v. 15

3.1 Chisel and highest supported Scala and Java versions. 37

5.1 Truth table for a 2 to 4 decoder. 64
5.2 Truth table for a 4 to 2 encoder. 66

8.1 State table for the alarm FSM. 110

12.1 An example address mapping. 194
12.2 Address mapping for the UART. 195
12.3 Status flags. 195

14.1 Leros instruction set. 218

B.1 Reserved keywords from the Scala language. 273
B.2 Reserved keywords from the Chisel language. 273

Listings

1.1 A hardware Hello World in Chisel 5

4.1 The adder component in Chisel. 50
4.2 The register component in Chisel. 51
4.3 A counter built out of components. 52
4.4 Definition of components `CompA` and `CompB` 54
4.5 Component `CompC` 55
4.6 Component `CompD` 56
4.7 Top-level component 57

6.1 A one-shot timer 84
6.2 1 KiB of synchronous memory. 90
6.3 A memory with a forwarding circuit. 92
6.4 Memory initialization. 93

7.1 Summarizing input processing with functions. 103

8.1 The Chisel code for the alarm FSM. 109
8.2 Rising edge detection with a Mealy FSM. 114
8.3 Rising edge detection with a Moore FSM. 116

9.1 Master FSM of the light flasher. 121
9.2 The down counter FSM. 123
9.3 The master FSM of the double refactored light flasher. 123
9.4 The top level of the popcount circuit. 127
9.5 Datapath of the popcount circuit. 128
9.6 The FSM of the popcount circuit. 129
9.7 A register as a buffer with a ready/valid interface 134

10.1 Binary to binary-coded decimal conversion. 140
10.2 Reading a text file to generate a logic table. 141

10.3 A register file with an optional debug port. . . . 149
10.4 Base class for our ticker implementations. . . . 150
10.5 Tick generation with a counter. . . . 150
10.6 A tester for different versions of the ticker. . . . 151
10.7 Tick generation with a down counter. . . . 152
10.8 Tick generation by counting down to -1. . . . 153
10.9 ChiselTest for the ticker tests. . . . 153
10.10 Minimum search including the index. . . . 155
10.11 A simple 2 to 1 arbiter. . . . 158
10.12 A fair 2-to-1 arbiter. . . . 159

11.1 A single stage of the bubble FIFO. . . . 163
11.2 A FIFO comprises an array of FIFO bubble stages. . . . 164
11.3 A transmitter for a serial port. . . . 166
11.4 A single-byte buffer with a ready/valid interface. . . . 168
11.5 A transmitter with an additional buffer. . . . 169
11.6 A receiver for a serial port. . . . 170
11.7 Sending “Hello World!” via the serial port. . . . 171
11.8 Echoing data on the serial port. . . . 171
11.9 Abstract class for FIFO veriations. . . . 172
11.10 A bubble FIFO with a ready/valid interface. . . . 172
11.11 A FIFO with double buffer elements. . . . 175
11.12 A FIFO with a register based memory. . . . 178
11.13 A FIFO with a on-chip memory. . . . 181
11.14 Combining a memory based FIFO with double-buffer stage. . . . 182
11.15 A multi-clock memory generator. . . . 183

12.1 An IO device consisting of four loadable counters. . . . 193
12.2 An IO device for a ready/valid device. . . . 197

13.1 Testing the counter device. . . . 206
13.2 Testing the counter device with fuctions. . . . 207
13.3 The tick generater as DUT. . . . 209
13.4 A top-level wrapper for our DUT. . . . 210
13.5 Testing the DUT with access to internal signals. . . . 211
13.6 Using assertions in Chisel. . . . 214
13.7 Formally verifying the circuit. . . . 215

14.1 Leros instruction encoding. . . . 220
14.2 The Leros ALU with the accumulator register. . . . 222
14.3 The Leros ALU function written in Scala. . . . 225
14.4 The main part of the Leros assembler. . . . 232
14.5 The instruction memory of Leros. . . . 233
14.6 The Leros state machine. . . . 233
14.7 The data memory module of Leros. . . . 235

15.1 Top level module of Wildcat. . . . 241
15.2 PC generation and instruction fetch. . . . 242
15.3 A ROM as a simple instruction memory. . . . 243
15.4 Decode stage. . . . 243
15.5 The core of the register file function. . . . 244
15.6 The core of the register file function. . . . 246
15.7 Address computation, including forwarding if needed. . . . 247
15.8 Memory access in decode. . . . 247
15.9 ALU operation in the execution stage. . . . 248
15.10 The ALU operation enumeration. . . . 248
15.11 The ALU function. . . . 249
15.12 Branch execution 250
15.13 Memory load. . . . 250

A.1 A simple component in Chisel. . . . 260
A.2 A simple component in VHDL. . . . 260
A.3 A simple component in Verilog. . . . 261
A.4 Using the `ChiselAdder` component in Chisel. . . . 262
A.5 Using the `adder` component in VHDL. . . . 262
A.6 Using the `adder` component in Verilog. . . . 262
A.7 A register with reset and enable in Chisel. . . . 263
A.8 A register with reset and enable in VHDL. . . . 263
A.9 A register with reset and enable in Verilog. . . . 264
A.10 A `switch` statement in Chisel. . . . 265
A.11 A `case` statement in VHDL. . . . 265
A.12 A `case` statement in Verilog. . . . 266
A.13 An “if...else if...else” statement in Chisel. . . . 267
A.14 An “if...else if...else” statement in VHDL. . . . 267
A.15 An “if...else if...else” statement in Verilog. . . . 268

Foreword

It is an exciting time to be in the world of digital design. With the end of Dennard Scaling and the slowing of Moore's Law, there has perhaps never been a greater need for innovation in the field. Semiconductor companies continue to squeeze out every drop of performance they can, but the cost of these improvements has been rising drastically. Chisel reduces this cost by improving productivity. If designers can build more in less time, while amortizing the cost of verification through reuse, companies can spend less on Non-Recurring Engineering (NRE). In addition, both students and individual contributors can innovate more easily on their own.

Chisel is unlike most languages in that it is embedded in another programming language, Scala. Fundamentally, Chisel is a library of classes and functions representing the primitives necessary to express synchronous, digital circuits. A Chisel design is really a Scala program that *generates* a circuit as it executes. To many, this may seem counterintuitive: "Why not just make Chisel a stand-alone language like VHDL or SystemVerilog?" My answer to this question is as follows: the software world has seen a substantial amount of innovation in design methodology in the past couple of decades. Rather than attempting to adapt these techniques to a new hardware language, we can simply *use* a modern programming language and gain those benefits for free.

A longstanding criticism of Chisel is that it is "difficult to learn." Much of this perception is due to the prevalence of large, complex designs created by experts to solve their own research or commercial needs. When learning a popular language like C++, one does not start by reading the source code of GCC. Rather, there are a plethora of courses, textbooks, and other learning materials that cater toward newcomers. In *Digital Design with Chisel*, Martin has created an important resource for anyone who wishes to learn Chisel.

Martin is an experienced educator, and it shows in the organization of this book. Starting with installation and primitives, he builds the reader's understanding like a building, brick-by-brick. The included exercises are the mortar that solidifies understanding, ensuring that each concept sets in the reader's mind. The book culminates with *hardware generators* like a roof giving the rest of the structure purpose. At

the end, the reader is left with the knowledge to build a simple, yet useful design: a RISC processor.

In *Digital Design with Chisel*, Martin has laid a strong foundation for productive digital design. What you build with it is up to you.

Jack Koenig
Chisel and FIRRTL Maintainer
Staff Engineer, SiFive

Preface

This book is an introduction to digital design with a focus on using the hardware construction language Chisel. Chisel brings advances from software engineering, such as object-oriented and functional languages, into digital design.

This book addresses hardware designers and software engineers. Hardware designers with knowledge of Verilog or VHDL can upgrade their productivity with a modern language for their next ASIC or FPGA design. Software engineers with knowledge of object-oriented and functional programming can leverage their knowledge to program hardware, for example, FPGA accelerators executing in the cloud.

The approach of this book is to present small to medium-sized typical hardware components to explore digital design with Chisel.

I have not used any large language model (LLM) to write even a single sentence. All mistakes are mine, not a hallucination by an LLM. I use Grammarly for grammar checking and Copilot for writing code.

Foreword for the Second Edition

As Chisel allows agile hardware design, so does open access and on-demand printing, which allows agile textbook publishing. Less than 6 months after the first edition of this book I can provide an improved and extended second edition.

Besides minor fixes, the main changes in the second edition are as follows. The testing section has been extended. The sequential building blocks chapter contains more example circuits. A new chapter on input processing explains input synchronization, shows how to design a debouncing circuit, and how to filter a noisy input signal. The example designs chapter has been extended to show different implementations of a FIFO. The FIFO variations also show how to use type parameters and inheritance in digital design.

Foreword for the Third Edition

Chisel has been moving forward in the last year, so it is time for a new edition of the Chisel book. We changed all examples to the latest version of Chisel (3.5.3), and the recommended Scala version (2.12.13).

With Chisel 3.5 the testing environment `PeekPokeTester` as part of the `iotesters` package has been deprecated. Therefore, we have changed the testing description to the new ChiselTest framework. As there are still many Chisel designs available that use the `PeekPokeTester`, we have moved the description for it into the appendix.

One of the fascinating aspects of the Chisel/Scala/Java environment is that we can piggyback on the available infrastructure to distribute open-source libraries. We can publish hardware components on Maven as simply as any other open-source Java library. Publishing on Maven means that a 3rd party component can be integrated into the compile flow with a single reference in the `build.sbt` configuration. This is the same process as how you include the chisel library for your design. We have added a section on how to publish a Chisel design on Maven Central.

We have improved the explanation of components with a simpler example.

Hardware *generators* are written in Scala. Therefore, we have added a short section on Scala. We have extended the hardware generator chapter with a section on using functional programming to write generators.

The appendix has been extended with a list of reserved keywords and a list of acronyms.

Hans Jakob Damsgaard has contributed the description of how to use external components through Chisel's `BlackBox` and how to use memories for clock domain crossing (multi-clock memories).

Foreword for the Fourth Edition

For the fourth edition, we have switched to the actual Chisel version 3.5.4. We have added an arbiter, priority encoder, and comparator to the chapter on combinational building blocks. We have extended the hardware generation chapter with more functional examples, including building a fair arbitration tree out of a simple 2-to-1 arbitration circuits. We have added a new chapter on interconnect, bus interfaces, and how to connect an IO device as a memory-mapped device. We have started a new chapter on debugging, testing, and verification. The plan is to extend the chapter on this important topic in the next edition. We have extended the processor chapter with a more gentle introduction to a microprocessor, including a figure

of the datapath.

Foreword for the Fifth Edition

For the fifth edition, we have upgraded to the actual Chisel version 3.5.6 and Scala 2.13. To make room for more advanced topics, I have removed the two appendices on Chisel 2 and the PeekPokeTester. All projects that are actively maintained have finally moved to Chisel 3, at least with the compatibility layer. The PeekPokeTester has been deprecated with Chisel 3.5, and it is recommended to switch to ChiselTest. If needed, those two chapters are available in the older versions of the Chisel book, available as PDF at the web page for this book.

The fifth edition does not include major changes; it is a consolidation version. We did considerable proofreading for more clarity in writing. Chisel has added small convenience features (e.g., `ChiselEnum`) that we cover in this edition. We extended the processor chapter and updated the Leros code snippets to the actual version of Leros.

Foreword for the Sixth Edition

Chisel has recently gone through some major changes. Up to Chisel 3.x, Chisel is mainly based on Scala, which means it runs on the Java virtual machine and is platform independent. After 3.x, the architecture of the Chisel compile pipeline has changed. To indicate that change, the version numbering system has changed to increment the major version number more often. Version 4.0 is skipped, and the next Chisel version is 5.0. From version 5.0, the compiler backend is based on CIRCT, a circuit compiler, which is part of the LLVM project. The main difference for the Chisel user is that CIRCT (and LLVM) are written in C++ and therefore need a different executable for each operating system (processor architecture).

We tested the book with Chisel Versions 3.5.6, 3.6.1, 5.3.0, and 6.5.0. Chisel 3.6 is the last version that uses the Scala-based backend to generate Verilog files. Chisel 5 is using CIRCT and needs `firtool` installed manually. The latest version of Chisel 6 includes the `firtool` binaries. Testing with ChiselTest is still supported until Chisel 6. Therefore, we recommend that Chisel 5 be skipped and Chisel 6 be used. Chisel 6 might switch to a different simulation and testing library. We used Scala 2.13 and tested it with Java 17 and Java 21. Chisel 3.5.6 does not support Java 21. Therefore, we recommend using Chisel 3.6.1, if one wants to work with Chisel

3.

TODO: Some of this might go into getting started.

TODO: We should also show more Leros code.

We extended the testing chapter with a description of how to access internal signals and how to use assertions. Furthermore, we started to explore formal verification in Chisel.

Organizing digital circuits as pipelines to increase performance is a key aspect of digital design and computer architecture. We added a chapter on pipelining where we use a 3-stage RISC-V pipeline as a running example.

We are aware that Chisel is still a niche language in hardware design, and one needs to at least be able to read VHDL and/or Verilog. However, we think that with a good knowledge of digital design, switching to a different hardware description language should be a matter of days. To simplify this transition, we have added an appendix showing and comparing basic constructs in Chisel, VHDL, and Verilog.

This book was written without any help from artificial intelligence or large language models to write or rewrite text. We used Grammarly[1] for grammar checking and improvement of the writing style.

Foreword for the Seventh Edition

Translations

This book has been translated into Chinese, Japanese, and Vietnamese. All translations are available as free PDF from the books web page. I would like to thank Yuda Wang, Qiwei Sun, and Yun Chen for the Chinese translation; Seiji Munetoh, Masatoshi Tanabata, and Takaaki Hagino for the Japanese translation; and VieLe Duc Hung for the Vietnamese translation. If you are interested in translating this book into another language, feel free to do it and publish it under the same license. Please contact me then, so I can point to your translation.

Acknowledgements

I want to thank everyone who has worked on Chisel for creating such a cool hardware construction language. Chisel is so joyful to use and therefore worth writing a

[1] https://app.grammarly.com/

book about. I am thankful to the whole Chisel community, which is so welcoming and friendly and never tired of answering questions on Chisel.

I would also like to thank my students in the last years of an advanced computer architecture course where most of them picked up Chisel for the final project. Thank you for moving out of your comfort zone and taking up the journey of learning and using a bleeding-edge hardware description language. Many of your questions have helped to shape this book.

It was a pleasure to use Chisel in teaching digital electronics at the Technical University of Denmark since 2020. I know it is a challenge to pickup Chisel and Java in parallel in the second semester. Thank you to all students from this course, who had on open mind to pickup a modern programming language for hardware description.

For the third edition, I would like to acknowledge Hans Jakob Damsgaard (@hansemandse) for rewriting all test code of this book to ChiselTest, adding ChiselTest to the testing chapter, adding the black box description, and an example of a multi-clock memory.

1 Introduction

This book is an introduction to digital system design using a modern hardware construction language, Chisel [5]. In this book, we focus on a higher abstraction level than usual in digital design books, to enable you to build more complex, interacting digital systems in a shorter time.

This book and Chisel are targeting two groups of developers: (1) hardware designers and (2) software programmers. Hardware designers who are fluent in VHDL or Verilog and use other languages, such as Python, Java, or Tcl to generate hardware can move to a single hardware construction language where hardware generation is part of the language. Software programmers may become interested in hardware design, e.g., as future chips from Intel will include programmable hardware to speed up programs. It is perfectly fine to use Chisel as your first hardware description language.

Chisel brings advances in software engineering, such as object-oriented and functional programming in the digital design domain. Chisel not only allows you to express hardware at the register-transfer level but also allows you to write hardware *generators*.

Hardware is now commonly described with a hardware description language. The time of drawing hardware components, even with CAD tools, is over. Some high-level schematics can give an overview of the system but are not intended to describe the system. The two most common hardware description languages are Verilog and VHDL. Both languages are old, contain many legacies, and have a moving line of what constructs of the language are synthesizable to hardware. Do not get me wrong: VHDL and Verilog are perfectly able to describe a hardware block that can be synthesized into an ASIC. For hardware design in Chisel, Verilog serves as an intermediate language for testing and synthesis.

This book is not a general introduction to digital design and the fundamentals of it. For an introduction to the basics of digital design, such as how to build a gate out of CMOS transistors refer to other digital design books. However, this book intends to teach digital design at an abstraction level that is current practice

to describe ASICs or designs targeting FPGAs.[1] As prerequisites for this book, we assume basic knowledge of Boolean algebra and the binary number system. Furthermore, some programming experience in any programming language is assumed. No knowledge of Verilog or VHDL is needed. Chisel can be your first programming language to describe digital hardware. As the build process of the examples is based on `sbt` and `make`, basic knowledge of the command-line interface (CLI, also called terminal or Unix shell) will be helpful.

Chisel itself is not a big language. The basic constructs fit on one page and can be learned within a few days. Therefore, this book is not a big book, as well. Chisel is for sure smaller than VHDL and Verilog, which carry many legacies. The power of Chisel comes from the embedding of Chisel within Scala, which itself is an expressive language. Chisel inherits the feature from Scala of being "a language that grows on you" [29]. However, Scala is not the topic of this book. We provide a short section on Scala for hardware designers. The textbook by Odersky et al. [29] provides a general introduction to Scala. This book is a tutorial in digital design and the Chisel language; it is not a Chisel language reference, nor is it a book on complete chip design.

All code examples in this book are extracted from complete programs compiled and tested in continuous integration at GitHub. Therefore, the code shall not contain any syntax errors. The code examples are available from the GitHub repository of this book. Besides showing Chisel code, we have also tried to show useful designs and principles of good hardware description style.

This book is optimized for reading on a tablet (e.g., an iPad) or a laptop. We include links to further reading in the running text, primarily to Wikipedia articles.

1.1 Installing Chisel and FPGA Tools

Chisel is a Scala library, and the easiest way to install Chisel and Scala is with `sbt`, the Scala build tool. Scala itself depends on the installation of Java JDK 8 (or a later version up to Java 21). As Oracle has changed the license for Java, it may be easier to install OpenJDK from AdoptOpenJDK or use SDKMAN to manage your Java installation.

More detailed setup instructions can be found in Setup.md from the chisel-lab. The first lab explains how to open an existing Chisel project in IntelliJ.

[1] As the author is more familiar with FPGAs than ASICs as target technology, some design optimizations shown in this book are targeting FPGA technology.

1.1.1 macOS

Install the Java OpenJDK 8 (up to 21) from AdoptOpenJDK. On Mac OS X, with the packet manager Homebrew, `sbt` and git can be installed with:

```
$ brew install sbt git
```

Install GTKWave and IntelliJ (the community edition). When importing a project, select the JDK you installed before.

1.1.2 Linux/Ubuntu

Install Java and useful tools in Ubuntu with:

```
$ sudo apt install openjdk-8-jdk git make gtkwave
```

For Ubuntu, which is based on Debian, programs are usually installed from a Debian file (.deb). However, as of the time of this writing, `sbt` is not available as a ready to install package. Therefore, the installation process is a little bit more involved. Follow the instructions from sbt download

1.1.3 Windows

Install the Java OpenJDK (8 or up to 21) from AdoptOpenJDK. Chisel and Scala can also be installed and used under Windows. Install GTKWave and IntelliJ (the community edition). When importing a project, select the JDK you installed before. `sbt` can be installed with a Windows installer, see: Installing sbt on Windows. Install a git client.

1.1.4 FPGA Tools

To build hardware for an FPGA, you need a synthesize tool. The two major FPGA vendors, Altera, an Intel Company[2] and AMD,[3] provide free versions of their tools that cover small to medium-sized FPGAs. Those medium-sized FPGAs are large enough to build a multicore RISC style processor. Intel provides the Quartus Prime Lite Edition and AMD the Vivado Design Suite, WebPACK Edition. Both tools are available for Windows and Linux, but not for macOS.

With F4PGA it is now possible to use a fully open-source synthesis tool for selected FPGAs.

[2] former Intel and former Altera
[3] former Xilinx

1.2 Hello World

Each book on a programming language shall start with a minimal example, called the *Hello World* example. The following code is a first approach:

```
object HelloScala extends App {
  println("Hello Chisel World!")
}
```

Compiling and executing this short program with `sbt`

```
$ sbt run
```

leads to the expected output of a Hello World program:

```
[info] Running HelloScala
Hello Chisel World!
```

However, is this Chisel? Is this hardware generated to print a string? No, this is plain Scala code and not a representative Hello World program for a hardware design.

1.3 Chisel Hello World

What is then the equivalent of a Hello World program for a hardware design? The minimal useful and visible design? A blinking LED is the hardware (or even embedded software) version of Hello World. If a LED blinks, we are ready to solve bigger problems!

Listing 1.1 shows a blinking LED, described in Chisel. It is not important that you understand the details of this code example. We will cover those in the following chapters. Just note that the circuit is usually clocked with a high frequency, e.g., 50 MHz, and we need a counter to derive timing in the Hz range to achieve a visible blinking. In the above example, we count from 0 up to 25000000-1 and then toggle the blinking signal (`blkReg := ~blkReg`) and restart the counter (`cntReg := 0.U`). That hardware then blinks the LED at 1 Hz.

1.4 An IDE for Chisel

This book makes no assumptions about your programming environment or editor you use. Learning the basics should be easy with just using `sbt` at the command

```
class Hello extends Module {
  val io = IO(new Bundle {
    val led = Output(UInt(1.W))
  })
  val CNT_MAX = (50000000 / 2 - 1).U

  val cntReg = RegInit(0.U(32.W))
  val blkReg = RegInit(0.U(1.W))

  cntReg := cntReg + 1.U
  when(cntReg === CNT_MAX) {
    cntReg := 0.U
    blkReg := ~blkReg
  }
  io.led := blkReg
}
```

Listing 1.1: A hardware Hello World in Chisel

line and an editor of your choice. In the tradition of other books, all commands that you shall type in a shell/terminal/CLI are preceded by a `$` character, which you shall not type in. As an example, here is the Unix `ls` command, which lists files in the current folder:

```
$ ls
```

That said, an integrated development environment (IDE), where a compiler is running in the background, can speed up coding. As Chisel is a Scala library, all IDEs that support Scala are also good IDEs for Chisel. It is possible in IntelliJ, Visual Studio Code, and Eclipse to generate a project from the sbt project configuration in `build.sbt`.

In IntelliJ you need to install the Scala plugin. Then you can create a new project from existing sources with: *File - New - Project from Existing Sources...* and then select the `build.sbt` file from the project.

In Eclipse you can create a project via

```
$ sbt eclipse
```

and import that project into Eclipse.[4]

Visual Studio Code is another option for a Chisel IDE. The Scala Metals extension provides Scala support. On the left bar select *Extensions* and search for *Metals* and install *Scala (Metals)*. To import an `sbt`-based project, open the folder with *File - Open*.

1.5 Source Access and eBook Features

This book is open source and hosted at GitHub: schoeberl/chisel-book. All Chisel code examples shown in this book are included in the repository. All code shown in the book passed the compiler and therefore should not contain any syntax errors. Furthermore, most examples also include a test bench. The code is extracted automatically from that source. We collect larger Chisel examples in the accompanying repository chisel-examples and in ip-contributions.

If you find an error or typo in the book, a GitHub pull request is the most convenient way to incorporate your improvement. You can also provide feedback or comments for improvements by filing an issue on GitHub or sending a plain, old school email.

The repository of the book also contains slides in Latex that I use for a 13 week course on Digital Electronics[5] at the Technical University of Denmark. That course also contains lab exercises in Chisel. If you are teaching digital design with Chisel, feel free to adapt the slides and lab exercises to your needs. All material is open-source. To build the book and slides you need a recent version of LaTeX and the needed tools for Chisel (`sbt` and a Java JDK installation). All code is compiled, tested, extracted, and the LaTeX compiled with a simple:

```
$ make
```

This book is freely available as a PDF eBook and in a classical printed version from Amazon. The eBook version features links to further resources and Wikipedia entries. We use Wikipedia entries for background information (e.g., binary number system) that does not directly fit into this book. We optimized the format of the eBook for reading on a tablet, such as an iPad.

[4] This function needs the Eclipse plugin for sbt.
[5] The course page contains the PDF versions of the slides

1.6 Further Reading

Here is a list of further reading for digital design and Chisel:

- Digital Design: A Systems Approach, by William J. Dally and R. Curtis Harting, is a textbook on digital design. It is available in two versions: using Verilog or VHDL as a hardware description language.

The official Chisel documentation and further documents are available online:

- The Chisel home page is the official starting point to download and learn Chisel.
- The website of the Digital Electronics 2 course at the Technical University of Denmark contains the slides for a 13 weeks course, based on Chisel. The source code for the slides is available as part of the source code for this book. Feel free to adapt them for your teaching needs.
- The schoeberl/chisel-lab GitHub repo contains Chisel exercises for the course Digital Electronics 2. The exercises also fit well for a selfstudy with this book.
- The empty Chisel project is a good starting point with a very minimal hardware (an adder) and a test. That project is a GitHub template where you can base your GitHub repository on.
- The Chisel3 Cheat Sheet summarizes the main constructs of Chisel on a single page.
- Scott Beamer's course Agile Hardware Design contains advanced Chisel examples. The lectures include executable source examples and are available as Jupyter notebooks.
- ChiselTest is in its own repository.
- The Generator Bootcamp is a Chisel course focusing on hardware generators, as a Jupyter notebook
- The Chisel Tutorial provides a ready setup project containing small exercises with testers and solutions. However, it is a bit outdated.
- A Chisel Style Guide by Christopher Celio.

1.7 Exercises

Each chapter ends with hands-on exercises. For the introduction exercise, we will use an FPGA board to get one LED blinking.[6] As a first step, clone (or fork) the chisel-examples repository from GitHub. The Hello World example is in the folder `hello-world`, set up as a minimal project. You can explore the Chisel code of the blinking LED in `src/main/scala/Hello.scala`. Compile the blinking LED with the following steps:

```
$ git clone https://github.com/schoeberl/chisel-examples.git
$ cd chisel-examples/hello-world/
$ sbt run
```

After some initial downloading of Chisel components, this will produce the Verilog file `Hello.v`. Explore this Verilog file. You will see that it contains two inputs `clock` and `reset` and one output `io_led`. When you compare this Verilog file with the Chisel module, you will notice that the Chisel module does not contain `clock` or `reset`. Those signals are implicitly generated, and in most designs, it is convenient not to need to deal with these low-level details. Chisel provides register components, and those are connected automatically to `clock` and `reset` (if needed).

The next step is to set up an FPGA project file for the synthesis tool, assign the pins, compile[7] the Verilog code, and configure the FPGA with the resulting bitfile. We cannot provide the details of these steps. Please consult the manual of your Intel Quartus or AMD Vivado tool. However, the examples repository contains some ready to use Quartus projects in folder `quartus` for several popular FPGA boards (e.g., DE2-115). If the repository contains support for your board, start Quartus, open the project, compile it by pressing the *Play* button, and configure the FPGA board with the *Programmer* button and one of the LEDs should blink.

Congratulations! You managed to get your first design in Chisel running in an FPGA!

If the LED is not blinking, check the status of reset. On the DE2-115 configuration, the reset input is connected to SW0.

Now change the blinking frequency to a slower or a faster value and rerun the build process. Blinking frequencies and also blinking patterns communicate differ-

[6] If you at the moment have no FPGA board available, continue to read as we will show you a simulation version at the end of the exercise.

[7] The real process is more elaborate with following steps: synthesizing the logic, performing place and route, performing timing analysis, and generating a bitfile. However, for the purpose of this introduction example we simply call it "compile" your code.

ent “emotions”. For example, a slow blinking LED signals that everything is ok, a fast blinking LED signals an alarm state. Explore which frequencies express best those two different emotions.

As a more challenging extension to the exercise, generate the following blinking pattern: the LED shall be on for 200 ms every second. For this pattern, you might decouple the change of the LED blinking from the counter reset. You will need a second constant where you change the state of the `blkReg` register. What kind of emotion does this pattern produce? Is it alarming or more like a sign-of-live signal?

If you do not (yet) have an FPGA board, you can still run the blinking LED example. You will use the Chisel simulation. To avoid a too-long simulation time, change the clock frequency in the Chisel code from 50000000 to 50000. Execute following instruction to simulate the blinking LED:

```
$ sbt test
```

This will execute the tester that runs for one million clock cycles. The blinking frequency depends on the simulation speed, which depends on the speed of your computer. Therefore, you might need to experiment a little bit with the assumed clock frequency to see the simulated blinking LED.

2 Basic Components

In this section, we introduce the basic components for digital design: combinational circuits and flip-flops. These essential elements can be combined to build larger, more interesting circuits.

Digital systems, in general, use binary signals, which means a single bit or signal can only have one of two possible values. These values are often called 0 and 1. However, we also use the following terms: low/high, false/true, and deasserted/asserted. These terms mean the same two possible values of a binary signal.

2.1 Chisel Types and Constants

Chisel provides three data types to describe connections, combinational logic, and registers: `Bits`, `UInt`, and `SInt`. `UInt` and `SInt` extend `Bits`, and all three types represent a vector of bits. `UInt` gives this vector of bits the meaning of an unsigned integer and `SInt` of a signed integer.[1] Chisel uses two's complement as signed integer representation. Here is the definition for different types, an 8-bit `Bits`, an 8-bit unsigned integer, and a 10-bit signed integer:

```
Bits(8.W)
UInt(8.W)
SInt(10.W)
```

The width of a vector of bits is defined by a Chisel width type (`Width`). The following expression casts the Scala integer `n` to a Chisel `width`, which we use for the definition of the `Bits` vector:

```
n.W
Bits(n.W)
```

Constants can be defined by using a Scala integer and converting it to a Chisel type:

[1] The type `Bits` in the current version of Chisel is missing operations and therefore not very useful for user code.

```
0.U  // defines a UInt constant of 0
-3.S // defines a SInt constant of -3
```

Constants can also be defined with a width, by using the Chisel width type:

```
3.U(4.W) // An 4-bit constant of 3
```

If you find the notation of `3.U` and `4.W` a little bit funny, consider it as a variant of an integer constant with a type. For example, `3L`, representing a long integer constant in C, Java, and Scala.

Possible pitfall: One possible error when defining constants with a dedicated width is missing the `.W` specifier for a width. E.g., `1.U(32)` will *not* define a 32-bit wide constant representing 1. Instead, the expression `(32)` is interpreted as bit extraction from position 32, which results in a single bit constant of 0. Probably not what the original intention of the programmer was.

Chisel benefits from Scala's type inference, and in many places, type information can be left out. The same is also valid for bit widths. In many cases, Chisel will automatically infer the correct width. Therefore, a Chisel description of hardware is more concise and better readable than VHDL or Verilog.

For constants defined in other bases than decimal, the constant is defined in a string with a preceding `h` for hexadecimal (base 16), `o` for octal (base 8), and `b` for binary (base 2). The following example shows the definition of constant 255 in different bases. In this example we omit the bit width and Chisel infers the minimum width to fit the constants in, in this case 8 bits.

```
"hff".U        // hexadecimal representation of 255
"o377".U       // octal representation of 255
"b1111_1111".U // binary representation of 255
```

The above code shows how to use an underscore to group digits in the string that represents a constant. The underscore is ignored.

Characters to represent text (in ASCII encoding) can also be used as constants in Chisel:

```
val aChar = 'A'.U
```

To represent logic values, Chisel defines the type `Bool`. `Bool` can represent a *true* or *false* value. The following code shows the definition of type `Bool` and the definition of `Bool` constants, by converting the Scala Boolean constants `true` and `false` to Chisel `Bool` constants.

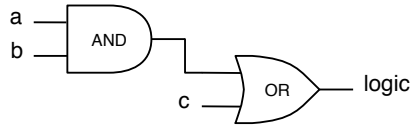


Figure 2.1: Logic for the expression `(a & b)| c`. The wires can be a single bit or multiple bits. The Chisel expression, and the schematics are the same.

```
Bool()
true.B
false.B
```

2.2 Combinational Circuits

Chisel uses Boolean algebra operators, as they are defined in C, Java, Scala, and several other programming languages, to described combinational circuits: `&` is the AND operator and `|` is the OR operator. Following line of code defines a circuit that combines signals `a` and `b` with *and* gates and combines the result with signal `c` with *or* gates and names it `logic`.

```
val logic = (a & b) | c
```

Figure 2.1 shows the schematic of this combinatorial expression. Note that this circuit may be for a vector of bits and not only single wires combined with the AND and OR circuits.

In this example, we do not define the type nor the width of signal `logic`. Both are inferred from the type and width of the expression. The standard logic operations in Chisel are:

```
val a_and_b = a & b // bitwise and of a and b
val a_or_b = a | b  // bitwise or of a and b
val a_xor_b = a ^ b // bitwise xor of a and b
val a_not = ~a      // bitwise negation of a
```

The arithmetic operations use the standard operators:

```
val a_plus_b = a + b // addition of a and b
val a_minus_b = a - b // subtraction of b from a
```

```
val neg_a = -a     // negate a
val a_mul_b = a * b // multiplication of a and b
val a_div_b = a / b // division of a by b
val a_mod_b = a % b // modulo operation of a by b
```

The resulting width of the operation is the maximum width of the operators for addition and subtraction, the sum of the two widths for the multiplication, and usually the width of the numerator for divide and modulo operations.[2]

A signal can also first be defined as a `Wire` of some type. Afterward, we can assign a value to the wire with the `:=` update operator.

```
val w = Wire(UInt())

w := a & b
```

A single bit can be extracted as follows:

```
val sign = x(31)
```

A subfield can be extracted from end to start position:

```
val lowByte = largeWord(7, 0)
```

Bit fields are concatenated with the `##` operator.[3]

```
val word = highByte ## lowByte
```

Table 2.1 shows the full list of operators (see also builtin operators). The Chisel operator precedence is determined by the evaluation order of the circuit, which follows the Scala operator precedence. If in doubt, it is always a good practice to use parentheses.[4]

Table 2.2 shows various functions defined on and for Chisel data types.

[2] The exact details are available in the FIRRTL specification.

[3] Note that there is a `Cat` function available that performs the same operation with `Cat(highByte, lowByte)`.

[4] The operator precedence in Chisel is a side effect of the hardware elaboration when the tree of hardware nodes is created by executing the Scala operators. The Scala operator precedence is similar but not identical to Java/C. Verilog has the same operator precedence as C, but VHDL has a different one. Verilog has precedence ordering for logic operations, but in VHDL those operators have the same precedence and are evaluated from left to right.

Operator	Description	Data types
`* / %`	multiplication, division, modulus	UInt, SInt
`+ -`	addition, subtraction	UInt, SInt
`=== =/=`	equal, not equal	UInt, SInt, returns Bool
`> >= < <=`	comparison	UInt, SInt, returns Bool
`<< >>`	shift left, shift right (sign extend on SInt)	UInt, SInt
`~`	NOT	UInt, SInt, Bool
`& | ^`	AND, OR, XOR	UInt, SInt, Bool
`!`	logical NOT	Bool
`&& ||`	logical AND, OR	Bool

Table 2.1: Chisel defined hardware operators.

Function	Description	Data types
`v.andR v.orR v.xorR`	AND, OR, XOR reduction	UInt, SInt, returns Bool
`v(n)`	extraction of a single bit	UInt, SInt
`v(end, start)`	bitfield extraction	UInt, SInt
`Fill(n, v)`	bitstring replication, n times	UInt, SInt
`a ## b`	bitfield concatenation	UInt, SInt
`Cat(a, b, ...)`	bitfield concatenation	UInt, SInt

Table 2.2: Chisel defined hardware functions, invoked on `v`.

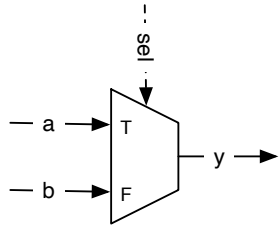


Figure 2.2: A basic 2:1 multiplexer.

2.2.1 Multiplexer

A multiplexer is a circuit that selects between alternatives. In the most basic form, it selects between two alternatives. Figure 2.2 shows such a 2:1 multiplexer, or mux for short. Depending on the value of the select signal (`sel`) signal `y` will represent signal `a` or signal `b`.

A multiplexer can be built from logic. However, as multiplexing is such a standard operation, Chisel provides a multiplexer,

```
val result = Mux(sel, a, b)
```

where `a` is selected when the `sel` is `true.B`, otherwise `b` is selected. The type of `sel` is a Chisel `Bool`; the inputs `a` and `b` can be any Chisel base type or aggregate (bundles or vectors) as long as they are the same type.

With logical and arithmetical operations and a multiplexer, every combinational circuit can be described. However, Chisel provides further components and control abstractions for a more elegant description of a combinational circuit, which are described in Chapter 5.

The second basic component needed to describe a digital circuit is a state element, also called register, which is described next.

2.3 Registers

Chisel provides a register, which is a collection of D flip-flops. The register is implicitly connected to a global clock and is updated on the rising edge. When an initialization value is provided at the declaration of the register, it uses a synchronous reset connected to a global reset signal. A register can be any Chisel type that can

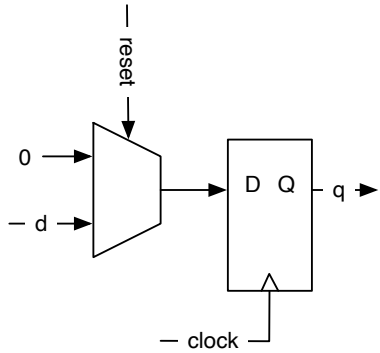


Figure 2.3: A D flip-flop based register with a synchronous reset to 0.

be represented as a collection of bits. Following code defines an 8-bit register, initialized with 0 at reset:

```
val reg = RegInit(0.U(8.W))
```

An input is connected to the register with the `:=` update operator and the output of the register can be used just with the name in an expression:

```
reg := d
val q = reg
```

A register can also be connected to its input at the definition:

```
val nextReg = RegNext(d)
```

Figure 2.3 shows the circuit of our register definition with a clock, a synchronous reset to `0.U`, input `d`, and output `q`. The global signals `clock` and `reset` are implicitly connected to each register defined.
A register can also be connected to its input and a constant as initial value at the definition:

```
val bothReg = RegNext(d, 0.U)
```

To distinguish between signals representing combinational logic and registers, a common practice is to postfix register names with `Reg`. Another common practice,

coming from Java and Scala, is to use CamelCase for identifiers consisting of several words. The convention is to start functions and variables with a lower case letter and classes (types), e.g., a `Module` name, with an upper case letter.

In Chisel you are relative free to name your identifiers. However, use taste and descriptive names. Furthermore, several words are reserved. They are listed in Appendix B.

2.3.1 Counting

Counting is a fundamental operation in digital systems. One might count events. However, more often counting is used to define a time interval. Counting the clock cycles and triggering an action when the time interval has expired.

A simple approach is counting up to a value. However, in computer science, and digital design, counting starts at 0. Therefore, if we want to count 10 clock cycles, we count from 0 to 9. The following code shows such a counter that counts till 9 and wraps around to 0 when reaching 9.

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === 9.U, 0.U, cntReg + 1.U)
```

2.4 Structure with Bundle and Vec

Chisel provides two constructs to group related signals: (1) a `Bundle` and (2) a `Vec`. A `Bundle` groups signals of different types as named fields. A `Vec` represents an indexable collection of signals (elements) of the same type. `Bundles` and `Vecs` create new, user-defined Chisel types and can be arbitrarily nested.

2.4.1 Bundle

A Chisel `Bundle` groups several signals. The entire bundle can be referenced as a whole, or individual fields can be accessed by their name. A `Bundle` is similar to a `struct` in C and SystemVerilog or a `record` in VHDL. We can define a bundle (collection of signals) by defining a class that extends `Bundle` and list the fields as `vals` within the constructor block.

```
class Channel() extends Bundle {
```

```
  val data = UInt(32.W)
  val valid = Bool()
}
```

To use a bundle, we create it with `new` and wrap it into a `Wire`. The fields are accessed with the dot notation:

```
  val ch = Wire(new Channel())
  ch.data := 123.U
  ch.valid := true.B

  val b = ch.valid
```

Dot notation is common in object-oriented languages, where `x.y` means `x` is a reference to an object and `y` is a field of that object. As Chisel is object-oriented, we use dot notation to access fields in a bundle. A bundle can also be referenced as a whole:

```
  val channel = ch
```

2.4.2 Vec

A Chisel `Vec` (a vector) represents a collection of Chisel types of the same type. Each element can be accessed by an index. A Chisel `Vec` is similar to array data structures in other programming languages.[5]

A `Vec` is used for three different purposes: (1) dynamic addressing in hardware, which is a multiplexer; (2) a register file, which includes multiplexing the read and generating the enable signal for the write; (3) parametrization of the number of ports of a `Module`. For other collections of *things*, being it hardware elements or other generator data, it is better to use the Scala collection `Seq`.

Combinational Vec

A `Vec` is created by calling the constructor with two parameters: the number of elements and the type of the elements. A combinational `Vec` needs to be wrapped into a `Wire`

```
  val v = Wire(Vec(3, UInt(4.W)))
```

[5] The name `Array` is already used in Scala.

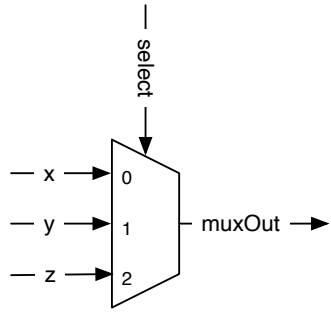


Figure 2.4: A vector wrapped in a `Wire` is just a multiplexer.

Individual elements are accessed with `(index)`. A vector wrapped into a `Wire` is just a multiplexer.

```
v(0) := 1.U
v(1) := 3.U
v(2) := 5.U

val index = 1.U(2.W)
val a = v(index)
```

Here is another example of using `Vec` as a multiplexer. The three inputs are connected to the three wires `x`, `y`, and `z`. The `select` wire selects which input is used and connects it to `muxOut`.

```
val m = Wire(Vec(3, UInt(8.W)))
m(0) := x
m(1) := y
m(2) := z
val muxOut = m(select)
```

Figure 2.4 shows the resulting schematic of the above code snippet.

Similar to using a `WireDefault`, we can set default values of a `Vec` with `VecInit`. The following code represents a 3:1 multiplexer with three constant defaults. Note that we specify the size (3 bits) of the `UInt` data types with the first constant. With the condition (`cond`) we can overwrite those default values. This overwrite hardware itself consists of three 2:1 multiplexers. The last line selects one of the three inputs

of the defVec multiplexer. Note that VecInit already returns Chisel hardware, so we do not need to wrap it in a Wire.[6]

```
val defVec = VecInit(1.U(3.W), 2.U, 3.U)
when (cond) {
  defVec(0) := 4.U
  defVec(1) := 5.U
  defVec(2) := 6.U
}
val vecOut = defVec(sel)
```

It is not only possible to set initial constants (like in WireDefault) for the Vec input, but we can also connect signals (wires) with VecInit to the inputs of the Vec. The following example connects the wires d, e, and f to the three inputs of the Vec.

```
val defVecSig = VecInit(d, e, f)
val vecOutSig = defVecSig(sel)
```

Register Vec

We can also wrap a Vec into a register to define an array of registers. The following code shows a vector of three registers.

```
val vReg = Reg(Vec(3, UInt(8.W)))

val dout = vReg(rdIdx)
vReg(wrIdx) := din
```

Figure 2.5 shows the schematic of that circuit. It contains three registers. The read index (rdIdx) selects the multiplexer connected to the output of the three registers. The output signal is dout The write index (wrIdx) selects which register will be written with the data from din. wrIdx is driving a decoder that selects one of the three enable signals of the registers.

Following example defines a register file for a processor; 32 registers each 32-bits wide, as used in a classic 32-bit RISC processor such as the 32-bit version of RISC-V.

```
val registerFile = Reg(Vec(32, UInt(32.W)))
```

[6] This is different from a plain Vec that needs to be wrapped into a Wire. We could wrap the VecInit into a WireDefault, but this uncommon coding style.

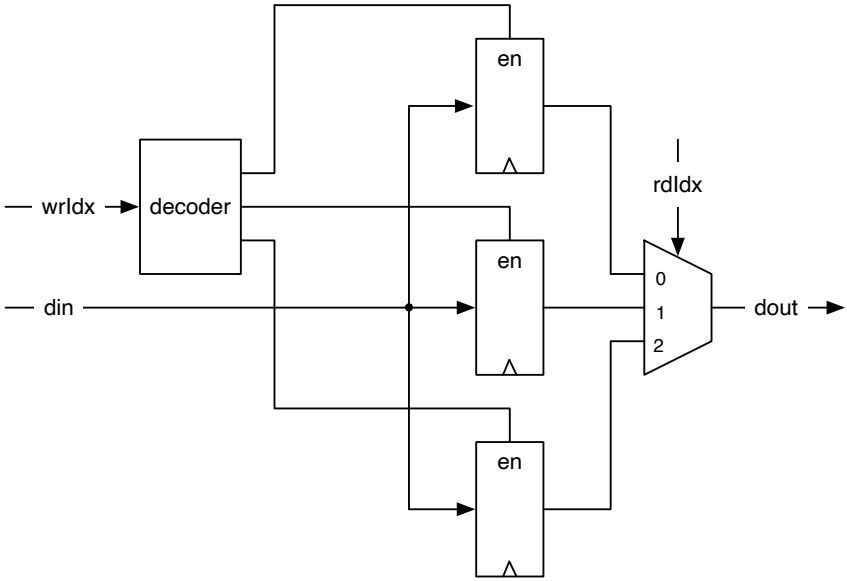


Figure 2.5: A vector of registers.

An element of that register file is accessed with an index and used as a normal register.

```
registerFile(index) := dIn
val dOut = registerFile(index)
```

A register of a vector can also be initialized. This is then the value that the register is reset to. To initialize the register file, we use A `VecInit` with the constants for reset, wrapped into a `RegInit`. The input of the three registers are then connected to wires `d`, `e`, and `f`.

```
val initReg = RegInit(VecInit(0.U(3.W), 1.U, 2.U))
val resetVal = initReg(sel)
initReg(0) := d
initReg(1) := e
initReg(2) := f
```

If we want to reset all elements of a large register file to the same value (probably 0), we can use a Scala sequence `Seq`. `VecInit` can be constructed with a sequence containing Chisel types. `Seq` contains a creation function `fill` to initialize a sequence with identical values. The following code constructs a register file containing 32 registers, each 32-bit wide and reset to 0:

```
val resetRegFile =
    RegInit(VecInit(Seq.fill(32)(0.U(32.W))))
val rdRegFile = resetRegFile(sel)
```

Combining Bundle and Vec

We can freely mix bundles and vectors. When creating a vector with a bundle type, we need to pass a prototype for the vector fields. Using our `Channel`, which we defined above, we can create a vector of channels with:

```
val vecBundle = Wire(Vec(8, new Channel()))
```

A bundle may as well contain a vector:

```
class BundleVec extends Bundle {
  val field = UInt(8.W)
  val vector = Vec(4,UInt(8.W))
}
```

When we want a register of a bundle type that needs a reset value, we first create a `Wire` of that bundle, set the individual fields as needed, and then pass this bundle to a `RegInit`:

```
val initVal = Wire(new Channel())

initVal.data := 0.U
initVal.valid := false.B

val channelReg = RegInit(initVal)
```

With combinations of `Bundles` and `Vecs` we can define our own data structures, which are powerful abstractions.

Possible pitfall: In Chisel 3, partial assignments are not allowed, although they have been allowed in Chisel 2 and are possible in Verilog and VHDL. The following

code will generate an error during circuit elaboration:

```
val assignWord = Wire(UInt(16.W))

assignWord(7, 0) := lowByte
assignWord(15, 8) := highByte
```

The argument is that it would be better to use bundles for this use case. One possible workaround for this issue is to create a (local) bundle, create a `Wire` from that bundle, assign the individual fields, casting that bundle with `asUInt` to a `UInt`, and assign this value to the target `UInt`. Note that we define here a `Bundle` as a local data structure as we need it only locally.

```
val assignWord = Wire(UInt(16.W))

class Split extends Bundle {
  val high = UInt(8.W)
  val low = UInt(8.W)
}

val split = Wire(new Split())
split.low := lowByte
split.high := highByte

assignWord := split.asUInt
```

The small drawback of this solution is that one needs to know in which orders bundle fields are merged to a single bit vector. Another option is to use a vector of `Bool` to individually assign values and then convert it to a `UInt`.

```
val vecResult = Wire(Vec(4, Bool()))

// example assignments
vecResult(0) := data(0)
vecResult(1) := data(1)
vecResult(2) := data(2)
vecResult(3) := data(3)

val uintResult = vecResult.asUInt
```

2.5 Wire, Reg, and IO

`UInt`, `SInt`, and `Bits` are Chisel types which by themselves do not represent hardware. Only wrapping them into a `Wire`, `Reg`, or `IO` generates hardware. A `Wire` represents combinational logic, a `Reg` represents a register (collection of D flip-flops), and an `IO` represents ports for the connection of a module (like pins of a concrete integrated circuit (IC)). Any Chisel type can be wrapped into a `Wire`, `Reg`, or `IO`.

You give a hardware component a name by assigning it to a Scala immutable variable:[7]

```
val number = Wire(UInt())
val reg = Reg(SInt())
```

You can later assign (or reassign) a value or expression to a `Wire`, `Reg`, or `IO` with the Chisel operator `:=`

```
number := 10.U
reg := value - 3.U
```

Note the small difference between the Scala assignment operator "=" and the Chisel operator ":=". You use Scala's "=" operator when *creating* a hardware object (and giving it a name) but you use Chisel's ":=" operator when assigning or reassigning a value to an *existing* hardware object.

Combinational values can be conditionally assigned, but they need to be assigned in every branch of the condition. Otherwise, one would describe a latch, which the Chisel compiler will reject. The best practice is to define a default value at the creation of the `Wire`. Therefore, the former code is better rewritten as follows.

```
val number = WireDefault(10.U(4.W))
```

Although Chisel infers the needed bit width for signals and registers, it is also a good practice to specify the intended bit width at the creation of the hardware object. In most cases it is also good practice to set registers to known initial values on reset:[8]

```
val reg = RegInit(0.S(8.W))
```

[7] Scala also supports mutable variables with `var`, but those are useless when describing hardware in Chisel.

[8] Leaving the register value undefined on reset may save some load on the reset wire. However, testing and verification is simplified with known reset values.

2.6 Chisel Generates Hardware

After seeing some initial Chisel code, it might look similar to classic programming languages such as Java or C. However, Chisel (or any other hardware description language) does define hardware components. While in a software program one line of code after the other is executed, in hardware all lines of code *execute in parallel*.

It is essential to keep in mind that Chisel code generates hardware. Try to imagine, or draw on a sheet of paper, the individual blocks that are generated by your Chisel circuit description. Each creation of a component adds hardware; each assignment statement generates gates and/or flip-flops.

More technically, when Chisel executes your code it runs as a Scala program, and by executing the Chisel statements, it *collects* the hardware components and connects those nodes. This network of hardware nodes is the hardware, which Chisel can spill out as Verilog code for ASIC or FPGA synthesis or can be tested with a Chisel tester. The network of hardware nodes is what is executed fully in parallel.

For a software engineer, imagine this immense parallelism that you can create in hardware without needing to partition your application into threads and get the locking correct for the communication.

2.7 Exercises

In the introduction you implemented a blinking LED on an FPGA board (from chisel-examples), which is a reasonable hardware *Hello World* example. It used only internal state, a single LED output, and no input. Copy that project into a new folder and extend it by adding some inputs to the `io Bundle` with `val sw = Input(UInt(2.W))`.

```
val io = IO(new Bundle {
  val sw = Input(UInt(2.W))
  val led = Output(UInt(1.W))
})
```

For those switches, you also need to assign the pins for the FPGA board. You can find examples of pin assignments in the Quartus project files of the ALU project (e.g., for the DE2-115 FPGA board).

When you have defined those inputs and the pin assignment, start with a simple test: drop all blinking logic from the design and connect one switch to the LED output; compile and configure the FPGA device. Can you switch the LED on an

off with the switch? If yes, you have now inputs available. If not, you need to debug your FPGA configuration. The pin assignment can also be done with the GUI version of the tool.

Now use two switches and implement one of the basic combinational functions, e.g., AND two switches and show the result on the LED. Change the function. The next step involves three input switches to implement a multiplexer: one acts as a select signal, and the other two are the two inputs for the 2:1 multiplexer.

Now you have been able to implement simple combinational functions and test them in real hardware in an FPGA. As a next step, we will take a first look at how the build process works to generate an FPGA configuration. Furthermore, we will also explore a simple testing framework from Chisel, which allows you to test circuits without configuring an FPGA and toggle switches.

3 Build Process and Testing

To get started with more interesting Chisel code we first need to learn how to compile Chisel programs, how to generate Verilog code for execution in an FPGA, and how to write tests for debugging and to verify that our circuits are correct.

Chisel is written in Scala, so any build process that supports Scala is possible with a Chisel project. One popular build tool for Scala is sbt, which stands for the Scala interactive build tool. Besides driving the build and test process, `sbt` also downloads the correct version of Scala and the Chisel libraries.

3.1 Building your Project with sbt

The Scala library that represents Chisel and the Chisel tester are automatically downloaded during the build process from a Maven repository. The libraries are referenced by `build.sbt`. It is possible to configure `build.sbt` with `latest.release` to always use the most recent version of Chisel. However, this means that on each build, the version is looked up from the Maven repository. This lookup needs an Internet connection for the build to succeed. It is better to use a dedicated version of Chisel and all other Scala libraries in your `build.sbt`. Sometimes, it is good to be able to write hardware code and test it without an Internet connection. For example, it is cool to do hardware design on a plane.

3.1.1 Source Organization

`sbt` inherits source conventions from the Maven build automation tool. Maven also organizes repositories of open-source Java libraries.[1]

Figure 3.1 shows the organization of the source tree of a typical Chisel project. The root of the project is the project home, which contains `build.sbt`. It may also include a `Makefile` for the build process, a README, and a LICENSE file. Folder `src` contains all source code. From there it is split between `main`, which contains

[1] That is also the place where you downloaded the Chisel library on your first build: `https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3`.

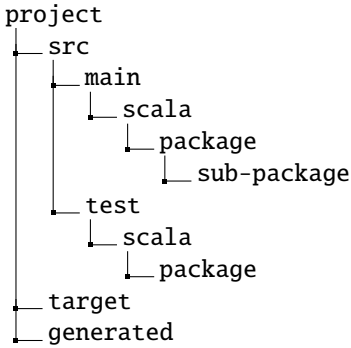


Figure 3.1: Source tree of a Chisel project (using `sbt`)

containing the hardware sources and `test` containing testers. The next folder in both cases is `scala`, as Chisel is based on Scala. If you want to include Java code, which may be useful for hardware generators, you would add a `java` folder. Chisel inherits from Scala, which itself inherits from Java, the organization of source in packages. Packages organize your Chisel code into namespaces. Packages can also contain sub-packages. The folder `target` contains the class files and other generated files. I recommend to also use a folder for generated Verilog files, which is usually call `generated`.

To use the facility of namespaces in Chisel, you need to declare that a class/module is defined in a package, in this example in `mypack`:

```
package mypack

import chisel3._

class Abc extends Module {
  val io = IO(new Bundle{})
}
```

Note that in this example we see the import of the `chisel3` package to use Chisel classes.

To use the module `Abc` in a different context (package name space), the components of packet `mypack` need to be imported. The underscore (_) acts as wildcard, meaning that all classes of `mypack` are imported.

```
import mypack._

class AbcUser extends Module {
  val io = IO(new Bundle{})

  val abc = new Abc()

}
```

It is also possible to not import all types from `mypack`, but use the fully qualified name `mypack.Abc` to refer to the module `Abc` in the package `mypack`.

```
class AbcUser2 extends Module {
  val io = IO(new Bundle{})

  val abc = new mypack.Abc()

}
```

It is also possible to import just a single class and create an instance of it:

```
import mypack.Abc

class AbcUser3 extends Module {
  val io = IO(new Bundle{})

  val abc = new Abc()

}
```

3.1.2 Running sbt

A Chisel project can be compiled and executed with a simple `sbt` command:

```
$ sbt run
```

This command will compile all your Chisel code from the source tree and search for classes that contain an `object` that either has a `main` method or extends `App`. If there is more than one such object, all objects are listed and one can be selected. You can also directly specify the object that shall be executed as a parameter to `sbt`:

```
$ sbt "runMain mypacket.MyObject"
```

By default, `sbt` searches only the `main` part of the source tree and not the `test` part.[2] To execute tests based on ChiselTest you can simply run them with

```
$ sbt test
```

If you have a test that does not follow the ChiselTest convention and it contains a `main` function, but is placed in the `test` part of the source tree you can execute it with following `sbt` command:

```
$ sbt "test:runMain mypacket.MyMainTest"
```

3.1.3 Generating Verilog

To synthesize Chisel code for an FPGA or ASIC we need to translate Chisel into a hardware description language that a synthesize tool understands. With Chisel, we can generate a synthesizable Verilog description of the circuit.

To generate the Verilog description, we need an application. A Scala object that `extends App` is an application that implicitly generates the main function where the application starts. The only action of this application is to create a new Chisel module, in that example `Hello`, and pass it to the Chisel function `emitVerilog()`. The following code will generate the Verilog file `Hello.v`.

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

Using the default version of `emitVerilog()` will put the generated files into the root folder of our project (where we run the `sbt` command). To put the generated files into a subfolder, we need to specify options to `emitVerilog()`. I recommend to specify a folder `generated`, as shown in Figure 3.1. The build options can be set as a second argument, which is an array of Strings. The following code will generate the Verilog file `Hello.v` in the subfolder `generated`.

```
object HelloOption extends App {
  emitVerilog(new Hello(), Array("--target-dir",
      "generated"))
}
```

[2] It is a convention from Java/Scala that the test folder contains unit tests and not objects with a `main`.

You can also request the Verilog code as a Scala `String` without writing a file. You can simply print out the string for testing.

```
object HelloString extends App {
  val s = getVerilogString(new Hello())
  println(s)
}
```

This form of output is popular when showing small Chisel examples in Scastie, a web-based Scala compiler and runtime. See Hello World on Scastie for an example.

3.1.4 Tool Flow

Figure 3.2 shows the tool flow of Chisel. The digital circuit is described in a Chisel class shown as `Hello.scala`. The Scala compiler compiles this class, together with the Chisel and Scala libraries, and generates the Java class file `Hello.class` that can be executed by a standard Java virtual machine (JVM). Executing this class with a Chisel driver generates the flexible intermediate representation for RTL (FIRRTL), an intermediate representation of digital circuits. In our example, the file is `Hello.fir`. The FIRRTL compiler performs transformations on the circuit.

Treadle is a FIRRTL interpreter that can simulate the circuit. Together with the Chisel tester, it can be used to debug and test Chisel circuits. With assertions, we can provide test results. Treadle can also generate waveform files (`Hello.vcd`) that can be viewed with a waveform viewer (e.g., the free viewer GTKWave or Modelsim).

One FIRRTL transformation, the Verilog emitter, generates Verilog code for synthesis (`Hello.v`). A circuit synthesize tool (e.g., Intel Quartus, AMD/Xilinx Vivado, or an ASIC tool) synthesizes the circuit. In an FPGA design flow, the tool generates the FPGA bitstream that is used to configure the FPGA, e.g., `Hello.bit`.

Note that Figure 3.2 shows the Chisel tool flow until Chisel 3.6. Later versions of Chisel use a different backend (see next section).

Now that we know the basic structure of a Chisel project and how to compile and run it with `sbt`, we can continue with a simple testing framework.

3.1.5 Chisel Versions

Chisel changed the version scheme from version on 5.0.0 on to a semantic versioning in the form of MAJOR.MINOR.PATCH. Prior to Chisel 5.0.0 the scheme was 3.MAJOR.MINOR. Changes in the minor version of Chisel contain new functionality, but in a backwards compatible way. A change in the major version may intro-

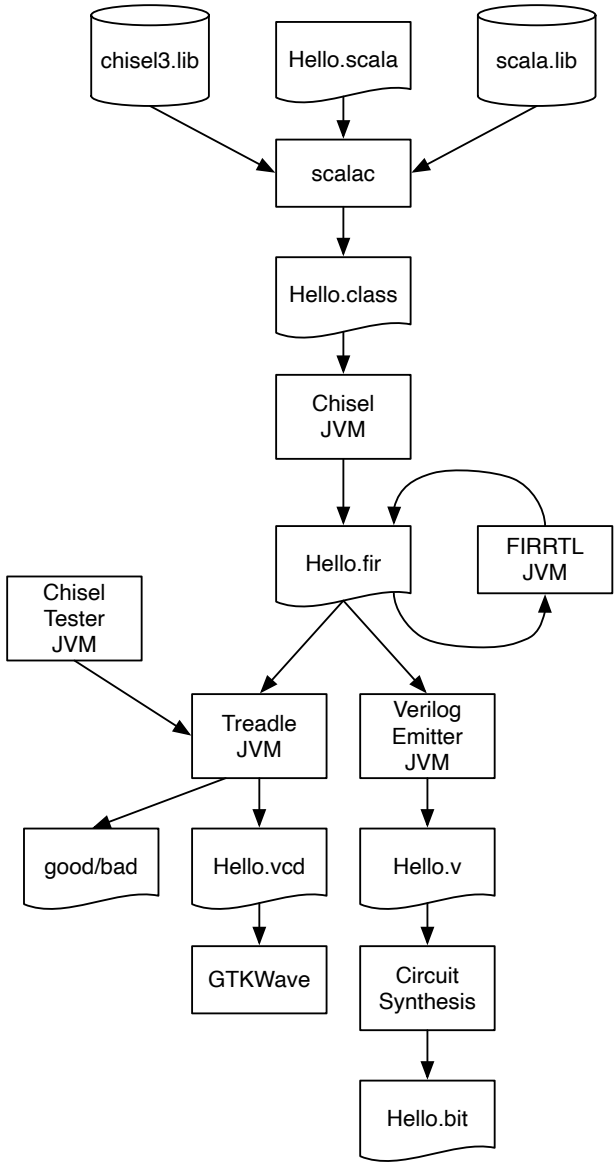


Figure 3.2: Tool flow of the Chisel ecosystem.

duce breaking changes. To emphasize this change in the versions, Chisel skipped version 4.

Starting with Chisel 5, the backend of Chisel has change from the Scala-based compiler for FIRRTL to using `firtool` to generate Verilog. `firtool` is part of a larger project, the LLVM CIRCT project. The latest version of "Chisel 3" is 3.6 and contains both backends, the Scala based compiler and `firtool`. The `build.sbt` for 3.6 is as follows:

```
scalaVersion := "2.13.14"

scalacOptions ++= Seq(
  "-deprecation",
  "-feature",
  "-unchecked",
  "-language:reflectiveCalls",
)

val chiselVersion = "3.6.1"
addCompilerPlugin("edu.berkeley.cs" % "chisel3-plugin" %
    chiselVersion cross CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
    chiselVersion
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.6.1"
```

With Chisel version 5, the `firtool` binary needs to be installed manually. Chisel 6 includes `firtool` as part of the Chisel library and is therefore the recommended latest version of Chisel.

The change to CIRCT has implications of the future testing infrastructure. Future testing will be performed with a Scala API on top of Verilator, which makes installation of Verilator mandatory. However, there is development within the CIRCT project to simulate within the CIRCT framework. This tool is called Arcilator. However, ChiselTest has been ported to Chisel 5 and Chisel 6. We expect a compatibility layer in Chisel 7 for a smooth transition to a new testing framework.

From Chisel 5 on the libraries are published under `org.chipsalliance`. Therefore, the `build.sbt` need to be changed as follows:

```
scalaVersion := "2.13.14"

scalacOptions ++= Seq(
```

```
  "-deprecation",
  "-feature",
  "-unchecked",
  "-language:reflectiveCalls",
)

val chiselVersion = "6.5.0"
addCompilerPlugin("org.chipsalliance" % "chisel-plugin" %
    chiselVersion cross CrossVersion.full)
libraryDependencies += "org.chipsalliance" %% "chisel" %
    chiselVersion
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "6.0.0"
```

This book covers, and has been tested with, Chisel 3.5, 3.6, 5, and 6.

Chisel, Scala, and Java Versions

Chisel is a Scala library and therefore is dependent on a Scala version. Scala itself is built on top of Java (using the Java library and running on the JVM) and therefore depends on a specific Java version. Changes in the Java version needs updates in the Scala (patch) version. A safe baseline Java version is Java 8, where all Scala and Chisel versions work.[3] Chisel itself has been maintained to work with Scala 2.12 and 2.13, up to version 5. Chisel 6 is based on Scala 2.13 only.[4] A safe selection is to use Scala 2.13. However, as Chisel is using a Scala compiler plugin, the highest supported patch version of Scala is restricted by the published compiler plugin.[5] Table 3.1 shows Chisel versions with their support of the highest Scala and Java versions.

3.1.6 Using a GitHub Template

The chisel-empty project is a minimal Chisel project that contains an adder circuit, a tester, and a `Makefile` to generate Verilog code, test the circuit, and cleanup the repository. That project is a GitHub template, which means you can simply start a new GitHub repository using this template.

[3] see https://docs.scala-lang.org/overviews/jdk-compatibility/overview.html
[4] Getting Chisel to work on top of Scala 3 is work in progress.
[5] see https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3-plugin and https://mvnrepository.com/artifact/org.chipsalliance/chisel-plugin

Chisel	Scala	Java
3.5.6	2.13.10	17
3.6.1	2.13.14	22
5.3.x	2.13.14	22
6.5.x	2.13.14	22
6.6 ...		
6.7.x	2.13.16	24

Table 3.1: Chisel and highest supported Scala and Java versions.

Navigate to that GitHub repository and press the button "Use this template" to create your GitHub project. Then clone your new project locally and explore the `Makefile`. Generate Verilog code with:

```
make
```

Test the adder with:

```
make test
```

Remove all generated files with:

```
make clean
```

You can also execute these tasks by running the `sbt` and `git` commands directly.

3.2 Testing with Chisel

Tests of hardware designs are usually called test benches. The test bench instantiates the design under test (DUT), drives input ports, observes output ports, and compares them with expected values. Chisel provides the ChiselTest in package `chiseltest`.

One strength of Chisel is that it can use the full power of Scala to write test benches. One can, for example, code the expected functionality of the hardware in a software simulator and compare the simulation of the hardware with the software simulation. This method is very efficient when testing an implementation of a processor [19].

3.2.1 ScalaTest

ScalaTest is a testing tool for Scala (and Java). ChiselTest is an extension of ScalaTest. Therefore, we first explore a simple ScalaTest example. To use it, include the library in your `build.sbt` with the following line:

```
libraryDependencies += "org.scalatest" %% "scalatest" %
    "3.1.4" % "test"
```

Tests are usually found in `src/test/scala` and the entire test suite can be run with:

```
$ sbt test
```

A minimal test (a testing hello world) to test a Scala integer addition and a multiplication looks as follows:

```
import org.scalatest._
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers

class ExampleTest extends AnyFlatSpec with Matchers {
  "Integers" should "add" in {
    val i = 2
    val j = 3
    i + j should be (5)
  }

  "Integers" should "multiply" in {
    val a = 3
    val b = 4
    a * b should be (12)
  }
}
```

ScalaTest enables simple unit tests that read like an executable specification. The example above contains two tests and the output of the test run will repeat the specification and show that both tests passed:

```
[info] ExampleTest:
[info] Integers
[info] - should add
[info] Integers
```

```
[info] - should multiply
[info] ScalaTest
[info] Run completed in 119 milliseconds.
[info] Total number of tests run: 2
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 2, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 2, Failed 0, Errors 0, Passed 2
```

`sbt test` executes all available tests, which is useful for regression tests.[6] However, if you want to run just a single test (suite) you can do this with:

```
$ sbt "testOnly ExampleTest"
```

If you misspell the class name, for example, `ExampleTest`, there will be a relatively silent error message: `No tests were executed`.

3.2.2 ChiselTest

ChiselTest is the standard testing tool for Chisel modules based on the ScalaTest tool for Scala and Java, which we can use to run Chisel tests. To use it, include the `chiseltest` library in your `build.sbt` with the following line:

```
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.5.6"
```

Including ChiselTest this way automatically includes the necessary version of ScalaTest. Therefore, you do not need to include a line for the ScalaTest library. To use ChiselTest, the following packages need to be imported:

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
```

Testing a circuit contains (at least) two components: the device under test (often called DUT) and the testing logic, also called a test bench. Tests are started with `sbt test`. No object with a `main` function is needed.

The following code shows our simple design under test. It contains two input ports (2-bit width) and two output ports, a 2-bit width and a `Bool`. The circuit does

[6] Try `sbt test` in the repository of this book and you will see more than 90 tests passing.

a bit-wise AND to its inputs `a` and `b` and outputs the result on `out` and tests the two signals for equality:

```
class DeviceUnderTest extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
    val equ = Output(Bool())
  })

  io.out := io.a & io.b
  io.equ := io.a === io.b
}
```

The test bench for this DUT extends `AnyFlatSpec` with `ChiselScalatestTester`, which provides ChiselTest functionality within ScalaTest. The method `test()` is invoked with the DUT as parameter and the test code as a function literal.

```
class SimpleTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      println("Result is: " + dut.io.out.peekInt())
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      println("Result is: " + dut.io.out.peekInt())
    }
  }
}
```

The input and output ports of the DUT are accessed with `dut.io`. You can set values via a `poke` on a port, which takes the value as a Chisel type of the input port as a parameter. An output port can be read by invoking `peekInt()` or `peekBoolean()` on the port, which will return the value as a Scala type. The tester advances the simulation by one clock cycle with `dut.clock.step()`. For advancing the simulation by several clock cycles, we can provide a parameter to `step()`. We can print the values

of the outputs with `println()`.

When you run the test

```
$ sbt "testOnly SimpleTest"
```

you will see the results printed to the terminal (besides other information):

```
...
Result is: 0
Result is: 2
[info] SimpleTest:
[info] DUT
[info] - should pass
...
```

We see that 0 AND 1 results in 0; 3 AND 2 results in 2. Besides manually inspecting printouts, which is a good starting point, we can also express our expectations in the test bench itself by invoking `expect(value)` on the output port and the expected value as a parameter. The following example shows testing with expectations:

```
class SimpleTestExpect extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.out.expect(0.U)
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      dut.io.out.expect(2.U)
    }
  }
}
```

Executing this test does not print out any values from the hardware, but that all tests passed as all expect values are correct.

```
[info] SimpleTestExpect:
[info] DUT
[info] - should pass
```

```
[info] ScalaTest
[info] Run completed in 1 second, 85 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
[info] Passed: Total 1, Failed 0, Errors 0, Passed 1
```

A failed test, when either the DUT or the test bench contains an error, produces an error message describing the difference between the expected and actual value. In the following, we changed the test bench to expect a 4, which is an error:

```
[info] SimpleTestExpect:
[info] DUT
[info] - should pass *** FAILED ***
[info]   io_out=2 (0x2) did not equal expected=4 (0x4)
                  (lines in testing.scala: 27) (testing.scala:35)
[info] ScalaTest
[info] Run completed in 1 second, 214 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 0, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed: Total 1, Failed 1, Errors 0, Passed 0
[error] Failed tests:
[error]  SimpleTestExpect
```

The `peek()` function returns a Chisel type, which would need conversion to be used as Scala type. To simplify using test values in Scala land, ChiselTest supports `peekInt()` and `peekBoolean()`. The following test example reads the output with `peekInt()`, which returns a Scala integer[7] that is used in the `assert()` statement. Similar we can read the `equ` output into a Scala `Boolean`, directly used in the assert statement.

```
class SimpleTestPeek extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
```

[7] To support arbitrarily wide integer values, the return value is a Scala `BigInt` instead of a Scala `Int`.

```
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.out.expect(0.U)
      val res = dut.io.out.peekInt()
      assert(res == 0)
      val equ = dut.io.equ.peekBoolean()
      assert(!equ)
    }
  }
}
```

This example is a bit too simple to see the benefit of reading values from the DUT into Scala types. However, with more complex tests, e.g., looping till some value is true, these functions become useful.

In this section, we described the basic testing facility with Chisel for simple tests. However, keep in mind that the full power of Scala is available to write testers. This includes, for example, writing a reference model of your hardware in Scala to test the DUT against.

3.2.3 Waveforms

Testers, as described above, work well for small designs and for unit testing, as it is common in software development. A collection of unit tests can also serve regression testing. However, for debugging more complex designs, one would like to investigate several signals at once. A classic approach to debug digital designs is displaying the signals in a waveform. In a waveform the signals are displayed over time.

Chisel testers can generate a waveform that includes all registers and all IO signals. In the following examples, we show waveform testers for the `DeviceUnderTest` from the former example (the 2-bit AND function). To generate a waveform for a test pass a definition of `writeVcd=1` to the test, as shown in the following `sbt` command:

```
sbt "testOnly SimpleTest -- -DwriteVcd=1"
```

You can view the waveform with the free viewer GTKWave or with ModelSim. Start GTKWave and select *File – Open New Window* and navigate to the folder where the Chisel tester put the `.vcd` file. By default, the generated files are in `test_run_dir` within a subfolder named after the test description. Within this folder, you should be able to find `DeviceUnderTest.vcd`. You can select the signals from

the left side and drag them into the main window. If you want to save a configuration of signals you can do so with *File – Write Save File* and load it later with *File – Read Save File*.

The generation of waveforms can also be initiated by passing the `WriteVcdAnnotation` annotation to the `test()` function.[8] We start with a simple tester that pokes values to the inputs and advances the clock with `step`. We do not read any output or compare it with `expect`. Instead, we will inspect the generated waveform in the `.vcd` file.

```
class WaveformTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "Waveform" should "pass" in {
    test(new DeviceUnderTest)
      .withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(0.U)
      dut.clock.step()
      dut.io.a.poke(1.U)
      dut.io.b.poke(0.U)
      dut.clock.step()
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.a.poke(1.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
    }
  }
}
```

Explicitly enumerating all possible input values does not scale. Therefore, we will use some Scala code to drive the DUT. The following tester enumerates all possible values for the two 2-bit input signals.

```
class WaveformCounterTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "WaveformCounter" should "pass" in {
    test(new DeviceUnderTest)
      .withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
```

[8] This is an alternative to using the command line options.

```
      for (a <- 0 until 4) {
        for (b <- 0 until 4) {
          dut.io.a.poke(a.U)
          dut.io.b.poke(b.U)
          dut.clock.step()
        }
      }
    }
  }
}
```

and execute it with

```
$ sbt "testOnly WaveformCounterTest"
```

3.2.4 printf Debugging

Another form of debugging is "printf debugging". The name of this debugging method comes from simply putting `printf` statements in C code to print variables of interest during the execution of the program. This printf debugging is also available during testing of Chisel circuits. The printing happens at the rising edge of the clock. A `printf` statement can be inserted just anywhere in the module definition, as shown in the printf debugging version of the DUT.

```
class DeviceUnderTestPrintf extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
  })

  io.out := io.a & io.b
  printf("dut: %d %d %d\n", io.a, io.b, io.out)
}
```

When testing this module with the counter-based tester, which iterates over all possible values, we get the following output, verifying that the AND function is correct:

```
Elaborating design...
Done elaborating.
dut:  0  0  0
```

```
dut:  0  1  0
dut:  0  2  0
dut:  0  3  0
dut:  1  0  0
dut:  1  1  1
dut:  1  2  0
dut:  1  3  1
dut:  2  0  0
dut:  2  1  0
dut:  2  2  2
dut:  2  3  2
dut:  3  0  0
dut:  3  1  1
dut:  3  2  2
dut:  3  3  3
dut:  0  0  0
test DeviceUnderTestPrintf Success: 0 tests passed in 18 cycles
in 0,031521 seconds 571,04 Hz
```

Chisel `printf` supports C and Scala style formatting.

3.3 Exercises

For this exercise, we will revisit the blinking LED from chisel-examples and explore Chisel testing.

3.3.1 A Minimal Project

First, let us find out what a minimal Chisel project is. Explore the files in the Hello World example. The `Hello.scala` is the single hardware source file. It contains the hardware description of the blinking LED (`class Hello`) and an `App` that generates the Verilog code.

Each file starts with the import of Chisel and related packages:

```
import chisel3._
```

Then follows the hardware description, as shown in Listing 1.1. To generate the Verilog description, we need an application. The only action of this application is to create a new `Hello` object and pass it to the `emitVerilog()` function.

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

Run the generation of the example manually with

```
$ sbt run
```

and explore the generated `Hello.v` with an editor. The generated Verilog code may not be very readable, but we can find out some details. The file starts with a module `Hello`, which is the same name as our Chisel module. We can identify our LED port as `output io_led`. Pin names are the Chisel names with a prepended `io_`. Besides our LED pin, the module also contains `clock` and `reset` input signals. Those two signals are added automatically by Chisel.

Furthermore, we can identify the definition of our two registers `cntReg` and `blkReg`. We may also find the reset and update of those registers at the end of the module definition. Note that Chisel generates a synchronous reset.

For `sbt` to be able to fetch the correct Scala compiler and the Chisel library, we need a `build.sbt`:

```
scalaVersion := "2.13.10"

scalacOptions ++= Seq(
  "-feature",
  "-language:reflectiveCalls",
)

val chiselVersion = "3.5.6"
addCompilerPlugin("edu.berkeley.cs" %% "chisel3-plugin" %
    chiselVersion cross CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
    chiselVersion
libraryDependencies += "edu.berkeley.cs" %% "chiseltest" %
    "0.5.6"
```

Note that in this example, we have a concrete Chisel version number to avoid checking on each run for a new version (which will fail if we are not connected to the Internet, e.g., when doing hardware design during a flight). Additionally, we have added the Chisel3 compiler plugin which is needed since Chisel 3.5. Change the `build.sbt` configuration to use the latest Chisel version by changing the library

dependency to

```
libraryDependencies += "edu.berkeley.cs" %% "chisel3" %
    "latest.release"
```

and rerun the build with `sbt`. Is there a newer version of Chisel available and will it be automatically downloaded?

For convenience, the project also contains a `Makefile`. It just contains the `sbt` command, so we do not need to remember it and can generate the Verilog code with:

```
make
```

Besides a `README` file, the example project also contains project files for different FPGA boards. E.g., in quartus/altde2-115 you can find the two project files to define a Quartus project for the DE2-115 board. The main definitions (source files, device, pin assignments) can be found in a plain text file hello.qsf. Explore the file and find out which pins are connected to which signals. If you need to adapt the project to a different board, this is where the changes are applied. If you have Quartus installed, open that project, compile with the green *Play* button, and then configure the FPGA.

Note that the *Hello World* is a minimal Chisel project. More realistic projects have their source files organized in packages and contain testers.

3.3.2 A Testing Exercise

In the last chapter's exercise, you have extended the blinking LED example with some input to build an AND gate and a multiplexer and run this hardware in an FPGA. We will now use this example and test the functionality with a Chisel tester to automate testing and also to be independent of an FPGA board. Use your designs from the previous chapter and add a Chisel tester to test the functionality. Try to enumerate all possible inputs and test the output with `expect()`.

Testing within Chisel can speed up the debugging of your design. However, it is always a good idea to synthesize your design for an FPGA and run tests with the FPGA. There you can perform a reality check on the size of your design (usually in LUTs and flip-flops) and your performance of your design in maximum clocking frequency. As a reference point, a textbook style pipelined RISC processor may consume about 3000 4-bit LUTs and may run around 100 MHz in a low-cost FPGA (Intel Cyclone or Xilinx Spartan).

4 Components

A larger digital design is structured into a set of components, often in a hierarchical way. Each component has an interface with input and output wires, sometimes called ports. These are similar to input and output pins on an integrated circuit (IC). Components are connected by wiring up the inputs and outputs. Components may contain subcomponents to build the hierarchy. The outermost component, which is connected to physical pins on a chip, is called the top-level component.

In this chapter, we will explain how components are described in Chisel and provide several simple examples of components. Note that the components in this section are very small (e.g., just an adder), just to show the principles: how to define, instantiate, and connect components. Real-world examples shall contain more "meat" than just a single line for an adder.

4.1 Components in Chisel are Modules

Hardware components are called modules in Chisel. Each module extends the class `Module` and contains a field `io` for the interface. The interface is defined by a `Bundle` that is wrapped into a call to `IO()`. The `Bundle` contains fields to represent input and output ports of the module. The direction is given by wrapping a field into either an `Input()` or an `Output()`. The direction is from the view of the component itself.

We show an example design where we build a counter out of two components: an adder and a register. Figure 4.1 shows the schematic of the adder component. It has two inputs (`a` and `b`) and one output (`y`). Listing 4.1 shows the Chisel definition of the adder. The input and output signals are accessed with the *dot notation*, such as `io.a`, as they are part of the `io Bundle`.

Figure 4.2 shows another simple component, an 8-bit register. The Chisel code for this component is shown in Listing 4.2.

Now we build a counter with those two components that counts from 0 to 9 and repeats. Figure 4.3 shows the schematic of the counter. We use an adder to add 1 to the value of `count`. A multiplexer selects between this sum and 0. That result,

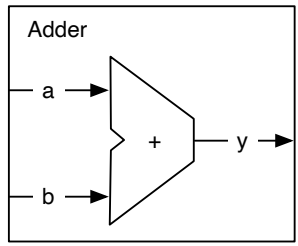


Figure 4.1: An adder component.

```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  io.y := io.a + io.b
}
```

Listing 4.1: The adder component in Chisel.

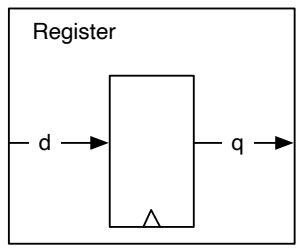


Figure 4.2: A register components.

```
class Register extends Module {
  val io = IO(new Bundle {
    val d = Input(UInt(8.W))
    val q = Output(UInt(8.W))
  })

  val reg = RegInit(0.U)
  reg := io.d
  io.q := reg
}
```

Listing 4.2: The register component in Chisel.

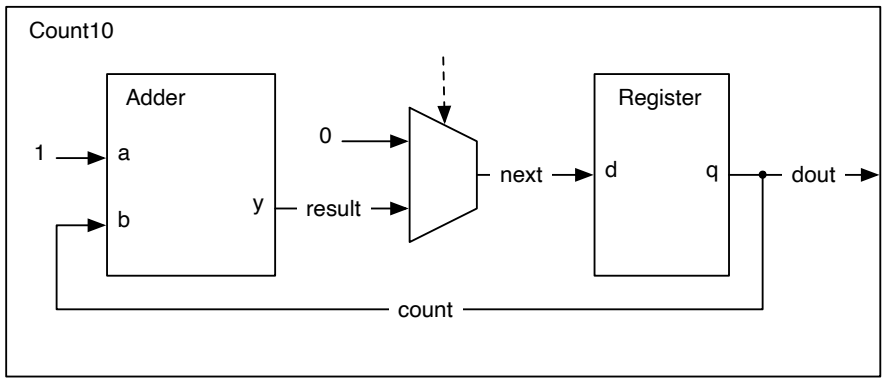


Figure 4.3: A counter built out of components.

```
class Count10 extends Module {
  val io = IO(new Bundle {
    val dout = Output(UInt(8.W))
  })

  val add = Module(new Adder())
  val reg = Module(new Register())

  // the register output
  val count = reg.io.q
  // connect the adder
  add.io.a := 1.U
  add.io.b := count
  val result = add.io.y
  // connect the Mux and the register input
  val next = Mux(count === 9.U, 0.U, result)
  reg.io.d := next
  io.dout := count
}
```

Listing 4.3: A counter built out of components.

called `next` is the input for the register component. The output of the register is the `count` value and also the output of the `Count10` component (`dout`).

Listing 4.3 shows the Chisel code for the `Count10` component. The two components are instantiated by creating them with `new`, wrapping them into a `Module()`, and assigning them the names `add` and `reg`. In this example, we give the output of the register (`reg.io.q`) the name `count`.

We connect `1.U` and `count` to the two inputs of the adder component. We give the output of the adder component the name `result`. The multiplexer selects between `0.U` and `result` depending on the current counter value `count`. We name the output of the multiplexer `next` and connect it to the input of the register components. Finally, we connect the counter value `count` to the single output of the `Count10` component, `io.dout`.

4.2 Nested Components

A medium- to high-complexity hardware design is built out of a hierarchy of nested components. Figure 4.4 shows the structure of such an example design. Component `CompC` has three input ports and two output ports. The component itself is assembled out of two subcomponents: `CompA` and `CompB`, which are connected to the inputs and outputs of `CompC`. One output of `CompA` is connected to an input of `CompB`. Component `CompD` is at the same hierarchy level as component `CompC` and connected to it.

Listing 4.4 shows the definition of the two example components `CompA` and `CompB` from Figure 4.4. Component `CompA` has two inputs, named `a` and `b`, and two outputs, named `x` and `y`. For the ports of component `CompB` we chose the names `in1`, `in2`, and `out`. All ports use an unsigned integer (`UInt`) with a bit width of 8. As this example code is about connecting components and building a hierarchy, we do not show any implementation within the components. The implementation of the component is written at the place where the comments states "function of X". As we have no function associated with those example components, we used generic port names. For a real design, use descriptive port names such as `data`, `valid`, or `ready`.

Component `CompC`, shown in Listing 4.5, has three input and two output ports. It is built out of components `CompA` and `CompB`. We show how `CompA` and `CompB` are connected to the ports of `CompC` and also the connection between an output port of `CompA` and an input port of `CompB`.

Components are created with `new`, e.g., `new CompA()`, and need to be wrapped into a `Module()`. The reference to that module is stored in a local variable, in this

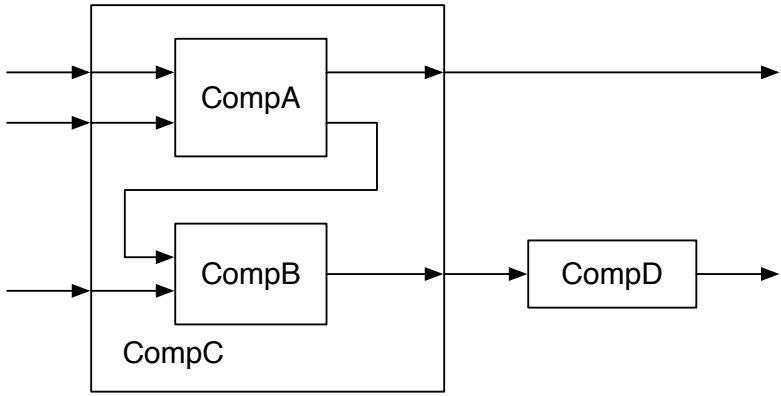


Figure 4.4: A design consisting of a hierarchy of components.

```
class CompA extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val x = Output(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  // function of A
}

class CompB extends Module {
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of B
}
```

Listing 4.4: Definition of components `CompA` and `CompB`

```
class CompC extends Module {
  val io = IO(new Bundle {
    val inA = Input(UInt(8.W))
    val inB = Input(UInt(8.W))
    val inC = Input(UInt(8.W))
    val outX = Output(UInt(8.W))
    val outY = Output(UInt(8.W))
  })

  // create components A and B
  val compA = Module(new CompA())
  val compB = Module(new CompB())

  // connect A
  compA.io.a := io.inA
  compA.io.b := io.inB
  io.outX := compA.io.x
  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.inC
  io.outY := compB.io.out
}
```

Listing 4.5: Component `CompC`

```
class CompD extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of D
}
```

Listing 4.6: Component `CompD`

example `val compA = Module(new CompA())`.

With this reference, we can access the IO ports by dereferencing the `io` field of the module and then the individual fields of the IO `Bundle`.

The simplest component (`CompD`) in our design, shown in Listing 4.6, has just an input port, named `in`, and an output port named `out`. The final missing piece of our example design is the top-level component, which itself is assembled out of components `CompC` and `CompD`, shown in Listing 4.7.

Good component design is similar to the good design of functions or methods in software design. One of the main questions is how much functionality shall we put into a component and how large should a component be. The two extremes are tiny components, such an adder, and huge components, such as a full microprocessor,

Beginners in hardware design often start with tiny components. The problem is that digital design books use tiny components to show the principles. The sizes of the examples (in those books, and also in this book) are small to fit onto a page and to avoid distracting details.

The interface to a component is a little bit verbose (with types, names, directions, IO construction). As a rule of thumb, I propose that the core of the component, the function, should be at least as long as the interface of the component.

For tiny components, such as a counter, Chisel provides a more lightweight way to describe them as functions that return hardware.

4.3 An Arithmetic Logic Unit

One of the central components for circuits that compute, e.g., a microprocessor, is an arithmetic-logic unit, or ALU for short. Figure 4.5 shows the symbol of an ALU.

```
class TopLevel extends Module {
  val io = IO(new Bundle {
    val inA = Input(UInt(8.W))
    val inB = Input(UInt(8.W))
    val inC = Input(UInt(8.W))
    val outM = Output(UInt(8.W))
    val outN = Output(UInt(8.W))
  })

  // create C and D
  val c = Module(new CompC())
  val d = Module(new CompD())

  // connect C
  c.io.inA := io.inA
  c.io.inB := io.inB
  c.io.inC := io.inC
  io.outM := c.io.outX
  // connect D
  d.io.in := c.io.outY
  io.outN := d.io.out
}
```

Listing 4.7: Top-level component

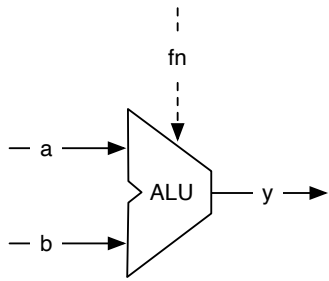


Figure 4.5: An arithmetic logic unit, or ALU for short.

The ALU has two data inputs, labeled a and b in the figure, one function input fn, and an output, labeled y. The ALU operates on a and b and provides the result at the output y. The input fn selects the operation on a and b. The operations are usually some arithmetic, such as addition and subtraction, and some logical functions such as and, or, xor. That's why it is called an ALU.

The ALU is usually a combinational circuit without any state elements. An ALU might also have additional outputs to signal properties of the result, such as zero or the sign.

The following code shows an ALU with 16-bit inputs and a 16-bit output that supports addition, subtraction, or, and and. operation, selected by a 2-bit fn signal.

```
class Alu extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(16.W))
    val b = Input(UInt(16.W))
    val fn = Input(UInt(2.W))
    val y = Output(UInt(16.W))
  })

  // some default value is needed
  io.y := 0.U

  // The ALU selection
  switch(io.fn) {
    is(0.U) { io.y := io.a + io.b }
    is(1.U) { io.y := io.a - io.b }
    is(2.U) { io.y := io.a | io.b }
    is(3.U) { io.y := io.a & io.b }
  }
}
```

In this example, we use a new Chisel construct, the switch/is construct to describe the table that selects the output of our ALU. To use this utility function, we need to import another Chisel package:

```
import chisel3.util._
```

4.4 Bulk Connections

For connecting components with multiple IO ports, Chisel provides the bulk connection operator <>. This operator connects parts of bundles in both directions. Chisel uses the names of the leaf fields for the connection. If a name is missing, it is not connected.

As an example, let us assume we build a pipelined processor. The fetch stage has the following interface:

```
class Fetch extends Module {
  val io = IO(new Bundle {
    val instr = Output(UInt(32.W))
    val pc = Output(UInt(32.W))
  })
  // ... Implementation of fetch
}
```

The next stage is the decode stage.

```
class Decode extends Module {
  val io = IO(new Bundle {
    val instr = Input(UInt(32.W))
    val pc = Input(UInt(32.W))
    val aluOp = Output(UInt(5.W))
    val regA = Output(UInt(32.W))
    val regB = Output(UInt(32.W))
  })
  // ... Implementation of decode
}
```

The final stage of our simple processor is the execute stage.

```
class Execute extends Module {
  val io = IO(new Bundle {
    val aluOp = Input(UInt(5.W))
    val regA = Input(UInt(32.W))
    val regB = Input(UInt(32.W))
    val result = Output(UInt(32.W))
  })
  // ... Implementation of execute
}
```

To connect all three stages we need just two <> operators. We can also connect the port of a submodule with the parent module.

```
val fetch = Module(new Fetch())
val decode = Module(new Decode())
val execute = Module(new Execute)

fetch.io <> decode.io
decode.io <> execute.io
io <> execute.io
```

5 Combinational Building Blocks

In this chapter, we explore various combinational circuits, basic building blocks that we can use to construct more complex systems. In principle, all combinational circuits can be described with Boolean equations. However, more often, a description in the form of a table is more efficient. We let the synthesize tool extract and minimize the Boolean equations. Two basic circuits, best described in a table form, are a decoder and an encoder.

5.1 Combinational Circuits

Before describing some standard combinational building blocks, we will explore how combinational circuits can be expressed in Chisel. The simplest form is a Boolean expression, which can be assigned a name:

```
val e = (a & b) | c
```

The Boolean expression is given a name (`e`) by assigning it to a Scala value. The expression can be reused in other expressions:

```
val f = ~e
```

Such an expression is considered fixed. A reassignment to `e` with `=` would result in a Scala compiler error: `reassignment to val`. A try with the Chisel operator `:=`, as shown below,

```
e := c & b
```

results in a runtime exception: `Cannot reassign to read-only`.

Chisel also supports describing combinational circuits with conditional updates. Such a circuit is declared as a `Wire`. Then you use conditional operations, such as `when`, to describe the logic of the circuit. The following code declares a `Wire` `w` of type `UInt` and assigns it a default value of `0`. The `when` block takes a Chisel `Bool` and reassigns `3` to `w` if `cond` is `true.B`.

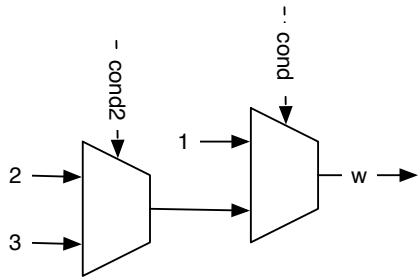


Figure 5.1: A chain of multiplexers.

```
val w = Wire(UInt())

w := 0.U
when (cond) {
  w := 3.U
}
```

The logic of the circuit is a multiplexer, where the two inputs are the constants `0` and `3` and the select signal is the condition `cond`. Keep in mind that we describe hardware circuits and not a software program with conditional execution.

The Chisel condition construct `when` also has a form of *else*, it is called `.otherwise`. With assigning a value under any condition we can omit the default value assignment:

```
val w = Wire(UInt())

when (cond) {
  w := 1.U
} .otherwise {
  w := 2.U
}
```

Chisel also supports a chain of conditionals (like a if/elseif/else chain) with `.elsewhen`:

```
val w = Wire(UInt())

when (cond) {
```

```
    w := 1.U
  } .elsewhen (cond2) {
    w := 2.U
  } .otherwise {
    w := 3.U
  }
```

This chain of `when`, `.elsewhen`, and `.otherwise` constructs a chain of multiplexers. Figure 5.1 shows this chain of multiplexers. That chain introduces a priority, i.e., when `cond` is true, the other conditions are not evaluated.

Note the '.' in `.elsewhen` needed to chain methods in Scala. Those `.elsewhen` branches can be arbitrarily long. However, if the chain of conditions depends on a single signal, it is better to use the `switch` statement, which is introduced in the following subsection with a decoder circuit.

For more complex combinational circuits, it might be practical to assign a default value to a `Wire`. A default assignment can be combined with the wire declaration with `WireDefault`.

```
val w = WireDefault(0.U)

when (cond) {
  w := 3.U
}
// ... and some more complex conditional assignments
```

One might ask, why do we use `when`, `.elsewhen`, and `.otherwise` when Scala has `if`, `else if`, and `else`? Those Scala statements are for conditional execution of Scala code, not generating Chisel (multiplexer) hardware. Those Scala conditionals have their use in Chisel when we write circuit generators, which take parameters to conditionally generate *different* hardware instances.

5.2 Decoder

A decoder converts a binary number of n bits to an m-bit signal, where $m \leq 2^n$. The output is one-hot encoded (where exactly one bit is one). Figure 5.2 shows a 2-bit to 4-bit decoder. We can describe the function of the decoder with a truth table, such as Table 5.1.

A Chisel `switch` statement describes the logic as a truth table. To use the `switch` statement, we need to include the package `chisel.util`.

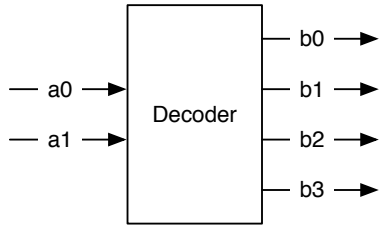


Figure 5.2: A 2-bit to 4-bit decoder.

a	b
00	0001
01	0010
10	0100
11	1000

Table 5.1: Truth table for a 2 to 4 decoder.

```
import chisel3.util._
```

The following code uses the `switch` statement of Chisel to describe a decoder:

```
  result := 0.U

  switch(sel) {
    is (0.U) { result := 1.U}
    is (1.U) { result := 2.U}
    is (2.U) { result := 4.U}
    is (3.U) { result := 8.U}
  }
```

The above `switch` statement lists all possible values of the `sel` signal and assigns the decoded value to the `result` signal. Note that even if we enumerate all possible input values, Chisel still needs us to assign a default value, as we do by assigning an initial 0 to `result`. This assignment will never be active and therefore optimized away by the synthesize tool. It is intended to avoid situations with incomplete assignments for combinational circuits (in Chisel a `Wire`) that will result in unintended latches in

hardware description languages such as VHDL and Verilog. Chisel does not allow incomplete assignments.

In the example before, we used unsigned integers for the signals. Maybe a clearer representation of an encode circuit uses binary notation:

```
switch (sel) {
  is ("b00".U) { result := "b0001".U}
  is ("b01".U) { result := "b0010".U}
  is ("b10".U) { result := "b0100".U}
  is ("b11".U) { result := "b1000".U}
}
```

A table gives a very readable representation of the decoder function but is also a little bit verbose. When examining the table, we see a regular structure: a 1 is shifted left by the number represented by `sel`. Therefore, we can express a decoder with the Chisel shift operation <<.

```
result := 1.U << sel
```

Decoders are used as a building block for a multiplexer by using the output as an enable with an AND gate for the multiplexer data input. However, in Chisel, we do not need to construct a multiplexer because a `Mux` is available in the core library. Decoders can also be used for address decoding of some bits of an address bus of a microprocessor. The outputs are used as select signals for memories and different IO devices connected to the microprocessor (see Section 12.1).

5.3 Encoder

An encoder converts a one-hot encoded input signal into a binary encoded output signal. The encoder does the inverse operation of a decoder.

Figure 5.3 shows a 4-bit one-hot input to a 2-bit binary output encoder, and Table 5.2 shows the truth table of the encode function. However, an encoder works only as expected when the input signal is one-hot coded. For all other input values, the output is undefined. As we cannot describe a function with undefined outputs, we use a default assignment that catches all undefined input patterns.

The following Chisel code assigns a default value of 0 and then uses the switch statement for the legal input values.

```
b := "b00".U
```

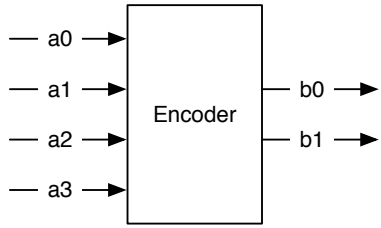


Figure 5.3: A 4-bit to 2-bit encoder.

a	b
0001	00
0010	01
0100	10
1000	11
????	??

Table 5.2: Truth table for a 4 to 2 encoder.

```
switch (a) {
  is ("b0001".U) { b := "b00".U}
  is ("b0010".U) { b := "b01".U}
  is ("b0100".U) { b := "b10".U}
  is ("b1000".U) { b := "b11".U}
}
```

For the decoder, we found an elegant single-line statement to express the logic. This also enables us to describe a wide decoder. However, we are not aware of such an expression for the encoder.

We need to write a (simple) hardware generator to express larger encoders. Therefore, we need to introduce the Scala loop construct. The following two lines of Scala code express a loop, counting from `0` to `9`.

```
// Loops i from 0 to 9
for (i <- 0 until 10) {
  // use i to index into a Wire or Vec
}
```

The loop variable `i` can be used to index individual bits from a `Wire` or `Reg`; or an element in a `Vec`. This Scala generator loop is the simplest form of describing a hardware generator. Chapter 10 describes how to write hardware generators in more detail. Note that the loop is executed at circuit generation time. This is not a hardware counter.

For the encoder generator, we will use a `Vec`, where each element represents one column of the encoder table. The following code shows a 16-bit encoder, where the output is 4 bits wide:

```
val v = Wire(Vec(16, UInt(4.W)))
v(0) := 0.U
for (i <- 1 until 16) {
  v(i) := Mux(hotIn(i), i.U, 0.U) | v(i - 1)
}
val encOut = v(15)
```

The input of the encoder is `hotIn` and the output is `encOut`. `Vec` element `0` is the default case (`0`), and also represents the output value when the least significant bit (LSB) is set in `hotIn`.

`Vec` elements 1 till 15 are connected to a multiplexer. If the bit at position `i` is set in `hotIn`, the multiplexer output is the index, otherwise it is 0. For the correct behavior of our encoder, we assume that the input signal is one-hot encoded. Finally we need to merge all vector elements for a single output. As the vector elements are `0` when the corresponding bit in the input is `0`, we can simply combine all elements with an OR function. In the loop, we OR the current element with one vector element before (`... | v(i-1)`). When several elements are combined with a function, we call this operation also reduce. Therefore, here we perform an OR reduction.

5.4 Arbiter

We use an arbiter to arbitrate requests from several clients to a shared resource. An example would be several processor cores sharing a single serial port (UART).

Figure 5.4 shows the schematic of a 4-bit arbiter. It consists of four request lines (`r0`–`r3`) and four grant lines (`g0`–`g3`). The arbiter grants only a single request. For example, a request input of `0101` will result in a grant output of `0001`. The arbiter prioritizes the lower inputs. Therefore, we call it a priority arbiter. The lower the bit number, the higher the priority.

To build a fair arbiter, we need to add state to remember the last arbitration. We

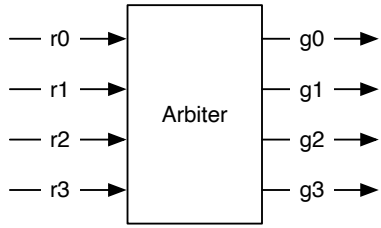


Figure 5.4: A symbol for a 4-bit arbiter.

will present a fair arbiter in Section 10.6.2.

Figure 5.5 shows the schematic of a 4-bit arbiter. The individual grant requests must check if a lower bit has already won the arbitration. For the first request, the grant `g0` depends only on the request `r0`. The second grant can only win the arbitration when request `r1` is asserted and request `r0` is deasserted. For the next requests, the lookup is further chained.

The following code shows an arbiter for 3 clients.

```
val grant = VecInit(false.B, false.B, false.B)
val notGranted = VecInit(false.B, false.B)

grant(0) := request(0)
notGranted(0) := !grant(0)
grant(1) := request(1) && notGranted(0)
notGranted(1) := !grant(1) && notGranted(0)
grant(2) := request(2) && notGranted(1)
```

The code is the same as the schematic in Figure 5.5 (except we show the code only for a 3-bit arbiter). We use vectors of `Bool` to represent the request, grant, and not-granted chain. We can see that `grant(0)` depends only on `request(0)`. `notGranted` is used to chain the information that no lower-bit requests have been granted.

Small arbiters can also be directly described with a logic table. The following code shows the table for a 3-bit arbiter.

```
val grant = WireDefault("b0000".U(3.W))
switch (request) {
  is ("b000".U) { grant := "b000".U}
  is ("b001".U) { grant := "b001".U}
  is ("b010".U) { grant := "b010".U}
```

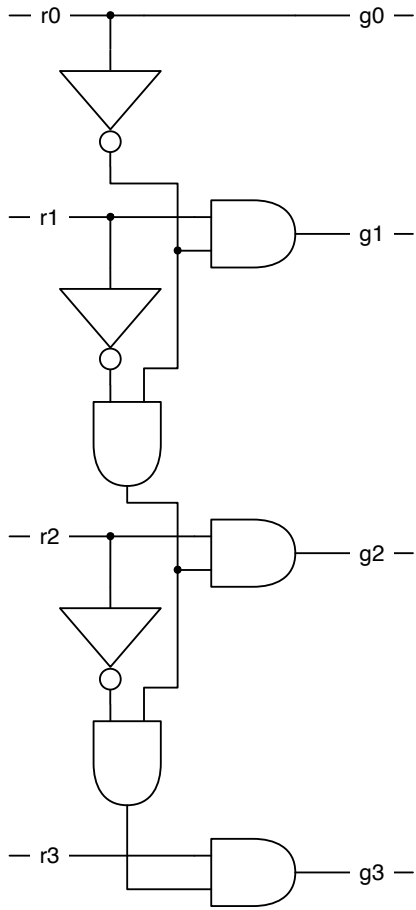


Figure 5.5: A 4-bit arbiter.

```
    is ("b011".U) { grant := "b001".U}
    is ("b100".U) { grant := "b100".U}
    is ("b101".U) { grant := "b001".U}
    is ("b110".U) { grant := "b010".U}
    is ("b111".U) { grant := "b001".U}
  }
```

TODO: Maybe also use the pattern matching with ??? in Chisel

However, for larger arbitration circuits, we will use our newly learned trick of a `for` loop as a generator loop. The following code shows a parameterized arbiter for `n` requests and grants. Here we use again `Vec` of `Bool`.

```
val grant = VecInit.fill(n)(false.B)
val notGranted = VecInit.fill(n)(false.B)

grant(0) := request(0)
notGranted(0) := !grant(0)
for (i <- 1 until n) {
  grant(i) := request(i) && notGranted(i-1)
  notGranted(i) := !grant(i) && notGranted(i-1)
}
```

The code shown above is the loop version of the initial arbiter version. It generates the arbitration circuit for `n` requests. The small difference to the manual version (unrolled loop) is that we generate a `notGranted` wire also for the last request (`n - 1`). That wire is not used and the synthesize tool will optimize it away.

5.5 Priority Encoder

With our original encoder design, we had to assume that the input was one-hot encoded, meaning only one bit is allowed to be `1`. Inputs with several bits set are illegal and lead to undefined behavior.

We can solve this problem by combining the encoder with an arbitration circuit, which selects only the highest-priority bit set. When we feed the output of the arbiter into an encoder, we create a priority encoder. Figure 5.6 shows the schematic.

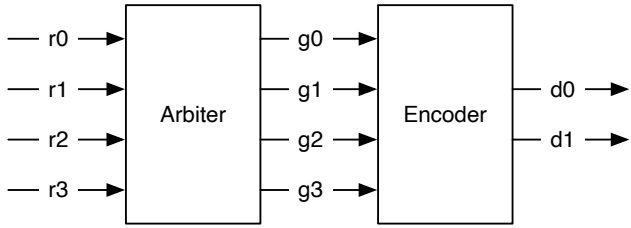


Figure 5.6: With an arbiter and an encoder, we can build a priority encoder.

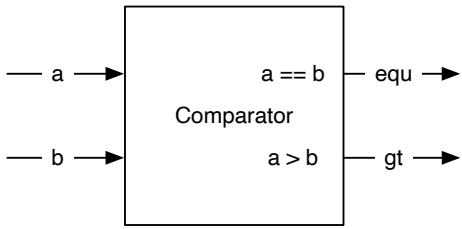


Figure 5.7: A simple comparator.

5.6 Comparator

As the last circuit for the chapter on combinational building blocks, we present the comparator. Figure 5.7 shows the schematic of a comparator. It has two multi-bit inputs and compares those two values. It has two outputs: (1) signaling that `a` and `b` are equal (`equ`) and (2) that `a` is greater than `b`. These two outputs are enough for all possible comparisons. For example, if `equ` or `gt` are asserted, we know that the condition $a >= b$ is true. For the condition $a <= b$, we test for not `gt`.

The following code snippet shows the comparator. As you can see, these are just two lines of Chisel code. Therefore, compare functions are usually directly used in other components and not wrapped into a module.

```
val equ = a === b
val gt = a > b
```

5.7 Exercise

Describe a combinational circuit to convert a 4-bit binary input to the encoding of a 7-segment display. You can either define the codes for the decimal digits, which was the initial usage of a 7-segment display, or additionally, define encodings for the remaining bit pattern to be able to display all 16 values of a single digit in hexadecimal. When you have an FPGA board with a 7-segment display, connect four switches or buttons to the input of your circuit and the output to the 7-segment display.

6 Sequential Building Blocks

Sequential circuits are circuits where the output depends on the input *and* previous values. As we are interested in synchronous design (clocked designs), we mean synchronous sequential circuits when we talk about sequential circuits.[1] To build sequential circuits, we need elements that can store state: the so-called registers.

6.1 Registers

The fundamental elements for building sequential circuits are registers. A register is a collection of D flip-flops. A D flip-flop captures the value of its input at the rising edge of the clock and stores it at its output. In other words, the register updates its output with the value of the input on the rising edge of the clock.

Figure 6.1 shows the schematic symbol of a register. It contains an input `D` and an output `Q`. Each register also contains an input for a `clock` signal. This global clock signal is connected to all registers in a synchronous circuit, so it is usually not drawn in the schematics. The little triangle on the bottom of the box symbolizes the clock input and tells us this is a register. We omit the clock signal in the following

[1] We can also build sequential circuits with asynchronous logic and feedback, but this is a niche topic and cannot be easily expressed in Chisel.

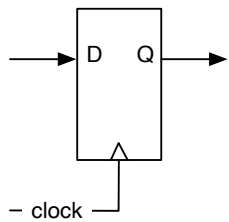


Figure 6.1: A D flip-flop-based register.

schematics. The omission of the global clock signal is also reflected by Chisel, where no explicit connection of a signal to the register's clock input is needed. In Chisel a register with input `d` and output `q` is defined with:

```
val q = RegNext(d)
```

Note that we do not need to connect a clock to the register; Chisel implicitly does this. A register's input and output can be arbitrarily complex types made from a combination of vectors and bundles.

A register can also be defined and used in two steps:

```
val delayReg = Reg(UInt(4.W))

delayReg := delayIn
```

First, we define the register and give it a name. Second, we connect the signal `delayIn` to the input of the register. Note also that the name of the register contains the string `Reg`. To easily distinguish between combinational and sequential circuits, it is common practice to have the marker `Reg` as part of the name.

A register can be initialized on reset. The `reset` signal is, like the `clock` signal, implicit in Chisel. We supply the reset value, for example, zero, as a parameter to the register constructor `RegInit`. The input for the register is connected with a Chisel assignment statement.

```
val valReg = RegInit(0.U(4.W))

valReg := inVal
```

The default reset implementation in Chisel is a synchronous reset.[2] For a synchronous reset, no change is needed in the D flip-flop itself; instead, a multiplexer must be added to the input that selects between the initialization value under reset and the data value. Figure 6.2 shows the schematic of a register with a synchronous reset where the reset drives the multiplexer. However, because synchronous reset is used quite often, modern FPGA flip-flops contain a synchronous reset (and set) input to the flip-flop to avoid wasting LUT resources for the multiplexer.

Sequential circuits change their value over time. Therefore, their behavior can be described by a diagram showing the signals over time. Such a diagram is called a waveform or timing diagram.

[2] Support for asynchronous reset is available.

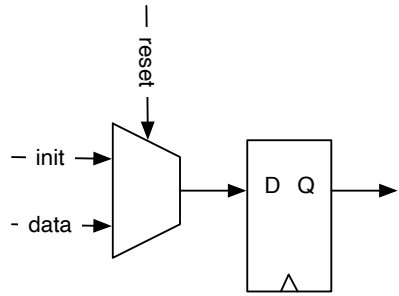


Figure 6.2: A D flip-flop based register with a synchronous reset.

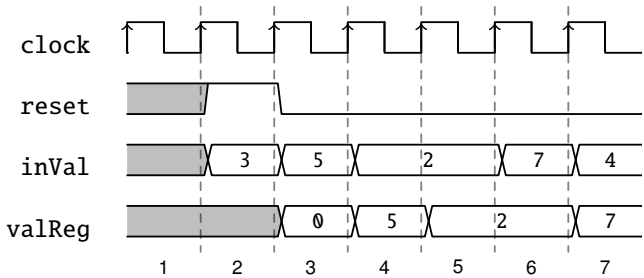


Figure 6.3: A waveform diagram for a register with a reset.

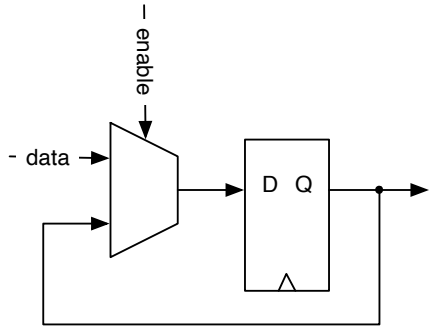


Figure 6.4: A D flip-flop-based register with an enable signal.

Figure 6.3 shows a waveform for the register with a reset and some input data applied to it. Time advances from left to right. On top of the figure, we see the clock that drives our circuit. In the first clock cycle (1), the register content is undefined before a reset. in the second clock cycle, reset is asserted high, and on the rising edge of this clock cycle, the register takes the initial value `0`. Input `inVal` is ignored. In the next clock cycle, `reset` is `0`, and the value of `inVal` is captured on the next rising edge. From then on, `reset` stays `0`, as it should be, and the register output follows the input signal with one clock cycle delay.

Waveforms are an excellent tool to specify the behavior of a circuit graphically. Timing diagrams are convenient, especially in more complex circuits where many operations happen in parallel and data moves pipelined through the circuit. Chisel testers can also produce waveforms during testing that can be displayed with a waveform viewer and used for debugging.

A common design pattern is a register with an enable signal. When the enable signal is `true` (high) the register captures the input; otherwise, it keeps its old value. The enable can be implemented, similar to the synchronous reset, with a multiplexer at the input of the register. One input to the multiplexer is the feedback of the output of the register.

Figure 6.4 shows the schematic of a register with an enable signal. As this is also a common design pattern, modern FPGA flip-flops contain a dedicated enable input, and no additional (LUT) resources are needed to implement the register enable.

Figure 6.5 shows an example waveform for a register with enable. Most of the time, `enable` is high (`true`) and the register follows the input with one clock cycle delay. Only in the fourth clock cycle `enable` is low, and the register keeps its value

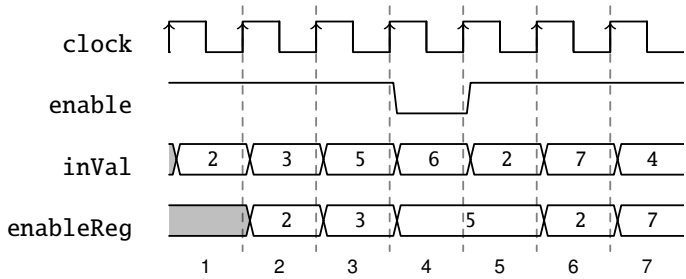


Figure 6.5: A waveform diagram for a register with an enable signal.

(5) in clock cycle 5.

A register with an enable can be described in a few lines of Chisel code with a conditional update:

```
val enableReg = Reg(UInt(4.W))

when (enable) {
  enableReg := inVal
}
```

Using an enable signal for a register is so common that Chisel defines `RegEnable` where the second parameter is the enable signal:

```
val enableReg2 = RegEnable(inVal, enable)
```

A register with enable can also be reset:

```
val resetEnableReg = RegInit(0.U(4.W))

when (enable) {
  resetEnableReg := inVal
}
```

The functionality of register initialization at reset and enable can be combined when using the three-parameter version of `RegEnable`. The first parameter is the input signal, the second parameter is the initialization value, and the third parameter is the enable signal:

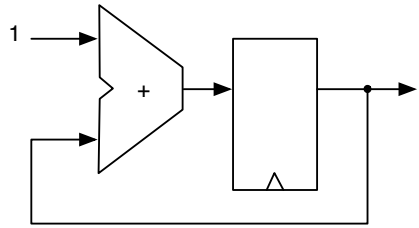


Figure 6.6: An adder and a register result in counter.

```
val resetEnableReg2 = RegEnable(inVal, 0.U(4.W), enable)
```

A register can also be part of an expression without giving it a name. The following circuit detects the rising edge of a signal by comparing its current value with the one from the last clock cycle (the delayed value).

```
val risingEdge = din & !RegNext(din)
```

Now that we have explored all the basic register uses, we put those registers to good use and build more interesting sequential circuits. For the next schematics, we will further simplify the register symbol and omit the `D` for the input and the `Q` for the output.

6.2 Counters

One of the most basic sequential circuits is a counter. In its simplest form, a counter is a register where the output is connected to an adder, and the adder's output is connected to the input of the register. Figure 6.6 shows such a free-running counter.

A free-running counter with a 4-bit register counts from 0 to 15 and then wraps around to 0 again. A counter shall also be reset to a known value.

```
val cntReg = RegInit(0.U(4.W))

cntReg := cntReg + 1.U
```

When we want to count events, we use a condition to increment the counter, as shown in Figure 6.7 and in the following code.

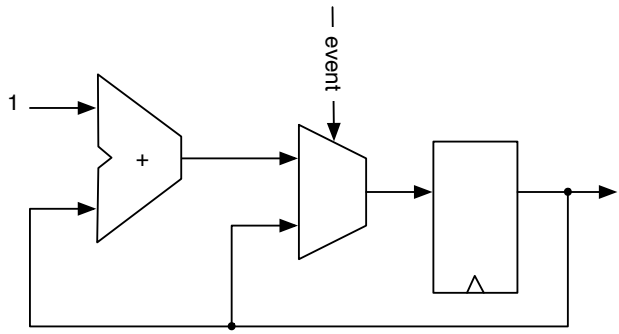


Figure 6.7: Counting events.

```
val cntEventsReg = RegInit(0.U(4.W))
when(event) {
  cntEventsReg := cntEventsReg + 1.U
}
```

6.2.1 Counting Up and Down

To count up to a value and then restart with `0`, we need to compare the counter value with a maximum constant (`N` in the following examples), for example, with a `when` conditional statement.

```
val cntReg = RegInit(0.U(8.W))

cntReg := cntReg + 1.U
when(cntReg === N) {
  cntReg := 0.U
}
```

We can also use a multiplexer for our counter:

```
val cntReg = RegInit(0.U(8.W))

cntReg := Mux(cntReg === N, 0.U, cntReg + 1.U)
```

If we are in the mood of counting down, we start by resetting the counter register

with the maximum value and reset the counter to that value when reaching 0.

```
val cntReg = RegInit(N)

cntReg := cntReg - 1.U
when(cntReg === 0.U) {
  cntReg := N
}
```

As we are coding and using more counters, we can define a function with a parameter to generate a counter for us.

```
// This function returns a counter
def genCounter(n: Int) = {
  val cntReg = RegInit(0.U(8.W))
  cntReg := Mux(cntReg === n.U, 0.U, cntReg + 1.U)
  cntReg
}

// now we can easily create many counters
val count10 = genCounter(10)
val count99 = genCounter(99)
```

The last statement of the function `genCounter` is the return value of the function, in this example, the output of the counting register `cntReg`.

Note that in all the examples, our counter had values between `0` and `N`, including `N`. To count 10 clock cycles, we need to set `N` to 9. Setting `N` to 10 would be a classic example of an off-by-one error.

6.2.2 Generating Timing with Counters

Besides counting events, counters are often used to generate a notion of time (time as wall clock time). A synchronous circuit runs with a clock with a fixed frequency. The circuit proceeds in those clock ticks. There is no notion of time in a digital circuit other than counting clock ticks. If we know the clock frequency, we can generate circuits that generate timed events, such as blinking an LED at some frequency, as shown in the Chisel "Hello World" example.

A common practice is to generate single-cycle *ticks* with a frequency f_{tick} that we need in our circuit. That tick occurs every n clock cycle, where $n = f_{clock}/f_{tick}$ and the tick is precisely one clock cycle long. This tick is *not* used as a derived

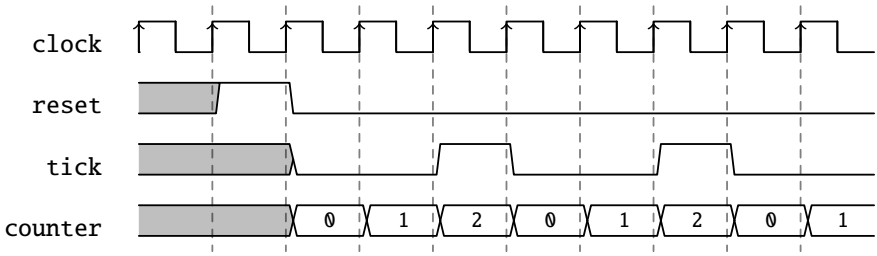


Figure 6.8: A waveform diagram for generating a slow frequency tick.

clock, but as an enable signal for registers in the circuit that shall logically operate at frequency f_{tick}. Figure 6.8 shows an example of a tick generated every 3 clock cycles.

In the following circuit, we describe a counter that counts from `0` to the maximum value of `N - 1`. When the maximum value is reached, the `tick` is `true` for a single cycle, and the counter is reset to `0`. When we count from `0` to `N - 1`, we generate one logical tick every `N` clock cycles.

```
val tickCounterReg = RegInit(0.U(32.W))
val tick = tickCounterReg === (N-1).U

tickCounterReg := tickCounterReg + 1.U
when (tick) {
  tickCounterReg := 0.U
}
```

This logical timing of one tick every n clock cycles can then be used to advance other parts of our circuit with this slower, logical clock. In the following code, we use just another counter that increments by `1` every n clock cycles.

```
val lowFrequCntReg = RegInit(0.U(4.W))
when (tick) {
  lowFrequCntReg := lowFrequCntReg + 1.U
}
```

Figure 6.9 shows the waveform of the tick and the slow counter that increments every tick (n clock cycles).

Examples of the usage of this slower *logical* clock are: blinking an LED, gen-

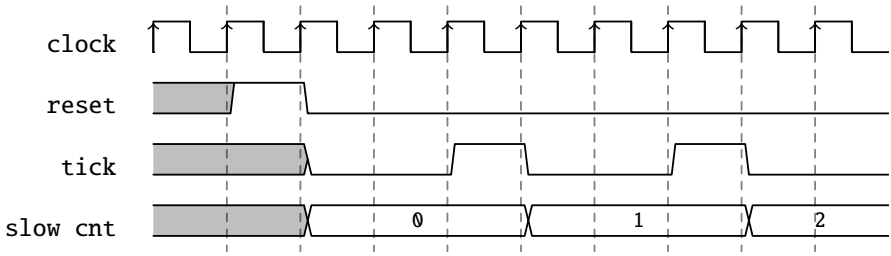


Figure 6.9: Using the slow frequency tick.

erating the baud rate for a serial bus,[3] generating signals for 7-segment display multiplexing, and subsampling input values for debouncing buttons and switches.

Although width inference should size the registers, it is better to explicitly specify the width with the type at the register definition or with the initialization value. Explicit width definition can avoid surprises when a reset value of `0.U` results in a counter with a width of a single bit.

6.2.3 The Nerd Counter

Sometimes, some of us feel like being a nerd. For example, we want to design a highly optimized version of our counter/tick generation. A standard counter needs the following resources: one register, one adder (or subtractor), and a comparator. We cannot do much about the register or the adder. If we count up, we need to compare against a number, which is a bit string. The comparator can be built from inverters for the zeros in the bit string and a large AND gate. When counting down to zero, the comparator is a large NOR gate, which might be slightly cheaper than the comparator against a constant in an ASIC. In an FPGA, where logic is built out of lookup tables, there is no difference between comparing against 0 or 1. The resource requirement is the same for the up and down counter.

However, there is still one more trick a clever hardware designer can pull off. Counting up or down needed a comparison against all counting bits so far. What if we count from N-2 down to -1? A negative number has the most significant bit set to 1, and a positive number has this to 0. We must only check this bit to detect that our counter reached -1. Here it is, the counter created by a nerd:

[3] *Baud rate* is a measure of information transmission speed, often in bits per second. It must be equivalent to the transmitter and receiver.

```
val MAX = (N - 2).S(8.W)
val cntReg = RegInit(MAX)
io.tick := false.B

cntReg := cntReg - 1.S
when(cntReg(7)) {
  cntReg := MAX
  io.tick := true.B
}
```

6.2.4 A Timer

Another form of timer we can design is a one-shot timer. A one-shot timer is like a kitchen timer: you set the number of minutes and press start. When the specified amount of time has elapsed, the alarm sounds. The digital timer is loaded with the time in clock cycles. Then it counts down until reaching zero. At zero, the timer asserts *done*.

Figure 6.10 shows the block diagram of a timer. The register can be loaded with the value of `din` by asserting `load`. When the `load` signal is deasserted, counting down is selected (by selecting `cntReg - 1` as the input for the register). When the counter reaches `0`, the signal `done` is asserted, and the counter stops counting by selecting the `0` input of the multiplexer.

Listing 6.1 shows the Chisel code for the timer. We use an 8-bit register `cntReg` that is reset to `0`. The boolean value `done` is the result of comparing `cntReg` with `0`. For the input multiplexer, we introduce the wire `next` with a default value of `0`. The `when/elsewhen` block introduces the other two inputs with the select function. The signal `load` has priority over the decrement selection. The last line connects the multiplexer, represented by `next`, to the input of the register `cntReg`.

If we aim for a bit more concise code, we can directly assign the multiplexer values to the register `reg`, instead of using the intermediate wire `next`.

6.2.5 Pulse-Width Modulation

Pulse-width modulation (PWM) is a signal with a constant period and a modulation of the time the signal is *high* within that period.

Figure 6.11 shows a PWM signal. The arrows point to the start of the periods of the signal. The percentage of time the signal is high is called the duty cycle. In the

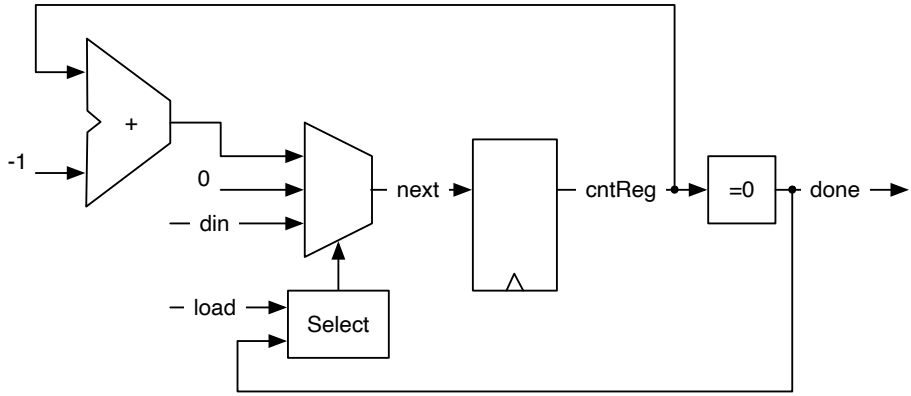


Figure 6.10: A one-shot timer.

```
val cntReg = RegInit(0.U(8.W))
val done = cntReg === 0.U

val next = WireDefault(0.U)
when (load) {
  next := din
} .elsewhen (!done) {
  next := cntReg - 1.U
}
cntReg := next
```

Listing 6.1: A one-shot timer

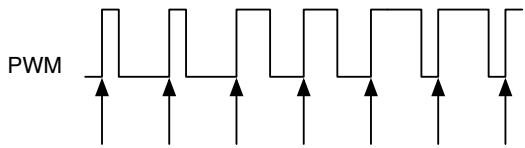


Figure 6.11: Pulse-width modulation.

first two periods, the duty cycle is 25 %; in the next two, 50 %, and in the last two cycles, it is 75 %. The pulse width is modulated between 25 % and 75 %.

Adding a low-pass filter to a PWM signal results in a simple digital-to-analog converter. The low-pass filter can be as simple as a resistor and a capacitor.

The following code example will generate a waveform with 3 clock cycles high every 10 clock cycles.

```
def pwm(nrCycles: Int, din: UInt) = {
  val cntReg =
      RegInit(0.U(unsignedBitLength(nrCycles-1).W))
  cntReg := Mux(cntReg === (nrCycles-1).U, 0.U, cntReg +
      1.U)
  din > cntReg
}

val din = 3.U
val dout = pwm(10, din)
```

We use a function for the PWM generator to provide a reusable, lightweight component. The function has two parameters: a Scala integer configuring the PWM with the number of clock cycles (`nrCycles`) and a Chisel wire (`din`) that gives the duty cycle (pulse width) for the PWM output signal. We use a multiplexer in this example to express the counter. The last line of the function compares the counter value with the input value `din` to return the PWM signal. The last expression in a Chisel function is the return value, the wire connected to the compare function.

We use the function `unsignedBitLength(n)` to specify the number of bits for the counter `cntReg` needed to represent unsigned numbers up to (and including) `n`.[4] Chisel also has a function `signedBitLength` to provide the number of bits for a signed representation of a number.

One application of a PWM signal is to dim an LED. In that case, the eye serves as a low-pass filter. We expand the above example to drive the PWM generation by a triangular function. The result is an LED with continuously changing intensity.

```
val FREQ = 100000000 // a 100 MHz clock input
val MAX = FREQ/1000  // 1 kHz

val modulationReg = RegInit(0.U(32.W))
```

[4] The number of bits to represent an unsigned number n in binary is $\lfloor log_2(n) \rfloor + 1$.

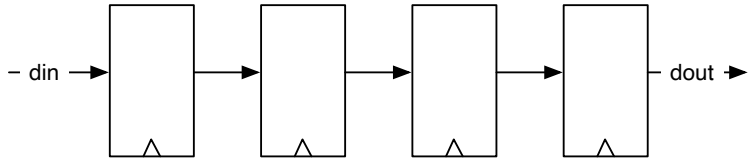


Figure 6.12: A 4-stage shift register.

```
val upReg = RegInit(true.B)

when (modulationReg < FREQ.U && upReg) {
  modulationReg := modulationReg + 1.U
} .elsewhen (modulationReg === FREQ.U && upReg) {
  upReg := false.B
} .elsewhen (modulationReg > 0.U && !upReg) {
  modulationReg := modulationReg - 1.U
} .otherwise { // 0
  upReg := true.B
}

// divide modReg by 1024 (about the 1 kHz)
val sig = pwm(MAX, modulationReg >> 10)
```

We use two registers for the modulation: (1) `modulationReg` for counting up and down and (2) `upReg` as a flag to determine if we shall count up or down. We count up to the frequency of our clock input (100 MHz in our example), which results in a signal of 0.5 Hz. The lengthy `when/.elsewhen/.otherwise` expression handles the up- or down-counting and the direction switch.

As our PWM counts only up to the 1000th of the frequency to generate a 1 kHz signal, we need to divide the modulation signal by 1000. As real division is very expensive in hardware, we simply shift by 10 to the right, which equates to a division by $2^{10} = 1024$. As we have defined the PWM circuit as a function, we can simply instantiate that circuit with a function call. Wire `sig` represents the modulated PWM signal.

6.3 Shift Registers

A shift register is a collection of flip-flops connected in a sequence. Each output of a flip-flop is connected to the input of the next flip-flop. Figure 6.12 shows a 4-stage shift register. The circuit *shifts* the data from left to right on each clock tick. In this simple form the circuit implements a 4 cycle delay from `din` to `dout`.

The Chisel code for this simple shift register (1) creates a 4-bit register `shiftReg`; (2) concatenates the lower 3 bits of the shift register with the input `din` for the next input to the register; and (3) uses the most significant bit (MSB) of the register as the output `dout`.

```
val shiftReg = Reg(UInt(4.W))
shiftReg := shiftReg(2, 0) ## din
val dout = shiftReg(3)
```

Shift registers are often used to convert from serial data to parallel data or from parallel data to serial data. Section 11.2 shows a serial port that uses shift registers for the receive and send functions.

6.3.1 Shift Register with Parallel Output

A serial-in parallel-out configuration of a shift register transforms a serial input stream into parallel words. This may be used in a serial port (UART) for the receive function. Figure 6.13 shows a 4-bit shift register, where each flip-flop output is connected to one output bit. After four clock cycles, this circuit converts a 4-bit serial data word to a 4-bit parallel data word available in `q`. In this example, we assume that bit 0 (the least significant bit) is sent first and therefore arrives in the last stage when we want to read the full word.

In the following Chisel code, we initialize the shift register `outReg` with 0. Then we shift in from the MSB, which means a right shift. The parallel result, `q`, is just the reading of the register `outReg`.

```
val outReg = RegInit(0.U(4.W))
outReg := serIn ## outReg(3, 1)
val q = outReg
```

Figure 6.13 shows a 4-bit shift register with a parallel output function.

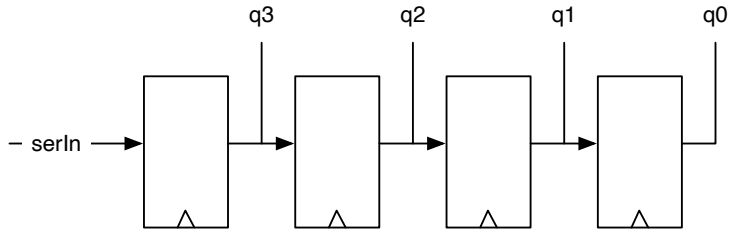


Figure 6.13: A 4-bit shift register with parallel output.

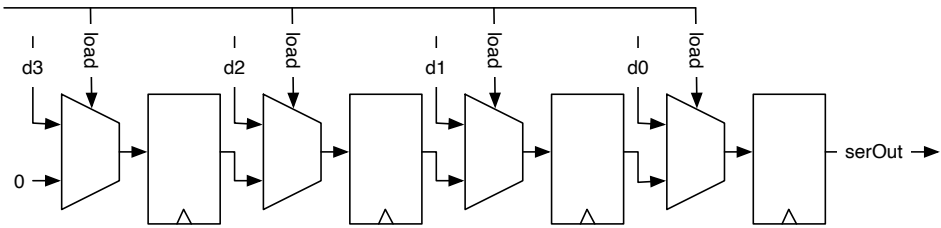


Figure 6.14: A 4-bit shift register with parallel load.

6.3.2 Shift Register with Parallel Load

A parallel-in serial-out configuration of a shift register transforms a parallel input stream of words (bytes) into a serial output stream. This may be used in a serial port (UART) for the transmit function.

Figure 6.14 shows a 4-bit shift register with a parallel load function. The Chisel description of that function is relatively straightforward:

```
val loadReg = RegInit(0.U(4.W))
when (load) {
  loadReg := d
} otherwise  {
  loadReg := 0.U ## loadReg(3, 1)
}
val serOut = loadReg(0)
```

Note that we are now shifting to the right, filling in zeros at the MSB.

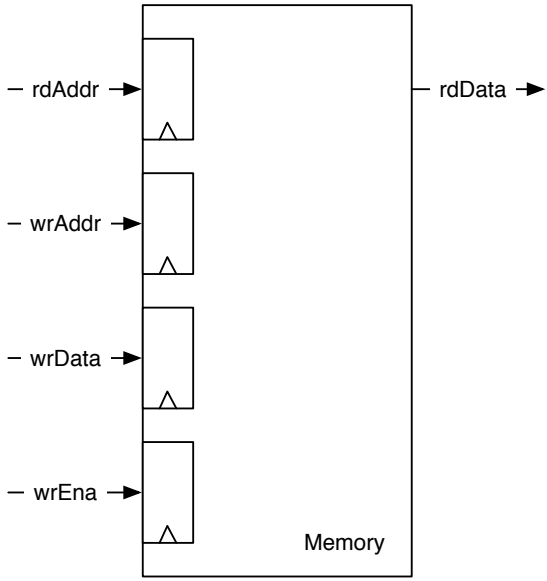


Figure 6.15: A synchronous memory.

6.4 Memory

A memory can be built out of a collection of registers, in Chisel a `Reg` of a `Vec`. However, this is expensive in hardware, and larger memory structures are built as SRAM. For an ASIC, a memory compiler constructs memories. FPGAs contain on-chip memory blocks, also called block RAMs. Those on-chip memory blocks can be combined for larger memories. Memories in an FPGA usually have one read and one write port or two ports that can be switched between read and write at runtime.

FPGAs (and also ASICs) usually support synchronous memories. Synchronous memories have registers on their inputs (read and write address, write data, and write enable). That means the read data is available one clock cycle after setting the address.

Figure 6.15 shows the schematics of such a synchronous memory. The memory is dual-ported with one read port and one write port. The read port has a single input, the read address (`rdAddr`), and one output, the read data (`rdData`). The write port has three inputs: the address (`wrAddr`), the data to be written (`wrData`), and a

```
class Memory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(8.W))
    val wrEna = Input(Bool())
  })

  val mem = SyncReadMem(1024, UInt(8.W))

  io.rdData := mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }
}
```

Listing 6.2: 1 KiB of synchronous memory.

write enable (`wrEna`). Note that for all inputs, there is a register within the memory showing the synchronous behavior.

To support on-chip memory, Chisel provides the memory constructor `SyncReadMem`. Listing 6.2 shows a component `Memory` that implements 1 KiB of memory with byte-wide input and output data and a write enable.

An interesting question is which value is returned from a read when in the same clock cycle, a new value is written to the same address that is read out. We are interested in the read-during-write behavior of the memory. There are three possibilities: the newly written value, the old value, or undefined (which might be a mix of some bits from the old value and some of the newly written data). Which possibility is available in an FPGA depends on the FPGA type and sometimes can be specified. Chisel documents that the read data is undefined.

If we want to read out the newly written value, we can build a forwarding circuit that detects that the addresses are equal and *forwards* the write data. Figure 6.16 shows the memory with the forwarding circuit. Read and write addresses are compared and gated with the write enable to select between the forwarding path of the write data or the memory read data. The write data is delayed by one clock cycle

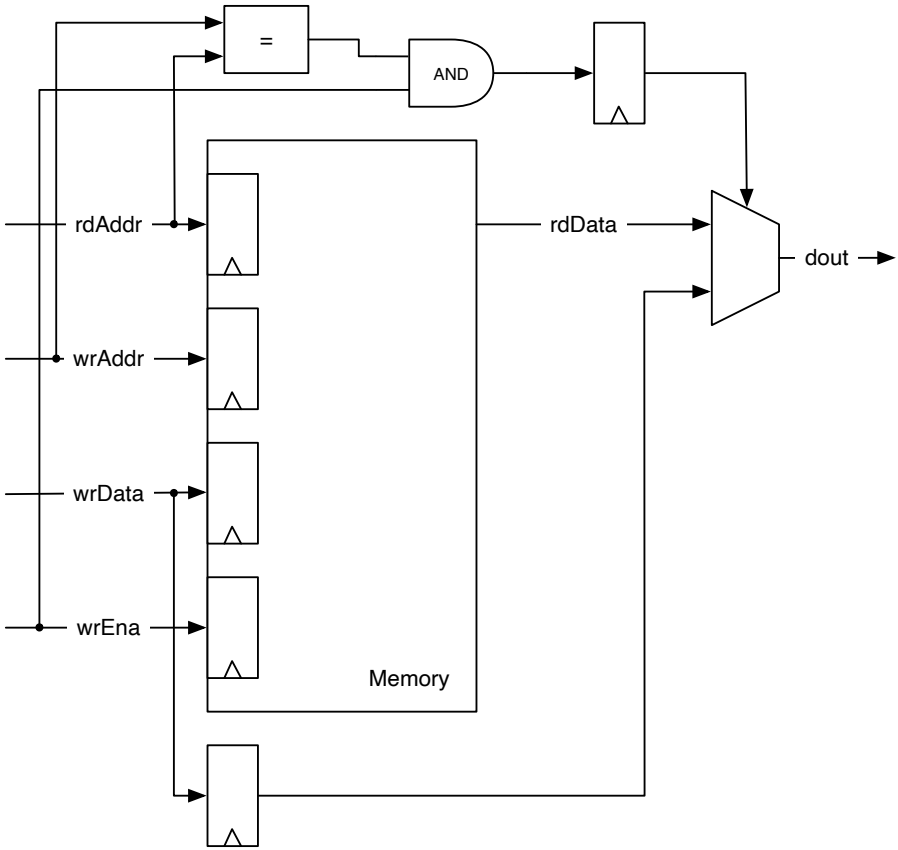


Figure 6.16: A synchronous memory with forwarding for a defined read-during-write behavior.

```
class ForwardingMemory() extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(10.W))
    val rdData = Output(UInt(8.W))
    val wrAddr = Input(UInt(10.W))
    val wrData = Input(UInt(8.W))
    val wrEna = Input(Bool())
  })

  val mem = SyncReadMem(1024, UInt(8.W))

  val wrDataReg = RegNext(io.wrData)
  val doForwardReg = RegNext(io.wrAddr === io.rdAddr &&
      io.wrEna)

  val memData = mem.read(io.rdAddr)

  when(io.wrEna) {
    mem.write(io.wrAddr, io.wrData)
  }

  io.rdData := Mux(doForwardReg, wrDataReg, memData)
}
```

Listing 6.3: A memory with a forwarding circuit.

with a register.

Listing 6.3 shows the Chisel code for a synchronous memory including the forwarding circuit. We need to store the written data in a register (`wrDataReg`) to be available in the next clock cycle to fit the synchronous memory that also provides the read value in the next clock cycle. We compare the two input addresses (`wrAddr` and `rdAddr`) and check if `wrEna` is true for the forwarding condition. That condition is also delayed by one clock cycle. A multiplexer selects between the forwarding (write) data or the read data from memory.

As this pattern is common, Chisel provides an optional parameter in `SyncReadMem` to define the read-during-write behavior. `WriteFirst` generates Verilog code that includes the forwarding if needed. The other two options are `ReadFirst` and `Undefined`.

```
val hello = "Hello, World!"
val helloHex =
    hello.map(_.toInt.toHexString).mkString("\n")
val file = new java.io.PrintWriter("hello.hex")
file.write(helloHex)
file.close()

val mem = SyncReadMem(1024, UInt(8.W))
loadMemoryFromFileInline(mem, "hello.hex",
    firrtl.annotations.MemoryLoadFileType.Hex)
```

Listing 6.4: Memory initialization.

```
val mem = SyncReadMem(1024, UInt(8.W),
    SyncReadMem.WriteFirst)
```

Memories in FPGAs can be initialized with either binary or hexadecimal initialization files. The files are simple ASCII text files with the same number of lines as there are entries in the corresponding memory. Each character represents either a single bit or four bits. Traditionally, binary files use the `.bin` file extension, while hexadecimal files use `.hex`. Using `loadMemoryFromFile` will result in the emission of a separate Verilog file and works in ChiselTest. Initializations are based on calls to `readmemb` or `readmemh`.

To initialize on-chip memory from Scala during generation time, we need to first write the content into a file and then use `loadMemoryFromFile`. Listing 6.4 shows an example of initializing a memory with a string.

Chisel also provides `Mem`, representing a memory with synchronous write and an asynchronous read. As this memory type is usually not directly available in an FPGA, the synthesize tool will build it out of flip-flops. Therefore, we recommend using `SyncReadMem`. If asynchronous read behavior is needed and the resources are available in the FPGA you are using (e.g., as a LUT RAM on Xilinx FPGAs), you can manually implement this as a `BlackBox`. Vendors typically provide code templates that can be used directly for this.

6.5 Exercises

Use the 7-segment encoder from the last exercise and add a 4-bit counter as input to switch the display from `0` to `F`. When you directly connect this counter to the clock of the FPGA board, you will see all 16 numbers overlapped (all 7 segments will light up). Therefore, you need to slow down the counting. Create another counter that can generate a single-cycle *tick* signal every 500 milliseconds. Use that signal as enable signal for the 4-bit counter.

Construct a PWM waveform with a generator function and set the threshold with a function (triangular or a sine function). A triangular function can be created by counting up and down. A sine function can be created with a lookup table that you can generate with a few lines of Scala code (see Section 10.3). Drive an LED on an FPGA board with that modulated PWM function. What frequency shall your PWM signal be? What frequency is the driver running?

Digital designs are often sketched as a circuit on paper. Not all details need to be shown. We use block diagrams, like in the figures in this book. It is an important skill to be able to fluently translate between a schematic representation of the circuit and a Chisel description. Sketch the block diagram for the following circuits:

```
val dout = WireDefault(0.U)

switch(sel) {
  is(0.U) { dout := 0.U }
  is(1.U) { dout := 11.U }
  is(2.U) { dout := 22.U }
  is(3.U) { dout := 33.U }
  is(4.U) { dout := 44.U }
  is(5.U) { dout := 55.U }
}
```

Here is a slightly more complex circuit containing a register:

```
val regAcc = RegInit(0.U(8.W))

switch(sel) {
  is(0.U) { regAcc := regAcc}
  is(1.U) { regAcc := 0.U}
  is(2.U) { regAcc := regAcc + din}
  is(3.U) { regAcc := regAcc - din}
}
```

As an advanced exercise, try using an on-chip memory. Instantiate a `SyncReadMem` and create two communicating state machines. The first state machine writes a string into the memory. When done, it starts the second state machine that reads back the string from the memory. Test with simple `printf` statements in the reading state machine. Be careful in the reading that the read value comes one clock cycle later than applying the address to the memory. You can extend that example by writing a Chisel test to test the memory.

Sometimes, one would like to download the content of a memory from a laptop, e.g., when building a processor to load a program. Assume you have a serial port (see Section 11.2) that connects your FPGA board to your laptop. Can you envision a protocol on the serial port with a state machine in the FPGA to download memory content into the FPGA after configuring it? Downloading at runtime also avoids synthesizing for the FPGA again after the memory content changes.

7 Input Processing

Input signals from the external world into our synchronous circuit are usually not synchronous to the clock; they are asynchronous. An input signal may come from a source that does not have a clean transition from 0 to 1 or 1 to 0. An example is a bouncing button or switch. Input signals may be noisy with spikes that could trigger a transition in our synchronous circuit. This chapter describes circuits that deal with such input conditions.

The latter two issues, debouncing switches and filtering noise, can also be solved with external, analog components. However, it is more cost-efficient to deal with those issues in the digital domain.

7.1 Asynchronous Input

Input signals that are not synchronous to the system clock are called asynchronous signals. Those signals may violate the setup and hold time of the input of a flip-flop. This violation may result in metastability of the flip-flop. The metastability may result in an output value between 0 and 1, or it may result in oscillation. However, after some time, the flip-flop will stabilize at 0 or 1.

Another common issue with external, asynchronous input signals is when that signal changes close to the rising clock edge and is used in more than one place of the circuit. Due to different delay times, those different usages of that input may be registered at different clock cycles, which might violate some assumptions.[1]

We cannot avoid metastability, but we can contain its effects. A classic solution is to use two flip-flops at the input. The assumption is that when the first flip-flop becomes metastable, it will resolve to a stable state within the clock period so that the setup and hold times of the second flip-flop will not be violated.

Figure 7.1 shows the border between the external world and the synchronous circuit. The input synchronizer consists of two flip-flops. The Chisel code for the input synchronizer is a one-liner that instantiates two registers.

[1] I experienced this issue once, and it took me quite some time to find the error.

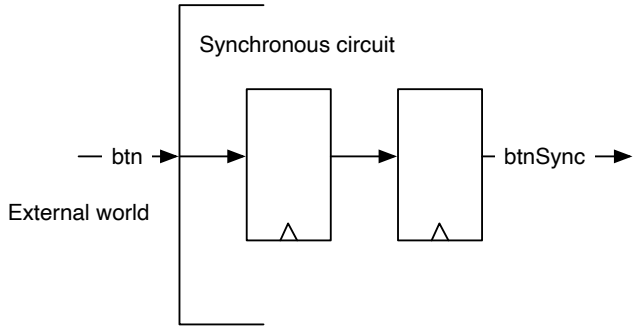


Figure 7.1: Input synchronizer.

```
val btnSync = RegNext(RegNext(btn))
```

All asynchronous external signals need an input synchronizer.[2] We also need to synchronize an external reset signal, also called the global reset. The reset signal shall pass through the two flip-flops before it is used as the reset signal for other flip-flops in the circuit. Concretely, the deassertion of the reset needs to be synchronous to the clock.

7.2 Debouncing

Switches and buttons may need some time to transition between on and off. During the transition, the switch may bounce between those two states. If we use such a signal without further processing, we might detect more transition events than we want to. One solution is to use time to filter out this bouncing. Assuming a maximum bouncing time of t_{bounce} we will sample the input signals with a period $T > t_{bounce}$. We will only use the sampled signal further downstream.

When sampling the input with this long period, we know that on a transition from 0 to 1, only one sample may fall into the bouncing region. The sample before will safely read a 0, and the sample after the bouncing region will safely read a 1. The sample in the bouncing region will either be 0 or 1. However, this does not matter

[2] The exception is when the input signal is dependent on a synchronous output signal, and we know the maximum propagation delay. A classic example is the interfacing of an asynchronous SRAM to a synchronous circuit, e.g., by a microprocessor.

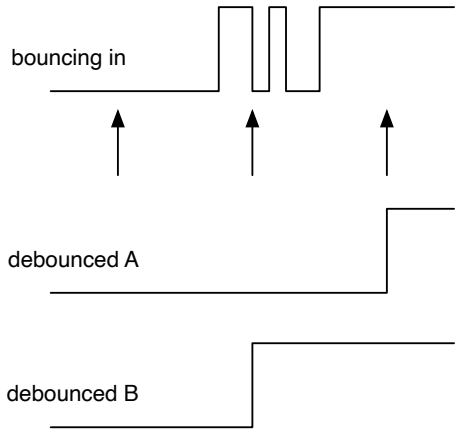


Figure 7.2: Debouncing an input signal.

as it then belongs either to the still 0 samples or to the already 1 sample. The critical point is that we have only one transition from 0 to 1.

Figure 7.2 shows the sampling for the debouncing in action. The top signal shows the bouncing input, and the arrows below show the sampling points. The distance between those sampling points must be longer than the maximum bouncing time. The first sample safely samples a 0, and the last sample in the figure samples a 1. The middle sample falls into the bouncing time. It may either be 0 or 1. The two possible outcomes are shown as `debounced A` and `debounced B`. Both have a single transition from 0 to 1. The only difference between these two outcomes is that the transition in version A is one sample period later. However, this is usually a non-issue.

The Chisel code for the debouncing is a little bit more evolved than the code for the synchronizer. We generate the sample timing with a counter that delivers a single cycle `tick` signal, as we have done in Section 6.2.2.

```
val fac = 100000000/100


val btnDebReg = RegInit(false.B)

val cntReg = RegInit(0.U(32.W))
val tick = cntReg === (fac-1).U
```

```
cntReg := cntReg + 1.U
when (tick) {
  cntReg := 0.U
  btnDebReg := btnSync
}
```

First, we need to decide on the sampling frequency. The above example assumes a 100 MHz clock and results in a sampling frequency of 100 Hz (assuming that the bouncing time is below 10 ms). The maximum counter value is `fac`, the division factor. We define a register `btnDebReg` for the debounced signal without a reset value. The register `cntReg` serves as a counter, and the `tick` signal is true when the counter has reached the maximum value. In that case, the `when` condition is `true`: (1) the counter is reset to 0 and (2) the debounce register stores the input sample. In our example, the input signal is named `btnSync` as the output from the input synchronizer shown in the previous section.

The debouncing circuit comes after the synchronizer circuit. First, we need to synchronize the asynchronous signal, and then we can further process it in the digital domain.

7.3 Filtering of the Input Signal

Sometimes our input signal may be noisy, maybe containing spikes that we might sample unintentionally with the input synchronizer and debouncing unit. One option to filter those input spikes is to use a majority voting circuit. In the simplest case, we take three samples and perform the majority vote. The majority function, which is related to the median function, results in the value of the majority. In our case, where we use sampling for the debouncing, we perform the majority voting on the sampled signal. Majority voting ensures that the signal is stable for longer than the sampling period.

Figure 7.3 shows the majority voting circuit. It consists of a 3-bit shift register enabled by the `tick` signal we used for the debouncing sampling. The output of the three registers is fed into the majority voting circuit. The majority voting function filters any signal change shorter than the sample period.

The following Chisel code shows the 3-bit shift register, enabled by the `tick` signal and the voting function, resulting in the signal `btnClean`.

Note that majority voting is very seldom needed.

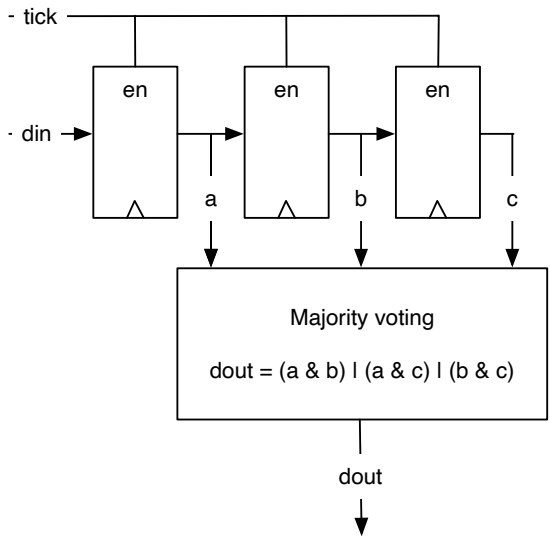


Figure 7.3: Majority voting on the sampled input signal.

```
val shiftReg = RegInit(0.U(3.W))
when (tick) {
  // shift left and input in LSB
  shiftReg := shiftReg(1, 0) ## btnDebReg
}
// Majority voting
val btnClean = (shiftReg(2) & shiftReg(1)) | (shiftReg(2)
    & shiftReg(0)) | (shiftReg(1) & shiftReg(0))
```

To use the output of our carefully processed input signal, we first detect the rising edge with a `RegNext` delay element and then compare this signal with the current value of `btnClean`. In this example, we use the single-cycle `risingEdge` signal to increment a counter.

```
val risingEdge = btnClean & !RegNext(btnClean)

// Use the rising edge of the debounced and
// filtered button to count up
val reg = RegInit(0.U(8.W))
when (risingEdge) {
  reg := reg + 1.U
}
```

7.4 Combining the Input Processing with Functions

To summarize the input processing, we show some more Chisel code. Because the presented circuits are tiny but reusable building blocks, we encapsulate them in functions. Section 10.2 shows how we can abstract small building blocks in lightweight Chisel functions instead of full modules. Those Chisel functions create hardware instances, e.g., the function `sync` creates two flip-flops connected to the input and to each other. The function returns the output of the second flip-flop. Listing 7.1 shows all input processing circuits as functions. If useful, those functions can be elevated to some utility class object.

```
def sync(v: Bool) = RegNext(RegNext(v))

def rising(v: Bool) = v & !RegNext(v)

def tickGen() = {
  val reg = RegInit(0.U(log2Up(fac).W))
  val tick = reg === (fac-1).U
  reg := Mux(tick, 0.U, reg + 1.U)
  tick
}

def filter(v: Bool, t: Bool) = {
  val reg = RegInit(0.U(3.W))
  when (t) {
    reg := reg(1, 0) ## v
  }
  (reg(2) & reg(1)) | (reg(2) & reg(0)) | (reg(1) & reg(0))
}

val btnSync = sync(io.btnU)

val tick = tickGen()
val btnDeb = RegInit(false.B)
when (tick) {
  btnDeb := btnSync
}

val btnClean = filter(btnDeb, tick)
val risingEdge = rising(btnClean)

// Use the rising edge of the debounced
// and filtered button for the counter
val reg = RegInit(0.U(8.W))
when (risingEdge) {
  reg := reg + 1.U
}
```

Listing 7.1: Summarizing input processing with functions.

7.5 Synchronizing Reset

Any digital circuit needs a reset signal to reset registers to a defined state. The reset state is set in Chisel with the `RegInit` constructor. A reset signal is usually an asynchronous input to the circuit. That means when directly connected to the reset of a flip-flop, it may violate timing constraints. In case of a synchronous reset it may violate setup and hold times of the flip-flop. Also when used as an asynchronous reset input, it still needs to be synchronized to the clock. Specifically, the *release* of the reset signal needs to be synchronized to the clock. Another failure with an asynchronous reset can be that different parts of the circuit may be reset in two different clock cycles and therefore be inconsistent.

The solution for this issue is to synchronize the reset signal in the very same way as any other asynchronous input with two flip-flops.

The reset and clock signals are usually hidden from the Chisel design. However, it is possible to access and set those signals. Each module has an implicit field `reset`. The solution is to have a top-level module that performs the synchronizing of the external reset signals and connects that synchronized signal to the reset input of the contained module.

```
class SyncReset extends Module {
  val io = IO(new Bundle() {
    val value = Output(UInt(1.W))
  })

  val syncReset = RegNext(RegNext(reset))
  val cnt = Module(new WhenCounter(5))
  cnt.reset := syncReset

  io.value := cnt.io.cnt
}
```

In the above example, `SyncReset` is the top-level module that contains a counter (`WhenCounter`). The reset signal of the top-level module is called `reset` and is connected to the input synchronizer (`RegNext(RegNext(reset))`). The output of that input synchronizer (`syncReset`) is connected to the reset *input* of the counter (`cnt.reset := syncReset`).

7.6 Exercise

Build a counter that is incremented by an input button. Display the counter value in binary with the LEDs on an FPGA board. First, observe if there are issues with a bouncing input button. Then resolve that issue by building the complete input processing chain with: (1) an input synchronizer, (2) a debouncing circuit, (3) a majority voting circuit to suppress noise, and (4) an edge detection circuit to trigger the increment of the counter.

As there is no guarantee that a modern button will always bounce, you can simulate the bouncing and the spikes by pressing the button manually in fast succession and using a low sample frequency. Select, e.g., one second as sample frequency, i.e., if the input clock runs at 100 MHz, divide it by 100,000,000. Simulate a bouncing button several times in fast succession before settling to a stable press. Test your circuit without and with the debouncing circuit sampling at 1 Hz. With the majority voting, you need to press between one and two seconds for a reliable counter increment. Also, the release of the button is majority voted. Therefore, the circuit only recognizes the release when it is longer than 1–2 seconds.

8 Finite-State Machines

A finite-state machine (FSM) is a basic building block in digital design. An FSM can be described as a set of *states* and conditional (guarded) *state transitions* between states. An FSM has an initial state, which is set on reset. FSMs are also called synchronous sequential circuits.

An implementation of an FSM consists of three parts: (1) a register that holds the current state, (2) combinational logic that computes the next state that depends on the current state and the input, and (3) combinational logic that computes the output of the FSM.

In principle, every digital circuit that contains a register or other memory elements to store state can be described as a single FSM. However, this might not be practical. For example, try to describe your laptop as a single FSM. In the next chapter, we describe how to build larger systems out of smaller FSMs by combining them into communicating FSMs.

8.1 Basic Finite-State Machine

Figure 8.1 shows the schematics of an FSM. The register contains the current `state`. The next state logic computes the next state value (`nextState`) from the current `state` and the input (`in`). On the next clock tick, `state` becomes `nextState`. The

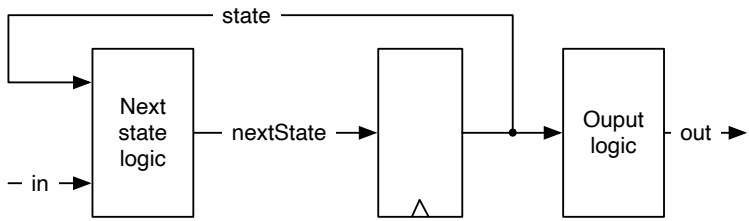


Figure 8.1: A finite-state machine (Moore type).

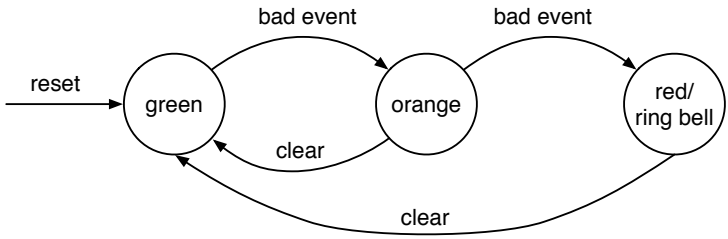


Figure 8.2: The state diagram of an alarm FSM.

output logic computes the output (`out`). As the output depends on the current state only, this state machine is called a Moore machine.

A state diagram describes the behavior of such an FSM visually. In a state diagram, individual states are depicted as circles labeled with the state names. State transitions are shown with arrows between states. The guard (or condition) when this transition is taken is drawn as a label for the arrow.

Figure 8.2 shows the state diagram of a simple example FSM. The FSM has three states: *green*, *orange*, and *red*, indicating a level of alarm. The FSM starts at the *green* level. When a *bad event* happens, the alarm level is switched to *orange*. On a second bad event, the alarm level is switched to *red*. In that case, we want to ring a bell; *ring bell* is the only output of this FSM. We add the output to the *red* state. The alarm can be reset with a *clear* signal.

Although a state diagram may be visually pleasing and the function of an FSM can be grasped quickly, a state table may be quicker to write down. Table 8.1 shows the state table for our alarm FSM. We list the current state, the input values, the resulting next state, and the output value for the current state. In principle, we need to specify all possible inputs for all possible states. This table would have $3 \times 4 = 12$ rows. We simplify the table by indicating that the *clear* input is a don't care when a *bad event* happens. That means *bad event* has priority over *clear*. The output column has some repetition. If we have a larger FSM and/or more outputs, we can split the table into two, one for the next state logic and one for the output logic.

Finally, after the design of our warning level FSM, we shall code it in Chisel. Listing 8.1 shows the Chisel code for the alarm FSM. Note that we use the Chisel type `Bool` for the inputs and the output of the FSM. To use the `switch` control instruction, we need to import `chisel3.util._`.

The complete Chisel code for this simple FSM fits into one page. Let us step

```
import chisel3._
import chisel3.util._

class SimpleFsm extends Module {
  val io = IO(new Bundle{
    val badEvent = Input(Bool())
    val clear = Input(Bool())
    val ringBell = Output(Bool())
  })

  // The three states
  object State extends ChiselEnum {
    val green, orange, red = Value
  }
  import State._
  // The state register
  val stateReg = RegInit(green)

  // Next state logic
  switch (stateReg) {
    is (green) {
      when(io.badEvent) {
        stateReg := orange
      }
    }
    is (orange) {
      when(io.badEvent) {
        stateReg := red
      } .elsewhen(io.clear) {
        stateReg := green
      }
    }
    is (red) {
      when (io.clear) {
        stateReg := green
      }
    }
  }
  // Output logic
  io.ringBell := stateReg === red
}
```

Listing 8.1: The Chisel code for the alarm FSM.

Table 8.1: State table for the alarm FSM.

	Input			
State	Bad event	Clear	Next state	Ring bell
green	0	0	green	0
green	1	-	orange	0
orange	0	0	orange	0
orange	1	-	red	0
orange	0	1	green	0
red	-	0	red	1
red	0	1	green	1

through the individual parts. The FSM has two input signals and a single output signal, captured in a Chisel `Bundle`:

```
val io = IO(new Bundle{
  val badEvent = Input(Bool())
  val clear = Input(Bool())
  val ringBell = Output(Bool())
})
```

At this place, we could spend some discussion on optimal state encoding. Two common options are binary or one-hot encoding. However, we leave those low-level optimizations to the synthesize tool and aim for readable code.[1] Therefore, we use the enumeration type `ChiselEnum` with symbolic names for the states:

```
object State extends ChiselEnum {
  val green, orange, red = Value
}
import State._
```

The individual state values are enumerated in a comma-separated list, followed by an assignment of `Value`. The register holding the state is defined with the *green* state as the reset value:

```
val stateReg = RegInit(green)
```

[1] In the current version of Chisel, the `ChiselEnum` type represents states in binary encoding. If we want a different encoding, e.g., one-hot encoding, we can define Chisel constants for the state names.

The meat of the FSM is in the next state logic. We use a Chisel switch on the state register to cover all states. Within each `is` branch we code the next state logic, which depends on the inputs, by assigning a new value to the state register:

```
switch (stateReg) {
  is (green) {
    when(io.badEvent) {
      stateReg := orange
    }
  }
  is (orange) {
    when(io.badEvent) {
      stateReg := red
    } .elsewhen(io.clear) {
      stateReg := green
    }
  }
  is (red) {
    when (io.clear) {
      stateReg := green
    }
  }
}
```

Last, but not least, we code our *ringing bell* output to be true when the state is *red*.

```
io.ringBell := stateReg === red
```

Note that we did *not* introduce a `nextState` signal for the register input, as it is common practice in Verilog or VHDL. Registers in Verilog and VHDL are described in a special syntax and cannot be assigned (and reassigned) within a combinational block. Therefore, the additional signal, computed in a combinational block, is introduced and connected to the register input. In Chisel a register is a base type and can be freely used and assigned within a combinational block.

8.2 Faster Output with a Mealy FSM

On a Moore FSM, the output depends only on the current state. That means that a change of an input can be seen as a change of the output no earlier than in the next clock cycle. If we want to observe an immediate change, we need a combinational

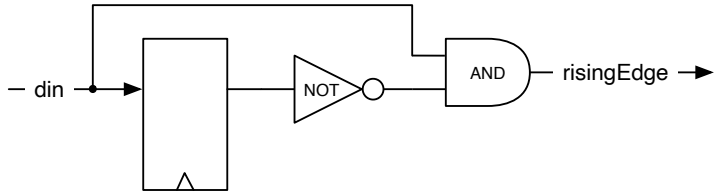


Figure 8.3: A rising edge detector (Mealy type FSM).

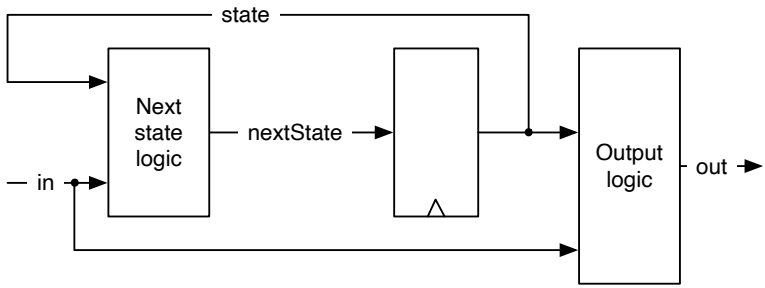


Figure 8.4: A Mealy type finite-state machine.

path from the input to the output. Let us consider a minimal example, an edge detection circuit. We have seen this Chisel one-liner before:

```
val risingEdge = din & !RegNext(din)
```

Figure 8.3 shows the schematic of the rising edge detector. The output becomes 1 for one clock cycle when the current input is 1 and the input in the last clock cycle was 0. The state register is just a single D flip-flop where the next state is just the input. We can also consider this as a delay element of one clock cycle. The output logic *compares* the current input with the current state.

When the output depends also on the input, i.e., there is a combinational path between the input of the FSM and the output, this is called a Mealy machine.

Figure 8.4 shows the schematic of a Mealy type FSM. Similar to the Moore FSM, the register contains the current `state`, and the next state logic computes the next state value (`nextState`) from the current `state` and the input (`in`). On the next clock tick, `state` becomes `nextState`. The output logic computes the output (`out`) from

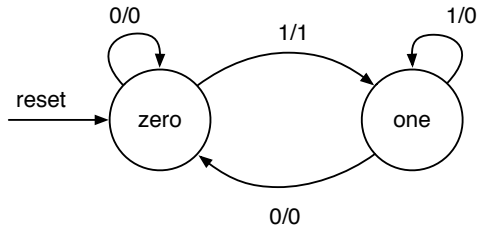


Figure 8.5: The state diagram of the rising edge detector as Mealy FSM.

the current state *and* the input to the FSM.

Figure 8.5 shows the state diagram of the Mealy FSM for the edge detector. As the state register consists just of a single D flip-flop, only two states are possible, which we name `zero` and `one` in this example. As the output of a Mealy FSM does not only depend on the state, but also on the input, we cannot describe the output as part of the state circle. Instead, the transitions between the states are labeled with the input value (condition) *and* the output (after the slash). Note also that we now need to draw self transitions, e.g., in state `zero` when the input is `0` the FSM stays in state `zero`, and the output is 0. The rising edge FSM generates the `1` output only on the transition from state `zero` to state `one`. In state `one`, which represents that the input is now `1`, the output is `0`. We only want a single (cycle) pulse for each rising edge of the input.

Listing 8.2 shows the Chisel code for the rising edge detection with a Mealy machine. As in the previous example, we use the Chisel type `Bool` for the single-bit input and output. The output logic is now part of the next state logic; on the transition from `zero` to `one`, the output is set to `true.B`. Otherwise, the default assignment to the output (`false.B`) counts.

One can ask if a full-blown FSM is the best solution for the edge detection circuit, especially, as we have seen a Chisel one-liner for the same functionality. The hardware consumptions are similar. Both solutions need a single D flip-flop for the state. The combinational logic for the FSM is probably a bit more complicated, as the state change depends on the current state and the input value. For this function, the one-liner is easier to write and easier to read, which is more important. Therefore, the one-liner is the preferred solution.

We have used this example to show one of the smallest possible Mealy FSMs. FSMs shall be used for more complex circuits with three or more states.

```
import chisel3._
import chisel3.util._

class RisingFsm extends Module {
  val io = IO(new Bundle{
    val din = Input(Bool())
    val risingEdge = Output(Bool())
  })

  // The two states
  object State extends ChiselEnum {
    val zero, one = Value
  }
  import State._

  // The state register
  val stateReg = RegInit(zero)

  // default value for output
  io.risingEdge := false.B

  // Next state and output logic
  switch (stateReg) {
    is(zero) {
      when(io.din) {
        stateReg := one
        io.risingEdge := true.B
      }
    }
    is(one) {
      when(!io.din) {
        stateReg := zero
      }
    }
  }
}
```

Listing 8.2: Rising edge detection with a Mealy FSM.

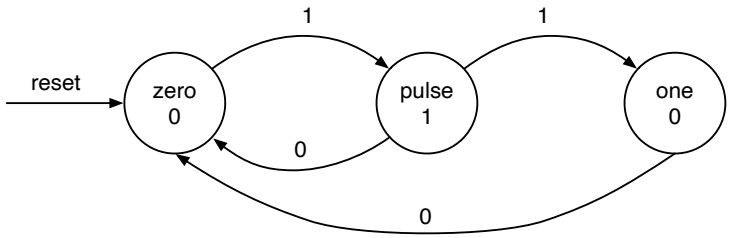


Figure 8.6: The state diagram of the rising edge detector as Moore FSM.

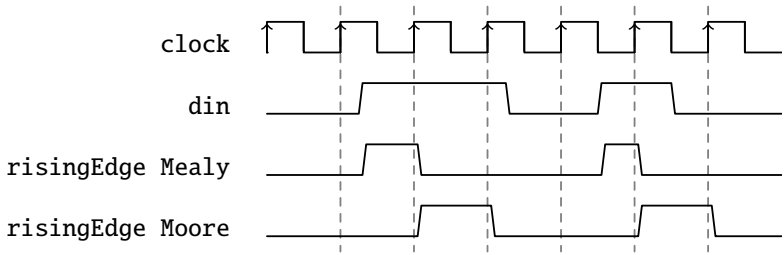


Figure 8.7: Mealy and a Moore FSM waveform for rising edge detection.

8.3 Moore versus Mealy

To show the difference between a Moore and Mealy FSM, we redo the edge detection with a Moore FSM.

Figure 8.6 shows the state diagram for the rising edge detection with a Moore FSM. The first thing to notice is that the Moore FSM needs three states, compared to the two states in the Mealy version. The state `pulse` is needed to produce the single-cycle pulse. The FSM stays in state `pulse` for just one clock cycle and then proceeds either back to the start state `zero` or to the `one` state, waiting for the input to become 0 again. We show the input condition on the state transition arrows and the FSM output within the state representing circles.

Listing 8.3 shows the Moore version of the rising edge detection circuit. It uses double the number of D flip-flops than the Mealy or directly coded version. The resulting next state logic is, therefore, also larger than the Mealy or directly coded version.

Figure 8.7 shows the waveform of a Mealy and a Moore version of the rising edge

```
import chisel3._
import chisel3.util._

class RisingMooreFsm extends Module {
  val io = IO(new Bundle{
    val din = Input(Bool())
    val risingEdge = Output(Bool())
  })

  // The three states
  object State extends ChiselEnum {
    val zero, puls, one = Value
  }
  import State._
  // The state register
  val stateReg = RegInit(zero)

  // Next state logic
  switch (stateReg) {
    is(zero) {
      when(io.din) {
        stateReg := puls
      }
    }
    is(puls) {
      when(io.din) {
        stateReg := one
      } .otherwise {
        stateReg := zero
      }
    }
    is(one) {
      when(!io.din) {
        stateReg := zero
      }
    }
  }

  // Output logic
  io.risingEdge := stateReg === puls
}
```

Listing 8.3: Rising edge detection with a Moore FSM.

detection FSM. We can see that the Mealy output closely follows the input rising edge, while the Moore output rises after the clock tick. We can also see that the Moore output is one clock cycle wide, whereas the Mealy output is usually less than a clock cycle.

From the above example, one is tempted to find Mealy FSMs the *better* FSMs as they need less state (and therefore logic) and react faster than a Moore FSM. However, the combinational path within a Mealy machine can cause troubles in larger designs. First, with a chain of communicating FSM (see next chapter), this combinational path can become lengthy. Second, if the communicating FSMs build a circle, the result is a combinational loop, which is an error in synchronous design. Due to a cut in the combinational path with the state register in a Moore FSM, all the above issues do not exist for communicating Moore FSMs.

In summary, Moore FSMs combine better for communicating state machines; they are *more robust* than Mealy FSMs. Use Mealy FSMs only when the reaction within the same clock cycle is of utmost importance. Small circuits such as the rising edge detection, which are practically Mealy machines, are fine as well.

8.4 Exercise

In this chapter, you have seen many examples of very small FSMs. Now it is time to write some *real* FSM code. Pick a little bit more complex example and implement the FSM and write a test bench for it.

A classic example of an FSM is a traffic light controller (see [6, Section 14.3]). A traffic light controller has to ensure that on a switch from red to green, there is a phase in between where both roads in the intersection have a no-go light (red and orange). Consider a priority road to make this example a bit more interesting. The minor road has two car detectors (on both entries into the intersection). Switch to green for the minor road only when a car is detected, and then switch back to green for the priority road.

TODO: Luca: Greatest common divisor with Euclide algorithm can be also a nice exercise. Martin: but this is shown at the Chisel homepage without an FSM.

TODO: Here a more interesting exercise. And not one from Dally.

9 Communicating State Machines

A problem is often too complex to describe it with a single FSM. In that case, the problem can be divided into two or more smaller and simpler FSMs. Those FSMs then communicate with signals. One FSM's output is another FSM's input; one FSM watches the output of the other FSM. When we split a large FSM into simpler ones, this is called factoring FSMs. Communicating FSMs are often directly designed from the specification of the design. A single FSM for the design would be too large in the first place.

9.1 A Light Flasher Example

To discuss communicating FSMs, we use an example from [6, Chapter 17], the light flasher. The light flasher has one input `start` and one output `light`. The specifications of the light flasher are as follows:

- when `start` is high for one clock cycle, the flashing sequence starts;
- the sequence is to flash three times;
- where the `light` goes *on* for six clock cycles, and the `light` goes *off* for four clock cycles between flashes;
- after the sequence, the FSM switches the `light` *off* and waits for the next start.

The FSM for a direct implementation[1] has 27 states: one initial state that is waiting for the input, 3×6 states for the three *on* states and 2×4 states for the *off* states. We do not show the code for this simple-minded implementation of the light flasher.

The problem can be solved more elegantly by factoring this large FSM into two smaller FSMs: the master FSM implements the flashing logic and the timer FSM implements the waiting. Figure 9.1 shows the composition of the two FSMs.

[1] The state diagram is shown in [6, p. 376].

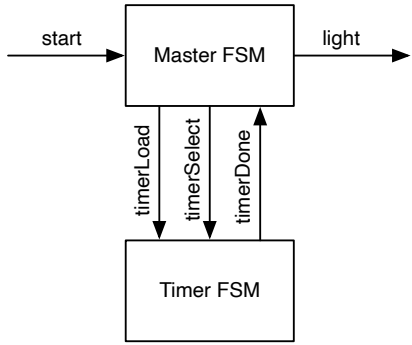


Figure 9.1: The light flasher split into a Master FSM and a Timer FSM.

The timer FSM counts down for 6 or 4 clock cycles to produce the desired timing. The timer specification is as follows:

- when `timerLoad` is asserted, the timer loads a value into the down counter, independent of the state;
- `timerSelect` selects between 5 and 3 for the load;
- `timerDone` is asserted when the counter completes the countdown and remains asserted;
- otherwise, the timer counts down.

The following code shows the timer FSM of the light flasher:

```
val timerReg = RegInit(0.U)
timerDone := timerReg === 0.U

// Timer FSM (down counter)
when(!timerDone) {
  timerReg := timerReg - 1.U
}
when (timerLoad) {
  when (timerSelect) {
    timerReg := 5.U
  } .otherwise {
```

```
      timerReg := 3.U
    }
  }
```

Listing 9.1 shows the master FSM. It has a starting state `off` and states for the complete blinking sequence. In each state, it waits for the time being done. The timer is loaded whenever it is done and in the initial `off` state. Signal `timerSelect` selects the value for the *next* state down counter.

```
object State extends ChiselEnum {
  val off, flash1, space1, flash2, space2, flash3 = Value
}
import State._

val stateReg = RegInit(off)

val light = WireDefault(false.B) // FSM output

// Timer connection
val timerLoad = WireDefault(false.B) // start timer
val timerSelect = WireDefault(true.B) // 6 or 4 cycles
val timerDone = Wire(Bool())

timerLoad := timerDone

// Master FSM
switch(stateReg) {
  is(off) {
    timerLoad := true.B
    timerSelect := true.B
    when (start) { stateReg := flash1 }
  }
  is (flash1) {
    timerSelect := false.B
    light := true.B
    when (timerDone) { stateReg := space1 }
  }
  is (space1) {
    when (timerDone) { stateReg := flash2 }
  }
  is (flash2) {
```

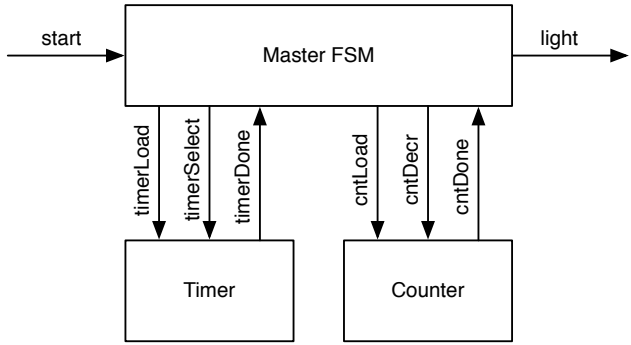


Figure 9.2: The light flasher split into a Master FSM, a Timer FSM, and a Counter FSM.

```
      timerSelect := false.B
      light := true.B
      when (timerDone) { stateReg := space2 }
    }
    is (space2) {
      when (timerDone) { stateReg := flash3 }
    }
    is (flash3) {
      timerSelect := false.B
      light := true.B
      when (timerDone) { stateReg := off }
    }
  }
```

Listing 9.1: Master FSM of the light flasher.

This solution with a master FSM and a timer has still redundancy in the code of the master FSM. States `flash1`, `flash2`, and `flash3` are performing the same function, states `space1` and `space2` as well. We can factor out the number of remaining flashes into a second counter. Then the master FSM is reduced to three states: `off`, `flash`, and `space`.

Figure 9.2 shows the design with a master FSM and two FSMs that count: one FSM to count clock cycles for the interval length of *on* and *off*; the second FSM to count the remaining flashes. Listing 9.2 code shows the down counter FSM:

```
val cntReg = RegInit(0.U)
cntDone := cntReg === 0.U

// Down counter FSM
when(cntLoad) { cntReg := 2.U }
when(cntDecr) { cntReg := cntReg - 1.U }
```

Listing 9.2: The down counter FSM.

Note that the counter is loaded with 2 for 3 flashes, as it counts the *remaining* flashes and is decremented in the state space when the timer is done. Listing 9.3 shows the master FSM for the double-refactored flasher.

```
object State extends ChiselEnum {
  val off, flash, space = Value
}
import State._

val stateReg = RegInit(off)

val light = WireDefault(false.B) // FSM output

// Timer connection
val timerLoad = WireDefault(false.B) // start timer with a
    load
val timerSelect = WireDefault(true.B) // select 6 or 4
    cycles
val timerDone = Wire(Bool())
// Counter connection
val cntLoad = WireDefault(false.B)
val cntDecr = WireDefault(false.B)
val cntDone = Wire(Bool())

timerLoad := timerDone

switch(stateReg) {
  is(off) {
    timerLoad := true.B
    timerSelect := true.B
    cntLoad := true.B
```

```
      when (start) { stateReg := flash }
    }
    is (flash) {
      timerSelect := false.B
      light := true.B
      when (timerDone & !cntDone) { stateReg := space }
      when (timerDone & cntDone) { stateReg := off }
    }
    is (space) {
      cntDecr := timerDone
      when (timerDone) { stateReg := flash }
    }
  }
```

Listing 9.3: The master FSM of the double refactored light flasher.

Besides having a master FSM that is reduced to just three states, our current solution is also better configurable. No FSM needs to be changed if we want to change the length of the *on* or *off* intervals or the number of flashes.

In this section, we have explored communicating circuits, especially FSMs, that only exchange control signals. To perform computation, we can combine a FSM with a datapath, as discussed in the next section.

9.2 State Machine with Datapath

One typical example of communicating state machines is a state machine combined with a datapath. This combination is often called a finite-state machine with a datapath (FSMD). The state machine controls the datapath, and the datapath performs the computation. The FSM inputs are the inputs from the environment and the outputs from the datapath. Some data from the environment is also fed into the datapath, and the data output comes from the datapath.

The FSMD shown in Figure 9.3 serves as an example that computes the popcount, also called the Hamming weight. The Hamming weight is the number of symbols different from the zero symbol. For a binary string, this is the number of '1's.

The popcount unit contains the data input `din` and the result output `popCount`, both connected to the datapath. For the input and the output, we use a ready/valid handshake. When data is available, valid is asserted. When a receiver can accept data it asserts ready. When both signals are asserted, the transfer takes place. The

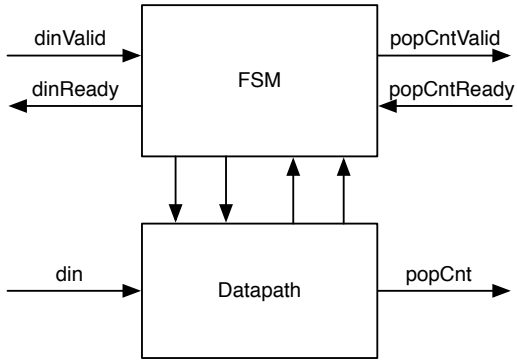


Figure 9.3: A state machine with a datapath.

handshake signals are connected to the FSM. The FSM is connected with the datapath with control signals towards the datapath and with status signals from the datapath.

We will co-design the FSM and the datapath. Figure 9.4 shows the state diagram of the FSM and Figure 9.5 shows the datapath for the popcount circuit. The FSM starts in state `Idle`, where the FSM waits for input. When data arrives, as signaled with an asserted `dinValid`, the FSM loads the shift register and advances to state `Count`. The data is loaded into the `shf` register. On the load also the `cnt` register is reset to 0.

In state `Count`, the number of '1's is counted sequentially. We use a shift register, an adder, an accumulator register, and a down counter (not shown in the datapath) to perform the computation. To count the number of '1's, the `shf` register is shifted right, and the least significant bit is added to `cnt` each clock cycle. A counter, not shown in the figure, counts down until all bits have been shifted through the least significant bit. When the counter reaches zero, the popcount has finished. The FSM switches to state `Done` and signals the result by asserting `popCntReady`. When the result is read, signaled by asserting `popCntValid`, the FSM switches back to `Idle`, ready to compute the next popcount.

The top-level component, shown in Listing 9.4, instantiates the FSM and the datapath components and connects them. Listing 9.5 shows the Chisel code for the datapath of the popcount circuit. On a `load` signal, the `regData` register is loaded with the input, the `regPopCount` register reset to 0, and the counter register `regCount` set to the number of shifts to be performed. Otherwise, the `regData` register is

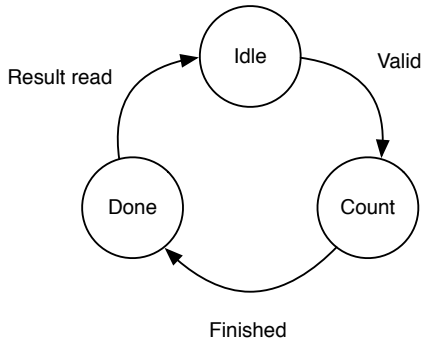


Figure 9.4: State diagram for the popcount FSM.

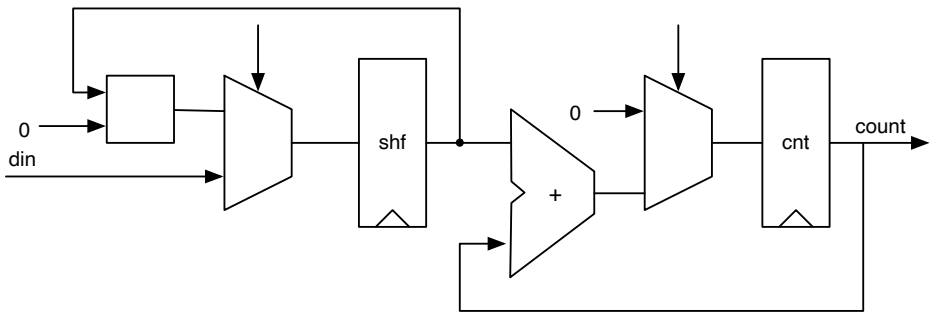


Figure 9.5: Datapath for the popcount circuit.

```
class PopulationCount extends Module {
  val io = IO(new Bundle {
    val dinValid = Input(Bool())
    val dinReady = Output(Bool())
    val din = Input(UInt(8.W))
    val popCntValid = Output(Bool())
    val popCntReady = Input(Bool())
    val popCnt = Output(UInt(4.W))
  })


  val fsm = Module(new PopCountFSM)
  val data = Module(new PopCountDataPath)

  fsm.io.dinValid := io.dinValid
  io.dinReady := fsm.io.dinReady
  io.popCntValid := fsm.io.popCntValid
  fsm.io.popCntReady := io.popCntReady

  data.io.din := io.din
  io.popCnt := data.io.popCnt
  data.io.load := fsm.io.load
  fsm.io.done := data.io.done
}
```

Listing 9.4: The top level of the popcount circuit.

shifted to the right, the least significant bit of the `regData` register added to the `regPopCount` register, and the counter decremented until it is 0. When the counter is 0, the output contains the popcount.

```
class PopCountDataPath extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(8.W))
    val load = Input(Bool())
    val popCnt = Output(UInt(4.W))
    val done = Output(Bool())
  })

  val dataReg = RegInit(0.U(8.W))
  val popCntReg = RegInit(0.U(8.W))
  val counterReg= RegInit(0.U(4.W))

  dataReg := 0.U ## dataReg(7, 1)
  popCntReg := popCntReg + dataReg(0)

  val done = counterReg === 0.U
  when (!done) {
    counterReg := counterReg - 1.U
  }

  when(io.load) {
    dataReg := io.din
    popCntReg := 0.U
    counterReg := 8.U
  }

  // debug output
  printf("%x %d\n", dataReg, popCntReg)

  io.popCnt := popCntReg
  io.done := done
}
```

Listing 9.5: Datapath of the popcount circuit.

```
class PopCountFSM extends Module {
  val io = IO(new Bundle {
    val dinValid = Input(Bool())
    val dinReady = Output(Bool())
    val popCntValid = Output(Bool())
    val popCntReady = Input(Bool())
    val load = Output(Bool())
    val done = Input(Bool())
  })

  object State extends ChiselEnum {
    val idle, count, done = Value
  }
  import State._
  val stateReg = RegInit(idle)
  io.load := false.B
  io.dinReady := false.B
  io.popCntValid := false.B

  switch(stateReg) {
    is(idle) {
      io.dinReady := true.B
      when(io.dinValid) {
        io.load := true.B
        stateReg := count
      }
    }
    is(count) {
      when(io.done) {
        stateReg := done
      }
    }
    is(done) {
      io.popCntValid := true.B
      when(io.popCntReady) {
        stateReg := idle
      }
    } } }
```

Listing 9.6: The FSM of the popcount circuit.

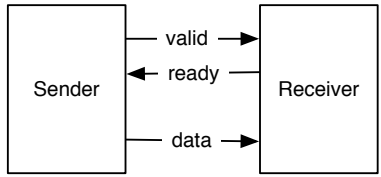


Figure 9.6: The ready/valid flow control.

Listing 9.6 shows the code of the FSM. The FSM starts in state `idle`. On a valid signal for the input data (`dinValid`) it switches to the `count` state and waits till the datapath has finished counting. When the popcount is `done`, the FSM switches to state `done` and waits till the popcount is read (signaled by `popCntReady`).

The popcount example consumed data (a word) and produced data (the popcount). For the coordinated exchange of data, we use handshake signals. The next section describes the ready/valid interface for flow control of unidirectional data exchange.

9.3 Ready/Valid Interface

Communication of subsystems can be generalized to the movement of data and handshaking for flow control. In the popcount example, we have seen a handshaking interface for the input and for the output data using valid and ready signals.

The ready/valid interface [6, p. 480] is a simple flow control interface consisting of `data` and a `valid` signal at the sender side (also called producer or source) and a `ready` signal at the receiver side (also called consumer or destination). Figure 9.6 shows the ready/valid connection. The sender asserts `valid` when `data` is available, and the receiver asserts `ready` when it is ready to receive one word of data. The transmission of the data happens when both signals, `valid` and `ready`, are asserted. If either of the two signals is not asserted, no transfer takes place.

Figure 9.7 shows a timing diagram of the ready/valid transaction where the receiver signals `ready` (from clock cycle 2 on) before the sender has data. The data transfer happens in clock cycle 4. From clock cycle 5 on neither the sender has data, nor the receiver is ready for the next transfer. When the receiver can receive data in every clock cycle, it is called an "always ready" interface and `ready` can be hardcoded to `true`.

Figure 9.8 shows a timing diagram of the ready/valid transaction where the sender

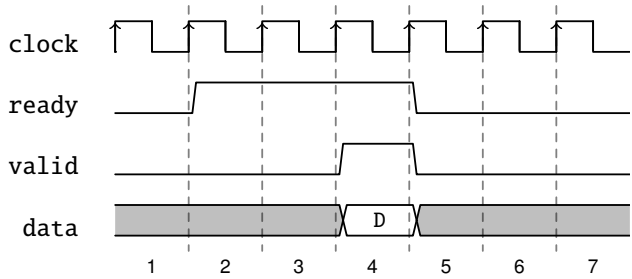


Figure 9.7: Data transfer with a ready/valid interface, early ready.

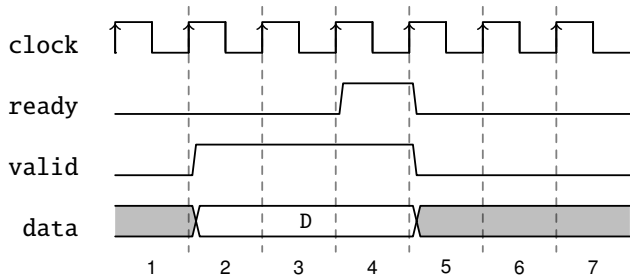


Figure 9.8: Data transfer with a ready/valid interface, late ready.

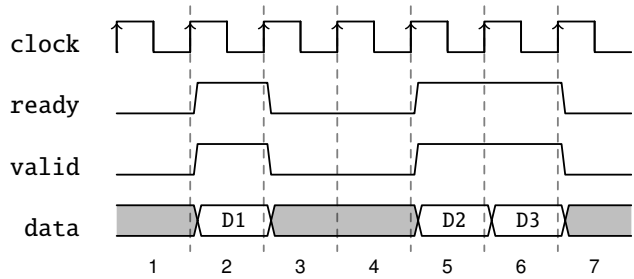


Figure 9.9: Single cycle ready/valid and back-to-back transfers.

signals `valid` (from clock cycle 2 on) before the receiver is ready. The data transfer happens in clock cycle 4. From clock cycle 5 on neither the sender has data, nor the receiver is ready for the next transfer. Similar to the "always ready" interface, we can envision an always valid interface. However, in that case, the data will probably not change on signaling `ready`, and we would simply drop the handshake signals.

Figure 9.9 shows further variations of using the ready/valid interface. In clock cycle 2 it happens that both signals (`ready` and `valid`) become asserted just for a single clock cycle, and the data transfer of `D1` happens. Data can be transferred back-to-back (in every clock cycle) as shown in clock cycles 5 and 6 with the transfer of `D2` and `D3`

To make this interface composable, neither `ready` nor `valid` is allowed to depend combinationally on the other signal. As this interface is so common, Chisel defines the `DecoupledIO` bundle, similar to the following:

```
class DecoupledIO[T <: Data](gen: T) extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen)
}
```

The `DecoupledIO` bundle is parameterized with the type for the `data`. The interface defined by Chisel uses the field `bits` for the data. `DecoupledIO` is part of the package `chisel3.util`.

One question remains if the `ready` or `valid` may be de-asserted after being asserted and *no* data transfer has happened. For example, a receiver might be ready for some time and not receive data, but due to some other events may become not

ready. The same can be envisioned with the sender, having data valid only some clock cycles and becoming non-valid without a data transfer happening. If this behavior is allowed or not is not part of the ready/valid interface, but needs to be defined by the concrete usage of the interface.

Chisel places no requirements on the signaling of `ready` and `valid` when using the class `DecoupledIO`. However, the class `IrrevocableIO` places following restrictions on the sender:

> A concrete subclass of `ReadyValidIO` that promises to not change the value of `bits` after a cycle where `valid` is high and `ready` is low. Additionally, once `valid` is raised it will never be lowered until after `ready` has also been raised.

Note that this is *just* a convention that cannot be enforced *just* by using the class `IrrevocableIO`.

The AXI bus [3] uses one ready/valid interface for each of the following parts of the bus: read address, read data, write address, and write data. AXI restricts the interface so that once the sender assets `valid`, it is not allowed to deassert it until the data transfer happens. This is the same restriction as just described in the comment of the `IrrevocableIO` interface. Furthermore, the sender is not allowed to wait for a receivers `ready` to assert `valid`. The receiver side is more relaxed. If `ready` is asserted, it is allowed to deassert it before `valid` is asserted. Furthermore, the receiver can wait for an asserted `valid` before asserting `ready`.

Listing 9.7 shows an example of using the ready/valid interface. The circuit represents a buffer built out of a register. The buffer has a ready/valid interface (`DecoupledIO`) at the input and one at the output. The `DecoupledIO` bundle is defined from the sender's viewpoint. Therefore, the input of the buffer (`in`) needs to change the direction with `Flipped`.

The module contains a register for the data (`dataReg`) and a single-bit register (`emptyReg`) signaling if the buffer is empty or full. This single bit represents a two-state Moore FSM with states empty and full. The input `ready` signal and the output `valid` signal depend only on the state of `emptyReg`. There is no combinational path between the buffer's input and output.

When the buffer is empty, and valid data is at the input, the data is registered and the state is changed to full. When the buffer is full, and the consumer side signals to be ready, the data is considered read, and the buffer is empty again.

```
class ReadyValidBuffer extends Module {
  val io = IO(new Bundle{
    val in = Flipped(new DecoupledIO(UInt(8.W)))
    val out = new DecoupledIO(UInt(8.W))
  })

  val dataReg = Reg(UInt(8.W))
  val emptyReg = RegInit(true.B)

  io.in.ready := emptyReg
  io.out.valid := !emptyReg
  io.out.bits := dataReg

  when (emptyReg & io.in.valid) {
    dataReg := io.in.bits
    emptyReg := false.B
  }

  when (!emptyReg & io.out.ready) {
    emptyReg := true.B
  }
}
```

Listing 9.7: A register as a buffer with a ready/valid interface

10 Hardware Generators

The strength of Chisel is that it allows us to write hardware generators. With older hardware description languages, such as VHDL and Verilog, we usually use another language, Java or Python, to generate hardware. Before using Chisel, I often wrote small Java programs to generate VHDL tables. In Chisel, the full power of Scala and Java, with its many open-source libraries, are available at hardware construction time. Therefore, we can write our hardware generators in the same language and execute them as part of the Chisel circuit generation. You can find further generator examples in [24].

10.1 A Little Bit of Scala

This subsection gives a very brief introduction to Scala. It should be enough to write simple hardware generators for Chisel. For an in-depth introduction to Scala, I recommend the textbook by Odersky et al. [29]. The Scala website also contains an online Scala book.[1]

Scala has two types of variables: `val`s and `var`s. A `val` gives an expression a name and cannot be reassigned to a value. The following snippet shows the definition of an integer value called `zero`. If we try reassigning a value to `zero`, we get a compile error.

```
// A value is a constant
val zero = 0
// No new assignment is possible
// The following will not compile
zero = 3
```

In Chisel, we use `val`s to name hardware components. Note that the `:=` operator is a Chisel operator and not a Scala operator.

Scala also provides the more classic version of a mutable variable as `var`. The following code defines an integer variable and reassigns it a new value:

[1] The link points to the Scala 2 version of the book, as Chisel is still based on Scala 2.

```
// We can change the value of a var variable
var x = 2
x = 3
```

We will need Scala vars to write hardware *generators*, but never need vars to name a hardware *component*.

You may have wondered what type those variables have. As we assigned an integer constant in the above example, the variable type is *inferred*; it is a Scala `Int` type. In most cases, the Scala compiler can infer the type. However, if we are in the mood of being more explicit, we can explicitly state the type as follows:

```
val number: Int = 42
```

Simple loops are written as follows:

```
// Loops from 0 to 9
// Automatically creates loop value i
for (i <- 0 until 10) {
  println(i)
}
```

We use a loop for circuit generators. The following loop connects individual bits of a shift register.

```
val regVec = Reg(Vec(8, UInt(1.W)))

regVec(0) := io.din
for (i <- 1 until 8) {
  regVec(i) := regVec(i-1)
}
```

Note that this is not the most concise expression of a shift register. It is better to use a plain `UInt` with the right size and assign the new value for the register with an expression using the `##` operator and proper indexing. This code snippet shows how a Scala `for` loop can be used for circuit generation.

Conditions are expressed with `if` and `else`. Note that this condition is evaluated at Scala runtime during circuit generation. This construct does *not* create a multiplexer, but allow writing *configurable* hardware generators.

```
for (i <- 0 until 10) {
  print(i)
```

```
    if (i%2 == 0) {
      println(" is even")
    } else {
      println(" is odd")
    }
  }
```

Scala has the notion of a tuple. A tuple can hold a sequence of different types. The tuple is built by placing the individual fields within parentheses. The fields are then accessed with `._n`, starting with 1 for the first field. The following code creates a tuple representing a city with the zip code and the name.

```
val city = (2000, "Frederiksberg")
val zipCode = city._1
val name = city._2
```

Tuples are useful when we want to return more than one value from a function. Tuples allow us to represent Chisel components with more than one output as a lightweight function instead of a full-blown module.

Scala has a powerful collection library. One of the simpler collection types is `Seq`, an ordered collection of elements (also called a sequence). The default implementation is immutable. We index into a `Seq` with `()`, with zero-based indexing. `Seq` is a base class with several different implementations. However, for most Chisel hardware generators, direct use of `Seq` is the preferred choice. The following code shows how to create a `Seq` that holds four Scala `Int` values. The second line accesses the second element, and `second` will be `15`.

```
val numbers = Seq(1, 15, -2, 0)
val second = numbers(1)
```

10.2 Lightweight Components with Functions

Modules are the standard way to structure your hardware description. However, there is some boilerplate code when declaring a module and when instantiating and connecting it. A lightweight way to structure your hardware is to use functions. Scala functions can take Chisel (and Scala) parameters and return generated hardware. As a simple example, we generate an adder:

```
def adder (x: UInt, y: UInt) = {
```

```
    x + y
  }
```

The return value of a function in Scala is the result of the last expression.[2] We can then create two adders by simply calling the function `adder`.

```
val x = adder(a, b)
// another adder
val y = adder(c, d)
```

Note that this is a *hardware generator*. That code does not execute any add operation during elaboration but creates two adders (hardware instances). In other words, it returns a wire to the output of the adder. We have written our first hardware generator!

The adder is an artificial example to keep it simple. Chisel has already an adder generator function, like `+(that: UInt)`.

As lightweight hardware generators, functions can also contain state (using a register). The following example returns a one-clock cycle delay element (a register). If a function has a single statement, we can write it in one line and omit the curly braces ({}).

```
def delay(x: UInt) = RegNext(x)
```

By calling the function with the function itself as parameter, this generated a two clock cycle delay.

```
val delOut = delay(delay(delIn))
```

Again, this is too short an example to be useful, as `RegNext()` is already that function that creates the register for the delay.

Functions return only one value. To provide more than one output, we can wrap several output wires into a Scala tuple. The following code generates hardware that compares two inputs and has two outputs.

```
def compare(a: UInt, b: UInt) = {
  val equ = a === b
  val gt = a > b
  (equ, gt)
}
```

[2] Scala also contains a `return` statement. The code could have been written a bit more verbose as `return x + y`.

With the parentheses, we wrap the two wires that are connected to the outputs of the comparator circuit into a Scala tuple.

Creating a comparator component with the `compare` function returns a tuple of two wires. We can access the two wires with the `._n` syntax.

```
val cmp = compare(inA, inB)
val equResult = cmp._1
val gtResult = cmp._2
```

However, we can directly decompose the tuple into two wires, in this case `equ` and `gt`, with following syntax.

```
val (equ, gt) = compare(inA, inB)
```

Functions can be declared as part of a `Module`. However, functions that will be used in different modules are better placed into a Scala object that collects utility functions.

10.3 Generate Combinational Logic

A logic table (truth table) is combinational logic. It is also called read-only memory (ROM), we can see, the input to the table is an address into such a ROM. We generate a logic table with `VecInit`. The following snippet of code creates a table to compute the square of a number `n`.

```
val squareROM = VecInit(0.U, 1.U, 4.U, 9.U, 16.U, 25.U)
val square = squareROM(n)
```

We can use the full power of Scala to generate our logic (tables). For example, we can generate a table of fixpoint constants to represent a trigonometric function, compute constants for digital filters, or write an assembler in Scala to generate code for a microprocessor written in Chisel. All those functions are in the same code base (same language) and can be executed during hardware generation.

A classic example of a table generation is the conversion of a binary number into a binary-coded decimal (BCD) representation. BCD represents a number in a decimal format using 4 bits for each decimal digit. For example, decimal `13` is in binary `1101` and BCD encoded as 1 and 3 in binary: `00010011`. BCD allows displaying numbers in decimal, a more user-friendly number representation than hexadecimal.

When using a classic hardware description language, such as Verilog or VHDL,

```
import chisel3._

class BcdTable extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val table = Wire(Vec(100, UInt(8.W)))

  // Convert binary to BCD
  for (i <- 0 until 100) {
    table(i) := (((i/10)<<4) + i%10).U
  }

  io.data := table(io.address)
}
```

Listing 10.1: Binary to binary-coded decimal conversion.

we would use another scripting or programming language to generate such a table. We can write a Java program that computes the table to convert binary to BCD. That Java program prints out VHDL code that can be included in a project. The Java program is about 100 lines of code; most of the code generating VHDL strings. However, the key part of the conversion is just two lines of code. With Chisel, we can compute this table directly as part of the hardware generation. Listing 10.1 shows the table generation for the binary to BCD conversion.

10.3.1 File Reading

We can also generate a logic table from a Scala `Array`. We may have data in a file that we want to read in during hardware generation time for the logic table. Listing 10.2 shows how to use the Scala `Source` class form the Scala standard library to read the file `data.txt`, which contains integer constants in a textual representation.[3]

[3] Scala is based on Java, and therefore, all Java file reading and writing libraries can be used, e.g., to read binary files.

```
import chisel3._
import scala.io.Source

class FileReader extends Module {
  val io = IO(new Bundle {
    val address = Input(UInt(8.W))
    val data = Output(UInt(8.W))
  })

  val array = new Array[Int](256)
  var idx = 0

  // read the data into a Scala array
  val source = Source.fromFile("data.txt")
  for (line <- source.getLines()) {
    array(idx) = line.toInt
    idx += 1
  }

  // convert the Scala integer array to a Seq
  // and then into a vector of Chisel UInt
  val table = VecInit(array.toIndexedSeq.map(_.U(8.W)))

  // use the table
  io.data := table(io.address)
}
```

Listing 10.2: Reading a text file to generate a logic table.

A few words on the maybe a bit intimidating expression:

```
val table = VecInit(array.toIndexedSeq.map(_.U(8.W)))
```

The method `toIndexedSeq` converts a Scala `Array` to a Scala sequence (`Seq`), which supports the mapping function `map`. `map` invokes a function on each element of the sequence and returns a sequence of the return value of the function. Our function `_.U(8.W)` represents each `Int` value from the Scala array as a `_` and performs the conversion from a Scala `Int` value to a Chisel `UInt` literal, with a size of 8 bits. The Chisel object `VecInit` creates a Chisel `Vec` from a sequence `Seq` of Chisel types.

We can use the initialization of a Chisel `Vec` from a Scala sequence to represent a message that we may send out to a serial port. A Scala/Java `String` is a Scala `Seq`. Therefore, the `map` method is available to map each Scala `Char` to a Chisel `UInt`. The following code converts the standard greeting from the `msg` string to a Chisel `Vec`:

```
val msg = "Hello World!"
val text = VecInit(msg.map(_.U))
val len = msg.length.U
```

This code is extracted from the serial port example, which is used later in this text to send a welcome message.

10.3.2 Type Conversion

Sometimes converting from one Chisel type to a different type is convenient. All types represent a collection of bits. Therefore, we can easily perform this mapping. As a first example, let us assume we receive bytes of data and would like to repackage 4 bytes into a 32-bit `UInt`. The following code maps the vector of bytes to an `UInt`. The bytes are mapped in the following order to the `UInt`: the first byte into the lower 8 bits, the next byte into the next, and so on.

```
val vec = Wire(Vec(4, UInt(8.W)))
val word = vec.asUInt
```

Given a `UInt` `word`, we can convert it back to a vector of four bytes.

```
val vec2 = word.asTypeOf(Vec(4, UInt(8.W)))
```

We can also convert a `Bundle` to a `UInt`. The bundle fields are ordered so that the last field (in this example, field `b`) is in the lower bits of the `UInt` (here 15 to 0),

followed by the second last, and so on. Note that this order is opposite to the order of mapping a `Vec`.

```
class MyBundle extends Bundle {
  val a = UInt(8.W)
  val b = UInt(16.W)
}

val bundle = Wire(new MyBundle)
val word2 = bundle.asUInt
```

A `UInt` can be converted (back) to a bundle as follows:

```
val bundle2 = word2.asTypeOf(new MyBundle)
```

That conversion can also be used to initialize all fields of a bundle to 0.

```
val bundle3 = 0.U.asTypeOf(new MyBundle)
```

10.4 Configuration with Parameters

Chisel components and functions can be configured with parameters. Parameters can be as simple as an integer constant, but can also be a Chisel hardware type.

10.4.1 Simple Parameters

The simplest way to parameterize a circuit is to define a bit width as a parameter. Parameters can be passed as arguments to the constructor of the Chisel module. The following is a toy example of a module that implements an adder with a configurable bit width. The bit width `n` is a parameter (of Scala type `Int`) of the component passed into the constructor that can be used in the IO bundle.

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val c = Output(UInt(n.W))
  })
```

```
  io.c := io.a + io.b
}
```

Parameterized versions of the adder can be created as follows:

```
  val add8 = Module(new ParamAdder(8))
  val add16 = Module(new ParamAdder(16))
```

10.4.2 Case Classes

If we need more parameters, we can add additional parameters to the constructor of the Chisel module. However, if we pass those parameters through several constructors, it might become tedious to use them. Furthermore, we need to edit several places when changing the number or type of parameters.

Scala has a very lightweight construct to package several fields into a class: a case classes. Case classes are like regular Scala classes, but with a very light-weight definition. The following code defines a case class to represent three parameters. It might be used for a device with a transmit (tx) buffer and a receive buffer(rx) of a certain width.

```
case class Config(txDepth: Int, rxDepth: Int, width: Int)
```

An object of that case class is created by simply calling the constructor. The fields are immutable and can be read by accessing them:

```
  val param = Config(4, 2, 16)

  println("The width is " + param.width)
```

We can also add code to the case class to check that the parameters are valid.

```
case class SaveConf(txDepth: Int, rxDepth: Int, width: Int) {

  assert(txDepth > 0 && rxDepth > 0 && width > 0,
      "parameters must be larger than 0")
}
```

10.4.3 Functions with Type Parameters

Having the bit width as a configuration parameter is just the starting point for hardware generators. Chisel type-parameters enable very flexible configurations. That feature allows Chisel to provide a multiplexer (`Mux`) that can accept any type of multiplexing. We build a multiplexer that accepts arbitrary types to show how to use Chisel types for the configuration. The following function defines the multiplexer:

```
def myMux[T <: Data](sel: Bool, tPath: T, fPath: T): T = {

  val ret = WireDefault(fPath)
  when (sel) {
    ret := tPath
  }
  ret
}
```

Chisel allows parameterizing functions with types, in our case with Chisel types. The expression in the square brackets `[T <: Data]` defines a type parameter `T` that is `Data` or a subclass of `Data`. `Data` is the root of the Chisel type system.

Our multiplexer function has three parameters: the boolean condition, one parameter for the true path, and one for the false path. Both path parameters are of type `T`, provided at the function call. The function itself is straightforward: we define a wire with the default value of `fPath` and change the value if the condition is true to the `tPath`. This condition is a classic multiplexer function. At the end of the function, we return the multiplexer hardware (the output). We can use our multiplexer function with simple types such as `UInt`:

```
val resA = myMux(selA, 5.U, 10.U)
```

The types of the two multiplexer paths need to be the same. The following wrong usage of the multiplexer results in a runtime error:

```
val resErr = myMux(selA, 5.U, 10.S)
```

To show a more complex multiplexer, we define a new type as a `Bundle` with two fields:

```
class ComplexIO extends Bundle {
  val d = UInt(10.W)
  val b = Bool()
```

```
}
```

We can define `Bundle` constants by first creating a `Wire` of the `Bundle` and then setting the subfields. Then we can use our parameterized multiplexer with this complex type.

```
val tVal = Wire(new ComplexIO)
tVal.b := true.B
tVal.d := 42.U
val fVal = Wire(new ComplexIO)
fVal.b := false.B
fVal.d := 13.U

// The multiplexer with a complex type
val resB = myMux(selB, tVal, fVal)
```

In our initial design of the function, we used `WireDefault` to create a wire with the type `T` with a default value. If we need to create a wire just of the Chisel type without using a default value, we can use `fPath.cloneType` to get the Chisel type. The following function shows the alternative way to code the multiplexer.

```
def myMuxAlt[T <: Data](sel: Bool, tPath: T, fPath: T): T
    = {

  val ret = Wire(fPath.cloneType)
  ret := fPath
  when (sel) {
    ret := tPath
  }
  ret
}
```

10.4.4 Modules with Type Parameters

We can also parameterize modules with Chisel types. We assume we want to design a network-on-chip to move data between different processing cores. However, we do not want to hardcode the data format in the router interface; we want to *parameterize* it. Similar to the type parameter for a function, we add a type parameter `T` to the Module constructor. Furthermore, we need to have one constructor parameter of that type. In this example, we also make the number of router ports configurable.

```
class NocRouter[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val address = Input(Vec(n, UInt(8.W)))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

To use our router, we first need to define the data type we want to route, e.g., as a Chisel `Bundle`:

```
class Payload extends Bundle {
  val data = UInt(16.W)
  val flag = Bool()
}
```

We create a router by passing an instance of the user-defined Bundle and the number of ports to the constructor of the router:

```
  val router = Module(new NocRouter(new Payload, 2))
```

10.4.5 Parameterized Bundles

In the router example, we used two different vectors of fields for the input of the router: one for the address and one for the data, which was parameterized. A more elegant solution would be to have a `Bundle` that itself is parametrized. Something like:

```
class Port[T <: Data](dt: T) extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

The `Bundle` has a parameter of type `T`, which is a subtype of Chisel's `Data` type. Within the bundle, we define a field `data` by invoking `cloneType` on the parameter. However, when we use a constructor parameter, this parameter becomes a public field of the class. When Chisel needs to clone the type of the `Bundle`, e.g., when used in a `Vec`, this public field is in the way. A solution (workaround) to this issue

is to make the parameter field private:

```
class Port[T <: Data](private val dt: T) extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

With that new `Bundle`, we can define our router ports

```
class NocRouter2[T <: Data](dt: T, n: Int) extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

and instantiate that router with a `Port` that takes a `Payload` as a parameter:

```
  val router = Module(new NocRouter2(new Port(new Payload),
      2))
```

10.4.6 Optional Ports

Some hardware generators might have IO ports that are dependent on a configuration. As an example, we implement a register file for a typical 32-bit RISC processor. For debugging, we want to be able to access all registers. Therefore, we want to have an additional port where we can read all registers in the tester. However, at Verilog generation, we do not want this expensive extra port.

Listing 10.3 shows such a register file. It has as parameter `debug` as a Scala `Boolean`. In the definition of the IO bundle, we use that parameter to either generate that port or not. In Scala, this is represented as an `Option`. An option can return a value wrapped in `Some` or represent the missing value as a `None`. In the main body of the module, we extract the value of the option with `get`.

Similar to the tester, we need to use the `get` method to get a reference of that IO port:

```
dut.io.dbgPort.get(4).expect(123.U)
```

```
class RegisterFile(debug: Boolean) extends Module {
  val io = IO(new Bundle {
    val rs1 = Input(UInt(5.W))
    val rs2 = Input(UInt(5.W))
    val rd = Input(UInt(5.W))
    val wrData = Input(UInt(32.W))
    val wrEna = Input(Bool())
    val rs1Val = Output(UInt(32.W))
    val rs2Val = Output(UInt(32.W))
    val dbgPort = if (debug)
      Some(Output(Vec(32, UInt(32.W)))) else None
  })
  val regfile = RegInit(VecInit(Seq.fill(32)(0.U(32.W))))
  io.rs1Val := regfile(io.rs1)
  io.rs2Val := regfile(io.rs2)
  when(io.wrEna) {
    regfile(io.rd) := io.wrData
  }
  if (debug) {
    io.dbgPort.get := regfile
  }
}
```

Listing 10.3: A register file with an optional debug port.

```
abstract class Ticker(n: Int) extends Module {
  val io = IO(new Bundle{
    val tick = Output(Bool())
  })
}
```

Listing 10.4: Base class for our ticker implementations.

```
class UpTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(0.U(8.W))

  cntReg := cntReg + 1.U
  val tick = cntReg === N
  when(tick) {
    cntReg := 0.U
  }

  io.tick := tick
}
```

Listing 10.5: Tick generation with a counter.

10.5 Use Inheritance

Chisel is an object-oriented language. A hardware component, the Chisel `Module`, is a Scala class. Therefore, we can use inheritance to factor a common behavior into a parent class. We explore how to use inheritance with an example.

In Section 6.2, we have explored different forms of counters which may be used for low-frequency tick generation. Let us assume we want to explore those different versions, for example, to compare their resource requirement. We start with an abstract class to define the ticking interface, shown in Listing 10.4.
Listing 10.5 shows a first implementation of that abstract class with a counter, counting up, for the tick generation.

We can test all different versions of our *ticker* logic with a single test bench. We

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec

trait TickerTestFunc {
  def testFn[T <: Ticker](dut: T, n: Int) = {
    // -1 means that no ticks have been seen yet
    var count = -1
    for (_ <- 0 to n * 3) {
      // Check for correct output
      if (count > 0)
        dut.io.tick.expect(false.B)
      else if (count == 0)
        dut.io.tick.expect(true.B)

      // Reset the counter on a tick
      if (dut.io.tick.peekBoolean())
        count = n-1
      else
        count -= 1
      dut.clock.step()
    }
  }
}
```

Listing 10.6: A tester for different versions of the ticker.

just need to define the test bench to accept subtypes of `Ticker`. Listing 10.6 shows the Chisel code for the tester. The `TickerTester` has several parameters: (1) the type parameter `[T <: Ticker]` to accept a `Ticker` or any class that inherits from `Ticker`, (2) the design under test, being of type `T` or a subtype thereof, and (3) the number of clock cycles we expect for each tick. The tester waits for the first occurrence of a tick (the start might be different for different implementations) and then checks that `tick` repeats every n clock cycles.

With a first, easy implementation of the ticker, we can test the tester itself, probably with some `println` debugging. When we are confident that the simple ticker and the tester are correct, we can proceed and explore two more versions of the ticker. Listing 10.7 shows the tick generation with a counter counting down to 0.

```
class DownTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

  val cntReg = RegInit(N)

  cntReg := cntReg - 1.U
  when(cntReg === 0.U) {
    cntReg := N
  }

  io.tick := cntReg === N
}
```

Listing 10.7: Tick generation with a down counter.

Listing 10.8 shows the nerd version of counting down to -1 to use less hardware by avoiding the comparator.

We can test all three versions of the ticker by using ScalaTest specifications, creating instances of the different versions of the ticker, and passing them to the generic test bench. Listing 10.9 shows the test specification. We run only the ticker tests with:

```
sbt "testOnly TickerTest"
```

10.6 Hardware Generation with Functional Programming

Scala supports functional programming, so Chisel does as well. We can use functions to represent hardware and combine those hardware components with functional programming by using a "higher-order function". Let us start with a simple example, the sum of a vector:

```
  def add(a: UInt, b:UInt) = a + b

  val sum = vec.reduce(add)
```

```
class NerdTicker(n: Int) extends Ticker(n) {

  val N = n

  val MAX = (N - 2).S(8.W)
  val cntReg = RegInit(MAX)
  io.tick := false.B

  cntReg := cntReg - 1.S
  when(cntReg(7)) {
    cntReg := MAX
    io.tick := true.B
  }
}
```

Listing 10.8: Tick generation by counting down to -1.

```
class TickerTest extends AnyFlatSpec with
    ChiselScalatestTester with TickerTestFunc {
  "UpTicker 5" should "pass" in {
    test(new UpTicker(5)) { dut => testFn(dut, 5) }
  }

  "DownTicker 7" should "pass" in {
    test(new DownTicker(7)) { dut => testFn(dut, 7) }
  }

  "NerdTicker 11" should "pass" in {
    test(new NerdTicker(11)) { dut => testFn(dut, 11) }
  }
}
```

Listing 10.9: ChiselTest for the ticker tests.

First, we define the hardware for the adder in function `add`. The vector (Chisel type `Vec`) is located in `vec`. The Scala method `reduce()` combines all collection elements with a binary operation, producing a single value. The `reduce()` method reduces the sequence starting from the first element. It takes the first two elements and operates. The result is combined with the next element until a single result is left.

The function to combine two elements is provided as a parameter to `reduce`, in our case `add`, which returns an adder. The resulting hardware is a chain of adders computing the sum of vector `vec` elements. Instead of defining the (simple) `add` function, we can provide the addition as anonymous function and use the Scala wildcard "_" to represent the two operands.

```
val sum = vec.reduce(_ + _)
```

With this one-liner, we have generated the chain of adders. For the sum function a chain is not ideal; a tree will have a shorter combinational delay. If we do not trust the synthesize tool to rearrange our adder chain, we can use Chisel's `reduceTree` method to generate a tree of adders:

```
val sum = vec.reduceTree(_ + _)
```

10.6.1 Minimum Search Example

As a more elaborate example, we will build a circuit to find the minimum value in a `Vec`. To express this circuit, we use an anonymous function, called *function literal* in Scala. The syntax for a function literal is parameters in parentheses, followed by a =>, followed by the function body:

```
(param) => function body
```

The function literal for the minimum function uses two parameters `x` and `y` and returns a multiplexer (`Mux`) that compares the two parameters and returns the smaller value.

```
val min = vec.reduceTree((x, y) => Mux(x < y, x, y))
```

Let us extend this circuit to return not only the minimal value from the `vec`, but also the position (index) of that minimal value in the `vec`. To return two values we define the Bundle `Two` to hold the value and the index. We declare the `vecTwo` `Vec`

```
class Two extends Bundle {
  val v = UInt(w.W)
  val idx = UInt(8.W)
}

val vecTwo = Wire(Vec(n, new Two()))
for (i <- 0 until n) {
  vecTwo(i).v := vec(i)
  vecTwo(i).idx := i.U
}

val res = vecTwo.reduceTree((x, y) => Mux(x.v < y.v, x, y))
```

Listing 10.10: Minimum search including the index.

that can hold these bundles and connect them in a loop to the original input and the index within the `Vec`, as shown in Listing 10.10.

As before, we use a function literal in the `reduceTree` method of the `vecTwo`, comparing the value field within the bundle and returning the multiplexer for the complete bundle. Value `res` points to the bundle containing the minimum value and the position.

As a more advanced variation of the minimum search circuit, we will use more Scala features to avoid creating the bundle to return the value and index. We will use a tuple to represent both values. The following code shows the application of a chain of functions to the original sequence. Chaining functions is a typical pattern in functional programming. This pattern can also be seen as a pipeline of operations.

TODO: This should be in a listing to have the right order between examples.

```
val resFun = vec.zipWithIndex
  .map((x) => (x._1, x._2.U))
  .reduce((x, y) => (Mux(x._1 < y._1, x._1, y._1),
      Mux(x._1 < y._1, x._2, y._2)))
```

The first function (`zipWithIndex`) transforms the original sequence of `UInts` to a sequence of tuples, where the first element is the unchanged `UInt` and the second element is the index value within the `vec` as Scala `Int`. In general, a `zip` function merges two sequences (zips them) into a single one containing the two elements as tuples.

The next function maps our tuple of a Chisel `UInt` and a Scala `Int` to two Chisel `UInts`. The `reduce` function provides the generation of the minimum finding. We compare the first element of the tuple in two multiplexers and return a tuple containing the minimum value and the position as Chisel `UInt` types.

Note that the whole functional expression uses a Scala `Vector` to hold intermediate results, but returns hardware (connected multiplexers) consisting of Chisel types only. As we use a Scala `Vector` here, we cannot use `reduceTree`, which is available on Chisel's `Vec` only.

To keep using `reduceTree` the following solution uses a Chisel `MixedVec` instead of a Scala tuple. A Chisel `MixedVec` is similar to a Scala tuple as it can have different types at different positions. Therefore, it cannot function as, for example, a multiplexer. However, we can use it as an indexable collection during hardware generation.

```
val scalaVector = vec.zipWithIndex
  .map((x) => MixedVecInit(x._1, x._2.U(8.W)))
val resFun2 = VecInit(scalaVector)
  .reduceTree((x, y) => Mux(x(0) < y(0), x, y))

val minVal = resFun2(0)
val minIdx = resFun2(1)
```

In the above example, we create a Scala `Vector` of the values with their index, but now using Chisel's "tuple". We then convert the Scala `Vector` into a Chisel `Vec`. Then we can again perform a tree-based reduction. Another benefit of this version is that we have only one multiplexer, which selects between two Chisel "tuples" that are actually `MixedVecs`. The result in `resFun2` is a `MixedVec` with two elements, accessed with an index, like a "normal" `Vec`.

10.6.2 An Arbitration Tree

With our tree reduction function, we can build an arbitration tree out of just 2:1 arbiters. We can generate the arbitration circuit as follows:

```
class Arbiter[T <: Data: Manifest](n: Int, private val gen:
    T) extends Module {
  val io = IO(new Bundle {
    val in = Flipped(Vec(n, new DecoupledIO(gen)))
    val out = new DecoupledIO(gen)
  })
```

```
  io.out <> io.in.reduceTree((a, b) => arbitrateSimp(a, b))
}
```

The input is a `Vec` of ready/valid interfaces and the output a single ready/valid interface. We *just* need a function that provides arbitration between two requests.

Simple Arbitration

As a first solution, we will build a priority-based arbitration, similar to the arbiter shown in Section 10.6.2. That arbiter in Section 10.6.2 was a purely combinational circuit. However, with the ready/valid interface, we are not allowed to have a combinational flow between a ready and a valid signal. Therefore, we need to introduce a state for those signals and register the incoming data.

Listing 10.11 shows the 2-to-1 arbitration function. This function assumes that a requester who has asserted `valid` will only deassert it when it is read by the receiver (signaled with a `ready`). Furthermore, we can set `ready` in the next clock cycle depending on `valid` in the current clock cycle. This is one specific interpretation of the ready/valid protocol, also used in AXI.

We need the following registers: `regData` to hold the data for the output, `regEmpty` as a flag to signal that the data register is empty, and two flags for the `ready` signals of the two inputs (`regReadyA` and `regReadyB`). The return value of the function (`out`) is a wire of type `DecoupledIO`.

When the data register is empty, and one of the two inputs signals a `valid` input, we signal a `ready` in the next clock cycle (via the `ready` register). Note that we can signal only one of the two inputs that the arbiter is `ready`, as we have only one data register. When we have a registered `ready`, we assume that the input is still `valid`, register the data, deassert `regEmpty`, and reset the `ready` flag.

The output is valid when the data register is not empty. When the receiver is `ready`, the data is transmitted, and the data register is empty again. As the last statement, the function returns `out`, the reference to the `DecoupledIO` wire.

Fair Arbitration

In Section 10.6.2, we presented a combinational version of an arbitration circuit. The combinational version is a priority arbiter, so one high-priority requester can dominate the arbitration. To avoid this domination, we must introduce some state to remember who won the arbitration last time. Our assumption is that if the 2:1 arbiter is fair, this results in a fair arbitration on a balanced arbitration tree.

```
  def arbitrateSimp(a: DecoupledIO[T], b: DecoupledIO[T]) = {

    val regData = Reg(gen)
    val regEmpty = RegInit(true.B)
    val regReadyA = RegInit(false.B)
    val regReadyB = RegInit(false.B)

    val out = Wire(new DecoupledIO(gen))

    when (a.valid & regEmpty & !regReadyB) {
      regReadyA := true.B
    } .elsewhen (b.valid & regEmpty & !regReadyA) {
      regReadyB := true.B
    }
    a.ready := regReadyA
    b.ready := regReadyB

    when (regReadyA) {
      regData := a.bits
      regEmpty := false.B
      regReadyA := false.B
    }
    when (regReadyB) {
      regData := b.bits
      regEmpty := false.B
      regReadyB := false.B
    }

    out.valid := !regEmpty
    when (out.ready) {
      regEmpty := true.B
    }

    out.bits := regData
    out
  }
```

Listing 10.11: A simple 2 to 1 arbiter.

```
def arbitrateFair(a: DecoupledIO[T], b: DecoupledIO[T]) = {
  object State extends ChiselEnum {
    val idleA, idleB, hasA, hasB = Value
  }
  import State._
  val regData = Reg(gen)
  val regState = RegInit(idleA)
  val out = Wire(new DecoupledIO(gen))
  a.ready := regState === idleA
  b.ready := regState === idleB
  out.valid := (regState === hasA || regState === hasB)
  switch(regState) {
    is (idleA) {
      when (a.valid) {
        regData := a.bits
        regState := hasA
      } otherwise {
        regState := idleB
      }
    }
    is (idleB) {
      when (b.valid) {
        regData := b.bits
        regState := hasB
      } otherwise {
        regState := idleA
      }
    }
    is (hasA) {
      when (out.ready) {
        regState := idleB
      }
    }
    is (hasB) {
      when (out.ready) {
        regState := idleA
      }
    }
  }
  out.bits := regData
  out
}
```

Listing 10.12: A fair 2-to-1 arbiter.

Listing 10.12 shows that fair 2:1 arbitration circuit. The arbiter contains one register for the data to store and one state register. To be fair, the arbiter switches between two idle states (`idleA` and `idleB`) when there is no request. Each of the two idle states accepts only one of the inputs. Note that with just a single register for storage, the arbiter can only be ready for one of the two inputs. To allow being ready for both inputs and switching priority, we would need a second data register to handle the case when both inputs are valid in the same clock cycle.

When a request is accepted, it stores the data and switches to one of the full states (`hasA` or `hasB`). When the consumer of the output accepts the data, the arbiter switches back to an idle state. It switches to the idle state that will accept a pending request from the other input in the next clock cycle.

11 Example Designs

In this section, we explore some small digital designs, such as a FIFO buffer, which is used as a building block for a larger design. As another example, we design a serial interface (also called UART), which may use the FIFO buffer. Furthermore, we will generalize the FIFO interface and show different possible implementations.

11.1 FIFO Buffer

We can decouple a writer (sender) and a reader (receiver) by a buffer between the writer and the reader. A common buffer is a first-in, first-out (FIFO) buffer. Figure 11.1 shows a writer, the FIFO, and a reader. The writer puts Data into the FIFO on `din` with an active `write` signal. Data is read from the FIFO by the reader on `dout` with an active `read` signal.

A FIFO is initially empty, singled by the `empty` signal. Reading from an empty FIFO is usually undefined. When data is written and never read, a FIFO will become `full`. Writing to a full FIFO is usually ignored and the data are lost. In other words, the signals `empty` and `full` serve as handshake signals

Several different implementations of a FIFO are possible: For example, using on-chip memory and read and write pointers or simply a chain of registers with a tiny state machine. For small buffers (up to tens of elements), a FIFO organized with individual registers connected into a chain of buffers is a simple implementation

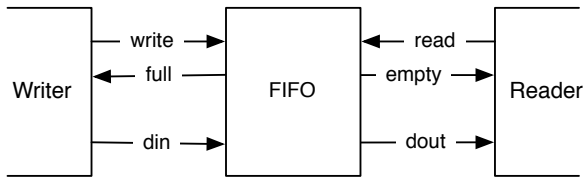


Figure 11.1: A writer, a FIFO buffer, and a reader.

with a low resource requirement. The code of the bubble FIFO is available in the chisel-examples repository.[1]

We start by defining the IO signals for the writer and the reader side. The size of the data is configurable with `size`. The write data are `din` and a write is signaled by `write`. The signal `full` performs the flow control at the writer's side.

```
class WriterIO(size: Int) extends Bundle {
  val write = Input(Bool())
  val full = Output(Bool())
  val din = Input(UInt(size.W))
}
```

The reader side provides data with `dout`, and the read is initiated with `read`. The `empty` signal is responsible for the flow control at the reader side.

```
class ReaderIO(size: Int) extends Bundle {
  val read = Input(Bool())
  val empty = Output(Bool())
  val dout = Output(UInt(size.W))
}
```

Listing 11.1 shows a single buffer. The buffer has an enqueueing port `enq` of type `WriterIO` and a dequeueing port `deq` of type `ReaderIO`. The state elements of the buffer are one register that holds the data (`dataReg`) and one state register for the simple FSM (`stateReg`). The FSM has only two states: either the buffer is `empty` or `full`. If the buffer is `empty`, a write will register the input data and change to the `full` state. If the buffer is `full`, a read will consume the data and change to the `empty` state. The IO ports `full` and `empty` represent the buffer state for the writer and the reader.

Listing 11.2 shows the complete FIFO. The complete FIFO has the same IO interface as the individual FIFO buffers. `BubbleFifo` has as parameters the `size` of the data word and `depth` for the number of buffer stages. We can build a `depth` stages bubble FIFO out of `depth` `FifoRegisters`. We create the stages by filling them into a Scala `Array`. The Scala array has no hardware meaning; it *just* serves as a container to have references to the created buffers. In a Scala `for` loop, we connect the individual buffers. The first buffer's enqueueing side is connected to the enqueueing IO of the complete FIFO, and the last buffer's dequeueing side to the dequeueing side of the complete FIFO.

[1] For completeness, the Chisel book repository also contains a copy of the FIFO code.

```
class FifoRegister(size: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  object State extends ChiselEnum {
    val empty, full = Value
  }
  import State._

  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(size.W))

  when(stateReg === empty) {
    when(io.enq.write) {
      stateReg := full
      dataReg := io.enq.din
    }
  }.elsewhen(stateReg === full) {
    when(io.deq.read) {
      stateReg := empty
      dataReg := 0.U // just to better see empty slots in
          the waveform
    }
  }.otherwise {
    // There should not be an otherwise state
  }

  io.enq.full := (stateReg === full)
  io.deq.empty := (stateReg === empty)
  io.deq.dout := dataReg
}
```

Listing 11.1: A single stage of the bubble FIFO.

```
class BubbleFifo(size: Int, depth: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  val buffers = Array.fill(depth) { Module(new
      FifoRegister(size)) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq.din := buffers(i).io.deq.dout
    buffers(i + 1).io.enq.write := ~buffers(i).io.deq.empty
    buffers(i).io.deq.read := ~buffers(i + 1).io.enq.full
  }
  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 11.2: A FIFO comprises an array of FIFO bubble stages.

The presented idea of connecting individual buffers to implement a FIFO queue is called a bubble FIFO, as the data bubbles through the queue. This is a simple, and good solution when the data rate is considerably slower then the clock rate, for example, as a decoupling buffer for a serial port, which is presented in the next section.

However, when the data rate approaches the clock frequency, the bubble FIFO has two limitations: (1) As each buffer's state has to toggle between *empty* and *full*, which means the maximum throughput of the FIFO is two clock cycles per word. (2) The data needs to bubble through the complete FIFO, therefore, the latency from the input to the output is at least the number of buffers. I will present other possible implementations of FIFOs in Section 11.3.

11.2 A Serial Port

A serial port (also called UART or RS-232) is one of the easiest options to communicate between your laptop and an FPGA board. As the name implies, data is transmitted serially. An 8-bit byte is transmitted as follows: one start bit (0), the 8-bit data, the least significant bit first, and then one or two stop bits (1). When

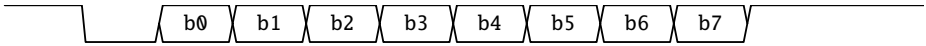


Figure 11.2: One byte transmitted by a UART.

no data is transmitted, the output is 1. Figure 11.2 shows the transmitted timing diagram of one byte.

We design our UART in a modular way with minimal functionality per module. We present a transmitter (TX), a receiver (RX), a buffer, and then usage of those base components.

First, we need an interface and a port definition. For the UART design, we use a ready/valid handshake interface (extending `DecoupledIO`), with a data size of 8 bits.

```
class UartIO extends DecoupledIO(UInt(8.W)) {
}
```

The convention of a ready/valid interface is that the data is transferred when both `ready` and `valid` are asserted.

Listing 11.3 shows a bare-bone serial transmitter (`Tx`). The IO ports are the `txd` port, where the serial data is sent, and a `channel`, where the transmitter can receive the characters to serialize and send. To generate the timing, we compute a constant for the time in clock cycles for one serial bit.

We use three registers: (1) register to shift the data (serialize them) (`shiftReg`), (2) a counter to generate the correct baud rate (`cntReg`), and (3) a counter for the number of bits that still need to be shifted out (`bitsReg`). No additional state register or FSM is needed; all state is encoded in those three registers.

Counter `cntReg` is continuously running (counting down to 0 and reloaded with the start value when 0). All action is only done when `cntReg` is 0. As we build a minimal transmitter, we have only the shift register to store the data. Therefore, the channel is only ready when `cntReg` is 0 and no bits are left to shift out. The IO port `txd` is directly connected to the least significant bit of the shift register.

When there are more bits to shift out (`bitsReg =/= 0.U`), we shift the bits to the right and fill with 1 (the idle level of a transmitter). If no more bits need to be shifted out, we check if the channel contains data (signaled with the `io.channel.valid` input). If so, the bit string to be shifted out is constructed with one start bit (0), the 8-bit data, and two stop bits (1). Therefore, the bit count is set to 11.

This very minimal transmitter has no additional buffer and can accept a new character only when the shift register is empty and at the clock cycle when `cntReg` is 0.

```
class Tx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
    val channel = Flipped(new UartIO())
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate -
      1).asUInt

  val shiftReg = RegInit(0x7ff.U)
  val cntReg = RegInit(0.U(20.W))
  val bitsReg = RegInit(0.U(4.W))

  io.channel.ready := (cntReg === 0.U) && (bitsReg === 0.U)
  io.txd := shiftReg(0)

  when(cntReg === 0.U) {

    cntReg := BIT_CNT
    when(bitsReg =/= 0.U) {
      val shift = shiftReg >> 1
      shiftReg := 1.U ## shift(9, 0)
      bitsReg := bitsReg - 1.U
    } .otherwise {
      when(io.channel.valid) {
        // two stop bits, data, one start bit
        shiftReg := 3.U ## io.channel.bits ## 0.U
        bitsReg := 11.U
      } .otherwise {
        shiftReg := 0x7ff.U
      }
    }

  } .otherwise {
    cntReg := cntReg - 1.U
  }
}
```

Listing 11.3: A transmitter for a serial port.

Accepting new data only when `cntReg` is zero means that the ready flag is also deasserted when there is space in the shift register. However, we do not want to add this "complexity" to the transmitter but delegate it to a buffer.

Listing 11.4 shows a single-byte buffer, similar to the FIFO register for the bubble FIFO. The input and the output are `UartIO`s. The buffer contains the minimal state machine to indicate `empty` or `full`. The buffer-driven handshake signals (`io.in.ready` and `io.out.valid`) depend on the state register.

When the state is `empty`, and data on the input is `valid`, we register the data and switch to state `full`. When the state is `full`, and the downstream receiver is `ready`, the downstream data transfer happens, and we switch back to state `empty`.

With that buffer, we can extend our bare-bone transmitter. Listing 11.5 shows the combination of the transmitter `Tx` with a single-buffer in front. This buffer now relaxes the issue that `Tx` was `ready` only for single clock cycles. We delegated the solution of this issue to the buffer module. An extension of the single-word buffer to a real FIFO can easily be done and needs no change in the transmitter or the single-byte buffer.

Listing 11.6 shows the code for the receiver (`Rx`). A receiver is a bit tricky, as it needs to reconstruct the timing of the serial data. The receiver waits for the falling edge of the start bit. From that event, the receiver waits 1.5-bit times to position itself into the middle of bit 0. Then, it samples and shifts in the bits every bit of time. You can observe these two waiting times as `BIT_CNT` and `START_CNT`. The same counter (`cntReg`) is used for both sample times. After 8 bits are shifted in, `validReg` signals an available byte.

Listing 11.7 shows the usage of the serial port transmitter by sending out a friendly message. We define the message as a Scala string (`msg`) and convert it to a Chisel `Vec` of `UInt`. A Scala string is a sequence that supports the `map` method. The `map` method takes as argument a function literal, applies this function to each element, and builds a sequence of the function's return values. If the function literal has only one argument, as in this case, the argument can be represented by `_`. Our function literal calls the Chisel method `.U` to convert the Scala `Char` to a Chisel `UInt`. The sequence is then passed to `VecInit` to construct a Chisel `Vec`. We index into the vector `text` with the counter `cntReg` to provide the individual characters to the buffered transmitter. With each `ready` signal, we increase the counter until the full string is sent out. The sender keeps `valid` asserted until the last character has been sent out.

Listing 11.8 shows the receiver and transmitter usage by connecting them. This connection generates an `Echo` circuit where each received character is sent back (echoed).

```
class Buffer extends Module {
  val io = IO(new Bundle {
    val in = Flipped(new UartIO())
    val out = new UartIO()
  })

  object State extends ChiselEnum {
    val empty, full = Value
  }
  import State._

  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(8.W))

  io.in.ready := stateReg === empty
  io.out.valid := stateReg === full

  when(stateReg === empty) {
    when(io.in.valid) {
      dataReg := io.in.bits
      stateReg := full
    }
  } .otherwise { // full
    when(io.out.ready) {
      stateReg := empty
    }
  }
  io.out.bits := dataReg
}
```

Listing 11.4: A single-byte buffer with a ready/valid interface.

```
class BufferedTx(frequency: Int, baudRate: Int) extends
    Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
    val channel = Flipped(new UartIO())
  })
  val tx = Module(new Tx(frequency, baudRate))
  val buf = Module(new Buffer())

  buf.io.in <> io.channel
  tx.io.channel <> buf.io.out
  io.txd <> tx.io.txd
}
```

Listing 11.5: A transmitter with an additional buffer.

11.3 FIFO Design Variations

At the beginning of this Chapter we introduced the design of a simple bubble FIFO. In this section we will generalize the FIFO and implement different variations of the queue. To make these implementations interchangeable, we will use inheritance, as introduced in Section 10.5.

11.3.1 Parameterizing FIFOs

We define an `abstract` FIFO class as a generic class with a Chisel type `T` as a parameter to be able to buffer any Chisel data type (see Listing 11.9). In the abstract class, we also test that the parameter `depth` has a useful value.

In Section 11.1 we defined our own types for the interface with common names for signals, such as `write`, `full`, `din`, `read`, `empty`, and `dout`. The input and the output of such a buffer consists of data and two signals for handshaking (for example, we `write` into the FIFO when it is not `full`).

Here we can generalize this handshaking to the ready/valid interface. We can enqueue an element (write into the FIFO) when the FIFO is `ready`. We signal this at the writer side with `valid`. As this ready/valid interface is so common, Chisel provides a definition of this interface in `DecoupledIO` as follows:[2]

[2] This is a simplification, as `DecoupledIO` actually extends an abstract class.

```
class Rx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val rxd = Input(UInt(1.W))
    val channel = new UartIO()
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate - 1)
  val START_CNT = ((3 * frequency / 2 + baudRate / 2) /
      baudRate - 2) // -2 for the falling delay

  // Sync in the asynchronous RX data
  val rxReg = RegNext(RegNext(io.rxd, 0.U), 0.U)
  val falling = !rxReg && (RegNext(rxReg) === 1.U)

  val shiftReg = RegInit(0.U(8.W))
  val cntReg = RegInit(BIT_CNT.U(20.W)) // have some idle
      time before listening
  val bitsReg = RegInit(0.U(4.W))
  val valReg = RegInit(false.B)

  when(cntReg =/= 0.U) {
    cntReg := cntReg - 1.U
  }.elsewhen(bitsReg =/= 0.U) {
    cntReg := BIT_CNT.U
    shiftReg := Cat(rxReg, shiftReg >> 1)
    bitsReg := bitsReg - 1.U
    // the last shifted in
    when(bitsReg === 1.U) {
      valReg := true.B
    }
  }.elsewhen(falling) { // wait 1.5 bits after falling edge
      of start
    cntReg := START_CNT.U
    bitsReg := 8.U
  }

  when(valReg && io.channel.ready) {
    valReg := false.B
  }

  io.channel.bits := shiftReg
  io.channel.valid := valReg
}
```

Listing 11.6: A receiver for a serial port.

```
class Sender(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
  })

  val tx = Module(new BufferedTx(frequency, baudRate))

  io.txd := tx.io.txd

  val msg = "Hello World!"
  val text = VecInit(msg.map(_.U))
  val len = msg.length.U

  val cntReg = RegInit(0.U(8.W))

  tx.io.channel.bits := text(cntReg)
  tx.io.channel.valid := cntReg =/= len

  when(tx.io.channel.ready && cntReg =/= len) {
    cntReg := cntReg + 1.U
  }
}
```

Listing 11.7: Sending “Hello World!” via the serial port.

```
class Echo(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(UInt(1.W))
    val rxd = Input(UInt(1.W))
  })
  val tx = Module(new BufferedTx(frequency, baudRate))
  val rx = Module(new Rx(frequency, baudRate))
  io.txd := tx.io.txd
  rx.io.rxd := io.rxd
  tx.io.channel <> rx.io.channel
}
```

Listing 11.8: Echoing data on the serial port.

```
abstract class Fifo[T <: Data](gen: T, val depth: Int)
    extends Module {
  val io = IO(new FifoIO(gen))

  require(depth > 0, "Number of buffer elements needs to be
      larger than 0")
}
```

Listing 11.9: Abstract class for FIFO veriations.

```
class DecoupledIO[T <: Data](gen: T) extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen)
}
```

With the `DecoupledIO` interface we define the interface for our FIFOs: a `FifoIO` with an enqueue (`enq`) and a dequeue (`deq`) port consisting of read/valid interfaces. The `DecoupledIO` interface is defined from the writer's (producer's) viewpoint. Therefore, the enqueue port of the FIFO needs to flip the signal directions.

```
class FifoIO[T <: Data](private val gen: T) extends Bundle {
  val enq = Flipped(new DecoupledIO(gen))
  val deq = new DecoupledIO(gen)
}
```

With the abstract base class and an interface, we can specialize in different FIFO implementations optimized for different parameters (speed, area, power, or just simplicity).

11.3.2 Redesigning the Bubble FIFO

We can redefine our bubble FIFO from Section 11.1 using standard ready/valid interfaces and being parametrizable with a Chisel data type.

```
class BubbleFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  private class Buffer() extends Module {
```

```
    val io = IO(new FifoIO(gen))

    val fullReg = RegInit(false.B)
    val dataReg = Reg(gen)

    when(fullReg) {
      when(io.deq.ready) {
        fullReg := false.B
      }
    }.otherwise {
      when(io.enq.valid) {
        fullReg := true.B
        dataReg := io.enq.bits
      }
    }

    io.enq.ready := !fullReg
    io.deq.valid := fullReg
    io.deq.bits := dataReg
  }

  private val buffers = Array.fill(depth) { Module(new
      Buffer()) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq <> buffers(i).io.deq
  }

  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

Listing 11.10: A bubble FIFO with a ready/valid interface.

Listing 11.10 shows the refactored bubble FIFO with a ready/valid interface. Note what we put the `Buffer` component inside `BubbleFifo` as a private class. This helper class is only needed for this component, and therefore, we hide it and avoid polluting the namespace. The buffer class has also been simplified. Instead of an FSM, we use only a single bit (`fullReg`) for the state of the buffer: full or empty.

The bubble FIFO is simple, easy to understand, and uses minimal resources. However, as each buffer stage has to toggle between empty and full, the maximum bandwidth of this FIFO is one word every two clock cycles.

One could consider looking at both interface sides in the buffer to be able to accept a new word when the producer `valid` and the consumer is `ready`. However, this introduces a combinational path from the consumer handshake to the producer handshake, which violates the semantics of the ready/valid protocol.

11.3.3 Double Buffer FIFO

One solution is to stay `ready` even when the buffer register is full. To accept a data word from the producer when the consumer is not `ready`, we need a second buffer, we call it the shadow register. When the buffer is full, new data is stored in the shadow register and `ready` is deasserted. When the consumer becomes `ready` again, data is transferred from the data register to the consumer and from the shadow register into the data register.

```
class DoubleBufferFifo[T <: Data](gen: T, depth: Int)
    extends Fifo(gen: T, depth: Int) {

  private class DoubleBuffer[T <: Data](gen: T) extends
      Module {
    val io = IO(new FifoIO(gen))

    object State extends ChiselEnum {
      val empty, one, two = Value
    }
    import State._

    val stateReg = RegInit(empty)
    val dataReg = Reg(gen)
    val shadowReg = Reg(gen)

    switch(stateReg) {
      is(empty) {
        when(io.enq.valid) {
          stateReg := one
          dataReg := io.enq.bits
        }
      }
      is(one) {
        when(io.deq.ready && !io.enq.valid) {
          stateReg := empty
        }
        when(io.deq.ready && io.enq.valid) {
          stateReg := one
          dataReg := io.enq.bits
        }
        when(!io.deq.ready && io.enq.valid) {
          stateReg := two
          shadowReg := io.enq.bits
        }
      }
      is(two) {
        when(io.deq.ready) {
          dataReg := shadowReg
          stateReg := one
```

```
        }
      }
    }

    io.enq.ready := (stateReg === empty || stateReg === one)
    io.deq.valid := (stateReg === one || stateReg === two)
    io.deq.bits := dataReg
  }

  private val buffers = Array.fill((depth + 1) / 2) {
      Module(new DoubleBuffer(gen)) }

  for (i <- 0 until (depth + 1) / 2 - 1) {
    buffers(i + 1).io.enq <> buffers(i).io.deq
  }
  io.enq <> buffers(0).io.enq
  io.deq <> buffers((depth + 1) / 2 - 1).io.deq
}
```

Listing 11.11: A FIFO with double buffer elements.

Listing 11.11 shows the double buffer FIFO. As each buffer element can store two entries, we need only half of the buffer elements (`depth/2`). The `DoubleBuffer` contains two registers, `dataReg` and `shadowReg`. The consumer is served always from `dataReg`. The double buffer has three states: `empty`, `one`, and `two`, which signal the fill level of the double buffer. The buffer is `ready` to accept new data when is it in state `empty` or `one`. The buffer has valid data when it is in state `one` or `two`.

If we run the FIFO at full speed, and the consumer is always `ready`, the steady state of the double buffers are `one`. Only when the consumer deasserts `ready`, the queue fills up, and the buffers enter state `two`. However, compared to a single bubble FIFO, a restart of the queue takes only half the number of clock cycles for the same buffer capacity. Similar the fall through latency is half of the bubble FIFO.

11.3.4 FIFO with Register Memory

When you come with a software engineering background,, you may have been wondering that we built hardware queues out of many small buffer elements, executing in parallel and handshaking with upstream and downstream elements. For small buffers, this is probably the most efficient implementation.

A queue in software is usually used by sequential code in two threads. We use a queue to decouple a producer and consumer thread. In this setting, a fixed size FIFO queue is usually implemented as a circular buffer. Two pointers point into read and write positions in a memory set aside for the queue. When the pointers reach the end of the memory, they are set back to the beginning of that memory. The difference between the two pointers is the number of elements in the queue. When the two pointers point to the same address, the queue is either empty or full. To distinguish between empty and full, we need another flag.

We can implement such a memory-based FIFO queue in hardware as well. For small queues, we can use a register file (i.e., a `Reg(Vec())`). Listing 11.12 shows a FIFO queue implemented with memory and read and write pointers.

```
class RegFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth - 1).U, 0.U, cntReg
        + 1.U)
    when(incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  // the register based memory
  val memReg = Reg(Vec(depth, gen))

  val incrRead = WireDefault(false.B)
  val incrWrite = WireDefault(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)

  val emptyReg = RegInit(true.B)
  val fullReg = RegInit(false.B)

  val op = io.enq.valid ## io.deq.ready
  val doWrite = WireDefault(false.B)

  switch(op) {
    is("b00".U) {}
    is("b01".U) { // read
      when(!emptyReg) {
        fullReg := false.B
        emptyReg := nextRead === writePtr
        incrRead := true.B
      }
    }
    is("b10".U) { // write
      when(!fullReg) {
        doWrite := true.B
        emptyReg := false.B
```

```
        fullReg := nextWrite === readPtr
        incrWrite := true.B
      }
    }
    is("b11".U) { // write and read
      when(!fullReg) {
        doWrite := true.B
        emptyReg := false.B
        when(emptyReg) {
          fullReg := false.B
        }.otherwise {
          fullReg := nextWrite === nextRead
        }
        incrWrite := true.B
      }
      when(!emptyReg) {
        fullReg := false.B
        when(fullReg) {
          emptyReg := false.B
        }.otherwise {
          emptyReg := nextRead === nextWrite
        }
        incrRead := true.B
      }
    }
  }

  when(doWrite) {
    memReg(writePtr) := io.enq.bits
  }

  io.deq.bits := memReg(readPtr)
  io.enq.ready := !fullReg
  io.deq.valid := !emptyReg
}
```

Listing 11.12: A FIFO with a register based memory.

As there are two pointers that are incremented on an action and wrap around at the end of the buffer, we define a function `counter()` that implements those wrapping counters. With `log2Ceil(depth).W`, we compute the bit length for the counter.

The next value is either an increment by 1 or a wrap-around to 0. The counter is incremented only when the input `incr` is `true.B`.

Furthermore, as we need also the possible next value (increment or 0 on wrap-around), we return this value from the `counter` function as well. In Scala we can return a *tuple*, which is simply a container to hold more than one value. The syntax to create a tuple is simply wrapping the comma-separated values in parentheses:

```
val t = (v1, v2)
```

We can deconstruct such a tuple by using the parenthesis notation on the left-hand side of the assignment:

```
val (x1, x2) = t
```

For the memory we us a register of a vector (`Reg(Vec(depth, gen))`) of Chisel data type `gen`. We define two signals to increment the read and write pointer and create the read and write pointers with the function `counter`. When both pointers are equal, the buffer is either empty or full. We define two flags to for the notion of empty and full.

When the producer asserts `valid` and the FIFO is not full we: (1) write into the buffer, (2) ensure `emptyReg` is deasserted, (3) mark the buffer full if the write pointer will catch up with the read pointer in the next clock cycle (compare the current read pointer with the next write pointer), and (4) signal the write counter to increment.

When the consumer is `ready` and the FIFO is not empty we: (1) ensure that the `fullReg` is deasserted, (2) mark the buffer empty if the read pointer will catch up with the write pointer in the next clock cycle, and (3) signal the read counter to increment.

A concurrent read and write is also possible. This case summarizes the two former cases. *TODO: I am not sure if the edge cases are correct. A read and write to on a full buffer should work, shouldn't it?*

The output of the FIFO is the memory element at the read pointer address. The ready and valid flags are simply derived from the full and empty flags.

11.3.5 FIFO with On-Chip Memory

The last version of the FIFO used a register file to represent the memory, which is a good solution for a small FIFO. For larger FIFOs, it is better to use on-chip memory. Listing 11.13 shows a FIFO using a synchronous memory for storage.

```
class MemFifo[T <: Data](gen: T, depth: Int) extends
    Fifo(gen: T, depth: Int) {

  def counter(depth: Int, incr: Bool): (UInt, UInt) = {
    val cntReg = RegInit(0.U(log2Ceil(depth).W))
    val nextVal = Mux(cntReg === (depth - 1).U, 0.U, cntReg
        + 1.U)
    when(incr) {
      cntReg := nextVal
    }
    (cntReg, nextVal)
  }

  val mem = SyncReadMem(depth, gen, SyncReadMem.WriteFirst)

  val incrRead = WireDefault(false.B)
  val incrWrite = WireDefault(false.B)
  val (readPtr, nextRead) = counter(depth, incrRead)
  val (writePtr, nextWrite) = counter(depth, incrWrite)

  val emptyReg = RegInit(true.B)
  val fullReg = RegInit(false.B)

  val outputReg = Reg(gen)
  val outputValidReg = RegInit(false.B)
  val read = WireDefault(false.B)

  io.deq.valid := outputValidReg
  io.enq.ready := !fullReg

  val doWrite = WireDefault(false.B)
  val data = Wire(gen)
  data := mem.read(readPtr)
  io.deq.bits := data
  when(doWrite) {
    mem.write(writePtr, io.enq.bits)
  }

  val readCond =
    !outputValidReg && ((readPtr =/= writePtr) || fullReg)
```

```
        // should add optimization when downstream is ready
        for pipielining
  when(readCond) {
    read := true.B
    incrRead := true.B
    outputReg := data
    outputValidReg := true.B
    emptyReg := nextRead === writePtr
    fullReg := false.B // no concurrent read when full (at
        the moment)
  }
  when(io.deq.fire) {
    outputValidReg := false.B
  }
  io.deq.bits := outputReg

  when(io.enq.fire) {
    emptyReg := false.B
    fullReg := (nextWrite === readPtr) & !read
    incrWrite := true.B
    doWrite := true.B
  }
}
```

Listing 11.13: A FIFO with a on-chip memory.

The handling of read and write pointer is identical to the register memory FIFO. However, a synchronous on-chip memory delivers the result of a read in the next clock cycle, where the read of the register file was available in the same clock cycle. Therefore, we need an additional register to handle this latency.

TODO: Maybe redo the FIFO again as described below (in the comment).

TODO: The memory based FIFO can efficiently hold larger amounts of data in the queue and has a short fall through latency. In the last design, the output of the FIFO may come directly from the memory read. If this data path is in the critical path of the design, we can easily pipeline our design by combining two FIFOs. Listing 11.14 shows such a combination. On the output of the memory based FIFO we add a single stage double buffer FIFO to decouple the memory read path from the output.

```
class CombFifo[T <: Data](gen: T, depth: Int) extends
```

```
    Fifo(gen: T, depth: Int) {

  val memFifo = Module(new MemFifo(gen, depth))
  val bufferFIFO = Module(new DoubleBufferFifo(gen, 2))
  io.enq <> memFifo.io.enq
  memFifo.io.deq <> bufferFIFO.io.enq
  bufferFIFO.io.deq <> io.deq
}
```

Listing 11.14: Combining a memory based FIFO with double-buffer stage.

11.4 A Multi-clock Memory

In large designs with multiple clock domains, you may need a way to safely pass data from one domain to another. We have previously seen synchronization as one solution to this issue. An alternative is to use a multi-clock memory as a buffer between the two (or more) domains.

Chisel supports multi-clock designs with the `withClock` and `withClockAndReset` constructs. All storage elements defined within a `withClock(clk)` block are clocked by `clk`. For multi-clock memories, the memory module should be defined outside all `withClock` blocks, while each port should have its own `withClock` block. A parameterized multi-clock memory is shown in Listing 11.15.

```
class MemoryIO(val n: Int, val w: Int) extends Bundle {
  val clk = Input(Bool())
  val addr  = Input(UInt(log2Up(n).W))
  val datai = Input(UInt(w.W))
  val datao = Output(UInt(w.W))
  val en = Input(Bool())
  val we = Input(Bool())
}

class MultiClockMemory(ports: Int, n: Int = 1024, w: Int =
    32) extends Module {
  val io = IO(new Bundle {
    val ps = Vec(ports, new MemoryIO(n, w))
  })

  val ram = SyncReadMem(n, UInt(w.W))
```

```
  /* does not work in Chisel 3.5.6
  for (i <- 0 until ports) {
    val p = io.ps(i)
    p.datao := ram.readWrite(p.addr, p.datai, p.en, p.we,
        p.clk.asClock)
  }
   */
}
```

Listing 11.15: A multi-clock memory generator.

Naturally, using these multi-clock memories introduces some constraints to the operations that can be performed simultaneously. Two (or more) ports cannot write to the same address at the same time as this may cause metastability. Similarly, one must make sure to define the wanted read-during-write behavior. The memory should be configured to either write first, in which the input data is forwarded to the read port, or read first, in which the *old* memory value is presented on the read port.

Beware, though, that multi-clock support in ChiselTest is still at a very early stage. You need to manually toggle clock signals to force transitions.

11.5 Exercises

This exercise section is a little bit longer as it contains two exercises: (1) exploring the bubble FIFO and implementing a different FIFO design; and (2) exploring the UART and extending it. The source code for both exercises is included in the chisel-examples repository.

11.5.1 Explore the Bubble FIFO

The FIFO source also includes a tester that provokes different read and write behavior and generates a waveform in the value change dump (VCD) format. The VCD file can be viewed with a waveform viewer, such as GTKWave. Explore the FifoSpec in the repository. The repository contains a `Makefile` to run the examples; for the FIFO example just type:

```
$ make fifo
```

This make command will compile the FIFO, run the test, and start GTKWave for waveform viewing.[3] Explore the tester and the generated waveform.

In the first cycle, the tester writes a single word. We can observe in the waveform how that word bubbles through the FIFO, therefore the name *bubble FIFO*. This bubbling also means that the latency of a data word through the FIFO is equal to the depth of the FIFO.

The next test fills the FIFO until it is full. A single read follows. Notice how the empty word bubbles from the reader side of the FIFO to the writer side. When a bubble FIFO is full, it takes a latency of the buffer depth for a read to affect the writer's side.

The end of the test contains a loop that tries to write and read at maximum speed. We can see the bubble FIFO running at maximum bandwidth, which is two clock cycles per word. A buffer stage has always to toggle between empty and full for a single-word transfer.

A bubble FIFO is simple and, for small buffers, has a low resource requirement. The main drawbacks of an n stage bubble FIFO are: (1) maximum throughput is one word every two clock cycles, (2) a data word has to travel n clock cycles from the writer's end to the reader's end, and (3) a full FIFO needs n clock cycles for the restart.

These drawbacks can be solved by a FIFO implementation with a circular buffer. The circular buffer can be implemented with a memory and read and write pointers. Rerun/rewrite the test with the other FIFO implementation and compare the bandwidth and latency. Synthesize the different FIFO versions and compare the resource requirements.

11.5.2 The UART

For the UART example, you need an FPGA board with a serial port and a serial port for your laptop (usually with a USB connection). Connect the serial cable between the FPGA board and the serial port on your laptop. Start a terminal program, e.g., Hyperterm on Windows or `gtkterm` on Linux:

```
$ gtkterm &
```

Configure your port to use the correct device; with a USB UART this is often something like `/dev/ttyUSB0`. Set the baud rate to 115200 and no parity or flow control (handshake). With the following command, you can create the Verilog code for the UART:

[3] Depending on your operating system, you might need to start GKTWave manually

```
$ make uart
```

Then use your synthesize tool to synthesize the design. The repository contains a Quartus project for the DE2-115 FPGA board. With Quartus, use the play button to synthesize the design and configure the FPGA. After configuration, you should see a greeting message in the terminal.

Extend the blinking LED example with a UART and write 0 and 1 to the serial line when the LED is off and on. Use the `BufferedTx`, as in the `Sender` example.

With the slow output of characters (two per second), you can write the data to the UART transmit register and can ignore the ready/valid handshake. Extend the example by writing repeated numbers 0-9 as fast as the baud rate allows. In this case, you must extend your state machine to poll the UART status to check if the transmit buffer is free.

The example code contains only a single buffer for the `Tx`. Feel free to add the FIFO that you have implemented to add buffering to the transmitter and receiver.

11.5.3 FIFO Exploration

Write a simple FIFO with four buffer elements in dedicated registers. Use 2-bit read and write counters, which can just overflow. As a further simplification, consider the situation when the read and write pointers are equal to empty FIFO. This means you can maximally store 3 elements. This simplification avoids the counter function from the example in Listing 11.12 and the handling of the empty or full with the same pointer values. We do not need empty or full flags, as this can be derived from the pointer values alone. How much simpler is this design?

The presented different FIFO designs have different design tradeoffs relative to the following properties: (1) maximum throughput, (2) fall through latency, (3) resource requirement, and (4) maximum clock frequency. Explore all FIFO variations in different sizes by synthesizing them for an FPGA; the source is available at ip-contributions. Where are the sweet spots for FIFOs of 4 words, 16 words, and 256 words?

12 Interconnect

We combine different components to build larger systems. To simplify the composition of components, interconnect standards such as Wishbone or AXI exists. This chapter explores different forms of interconnect.

Interconnects can be used between chips (external) or within a chip, often then called a system-on-chip (SoC).

12.1 A Classic Microprocessor Bus

Figure 12.1 shows the schematic of a simple, classic computer. The central processing unit (CPU) is connected via a system bus to external memory and input/output (I/O) devices. This type of bus interconnection was common with early microprocessors such as Z80 or 6502.

The bus is split into an address bus, a data bus, and control signals such as *read* and *write*. The CPU drives the address and control signals. The data bus is bidirectional. The CPU is the master in the system, and issues read or write commands. Both commands include an address (`addr` in the schematic) to select a data word from memory or a register from an I/O device. Not all address lines are connected to all peripheral devices. To select between different devices, the upper bits of the address bus are the input of a decoder. The outputs of the decoder are connected to the chip select (CS) inputs of the peripheral devices.

On a read command, the selected device will provide the data after some access time on the data bus. The peripheral device drives the data bus. On a write command, the processor provides the data, and the peripheral has to accept that data (often on a rising edge of a signal). The CPU drives the data bus. As the data bus is bi-directional and the data lines are shared between all devices, the outputs must contain a tri-state driver. In a tri-state configuration, both output transistors are disabled and the output pin is practically disconnected from logic.

Note that the bus does not have a clock in the simplest form. The timing is defined by read and write access times of the peripheral devices.

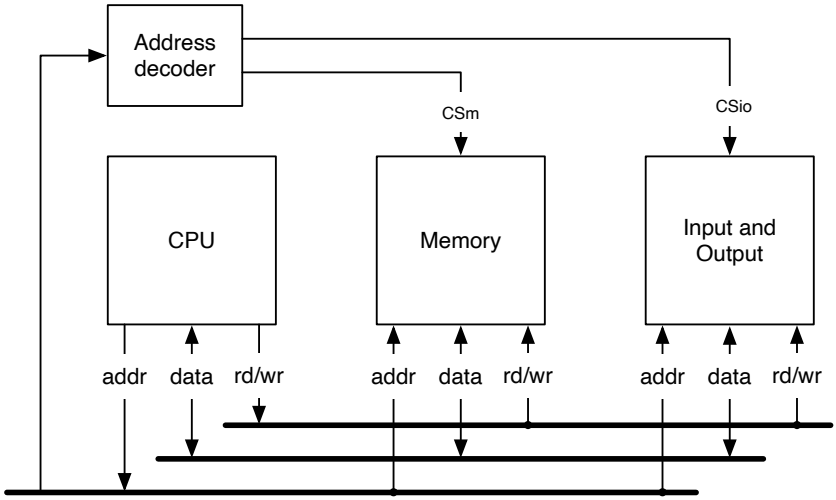


Figure 12.1: A classic computer consisting of a processor (CPU), memory, and I/O; connected via address, data, and control buses.

Modern computers have different buses for different peripheral devices, for example, a dedicated memory bus for external memory and I/O buses for peripheral devices. Furthermore, modern I/O buses, such as PCI Express, are serial buses and use point-to-point connections.

Nevertheless, the classic processor bus with an address bus, a data bus, and chip select signals is still the mainstream mindset for core interconnections. We will adapt this concept for on-chip interconnect in the next section.

12.2 An On-Chip Bus

We can translate the concept of such an external bus to an on-chip bus. However, we need to adapt some aspects. Shared buses requiring tri-state drivers are not practical within a chip. Furthermore, wires in a chip are cheaper than wires on a PCB or at a connector. Therefore, we split the data bus into two collections of wires: one for the read and one for the write signals. Furthermore, on-chip connections use a clock to define the timing.

Figure 12.2 shows the implementation of the bus concept within a chip. The

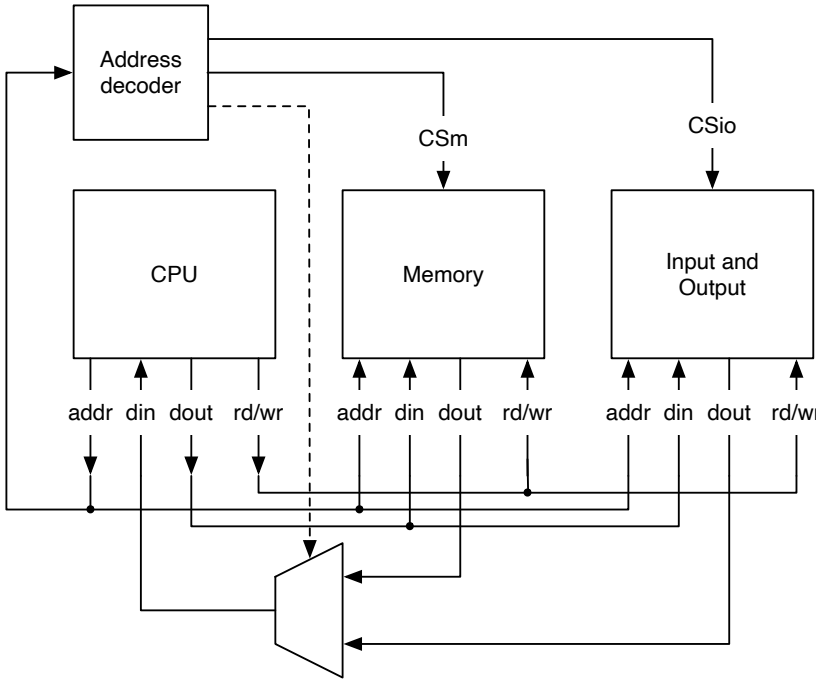


Figure 12.2: The translation of the off-chip bus concept to an on-chip “bus”.

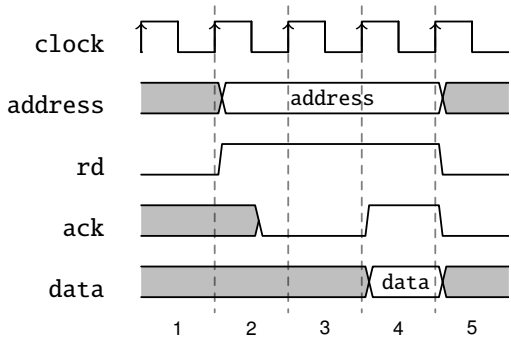


Figure 12.3: A read transaction with a combinational acknowledge.

address, data output, and control signals are connected from the CPU to all peripherals. We use a multiplexer (instead of a tri-state bus) for the data input. Besides generating the chip select signals, the address decoder drives the selection of the data input multiplexer.

With that simple setup, we assume that each operation (read or write) can be executed in a single clock cycle. This is only possible for very small systems. We can extend this by defining that we expect the read result in the next clock cycle following the read request. This fits well for on-chip memories with synchronous reads with one clock cycle latency. For IO devices, this additional clock cycle latency also relaxes the timing constraints. We still assume that a write is performed in one clock cycle.

If we want to communicate with devices with different or varying latency, we must introduce handshaking. The processor signals the start of a transaction with a read or write request, and the memory or peripheral device signals the end of a transaction with an acknowledgment signal.

12.2.1 Combinational Handshake

Figure 12.3 shows a read request with an acknowledgment. The processor drives the address bus (`address`) and the read signal (`rd`) in clock cycle 2. The signal `ack` needs to react within that first clock cycle. In our example, the read data is not available within one clock cycle but two clock cycles later, as seen in clock cycle 4. Data and the acknowledgment are valid for a single clock cycle. The benefit of that

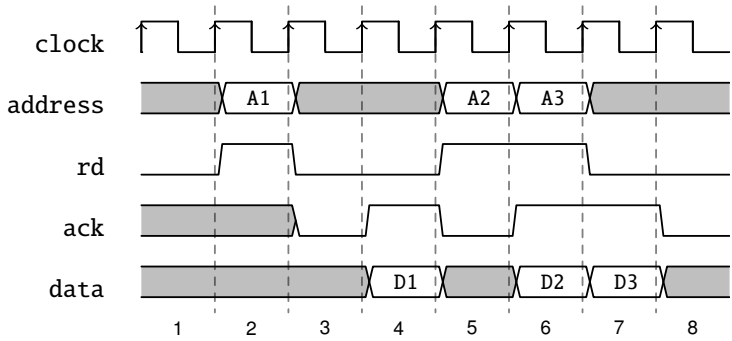


Figure 12.4: Read transaction with a pipelined acknowledgement.

protocol specification is that it allows for a single-cycle transaction. However, the price is that the handshake process, including decoding, is a combinational circuit, which can lead to issues with the maximum frequency. The standard Wishbone [16] protocol uses same-cycle acknowledgement. The newer version of Wishbone added a pipelined protocol.

Same-cycle acknowledgment (or ready signal) has been criticized in [17]. A single-cycle transaction is usually not realistic in a larger system. Therefore, we can define a specification where the acknowledge (or busy or ready) signal does not need to be valid in the request cycle. That paper proposes SimpCon, a protocol that enables pipelined transactions and avoids the combinational path between the processor, the address decoding, and the peripheral device.

12.2.2 Pipelined Handshake

TODO: Don't formulate it just relative to the other. Formulate it as its own thing. Maybe then compare. This needs some better writing, also put it into the soc-comm README. Also have a write timing diagram. And find a better font for the diagram to have nicer rendering in the SVG.

Here, we define a simple pipelined handshake bus protocol that avoids the single-cycle combinational loop and is better suited to modern SoC designs. A read or write command is signaled by an assertion of `rd` or `wr` for a single clock cycle. The address and write data (if it is a write) need to be valid during the command. Commands are only valid for a single cycle. Each command must be acknowledged

by an active ack, the earliest one cycle after the command. It can also insert wait states by delaying ack. Read data is available with the ack signal for one clock cycle.

Figure 12.4 shows a bus protocol that does not need a combinational reaction of the peripheral device. The request from the processor is only a single clock cycle long. The address bus and the read signal do not need to be driven until the acknowledgment. Compared to the former protocol, the ack signal needs to be valid (low or high) no earlier than one clock cycle after the rd command in clock cycle 3. In this example, the first read sequence has two clock cycles of latency. It has the same latency as in the former example. However, as the request needs to be valid for only one clock cycle, we can pipeline requests. Read of addresses A2 and A3 can be requested back to back, allowing a throughput of 1 data word per clock cycle.

We call this interconnect PipeCon to emphasize the pipelined nature of the interface. The interface is defined as follows:

```
class PipeCon(private val addrWidth: Int) extends Bundle {
  val address = Input(UInt(addrWidth.W))
  val rd = Input(Bool())
  val wr = Input(Bool())
  val rdData = Output(UInt(32.W))
  val wrData = Input(UInt(32.W))
  val wrMask = Input(UInt(4.W))
  val ack = Output(Bool())
}
```

The Patmos processor [27] uses an OCP version with exactly this protocol for accessing IO devices. Memory is connected via a burst interface. The Patmos Handbook [22] gives a detailed description of the used OCP interfaces. Furthermore, we have started a Chisel repository with multicore devices, such as a network-on-chip, that implement the described pipelined interface.

The on-chip version of an interconnect definition can be generalized to a point-to-point connection. The processor and peripheral devices are connected with such a point-to-point interface to a switching fabric. If the system contains more than one processor (or master) we need arbitration within the switching fabric to decide which master can issue read and write commands.

```
class CounterDevice extends Module {
  val io = IO(new PipeCon(4))

  val ackReg = RegInit(false.B)
  val addrReg = RegInit(0.U(2.W))
  val cntRegs = RegInit(VecInit(Seq.fill(4)(0.U(32.W))))

  ackReg := io.rd || io.wr
  when(io.rd) {
    addrReg := io.address(3, 2)
  }
  io.rdData := cntRegs(addrReg)

  for (i <- 0 until 4) {
    cntRegs(i) := cntRegs(i) + 1.U
  }
  when (io.wr) {
    cntRegs(io.address(3, 2)) := io.wrData
  }

  io.ack := ackReg
}
```

Listing 12.1: An IO device consisting of four loadable counters.

Address	Device
0x0000–0x0fff	ROM
0x1000–0x1fff	RAM
0xf000	UART
0xf010	LEDs
0xf020	Keys

Table 12.1: An example address mapping.

12.2.3 Example IO Device

Listing 12.1 shows an IO device that implements the specification of the pipelined interconnect. The IO device contains four loadable counters. To address those four counters, we need four address bits, as the addresses count in bytes, but the counters are 32-bit wide. We read the value from the counter with a read transaction (`rd` is asserted) and get the result in the next clock cycle (in `dout`). We write to a counter with `wr` asserted and the value set in `din`.

To implement the delayed acknowledge, we use a single bit register (`ackReg`) to delay any asserted `rd` or `wr`. As we provide the read result in the clock cycle that follows the read command and the address is only valid during this command cycle, we need to store the address in `addrReg`.

The counters themselves consist of a small register file of 4 elements (a `Reg` of a `Vec`). The counters are initialized to zero by using a Scala `Seq`, created with `fill` containing the reset values as Chisel constants. That `Seq` is the input to the `VecInit`. The counters are freely running and increment by one each clock cycle, except when a new value is written.

12.2.4 Memory Mapped Devices

Our example system connects all memory or IO devices to shared address lines. Therefore, they appear in the shared address space. To select individual devices, we use address decoding of some upper bits. These are called memory-mapped devices, and as part of the system design, we need to decide on the address mapping.

Table 12.1 shows an example address map for a (16-bit) microcontroller. We assume 16-bit addresses; therefore, the range of the addresses is between 0x0000 and 0xffff. At the lowest address (the start of the program to be executed), we map a read-only memory (ROM) that contains the program. In the next memory area,

Address	I/O Device	read	write
0xf000	UART	status	control
0xf001	UART	receive buffer	transmit buffer

Table 12.2: Address mapping for the UART.

Bit	Status	
0	TDRE	Transmit (TX) data register empty
1	RDRF	Receive (RX) data register full

Table 12.3: Status flags.

we map a writable memory (RAM) for the data. We decided to map all IO devices into the upper area of the address space (above 0xf000) so they are out of the way in case we want to extend the memory. In this example, we reserve 16 bytes of address for each IO device. Note that this is a made-up example and that we have all the flexibility when deciding on an address map.

Some IO devices do not have memory-mapped registers, as the counter device, but a ready/valid interface, as explained in Section 9.3. The UART, as presented in Section 11.2, for example, has two ready/valid interfaces: one for writing and one for reading a value. A common solution is to map the write and read channel to one address and drive the corresponding signals on the write or read command. We map those two signals into a status register at a different address to signal if the write channel is ready to receive a new data word or if the read channel has valid data.

Table 12.2 shows an address mapping for the UART. At the base address (0xf000) we access a status register on a read and an optional control register on a write. At the next address (0xf001), we read from a read buffer and write into a transmit buffer.

Table 12.3 shows the mapping of two flags into the status register. Both bits signal that we can perform a write or a read transaction. When the transmit data register is empty (TDRE), we can write (send) new data to the transmitter (TX). When the receive data register is full, we can read data from the receiver (RX). The terminology might sound a bit like using old terms. And this is true for our interface. This is precisely the mapping of the first serial port for the IBM PC built with the 8250 chip, and it is still valid.

Note that to use ready and valid in a status register for polling, the ready signal from the transmitter and the valid signal from the receiver are not allowed to be deasserted once they have been asserted. If this cannot be guaranteed, two single-word buffers, as shown in Listing 9.7, can be inserted between the IO interface and device with the read/valid interface.

TODO: Talk on what happens when reading from an empty device or writing to a full device.

For our memory-mapped devices, we define a bundle:

```
class MemoryMappedIO extends Bundle {
  val address = Input(UInt(4.W))
  val rd = Input(Bool())
  val wr = Input(Bool())
  val rdData = Output(UInt(32.W))
  val wrData = Input(UInt(32.W))
  val ack = Output(Bool())
}
```

Listing 12.2 shows the memory-mapped interface to a streaming device, like a serial port. The streaming device is connected to `tx` for the output and to `rx` for the input. The `mem` port is connected to a processor bus, using the pipelined handshake.

The device can be accessed with one clock cycle latency, as defined being the minimal access latency in the pipelined handshake. Therefore, we generate the `ack` signal when it was either a read or write one clock cycle later (`ackReg`).

The status register `statusReg` contains two flags: one for transmit channel ready (there is space in the send buffer) and receive channel valid (there is at least one byte in the receive buffer).

On read command, we store the read address in a register (`addrReg`) for use in the next clock cycle when we return the read value. Depending on the address we either return the value of the status register or the receive data. The write data is directly connected to the transmit channel, and the write signal (`wr`) signals a valid input.

To simplify the code example, we return from a read also if there is no data in the receive channel. An alternative would be to wait with the `ack` till data is available. Also in the send part, we ignore if there is room in the send buffer. We could as well block the write by not asserting `ack` until space is available in the send buffer. We delegate this check into software that reads the status register before trying to read or write a value.

TODO: decoding Chisel code, but maybe in the Leros section when building a complete system.

```
class MemMappedRV[T <: Data](gen: T, block: Boolean = false)
    extends Module {
  val io = IO(new Bundle() {
    val mem = new MemoryMappedIO()
    val tx = Decoupled(gen)
    val rx = Flipped(Decoupled(gen))
  })

  val statusReg = RegInit(0.U(2.W))
  val ackReg = RegInit(false.B)
  val addrReg = RegInit(0.U(1.W))
  val rdDlyReg = RegInit(false.B)

  statusReg := io.rx.valid ## io.tx.ready

  // ack
  ackReg := io.mem.rd || io.mem.wr
  io.mem.ack := ackReg

  // read from status or rx
  when (io.mem.rd) {
    addrReg := io.mem.address
  }
  rdDlyReg := io.mem.rd
  io.rx.ready := false.B
  when (addrReg === 1.U && rdDlyReg) {
    io.rx.ready := true.B
  }
  io.mem.rdData := Mux(addrReg === 0.U, statusReg,
      io.rx.bits)

  // write to tx
  io.tx.bits := io.mem.wrData
  io.tx.valid := io.mem.wr
}
```

Listing 12.2: An IO device for a ready/valid device.

TODO: Next step to multiple masters and a generic interconnect (cloud)

12.3 Bus and Interface Standards

Several point-to-point and bus standards have been proposed over the years. The following sections give a brief overview of common SoC interconnection standards.

12.3.1 Wishbone

The Wishbone [16] specification defines a point-to-point communication and not a bus in the classic sense. Wishbone is a public domain standard used by several open-source IP cores. The Wishbone interface specification is still in the tradition of microcomputers or backplane buses. However, for an SoC interconnect, which is usually point-to-point[1], this is not the best approach. The master is requested to hold the address and data valid through the whole read or write cycle. This complicates the connection to a master with data valid only for one cycle. In this case, the address and data must be registered *before* the Wishbone connect, or an expensive (in terms of time and resources) multiplexer must be used. A register results in one additional cycle of latency. A better approach would be to register the address and data in the slave. In that case, the address decoding in the slave can be performed in the same cycle as when the address is registered. A similar issue concerning the master exists for the output data from the slave: As it is only valid for a single cycle, the data has to be registered by the master when the master does not read it immediately. Therefore, the slave should keep the last valid data at its output even when the Wishbone strobe signal (*wb.stb*) is not assigned anymore. Holding the data in the slave is usually *free* in terms of hardware complexity—it is *just* a specification issue. In the classic Wishbone specification, there is no way to perform a pipelined read or write. However, the latest Wishbone specification (B4) also contains a pipelined definition. Note that the specification now includes two different, not necessarily compatible, specifications.

TODO: Sample code for a peripheral, how to test? Explain figure (text from other book)

[1] Multiplexers are used instead of buses to connect several slaves and masters.

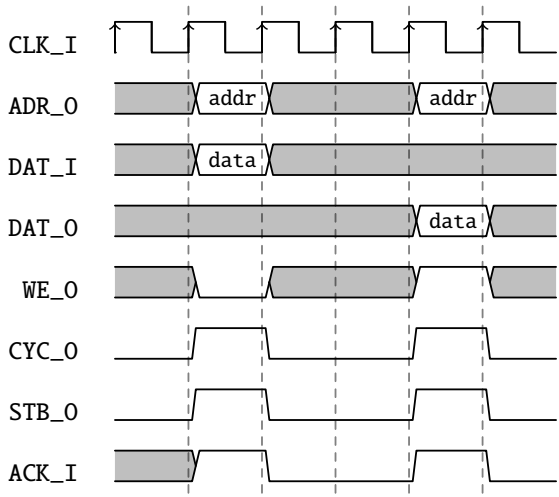


Figure 12.5: Wishbone asynchronous read followed by an asynchronous write.

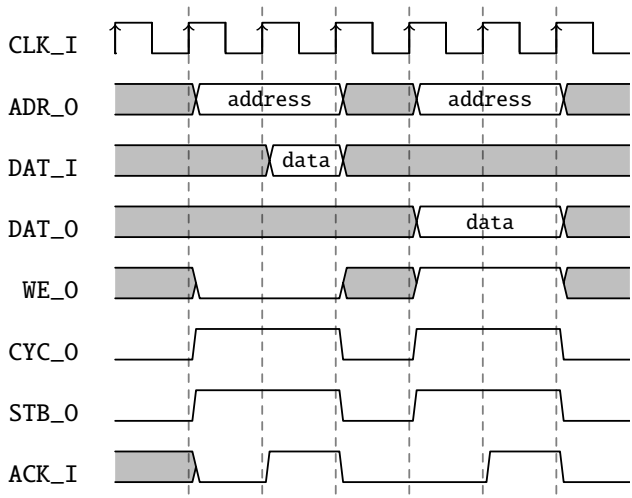


Figure 12.6: Wishbone synchronous read followed by a synchronous write.

12.3.2 AXI

The Advanced Microcontroller Bus Architecture (AMBA) [2] is an interconnection definition from ARM. The specification defines three different buses: Advanced High-performance Bus (AHB), Advanced System Bus (ASB) and Advanced Peripheral Bus (APB). The AHB connects on-chip memory, cache, and external memory to the processor. Peripheral devices are connected to the APB. A bridge connects the AHB to the lower-bandwidth APB. An AHB bus transfer can be one cycle with burst operation. With the APB, a bus transfer requires two cycles, and no burst mode is available. Peripheral bus cycles with wait states are added in version 3 of the APB specification. ASB is the predecessor of AHB and is not recommended for new designs (ASB uses both clock phases for the bus signals – very uncommon for today's synchronous designs).

Amba AXI (Advanced eXtensible Interface) and ACE version 4 [3] is the latest extension to AMBA. AXI introduces out-of-order transaction completion with the help of a 4-bit transaction ID tag. A ready signal acknowledges the transaction's start. The master has to hold the transaction information (e.g., address) till the interconnect signals are ready. This enhancement ruins the elegant single-cycle address phase from the original AHB specification.

The AXI bus uses ready/valid handshaking for all signals (read address, read data, write address, write data, and write response). The decoupling of the write address and the write data needs a more complex slave that can accept any order of arriving address and data.

TODO: Sample code for a servant, how to test? Probably with Xilinx stuff and maybe code from the zipcpu.

12.3.3 Open Core Protocol

Sonics Inc. defined the Open Core Protocol (OCP) [13] as an open, freely available standard. The standard is now handled by the OCP International Partnership (www.ocpip.org). The Patmos processor [27] and the T-CREST [21] multicore platform uses the OCP standard. The Patmos repository[2] contains several memory controllers, many peripherals devices, and a network-on-chip with an OCP interface.

TODO: Copy some code from Patmos for an example.

[2] https://github.com/t-crest/patmos

12.3.4 Further Bus Specifications

The Avalon [1] interface specification is provided by Intel for a system-on-a-programmable-chip interconnection. Avalon defines an extensive range of interconnection devices, from a simple asynchronous interface for direct static RAM connection to sophisticated pipeline transfers with variable latencies. This flexibility provides an easy path to connect a peripheral device to Avalon. How is this flexibility possible? The *Avalon Switch Fabric* translates between all those different interconnection types. The switch fabric is generated by Intel's SOPC Builder tool. However, it seems that this switch fabric is Intel proprietary, thus tying this specification to Intel FPGAs.

The On-Chip Peripheral Bus (OPB) [11] is an open standard provided by IBM and has been used by Xilinx several years ago. The OPB specifies a bus for multiple masters and slaves. The implementation of the bus is not directly defined in the specification. A distributed ring, a centralized multiplexer, or a centralized AND/OR network are suggested. Xilinx used the AND/OR approach, and all masters and slaves had to drive the data buses to zero when inactive. Xilinx switched to AXI for all their interconnects.

12.4 Exercise

Use the code shown in Listing 12.2 and a streaming device to the `rx` and `tx`. Or write a ChiselTest simulation of such a device. Write a testbench to test the memory interface. Explore what happens if the test ignores the status flags, i.e., reading invalid data or dropping write data when the transmit channel is not ready. Update the `MemeoryMappedRV` module so that the `ack` signal is delayed until the streaming device is ready or valid. Can your testbench handle the delayed `ack`? I you are simulating both the streaming device and the memory interface does the Scala code become a bit awkward, you need two software state machines to handle two devices in parallel? You can solve this issue elegant with multithreaded testing, as described in the testing chapter.

13 Debugging, Testing, and Verification

In Chapter 3, we gave a quick introduction to how to test Chisel designs. In this chapter, we will dig deeper into testing and verification.

Testing and verification have slightly different meanings in software development than in digital design. In software development, testing means running tests on components, whereas verification is usually short for formal verification (mathematical proofs or exhaustive testing with model checking). In digital design, we use the term testing similar to software when writing *test benches* that stimulate and check a *device under test* (DUT). However, the term testing is also used in the actual test of the physical chip (on a tester using built-in self-tests). Therefore, the digital design community is slowly moving toward using the term verification to test hardware descriptions. If we want to apply formal methods to verify hardware components, we call it formal verification. In this book, we will stick to the term *testing* when we write tests for our hardware description. Verification (testing) can be performed as dynamic verification by executing the design on a simulator or as formal verification with a model checker or an SMT solver.

13.1 Debugging

During your design and coding phase, you often debug your design. Debugging is the process of finding defects in your code. Those defects are called bugs. Debugging is often performed in parallel with writing new code.

One can debug a program by using a debugger or simply by printing interesting values to the terminal, called printf debugging. In hardware, elements are *executing* in parallel. Therefore, a common form of hardware debugging is generating waveforms and watching how signals of interest evolve over time. We call this *waveform debugging*.[1]

[1] There is no entry in Wikipedia for this; we should create one.

A Chisel tester can generate waveforms, which can be viewed, for example, with GTKWave. However, it is also possible to print signal values during circuit simulation for quick checks. Values are printed at the rising edge of the clock.

13.2 Testing in Chisel

ChiselTest is based on ScalaTest. Therefore, we can run all tests with a simple `sbt test`. ScalaTest also supports multithreaded testing out of the box, so if you have multiple test classes in your project, they can run in parallel. Additionally, you can use the `FlatSpec` syntax to write clear test descriptions and make debugging easier.

You define Chisel tests as classes that extend `AnyFlatSpec` with the `ChiselScalatestTester` trait. ChiselTest uses `peek`, `poke`, `expect`, and `step` methods that operate on the DUT's IO ports. The ChiselTest methods operate on Chisel types (i.e., `UInts`, `SInts`, and `Bools`). However, when peeking a value, we usually want Scala types for the test written in Scala. Therefore, two additional methods exist: (1) `peekInt()` returns a Scala `Int` and (2) `peekBoolean()` returns a Scala `Boolean`. To advance the simulation by one clock cycle, we call `step()` on the DUT's implicit clock port.

You can define a test within the `test()` function, which takes the module to test as a parameter. As an example, the following tester tests a few inputs to a BCD table:

```
class BcdTableTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "BCD table" should "output BCD encoded numbers" in {
    test(new BcdTable) { dut =>
      dut.io.address.poke(0.U)
      dut.io.data.expect("h00".U)
      dut.io.address.poke(1.U)
      dut.io.data.expect("h01".U)
      dut.io.address.poke(13.U)
      dut.io.data.expect("h13".U)
      dut.io.address.poke(99.U)
      dut.io.data.expect("h99".U)
    }
  }
}
```

Alternatively, you can use the `behavior of ``module name''` syntax to refer to the module with `it`. This is useful when you have several tests for a single module.

```
class BcdTableTest extends FlatSpec with
    ChiselScalatestTester {
  behavior of "BCD table"

  it should "output BCD encoded numbers" in {
    test(new BcdTable) { dut =>
      ...
    }
  }
}
```

Simple tests start by writing test vectors with `poke` to the DUT, advancing the clock, and testing the outputs with an `expect`. For debugging purposes, we can also `peek` values and print them out for manual inspection. The code in Listing 13.1 tests the counter device we introduced in Chapter 12 as an example IO device.

13.2.1 Use Functions

As you can see, the test covers only a few cases but is already very long to read. All those pokes and expects are cumbersome. As a first step, we shall introduce functions to represent a read and a write request. Those functions abstract away the manual "bit banging" at the interface pins in the testing code. Listing 13.2 shows a test with those functions. For a shortcut, we also define the function `step` to advance the clock.

The `read` function takes an address as a parameter and returns the read value. After poking the address and the read signal, the function advances the clock by one clock cycle and deasserts the read function. In our example device, the read value should be available after one clock cycle. However, we generalize the read function also for devices with longer latencies, and the read function waits in an endless loop that `ack` will become true. Note that we use `peekBoolean` to read a Scala `Boolean`. However, if a device fails to never assert `ack` after a request, the test will hang in an endless loop. A more robust read function shall contain a timeout for the `ack` polling. Finally, we read the data from `rdData` with `peekInt()` to read a Scala integer value (concrete a `BigInt` to express integers of any size).

The `write` function takes an address and the data parameters as Scala `Int`. Similar to the read function, the values are poked into the device, the clock is advanced by

```
"CounterDevice" should "work" in {
  test(new CounterDevice()) { dut =>
    dut.io.ack.expect(false.B)
    dut.clock.step()
    dut.io.address.poke(0.U)
    dut.io.rd.poke(true.B)
    dut.io.ack.expect(false.B)
    dut.clock.step()
    dut.io.rd.poke(false.B)
    dut.io.ack.expect(true.B)
    dut.clock.step(100)
    dut.io.rd.poke(true.B)
    dut.io.address.poke(4.U)
    dut.clock.step()
    assert(dut.io.rdData.peekInt() > 100)
    dut.io.wr.poke(true.B)
    dut.io.wrData.poke(0.U)
    dut.clock.step()
    dut.io.wr.poke(false.B)
    dut.io.rd.poke(true.B)
    dut.clock.step()
    dut.io.rdData.expect(1.U)
    dut.io.address.poke(0.U)
    dut.clock.step()
    assert(dut.io.rdData.peekInt() > 100)
  }
}
```

Listing 13.1: Testing the counter device.

```
"CounterDevice" should "work with functions" in {
  test(new CounterDevice()) { dut =>

    def step(n: Int = 1) = {
      dut.clock.step(n)
    }
    def read(addr: Int) = {
      dut.io.address.poke(addr.U)
      dut.io.rd.poke(true.B)
      step()
      dut.io.rd.poke(false.B)
      while (!dut.io.ack.peekBoolean()) {
        step()
      }
      dut.io.rdData.peekInt()
    }
    def write(addr: Int, data: Int) = {
      dut.io.address.poke(addr.U)
      dut.io.wrData.poke(data.U)
      dut.io.wr.poke(true.B)
      step()
      dut.io.wr.poke(false.B)
      while (!dut.io.ack.peekBoolean()) {
        step()
      }
    }

    for (i <- 0 until 4) {
      assert(read(i*4) < 10, s"Counter $i should have just
          started")
    }
    step(100)
    for (i <- 0 until 4) {
      assert(read(i*4) > 100, s"Counter $i should advance")
    }
    write(2*4, 0)
    write(3*4, 1000)
    assert(read(2*4) < 5, "Counter should reset")
    assert(read(3*4) > 1000, "Counter should load")
  }
}
```

Listing 13.2: Testing the counter device with fuctions.

one clock cycle, and the write signal deasserted. We also wait in an endless loop for `ack` to become true.

With those three functions available, we can write more readable tests with fewer lines of code. This testing code already covers more cases than the original bit-banging tester.[2]

13.2.2 Selecting Tests

If you have a large test suite, you may wish to run only a subset of your tests as part of a continuous integration run. The easiest way to achieve this and still have to run only a single SBT command is by tagging your tests.

```
object Unnecessary extends Tag("Unnecessary")

class TagTest extends AnyFlatSpec with Matchers {
  "Integers" should "add" taggedAs(Unnecessary) in {
    17 + 25 should be (42)
  }
}
```

By default, all tests are run using `sbt test` or `sbt testOnly *`. To leave out tests tagged with, for example, `Unnecessary`, you can run:

```
$ sbt "testOnly * -- -l Unnecessary"
```

When you run the command, the test will show up as ignored in the terminal:

```
[info] TagTest:
[info] Integers
...
[info] No tests were executed.
```

If your tests (and tags) are part of a package, remember to provide the full reference path to both.

[2] It happened that I had an off-by-one error (`until 3` instead of `until 4`) in the counter device that I found only with the second, more comprehensive test.

```
class TickGen extends Module {
  val io = IO(new Bundle {
    val tick = Output(Bool())
  })

  val cntReg = RegInit(0.U(8.W))
  cntReg := cntReg + 1.U
  io.tick := cntReg === 9.U
  when(io.tick) {
    cntReg := 0.U
  }
}
```

Listing 13.3: The tick generater as DUT.

13.2.3 Accessing Internal Signals

When testing a circuit, the test code usually has access only to the ports of the DUT. This abstraction is generally good practice, as access to internal signals and state is considered bad practice (in hardware and software testing).

However, it is sometimes beneficial to access the internal state. For example, testing a microprocessor with small assembler test programs. In that case, one would usually compare the state of the hardware implementation of the processor against a software-based simulator of the same processor. For a RISC-style processor comparing the content of the register file of the two implementations would be enough for this kind of testing, as all data that is computed, loaded, or stored passes the register file at some time. Another use case is to explore and test a state machine (with or without datapath) with access to the internal state.

TODO: Explain and explore the co-simulation in the Leros chapter.

To show the access to internal signals in action, we use a very minimal example: A tick generator with an internal counter. Listing 13.3 shows the code. Only the necessary signal `tick` is connected to an output port. The counter is not exposed, which is good design practice. For example, we want to access this internal counter in our test code. We could add a port to expose the counter. We could even use a Scala `Boolean` flag to conditionally have this port when we debug and disable it when generating hardware. However, mixing debugging code into the hardware description is not good practice.

```
class TickGenTestTop extends Module {
  val io = IO(new Bundle {
    val tick = Output(Bool())
    val counter = Output(UInt(8.W))
  })

  val tickGen = Module(new TickGen)
  io.tick := tickGen.io.tick
  io.counter := DontCare
  BoringUtils.bore(tickGen.cntReg, Seq(io.counter))
}
```

Listing 13.4: A top-level wrapper for our DUT.

To get access to internal signals, we can use the `BoringUtils`. `BoringUtils` allows us to *bore* a connection through a module hierarchy. Behind the scenes, it adds the additional needed ports throughout the hierarchy. It does exactly what we could do manually but avoids cluttering the original code.

At the time of this writing, `BoringUtils` is still considered experimental. Therefore, we need to include the following package when using it:

```
import chisel3.util.experimental.BoringUtils
```

To support additional ports, we wrap our DUT into another top-level module for testing. Listing 13.4 shows that top-level module with the additional port counter. Within the `TickGenTestTop`, we instantiate our original DUT and connect the `tick` port. For the `counter` output, first, we must assign a value to make the Chisel compiler happy. As we connect it to the inner module later, we connect it to `DontCare`. The next line connects the `io.counter` to the count register in `tickGen`. We need to wrap `io.counter` into a Scala `Seq`, as `boring()` supports connecting to several signals.

Finally, Listing 13.5 shows the testing code for our DUT. Note that we instantiate the top-level wrapper to use the additional output port.

13.2.4 Multithreaded Testing

Digital hardware is inherently parallel. It is also helpful to represent parallelism in the testing code to test such a parallel circuit. For example, one thread fills data into

```
  test(new TickGenTestTop()) { dut =>
    dut.io.tick.expect(false.B)
    dut.io.counter.expect(0.U)

    dut.clock.step()
    dut.io.tick.expect(false.B)
    dut.io.counter.expect(1.U)

    dut.clock.step(8)
    dut.io.tick.expect(true.B)
    dut.io.counter.expect(9.U)

    dut.clock.step()
    dut.io.tick.expect(false.B)
    dut.io.counter.expect(0.U)
  }
```

Listing 13.5: Testing the DUT with access to internal signals.

a circuit while another thread checks the outputs from that circuit. We could do this in a single thread, but then we need to mix code for different tasks into one function that shares the advancement of the clock. With multithreaded tests, each thread can independently advance the clock. The threads are synchronized internally at the call of `clock()`.

ChiselTest supports multithreaded testing by using `fork` and `join` calls. `fork` spawns a new tester thread with a block of test code as its parameter, while `join` may be called on a tester thread variable (returned by a `fork` call) to wait for it to join the main thread.

Running multiple threads does present some new limitations to `peek`s and `poke`s in that no two threads can `peek` (respectively, `poke`) the same signal at the same time. Similarly, the threads are synchronized on calls to `step` to guarantee correct operation. The following snippet is a small test of a FIFO that enqueues an element in one thread and dequeues it in the main thread:

```
it should "work with multiple threads" in {
  test(new BubbleFifo(8, 4)) { dut =>
    val enq = fork {
      while (dut.io.enq.full.peekBoolean())
```

```
        dut.clock.step()
      dut.io.enq.din.poke(42.U)
      dut.io.enq.write.poke(true.B)
      dut.clock.step()
      dut.io.enq.write.poke(false.B)
    }
    while (dut.io.deq.empty.peekBoolean())
      dut.clock.step()
    dut.io.deq.dout.expect(42.U)
    dut.io.deq.read.poke(true.B)
    dut.clock.step()
    dut.io.deq.empty.expect(true.B)
    enq.join()
  }
}
```

Multiple threads are spawned with stacked calls to `fork`. The spawned threads represent a hierarchy in which the first thread should not finish before any of the subsequent threads.

13.2.5 Simulator Backends

By default, tests written with ChiselTest are run by the Treadle simulation backend. The benefits of Treadle are its quick startup time and the fact that it does not require any additional tools to be installed.

However, large system tests may either require another backend to support, for example, latches or may benefit in terms of simulation time. To enable this, ChiselTest supports two other backends: Verilator and Synopsys VCS. Because Verilator is open-source, we will use it for the examples presented in this section. Note that VCS can be an alternative to Verilator in all shown cases.

Switching to a different backend adds another annotation to the `withAnnotations` call, as shown in the Waveforms section. To use Verilator, add the following annotation:

```
test(new
    Dut()).withAnnotations(Seq(VerilatorBackendAnnotation))
    {
```

Additional flexibility arises from supplying your switches to the simulator command that starts the backend. This is done by using `VerilatorFlags` to add switches

to the Verilator simulation command, or `VerilatorCFlags` to add switches to GCC. They should be in the list of annotations along with the backend annotation. You need to refer to the tool's user manual to find a detailed list of command-line arguments. Note that `VerilatorFlags` and `VerilatorCFlags` annotations are advanced features that should generally not be needed. Furthermore, the flags are not guaranteed to remain stable.

Note that ChiselTest 0.3.4 and later support code coverage measures directly in simulation. To support this, install version 4.028 or a newer version of Verilator.

Also, beware that different simulators work in different ways. Verilator is a synchronous simulator, meaning it runs updates only at the rising edge of the clock and thus does not support latches. It also does not officially support multiple clocks. VCS, on the other hand, is an event-based simulator, which is significantly more detailed in its simulations and supports all synthesizable Verilog constructs. Generally, for single-clock circuits, Verilator is the fastest and most widely available tool.

13.3 Assertions and Formal Verification

An assertion statement in a programming language allows one to state assumptions about a program. If the assertion condition evaluates to `false`, the program usually stops with an exception. Chisel also supports assertions to state assumptions in the hardware. Those assertions are tested when simulating the digital design. The simulation stops with an error message if the condition evaluates to `false`. Assertions are ignored when generating hardware, as there is no (easy) way that hardware can communicate a failing assertion.

Listing 13.6 shows an example of using assertions. These are trivial assertions to show how to use them. If we would make an error and use subtraction instead off addition or reassigning `sum` later a different value, we would catch that error during testing and get a similar error message as below:

```
Assertion failed
    at Assert.scala:20 assert(sum <= a + b)
```

However, as you can read in the comments, the first two assertions are not always true, so we cannot use them. We found this out by using formal verification. I would propose to place all assertions at the end of a module, so they do not clutter the reading of the intended design.

```
class Assert extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val sum = Output(UInt(8.W))
  })
  io.sum := io.a + io.b

  /* the following will not be true when
  the addition overflows
  assert(io.sum >= io.a)
  assert(io.sum >= io.b)
   */
  assert(io.sum === io.a + io.b)
}
```

Listing 13.6: Using assertions in Chisel.

TODO: Write on overlooking edge cases, just use too small numbers and not test the overflow.

The assertions are executed during simulation; we must write the test cases for them. However, writing tests to trigger all possibilities is hard (impossible). To catch errors in overlooked corner cases, we can use formal verification. Kevin Laeufer has added formal verification to ChiselTest [12]. The very same assertions are used for formal verification.

To explore formal verification with Chisel, install the open Z3 theorem prover. To use formal verification you substitute `test(..)` by `verify(..)`. Listing 13.7 shows how to run formal verification on our simple adder circuit with the (naive) assertions.

It surprised the author that Chisel formal immediately found an error in the circuit or the assertions. To investigate the issue, we can look into the waveform to find the input data that leads to an assertion violation. It uses 0xdb and 0x65 as inputs, which results in a sum of 0x40. The input values led to an overflow of the 8-bit addition. With our simple test case, we missed testing overflowing values. This verification showed that the assertions that the sum is larger or equal to the inputs are wrong due to possible overflow.

TODO: read on https://github.com/tdb-alcorn/chisel-formal. How

```
"AssertTest" should "pass" in {
  verify(new Assert(), Seq(BoundedCheck(5),
      WriteVcdAnnotation))
}
```

Listing 13.7: Formally verifying the circuit.

is this related to Kevin's work?
TODO: What are the limitations?

13.4 Exercise

Extreme programming is an agile software development style, focusing on quick turnaround times and a strong dependency on unit tests. In its pure form, one writes the tests first before implementing a feature. This style is not used so often in real life. However, exploring it may help to focus on testing as an important part of developing artifacts.

Therefore, the proposed exercise is to write test benches for designs you have not yet implemented. Pick one of the small projects from Chapter 7, e.g., the debouncing circuit or the majority-based filtering design, and write tests for it. Then, implement the hardware design itself.

Explore the experience of this little experiment. Did you implement tests that found errors in your design? If all tests pass, are you sure you have tests that cover a reasonable design space? How do you test your tests? Add a fault into your DUT and see if your tests will catch it.

As you work through this exercise, you may experience an unpleasant feeling that testing is hard and it is probably impossible to catch all errors.[3] However, there is hope in recent developments in formal verification to complement testing. The topic of formal verification with Chisel will be extended in a future edition of this book.

[3] A famous quote by Dijkstra is, "Program testing can be used to show the presence of bugs, but never to show their absence!"

14 Design of a Processor

As one of the last chapters in this book, we present a medium-sized project: The design, simulation, and testing of a microprocessor. To keep this project manageable, we design a simple accumulator machine. The processor is called Leros [25] and is available in open source at `https://github.com/leros-dev/leros`. We would like to mention that this is an advanced example, and some computer architecture knowledge is needed to follow the presented code examples.

14.1 The Instruction Set Architecture

The definition of an instruction set is also called instruction set architecture (ISA). The ISA is the most important abstraction in computer architecture. The ISA is the contract between the compiler (or assembler programmer) and the concrete processor implementation. The ISA is independent of the actual implementation. Different implementations of a microprocessor (also called microarchitecture) can execute the same ISA.

Leros is designed to be simple but still a good target for a C compiler. The description of the instructions fits on one page; see Table 14.1.

Leros is an accumulator machine. This means that all operations have the accumulator as one of the source inputs, and the result is usually written into the accumulator. The second operand of an arithmetic or logic operation can either be an immediate (a constant) or from one of the 256 on-chip registers. Access to memory (load or store) is also performed via the accumulator: on a load, the value from memory is stored in the accumulator, and for a store, the value is taken from the accumulator. The address for the memory access is stored in the address register (AR).

The program counter (PC) is pointing to the current instruction in the instruction memory. The PC is usually incremented to the following instructions. To implement control flow, the PC can be manipulated by a branch or jump instruction. Leros has unconditional and conditional branch instructions. Conditional branches depend on the content of the accumulator. E.g., `brz` branches only when the content of the ac-

Opcode	Function	Description
add	A = A + Rn	Add register Rn to A
addi	A = A + i	Add immediate value i to A
sub	A = A - Rn	Subtract register Rn from A
subi	A = A - i	Subtract immediate value i from A
shr	A = A >>> 1	Shift A logically right
load	A = Rn	Load register Rn into A
loadi	A = i	Load immediate value i into A
and	A = A and Rn	And register Rn with A
andi	A = A and i	And immediate value i with A
or	A = A or Rn	Or register Rn with A
ori	A = A or i	Or immediate value i with A
xor	A = A xor Rn	Xor register Rn with A
xori	A = A xor i	Xor immediate value i with A
loadhi	A_{15-8} = i	Load immediate into second byte
loadh2i	A_{23-16} = i	Load immediate into third byte
loadh3i	A_{31-24} = i	Load immediate into fourth byte
store	Rn = A	Store A into register Rn
jal	PC = A, Rn = PC + 2	Jump to A and store return address
ldaddr	AR = A	Load address register AR with A
loadind	A = mem[AR+(i << 2)]	Load a word from memory into A
loadindb	A = mem[AR+i]$_{7-0}$	Load a byte from memory into A
loadindh	A = mem[AR+(i << 1)]$_{15-0}$	Load a half word from memory into A
storeind	mem[AR+(i << 2)] = A	Store A into memory
storeindb	mem[AR+i] = A_{7-0}	Store a byte into memory
storeindh	mem[AR+(i << 1)] = A_{15-0}	Store a half word into memory
br	PC = PC + o	Branch
brz	if A == 0 PC = PC + o	Branch if A is zero
brnz	if A != 0 PC = PC + o	Branch if A is not zero
brp	if A >= 0 PC = PC + o	Branch if A is positive
brn	if A < 0 PC = PC + o	Branch if A is negative
scall		System call (simulation hook)

Table 14.1: Leros instruction set.

cumulator is 0. Leros branches are relative to the current instruction and can branch forward and backward around 2000 instructions. For larger control flow changes and function calls and returns, Leros has a jump-and-link (`jal`) instruction. That instruction jumps to the address that is in the accumulator and stores the address of the following instruction into a register. That value can then be used to return from a function with `jal`.

The accumulator and the register file are in our current implementation 32-bit wide.[1]

Table 14.1 shows the instruction set of Leros. `A` represents the accumulator, `PC` is the program counter, `i` is an immediate value (0 to 255), `Rn` a register `n` (0 to 255), `o` a branch offset relative to the `PC`, and `AR` an address register for memory access.

The following code snippet show examples of Leros' instructions in assembly:

```
loadi 1
addi 2
ori 0x50
andi 0x1f
subi 0x13
loadi 0xab
addi 0x01
subi 0xac

scall 0
```

We can see that each instruction consists of the instruction name (also called opcode mnemonic) and a constant. The constant can be written in decimal or hexadecimal notion. The code shows immediate versions of load, arithmetic, and logic instructions. The last instruction (`scall 0`) is a system call and ends the execution (or simulation). This short program is part of the Leros test suit. The convention of the test is that at the end of the program, the accumulator shall contain 0.

Instructions are 16 bits wide. The higher byte is used to encode the instruction, and the lower byte contains either an immediate value, a register number, or a branch offset (part of the branch offset also uses bits in the upper byte). For example, `00001001.00000010` is an add immediate instruction that adds 2 to the accumulator, whereas `00001000.00000011` adds the content of R3 to the accumulator. For branches, we use 3 of the instruction bits for larger offsets.

Listing 14.1 shows the encoding of the instructions in the upper 8 bits of each instruction. Not all instruction bits are currently used (unused are marked with -)

[1] We try to keep it configurable also to implement 16-bit or 64-bit versions of Leros.

```
+--------+----------+
|00000---| nop      |
|000010-0| add      |
|000010-1| addi     |
|000011-0| sub      |
|000011-1| subi     |
|00010---| sra      |
|00011---| -        |
|00100000| load     |
|00100001| loadi    |
|00100010| and      |
|00100011| andi     |
|00100100| or       |
|00100101| ori      |
|00100110| xor      |
|00100111| xori     |
|00101001| loadhi   |
|00101010| loadh2i  |
|00101011| loadh3i  |
|00110---| store    |
|001110-?| out      |
|000001-?| in       |
|01000---| jal      |
|01001---| -        |
|01010---| ldaddr   |
|01100-00| ldind    |
|01100-01| ldindb   |
|01100-10| ldindh   |
|01110-00| stind    |
|01110-01| stindb   |
|01110-10| stindh   |
|1000nnnn| br       |
|1001nnnn| brz      |
|1010nnnn| brnz     |
|1011nnnn| brp      |
|1100nnnn| brn      |
|11111111| scall    |
+--------+----------+
```

Listing 14.1: Leros instruction encoding.

14.2 The Datapath

Section 9.2 describes how to implement an algorithm in hardware with a state machine and datapath. We will use the same approach for the initial implementation of Leros.

We need a datapath that allows the data flow for all instructions, possible in several clock cycles. We aim for a executing each instruction in two clock cycles. Therefore, the base state machine has two states: `fetch` and `execute`.

Figure 14.1 shows the datapath for implementing Leros. The figure is slightly simplified. The data flows from left to right. The PC points to the instruction to be fetched. On-chip memories usually have input registers that cannot be read.[2] Therefore, we feed to the PC and the input register of the instruction memory the same value as the next PC. The next PC is for non-branching instructions the PC plus 1.[3] For a relative branch, the decode component sign extends the immediate value and adds it to the PC. For the `jal` instruction, the PC can also be loaded from `A`.

In the first state, an instruction is fetched from the instruction memory and decoded. The decode component decides what will happen in the next state, the execute state. The decode also includes the generation of an operand for instructions with an immediate operand (e.g., `addi`, or `loadhi`). As this operand is consumed in the execution state, we must store it in a register.

The second memory serves as general data memory but also stores the values for the 255 registers. The address is part of the instruction for a read or write of one of the registers. We use the address register `AR` for memory load or store instructions. `AR` itself is loaded from `A`. The result of a load is placed into `A`. On a store, `A` delivers the data to be written into memory.

Finally, arithmetic and logic operations are performed with the ALU. One operand comes from `A`, the other is either an immediate value (from the instruction) or a register value (from the memory).

14.3 Start with an ALU

A central component of a processor is the arithmetic logic unit, or ALU for short. Therefore, we start with the coding of the ALU and a test bench. First, we define constants to represent the different operations of the ALU:

[2] At least this is true for FPGA on-chip memories

[3] We count in this simple organization in 16-bit instruction words and not in bytes

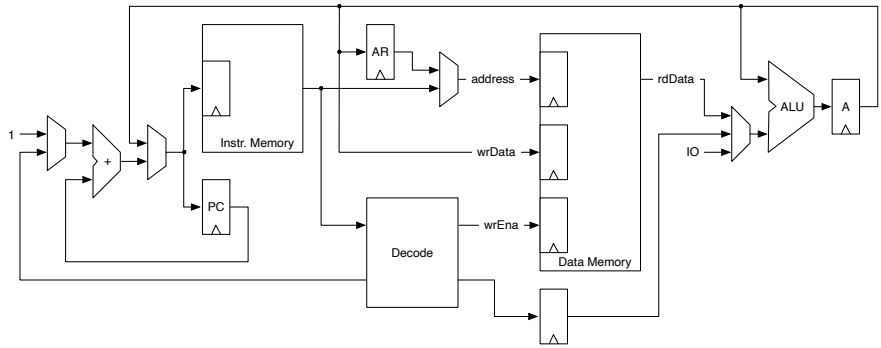


Figure 14.1: The Leros datapath.

```
// Alu ops
val nop = 0
val add = 1
val sub = 2
val and = 3
val or = 4
val xor = 5
val ld = 6
val shr = 7
```

An ALU usually has two operand inputs (call them `a` and `b`), an operation `op` (or opcode) input to select the function and an output `y`. Listing 14.2 shows the ALU.

TODO: draw a nice ALU, see Wikipedia

We first define shorter names for the three inputs. The `switch` statement defines the logic for the computation of `res`. Therefore, it gets a default assignment of 0. The switch statement enumerates all operations and assigns the expression accordingly. All operations map directly to a Chisel expression. In the end, we assign the result `res` to the ALU output `y`

```
class AluAccu(size: Int) extends Module {
  val io = IO(new Bundle {
    val op = Input(UInt(3.W))
    val din = Input(UInt(size.W))
    val enaMask = Input(UInt(4.W))
    val enaByte = Input(Bool())
```

```
  val enaHalf = Input(Bool())
  val off = Input(UInt(2.W))
  val accu = Output(UInt(size.W))
})

val accuReg = RegInit(0.U(size.W))

val op = io.op
val a = accuReg
val b = io.din
val res = WireDefault(a)

switch(op) {
  is(nop.U) {
    res := a
  }
  is(add.U) {
    res := a + b
  }
  is(sub.U) {
    res := a - b
  }
  is(and.U) {
    res := a & b
  }
  is(or.U) {
    res := a | b
  }
  is(xor.U) {
    res := a ^ b
  }
  is(shr.U) {
    res := a >> 1
  }
  is(ld.U) {
    res := b
  }
}

val byte = WireDefault(res(7, 0))
val half = WireDefault(res(15, 0))
```

```
  when(io.off === 1.U) {
    byte := res(15, 8)
  }.elsewhen(io.off === 2.U) {
    byte := res(23, 16)
    half := res(31, 16)
  }.elsewhen(io.off === 3.U) {
    byte := res(31, 24)
  }
  val signExt = Wire(SInt(32.W))
  when (io.enaByte) {
    signExt := byte.asSInt
  } .otherwise {
    signExt := half.asSInt
  }

  // Workaround for missing subword assignments
  val split = Wire(Vec(4, UInt(8.W)))
  for (i <- 0 until 4) {
    split(i) := Mux(io.enaMask(i), res(8 * i + 7, 8 * i),
        accuReg(8 * i + 7, 8 * i))
  }

  when((io.enaByte || io.enaHalf) & io.enaMask.andR) {
    accuReg := signExt.asUInt
  } .otherwise {
    accuReg := split.asUInt
  }

  io.accu := accuReg
}
```

Listing 14.2: The Leros ALU with the accumulator register.

For the testing, we write the ALU function in plain Scala, as shown in Listing 14.3.
While this duplication of hardware written in Chisel and Scala implementation does not detect errors in the specification; it is at least some sanity check. We use some corner case values as the test vector:

```
def alu(a: Int, b: Int, op: Int): Int = {

  op match {
    case 0 => a
    case 1 => a + b
    case 2 => a - b
    case 3 => a & b
    case 4 => a | b
    case 5 => a ^ b
    case 6 => b
    case 7 => a >>> 1
    case _ => -123 // This shall not happen
  }
}
```

Listing 14.3: The Leros ALU function written in Scala.

```
// Some interesting corner cases
val interesting = Seq(1, 2, 4, 123, 0, -1, -2,
    0x80000000, 0x7fffffff)
```

We define a function (`testOne()`) to test one pair of inputs.

```
def testOne(a: Int, b: Int, fun: Int): Unit = {
  dut.io.op.poke(ld.U)
  dut.io.enaMask.poke("b1111".U)
  dut.io.din.poke((a.toLong & 0x00ffffffL).U)
  dut.clock.step(1)
  dut.io.op.poke(fun.U)
  dut.io.din.poke((b.toLong & 0x00ffffffL).U)
  dut.clock.step(1)
  dut.io.accu.expect((alu(a, b, fun.toInt).toLong &
      0x00ffffffL).U)
}
```

Then we test all functions with those values on both inputs:

```
def test(values: Seq[Int]) = {
  for (fun <- 0 to 7) {
    for (a <- values) {
```

```
            for (b <- values) {
              testOne(a, b, fun)
            }
          }
        }
      }
```

Full, exhaustive testing for 32-bit arguments is not possible, which was why we selected some corner cases as input values. Besides testing against corner cases, it is also useful to test against random inputs:

```
    val randArgs =
        Seq.fill(10)(scala.util.Random.nextInt())
    test(randArgs)
```

You can run the tests within the Leros project with

```
$ sbt "testOnly leros.AluAccuTest"
```

The test shall produce a success message similar to:

```
[info] AluAccuTest:
[info] AluAccu
[info] - should pass
[info] Run completed in 1 second, 794 milliseconds.
[info] Total number of tests run: 1
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

14.4 Decoding Instructions

From the ALU, we work backward and implement the instruction decoder. Instruction decoding is generating the signals to drive the multiplexers and the ALU in the next stage/state. However, first, we define the instruction encoding in its own Scala class and a *shared* package. We want to share the encoding constants between the hardware implementation of Leros, an assembler for Leros, and an instruction set simulator.

```
object Constants {
```

```
  val NOP = 0x00
  val ADD = 0x08
  val ADDI = 0x09
  val SUB = 0x0c
  val SUBI = 0x0d
  val SHR = 0x10
  val LD = 0x20
  val LDI = 0x21
  val AND = 0x22
  val ANDI = 0x23
  val OR = 0x24
  val ORI = 0x25
  val XOR = 0x26
  val XORI = 0x27
  val LDHI = 0x29
  val LDH2I = 0x2a
  val LDH3I = 0x2b
  val ST = 0x30
  // ...
```

TODO: Update code when Leros is more complete, as stuff is missing.

For the decode component, we define a `Bundle` for the output, which is later used in the execution state and fed partially into the ALU. The `DecodeOut Bundle` contains more fields, not showing here. See the original Leros code base for the details.

```
class DecodeOut extends Bundle {
  val operand = UInt(32.W)
  val enaMask = UInt(4.W)
  val op = UInt()
  val off = SInt(10.W)
  val isRegOpd = Bool()
  val useDecOpd = Bool()
  val isStore = Bool()
  // ... and more fields
```

We also define a companion object for the `DecodeOut` class that includes a function `default()` to create a `DecodeOut` object and sets all fields to default values.

```
object DecodeOut {

  val MaskNone = "b0000".U
```

```
  val MaskAll = "b1111".U

  def default: DecodeOut = {
    val v = Wire(new DecodeOut)
    v.operand := 0.U
    v.enaMask := MaskNone
    v.op := nop.U
    v.off := 0.S
    v.isRegOpd := false.B
    v.useDecOpd := false.B
    v.isStore := false.B
    // ... and more fields
```

Decode takes as input an 8-bit opcode and delivers the decoded signals as output. Those driving signals are assigned a default value by using the `default` function to create that object.

```
class Decode() extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(16.W))
    val dout = Output(new DecodeOut)
  })

  import DecodeOut._

  val d = DecodeOut.default
```

The decoding itself is just a large switch statement on the part of the instruction that represents the opcode (in Leros for most instructions the upper 8 bits.)

```
  switch(instr(15, 8)) {
    is(ADD.U) {
      d.op := add.U
      d.enaMask := MaskAll
      d.isRegOpd := true.B
    }
    is(ADDI.U) {
      d.op := add.U
      d.enaMask := MaskAll
      d.useDecOpd := true.B
    }
```

```
is(SUB.U) {
  d.op := sub.U
  d.enaMask := MaskAll
  d.isRegOpd := true.B
}
is(SUBI.U) {
  d.op := sub.U
  d.enaMask := MaskAll
  d.useDecOpd := true.B
}
is(SHR.U) {
  d.op := shr.U
  d.enaMask := MaskAll
}
// ...
```

Additionally, the decode module also generates sign extended version of the constant in the instruction and computes the offset for the indirect load and store instructions.

TODO: there is more on decoding TODO: and the whole execution is missing

14.5 Assembling Instructions

To write programs for Leros, we need an assembler. However, for the very first test, we can hard code a few instructions and put them into a Scala array, which we use to initialize the instruction memory.

```
val prog = Array[Int](
  0x0903, // addi 0x3
  0x09ff, // -1
  0x0d02, // subi 2
  0x21ab, // ldi 0xab
  0x230f, // and 0x0f
  0x25c3, // or 0xc3
  0x0000
)

def getProgramFix() = prog
```

However, this is a very inefficient approach to testing a processor. Writing an assembler with an expressive language like Scala is not a big project. Therefore, we write a simple assembler for Leros, which is possible within about 100 lines of code. We define a function `getProgram` that calls the assembler. We need a symbol table for branch destinations, which we collect in a `Map`. A classic assembler runs in two passes: (1) collects the values for the symbol table and (2) assembles the program with the symbols collected in the first pass. Therefore, we call `assemble` twice with a parameter to indicate which pass it is.

```
def getProgram(prog: String) = {
  assemble(prog)
}

// collect destination addresses in first pass
val symbols = collection.mutable.Map[String, Int]()

def assemble(prog: String): Array[Int] = {
  assemble(prog, false)
  assemble(prog, true)
}
```

The `assemble` function starts with opening the source file and defining two helper functions to parse the two possible operands: (1) an integer constant (allowing decimal or hexadecimal notation) and (2) to read a register number.

```
def assemble(prog: String, pass2: Boolean): Array[Int] = {

  val source = Source.fromFile(prog)
  var program = List[Int]()
  var pc = 0

  def toInt(s: String): Int = {
    if (s.startsWith("0x")) {
      Integer.parseInt(s.substring(2), 16)
    } else {
      Integer.parseInt(s)
    }
  }

  def regNumber(s: String): Int = {
    assert(s.startsWith("r"), "Register numbers shall
```

```
        start with \'r\'")
    s.substring(1).toInt
  }
```

Listing 14.4 shows the core of the assembler for Leros. A Scala `match` expression covers the core of the assembly function. *TODO: Some more words on the code.*

14.6 The Instruction Memory

Listing 14.5 shows the instruction memory module for Leros. The memory is configured with the size as the number of address bits (`memAddrWidth`) and a path to the program (`prog`). The constructor of the instruction memory calls the assembler, shown in the previous section, to assemble the program. This is an example of a hardware generator that assembles code for an embedded processor during hardware generation. The Scala array that contains the Leros program is converted to a Scala `Seq` and than mapped to a Chisel `Vec` with the anonymous function `_.asUInt(16.W)`. The instruction memory contains a register for the address (`memReg`) to enable the implementation of the instruction memory as an on-chip memory in an FPGA.[4]

14.7 A State Machine with Data Path Implementation

As stated at the beginning of the chapter, the Leros ISA definition does not define a concrete implementation. Throughout the chapter, we made implicit design decisions. Here, we will discuss one design option.

In the presented implementation, we share the data memory with the register file. The 256 registers are just an array in the data memory. A different implementation might have dedicated on-chip memories for data and the registers.

We implemented Leros in the idea of a state machine with a datapath. This also means that each instruction takes more than one clock cycle. We use two states: `fetch` and `execute`, as shown in Listing 14.6.

The state machine switches between the two states. In the `fetch` state, we fetch an instruction from the instruction memory and also decode that instruction. We also

[4] In the current version of Chisel, the generated code contains a large priority Mux that the FPGA synthesize tools cannot map to an on-chip memory. Using the MLIR backend shall fix this issue. Another workaround for this issue, as done in the Patmos project for the bootloader is to generate Verilog code that fits the synthesized tools and include it as a black box.

```
  for (line <- source.getLines()) {
    if (!pass2) println(line)
    val tokens = line.trim.split(" ")
    val Pattern = "(.*:)".r
    val instr = tokens(0) match {
      case "//" => // comment
      case Pattern(l) => if (!pass2) symbols +=
          (l.substring(0, l.length - 1) -> pc)
      case "add" => (ADD << 8) + regNumber(tokens(1))
      case "sub" => (SUB << 8) + regNumber(tokens(1))
      case "and" => (AND << 8) + regNumber(tokens(1))
      case "or" => (OR << 8) + regNumber(tokens(1))
      case "xor" => (XOR << 8) + regNumber(tokens(1))
      case "load" => (LD << 8) + regNumber(tokens(1))
      case "addi" => (ADDI << 8) + toInt(tokens(1))
      case "subi" => (SUBI << 8) + toInt(tokens(1))
      case "andi" => (ANDI << 8) + toInt(tokens(1))
      case "ori" => (ORI << 8) + toInt(tokens(1))
      case "xori" => (XORI << 8) + toInt(tokens(1))
      case "shr" => (SHR << 8)
      // ...
      case "" => // println("Empty line")
      case t: String => throw new Exception("Assembler
          error: unknown instruction: " + t)
      case _ => throw new Exception("Assembler error")
    }
```

Listing 14.4: The main part of the Leros assembler.

```
class InstrMem(memAddrWidth: Int, prog: String) extends
    Module {
  val io = IO(new Bundle {
    val addr = Input(UInt(memAddrWidth.W))
    val instr = Output(UInt(16.W))
  })
  val code = Assembler.getProgram(prog)
  assert(scala.math.pow(2, memAddrWidth) >= code.length,
      "Program too large")
  val progMem =
      VecInit(code.toIndexedSeq.map(_.asUInt(16.W)))
  val memReg = RegInit(0.U(memAddrWidth.W))
  memReg := io.addr
  io.instr := progMem(memReg)
}
```

Listing 14.5: The instruction memory of Leros.

```
  object State extends ChiselEnum {
    val fetch, execute = Value
  }
  import State._

  val stateReg = RegInit(fetch)

  switch(stateReg) {
    is(fetch) {
      stateReg := execute
    }
    is(execute) {
      stateReg := fetch
    }
  }
```

Listing 14.6: The Leros state machine.

start a read operation from the data memory in the `fetch` state, as the data memory is synchronous and needs one clock cycle to deliver the read result.

In the execute state, we compute a new value for the accumulator or store the read result from the data memory into the accumulator. We also perform a write in the `execute` state.

The following code shows the instantiation of the ALU, including the accumulator and the two main state registers: the program counter (`pcReg`) and the address register (`addrReg`).

```
val alu = Module(new AluAccu(size))
val accu = alu.io.accu

// The main architectural state
val pcReg = RegInit(0.U(memAddrWidth.W))
val addrReg = RegInit(0.U(memAddrWidth.W))

val pcNext = WireDefault(pcReg + 1.U)
```

The following code shows the instantiation of the instruction memory. Note that the instruction memory has as a parameter the file name of the program.

```
val mem = Module(new InstrMem(memAddrWidth, prog))
mem.io.addr := pcNext
val instr = mem.io.instr
```

The following code shows the instantiation of the decode module. The input of the decode module is the instruction from the fetch module, and the outputs are the decode signals. As we need those signals in the `execute` state, they are registered in `decReg`.

```
val dec = Module(new Decode())
dec.io.din := instr
val decout = dec.io.dout

val decReg = RegInit(DecodeOut.default)
when (stateReg === fetch) {
  decReg := decout
}
```

Listing 14.7 shows the data memory of Leros. The memory is organized in 32-bit words. To enable byte access to those 32-bit words, the word is split into a

```
class DataMem(memAddrWidth: Int) extends Module {
  val io = IO(new Bundle {
    val rdAddr = Input(UInt(memAddrWidth.W))
    val rdData = Output(UInt(32.W))
    val wrAddr = Input(UInt(memAddrWidth.W))
    val wrData = Input(UInt(32.W))
    val wr = Input(Bool())
    val wrMask = Input(UInt(4.W))
  })

  val mem = SyncReadMem(1 << memAddrWidth, Vec(4, UInt(8.W)))

  val rdVec = mem.read(io.rdAddr)
  io.rdData := rdVec(3) ## rdVec(2) ## rdVec(1) ## rdVec(0)
  val wrVec = Wire(Vec(4, UInt(8.W)))
  val wrMask = Wire(Vec(4, Bool()))
  for (i <- 0 until 4) {
    wrVec(i) := io.wrData(i * 8 + 7, i * 8)
    wrMask(i) := io.wrMask(i)
  }
  when (io.wr) {
    mem.write(io.wrAddr, wrVec, wrMask)
  }
}
```

Listing 14.7: The data memory module of Leros.

`Vec` of four 8-bit bytes. A `read()` operation returns a vector of four bytes that we concatenate to a 32-bit word with the `##` operator. For the write, we split the written word into those four bytes and use the write mask (`wrMask`) to select which bytes are written. The `SyncReadMem` components contain a `write()` function that takes a vector and a write mask as parameters.

The following code shows the instantiation of the data memory and the connections of the ports.

```
  val dataMem = Module(new DataMem((memAddrWidth)))

  val memAddr = Mux(decout.isDataAccess, effAddrWord,
```

```
    instr(7, 0))
val memAddrReg = RegNext(memAddr)
val effAddrOffReg = RegNext(effAddrOff)
dataMem.io.rdAddr := memAddr
val dataRead = dataMem.io.rdData
dataMem.io.wrAddr := memAddrReg
dataMem.io.wrData := accu
dataMem.io.wr := false.B
dataMem.io.wrMask := "b1111".U
```

14.8 Implementation Variations

Actual processors perform instruction pipelining. In an instruction pipeline, more than one instruction is on the fly. For example, with Leros, we could implement three pipeline stages: instruction fetch, instruction decode, and execute. In that case, three instructions would be in the pipeline, and we can execute one instruction each clock cycle. Compared to the presented implementation, pipelining could about double the performance of Leros. In the next chapter, we will present a pipelined processor implementing the RISC-V instruction set.

14.9 Exercise

This exercise assignment in one of the last chapters is in a very free form. You are at the end of your learning tour through Chisel and ready to tackle design problems that you find interesting.

One option is to reread the chapter and read along with all the source code in the Leros repository, run the test cases, fiddle with the code by breaking it, and see that tests fail.

Another option is to write your implementation of Leros. The implementation in the repository is just one possible organization. You could write a Chisel simulation version of Leros with a single pipeline stage or go crazy and superpipeline Leros for the highest possible clocking frequency.

A third option is to design your processor from scratch. Maybe the demonstration of how to build the Leros processor and the needed tools have convinced you that processor design and implementation are no magic art, but the engineering can be very joyful.

15 A RISC-V Pipeline

Pipelining is a technique to improve throughput of some processing. For example, in an assembly line of a car factory, each stage performs a specialized task, such as engine installation. When that task is done, the car moves to the next stage. This pipelining enables the execution of different tasks in parallel at different stages of a car manufacturing line.

We can also use pipelining in digital design to speed up data stream processing. A prominent example of pipelining in digital design is building a pipelined microprocessor. In that case, instructions flow through the pipeline, performing one operation at each stage by a different processor unit. In contrast to the processor presented in Chapter 14, where each instruction took several clock cycles, in a pipelined processor, instructions run in parallel in different stages. The resulting (ideal) throughput is one instruction per clock cycle.

In this chapter, we present Wildcat, a simple pipelined RISC-V processor [20]. RISC-V is an open instruction set architecture originally developed at the University of California, Berkeley. Wildcat focuses on providing readable Chisel code that can be used in education. This chapter contains the most important source snippets for building a RISC-V pipeline. More details and tests are available in the Wildcat GitHub repository. The repository also contains a RISC-V instruction set simulator, written in Scala, and a single-cycle version in Chisel for demonstration.

15.1 The RISC-V Instruction Set Architecture

Andrew Waterman defined the RISC-V instruction set architecture (ISA) in his PhD thesis [30] at the University of California, Berkeley. Krste Asanovic and Dave Patterson supervised him. Waterman explored RISC architectures from the last three decades, including MIPS, SPARC, and Alpha, to distill the essence of RISC into the RISC-V ISA definition. The RISC-V ISA is available as an open source. Therefore, many microcontrollers providers have switched to RISC-V in the last few years.

RISC-V is an ISA definition; it does not define an implementation. The "V" stands for the fifth RISC project at the University of California, Berkeley, and also

indicates that vector instructions are a part of the standard.

The RISC-V ISA definition defines two base versions, RV32I and RV64I, for the integer instruction set for 32-bit and 64-bit architectures. Optional extensions, such as M, extend ISA for multiply and divide, F and D for floating point, and many more extend the base ISA. In this section, we show the implementation of base RV32I.

The RV32I ISA defines a processor containing 32 registers of 32-bit size (also called the register file), where register X0 is always zero. Furthermore, a program counter (PC) that points to the instruction executed as part of the state of the processor. An instruction is 32-bit wide. Instructions and data reside in a byte-addressable memory. The processor state is 31 registers and the PC.

A RISC architecture is also called a load-store architecture, as all operands first need to be loaded from memory and, after an operation (e.g., an addition) need to be stored back to memory. The base instruction set consists of the following types of operations:

- Arithmetic and logic operations between registers, where the result is written into a register. As an example: `add x1, x2, x3` adds the content of registers `x2` and `x3` and puts the result into register `x1`.
- Arithmetic and logic operations between a register and an immediate value (constant), where the result is written into a register. As an example: `add x1, x2, 42` adds `42` to the content of register `x2` and puts the result into register `x1`.
- A load instruction loads a value from memory into a register. The instruction uses the content of a register as the base address and adds an offset to compute the effective address. An example of a load instruction is: `lw x3, 4(x1)`, where the base address is in `x1` and the offset is 4 bytes. The result is placed into register `x3`. Load instructions for byte, half-word, and word sizes are available as unsigned and signed extension versions.
- Store instructions store a value from a register into memory. The address computation is the same as for load instructions. An example is: `sw x2, 4(x1)` , where the content of register `x2` is stored into memory at address x1 + 4. Store instructions are available for byte, half-word, and word sizes.
- Control instructions can change the program flow by changing the PC. Conditional branches evaluate a condition (comparison between registers) and branch if the condition is met. The branch offset is relative to the current PC.

An example branch instruction is: `bne x2, x3, fail`. The jump instruction changes the PC unconditionally. A variant of the jump instruction stores the next instruction address into a register to enable the return from a function call. An example of such an instruction is: `jal x1, foo`, where the processor jumps unconditionally to function `foo` and stores the return address in register `x1`.

For a detailed description of the RISC-V ISA, read the classic textbook from Patterson and Hennessy [14] or consult the official RISC-V Instruction Set Manual.

15.2 Pipeline Stage Definition

A pipeline stage performs a specific function within one clock cycle. That function is a combinational function. Registers are used between pipeline stages to store intermediate results.

As registers are between those stages, we should define which register belongs to that stage: Either the input register or the output register. For practical reasons, with available on-chip memories, we will consider the input register as part of a pipeline stage.

15.3 Number of Pipeline Stages

The architecture defines the number of pipeline stages. Longer pipelines may allow for clocking of the design at a higher frequency. However, with each additional pipeline stage, the overhead due to the registers increases, and the design becomes more complex.

The classic organization of a reduced instruction set computer (RISC) is as a 5-stage pipeline:

1. Instruction fetch
2. Instruction decode and register file read
3. Execute
4. Memory access
5. Write-back

This organization is used in many computer architecture textbooks, e.g., [14]. However, RISC-V pipeline designs exist from two up to several more stages.

One aspect that determines the number of pipeline stages is the available on-chip memory for instruction and data scratchpad memory or instruction and data caches. Current on-chip memories have registers at their inputs (address and write data); see Section 6.4. That input register is part of the pipeline; it is the pipeline register. A memory read has one clock cycle latency. To simplify the design, we present a three stages RISC-V pipeline:

1. Instruction fetch
2. Instruction decode, register file read, and address computation
3. Execute and memory access

15.4 The Wildcat Pipeline

Figure 15.1 shows our 3-stages Wildcat pipeline. For simplicity, we assume on-chip memories are used for the instruction memory (IM), the register file (RF), and the data memory (DM). Those three on-chip memories set the lower bound on the number of stages to three.[1]

Instructions flow from the left to the right. The result of the instruction is written back from right to left, either in the register file (RF) or the data memory (DM).

15.4.1 Top Level

Listing 15.1 shows the abstract superclass for the top-level module for wildcat implementations.[2] The interface consists only of connections to instruction memory and data memory. We are flexible about which memories we can use, e.g., scratchpad memories for small implementations or caches. IO devices are multiplexed with the data memory. The definitions of the memories and IO shall be made at the top level of a system-on-chip (SoC).

[1] If we build the RF out of concrete flip-flops, we can reduce the number of stages to two, as we can read asynchronously from such an RF.

[2] We share this interface with several implementation variants, e.g., different pipeline organizations.

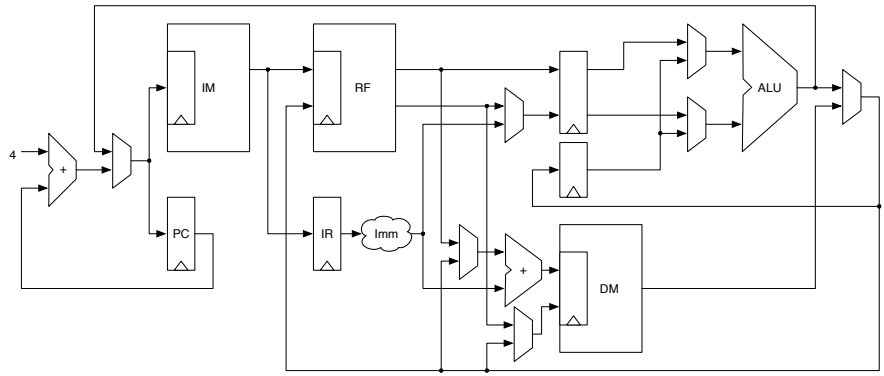


Figure 15.1: The 3-stage Wildcat processor pipeline (simplified, omitting control and decoded signals).

```
abstract class Wildcat() extends Module {
  val io = IO(new Bundle {
    val imem = new InstrIO()
    val dmem = new MemIO()
  })
}
```

Listing 15.1: Top level module of Wildcat.

```
// PC generation
// val pcReg = RegInit(-4.S(32.W).asUInt)
val pcReg = RegInit(0.S(32.W).asUInt)
val pcNext = WireDefault(Mux(doBranch, branchTarget, pcReg
    + 4.U))
pcReg := pcNext
io.imem.address := pcNext

// Fetch
val instr = WireDefault(io.imem.data)
when (io.imem.stall) {
  instr := 0x00000013.U
  pcNext := pcReg
}
```

Listing 15.2: PC generation and instruction fetch.

15.4.2 Instruction Fetch

The program counter (PC) points to the next instruction that shall be executed. RISC-V has 32-bit wide instructions, so the PC is incremented by 4 for each sequential instruction. The PC is set accordingly for a branch instruction, shown by the multiplexer before the adder.

As we can see in the figure, IM contains an input register, which is part of the fetch stage's pipeline register. However, the PC is also part of the pipeline register. Therefore, we cannot feed the output of the PC register to the IM, but the *next* value of the PC. The IM and the PC's address input always contain the same data.

Listing 15.2 shows the code of the fetch stage. The PC (`pcReg`) is initialized to `-4` so that the value of `pcNext` is 0 after reset. The signal `doBranch` selects between a branch target of the current PC plus 4. The `pcNext` signal is connected to the input of the instruction memory. The output of the instruction memory is connected to `instr`. In case that the pipeline needs to be stalled (e.g., for a cache miss) a NOP substitutes the instruction, and `pcNext` will not be incremented.

For simulation and small experiments in an FPGA, we use a read-only memory (ROM) preloaded at hardware generation time. Listing 15.3 shows that memory. It uses a `Vec` that is initialized with the content of a Scala `Array`. The ROM contains a register for the address, which is part of the pipeline register.

```
class InstructionROM(code: Array[Int]) extends Module {
  val io = IO(Flipped(new InstrIO()))

  val addrReg = RegInit(0.U(32.W))
  addrReg := io.address
  val instructions =
      VecInit(code.toIndexedSeq.map(_.S(32.W).asUInt))
  io.data := instructions(addrReg(31, 2))
  io.stall := false.B
}
```

Listing 15.3: A ROM as a simple instruction memory.

```
  val instrReg = RegInit(0x00000033.U) // nop on reset
  instrReg := Mux(doBranch, 0x00000033.U, instr)
  val rs1 = instr(19, 15)
  val rs2 = instr(24, 20)
  val rd = instr(11, 7)
  val (rs1Val, rs2Val, debugRegs) = registerFile(rs1, rs2,
      wbDest, wbData, wrEna, true)

  val decOut = decode(instrReg)
```

Listing 15.4: Decode stage.

15.4.3 Instruction Decode and Register File Read

The second pipeline stage performs the decoding of instructions, register file read, and address calculation for a memory operation. The pipeline register consists of the instruction register and the address register for the register file. In the RISC-V ISA encoding, the fields containing register addresses are always in the same place in the instruction. Therefore, we can read the register file, even when the instruction is not yet decoded. If those values are not needed, they are just ignored.

Listing 15.4 shows the pipeline register for the decode stage (`instrReg`) and the signals `rs1` and `rs2` that are the inputs for the register file. The register file itself and the decode hardware are implemented as a function, a more lightweight form, then a module.

```
val regs = SyncReadMem(32, UInt(32.W),
    SyncReadMem.WriteFirst)
val rs1Val = Mux(RegNext(rs1) === 0.U, 0.U,
    regs.read(rs1))
val rs2Val = Mux(RegNext(rs2) === 0.U, 0.U,
    regs.read(rs2))
when(wrEna && rd =/= 0.U) {
  regs.write(rd, wrData)
}
(rs1Val, rs2Val, debugRegs)
```

Listing 15.5: The core of the register file function.

The core of the register file implementation is shown in Listing 15.5. If possible, we use `SyncReadMem` to implement the register file in an on-chip memory.[3] Note that we use the parameter `SyncReadMem.WriteFirst` to enable forwarding when the same register is written and read in the same clock cycle instead of doing it explicitly, as shown in Section 6.4.

In an FPGA, on-chip memories are initialized to zero during configuration time. However, in an ASIC, the content of the memory is undefined. Therefore, we need to handle the special case when reading from register 0 to return a zero and not the random content of the on-chip memory. Note that the `read()` function returns the result with one clock cycle latency. Therefore, the multiplexer at the memory output needs to be compared against the register address delayed by one clock cycle. (This is similar to the manual forwarding on a read during write, as shown in Section 6.4.) The function returns a Scala tuple returning the two read register values.[4]

The register file has two read ports, which is uncommon in an FPGA. Therefore, the practical implementation will use two on-chip memories written with the same data, each providing one read port.

Listing 15.6 shows part of the decode function. To decode a RISC-V instruction, we need to read the `opcode` field of the instruction and the `func3` field. `decOut` is the output of the decoding. As an initial step, we assign default values to all fields. That is followed by a large `switch` statement comparing the `opcode` against the constants representing the instruction type. The constants are defined as Scala integers so

[3] If synthesizing the memory for an ASIC, e.g., with the open-source OpenLane flow, without using a memory compiler, the memory will be implemented as dedicated flip-flops.

[4] It also returns debugging registers, which are ignored at synthesis.

that we can share them between the hardware implementation of Wildcat and an instruction set simulator written in Scala. Therefore, the constants are converted to Chisel constants with the `.U` method.

A similar function decodes the ALU operation from the instruction. Furthermore, a possible immediate value is extracted from the instruction. All those values are part of the decoded output.

Listing 15.7 shows the address computation for memory load or store instructions. The address (`memAdress`) is computed by using the value of register `rs1` and adding the immediate field (`dec.Out.imm`). However, the code is slightly more complicated, as we need to implement forwarding. Forwarding is needed if we use a register for the address that was just written in the preceding instructions. In that case, the value is not yet written back into the register file, and we *forward* from the execution stage. The code snippet also shows where the write data comes from for a memory store operation. The value of register `rs2` is used, or a value is forwarded from the execution stage.

Listing 15.8 shows the memory access part in decode. As the input registers of the memory are part of the execution stage, the memory address and the store operation are computed in the decode stage.

15.4.4 Execute and Memory Read

The third pipeline stage is responsible for either executing either an ALU operation, execute a control instruction (branch or jump), or perform a memory load. Listing 15.9 shows part of the execution stage. Depending on the instruction type, either the value of a register `v2` or the immediate value `decOut.imm` is used as the second operand for an ALU instruction. The ALU is defined in a Scala function, similar to the decode function. The code snippet also shows the implementation of loading the upper immediate and adding the upper immediate to the PC.

We use a Scala `Enumeration` to define as ALU operations, as shown in Listing 15.10. As with the other constants, they are shared between the ISA simulator and the pipeline implementation. Therefore, they need to be converted to Chisel constants.

Listing 15.11 shows the function that implements the ALU operations.

Listing 15.12 shows the logic for branch execution. The branch target is computed by adding the immediate value of the branch instruction to the PC of that instruction or using the register value for an indirect branch. Similar to the ALU, we use a function (`compare()`) for the compare operation that is needed to implement conditional branches.

```
  def decode(instruction: UInt) = {

    val opcode = instruction(6, 0)
    val func3 = instruction(14, 12)
    val decOut = Wire(new DecodedInstr())
    decOut.instrType := R.id.U
    decOut.isImm := false.B
    decOut.isLui := false.B
    decOut.isAuiPc := false.B
    decOut.isLoad := false.B
    decOut.isStore := false.B
    decOut.isBranch := false.B
    decOut.isJal := false.B
    decOut.isJalr := false.B
    decOut.rfWrite := false.B
    decOut.isECall := false.B
    decOut.isCssrw := false.B
    decOut.rs1Valid := false.B
    decOut.rs2Valid := false.B
    switch(opcode) {
      is(AluImm.U) {
        decOut.instrType := I.id.U
        decOut.isImm := true.B
        decOut.rfWrite := true.B
        decOut.rs1Valid := true.B
      }
      is(Alu.U) {
        decOut.instrType := R.id.U
        decOut.rfWrite := true.B
        decOut.rs1Valid := true.B
        decOut.rs2Valid := true.B
      }
      // and more cases
```

Listing 15.6: The core of the register file function.

```
// Forwarding to memory
val address = Mux(wrEna && (wbDest =/= 0.U) && wbDest ===
    decEx.rs1, wbData, rs1Val)
val data = Mux(wrEna && (wbDest =/= 0.U) && wbDest ===
    decEx.rs2, wbData, rs2Val)

val memAddress = (address.asSInt + decOut.imm).asUInt
decEx.memLow := memAddress(1, 0)
```

Listing 15.7: Address computation, including forwarding if needed.

```
io.dmem.rdAddress := memAddress
io.dmem.rdEnable := false.B
io.dmem.wrAddress := memAddress
io.dmem.wrData := data
io.dmem.wrEnable := VecInit(Seq.fill(4)(false.B))
when(decOut.isLoad && !doBranch) {
  io.dmem.rdEnable := true.B
}
when(decOut.isStore && !doBranch) {
  val (wrd, wre) = getWriteData(data, decEx.func3,
      memAddress(1, 0))
  io.dmem.wrData := wrd
  io.dmem.wrEnable := wre
}
```

Listing 15.8: Memory access in decode.

```
  val res = Wire(UInt(32.W))
  val val2 = Mux(decExReg.decOut.isImm,
      decExReg.decOut.imm.asUInt, v2)
  res := alu(decExReg.decOut.aluOp, v1, val2)
  when(decExReg.decOut.isLui) {
    res := decExReg.decOut.imm.asUInt
  }
  when(decExReg.decOut.isAuiPc) {
    res := (decExReg.pc.asSInt + decExReg.decOut.imm).asUInt
  }
```

Listing 15.9: ALU operation in the execution stage.

```
object AluType extends Enumeration {
  type AluType = Value
  val ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND =
      Value
}
```

Listing 15.10: The ALU operation enumeration.

```
def alu(op: UInt, a: UInt, b: UInt): UInt = {
  val res = Wire(UInt(32.W))
  res := DontCare
  switch(op) {
    is(ADD.id.U) {
      res := a + b
    }
    is(SUB.id.U) {
      res := a - b
    }
    is(AND.id.U) {
      res := a & b
    }
    is(OR.id.U) {
      res := a | b
    }
    is(XOR.id.U) {
      res := a ^ b
    }
    is(SLL.id.U) {
      res := a << b(4, 0)
    }
    is(SRL.id.U) {
      res := a >> b(4, 0)
    }
    is(SRA.id.U) {
      res := (a.asSInt >> b(4, 0)).asUInt
    }
    is(SLT.id.U) {
      res := (a.asSInt < b.asSInt).asUInt
    }
    is(SLTU.id.U) {
      res := (a < b).asUInt
    }
  }
  res
}
```

Listing 15.11: The ALU function.

```
branchTarget := (decExReg.pc.asSInt +
    decExReg.decOut.imm).asUInt
when(decExReg.decOut.isJalr) {
  branchTarget := res
}
doBranch := ((compare(decExReg.func3, v1, v2) &&
    decExReg.decOut.isBranch) || decExReg.decOut.isJal ||
    decExReg.decOut.isJalr) && decExReg.valid
```

Listing 15.12: Branch execution

```
when(decExReg.decOut.isLoad && !doBranch) {
  res := selectLoadData(io.dmem.rdData, decExReg.func3,
      decExReg.memLow)
}
```

Listing 15.13: Memory load.

The memory read operation is performed in the execution stage. Listing 15.13 shows the multiplexer for the load instruction. The function `selectLoadData` selects, depending on the instruction and the lower 2 bits of the address, which part of the read data is used.

15.5 Summary and Exercise

This chapter showed the usage of pipelining in digital design. We used the design of a RISC processor to demonstrate pipelining. The most important parts of the Wildcat RISC-V processor are shown in this chapter. See the complete source code, including test cases, at the Wildcat GitHub repository.

As an exercise, you can build on top of those code fragments shown and implement your own version of a RISC-V processor.

16 Contributing to Chisel

Chisel is an open-source project that is constantly being developed and improved. Therefore, you can also contribute to the project. Your contribution can be twofold: (1) publish your Chisel circuits in open-source and as a library, or (2) contribute enhancements to Chisel itself. Here we describe first how to publish a library and second how to set up your environment for Chisel library development and how to contribute to Chisel.

16.1 Publishing a Chisel Library

When you develop an open-source circuit and share it, for example, on GitHub, this is a very educational act, as others can learn to describe hardware in Chisel from your code example. However, sharing just the source code forces others to copy your code into their project. This leads to at least two problems: (1) *Two copies are never the same*.[1] This means that changes will happen to the source of one copy, and they are not in sync anymore. (2) It is cumbersome to update the copy when the original design has been improved with a bug fix or a new feature.

A better approach is to publish that open-source circuit as a library. Compiled Chisel code is simply a Java class file. And those class files are platform-independent. Therefore, this is an ideal way to share Chisel libraries. Java (and Scala) have a long tradition and good infrastructure to support the public sharing of libraries with unique group identifiers and version control. That is also how Chisel itself and some support libraries are published.

This section describes the steps needed to publish a Chisel library. As the tools you use for publishing may change quickly, consider finding the latest information on the Internet. A good blog entry on the topic can be found here.

[1] I learned this phrase from Doug Locke during discussion sessions developing the safety-critical specification for Java

16.1.1 Using a Library

A Chisel library can be used in your project by adding it to `build.sbt`. Here is an example of a collection of Chisel circuits in ip-contributions:

```
libraryDependencies += "edu.berkeley.cs" % "ip-contributions" % "0.5.0"
```

The `ip-contributions` library also contains the UART and the FIFOs, described in this book. Modern IDEs let you automatically download the library's source code for inspection when configured in `build.sbt`.

If you have a Chisel circuit that you would like to share, consider contributing it to `ip-contributions`. Contribution starts with a git pull request of your addition. This will start a friendly review process.

16.1.2 Prerequisite

Maven Central is one of the largest repositories for hosting software libraries. Publishing to Maven Central is easiest via Sonatype. Sonatype offers free hosting of open-source projects via the Sonatype Repository. The following initial steps are needed before publishing a library:

1. Create a Sonatype JIRA account

2. You need a unique `groupId`, which is usually a domain name in reverse order, e.g., `edu.berkeley.cs`. You can also use your GitHub account as a groupId, for example, mine is `io.github.schoeberl`. You register this groupId by opening an issue. This is a manual process where you get a request to prove that you *own* the requested groupId. When using the GitHub domain name, you are requested to set up a repository to show your ownership.

3. Create `sonatype.sbt` in `$HOME/.sbt/1.0` with your Sonatype login information:

   ```
   credentials += Credentials(
       "Sonatype Nexus Repository Manager",
       "oss.sonatype.org",
       "<user name>",
       "<password>"
   )
   ```

4. All artefacts must be signed with a PGP key pair. You can use the open-source GNU Privacy Guard. You can create, list, and upload your public PGP key with:

```
gpg --gen-key
gpg --list-keys
gpg --keyserver keyserver.ubuntu.com --send-keys keyID
```

Note that the `keyID` is the long hexadecimal string you find in your list of keys. If that step fails, one can manually upload the public key to a key server.

16.1.3 Library Setup

For each library, you need to set up the following:

1. Install sbt plugins in `project/plugins.sbt`:

```
addSbtPlugin("org.xerial.sbt" % "sbt-sonatype" % "2.3")
addSbtPlugin("com.jsuereth" % "sbt-pgp" % "2.0.2")
```

2. Add information about the library into `build.sbt`. As an example, the relevant section of build.sbt in `ip-contributions`:

```
name := "ip-contributions"

version := "0.4.0"

// groupId, SCM, license information
organization := "edu.berkeley.cs"
homepage := Some(url("https://github.com/freechipsproject/ip-contributions"))
scmInfo := Some(ScmInfo(url(
  "https://github.com/freechipsproject/ip-contributions"),
  "git@github.com/freechipsproject/ip-contributions"))
developers := List(Developer("schoeberl", "schoeberl",
  "martin@jopdesign.com", url("https://github.com/schoeberl")))
licenses += ("Unlicense", url("https://unlicense.org/"))
publishMavenStyle := true

// disable publish with Scala version
crossPaths := false
```

```
publishTo := Some(
  if (isSnapshot.value)
    Opts.resolver.sonatypeSnapshots
  else
    Opts.resolver.sonatypeStaging
)
```

16.1.4 Regular Publishing

All is now set up to sign and publish the library to Sonatype with:

```
sbt publishSigned
```

In my setup, the signing from `sbt` does not work (sometimes), so I have to copy out the `pgp` command to sign, something similar to:

```
gpg --detach-sign --armor --use-agent --output path-to.asc\\
path-to-0.1.pom
```

and repeat `sbt publishSigned`.

You can already use that library from Sonatype, for example, for an internal project. However, to release your library to Maven Central run:

```
sbt sonatypeRelease
```

A few minutes later, it should be visible on Maven Central Repository Search.

16.2 Contributing to Chisel

The following is an advanced topic, and I propose that you first start following the discussions of the Chisel project issues on GitHub before you create your first pull request (PR).

16.2.1 Setup the Development Environment

Chisel consists of several different repositories: the main repositories are hosted on CHIPS Alliance and others on the freechips organization at GitHub.

Fork the repository to which you would like to contribute into your personal GitHub account. You can fork the repository by pressing the `Fork` button in the

GitHub web interface. Then, from that fork, clone your fork of the repository.[2] In our example, we change `chisel3`, and the clone command for my local fork is:

```
$ git clone git@github.com:schoeberl/chisel3.git
```

To compile Chisel 3 and publish it as a local library, execute:

```
$ cd chisel3
$ sbt compile
$ sbt publishLocal
```

Watch out during the publish local command for the version string of the published library, which contains the string `SNAPSHOT`. If you use the tester and the published version is not compatible with the Chisel SNAPSHOT, fork and clone the chisel-tester repo as well and publish it locally.

To test your changes in Chisel, you probably also want to set up a Chisel project, for example, by forking/cloning an empty Chisel project, renaming it, and removing the `.git` folder from it.

Change the `build.sbt` to reference the locally published version of Chisel. Compile your Chisel test application and take a close look to ensure that it picks up the locally published version of the Chisel library (there is also a SNAPSHOT version published, so if, for example, the Scala version is different between your Chisel library and your application code, it picks up the SNAPSHOT version from the server instead of your local published library.)

See also some notes at the Chisel repo.

16.2.2 Testing

When you change the Chisel library, you should run the Chisel tests. In an `sbt`-based project, they are usually run with:

```
$ sbt test
```

Furthermore, if you add functionality to Chisel, you should also provide tests for the new features.

[2] Note that on a breaking FIRRTL/Chisel change, you might need also to fork and clone `firrtl`.

16.2.3 Contribute with a Pull Request

In the Chisel project, no developer commits directly to the main repository. A contribution is organized via a pull request from a branch in a forked version of the library. For further information, see the documentation on GitHub on collaboration with pull requests. The Chisel group started to document contribution guidelines.

16.3 Exercise

Invent a new operator for the `UInt` type, implement it in the Chisel library, and write some usage/test code to explore the operator. It does not need to be a useful operator; just anything will be good, for example, a ? operator that delivers the lefthand side if it is different from 0 and the righthand side otherwise. Sounds like a multiplexer, right? How many lines of code did you need to add?[3]

As simple as this was, please do not be tempted to fork the Chisel project and add your little extensions. Changes and extensions shall be coordinated with the main developers. This exercise was just a simple exercise to get you started.

If you are getting bold, you could pick one of the open issues and try to solve it. Then, contribute with a pull request to Chisel. However, probably first watch the style of development in Chisel by watching the GitHub repositories. See how changes and pull requests are handled in the Chisel open-source project.

[3] A quick and dirty implementation needs just two lines of Scala code.

17 Summary

This book presented an introduction to digital design using the hardware construction language Chisel. We have seen several simple to medium-sized digital circuits described in Chisel. Chisel is embedded in Scala and, therefore inherits the powerful abstraction of Scala. As this book is intended as an introduction, we have restricted our examples of simple uses of Scala. The next logical step is to learn a few basics of Scala and apply them to your Chisel project.

I would be happy to receive feedback on the book, as I will further improve it and publish new editions. You can contact me at mailto:masca@dtu.dk or with an issue request on the GitHub repository. I also happily accept pull requests for the book repository for fixes and improvements.

Source Access

This book is available in open source, including all Chisel code presented. The repository also contains slides for a digital design course with Chisel and all Chisel examples: https://github.com/schoeberl/chisel-book

A collection of medium-sized examples, which are referenced in the book, is also available in open source. This collection also contains projects for various popular FPGA boards: https://github.com/schoeberl/chisel-examples

A VHDL and Verilog

This book teaches digital design with Chisel. However, you might be confronted with other hardware description languages in your professional career. When you understand the basic concepts of digital design and how to describe digital circuits in Chisel, switching to another language will take just a few days.

The two dominating hardware description languages are VHDL and Verilog, with an update to SystemVerilog (SV). The recent movement in hardware development and testing is from VHDL to SV. However, as many open-source tools still mainly support standard Verilog, we will use plain Verilog in the examples.

We will show you code examples in Chisel, VHDL, and Verilog. Note that this appendix is not a complete introduction to VHDL or Verilog. It will just get you started on reading VHDL and Verilog and enable you to translate hardware constructs from Chisel to VHDL and Verilog. For the translation from Chisel to Verilog, you can also start with the Chisel generated Verilog. It might be less readable than hand-written code, but it is a working starting point. A nice side-by-side introduction of VHDL and SystemVerilog can be found in [10]. Furthermore, we will show how we can integrate legacy code in Verilog into a Chisel design.

A.1 Code Examples

We will present the VHDL and Verilog examples together with the Chisel version of the code.

A.1.1 Components

Listing A.1 shows a simple adder[1] as our example component in Chisel. To make this a complete example, the listing also includes the `import` statements.

[1] We should never use such small components in real designs when the code that defines the function is just a few lines. In this example, just one line. However, we keep it short for the presentation of the concept of a component.

```
import chisel3._
import chisel3.util._

class ChiselAdder extends Module {
  val io = IO(new Bundle() {
    val a, b = Input(UInt(8.W))
    val sum = Output(UInt(8.W))
  })
  io.sum := io.a + io.b
}
```

Listing A.1: A simple component in Chisel.

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

entity adder is
  port (
    a, b : in unsigned(7 downto 0);
    sum : out unsigned(7 downto 0)
  );
end entity;



architecture rtl of adder is
begin
  sum <= a + b;
end architecture;
```

Listing A.2: A simple component in VHDL.

```
module adder(
  input  [7:0] a,
  input  [7:0] b,
  output [7:0] sum
);

  assign sum = a + b;

endmodule
```

Listing A.3: A simple component in Verilog.

Listing A.2 shows the adder component in VHLD. Similar to Chisel, the file starts with import statements to include some standard functions. In VHDL, a component needs a declaration of an `entity` and the definition of an `architecture`. The `entity` declares the ports for the component. The `architecture` defines the function of the component. The `architecture` needs to be named, in our example that name is `rtl`. However, today, that naming has lost its practical meaning. VHDL is case insensitive, that means `Adder` and `adder` are treated equally. However, the current practice is to stick to lower cases for identifiers.

Listing A.3 shows the adder component in Verilog. Similar to Chisel, a component is also called a module.[2] The adder has two 8-bit inputs and one 8-bit output. The `assign` statement is a concurrent statement that continuously assigns the value of the sum of `a` and `b` to the output port `sum`. The difference between Verilog and Chisel is minimal for such a simple component.

A.1.2 Using a Component

Listing A.4 shows how to create a component with `new` in Chisel, give it a name (`m`), and connect the input and output ports to wires.

Listing A.5 shows how to declare and use the `adder` component in VHDL. VHDL is strongly typed, so a component must be declared before it is instantiated. The component declaration is very similar to the entity declaration of the adder itself. The component is instantiated by calling it, prepending it with a name (`m`) and connecting the input and output ports with a `port map` to signals.

[2] In fact, the naming in Chisel was inspired by Verilog.

```
  val in1 = Wire(UInt(8.W))
  val in2 = Wire(UInt(8.W))
  val result = Wire(UInt(8.W))

  val m = Module(new ChiselAdder)
  m.io.a := in1
  m.io.b := in2
  result := m.io.sum
```

Listing A.4: Using the `ChiselAdder` component in Chisel.

```
  signal in1, in2, result : unsigned(7 downto 0);

  component adder
    port (
      a, b : in unsigned(7 downto 0);
      sum : out unsigned(7 downto 0)
    );
  end component;

begin

    m: adder
        port map (
        a => in1,
        b => in2,
        sum => result
        );
```

Listing A.5: Using the `adder` component in VHDL.

```
    wire [7:0] in1;
    wire [7:0] in2;
    wire [7:0] result;

    adder m(.a(in1), .b(in2), .sum(result));
```

Listing A.6: Using the `adder` component in Verilog.

```
  val reg = RegEnable(data, 0.U(8.W), enable)
```

Listing A.7: A register with reset and enable in Chisel.

```
    signal reg : std_logic_vector(7 downto 0);

begin
    process (clock)
    begin
        if rising_edge(clock) then
            if reset = '1' then
                reg <= (others => '0');
            elsif enable = '1' then
                reg <= data;
            end if;
        end if;
    end process;
```

Listing A.8: A register with reset and enable in VHDL.

Listing A.6 shows how to create and use a component (module) in Verilog. Here, it is created by calling it, followed by a name (`m`), and connecting the wires to the ports.

Note `a`, `b`, and `sum` represent the ports of the adder; `in1`, `in2`, and `result` are the wires/signals connected to those ports. The instances of the adders in all three examples were named `m`. In VHDL and Verilog, the name of the component is not used so often; in Chisel, we need the module's name to access the IO ports.

A.1.3 Register

Listing A.7 shows an 8-bit register with reset (to 0) and an enable input in Chisel, using the three-parameter version of `RegEnable`. The inputs are connected at the creation, and the output is available as the returned value of `RegEnable`. Note that the `clock` and `reset` signals are implicit in Chisel.

Listing A.8 shows VHDL code for a register (`reg`) with a synchronous reset and an enable input. Note that the VHDL `process` includes the `clock` signals in the sen-

```
reg [7:0] reg_data;

always @(posedge clk) begin
  if (reset)
    reg_data <= 8'b0;
  else if (enable)
    reg_data <= data;
end
```

Listing A.9: A register with reset and enable in Verilog.

sitivity list. The register is inferred using a clocked process and `rising_edge(clock)`. In contrast to Chisel both, `clock` and `reset`, need to be explicitly used.

Listing A.9 shows the definition of a register (`reg_data`) with a synchronous reset and an enable input in Verilog. The Verilog `reg` keyword declares a variable. This does not necessarily mean that the variable represents a register. It can also represent combinational logic when used in an `always @(*)` block. The `always @(posedge clk)` statement with a clock implies that the code within that block represents a register.

Registers are defined implicitly by a code patterns in Verilog and VHDL. In Chisel the registers are defined explicitly.

A.1.4 Combinational Blocks

Simple combinational logic can be written by an assignment as a concurrent statement in all three languages. We have seen examples in the adder components. However, for more complex logic, e.g., describing nested conditions, it is more convenient to use blocks for combinational logic. In Verilog, this is an `always` block; in VHDL this is a `process`.

Listing A.10 shows as an example of a combinational block a `switch` statement in Chisel. This code is a 4:1 multiplexer. In Chisel the default value for the output needs to be assigned before the `switch` statement.

Listing A.11 shows the same combinational block in VHDL. A combinational block is a `process`. Start is additionally marked with a `begin` and the end of the process with `end process;` The default value for that `case` statement is assigned in with a `when others` entry. Note that a VHDL process has a sensitive list, where all signals that appear on the right hand side of an expression or a a condition need to

```
io.out := 0.U
switch(io.sel) {
  is("b00".U) { io.out := io.in(0) }
  is("b01".U) { io.out := io.in(1) }
  is("b10".U) { io.out := io.in(2) }
  is("b11".U) { io.out := io.in(3) }
}
```

Listing A.10: A `switch` statement in Chisel.

```
process (sel, input)
begin
  case sel is
    when "00" => output <= input(0);
    when "01" => output <= input(1);
    when "10" => output <= input(2);
    when "11" => output <= input(3);
    when others => output <= '0';
  end case;
end process;
```

Listing A.11: A `case` statement in VHDL.

```
module comb(
  input  [1:0] sel,
  input  [3:0] in,
  output reg out
);

  always @(*) begin
    case (sel)
      2'b00: out = in[0];
      2'b01: out = in[1];
      2'b10: out = in[2];
      2'b11: out = in[3];
      default: out = 1'b0;
    endcase
  end

endmodule
```

Listing A.12: A `case` statement in Verilog.

be listed. In our example, this is `sel` and `input`.

Listing A.12 shows the same combinational block in Verilog. Note the marking of the start of a combinational block with `always @(*)begin` and the end of the block with `end`. The default value for that `case` statement is assigned in with a `default` entry. This example includes the module header to show that the output `out` needs to be declared as `reg`, as is is assigned in an `always` block.

Note that Verilog has two different assignment operators: the = is called a *blocking* statement and the <= a *non-blocking* statement. This might also be a source of confusion. Use the <= assignment operator for registers (in an `always @(posedge clk)` block) and the = assignment operator for combinational logic (in an `always @(*)` block).

Another convenient construct in a combinational block is a combination of conditionals, also known as "if...else if...else" statements. We will show an example with three inputs (`in1`, `in2`, and `in3`), two boolean conditions (`c1` and `2`), and the output `out`.

Listing A.13 shows the Chisel version. However, as `if` and `else` are conditional expressions in Scala (reserved keywords), Chisel uses `when`, `.elsewhen`, and

```
when (io.c1) {
  io.out := io.in1
} .elsewhen (io.c2) {
  io.out := io.in2
} .otherwise {
  io.out := io.in3
}
```

Listing A.13: An “if...else if...else” statement in Chisel.

```
process(c1, c2, in1, in2, in3)
begin
  if c1 = '1' then
    output <= in1;
  elsif c2 = '1' then
    output <= in2;
  else
    output <= in3;
  end if;
end process;
```

Listing A.14: An “if...else if...else” statement in VHDL.

`.otherwise`.

Listing A.14 shows the same construct in VHDL. Note that all input signals (including the condition signals) need to be part of the sensitivity list of the process.

Listing A.15 shows the same expression in Verilog within an `always` block.

A.1.5 Advanced Chisel Features

The basic elements look similar in Chisel, Verilog, and VHDL, and the verbosity is also in the same range (although VHDL is a bit more chatty). However, neither VHDL nor Verilog contains object-oriented features for hardware description. SystemVerilog includes object-oriented programming only for writing test benches. Both languages are missing functional programming, which is important when writing reusable hardware generators. Furthermore, `Bundles` with directions and the

```
always @(*) begin
  if (c1)
    out = in1;
  else if (c2)
    out = in2;
  else
    out = in3;
end
```

Listing A.15: An “if...else if...else” statement in Verilog.

possibility to flip the direction are also missing.

Although any hardware can be equally described in Chisel, VHDL, and Verilog, the modern features in Chisel allow the programming of a higher abstractions by writing hardware generators.

A.2 External Modules and Integration of Legacy Code

TODO: describe Verilog and VHDL (with GHDL)

Sometimes, you might wish to include a component whose description is written in Verilog, e.g., legacy code, or you might wish to ensure the emitted Verilog of a component has a concrete structure that your synthesis tool can recognize and map to an available primitive. Chisel provides support for this through its `BlackBox` and `ExtModule` classes, which allow you to define components with Verilog sources. Both are parameterized with a `Map[String, Param]`, translated to module parameters in the emitted Verilog. `BlackBox`es are emitted as individual Verilog files, while `ExtModules` act as placeholders and are emitted as source-less module instantiations. This feature makes `ExtModules` particularly useful for, e.g., Xilinx or Intel device primitives such as clock or input buffers.

```
class BUFGCE extends BlackBox(Map("SIM_DEVICE" ->
    "7SERIES")) {
  val io = IO(new Bundle {
    val I = Input(Clock())
    val CE = Input(Bool())
    val O = Output(Clock())
  })
```

```
}


class alt_inbuf extends ExtModule(
  Map("io_standard" -> "1.0 V",
    "location" -> "IOBANK_1",
    "enable_bus_hold" -> "on",
    "weak_pull_up_resistor" -> "off",
    "termination" -> "parallel 50 ohms")) {
  val io = IO(new Bundle {
    val i = Input(Bool())
    val o = Output(Bool())
  })
}
```

BlackBoxes, on the other hand, can represent any component. They can be declared in three different ways, with their source either inlined or available in a separate file. As an example, consider a 32-bit adder with the following IO.

```
class BlackBoxAdderIO extends Bundle {
  val a = Input(UInt(32.W))
  val b = Input(UInt(32.W))
  val cin = Input(Bool())
  val c = Output(UInt(32.W))
  val cout = Output(Bool())
}
```

The inlined version is declared as follows:

```
class InlineBlackBoxAdder extends HasBlackBoxInline {
  val io = IO(new BlackBoxAdderIO)
  setInline("InlineBlackBoxAdder.v",
    s"""
       |module InlineBlackBoxAdder(a, b, cin, c, cout);
       |input  [31:0] a, b;
       |input  cin;
       |output [31:0] c;
       |output cout;
       |wire   [32:0] sum;
       |
       |assign sum  = a + b + {31'b0, cin};
       |assign c    = sum[31:0];
```

```
        |assign cout = sum[32];
        |
        |endmodule
  """.stripMargin)
}
```

Providing the source code within a string literal (denoted by an `s` or an `f` before the double quotes) and using pipes allows for the inclusion of nicely formatted Verilog code. Additionally, it enables support for parameterization because Scala variables can be inserted using the `$` or `${}` escape characters. The `stripMargin` method removes the pipes and tabs when emitting the code.

There are two alternatives to inlined BlackBoxes, both expecting the Verilog source in a separate file. They are declared as follows.

```
class ResourceBlackBoxAdder extends HasBlackBoxResource {
  val io = IO(new BlackBoxAdderIO)
  addResource("/ResourceBlackBoxAdder.v")
}


class PathBlackBoxAdder extends HasBlackBoxPath {
  val io = IO(new BlackBoxAdderIO)
  addPath("./src/main/resources/PathBlackBoxAdder.v")
}
```

The `HasBlackBoxResource` version expects to find its Verilog source in the `./src/main/resource` folder. The `HasBlackBoxPath` version can be provided with any relative path from the project folder.

BlackBoxes are instantiated like other modules by wrapping them as `Module(new BlackBoxModule)`. They cannot be tested directly but must be wrapped either in a named class or an anonymous class by the tester. Both are allowed to have the same IO as the Blackbox.

```
  class InlineAdder extends Module {
    val io = IO(new BlackBoxAdderIO)
    val adder = Module(new InlineBlackBoxAdder)
    io <> adder.io
  }


  test(new Module {
```

```
    val io = IO(new BlackBoxAdderIO)
    val adder = Module(new InlineBlackBoxAdder)
    io <> adder.io
  })
```

Note that `HasBlackBoxInline`, `HasBlackBoxPath`, and `HasBlackBoxResource` are `traits` that extend Chisel's `BlackBox` class meaning that, e.g., `class Example extends BlackBox with HasBlackBoxInline` is equivalent to `class Example extends HasBlackBoxInline`.

A.3 Exercise

Use one of your Chisel designs and translate it manually to Verilog. To avoid writing a test bench in Verilog, you can reuse your test code from ChiselTest and instantiate the Verilog code as a black box for testing. You can also change your test code to instantiate your Chisel and Verilog components and compare them in the same Chisel tester. The examples in the book are tested in that way. You can take a look at the code from the book examples to see how this can be done. This form of cosimulation can also be driven from random test vectors and comparing the Chisel "golden model" with your Verilog translation.

The VHDL integration within Chisel/Verilator is not yet so smooth, although a solution with GHLD plugin for yosys should be able to translate VHDL to Verilog can then be tested in Chisel similar to the Verilog designs.

For VHDL, there are fewer open-source options available. The code in this chapter is tested with test benches written in VHDL (generated by copilot) and simulated with GHDL.[3]

[3] http://ghdl.free.fr/

B Reserved Keywords

Several keywords are reserved in Chisel (and Scala) and cannot be used as identifiers for your hardware design. Table B.1 lists the reserved words from Scala.

:	<-	<:	<%	=	=>
>:	@	#	_	abstract	case
catch	class	def	do	else	extends
false	final	finally	for	forSome	if
implicit	import	lazy	match	new	null
object	override	package	private	protected	return
sealed	super	this	throw	trait	true
try	type	val	var	while	with
yield					

Table B.1: Reserved keywords from the Scala language.

Table B.2 lists the reserved words added by the Chisel library. In contrast to the Scala reserved word listing, it also contains type/class names defined by Chisel. Although technically possible, you should also avoid using Chisel (and Scala) operators, such as + or <<, for example.

:=	##	andR	Bits	Bool	Cat
clock	elsewhen	io	is	Mem	Module
name	orR	otherwise	Reg	RegEnable	RegInit
RegNext	reset	SInt	switch	SyncMem	UInt
Vec	VecInit	when	Wire	WireDefault	xorR

Table B.2: Reserved keywords from the Chisel language.

C Chisel Projects

Chisel is not (yet) used in many projects. Therefore, open-source Chisel code to learn the language and the coding style is rare. Here, we list several projects we know that use Chisel and are open source.

Rocket Chip is a RISC-V [31] processor-complex generator that comprises the Rocket microarchitecture and TileLink interconnect generators. Originally developed at UC Berkeley as the first chip-scale Chisel project [4], Rocket Chip is now commercially supported by SiFive.

Sodor is a collection of RISC-V implementations intended for educational use. It contains 1, 2, 3, and 5 stages pipeline implementations. All processors use a simple scratchpad memory shared by instruction fetch, data access, and program loading via a debug port. Sodor is mainly intended to be used in simulation.

Patmos is an implementation of a processor optimized for real-time systems [27]. The Patmos repository includes several multicore communication architectures, such as a time-predictable memory arbiter [23], a network-on-chip [26], and a shared scratchpad memory with an ownership [28].

FlexPRET is an implementation of a precision timed architecture [34]. FlexPRET implements the RISC-V instruction set and has been updated to Chisel 3.1.

Lipsi is a tiny processor intended for utility functions on a system-on-chip [19]. As the code base of Lipsi is very small; it can serve as an easy starting point for processor design in Chisel. Lipsi also showcases Chisel/Scala's productivity. It took me 14 hours to describe the hardware in Chisel and run it on an FPGA, write an assembler in Scala, write a Lipsi instruction set simulator in Scala for co-simulation, and write a few test cases in Lipsi assembler.

Leros is a small accumulator based processor processor, originally as a16-bit version written in VHDL [18]. The redesign is now coded in Chisel and a 32-bit

version [25]. Morten Borup Petersen ported the LLVM C compiler to support the Leros ISA [15].

OpenSoC Fabric is an open-source NoC generator written in Chisel [9]. It is intended to provide a system-on-chip for large-scale design exploration. The NoC is a state-of-the-art design with wormhole routing, credits for flow control, and virtual channels. OpenSoC Fabric is still using Chisel 2.

DANA is a neural network accelerator [7] that integrates with the RISC-V Rocket processor using the Rocket Custom Coprocessor (RoCC) interface [8]. DANA supports inference and learning.

Chiselwatt is an implementation of the POWER Open ISA. It includes instructions to run Micropython.

VTA Hardware Design Stack is an accelerator for machine learning for the Apache TVM machine learning compiler framework.

Chisel IP Contributions started to collect small Chisel components. It includes the source of the UART and FIFOs that are presented in this book.

Constellation is a NoC Generator developed at UC Berkeley [33]. Constellation generates packet-switched NoCs with wormhole routing, virtual channels, and credit-based flow control. The interface to the NoC can use AXI-4 or TileLink. Compared to other NoCs and NoC generators, Constellation can generate any topology with application-specific routes.

XiangShan is an open-source, out-of-order RISC-V processor [32]. XiangShan promotes Chisel for agile hardware design.

Wildcat is a simple pipelined RISC-V processor [20]. The focus on Wildcat is to provide readable Chisel code that can be used in education.

If you know an open-source project that uses Chisel, please drop me a note so I can include it in a future edition of the book.

D Acronyms

Hardware designers and computer engineers like to use acronyms. However, it takes time to get used to them. Here is a list of common digital design and computer architecture terms.

ADC analog-to-digital converter

ALU arithmetic and logic unit

ASIC application-specific integrated circuit

CFG control flow graph

Chisel constructing hardware in a Scala embedded language

CISC complex instruction set computer

CPI clock cycles per instruction

CPU central processing unit

CRC cyclic redundancy check

DAC digital-to-analog converter

DFF D flip-flop, data flip-flop

DMA direct memory access

DRAM dynamic random access memory

EMC electromagnetic compatibility

ESD electrostatic discharge

FF flip-flop

FIFO first-in, first-out

FPGA field-programmable gate array

HDL hardware description language

HLS high-level synthesis

IC instruction count

IDE integrated development environment

ILP instruction-level parallelism

IC integrated circuit

IO input/output

ISA instruction set architecture

JDK Java Development Kit

JIT just-in-time

JVM Java virtual machine

LC logic cell

LRU least-recently used

LSB least significant bit

MMIO memory-mapped IO

MSB most significant bit

MUX multiplexer

OO object-oriented

OOO out-of-order

OS operating system

RAM random access memory

RISC reduced instruction set computer

SDRAM synchronous DRAM

SRAM static random access memory

TOS top of stack

UART universal asynchronous receiver/transmitter

VHDL VHSIC hardware description language

VHSIC very high speed integrated circuit

Bibliography

[1] Altera. Avalon interface specification, April 2005.

[2] ARM. AMBA specification (rev 2.0), May 1999.

[3] ARM. AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite. https://developer.arm.com/documentation/ihi0022/e/, 2011.

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference (DAC 2012)*, pages 1216–1225, San Francisco, CA, USA, June 2012. ACM.

[6] William J. Dally, R. Curtis Harting, and Tor M. Aamodt. *Digital design using VHDL: A systems approach*. Cambridge University Press, 2016.

[7] Schuyler Eldridge, Amos Waterland, Margo Seltzer, Jonathan Appavoo, and Ajay Joshi. Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, pages 99–112, 2015.

[8] Schuyler Eldridge, Amos Waterland, Margo Seltzer, and Jonathan Appavooand Ajay Joshi. Towards general-purpose neural network computing. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 99–112, Oct 2015.

[9] Farzaf Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. Opensoc fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–203, April 2016.

[10] S. Harris and D. Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Elsevier Science, 2021.

[11] IBM. On-chip peripheral bus architecture specifications v2.1, April 2001.

[12] Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. Open-source formal verification for chisel. In *Proceedings of the Fourth Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[13] OCP-IP Association. Open core protocol specification 2.1. http://www.ocpip.org/, 2005.

[14] David A. Patterson and John L. Hennessy. *Computer Organization and Design, RISC-V Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2020.

[15] Morten Borup Petersen. A compiler backend and toolchain for the leros architecture. B.sc.eng. thesis, Technical University of Denmark, 2019.

[16] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.3. Available at http://www.opencores.org, September 2002.

[17] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workshop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[18] Martin Schoeberl. Leros: A tiny microcontroller for FPGAs. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL 2011)*, pages 10–14, Chania, Crete, Greece, September 2011. IEEE Computer Society.

[19] Martin Schoeberl. Lipsi: Probably the smallest processor in the world. In *Architecture of Computing Systems – ARCS 2018*, pages 18–30. Springer International Publishing, 2018.

[20] Martin Schoeberl. The educational risc-v microprocessor wildcat. In *Proceedings of the Sixth Workshop on Open-Source EDA Technology (WOSET)*, 2024.

[21] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.

[22] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos reference handbook. Technical report, Technical University of Denmark, 2014.

[23] Martin Schoeberl, David VH Chong, Wolfgang Puffitsch, and Jens Sparsø. A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, pages 53–62, Madrid, Spain, July 2014.

[24] Martin Schoeberl, Hans Jakob Damsgaard, Luca Pezzarossa, Oliver Keszocze, and Erling Rennemo Jellum. Hardware generators with chisel. In *2024 27th Euromicro Conference on Digital System Design (DSD)*, pages 168–175, 2024.

[25] Martin Schoeberl and Morten Borup Petersen. Leros: The return of the accumulator machine. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019 - 32nd International Conference, Proceedings*, pages 115–127. Springer, 1 2019.

[26] Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. A minimal network interface for a simple network-on-chip. In Martin Schoeberl, Thilo Pionteck, Sascha Uhrig, Jürgen Brehm, and Christian Hochberger, editors, *Architecture of Computing Systems - ARCS 2019*, pages 295–307. Springer, 1 2019.

[27] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[28] Martin Schoeberl, Tórur Biskopstø Strøm, Oktay Baris, and Jens Sparsø. Scratchpad memories with ownership. In *2019 Design, Automation and Test in Europe Conference Exhibition (DATE)*, 2019.

[29] Bill Venners, Lex Spoon, and Martin Odersky. *Programming in Scala, 3rd Edition*. Artima Inc, 2016.

[30] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2016.

[31] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume I: Base user-level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley, May 2011.

[32] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022.

[33] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. Constellation: An open-source SoC-capable NoC generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*, pages 1–7, 2022.

[34] Michael Zimmer. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.

Index

ALU, 56, 221
Arbiter, 67, 156
arithmetic operations, 13
Array, 18
Assembler, 229
Assertion, 213
Asynchronous Input, 97
AXI, 200

BCD, 139
Binary-coded decimal, 139
Bit
    concatenation, 14
    extraction, 14
    reduction, 14
Bitfield
    concatenation, 14
    extraction, 14
Blackbox, 268
Bool, 13
BoringUtils, 209
Bubble FIFO, 162
build.sbt, 47
Bulk connection, 59
Bundle, 18

Case classes, 144
Chisel
    Contribution, 251
    Examples, 6, 275
    Versions, 33
ChiselEnum, 110
ChiselTest, 39
Circular buffer, 176
    read pointer, 176
    write pointer, 176
Clock, 73
Collection, 18
Combinational circuit, 61
Communicating state machines, 119
Comparator, 71
Component, 49
Counter, 78
Counting, 18

Data forwarding, 90
Datapath, 125
Debouncing, 98
Debugging, 203
Decoder, 63
DecoupledIO, 172
Double buffer FIFO, 174

Edge detection, 102
elsewhen, 62
emit Verilog, 32
Encoder, 65
    Priority encoder, 70

FIFO, 161, 169
FIFO buffer, 161
File reading, 140
Finite-State Machine
    Mealy, 112
    Moore, 108
Finite-state machine, 107
First-in, first-out buffer, 161
Flip-flop, 73
Flipped, 172
FSM, 107
FSMD, 124
Function components, 137
Functional programming, 152

generate Verilog, 32

Hardware generators, 135

if/elseif/else, 62
Inheritance, 150
Initialization, 74
Integer
    constant, 12
    signed, 11
    unsigned, 11
    width, 11
Interconnect, 187
IO, 25
IO interface, 49

Java
    Versions, 36

Leros, 217
Logic generation, 139
Logic table generation, 139
Logical clock, 81
logical operations, 13

Majority voting, 100
Memory, 89
Memory mapped IO, 194
Metastability, 97
Module, 49
Multiplexer, 16

Object-oriented, 150
Operators, 14
otherwise, 62

Parameters, 143
Pipelining, 237
Ports, 49
printf, 45
Processor, 217
    ALU, 221
    instruction decode, 226

RAM, 89
Ready/valid interface, 130, 172
Reg, 16, 25
Register, 16, 73
    with enable, 76
Register file, 21
Reserved keywords, 273
Reset, 74
RISC-V, 237
ROM, 139

sbt, 29
Scala, 135
    for loop, 136
    Seq, 137
    tuple, 137
    val, 135
    Versions, 36
ScalaTest, 38

Serial port, 164
Source organization, 29
SRAM, 89
State diagram, 108
State machine with datapath, 124
Structure, 18
switch, 64
Synchronous memory, 89
Synchronous sequential circuit, 107

Testing, 37, 203
Tick, 81
Timer, 83
Timing diagram, 74
Timing generation, 80
tuple, 180
Type
    conversion, 142
    parameters, 145

UART, 164

Vcd, 43
VCS, 212
Vec, 18
VecInit, 20, 140
Vector, 18
Verification, 203
Verilator, 212
Verilog, 32, 259
VHDL, 259

Waveform, 43
Waveform diagram, 74
when, 62
Wildcat, 237
Wire, 14, 25
Wishbone, 198