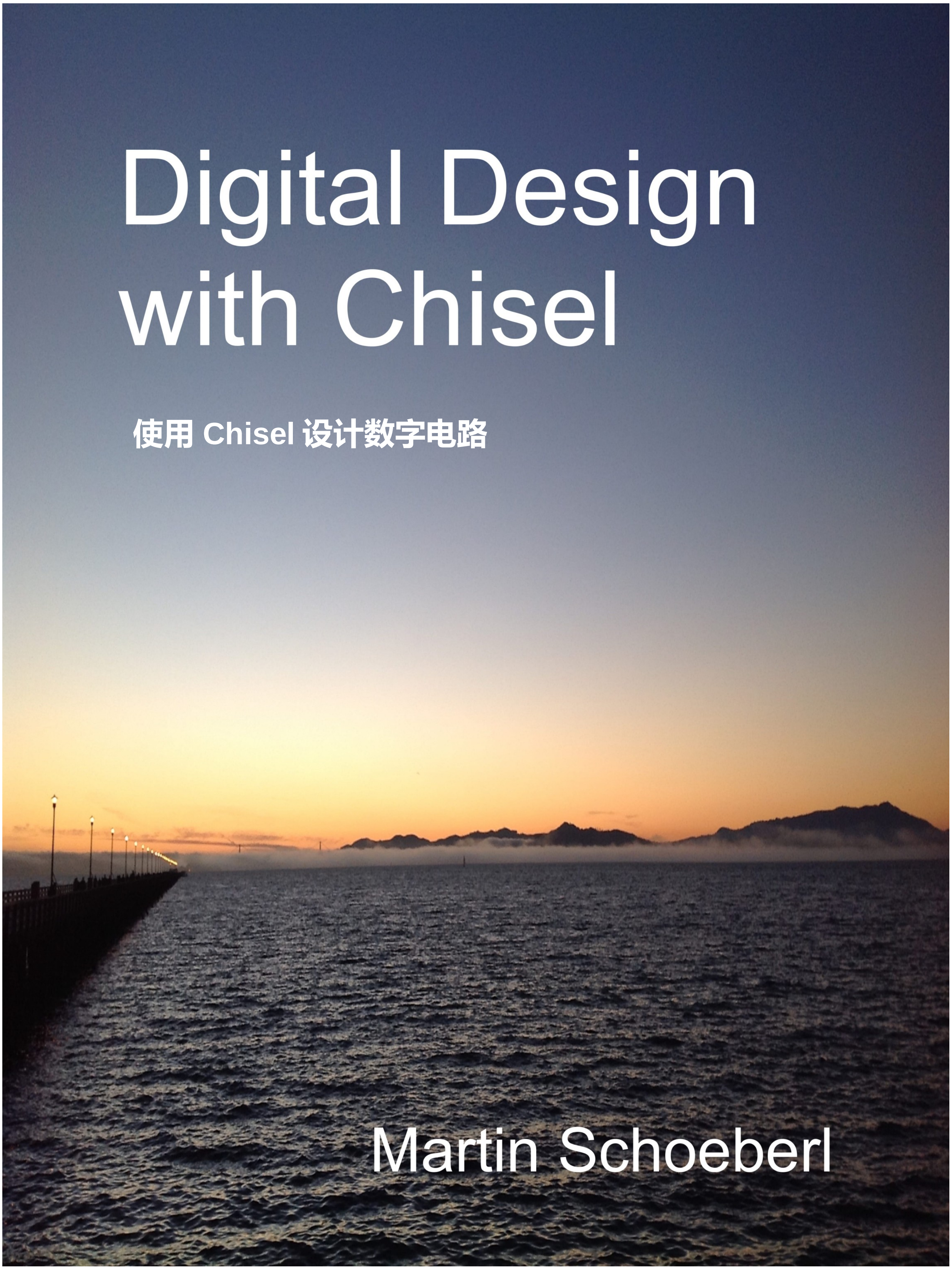

Digital Design with Chisel

使用 Chisel 设计数字电路

Martin Schoeberl

使用Chisel设计数字电路

作者： Martin Schoeberl

translated by

翻译： Yuda Wang, Qiwei Sun
审核： Yun Chen

该讲义将会成为一本书，是一本硬件设计的导论，专注于使用硬件构建语言Chisel。这本书的目的是展示小到中型的硬件部分，用于探索使用Chisel进行硬件设计。我们会随后详细讨论这些问题。

Jun 2020

Contents

Chapter 1

写在前边

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.

非常激动，处于一个数字设计的世界里。随着Dennard缩规和摩尔定律的放缓，在这个领域可能再也没有一个更伟大的创新。半导体公司尽可能地继续压榨每粒性能，但是这些改进带来的成本巨大地上升。chisel通过改进生产力，减少了这个成本。如果设计者能在短时间内搭建更多，同时使用复用摊销验证的成本，公司可以在非重复工程（NRE）消耗得更少。另外，学生和个体贡献者可以更简单地自行创新。

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.

Chisel不像是大多数语言那样，嵌入在scala编程语言内。基本上，chisel是一个类和函数的库，用来表示同步数字电路的必要成分。一个chisel的设计是真正的scala程序，随着它的执行，生成一个电路。对于很多人来说，这是反直觉的："为什么不只是把chisel做成一个单个的语言，像是VHDL或是SystemVerilog？"我对这个问题的回答是如下：软件世界已经在设计方法上，过去的几十年里看到太多创新性发明。我们可以简单使用一个现代的编程语言，并免费享受它带来的好处，而不是把这些技术用在一个新的编程语言。

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.

一个常有的对chisel的批判是它“难以学会”。大多数这种想法是由于大多数大型，复杂的设计，这些被专家所设计出来，是用来满足他们自身的研究或是商业用途。当一个人学习一个受欢迎的语言，像是C++，他不需要从阅读GCC的源码开始，而是从，大量的课程，教科书，或是其它新手入门的学习资料开始。在这本《使用Chisel做数字设计》里，Martin创造了一个重要的资源，为了一些想要学习chisel的人们。

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.

Martin是一个有经验的教育者，从这本书的组织结构可以看出。从安装到重点开始，他像是搭建建筑一样，每一个砖块，一个砖块地，搭建了读者的理解。内附的练习部分是帮助巩固理解的水泥，确保每个概念都在读者的脑子里。这本书重点放在了硬件生成器，像是一个屋脊，撑起其余部分的结构。在最后，读者被提供了一个简单，有用的设计：一个RISC处理器。

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

在《使用Chisel做数字设计》，Martin提供了很强的基础知识，用于有效的数字设计。你怎么搭建，取决于你。

Jack Koenig Chisel and FIRRTL Maintainer Staff Engineer, SiFive

Jack Koenig Chisel和FIRRTL维护者 Staff Engineer，Sifive

Chapter 2

前言

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

这本书是一个专注于使用硬件构建语言chisel进行数字设计的导论。Chisel带来了软件工程的优势，像是面向对象和函数编程，进入数字设计。

这本书是给硬件设计师和软件工程师的。硬件设计师，有着Verilog或VHDL的知识，可以通过新型语言增加生产力，用于下一个ASIC或是FPGA设计。

软件工程师，有着面向对象语言和函数编程的知识，可以把这些知识用于生产硬件，例如，云FPGA。

这本书的方法是呈现小型到中型大小的典型硬件部分，去探索使用chisel做数字设计。

第二版的前言

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

随着Chisel允许敏捷硬件设计，所以也出现了公开访问和随时打印，允许敏捷教科书发行。在第一版以后的不到六个月时间，我可以提供一个改良和拓展的第二版。

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.

除了小的修改，主要的变更如下。测试部分增加。顺序搭建模块章节包含了更多的示例电路。一个新的章节在输入处理，解释了输入同步，表示了如何设计一个防抖动电路，和如何滤波一个噪声输入信号。这个示例设计章节已经增加了，表明了一个FIFO的不同设计。FIFO的变化也表明了在数字设计里如何使用类型参数和继承。

致谢

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

我想感谢所有在chisel上进行开发的人，他们创造了这样一个酷的硬件构建语言。chisel使用起来非常愉快， 于是写这样一本书很值得。

我很感谢整个chisel社区，非常对外欢迎友好，并且我从不对在chisel问问题感到疲惫。

我也想感谢去年在我的高级计算机架构课上的学生，他们大多数选择使用chisel作为最终项目。 感谢你们走出舒适区，走入了学习和使用前沿的硬件构建语言。 你们大多数的问题帮助规范了这本书。

Chapter 3

导论

This book is an introduction to digital system design using a modern hardware construction language, Chisel. 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.

这本书是一个使用现代硬件构建语言Chisel做数字系统设计的导论。在这本书，我们聚焦在比平常的硬件设计更高抽象层，使搭建更为复杂和交互性的硬件系统变得可能。

This book and Chisel are targeting two groups of developers: (1) hardware designers and (2) software programmers. Hardware designers who are fluid in VHDL or Verilog and using 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.

这本书和Chisel的目标群体是两种开发者：（1）硬件设计者和（2）软件工程师。精通VHDL或是Verilog，使用其它语言，像是Python, Java, 或是TCL/TK去生成硬件，使其变成一个硬件建造语言，这样，生成硬件作为语言的一部分。（2）对硬件设计有兴趣的软件程序员，例如，像是Intel未来的芯片会添加可编程硬件用来提高程序速度。

Chisel brings advances in software engineering, such as object-orientated and functional languages, into digital design. Chisel does not only allow to express hardware at the register-transfer level but allows you to write hardware generators.

Chisel将诸如面向对象和函数式语言之类的软件工程的优势引入数字系统设计。Chisel不仅仅允许表示硬件在寄存器传输的抽象层，而且允许你编写硬件生成器。

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.

硬件现在普遍通过硬件描述语言进行描述。描画硬件部分的时间，甚至是使用CAD工具画图，这个时代已经过去了。有一些高级草图可以给你一个系统的整体描绘，但是它们不是用来描述系统的。

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.

两个最常用的硬件描述语言是Verilog和VHDL。 这两个语言是古老的，包括大量规则，并且在综合到硬件的构建语言之间有一个变动的规则。 别理解错我的意思：VHDL和Verilog可以完美描述用于综合成ASIC的硬件模块。 对于Chisel的硬件设计，verilog充当一个测试和综合的中间语言。

This book is not a general introduction to hardware design and the fundamentals of it. For an introduction of the basics in 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 FPGA.[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 in 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.

这本书不是对硬件的设计和基础一般性介绍。 对有关数字设计基础知识的介绍，例如如何使用CMOS晶体管搭建一个门电路，你要参考其它数字设计书。 但是，这本书的目的是教你在一个抽象层中进行数字设计，作为当今描述ASIC或是设计 FPGA的例子。[2] 作为这本书的前置需求，我们假设你有一些基本的Boolean algebra和二进制数系统的知识。 更多的，一些任意编程语言的编程经验也是需要的。不需要Verilog或是VHDL的知识。 Chisel能够成为你的第一个编程语言用来描述数字硬件。 作为例子中的搭建过程是基于*sbt*和*make*，基本的命令行界面知识(CLI，又称terminal或是Unix shell)会是有用的。

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 in an expressive language. Chisel inherits the feature from Scala being “a language that grows on you”Scala. However, Scala is not the topic of this book.

Chisel用来本身不是一个大的语言。基本的部分在这里one page， 可以在数日内学习。 于是，这本书也不是一本大书。Chisel应该是比VHDL和Verilog更小, 继承很多规则。Chisel的力量来自于它是嵌入在Scala里的，Scala本身是一个有力的语言。 Chisel继承Scala作为“a language that grows on you’的Scala的特性 但是，Scala不是本书的话题。

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.

这本书是一个数字设计和Chisel语言的教学，不是Chisel语言的参考书籍也不是一本完整的芯片设计书

All code examples shown in this book are extracted from complete programs that have been compiled and tested. 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.

所有的书中代码例子来自经过编译和测试过的程序。所以，代码不应该含有任何语法问题。代码例子在本书的github repo里。除了提供Chisel代码以外，我也试图提供有用的设计和好的硬

[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.

[2] 因为作者相比于ASIC，更熟悉FPGA一些，书中的一些优化是针对FPGA的技术

件描述风格的规则。

所有的书中代码例子来自经过编译和测试过的程序。所以，代码不应该含有任何语法问题。代码例子在本书的GitHub 仓库里。除了提供Chisel代码以外，我也试图提供有用的设计和好的硬件描述风格的规则。

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

这本书在笔电或是平板上为阅读经过了优化。我们在字里行间提供了链接，大多数是Wikipedia的文章。

3.1 安装Chisel和FPGA工具

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 the Java JDK 1.8. As Oracle has changed the license for Java, it may be easier to install OpenJDK from AdoptOpenJDK

Chisel是一个Scala库，安装Chisel和Scala最简单的方法时通过*sbt*, Scala Build Tool。 Scala本身依赖于安装Java JDK 1.8。因为Oracle已经对Java更改了license，通过AdoptOpenJDK安装起来更容易一些。

3.1.1 macOS

Install the Java OpenJDK 8 from AdoptOpenJDK.On Mac OS X, with the packet manager Homebrew, *sbt* is installed with:

从AdoptOpenJDK安装Java OpenJDK 8。在Mac OS X中，通过包管理软件Homebrew，*sbt*可以通过此命令安装

```
$ brew install sbt git
```

Install GTKWave and IntelliJ (the community edition). When importing a project, select the JDK 1.8 you installed before (not Java 11!)

安装 GTKWave 和 IntelliJ (社区版本). 当引入一个项目，ĂL'择JDK 1.8，这是你在前边安装的 (不是 Java 11!)

3.1.2 Linux/Ubuntu

Install Java and useful tools in Ubuntu with:

在Ubuntu下安装Java和有用的工具，使用:

```
$ 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:

对于基于Debian的Ubuntu系统，软件通常通过(.deb)文件进行安装。 然而，当编写本文时，*sbt*还不能通过这种方法安装。 因此，安装过程有点麻烦:

```
echo "deb https://dl.bintray.com/sbt/debian /" | \
  sudo tee -a /etc/apt/sources.list.d/sbt.list
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 \
  --recv 2EE0EA64E40A89B84B2DF73499E82A75642AC823
sudo apt-get update
sudo apt-get install sbt
```

3.1.3 Windows

Install the Java OpenJDK 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 1.8 you installed before (not Java 11!) *sbt* can be installed with a Windows installer, see: Installing sbt on Windows. Install a git client.

从 AdoptOpenJDK安装Java OpenJDK。 Chisel和Scala也支持Windows操作系统。 安装 GTK-Wave 和 IntelliJ (社区版本)。 当引入一个项目, ĂĽ择JDK 1.8，你先前安装的（不是Java 11！）*sbt* 可以使用windows安装器安装，参照： 在windows安装sbt. 安装 git 客户端.

3.1.4 FPGA 工具

To build hardware for an FPGA, you need a synthesize tool. The two major FPGA vendors, Intel[3] and Xilinx, provide free versions of their tools that cover small to medium-sized FPGAs. Those medium-sized FPGAs are large enough to build multicore RISC style processors. Intel provides the Quartus Prime Lite Edition and Xilinx the Vivado Design Suite, WebPACK Edition.

为了构建FPGA硬件，你需要综合工具。两大主要的FPGA工具供应商，Intel[4]和Xilinx都提供了他们免费的工具，涵盖了小容量到中等容量大小的FPGA。 其中，中等规模的FPGA已经足够容纳多核RISC架构处理器。 Intel提供了Quartus Prime Lite 版本 和Vivado Design Suite, WebPACK 版本。

3.2 Hello World

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

每一本编程语言书都会从叫做*Hello World*的最小例子开始。以下代码是第一种方法。

```

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

Compiling and executing this short program with *sbt*
编译, 执行这个简短的程序通过*sbt*:

```
$ sbt "runMain HelloScala"
```

[3]former Altera
[4]曾经是Altera

leads to the expected output of a Hello World program:
随后，Hello World程序输出了预期的内容：

```
[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.
然而，这就是Chisel吗？这个硬件被生成用于出书字符串吗？并不是这样，这里实际上是解释了Scala代码，并不代表着硬件设计的Hello World程序。

3.3 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!
那么Hello World程序等价的硬件设计是什么？是最小的可用并且可视的设计？一个闪烁的LED是硬件（或者嵌入式软件）版本的Hello World。如果一个LED闪烁，我们就可以解决更大的问题。

```
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 3.1: hello

listing*hello* 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.
*hello*展现的是一个通过Chisel描述的闪烁LED。你是否理解这个代码案例的细节是不重要的。我们将会介绍这些内容在以下的章节中。 仅仅需要注意这个电路通常有着高频率的时钟，

例如50MHz，我们需要一个计数器来分频得到Hz级别的频率来实现可视化的闪烁。在这个例子中我们从0计数到25000000-1, 然后触发闪烁信号(*blkReg := ~blkReg*) , 然后复位计数器 (*cntReg := 0.U*)。这个硬件以1 Hz频率闪烁LED。

3.4 Chisel的IDE

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

这本书没有假设你的编程环境和使用的编辑器。 在命令行中使用*sbt*同时使用一个你偏好的编辑器可以很容易的入门。在其他传统的书籍中， 所有你需要键入终端的命令之前会有一个*$*字符，这个字符你不需要输入。一个例子，Unix的列出当前文件夹下的文件*ls* 命令：

```
$ 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 and Eclipse to generate a project from the sbt project configuration in *build.sbt*.

也就是说，一个编译器后台运行的集成开发环境(IDE)，可以加速编程。因为Chisel是一个Scala库，所有的IDEs都支持Scala，也支持Chisel。 在 IntelliJ 和 Eclipse 中生成一个包含*build.sbt*配置文件的sbt工程是可行了。

In IntelliJ 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.

在IntelliJ，你可以通过*File - New - Project from Existing Sources...*创建新的项目，从已有的资源，然后选择项目的build.sbt文件。

In Eclipse you can create a project via 在Eclipse你可以建立一个项目通过

```
$ sbt eclipse
```

and import that project into Eclipse.[5]

并且导入一个工程入Eclipse。[6]

Visual Studio Code is another option for a Chisel IDE. TheScala 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*.

Visual Studio Code 是另一个Chisel IDE的选项。 Scala Metals 拓展提供了Scala支持。 左侧栏选择 *Extensions* ， 搜索 *Metals*，安装*Scala (Metals)*. 为了引入 sbt 为基础的项目， *File - Open*打开文件夹。

[5]This function needs the Eclipse plugin for sbt.
[6]这个功能需要sbt的Eclipse插件。

3.5 访问源码和电子书功能

This book is open source and hosted at GitHub: chisel-book. All Chisel code examples, shown in this book, are included in the repository. The code compiles with a recent version of Chisel, and many examples also include a test bench. We collect larger Chisel examples in the accompanying repository chisel-examples. 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.

这本书是在GitHub: chisel-book上开源。书中所有的Chisel例子都包含在该仓库中。 这些代码通过最近的Chisel版面编译，很多例子都包含了test bench。 我们收集了很多Chisel例子在chisel-examples仓库里。如果你找到了错误或者笔误，可以提交pull request。 你可以在github上记录一个issue来提供反馈或者评论或者发送一个简单的学校邮件。

This book is freely available as a PDF eBook and in classical printed form. 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.

这本书有免费的古典印刷体PDF电子版。 电子书提供了更多的资源和Wikipedia入口。 我们使用Wikipedia来说明与本书部直接相关的背景知识(例如二进制数字系统)。 我们优化了电子书的排版，以方便便携式设备阅读，例如ipad。

3.6 更多读物

Here a list of further reading for digital design and Chisel: 这儿例出了关于数字电路设计和Chisel更深层次的读物：

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

  Digital Design: A Systems Approach, 作者是William J. Dally 和 R. Curtis Harting, ，是一本现代的数字设计的教科书。它有两个版本，使用Verilog或是VHDL作为硬件描述语言。

The official Chisel documentation and further documents are available online:
Chisel官方文档和之后的文档都会更新到网上：

- The Chisel home page is the official starting point to download and learn Chisel.

  Chisel主页是开始安装和学习chisel的地方。

- The Chisel Tutorial provides a ready setup project containing small exercises with testers and solutions.

  Chisel Tutorial 提供了一个准备安装的项目， 包括小的测试和解决答案的练习。

  The Chisel Wiki contains a short users guide to Chisel and links to further information.

  Chisel Wiki包含了小的chisel的用户手册 和更多信息的链接。

- The Chisel Testers are in their repository that contains a Wiki documentation.

  Chisel Testers 包括在他们的仓库， 包含了一个wiki文档。

- The Generator Bootcamp is a Chisel course focusing on hardware generators, as a Jupyter notebook

  Generator Bootcamp 是chisel课程，专注在硬件生成器， 作为jupyter notebook。

- A Chisel Style Guide by Christopher Celio.

  Chisel Style Guide，作者是Christopher Celio。

- The chisel-lab contains Chisel exercises for the course “Digital Electronics 2” at the Technical University of Denmark.

- The chisel-lab 包括用于在丹麦技术大学“Digital Electronics 2” 课程的Chisel练习。

3.7 练习

Each chapter ends with a hands-on exercise. For the introduction exercise, we will use an FPGA board to get one LED blinking. 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:

每个章节结尾都有手写的练习题。为了介绍练习，我们通常会使用一个FPGA板子上的一颗LED。 第一步clone或者forkchisel-examples仓库。 Hello World例子在*hello-world*文件夹里，是一个最小的工程。你可以探索LED闪烁的Chisel代码在*src/main/scala/Hello.scala*。 按下下述步骤编译LED闪烁工程:

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

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).

在Chisel部分下载完成之后，程序会产出一个Verilog文件*Hello.v*。 浏览这个Verilog文件，你会看到电路包含两个输入*clock*和*reset* 和一个输出*io_led*。当你对比Verilog文件和Chisel模块时，你将会发现Chisel模块申明时不包含*clock*和*reset*。 这些信号是隐式生成的，在大部分设计中处理这些底层细节是不方便的。Chisel提供的寄存器如果需要，会自动连接*clock*和*reset*.

The next step is to set up an FPGA project file for the synthesize tool, assign the pins, and compile[7] the Verilog code, and configure the FPGA with the resulting bitfile. We cannot provide the details of these

[7]The real process is more elaborated 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.

steps. Please consult the manual of your Intel Quartus or Xilinx 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.

下一步是建立FPGA工程文件，分配引脚，然后编译Verilog代码，最后利用生成的bit文件烧写进FPGA。 我们无法提供这些步骤的详细过程。请参考你的EDA工具手册。 然而，例程仓库里面包含了一些针对几个流行的FPGA的Quartus可用的工程在*quartus*文件夹中。 如果仓库支持的板子包括你的FPGA，启动Quartus，打开工程，编译，烧录，然后一个LED将会闪烁。

Gratulation! You managed to get your first design in Chisel running in an FPGA! 厉害了！ 你成功在FPGA上运行了你第一个Chisel电路！

If the LED is not blinking, check the status of reset. On the DE2-115 configuration, the reset input is connected to SW0. 如果LED没有闪烁，检查reset的状态。在DE2-115配置中，reset的输入连接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 different "emotions". E.g., 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.

现在改变闪烁的频率然后重新运行一遍构建工程。闪烁的频率和闪烁的方式可以预示不同的状态。例如一个缓慢的闪烁信代表一切正常，一个快速的闪烁信号代表一个报警。探索哪个频率对应这两个不同的动作。

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? 一个更具有挑战性的练习，生成一个下述的闪烁模式： LED一秒中亮200 ms。对于这个模式，你可以会从计数器复位来分离LED的闪烁。你将会需要第二个常量来改变*blkReg*寄存器的状态。这个模式会产生什么种类的状态呢？ 是警报还是更像一个正常运行的信号呢？

If you do not have an FPGA board (yet), 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:

如果你（还）没有一个FPGA板，你仍然可以运行闪烁LED利子。 你会使用模拟Chisel。为了防止太长的模拟时间，改变Chisel代码的时钟频率，从50000000改为50000。执行以下指令，去模拟闪烁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.

这会执行测试，运行时钟100万次。 闪烁频率取决于模拟速度，也就是你的计算机速度。于是，你需要实验一下，使用假设的时钟频率，去看到模拟的闪烁LED。

Chapter 4

基本组成部分

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.

在这个章节，我们介绍数字设计的组成部分。这些基本的组成部分可以被合并到更大，更有趣的电路中。

Digital systems in general built 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 following terms: low/high, false/true, and de-asserted/asserted. These terms mean the same two possible values of a binary signal.

数字系统总的来说，使用二进制信号，意味着一个bit或是信号，只可以有二者之一的可能值。这些数值常常被称为0或1。但是，我们经常使用如下的术语：低/高，假/真，和非声明的/已声明的。这些术语的含义是相同的，指的是一个二进制信号的两个可能值。

4.1 信号类型和常量

Chisel provides three data types to describe signals, 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:

Chisel提供了三种数据类型用来描述型号，组合逻辑，和寄存器：*Bits*，*UInt*，和*SInt*。*UInt*和codeSInt是*Bits*的拓展，并且所有的三种类型代表bits的矢量。*UInt*表示这个bits的矢量是一个无符号的整型，*SInt*表示一个有符号的整型。[2] Chisel使用two's complement表示有符号数的整型。以下是不同类型的定义，8位*Bits*，8位无符号整型，和一个10位有符号整型：

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

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

[2]*Bits*类型在本Chisel版本是遗失的操作，并且不是特别在使用者的代码中有用.

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 is used for the definition of the *Bits* vector:

Bits的矢量宽度被Chisel的width类型(*Width*)定义。以下表示把scala的整型*n*转换成Chisel的*width*，用于*Bits*矢量的定义。。

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

Constants can be defined by using a Scala integer and converting it to a Chisel type:
常量可以通过Scala整型定义并把它转换成Chisel类型。定义一个为0的UInt常量，和定义一个为03的SInt常量。

```
0.U
−3.S
```

Constants can also be defined with a width, by using the Chisel width type:
常量也可以随着宽度被定义，使用Chisel的width类型。 定义一个4位的常量8。

```
8.U(4.W)
```

If you find the notion of 8.U and 4.W a little bit funny, consider it as a variant of an integer constant with a type. This notation is similar to 8L, representing a long integer constant in C, Java, and Scala.
当你发现8.U和4.W的表示方式有些有趣，你可以认为它是一个整型常量附带一个类型。这个表示方式类似于8L，代表一个C，Java或Scala中的长整型。

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.

常见问题: 一个可能的错误是，当我们定义一个常量 *.W* ， 定义一个宽度。 例如, *1.U(32)* 不会定义一个32位宽的代表1的常量。 相反，表达式 *(32)*被翻译为从32位的按位抓取，结果是一个单位元的常量0。可能不是编程者的原意。

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.

Chisel受益于Scala的类型推断，并且很多地方类型信息可以被省略。类似的也适用于位宽。在很多时候，Chisel会自动推断正确的宽度。于是，chisel描述的硬件语言比VHDL或Verilog更加简洁和可读。

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.

对于以其它作为基底的十进制以外的常量，常量被定义为字符串，开头*h*是16进制，*o*是8进制，*b*是2进制。 以下的例子表明了常量255的定义，在不同的基底。在这个例子，我们省略了位宽，chisel推断了最小宽度用来表示常量，在这个例子是8位。（16进制表示255， 8进制表示255， 二进制表示255）

```
"hff".U
"o377".U
"b1111_1111".U
```

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

以上代码也表示了如何使用下划线去群组数字来标识常量。下划线是被忽略的。

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.

为了表示逻辑值，Chisel定义了*Bool*类型。Bool可以表示*true*或*false*值。 以下的代码表示了*Bool*类型的定义以及*Bool*常量的定义，通过转换Scala Boolean常量*true*和*false*， 到Chisel的*Bool*类型。

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

4.2 组合电路

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.

Chisel使用Boolean algebra操作符， 和C，Java，Scala和可能很多其它编程语言中定义的一样，去描述组合电路。 以下代码定义了一个对*a*和*b*进行*and*逻辑，然后把它的结果和*c*进行*or*逻辑的电路：

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

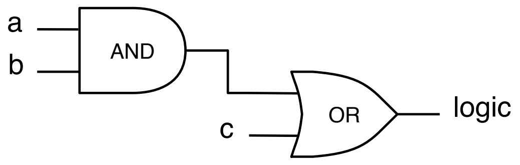

Figure 4.1: Logic for the expression . The wires can be a single bit or multiple bits. The Chisel expression, and the schematics are the same.

Figure 4.1 shows the schematic of this combinatorial expression. Note that this circuit may be for a vector of bits and not only single wires that are combined with the AND and OR circuits.

图片 4.1 表明了这个组合逻辑的表达的草图。 注意到这个电路可能用来表示一个向量的bits，并不仅是并不仅是单独的AND和OR组成的线路

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:

在这个例子，我们不定义类型或是信号*logic*的宽度。两个都是从类型和表达式位宽推演过来的. 标准Chisel的逻辑操作是：

```
val and = a & b // bitwise and
val or = a | b // bitwise or
val xor = a ^ b // bitwise xor
val not = ~ a // bitwise negation
```

The arithmetic operations use the standard operators:

```
val add = a + b // addition
val sub = a - b // subtraction
val neg = -a // negate
val mul = a * b // multiplication
val div = a / b // division
val mod = a % b // modulo operation
```

算术操作使用标准操作符

算术操作使用标准操作符1. 加法 2. 减法 3. 相反数 4. 乘法 5. 除法 6. 余数：

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.[3]

加法和减法操作结果的宽度是操作数的最大宽度，乘法操作结果的库纳杜是两个操作数的位宽加和, 除法和余数操作的结果是被除数的位宽。[4]

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.

一个信号也可以先被定义为某种类型的*Wire*。随后，我们可以给这个wire赋一个值，使用:=。

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

w := a & b
```

A single bit can be extracted as follows:

一个简单的bit可以被如下提取：

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

A subfield can be extracted from end to start position:

一个分割可以从终点到起点提取：

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

Bit fields are concatenated with *Cat*.

bit域通过*Cat*合并.

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

[4]详见FIRRTL specification.

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	
Γ	NOT	UInt, SInt, Bool
&| `ˆ	AND, OR, XOR	UInt, SInt, Bool
!	logical NOT	Bool
&& ||	logical AND, OR	Bool

Table 4.1: 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
Cat(a, b, ...)	bit field concatenation	UInt, SInt

Table 4.2: Chisel defined hardware functions, invoked on *v*.

```
val word = Cat(highByte, lowByte)
```

Table 4.2 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 praxis to use parentheses.[5]

Table 4.2表示了操作符号的全列表 (查看builtin operators)。 Chisel操作符优先级取决于电路的赋值顺序，遵守Scala operator precedence。 如果有疑问的话，使用括号是一个好的习惯。[6]

Table 4.2 shows various functions defined on and for Chisel data types.

Table 4.2表示了操作符号的全列表

4.2.1 复用器

A multiplexer is a circuit that selects between alternatives. In the most basic form, it selects between two alternatives. Figure 4.2 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*.

复用器是一个选择选项的电路。 在最基本的形式，它在二者选择其一。图2.2 4.2表示一个二选一复用器，或是用mux简单表示。 取决于选择信号(*sel*)，*y*会表示信号*a*或是信号*b*。

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

[5]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.

[6]在chisel里操作符号的优先级是硬件优化的副产品，因为硬件节点数 是通过执行scala操作符创建的。Scala操作符的先后顺序与Java/C类似但是不等同。Verilog和C有着同样的操作符优先级，但是VHDL与之不同。 Verilog有逻辑上的操作符优先级，但是VHDL里，这些操作符的优先级是等同的，按照从左到右的顺序

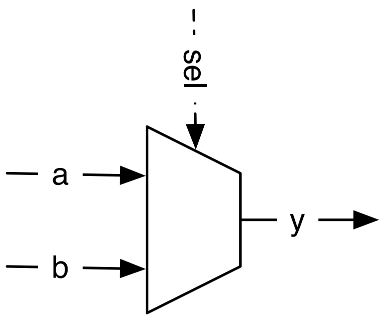

Figure 4.2: A basic 2:1 multiplexer.

一个复用器可以通过逻辑简单搭建。但是，复用是一个常用操作，Chisel提供了一个复用器。

```
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.

这里如果*sel*是*true.B*的话，选择*a*，反之选择*b*。 *sel*的类型是Chisel*Bool*， *a*和*b*作为输入可以是任何Chisel基本类型或是集合（捆束或是矢量）， 只要它们的类型相同。

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 a later chapter.

通过逻辑和算术操作，一个复用器，每个组合电路都能被描述。 但是，Chisel提供了更多部分和控制的抽象用来更优雅地描述组合电路，这些在稍后章节讲。

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

第二个需要描述一个数字电路的基本组成部分是一个状态单元，又称寄存器，下边讲。

4.3 状态寄存器

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.

Chisel提供了一个寄存器，这是一个D flip-flops的集合。这个寄存器隐含连接到一个总时钟，并且上升触发。当一个初始值随着在寄存器声明的时候被提供，它使用的是一个同步复位，连接到总复位信号。

A register can be any Chisel type that can be represented as a collection of bits. Following code defines an 8-bit register, initialized with 0 at reset:

一个寄存器可以是任何bits集合的Chisel类型。下边的代码定义了一个八位寄存器，在复位初始化为0。

```
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:
一个输入连接到寄存器，通过:=更新操作数，输出的寄存器可以使用表达式通过名字调用。

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

A register can also be connected to its input at the definition:
寄存器也可以连接到它的输入使用如下定义:

```
val regNxt = RegNext(d);
```

A register can also be connected to its input and a constant as initial value at the definition:
也可以连接到它的输入并使用一个常量作为初始值作为定义:

```
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 identifier consisting of several words. The convention is to start functions and variables with a lower case and classes (types) with an upper case.

为了区分表示组合逻辑和寄存器的信号，一个常见的方式是在寄存器前边加上前缀*Reg*。另一个常见的方式，来自Java和Scala，是去使用camelCase，由几个单词组成的标识。这个方式是函数和变量用首字母小写，类用首字母大写。

4.3.1 计数

Counting is a fundamental operation in digital systems. On 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.

计数是一个非常基本的操作在数字系统。计数可能时常发生。但是，经常计数是被使用去定义一个时间间隔。计数时钟，并引发一个操作，当时间间隔过期的时候。

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 till 10, 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.

一个简单的方式是计数到一个值。但是，在计算机科学和数字设计，计数从0开始。于是，如果我们想要数到10，我们是从0数到9.以下代码表示一个计数器，数到9，并返回到0当数到9的时候。

```
val cntReg = RegInit (0.U(8.W))
cntReg := Mux(cntReg === 100.U, 0.U, cntReg + 1.U)
```

4.4 使用Bundle和Vec进行结构

Chisel provides two constructs to group related signals: (1) a *Bundle* to group signals of different types and (2) a *Vec* to represent an indexable collection of signals of the same type. *Bundle*s and *Vec*s can be arbitrarily nested.

Chisel提供两个集成小组相关的信号：(1) 一个是*Bundle*， 组织不同类型信号 和 (2) 一个是*Vec*， 去代表一个可索引的相同类型的信号。 *Bundles*和*Vecs*可以任意交织。

A Chisel bundle groups several signals. The entire bundle can be referenced as a whole, or individual fields can be accessed by their name. We can define a bundle (collection of signals) by defining a class that extends Bundle and list the fields as *val*s within the constructor block.

Chisel捆束多个信号。整个bundle可以被整体引用，或是通过他们的名字分别访问。 我们可以定义一个捆束（信号的集合），通过定义一个类型，拓展了*Bundle*， 并在域内通过*val*列出。

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

val b = ch.valid
```

To use a bundle, we create it with *new* and wrap it into a *Wire*. The fields are accessed with the dot notation:
为了使用捆束，我们使用*new*，并把它包裹进*Wire*。域通过点标识访问。

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 is similar to a *struct* in C, a *record* in VHDL, or a *struct* in SystemVerilog. A bundle can also be referenced as a whole:

点标识是面向对象语言的常见做法，*x.y*意味着*x*是一个对象的引用，*y*是那个对象的域。 因为Chisel是面向对象的，我们使用点标识去访问捆束的域。 一个捆束类似于C语言的*struct*， VHDL的*record*， 或是SystemVerilog的*struct*。 捆束可以整体被引用。

```
val channel = ch
```

A Chisel *Vec* represents a collection of signals of the same type (a vector). Each element can be accessed by an index. A Chisel *Vec* is similar to array data structures in other programing languages.[7] 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*

Chisel的*Vec*代表一系列相同类型的信号。 每个元素可以通过索引访问。Chisel的*Vec*类似于一串数据结构在其它编程语言。[8] *Vec*通过使用两个参数传入构造函数：元素的数量和元素的类型。组合逻辑 *Vec* 需要被打包进*Wire*。
Individual elements are accessed with *(index)*. 单一元素通过*(index)*被访问到。

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

[7]The name *Array* is already used in Scala.
[8]*Array* 已经在scala中被使用。

```
v(2) := 5.U

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

A vector wrapped into a *Wire* is a multiplexer. We can also wrap a vector into a register to define an array of registers. Following example defines a register file for a processor; 32 registers each 32-bits wide, as for a classic 32-bit RISC processor, like the 32-bit version of RISC-V.

被包裹进*Wire*的向量是一个复用器。 我们也可以把向量传入寄存器去定义一列寄存器。 以下例子定义了用于处理器的寄存器文件；32位寄存器，每个寄存器有32位宽，例如经典的32位RISC处理器， 或是32-bit版本的RISC-V。

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

An element of that register file is accessed with an index and used as a normal register. 寄存器的一个元素通过索引访问，并用作正常寄存器。

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

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:

我们可以随意混搭捆束和向量。当创造一个有着捆束类型的向量，我们需要把这个类型传入向量域。 使用我们的*Channel*， 像我们以上定义的，我们可以创建一个向量的通道通过：

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

A bundle may as well contain a vector: 一个捆束可能包含一个向量：

```
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 passing this bundle to a *RegInit*:

当我们想要一个需要reset值的寄存器的类型，我们首先创造一个具有那个捆束的*Wire*，设置单独的域， 然后把捆束传给*RegInit*。

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

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

val channelReg = RegInit(initVal)
```

With combinations of *Bundle*s and *Vec*s we can define our own data structures, which are powerful abstractions. 通过*Bundle*和*Vec*的组合，我们可以定义我们自己的数据结构，这个是有力的抽象。

4.5 Chisel生成的硬件

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*.

在看到一些Chisel代码，可能看起来和经典编程语言像是Java或C比较相似。但是，Chisel（或是其它硬件描述语言） 描述了硬件部分。而在软件项目中，一行代码在前一行过后执行，而硬件执行全部*execute in parallel*。

It is essential to keep in mind that Chisel code does generate 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.

必须记住，Chisel代码确实会生成硬件。 尝试想象或画在纸上的单个模块，通过您的Chisel电路描述生成。 每次创建组件都会添加硬件。每个赋值声明产生门和/或触发器。

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 can spill out Verilog code for ASIC or FPGA synthesis or can be tested with a Chisel tester. The network of hardware nodes is what is executed in fully parallel.

从技术上讲，Chisel执行代码时，它将作为Scala程序运行，并且 通过执行Chisel语句，它收集硬件组件 并连接这些节点。硬件节点网络就是硬件， 可能会产生用于ASIC或FPGA综合的Verilog代码，或者 用Chisel测试仪测试。 硬件节点网络是完全并行执行的。

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

对于软件工程师，可以想象一下，您可以 在硬件中创建，而无需将应用程序划分为线程 并正确锁定通信。

4.6 练习

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))*.

在导论中，你补充了一个闪烁的LED在一个FPGA板上（来自chisel-example），这个是一个类似硬件*Hello world*的例子。它只是使用内部的状态，一个LED输出，没有输入。把那个项目复制到一个新的文件夹，并使用*val sw = Input(UInt(2.W))*添加一些输入到*io*捆束。

```
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 and list the fields as pin names 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).

对于这些开关，你也可以给FPGA板上的PIN分配名字。你可以找到管脚命名的例子在Quartus项目文件中的ALU项目（例如，DE2-115 FPGA 板子） (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.

当你已经定义了这些输入和管脚分配，开始一个简单的测试：关掉设计中所有闪烁逻辑，连接其中一个到LED输出；编译并设置FPGA期间。你可以通过开关把LED打开并关闭吗？如果是对的，你就有了可用的输入。如果没有，你需要给你的FPGA设置debug。这个也可以通过GUI版本的工具完成。

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.

现在使用两个开关并完成一个基本的组合电路函数，例如，二输入与门并在LED显示结果。改变函数。下一步包括三输入开关来完成一个复用器，一个充当选择信号，另外两个充当双输入复用器的输入。

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.

现在你已经可以补充简单的组合函数并在一个实际的FPGA进行简单的测试。作为下一步，我们会看一下搭建过程如何生成FPGA设置。更多地，我们也会探索一个简单的Chisel测试框架，这允许你不去设置FPGA和开关来测试电路。

Chapter 5

搭建过程和测试

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.

为了开始更有趣的Chisel代码，我们第一需要学习如何编译Chisel程序，如何生成Verilog代码用来在FPGA执行，和如何写测试用于debug和验证我们的电路是正确的。

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.

Chisel是用Scala写的，所以任何的支持Scala的搭建过程适用于Chisel项目。一个受欢迎的Scala搭建工具是sbt，sbt是Scala interactive Build Tool的简写。除了驱动搭建和测试过程，*sbt*也下载正确的Scala版本和Chisel 库。

5.1 使用sbt搭建你的项目

The Scala library that represents Chisel and the Chisel testers 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 actual version of Chisel. However, this means on each build the version is looked up from the Maven repository. This lookup needs an Internet connection for the build to succeed. Better use a dedicated version of Chisel and all other Scala libraries in your *build.sbt*. Maybe sometimes it is also 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.

Scala 库中表示Chisel和Chisel测试器的部分，是通过搭建过程从一个Maven仓库中下载的。库文件通过build.sbt被引用。它可以通过build.sbt设置，使用*latest.release*总是用最新的Chisel版本。但是，这意味这每次搭建都要查看Maven仓库。查看需要互联网连接，为了搭建成功。最好使用一个特定的Chisel版本，所有其它的Scala库在你的*build.sbt*。可能有的时候你能够手写硬件代码并且在无网络连接的情况下测试是值得提倡的。例如，帅气地在飞机上进行硬件设计。

5.1.1 源文件结构

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

sbt继承来自于Maven自动化搭建工具的源文件法则。 Maven也管理者开源Java函数库的仓库。[2]

```
project
└─src
│  └─main
│  │  └─scala
│  │     └─package
│  │        └─sub-package
│  └─test
│     └─scala
│        └─package
└─target
└─generated
```

Figure 5.1: Source tree of a Chisel project (using *sbt*)

Figure 5.1 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*, containing the hardware sources and *test* containing testers. Chisel inherits from Scala, which 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*.

Figure 5.1表示了一个常见的Chisel项目的源文件树的结构。 项目的根目录是项目home地址， 这里包含了build.sbt。 也可以包含一个Makefile用于搭建过程， 一个README， 和一个LICENSE文件。 文件夹sec包含了所有源代码。 在这里， main包含硬件源代码和test包含测试器， 有一个分叉。 Chisel继承自Scala， Scala继承自Java的源文件软件包的结构。 软件包把你的Chisel代码组织成命名空间。软件包也可以包括下属软件包。 文件夹target包括类文件和其它产生的文件。 我也推荐用一个文件放置生成的Verilog文件， 这个文件夹一般称为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 *mypacket*:

为了使用Chisel命名空间的工具， 你需要声明类或模块在软件包被定义， 在这个例子里， 在*mypacket*:

```
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* packet to use Chisel classes.

注意在这个例子我们看到引入chisel3软件包， 和使用chisel类型。

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

[2]: `https://mvnrepository.com/artifact/edu.berkeley.cs/chisel3`.

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

为了使用*Abc*模块在一个不同的地方（软件包命名空间），软件包的*mypacket*需要被引用。下划线(_)充当万用字元，意味着所有*mypacket*的类都要被引用。

```
import mypack._
class AbcUser extends Module {
val io = IO(new Bundle {})
val abc = new Abc ()
}
```

也可以不去引用来自mypack的所有类别，而是使用全名mypack.Abc代表mypack中的模块Abc

```
class AbcUser2 extends Module {
val io = IO(new Bundle {})
val abc = new mypack.Abc ()
}
```

引用只是一个单个的类，并创造它也是可以的

```
import mypack.Abc
class AbcUser3 extends Module {
val io = IO(new Bundle {})
val abc = new Abc ()
}
```

5.1.2 运行sbt

一个Chisel项目可以被编译并执行通过一个简单的sbt命令：

```
$ sbt run
```

This command will compile all your Chisel code from the source tree and searches for classes that contain an *object* that includes a *main* method, or simpler that 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*:

这个命令会编译你所有的源文件树下的Chisel代码并搜索含有object和含有main方法的类别，或是简化的App。如果由多余一个类似的对象，所有的对象都会被列出以及可选。你也可以直接执行被传入sbt作为参数的对象：

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

Per default *sbt* searches only the *main* part of the source tree and not the *test* part.[3] However, Chisel testers, as described here, contain a *main*, but shall be placed in the *test* part of the source tree. To execute a *main* in the tester tree use following *sbt* command:

[3]This is a convention form Java/Scala that the test folder contains unit tests and not objects with a *main*.

根据默认sbt只是搜索main部分的源文件树而不是测试部分。但是，Chisel测试器，在这里描述的，含有一个main，但是应该放在源文件树的test部分。为了执行测试树的main，使用如下sbt命令：

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

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.

现在我们知道了Chisel项目的基本构成和如何使用sbt编译运行，我们可以继续开始一个简单的测试框架了。

5.1.3 工具流程

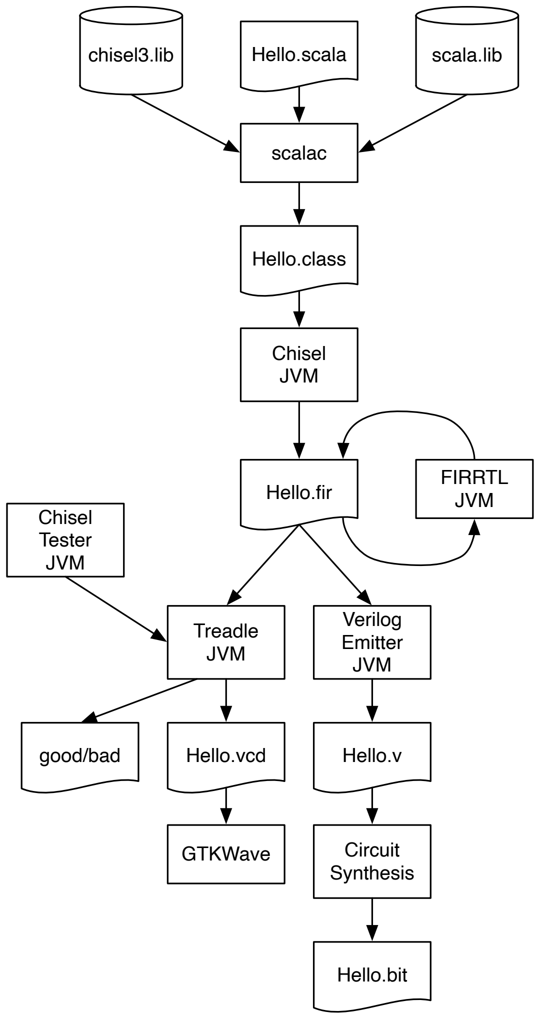

Figure 5.2: Tool flow of the Chisel ecosystem.

Figure 5.2 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 Hello.class that can be executed by a standard Java virtual machine (JVM). Executing this class with a Chisel driver generates the so-called 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.

图片 5.2表示了Chisel的工具流程。这个数字电路，在Chisel类，被表示位Hello.scala。scala编译器编译了这个类，和Chisel和Scala库，并生成了能被一个标准Java虚拟机 (JVM)执行的Java类。通过Chisel驱动器，执行这个类，生成了所谓RTL的灵活中间表达（FIRRTL），一个数字电路的中间表达。在我们的例子里，这个文件是Hello.fir。 这个FIRRTL编译器完成了电路的转换。

Treadle is a FIRRTL interpreter to 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).

Treadle是一个FIRRTL表示器，去模拟一个电路。联合Chisel测试器，它可以用来debug和测试Chisel电路。 通过assertion，我们可以提供测试结果。 Treadle也可以生成波形文件(Hello.vcd)，可以通过波形观察器（例如，免费的观察器GTKWave或是Modelsim）进行观察。

One FIRRTL transformation, the Verilog emitter, generates Verilog code for synthesis (Hello.v). A circuit synthesize tool (e.g., Intel Quartus, 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.

一个FIRRTL变换，Verilog发射器，生成用于综合的Verilog代码(Hello.v)。 电路综合工具（例如，Intel的Quartus，Xilinx Vivado，或是ASIC工具）综合电路。 在一个FPGA设计流程里，工具产生了FPGA的bitstream，用于设置FPGA，例如，Hello.bit。

5.2 使用Chisel测试

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.

测试硬件设计一般称为testbench。 这些testbench初始化被测试的设计（DUT），驱动输入端口，观察输出端口，与它们和期待值比较。

5.2.1 PeekPokeTester

Chisel provides test benches in the form of a *PeekPokeTester*. One strength of Chisel is that it can use the full power of Scala to write those 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 [**?**].

Chisel提供的testbench叫PeekPokeTester。 其中Chisel的一个优势是它能够全力使用Scala写入这些testbench。 一个人，比方说，可以在软件模拟器编写期望的硬件功能，并把硬件仿真和软件仿真进行比较。 这个方法是非常有效的，当测试写好的处理器的时候。

To use the *PeekPokeTester*, following packages need to be imported:

使用PeekPokeTester，以下软件包需要引入：

```
import chisel3._
import chisel3.iotesters._
```

Testing a circuit contains (at least) three components: (1) the device under test (often called DUT), (2) the testing logic, also called test bench, and (3) the tester objects that contains the *main* function to start the testing.

测试电路需要（至少）三个部分：1. 接受测试的器件（经常称为DUT）2. 测试逻辑，也称testbench 3. 包含main函数的测试对象用来开始测试。

The following code shows our simple design under test. It contains two input ports and one output port, all with a 2-bit width. The circuit does a bit-wise AND to it returns on the output:

以下代码表明了我们简单的接受测试的设计。它包含输入端口和一个输出端口，全是2位宽的。这个电路执行按位AND并返回给输出：

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

The test bench for this DUT extends *PeekPokeTester* and has the DUT as a parameter for the constructor:

该DUT的testbench拓展了*PeekPokeTester*，并把DUT作为建造器的参数：

```
class TesterSimple(dut: DeviceUnderTest ) extends
PeekPokeTester(dut) {
poke(dut.io.a, 0.U)
poke(dut.io.b, 1.U)
step (1)
println("Result is: " + peek(dut.io.out).toString)
poke(dut.io.a, 3.U)
poke(dut.io.b, 2.U)
step (1)
println("Result is: " + peek(dut.io.out).toString)
}
```

A *PeekPokeTester* can set input values with *poke()* and read back output values with *peek()*. The tester advances the simulation by one step (= one clock cycle) with *step(1)*. We can print the values of the outputs with *println()*.

*PeekPokeTester*可以使用*poke()*设置初始值，并通过*peek()*读出数值。测试器通过进一步的*step(1)*声明，增加一个周期过后的数值。我们可以使用*println()*打印输出。

The test is created and run with the following tester main: 测试建立，并通过以下运行：

```
object TesterSimple extends App {
  chisel3.iotesters.Driver(() => new DeviceUnderTest()) { c =>
    new TesterSimple(c)
  }
}
```

When you run the test, you will see the results printed to the terminal (besides other information): 当你运行测试，你会看见除了其它信息以外，打印到端口的结果。

```
[info] [0.004] SEED 1544207645120
[info] [0.008] Result is: 0
[info] [0.009] Result is: 2
test DeviceUnderTest Success: 0 tests passed in 7 cycles
taking 0.021820 seconds
[info] [0.010] RAN 2 CYCLES PASSED
```

We see that 0 AND 1 results in 0; 3 AND 2 results in 2. Besides manually inspecting printouts, which is an excellent starting point, we can also express our expectations in the test bench itself with *expect()*, having the output port and the expected value as parameters. The following example shows testing with *expect()*:
我们看到0和1的与是0；3和2的与是2。除此之外手动查看结果，这个是一个极好的开始点，我们也可以在测试台上使用*expect()*， 用来表示我们的期待值，放在输出端口，并传入期待值。以下的例子是使用*expect()*作为测试的例子。

```scala
class Tester(dut: DeviceUnderTest) extends PeekPokeTester(dut) {

  poke(dut.io.a, 3.U)
  poke(dut.io.b, 1.U)
  step(1)
  expect(dut.io.out, 1)
  poke(dut.io.a, 2.U)
  poke(dut.io.b, 0.U)
  step(1)
  expect(dut.io.out, 0)
}
```

Executing this test does not print out any values from the hardware, but that all tests passed as all expect values are correct.
执行测试不打印出任何硬件的值， 但是所有通过期待值的测试代表正确。

```
[info] [0.001] SEED 1544208437832
test DeviceUnderTest Success: 2 tests passed in 7 cycles
taking 0.018000 seconds
[info] [0.009] RAN 2 CYCLES PASSED
```

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:
一个失败的测试，当DUT或是测试台包含了一个错误， 产生了一个错误消息，描述了期待值和实际值的差异。在以下，我们把测试台的期待值为4，产生了一个错误：

```
[info] [0.002] SEED 1544208642263
[info] [0.011] EXPECT AT 2   io_out got 0 expected 4 FAIL
test DeviceUnderTest Success: 1 tests passed in 7 cycles
taking 0.022101 seconds
[info] [0.012] RAN 2 CYCLES FAILED FIRST AT CYCLE 2
```

In this section, we described the basic testing facility with Chisel for simple tests. However, in Chisel, the full power of Scala is available to write testers.

在本节中，我们描述了Chisel用于简单测试的基本测试工具。 但是，在Chisel中，Scala的全部功能可用于编写测试人员。

5.2.2 使用ScalaTest

ScalaTest is a testing tool for Scala (and Java), which we can use to run Chisel testers. To use it, include the library in your *build.sbt* with the following line:

ScalaTest 是一个Scala(和Java)的测试工具， 我们可以用来运行Chisel测试。 为了使用它，把下边这段放在*build.sbt*里边:

```
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % "test"
```

Tests are usually found in src/test/scala and can be run with: 测试通常在*src/test/scala*被找到，使用以下:

```
$ sbt test
```

A minimal test (a testing hello world) to test a Scala Integer addition:
一个最小测试（测试hello world）来测试Scala整型

```scala
import org.scalatest._

class ExampleSpec extends FlatSpec with Matchers {

  "Integers" should "add" in {
    val i = 2
    val j = 3
    i + j should be (5)
  }
}
```

Although Chisel testing is more heavyweight than unit testing of Scala programs, we can wrap a Chisel test into a ScalaTest class. For the *Tester* shown before this is:
尽管Chisel测试比Scala程序的单元测试更重要，我们可以将Chisel测试包装到ScalaTest类中。对于显示的Tester 在此之前:

```scala
class SimpleSpec extends FlatSpec with Matchers {

  "Tester" should "pass" in {
    chisel3.iotesters.Driver(() => new DeviceUnderTest()) { c =>
      new Tester(c)
    } should be (true)
  }
}
```

The main benefit of this exercise is to be able to run all tests with a simple *sbt test* (instead of a running *main*). You can run just a single test with *sbt*, as follows:

这个练习的主要好处是可以通过*sbt test*运行所有的测试（而不是*main*）。你可以只是使用一个简单的*sbt*进行测试，像是如下

```
$ sbt "testOnly SimpleSpec"
```

5.2.3 波形

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 for regression testing.

测试器，像上述描述的那样，对于小型设计，和单元测试，在软件开发是普遍的。作为一个单元测试集合，也为回归测试服务。

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.

但是，对于debug更加复杂的设计，人们会喜欢一次性观察多个信号。一个经典的方式是显示信号的波形图来debug数字设计。在波形图里，信号随着时间显示。

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). For the following example we import following classes:

Chisel测试器可以生成包括所有寄存器和io信号的波形图。 以下例子，我们表示了被测试器件的波形图，来自前边的例子（2位的AND函数）。下边的例子，我们引入下边的类:

```scala
import chisel3.iotesters.PeekPokeTester
import chisel3.iotesters.Driver
import org.scalatest._
```

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.

我们开始使用一个简单的测试器，把数值置入输入，并使用 step增加时钟. 我们不会读取任何输出并使用 expect进行比较.

```scala
class WaveformTester(dut: DeviceUnderTest) extends PeekPokeTester(dut) {

  poke(dut.io.a, 0)
  poke(dut.io.b, 0)
  step(1)
  poke(dut.io.a, 1)
  poke(dut.io.b, 0)
  step(1)
  poke(dut.io.a, 0)
  poke(dut.io.b, 1)
  step(1)
  poke(dut.io.a, 1)
  poke(dut.io.b, 1)
  step(1)
}
```

Instead we call Driver.execute with parameters to generate waveform files (.vcd files).
相反，我们使用参数呼叫 Driver.execute，生成波形文件(.vcd 文件)。

```
class WaveformSpec extends FlatSpec with Matchers {
  "Waveform" should "pass" in {
    Driver.execute(Array("--generate-vcd-output", "on"), () => new DeviceUnderTest()
    ) { c =>
      new WaveformTester(c)
    } should be (true)
  }
}
```

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. Per default the generated files are in test_run_dir then the name of the tester appended with a number. 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*.

你可以观察波形文件，使用免费的波形观察器，GTKWave或是ModelSim。打开GTKWave，并选择*File – Open New Window*，选择存放Chisel测试器存放.vcd文件的文件夹。根据默认，生成的文件在test_run_dir，测试器的名字增加了一个数字。在这个文件夹下，你应该能够找到DeviceUnderTest.vcd。你可以从左侧选择信号，并把它们拉拽到主窗口下。如果你想要保存信号的设置，你可以使用*File – Write Save File*，使用*File – Read Save File*以后读取。

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

很明显地去列举所有的可能值到输入端不能变少。于是，我们会使用一些Scala代码驱动DUT。以下测试器列举了所有可能值，到两个2bit的输入信号。

```
class WaveformCounterTester(dut: DeviceUnderTest) extends PeekPokeTester(dut) {

  for (a <- 0 until 4) {
    for (b <- 0 until 4) {
      poke(dut.io.a, a)
      poke(dut.io.b, b)
      step(1)
    }
  }
}
```

We add a ScalaTest spec for this new tester
我们给这个测试增加了ScalaTest的spec。

```
class WaveformCounterSpec extends FlatSpec with Matchers {

  "WaveformCounter" should "pass" in {
    Driver.execute(Array("--generate-vcd-output", "on"), () => new DeviceUnderTest()
    ) { c =>
```

```
      new WaveformCounterTester(c)
    } should be (true)
  }
}
```

并且使用如下执行

```
sbt "testOnly WaveformCounterSpec"
```

5.2.4 printf Debugging

Another form of debugging is the so-called “printf debugging”. This form 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.

另一种形式的debug是所谓的“printf debugging”。这种方式是来自把C语言的printf声明用来在执行程序的过程中，打印感兴趣的变量。printf debugging也可以在测试Chisel电路的时候有效。打印过程在时钟上升沿。 printf可以在模块定义中被插入到任何位置，像是如下DUTS的printf debugging版本。

```
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 following output, verifying that the AND function is correct:
当使用计数器为基础的测试器进行测试，它循环了所有可能的值，我们得到以下输出，证明了“与”功能是正确的。

```
Circuit state created
[info] [0.001] SEED 1579707298694
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
test DeviceUnderTestPrintf Success: 0 tests passed in 21 cycles
  taking 0.036380 seconds
[info] [0.024] RAN 16 CYCLES PASSED
```

Chisel printf supports C and Scala style formatting.
Chisel的printf支持C and Scala 风格的格式。

5.3 练习

For this exercise, we will revisit the blinking LED from chisel-examples and explore Chisel testing.
对于这个测试，我们需要重温一下闪烁LED从chisel-examples 并探索Chisel测试。

5.3.1 一个最小项目

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.
首先，我们先看一下什么是做小Chisel项目。探索文件你好世界。例子。这个代码*Hello.scala*是一个简单硬件源代码。它包含了(*class Hello*)的硬件描述，和一个*App*用来生成verilog代码。

Each file starts with the import of Chisel and related packages:
每个文件以引用chisel和相关包裹开始：

```
import chisel3._
```

Then follows the hardware description, as shown in Listing *hello*. 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 *HelloWorld* object and pass it to the Chisel driver *execute* function. The first argument is an array of Strings, where build options can be set (e.g., the output folder). The following code will generate the Verilog file *Hello.v*.
以下硬件描述，像是 *hello*出现的。为了生成一个Verilog描述，我们需要一个应用。Scala对象*extends App* 是一个应用，直接在应用开始的地方隐式生成主函数。这个的唯一举动是生成一个新的*HelloWorld*对象，并把它传入Chisel驱动*execute*函数。这个的第一个参数是一个字符串的数组，

```
object Hello extends App {
  chisel3.Driver.execute(Array[String](), () => new Hello())
}
```

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.

并使用编辑器去探索生成的*Hello.v*。生成的Verilog代码可能不是很可读，但是我们可以找到一些细节。这个文件以*Hello*模块开头， 这个和我们的Chisel模块同名。我们可以检查我们的LED端口为*output io_led*。引脚名字是Chisel的名字前置*io_*。 除了我们的LED引脚，这个模块也包含了*clock*和*reset*输入信号。 这两个信号被chisel自动添加。

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.

更多地，我们可以检查我们的两个寄存器*cntReg*和*blkReg*。我们也可以发现reset和这些寄存器的更新， 在模块定义的后边。注意到，Chisel生成了一个同步复位。

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

为了让*sbt* 可以抓取正确的scala编译器和chisel library，我们需要*build.sbt*:

```
scalaVersion := "2.11.7"
resolvers ++= Seq(
  Resolver.sonatypeRepo("snapshots"),
  Resolver.sonatypeRepo("releases")
)
libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.2.2"
libraryDependencies += "edu.berkeley.cs" %% "chisel-iotesters" % "1.3.2"
```

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). Change the *build.sbt* configuration to use the latest Chisel version by changing the library dependency to

注意到在这个例子中，我们有一个具体的Chisel版本数字，防止每次运行的时候检查最新版本（如果不联网的话会失败， 例如，我们在飞机上） 。 更改*build.sbt*设置去使用最新Chisel版本，通过更改library的依赖

```
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?

并且使用*sbt*重新运行build。如何一个最新版本的chisel，并且自动下载呢?

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:

为了方便，这个项目也包含了一个*Makefile*。它包含了*sbt*的命令，所以我们不需要记得自动更新的方式， 并且可以自动生成verilog代码:

```
make
```

Besides a *README* file, the example project also contains project files for different FPGA board. 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, there 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.

除了*README*文件，这个工程的例子也包含了对于不同FPGA的项目文件。例如，在quartus/altde2-115 你可以发现两个项目文件去定义针对于DE2-115板子的Quartus项目。主要定义（源文件，器件，针脚定义）可以在 hello.qsf 发现文本文件。探索文件并发现哪个针脚连接哪些信号。如果你需要采用一个不同开发板的项目，这里需要做出一些改变。如果你安装了Quartus，打开哪个项目，使用*Play*compiler项目，然后在FPGA上进行设置。

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

注意到*Hello World*是一个最小Chisel项目。更多实际的项目在源文件上组织成一个包裹，并包含测试器。下个测试将会探索这样的项目。

5.3.2 一个测试练习

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. 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 *except()*.

在上一章的练习中，您通过一些输入扩展了LED闪烁的示例 来构建一个"与"门和一个多路复用器，并在FPGA中运行该硬件。现在，我们将使用此示例，并使用Chisel测试仪测试功能 自动化测试并独立于FPGA。使用上一章中的设计并添加Chisel测试器以测试功能。尝试枚举所有可能的输入，并使用except测试输出。

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).

在Chisel中进行测试可以加快设计的调试速度。但是，将您的设计综合到FPGA并运行测试始终是一个好主意。 FPGA。在那里，您可以检查设计的大小（通常是 （在LUT和触发器中），以及您在最大时钟频率下的设计性能。作为参考，教科书式流水RISC处理器可能消耗约3000 4位LUT，可以在低成本FPGA（Intel Cyclone或 Xilinx Spartan）。

Chapter 6

组成部分

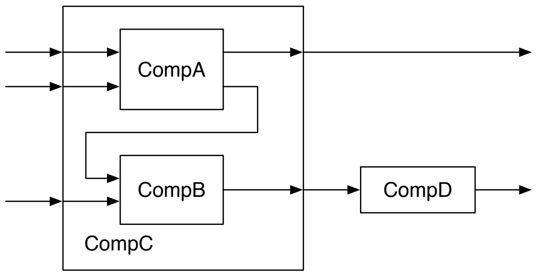

Figure 6.1: A design consisting of a hierarchy of components.

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, usually 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.

一个大的数字设计是一系列组件部分构建而成的，经常是以层级的方式。 每个组件部分都有一个输入输出的接口，经常被称作端口。 这些和IC中的输入输出引脚类似。组成部分被输入和输出连接。 组件可能含有下属组件用来构建层级。最外层的组件，连接到芯片的物理引脚，称为上层组件。

Figure 6.1 shows an example design. Component C has three input ports and two output ports. The component itself is assembled out of two subcomponents: B and C, which are connected to the inputs and outputs of C. One output of A is connected to an input of B. Component D is at the same hierarchy level as component C and connected to it.

6.1表明了一个示例设计。 C组件由两个输入端口和一个输出端口。 组件本身分为两个下属组件：A和B，连接到输入和C的输出。 A的一个输出连接到B的输入。 组件D和组件C有着相同层级，并且互相连接。

In this chapter, we will explain how components are described in Chisel and provide several examples of standard components. Those standard components serve two purposes: (1) they provide examples of Chisel code and (2) they provide a library of components ready to be reused in your design.

在本章节我们会解释组件在Chisel如何被描述，并提供一些标准组件的例子。 这些标准组件充当两个作用：1.他们提供Chisel代码的例子。 2. 他们提供组件库，为你的设计重新使用。

6.1 chisel的组成部分是模块

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()*.

硬件组件在Chisel里称为module。每个module拓展了*Module*类，并包含了一个界面的*io*域。界面通过*Bundle*进行定义，被包裹进*IO()*。

The *Bundle* contains fields to represent input and output ports of the module. The direction is given by wrapping a field into either a call to *Input()* or *Output()*. The direction is from the view of the component itself.

*Bundle*包含了域去表示模块的输入和输出端口。方向通过呼叫*Input()*或是*Output()*来决定方向。 方向是从组成部分本身来说的。

The following code shows the definition of the two example components A and B from Figure 6.1:

以下代码表明了组成部分A和B从Figure 6.1:

```
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))
  })
```

Component A has two inputs, named *a* and *b*, and two outputs, named *x* and *y*. For the ports of component B we chose the names *in1*, *in2*, and *out*.
组成部分有两个输入，命名为*a*和*b*，和两个输出，命名为*x*和*y*。 对于组成部分B，我们选取名称*in1*，*in2*，和*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".

所有的端口使用位宽为8的(*UInt*)非符号整型，作为这个例子的代码，是关于连接组成部分和搭建层级， 我们不编写这个部分的内部。这个部分的编写作为常见形式"X的函数"。

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*.

因为我们对于这些例子部分没有函数，我们使用原本端口名称。对于一个真正的设计，我们使用描述性的端口名称， 例如*data*，*valid*或是*ready*。

Component C has three input and two output ports. It is built out of components A and B. We show how A and B are connected to the ports of C and also the connection between an output port of A and an input port of B:

组成部分C有三个输入和两个输出端口。它来自组成部分A和B。我们表明A和B如何连接到C的端口， 以及A作为输出和和B作为输入如何连接。

```
class CompC extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_x = Output(UInt(8.W))
    val out_y = 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.in_a
  compA.io.b := io.in_b
  io.out_x := compA.io.x
  // connect B
  compB.io.in1 := compA.io.y
  compB.io.in2 := io.in_c
  io.out_y := compB.io.out
}
```

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

组成部分使用*new*进行创建，例如，*new CompA()*，需要被包裹进*Module()*。引用那个模块作为本地变量存入，在这个例子*val compA = Module(new CompA())*。

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

有了这个引用，我们可以通过IO，访问模块的*io*域和单独IO域下的*Bundle*。

The simplest component in our design has just an input port, named *in*, and an output port named *out*.

我们设计中最简单的部分只是一个输入端口，命名为*in*和输出端口*out*。

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

The final missing piece of our example design is the top-level component, which itself is assembled out of components C and D:

最后我们的例子缺少的一部分是顶层部分，它本身由C和D组成。

```
class TopLevel extends Module {
  val io = IO(new Bundle {
    val in_a = Input(UInt(8.W))
    val in_b = Input(UInt(8.W))
    val in_c = Input(UInt(8.W))
    val out_m = Output(UInt(8.W))
    val out_n = Output(UInt(8.W))
  })

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

  // connect C
  c.io.in_a := io.in_a
  c.io.in_b := io.in_b
  c.io.in_c := io.in_c
  io.out_m := c.io.out_x
  // connect D
  d.io.in := c.io.out_y
  io.out_n := d.io.out
}
```

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,

良好的组件设计类似于以下功能或方法的良好设计： 软件设计。主要问题之一是我们应该投入多少功能 组件以及组件应多大。这两个极端很小 组件，例如加法器，以及庞大的组件，例如完整的微处理器，

Beginners in hardware design often start with tiny components. The problem is that digital design books use tiny components to show the principles. But the sizes of the examples (in those books, and also in this book) is small to fit into a page and to not distract by too many details.

硬件设计的初学者通常从微型组件开始。 问题在于数字设计书使用微小的组件来显示原理。 但是示例的大小（在这些书以及本书中）都很小 以适合页面并且不会分散太多细节。

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

组件的界面有点冗长（包括类型，名称，方向， IO建设）。根据经验，我建议组件的核心是 功能，应至少与组件的接口一样长。

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

对于细微的部件（例如计数器），Chisel提供了更轻巧的功能 将它们描述为返回硬件的函数的方式。

6.2 一个运算逻辑单元

One of the central components for circuits that compute, e.g., a microprocessor, is an arithmetic-logic unit, or ALU for short. Figure 6.2 shows the symbol of an ALU.

我们的一个核心计算的电路部分，例如，微处理器，是arithmetic-logic unit， 或是简单来说是ALU。Figure 6.2表示了ALU的符号。

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. 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 ALU.

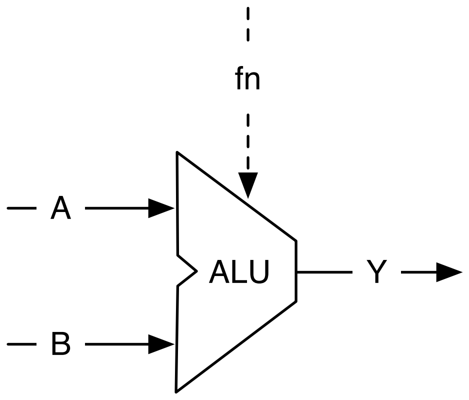

Figure 6.2: An arithmetic logic unit, or ALU for short.

ALU有两个数据输入，图中标明为*A*和*B*，一个函数的输入*fn*， 一个的输出labeled Y。ALU操作在*A*和*B*，并且提供结果给输出。 *fn*选取了*A*和*B*进行操作。 这个操作经常是一些算法，例如加法和减法，并且一些逻辑性函数例如，和，或，异或。那是为什么被称为ALU。

The function input *fn* selects the operation. 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.

函数输入*fn*选取操作。ALU经常是一个组合逻辑电路，没有任何的状态元素。 一个ALU可能有一些额外的输出对于信号结果的属性，例如0或是正负符号。

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

以下的代码表明了一个具有16位输入和输出的ALU，支持加法，减法，或，和与操作，通过二位的*fn*信号。

```
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:

在这个例子，我们使用new Chisel的结构，*switch/is*用于表示选取我们的ALU的输出表。 为了使用这个函数，我们需要引入其它chisel包裹:

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

6.3 整体连接

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 leave fields for the connection. If a name is missing, it is not connected.

为了连接具有多个IO端口的组件，Chisel提供了 批量连接运算符<>。 该运算符在双方连接束。 Chisel使用离开字段的名称进行连接。 如果缺少名称，则表示未连接。

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

作为一个例子，让我们假定，我们搭建一个流水线的处理器。抓取阶段有一个如下的接口：

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

The next stage is the decode stage.

下一阶段是译码阶段。

```
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.

我们的简单处理器的最后阶段是执行。

```
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.

为了连接三个阶段，我们需要两个<>操作符。 我们也可以使用它连接父模块下的子模块端口。

```
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
```

6.4 使用函数的轻量级组成部分

Modules are the general way to structure your hardware description. However, there is some boilerplate code when declaring a module and when instantiating and connecting it.

模块是构造硬件描述的通用方法。 但是，有一些样板代码在声明模块以及实例化连接的时候可以去使用。

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:

构造硬件的轻型方法是使用功能。 Scala函数可以采用Chisel（和Scala）参数并返回生成的硬件。 作为一个简单的示例，我们生成一个加法器：

```
def adder (x: UInt, y: UInt) = {
x + y
}
```

We can then create two adders by simply calling the function *adder*.

我们可以接下来创造两个加法器，通过呼叫函数*adder*.

```
//- start components_fn_use
val x = adder(a, b)
// another adder
val y = adder(c, d)
```

Note that this is a *hardware generator*. You are not executing any add operation during elaboration, but create two adders (hardware instances). The adder is an artificial example to keep it simple. Chisel has already an adder generation function, like +*(that: UInt)*.

注意到这个是*hardware generator*。你不只是在生成过程中执行任何相加操作，而是建造两个加法器（硬件模块）。 加法器是一个人工的例子，去使它变得简单。chisel本身就有加法器生成函数，像是+*(that: UInt)*。

Functions as lightweight hardware generators can also contain state (including a register). Following example returns a one clock cycle delay element (a register). If a function has just a single statement, we can write it in one line and omit the curly braces ().

函数作为轻量级硬件生成器，也可以包含状态，包括几个寄存器。 以下例子返回了一个时钟周期的元素（一个寄存器）。 如果一个函数只是一个单一的声明，我们可以把它用一行写入，并忽略括号()。

```
def delay(x: UInt) = RegNext(x)
```

By calling the function with the function itself as parameter, this generated a two clock cycle delay.
通过呼叫这个函数，传入这个函数本身，这产生两个周期的延迟。

```
val delOut = delay(delay(delIn))
```

Again, this is a too short example to be useful, as *RegNext()* already is that function creating the register for the delay.
再次，这个例子太短，以致于不够有用, 像 *RegNext()* 已经是那个函数，创造一个延时。

Functions can be declared as part of a *Module*. However, functions that shall be used in different modules are better placed into a Scala object that collects utility functions.

可以将函数声明为Module的一部分。但是，函数应 最好将用于不同模块的模块放置在一个收集实用程序的Scala对象中 。

Chapter 7

组合搭建模块

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.

在本章节，我们探索各种组合电路，基本的建造模块，我们可以用来构建更加复杂的系统·。基本上所有的组合电路都能通过布尔算式被编写。但是，更常见的情况，一个表格的描述是更为有效的。我们让综合工具抓取并缩小布尔算式。两个基本电路，最好是通过表格方式描述，一个是译码器，另一个是编码器。

7.1 组合电路

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:

在我们描述一些标准的组合建造模块，我们会探索组合电路如何在Chisel种被表示。最简单的是布尔算式，这个可以通过给名字分配：

```
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:

布尔逻辑表达式通过给一个名字*e*一个Scala值。这个表达式可以在其它表达式重新使用

```
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,

这样的表达式被认为是固定的。通过=给*e*命名会导致Scala编译器错误：reassignment to val。尝试Chisel操作符:=, 像是如下，

```
e = c & b
```

results in a runtime exception: *Cannot reassign to read-only*.
导致runtime例外：不能重新分配给只读。

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

Chisel也支持描述组合电路通过条件性更新。这样的电路先被声明为一个Wire，然后使用条件操作，例如when，然后描述电路的逻辑。接下来的代码表明了类型UInt的Wire，并把它分配给默认值0。when部分接受一个Chisel的Bool，并重新分配给w，如果条件true.B。

```
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 condition *cond* the select signal. Keep in mind that we describe hardware circuits and not a software program with conditional execution.

电路的逻辑是一个复用器，这里两个输入是0和3，然后条件cond是选择信号。记住我们描述硬件电路，而不是使用软件程序中的条件执行。

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:

Chisel条件建造when也有一个else的类型，它是otherwise。在某些条件下通过给一个值分配，我们可以忽略默认值：

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

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

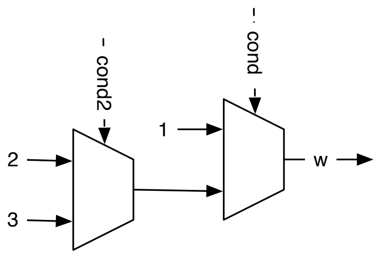

Figure 7.1: A chain of multiplexers.

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

Chisel也支持一系列条件（if/elseif/else系列），使用.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 construct a chain of multiplexers. Figure 7.1 shows this chain of multiplexers. That chain introduce a priority, i.e., when cond is true, the other conditions are not evaluated.

一连串when, .elsewhen, 和 .otherwise组成了一连串复用器。图片 7.1表示了这串复用器。那个串引入了先后顺序，例如 cond是真的话，其它的条件不被评价。

Note the '.' in .elsewhen that is needed to chain methods in Scala. Those .elsewhen branches can be arbitrary 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.

注意到'.'在.elsewhen是被需要的一个串方法，在scala。那些 .elsewhen 分支可以是任意长的。但是，如果条件串取决于一个单一信号，最好使用switch声明，在下一章介绍解码器电路中被介绍。

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.

对于更复杂的电路，可能赋值给Wire一个默认值是更实际的。一个默认赋值可以直接使用WireDefault作为线声明。

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

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

One might question why using when, .elsewhen, and otherwise when Scala has if, else if, and else? Those 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.

一个可能的问题是，既然Scala有if， else if, 和 else，为什么使用when, .elsewhen, 和 otherwise? 那些声明是用来条件执行Scala代码的，而不是生成Chisel（复用器）硬件的。那些Scala条件在我们写电路生成器的时候在chisel中有具体作用的，是在抓取参数条件生成不同硬件实例中被使用。

a	b
00	0001
01	0010
10	0100
11	1000

Table 7.1: Truth table for a 2 to 4 decoder.

7.2 解码器

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).

一个 解码器 把一个 n 位的二进制数转换为一个 m位的二进制数, 这里 $m \leq 2^n$. 输出是独热码（这里每一位代表计一）。

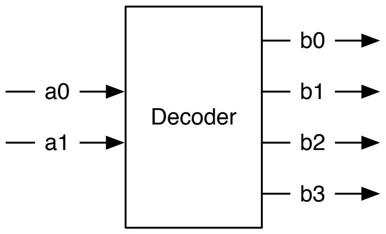

Figure 7.2: A 2-bit to 4-bit decoder.

Figure 7.2 shows a 2-bit to 4-bit decoder. We can describe the function of the decoder with a truth table, such as Table 7.2.

图片7.2表示了一个2位到4位的解码器。我们可以描述这个函数，使用真值表，像是 7.2。

A Chisel switch statement describes the logic as a truth table. The switch statement is not part of the core Chisel language. Therefore, we need to include the elements of the package chisel.util.

一个Chiselswitch的声明描述了这个逻辑，作为一个真值表。 switch声明不是Chisel语言的核心部分。 于是，我们需要引用chisel.util软件包的元素。

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

The following code introduces the switch statement of Chisel to describe a decoder:
以下代码引入了Chisel的switch声明，描述一个解码器。

```
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 0 to result. This assignment will never be active and therefore optimized away by the backend 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.
以上switch声明列举了所有sel信号的可能值，并把解码的值赋给result信号。注意到，即使我们列举所有的可能值，Chisel仍然需要赋一个默认值，就像我们把0赋值给result。这个赋值不会被激活，直到有后端工具去优化。这个是故意的，避免组合电路的非完全赋值，（Chisel里，Wire）会导致一个不理想的锁存器，在硬件描述语言，像是VHDL或是Verilog。Chisel不会允许非完全赋值。

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

在我们先前的例子，我们使用信号的非赋号整型。可能一个更清晰的编码电路使用二进制表示方式:

```
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 «.

一个表格提供了更为刻度的译码器函数表示方式，但是这也变得拖沓。当检查表格的时候，我们看到了一个常见的结构：1被向左移动了sel个单位。于是，我们可以表达一个译码器通过Chisel移位操作«。

```
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, as a Mux is available in the core library. Decoders can also be used for address decoding, and then the outputs are used as select signals, e.g., different IO devices connected to a microprocessor.

译码器作为一个由复用器组成的部分，输出结果和使能，选用一个与门选择信号作为输入。但是，在Chisel我们不需要搭建复用器，尽管库的Mux是可用的。译码器可以用来翻译地址，然后输出是被选择的信号，例如，不同的IO器件连接到一个处理器。

7.3 编码器

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

编码器把独热码输入信号转换为二进制编码形式作为输出。编码器是译码器的反向操作。

Figure 7.3 shows a 4-bit one-hot input to a 2-bit binary output encoder, and Table 7.3 shows the truth table of the encode function. However, an encoder works only as expected when the input signal is

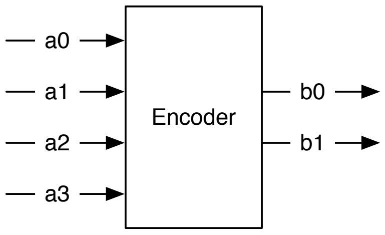

Figure 7.3: 4-2编码器。

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

Table 7.2: 4-2编码器的真值表。

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.

图片7.3表明了一个4位独热输入到一个2位二进制输出的编码器，并且表格7.3表明了编码函数的真值表。但是，一个编码器只在输入信号是独热的情况下正常工作。对于其它的输入，输出是未定义的。所以我们不能描述未定义输出的函数，我们使用默认赋值来表示未定义的输入情况。

The following Chisel code assigns a default value of 00 and then uses the switch statement for the legal input values.

以下Chisel代码定义默认输入00，并且使用switch声明用作合法输入值。

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

7.4 练习

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 4 switches or buttons to the input of your circuit and the output to the 7-segment display.

描述一个组合电路，转换一个4位输入显示，到7段显示。 你可以定义十进制数字的显示码，这个是7段显示的初始用法，或是，定义其它位的编码方式，用来显示所有十六进制的16个

可能值。 当你有一个7段显示的FPGA板子，连接4个开关或是按钮，到你的电路，和7段显示的输出。

Chapter 8

时序建造模块

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.

时序电路的输出取决于输入和前一个值。因为我们感兴趣的是同步设计（时钟设计），我们说时序电路，我们说的是同步时序电路。[2]。为了搭建时序电路，我们需要储存状态的元素，所以我们称为寄存器。

8.1 寄存器

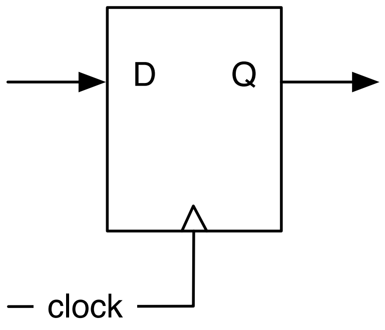

Figure 8.1: 一个D型寄存器

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. Alternatively, in other words: the register updates its output with the value of the input on the rising edge of the clock.

最基本的搭建时序电路的元素是寄存器。寄存器是D触发器的集合。D触发器在时钟上升沿抓取它的输入，并把它作为输出储存起来。或者用另一句话，寄存器在时钟上升沿更新其输出，变为输入值。

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

[2]我们也可以搭建时许电路，使用非同步逻辑和反馈，但是这个是一个特殊的话题并且不再Chisel中表示

Figure 8.1 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. As this global clock signal is connected to all registers in a synchronous circuit, it is usually not drawn in our schematics. The little triangle on the bottom of the box symbolizes the clock input and tells us that this is a register. We omit the clock signal in the following schematics.

图8.1表明寄存器的草图符号。它包含输入D和输出Q。每个寄存器也包含了clock作为输入。在一个同步时序电路，作为全局时钟信号连接到所有寄存器，它一般在草图不画。一个小的三角形在盒子的代表时钟输入，告诉我们这是一个寄存器。我们在以下草图忽略时钟信号。

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.

忽略总时钟信号，也在Chisel中被反映出来，这里没有规定寄存器的输入时钟信号。

In Chisel a register with input d and output q is defined with:

在Chisel一个寄存器d输入和q输出被被定义为：

```
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 arbitrary complex types made out of a combination of vectors and bundles.

这里注明一下，我们不需要给寄存器连接时钟，这个在Chisel内部间接完成。 寄存器的输入和输出可以是任何来自于vector和bundle组合的复杂类型。

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

寄存器也可以被定义和两步使用：

```
val regDelay = Reg(UInt (4.W))
regDelay := 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 circuits and sequential circuits, it is common practice to have the marker *Reg* as part of to the name. Also, note that names in Scala (and therefore also in Chisel) are usually in CamelCase. Variable names start with lowercase and classes start with upper case.

首先我们定义了寄存器并给它一个名字，其次我们连接了信号*delayIn*给寄存器的输入。 注意寄存器的名字是*Reg*开始的。 为了简单区分组合电路和舒徐电路的元素，一般常用的方式是在开头添加*Reg*的名字作为开头。 并且记得scala的名字（同样适用于chisel）经常以CamelCase的形式。 变量名称以小写开头，接下来的类别以首字母大写的形式。

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

一个寄存器也可以被复位初始化。*reset*信号是像*clock*信号一样，在Chisel是隐性的。我们提供了复位值，例如，零， 作为一个参数传给寄存器构造器*RegInit*。寄存器的输入是通过Chisel赋值声明连入的。

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

The default implementation of reset in Chisel is a synchronous reset.[3] For a synchronous reset no change is needed on a D flip-flop, just a multiplexer needs to be added to the input that selects between the initialization value under reset and the data values.

Chisel的默认重置是同步复位。[4] 对于一个同步复位，不需要在D触发器需要改变，只是一个复用器需要被添加到输入，用于选取复位数值和数据数值。

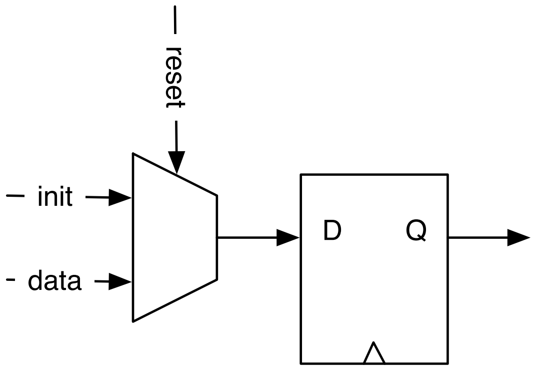

Figure 8.2: A D flip-flop based register with a synchronous reset.

Figure 8.2 shows the schematics of a register with a synchronous reset where the reset drives the multiplexer. However, as synchronous reset is used quite often modern FPGAs flip-flops contain a synchronous reset (and set) input to not wast LUT resources for the multiplexer.

图片 8.2表明一个具有同步复位的寄存器草图，这里同步复位驱动复用器。 但是，一个同步复位经常被使用，是因为现代FPGA包含一个重置（和设置）是用来不浪费复用器的LUT资源。

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.

时序电路不断地改变数值。于是，他们的行为可以被通过图像表示不同时间下的信号。这样的图像被称为波形图或是 时序图。

Figure 8.3 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, before a reset, the register is undefined. In the second clock cycle reset is asserted high, and on the rising edge of this clock cycle (labeled B) the register captures the initial value of 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 (labeled C). From then on reset stays 0, as it should be, and the register output follows the input signal with one clock cycle delay.

[3]Support for asynchronous reset is currently under development

[4]目前非同步复位

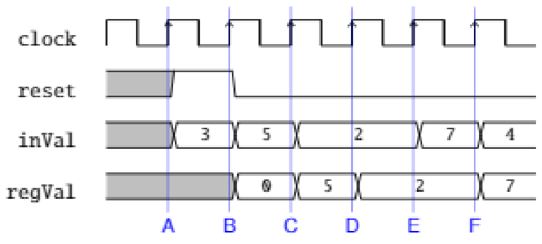

Figure 8.3: A waveform diagram for a register with a reset.

Figure 8.3 表明了一个具有重置和输入数据的寄存器波形图。 时间按照从左到右的顺序。在图片的上边，我们看到有时钟驱动我们的电路。 在第一个时钟周期，在重置之前，寄存器是未定义的。在第二个时钟周期插入高电位，以及在时钟周期的上升沿（标为B）， 寄存器捕捉了0的初始值。inVal的输入是被忽略的。在下一个周期，reset是0， 并且inVal在下一个上升沿（标为C）被捕捉到。从此reset呆在0，结果应该是， 寄存器的输出跟随输入信号，在一个周期延迟。

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

波形是以图形方式指定电路行为的出色工具。 特别是在许多操作并行发生的更复杂的电路中 并且数据通过电路流水线传输，时序图很方便。 chisel测试器还可以在测试过程中产生可显示的波形 带有波形查看器并用于debug，其功能应为 最好将用于不同模块的模块放置在一个收集实用程序的Scala对象中 职能。

A typical design pattern is a register with an enable signal. Only 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.

一个典型的寄存器类型是一个寄存器和一个使能信号。只有当使能信号是true（高）的时候，寄存器抓取输入； 否则的话，它保持原有的值。 使能信号可以通过类似于同步复位，在输入端前置复用器的方式被补充。 其中一个复用器的输入是寄存器输出的反馈。

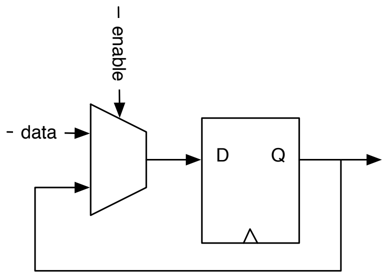

Figure 8.4: A D flip-flop based register with an enable signal.

Figure 8.4 shows the schematics of a register with enable. As this is also a common design pattern, modern FPGA flip-flops contain a dedicated enable input, and no additional resources are needed.

图片 8.4表明一个具有使能信号的草图。 这个是一个常见的设计类型，当代的FPGA触发器包含一个

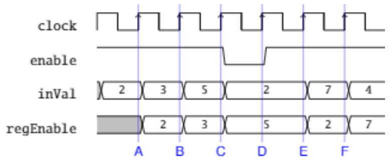

Figure 8.5: A waveform diagram for a register with an enable signal.

Figure 8.5 shows an example waveform for a register with enable. Most of the time, enable it 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 (5) at rising edge D.

图片 8.5表明了一个波形图例子，用于含有使能信号的寄存器。 大多数时间，使其变为高位(true)，寄存器遵守输入的一个周期延迟。只是在第四个周期enable 是低位的，于是寄存器在在上升沿D（5）保持他的值。

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

一个具有使能的寄存器可以通过数行chisel码和一个条件更新被描述:

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

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

A register with enable can also be reset:

具有使能的寄存器也可以被重置:

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

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

A register can also be part of an expression. Following circuit detects the rising edge of a signal by comparing its current value with the one from the last clock cycle.

一个寄存器也可以是表达式的一部分。以下电路检测了信号的上升沿，通过比较其当前值和上周期的值。

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

Now that we have explored all basic uses of a register, we put those registers to good use and build more interesting sequential circuits.

现在，我们已经探索了寄存器的所有基本用途，我们将这些寄存器放入 善用并构建更有趣的时序电路。

8.2 计数器

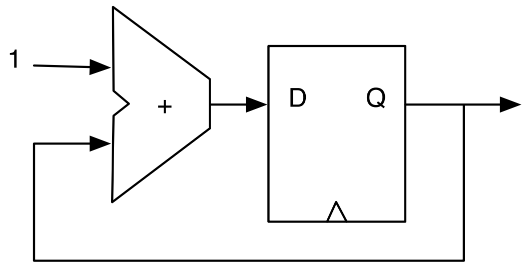

Figure 8.6: An adder and a register result in counter.

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 8.6 shows such a free-running counter.

计数器是最基本的时序电路之一。以最简单的形式，计数器就是寄存器 输出连接到加法器，而加法器的输出连接到输入 寄存器。图 8.6显示了这样一个自由运行的计数器。

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.

具有4位寄存器的自由运行计数器从0到15计数，然后回绕 再次设为0。计数器也应重置为已知值。

```
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.
当我们想要去计数事件的时候，我们使用条件去增加计数器读数。

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

8.2.1 向上和向下计数

To count up to a value and then restart with *0*, we need to compare the counter value with a maximum constant, e.g., with a *when* conditional statement.

要计算一个值，然后使用*0*重新启动，我们需要比较 具有最大常数的计数器值，例如带有*when* 有条件的声明。

```
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:
我们也可以使用复用器作为一个计数器的增加读数：

```
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 (reset the counter register) with the maximum value and reset the counter to that value when reaching 0.
如果我们有倒数的想法，我们开始（重置计数器寄存器） 设置为最大值，并在达到0时将计数器重置为该值。

```
val cntReg = RegInit(N)

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

As we are writing and using more counters, we can define a function with a parameter to generate a counter for us.
当我们写入和使用更多的计数器的时候，我们可以定义一个具有参数的函数去为我们生成计数器。

```
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)
//- end

// and one more for testing
val testCounter = genCounter(n-1)
io.tick := testCounter === (n-1).U
io.cnt := testCounter
}

class NerdCounter(n: Int) extends Counter(n) {

val N = n

//- start nerd_counter
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
}

```

The last statement of the function *genCounter* is the return value of the function, in this example, the counting register *cntReg*.
最后的函数声明*genCounter*返回了函数数值，在这个例子，是计数器寄存器*cntReg*。

Note, that in all the examples our counter had values between *0* and *N*, including *N*. If we want 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

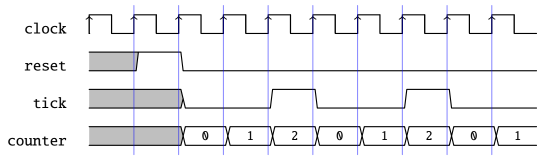

Figure 8.7: A waveform diagram for the generation of a slow frequency tick.

error.

注意，所有的例子里，我们的计数器在*0*和*N*，包括*N*都有数值。 如果我们想要数10个时钟周期，我们需要把*N*设为9。 把*N*设为10可能会成为一个经典的例子， 产生off-by-one错误

8.2.2 使用计数器产生时序

Besides counting events, counters are often used to generate timing. 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 a LED at some frequency as we have shown in the Chisel "Hello World" example.

除了计数时间，计数经常被使用去生成延迟。 一个同步电路使用一个时钟和特定的频率。 电路在这些时钟变化中生成结果。 在数字电路中没有时间的概念，除了计算时钟周期。 如果我们知道时钟频率，我们可以生成电路用于生成时间性事件，类似在一些特定频率闪烁LED， 像是我们前边知道的Chisel"Hello World"例子。

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 cycles, where $n = f_{clock}/f_{tick}$ and the tick is precisely one clock cycle long. This tick is *not* used as a derived clock, but as an enable signal for registers in the circuit that shall logically operate at frequency f_{tick}. Figure 8.7 shows an example of a tick generated every 3 clock cycles.

一个常见的生成单个具有频率f_{tick}的我们电路中需要的*ticks*的方式。 每隔n个周期发生一次tick，这里$n = f_{clock}/f_{tick}$, tick是被设为一个时钟的长度。 这个tick是、emph不用做推演的时钟的，而是作为一个作为电路的使能信号，逻辑上在f_{tick}下操作。 8.7表明一个每三个周期产生tick的例子。

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.

在下边的这个例子，我们介绍了一个计数器，从*0*数到最大值*N - 1*。 当达到最大值的时候，*tick*在一个单周期内变为*true*，计数器被重置为*0*。 当我们从*0*数到*N - 1*, 我们每*N*时钟周期生成一个逻辑tick。

```
val tickCounterReg = RegInit(0.U(4.W))
val tick = tickCounterReg === (N-1).U

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

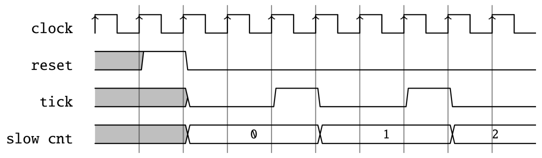

Figure 8.8: Using the slow frequency tick.

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.
每隔n时钟生成一个逻辑上的tick可以被使用区增加其它电路中的时钟部分，通过这个慢的逻辑电路。 在下边的代码，我们使用另一个计数器，每过n时钟周期增加*1*单位。

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

Figure 8.8 shows the waveform of the tick and the slow counter that increments every tick (n clock cycles).
图片 8.8 表示了波形的变动，慢速计数器在每个变动的时候增加一位（n 个时钟周期）。

Examples of the usage of this slower *logical* clock are: blinking an LED, generating the baud rate for a serial bus, generating signals for 7-segment display multiplexing, and subsampling input values for debouncing of buttons and switches.

使用这个更慢的*logical*时钟的场景: 闪烁一个lED，对于一个序列串口胜场波特率， 生成7段显示复用器信号，向下采样输入值用于防止上下抖动。

Although width inference should size the registers, it is better to explicitly specify the width with the type at 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.

尽管位宽推断应该规定寄存器尺寸大小，但是更明确位宽，通过寄存器的类型的定义，或是通过初始值的方式，是更好的。 定义明确的位宽可以防止当重置为*0.U*的时候，产生一个位宽为1的意外。

8.2.3 nerd计数器

Many of us feel like being a nerd, sometimes. For example, we want to design a highly optimized version of our counter/tick generation. A standard counter needs 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 out of 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 a little bit 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 a 0 or 1 bit. The resource requirement is the same for the up and down counter.

我们很多人有时会感觉自己像一个nerd。 例如，我们想要设计一个超级优化的计数器（触发信号的生成）。 一个标准的计数器需要以下资源：一个寄存器，一个加法器（或是减法器），一个比较器。 我们不能再寄存器或是加法器做很多。如果我们向上数，我们需要比较一个目标

数，这是一个字符。 比较器可以从0开始反向生成一个字符，然后一个大的与们。 当我们向下数零的时候，比较器是一个大的或非们，这个从ASIC的角度是更加便宜的。 在FPGA，逻辑来源于查找表，所以在比较1或是0的时候就没什么区别了，此时向上数和向下数对于资源的要求是一样的。

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 bit set to 0. We need to check this bit only to detect that our counter reached -1. Here it is, the counter created by a nerd:

但是，这里仍然有一个小窍门，硬件设计者可以借鉴。 目前向上数或向下数需要与所有的计数位比较。 当你向下数太多会发生什么？ 那种情况，计数变为负数。探测一个负数只是简单比较最高位“a”就好。

```
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
}
```

8.2.4 一个计时器

Another form of timer we can create, 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*.

我们能够创造的另一种计时器的形式，是一个只响一次的计时器。一个只响一次的计时器像是一个厨房计时器：你设定时间，并按开始。当过了一段规定的时间，闹铃响了。这个数字计时器以时钟形式读取时间。它向下数直到零。在零时刻插入*done*。

Figure 8.9 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 de-asserted 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 input of the multiplexer that provides 0.

图片8.9表示了计时器的框图。寄存器可以通过插入load读取din。当load被置零，选择向下计数（通过选择cntReg - 1作为寄存器的输入）。 当计数器到达0，信号done置一，并且通过复用器选择0，计数器停止向下计数。

Listing 8.1 shows the Chisel code for the timer. We use an 8-bit register reg, that is reset to 0. The boolean value done is the result of comparing reg 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. Signal load has priority over the decrement selection. The last line connects the multiplexer, represented by next, to the input of the register reg.

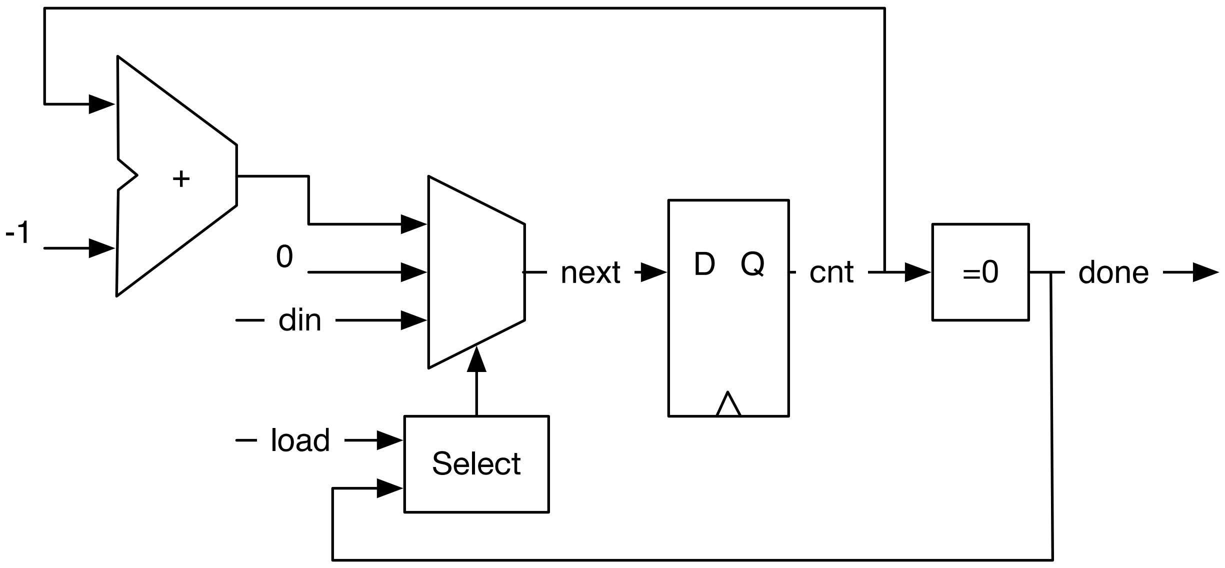

Figure 8.9: A one-shot timer.

代码8.1表示了计时器的chisel代码，我们使用一个8位寄存器reg，被重置为0。布尔值done是reg和零比较的结果。对于输入复用器，我们引入线next，默认值为0。when/elsewhen随着选择函数，引入了其它两个输入。信号load比向下数更有优先性。 最后一连接了复用器，被next表示，到寄存器reg的输入。

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

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

Listing 8.1: A one-shot timer

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。 .

如果我们想要更加详细的代码，我们可以直接赋值给复用器的值到寄存器reg，而不是使用中间线next。

8.2.5 脉冲宽度调制

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

脉冲宽度调制 (PWM)是一个常量周期的信号，调制的时间是周期*high*的时间。

Figure 8.10 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 also called the duty cycle. In the first two periods the duty cycle is

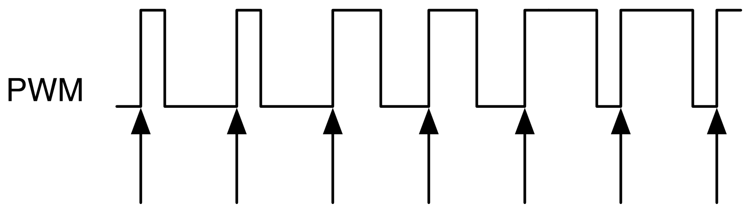

Figure 8.10: 脉冲宽度调制

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

图片 8.10 表明了一个PWM信号。 箭头指向周期开始的地方。高电平信号的占比被称为占空比（duty cycle）。 在前两个周期，占空比是 25 %, 在后边两个周期是 50 %, 并且在最后两个周期是 75 % 。脉冲在25 %和75 %之间。

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.

增加低通滤波器到一个PWM成为一个简单的数字模拟转换器。 低通滤波器可以简单地像是一个电阻和一个电容。

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

以下代码例子会生成一个每10个时钟周期出现3个时钟周期高电平的波形。

```scala
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 (pulswidth) 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, in our case the wire connected to the compare function.

我们使用一个函数，用作PWM的生成器，提供一个可重复使用，轻量级的部分。这个函数有两个参数：一个Scala整型，通过调节时钟周期数量（nrCycles）和一个Chisel线（din）提供占空比（脉冲宽度）用作PWM的输出。我们使用一个复用器，在这个例子里，来表示计数器。最后一行的函数，比较计数器的值，和输入值din用来返回PWM信号。最后Chisel函数的表示，是返回值，在我们例子里，是连接到比较函数的线。

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.[5] Chisel also has a function signedBitLength to provide the number of bits for a signed representation of a number.

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

我们使用函数unsignedBitLength(n)去规定计数器用来表示计数cntReg的上限（并包括）n。[6]Chisel也有一个函数signedBitLength提供位数的数量，提供用来表示一个符号位的数的位宽。

Another application is to use PWM to dim an LED. In that case the eye serves as 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.

另一个使用PWM的应用是亮起一个LED。在那种情况下，我们的眼睛像是一个低通滤波器。我们拓展以上例子去驱动PWM，通过一个三角函数。这个例子是LED可以连续改变强度。

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

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

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 switch of the direction.

我们使用两个寄存器用于调制：（1）modulationReg用于向上计数和向下计数和 (2)upReg作为一个旗帜，去确定我们是否应该向上数或向下数。我们根据我们的时钟输入向上数频率（我们的例子里是100 MHz），导致一个0.5 Hz的信号。when/.elsewhen/.otherwise的表达式处理向上数或向下数，和方向的改变。

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 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.

随着我们的PWM数到1000个频率，产生一个1 kHz的信号，我们需要把这个调制信号除以1000。因为一个真实的除法在硬件上是非常昂贵的，我们简单地向右移10位，等于除以$2^{10} = 1024$。因为我们已经定义了PWM电路作为一个函数，我们可以简单地实体化那个电路，调用函数。电线sig表示这个调制的PWM信号。

[6]用来表示非符号整型 n 的二进制宽度是 $\lfloor log_2(n) \rfloor + 1$。

8.3 移位寄存器

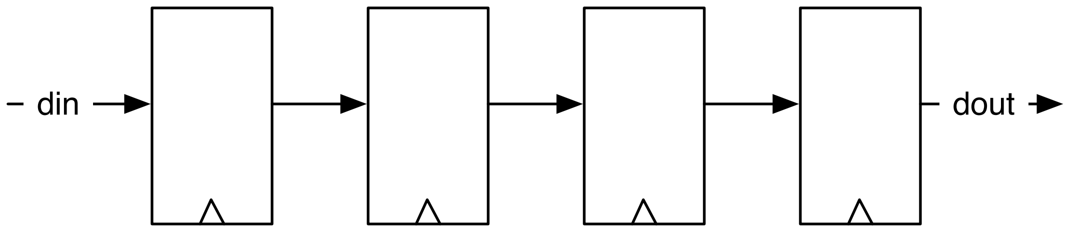

Figure 8.11: 一个4级移位寄存器

A shift register is a collection of flip-flops connected in a sequence. Each output of a register (flip-flop) is connected to the input of the next register. Figure 8.11 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-tap delay from din to dout.

一个移位寄存器是一个顺序连接的触发器的集合。每个寄存器（触发器）的输出接到下一个寄存器的输入。 图片8.11 表示一个4级的移位寄存器。 电路移动数据从左边到右边，在每个时钟的触发。在这个简单的形式，电路形成了4级的延迟，从din到dout。

The Chisel code for this simple shift register does: (1) create a 4-bit register shiftReg, (2) concatenate 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.

用于这个简单移位寄存器做了：(1) 创造一个4位寄存器shiftReg， (2) 合并移位寄存器的低3位到输入din用于下一个寄存器的输入，并且(3)使用了最高位的寄存器用于输出dout。

```
val shiftReg = Reg(UInt(4.W))
shiftReg := Cat(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 13.2 shows a serial port that uses shift registers for the receive and send functions.

移位寄存器经常被用来转换串口数据到平行数据，或是从平行数据转换到串口数据。13.2表明了一个串口数据，使用移位寄存器，用作接收数据和发送的功能。

8.3.1 使用并行输出的移位寄存器

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 8.12 shows a 4-bit shift register, where each flip-flop output is connected to one output bit. After 4 clock cycles this circuit converts a 4-bit serial data word to a 4-bit parallel data word that is 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.

一个串口并行设置的移位寄存器把一个串口输入变成并行输出的字。这个可以在串口(UART)用来接收功能。图片8.12表示了一个4位的移位寄存器，这里每个触发器输出接入一个输出位。经过4个周期，这个电路把一个4位串口字转换位一个4位平行数据字，在q有效。在这

个例子，我们假定，0位(最低位)先被发送，当我们想要读取整个字的时候，最低位到达最后一级。

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

以下Chisel代码，我们使用0初始化移位寄存器outReg。然后我们从最高位开始移位，也就是右移。并行结果，q，是寄存器outReg的读出值。

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

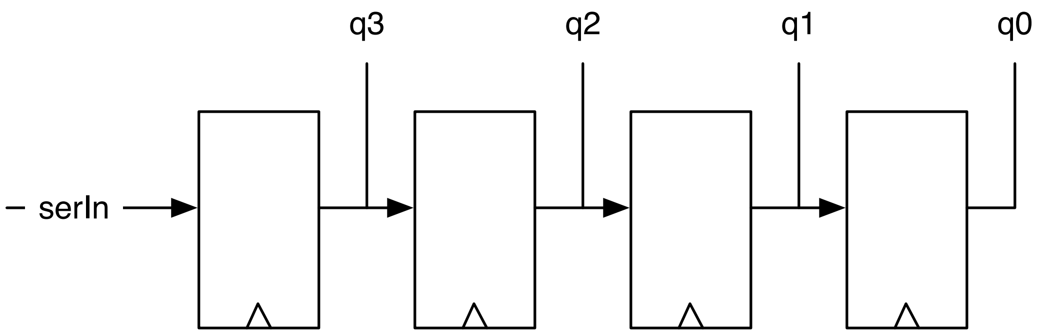

Figure 8.12: 一个具有并行输出的4级移位寄存器

Figure 8.12 shows a 4-bit shift register with a parallel output function.
图片 8.12 表示了一个具有并行输出功能的4位移位寄存器。

8.3.2 并行读取的移位寄存器

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.

一个并行输入串行输出设置的移位寄存器，把一个并行输入的字（字节）转换位串行输出流。这个可以用在传输功能的串口(UART)。

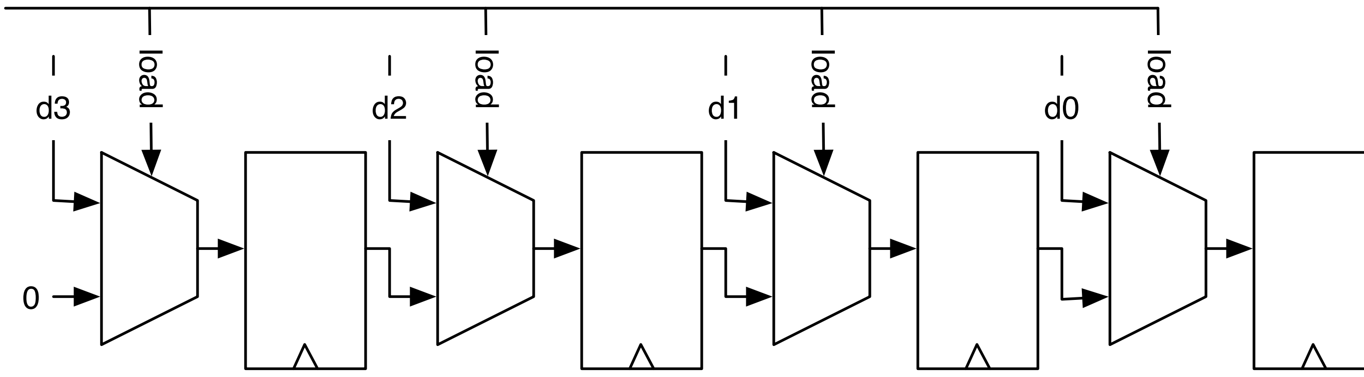

Figure 8.13: 一个具有并行读取的4级移位寄存器

Figure 8.13 shows a 4-bit shift register with a parallel load function. The Chisel description of that function is relatively straight forward:

图片8.13表示了一个4位移位寄存器，具有并行读取的功能。这个功能的Chisel描述是相对直接的：

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

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

注意到我们现在移位到右边，在最高位填入零。

8.4 存储器

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 where the direction can be switched.

存储器可以通过一系列的寄存器搭建，在chisel，一个*Vec*的*Reg*。但是，这个在硬件上是昂贵的，更大的存储器是通过SRAM搭建的。对于一个ASIC, 存储器编译器构建出存储器。FPGA自带片上存储单元，也称为模块化RAM。这些片上存储单元可以组合成为更大的存储器。FPGA上的存储器一般有一个读端和一个写端，或者可以切换方向的两个端口。

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.

FPGA（或是ASIC）经常支持同步存储器。同步存储器在输入上具有寄存器(读和写地址，写数据，写使能)。那意味着读数据在设置地址后的一个周期的是可用的。

Figure 8.14 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 write enable (*wrEna*). Note that for all inputs, there is a register within the memory showing the synchronous behavior.

8.14图片表明了这样一个同步存储器的草图。这个存储器是双端口的，具有一个读出和写入端口。这个读出端口有一个单一输入，读出地址(*rdAddr*)和一个输出数据(*rdData*)。写入端口有三个输入：地址(*wrAddr*)，写入的数据(*wrData*)，和写入使能(*wrEna*)。注意到对于所有的输入，存储器的寄存器表明了同步的行为。

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

为了支持片上存储，chisel提供了存储器构建器*SyncReadMem*。**??**表明了*Memory*的构成去支持1KB的存储器，具有字节位宽的输入和输出数据，和一个写入使能。

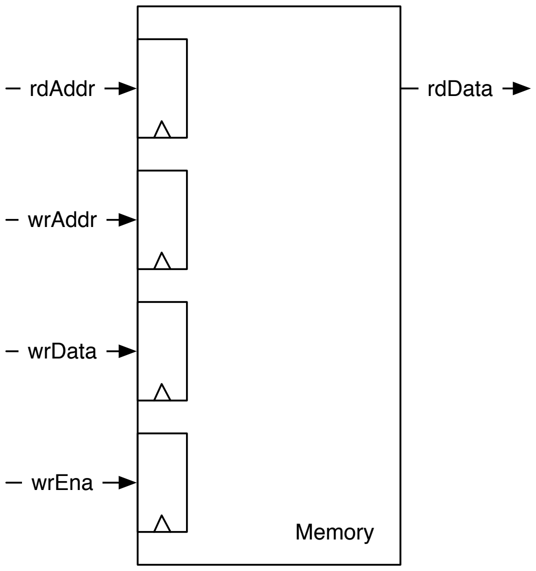

Figure 8.14: A synchronous memory.

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

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

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

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

An interesting question is which value is returned from a read when in the same clock cycle a new value is written 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.

一个有趣的问题是，在同一时钟中，从读取返回哪个值 循环将一个新值写入与读出相同的地址。 我们对内存的read-during-write行为感兴趣。 有三种可能：新写入的值，旧值或未定义 （这可能是旧值中的一些位与一些新写入的数据的混合）。 FPGA中可用的可能性取决于FPGA类型和 有时可以指定。

Chisel documents that the read data is undefined.

Chisel档案中读取数据是未定义的。

If we want to read out the newly written value, we can build a forwarding circuit that detects that

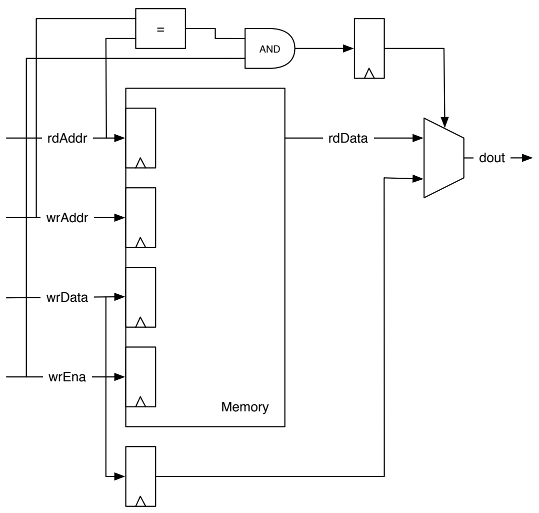

Figure 8.15: A synchronous memory with forwarding for a defined read-during-write behavior.

the addresses are equal and *forwards* the write data. Figure 8.15 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 with a register.

如果我们想要读出新写入的值，我们可以搭建一个前递电路，能够检查出相同地址情况下*forwards*出写入数值。 8.15表明具有前递电路的存储器。读出和写入地址会相互比较，并受到写入使能控制，去选择 写入数据的前递路径，或是存储器读取数据。这个写入数据是受到一个周期的寄存器延迟。

Listing **??** shows the Chisel code for a synchronous memory including the forwarding circuit. We need to store the write data into a register (*wrDataReg*) to be available in the next clock cycle the synchronous memory also has a one clock cycle latency. 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.

列表 **??** 表明一个具有同步存储器，包含前馈电路的Chisel代码。 我们需要存储写入数据到一个寄存器(*wrDataReg*)，使之在下个周期变得有效，同步存储器也有一个周期的延迟。 我们比较了两个写入地址(*wrAddr* 和 *rdAddr*)并且去检查，看*wrEna*是否为真，在前馈的条件下。 那个条件也受到一个周期的延迟。 一个复用器在前馈写入数据和读出数据做出选择。

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

  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)
}
```

Chisel also provides *Mem*, which represents 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*.

Chisel也提供了*Mem*，代表了一个具有同步写入和同步读出的存储器。但是这个存储器一般不在FPGA可以使用， 综合工具会把它变为触发器。于是，我们推荐使用*SyncReadMem*。

8.5 练习

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 a second counter that can generate a single-cycle *tick* signal every 500 milliseconds. Use that signal as enable signal for the 4-bit counter.

使用来自上一章的7段编码器并添加一个4位计数器作为输入，去切换显示，从*0*到*F*。 当你直接连接这个计数器到FPGA的时钟，你会看到所有的16个数字相互交叠(所有的7段显示一起亮)。 于是，你需要调慢计数。创建一个秒计数器，每个周期生成一个*tick*，每过500毫秒。 使用那个信号作为使能信号，用于4位计数器。

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 sinus function with the use of a lookup table that you can generate with a few lines of Scala code (see Section 12.3). Drive a LED on an FPGA board with that modulated PWM function. What frequency shall your PWM signal be? What frequency is the driver running?

创建一个PWM波形图，使用生成器函数，并给函数设置一个阈值（三角或是正弦函数）。一个三角函数可以通过向上数和向下数创造。一个具有查找表的弦性函数，你可以使用几行Scala代码生成(见 12.3部分)。 使用FPGA板驱动一个LED，使用那个调制过的PWM函数。你的PWM信号频率应该是什么？驱动器驱动的频率是多少?

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:

数字设计经常被在纸上描画。不是所有的细节都需要被表示。我们使用框图，像是书上的图。这个是一个重要的技能，熟练地在电路和Chisel的原理图表述之间翻译。描绘下边电路的原理图草图:

```
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 a little bit more complex circuit, containing a register:
这里有一个更加难的复杂电路，包括一个寄存器:

```
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}
}
```

Chapter 9

输入处理

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.

来自外部世界的信号，到我们的同步电路，经常对于时钟来讲不是同步的；他们是异步的。一个输出信号可能来自一个源，没有一个干净的转换，从0到1，或是从1到0。一个例子是一个震荡按钮或是开关。 输入信号可能是有噪声的，有毛刺，可能造成我们的同步电路一次转变。这章描述了处理此类输入情况的电路。

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.

最后两个情况，防抖动开关，和滤波噪声，也可以被外部模拟部件解决。但是，这个是更（消耗性）有效的，去解决这些问题，在数字领域。

9.1 异步输入

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.

没有同步到系统时钟的输入信号被称为异步信号。这些信号可能违反了触发器输入的建立和保持时间。这个违反可能会导致触发器的多稳态。这个多稳态可能会导致输出值在0和1，或者会导致震荡。但是，在一些时间过后，触发器会稳定在0或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: 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.

我们不能防止多稳态，但是我们可以容纳这个问题。一个经典的解法是在输入使用两个触发器。这个的假设是：当这个触发器在多稳态的时候，他会在一个周期后收敛到稳定，所以第二个触发器的建立和保持时间都不会被违反。

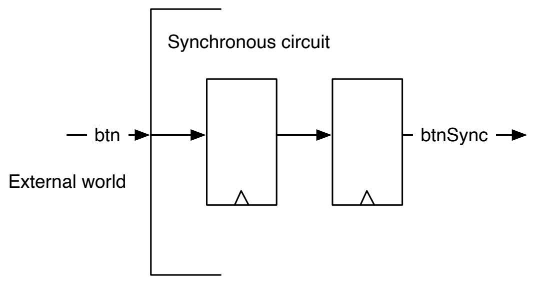

Figure 9.1: 同步输入

Figure 9.1 shows the border between the synchronous circuit and the external world. The input synchronizer consists of two flip-flops. The Chisel code for the input synchronizer is a one-liner that instantiates two registers.

图片9.1表示了同步电路和外部世界的边界。输入同步包括两个触发器。用于输入的同步器的Chisel代码是一行代码实现的两个寄存器。

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

All asynchronous external signals need an input synchronizer.[1] We also need to synchronize an external reset signal. 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. Concrete the de-assertion of the reset need to be synchronous to the clock.

所有的同步外部信号需要一个输入同步。[2]我们也需要同步一个外部的重置信号。这个重置信号应该先通过两个触发器，再被其它触发器使用，作为重置信号。具体的重置信号置零，需要同步到时钟。

9.2 防抖动

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.

开关和按钮可能需要在开和关的变换需要一些时间。 在变换的过程中，开关可能会在二者之间震荡。 如果我们使用这样一个信号，没有更多的处理的话，我们会检测到比我们想要的更多转换事件。一个解决方式是使用时间去滤波掉这个震荡。假设最大震荡时间是t_{bounce}，我们会采样这个输入信号，使用$T > t_{bounce}$。我们只会使用采样信号用于向下传播。

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 a 1. However,

[1]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 an asynchronous SRAM to a synchronous circuit, e.g., by a microprocessor.

[2]例外是，当输入信号依赖于同步输出信号，我们知道最大传播延迟。一个经典的例子是，异步SRAM到一个同步电路的接口，例如，一个微处理器。

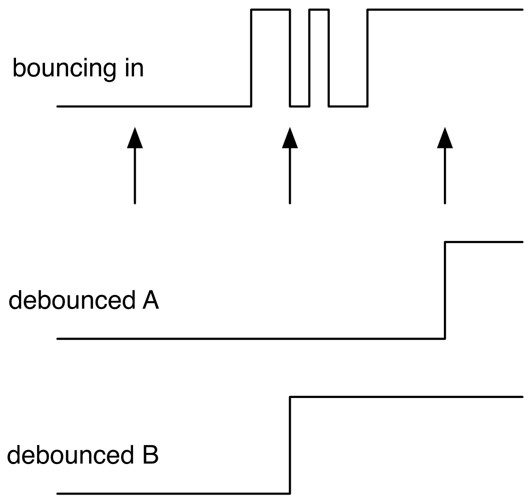

Figure 9.2: 解除一个输入信号的震荡。

this does not matter as it then belongs either to the still 0 samples or to the already 1 samples. The critical point is that we have only one transition from 0 to 1.

当使用这个长周期采样，我们直到从0到1的转变，只会有一个样本进入震荡区。这个前边的样本会安全读入0，在震荡区样本以后，会安全读入1.震荡区的样本会在0或1。但是，当它属于还是0的位置或是已经到了1的位置，这并不影响，这个的关键点在于我们只有一次从0到1的转变。

Figure 9.2 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 needs to 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 debounce A and debounce B. Both have a single transition from 0 to 1. The only difference between these two outcomes is that the transition in version B is one sample period later. However, this is usually a non-issue.

图片9.2表示了用于防抖动的采样过程。最上边的信号表示了震荡输入，下边的箭头表示了采样点。 这些采样点的距离应该比最大震荡时间更长。第一个采样，安全地采样了一个0，最后图中的采样，采样了1。中间的采样，落入了震荡时间。它可能是0或是1。这两个可能的结果被debounce A和debounce B表示。这两个有一个从0到1的转变。唯一的区别是，这两个输出是B版本的转变是一个周期以后。但是，这个一般不是一个问题。

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 8.2.2.

这个用来防抖动的Chisel代码是比同步器更加改进的。我们使用计数器生成一个采样书记兼，表达一个周期的tick信号，就像我们在8.2.2部分做的。

```
val FAC = 100000000/100
```

```
val btnDebReg = Reg(Bool())

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 counter, and the tick signal is true when the counter has reached the maximum value. In that case, the when condition is true and (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 it is the output from the input synchronizer shown in the previous section.

首先，我们需要决定采样频率。以上的例子是假设一个100 MHz的时钟，和导致的采样频率为100 Hz（假设震荡时间小于10 ms）。最大计数器值是FAC，除法的因子。我们定义一个寄存器btnDebReg用于防抖动的信号，只是没有重置的值。寄存器cntReg作为计数器，tick信号为真，当计数器到达了最大值。在那种情况下，when的条件是true，(1)计数器的重置是0，(2)防抖动寄存器存储输入采样。在我们的例子，输入信号被称为btnSync，

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

防抖动电路在同步电路后边一级。首先，我们需要去同步化异步信号，然后我们可以更多在数字领域处理。

9.3 输入信号滤波

Sometimes our input signal may be noisy, maybe containing spikes that we might sample unintentionally with the input synchronizer and debouncing unit.

有的时候我们的输入信号是有噪声的，可能包含我们不想要使用输入同步和防抖动单元采样的毛刺。

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.

一个滤掉这些毛刺的方式是使用大多数投票电路。在最简单的情况，我们做三次采样，并执行大多数投票。The 大多数函数,是和中值函数相关的，也就是大多数的数值。 在我们的例子里，我们使用防抖动采样，我们采用了大多数投票，在采样信号。大多数投票确保了信号可以比采样周期的稳定时间更长。

Figure 9.3 shows the circuit of the majority voter. 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 feed into the majority voting circuit. The majority voting function filters any signal change shorter than the sample period.

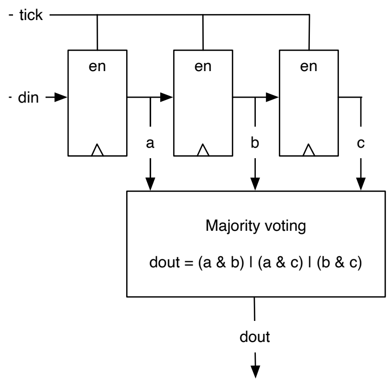

Figure 9.3: 在采样信号上做大多数投票。

图片9.3表示了大多数投票器的电路。它包括3位移位寄存器，使能端被用于防抖动的tick信号触发。输出的三个寄存器反馈到大多数投票电路。大多数投票函数滤掉任何比采样信号变化短的信号。

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

以下Chisel代码表示了3位移位寄存器，被tick信号使能触发，并且投票函数，导致了信号btnClean。

Note, that a majority voting is very seldom needed.

注意，这个大多数投票是很少需要的。

```
val shiftReg = RegInit(0.U(3.W))
when (tick) {
// shift left and input in LSB
    shiftReg := Cat(shiftReg(1, 0), btnDebReg)
}
// Majority voiting
    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 to enable the counter to increment.

为了使用我们经过精心处理的输入信号的输出，我们首先使用RegNext延迟元素检测上升沿，然后把这个信号和当前值btnClean比较，去使能计数器去增加。

```
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
}
```

9.4 使用函数合并输入处理

To summarize the input processing, we show some more Chisel code. As the presented circuits might be tiny, but reusable building blocks, we encapsulate them in functions. Section **??** showed 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. If useful, those functions can be elevated to some utility class object.

为了总结输入处理，我们显示了更多的Chisel代码。 我们呈现的电路可能很小，但是这些是可复用的搭建模块，我们把这些包裹成函数。**??**部分表示我们如何使用轻量级Chisel函数把小的搭建模块抽象化，而不是整个模块。 那些Chisel函数创造了硬件实例，例如，sync函数创造了两个触发器相互连接到输入。 函数返回了第二个触发器的输出。 如果有用的话，这些函数可以用来当作工具类的对象。

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

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

def tickGen(fac: Int) = {
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 := Cat(reg(1, 0), v)
}
(reg(2) & reg(1)) | (reg(2) & reg(0)) | (reg(1) & reg(0))
}

val btnSync = sync(btn)

val tick = tickGen(fac)
val btnDeb = Reg(Bool())
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 9.1: Summarizing input processing with functions

9.5 练习

Build a counter that is incremented by an input button. Display the counter value in binary with the LEDs on an FPGA board. Build 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.

搭建一个计数器，通过一个输入按钮增加计数值。二进制显示这个计数值，使用具有LED显示的FPGA板。搭建整个输入处理链，使用：（1）输入同步器，（2）防抖动电路，（3）一个大多数投票电路去压制噪声，和（4）边沿检测电路去引发计数器数值的增加。

As there is no guarantee that modern button will always bounce, you can simulate the bouncing and the spikes by pressing the button manually in a 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 by pressing several times in fast succession before settling to a stable press. Test your circuit without and with the debouncing circuit sampling at 1 Hz.

因为这里不能保证现代的按钮会经常震荡，你可以模拟这个震荡和毛刺，通过手动按压按钮，以快速和低频的方式。选取，例如，一秒一次的频率，即，如果输入时钟在100 MHz，除以100M。在稳定按压前，通过快速按压数次，模仿震荡按钮。测试你的电路，拿掉防抖动电路，和附带震荡电路，采样频率在1 Hz。

Chapter 10

有限状态机

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.

有限状态机（FSM）是一个数字电路中的基本的搭建模块。一个FSM可以被描述为一个*states*和（有限制的）状态条件 *state transitions*。一个FSM有一个初始状态，这是在reset中被确定的。FSMs也被称为同步时序电路。

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.

FSM的实现包括三个部分：(1) 一个具有现在状态的寄存器, (2) 组合逻辑根据目前状态和输入计算下个状态, (3) 组合逻辑能够计算FSM的输出。

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, e.g., 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.

原则上，每个数字电路包含一个寄存器或是其它存储器元件去储存状态，可以被称为一个FSM。但是，着可能不实用，例如，把你的电脑描述为一个FSM。下个章节，我们描述如何通过组合小的FSM进行通信FSM，搭建更大的系统。

10.1 基本有限状态机

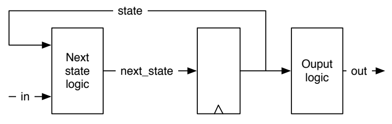

Figure 10.1: A finite state machine (Moore type).

Figure 10.1 shows the schematics of an FSM. The register contains the current *state*. The next state logic computes the next state value (*next_state*) from the current *state* and the input (*in*). On the next clock tick, *state* becomes *next_state*. The output logic computes the output (*out*). As the output depends on the current state only, this state machine is called a Moore machine.

图片 10.1 表明FSM的草图. 寄存器包含目前的*state*. 通过*state*和输入(*in*)用来计算下个状态的状态逻辑(*next_state*)。 在下个时钟, *state* 变为 *next_state*. 输出逻辑计算输出 (*out*). 因为输出只取决于目前状态，这样的状态机被称为 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.

state diagram 使用图形描述FSM的行为。 在状态图中，单独的状态被画成标注状态名字的圆圈。 状态转移被描述为状态之间的箭头。 当这个转移被采取的时候，改转移条件会在箭头上被标出。

Figure 10.2 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* it the only output of this FSM. We add the output to the *red* state. The alarm can be reset with a *clear* signal.

10.2图片表明了一个简单FSM的例子图表。 这个FSM有三个状态：*green*, *orange*, and *red*，表明闹铃的等级。FSM以绿色等级为开始，当*bad event*发生的时候，闹铃等级切换到*orange*。在下一个坏的时刻，闹铃等级被切换到*red*。 在那种情况下，我们想要响铃；*ring bell*是唯一的FSM的输出。 我们在*red*状态下增加输出。 然后响铃可以被重置到*clear*信号。

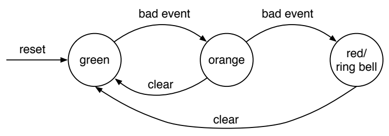

Figure 10.2: The state diagram of an alarm FSM.

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 10.1 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 would 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.

尽管状态图看起来很舒服，一个FSM可以被快速掌握，那么状态表可以更快地写下来。10.1图标表明我们的闹铃FSM的状态表。 我们列出了当前状态，输入值，和接下来的状态，和当前状态的输出值。原则上，我们需要 列出所有可能状态下的所有输入值。这个表格可以有3*4=12列。我们把这个表格进行简化， 当*bad event*发生的时候，输入*clear*是不管的。那意味着*bad event*对于*clear* 有优先级。输出列有一些重复。如果我们有一个更大的FSM或是更多输出，我们可以把列表分为两个，一个用于计算下个状态逻辑， 一个用于计算输出逻辑。

Finally, after all the design of our warning level FSM, we shall code it in Chisel. Listing **??** 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 *Enum* and the *switch* control instruction, we need to import *chisel3.util._*.

最后，得到我们的等级警报FSM设计，我们使用chisel写下代码。 **??**表明了闹铃FSM的Chisel代码。 注意到，我们使用Chisel的*Bool*作为输入和输出类型。 为了使用*Enum*和*switch*控制指令，

Table 10.1: 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	0	red	1
red	0	1	green	1

我们需要引入 *chisel3.util._*。

```scala
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
  val green :: orange :: red :: Nil = Enum(3)

  // 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
}
```

The complete Chisel code for this simple FSM fits into one page. Let us step through the individual parts. The FSM has two input and a single output signal, captured in a Chisel *Bundle*:

这个简单的FSM的Chisel完整代码可以放得下一页纸。 让我们从独立的部分开始。 这个FSM有两个输入和一个输出信号，在Chisel*Bundle*打包：

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

Quite some work has been spent in optimal state encoding. Two common options are binary or one-hot encoding. However, we leave those low-level decisions to the synthesize tool and aim for readable code.[1] Therefore, we use an enumeration type with symbolic names for the states:
尽管曾经在最佳状态编码上进行过一些研究。两种常用选择是二进制或是读热码。但是，我们把这些底层决定交给综合工具， 并实现可读代码。[2] 于是，我们使用了*Enum*类和符号化名字用于表示状态：

```
val green :: orange :: red :: Nil = Enum(3)
```

The individual state values are described as a list where the individual elements are concatenated with the *::* operator; *Nil* represents the end of the list. An *Enum* instance is *assigned* to the list of states. The register holding the state is defined with the *green* state as the reset value:
单独的状态变量被描述为一个序列，这里单独的元素通过*::*进行合并；*Nil*代表序列的结尾。*Enum*实例是被*assigned*赋值到一列状态。 含有状态的寄存器通过*green*状态被定义到重置值。

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

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 for our state register:
FSM的核心在下一个状态逻辑的定义。我们使用Chisel的switch声明，在状态寄存器，去抵消所有状态。 通过每个*is*分支，我们编写了下个状态逻辑，依赖于输入，通过给我们的状态寄存器赋新的数值。

```
switch (stateReg) {
  is (green) {
    when(io.badEvent) {
```

[1]In the current version of Chisel the *Enum* 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.

[2]在目前的版本，*Enum*代表二进制编码的状态。 如果我们想要另一种编码，例如，独热码，我们为状态名称定义chisel常量。

```
        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*.
最后重要的，我们编写了*ringing bell*的输出为真，当状态是*red*。

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

Note that we did *not* introduce a *next_state* 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 within a combinational block.

记着我们没有引入*next_state*作为寄存器输入， 尽管这个在verilog或是vhdl是常见的。 verilog和VHDL的寄存器被表述为一个特定的形式，并且不能赋值或是重复赋值，在一个组合框架。 于是，额外的信号，在组合框架被计算出来，并连接给寄存器的输入。 在Chisel，寄存器是一个基础类型，可以通过组合框架免费使用。

10.2 使用Mealy FSM产生快速输出

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 *earliest* in the next clock cycle. If we want to observe an immediate change, we need a combinational 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:

在一个Moore FSM，输出值只取决于当前状态。 那意味着输入改变可以被看成导致最早的下个周期的输出改变。 如果我们想要观察一个间接的改变，我们需要一个组合路径，从输入到输出。 让我们想一个最小的例子，在边沿检测电路。 我们可能以前看过这一行Chisel代码:

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

Figure 10.3 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.

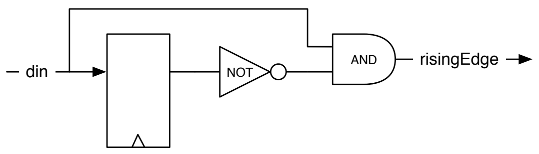

Figure 10.3: A rising edge detector (Mealy type FSM).

图片 10.3表明了上升边沿检测器的草图。 当这个周期输入为1的时候，经过一个周期，输出变为1，并且上个周期的输入是0. 这个状态寄存器只是一个简单的D触发器，下个状态只是输入值。我们也可以把这个看成一个时钟延迟的元素。 输出逻辑比较当前输入和当前状态。

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.

当输出也取决于输入，例如，有一个组合逻辑的路径在FSM的输入和输出之间，这被称为Mealy machine。

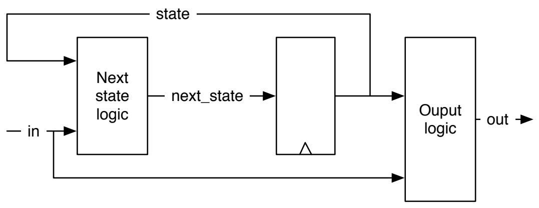

Figure 10.4: A Mealy type finite state machine.

Figure 10.4 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 (*next_state*) from the current *state* and the input (*in*). On the next clock tick, *state* becomes *next_state*. The output logic computes the output (*out*) from the current state *and* the input to the FSM.

Figure 10.4 表明了Mealy类型的FSM的草图。 类似于Moore FSM，寄存器包含了当前*state*，和来自当前*state*和输入(*in*)的用于计算(*next_state*)的状态逻辑。 在下一个时钟，*state*变为*next_state*。输出逻辑从当前状态的(*out*)计算输出。

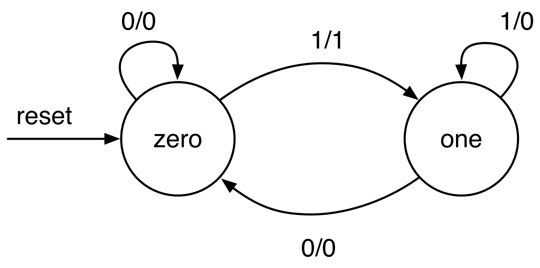

Figure 10.5: The state diagram of the rising edge detector as Mealy FSM.

Figure 10.5 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.

图片 10.5表明Mealy FSM的状态图，用于边沿检测。作为状态寄存器，包含了只是一个D触发器， 只有两个状态是可能的，在这个例子中，是*zero*和*one*。

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 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) puls for each rising edge of the input.

作为Mealy FSM的输出，并不只是依赖于状态，也依赖于输入，我们不能描述输出作为状态圆的一部分。 相反，状态间的转移函数和输入值和输出值是一起被标出的。 注意到我们画本身的状态转移，例如，在状态*0*当输入是*0*，FSM停留在*0*， 输出是0。 FSM的上升沿产生*1*作为输出，只有当从状态*0*转移到*1*。在状态*1*， 代表输入现在是*1*，输出是*0*。我们现在想要一个对于每个输入的上升沿， 产生单一周期的跳变。

```scala
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
  val zero :: one :: Nil = Enum(2)

  // 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 10.1: Rising edge detection with a Mealy FSM

Listing 10.1 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.

10.1表明了使用Mealy机器用于上升沿检测的chisel代码。 像是前边的例子，我们使用chisel类型*Bool*用于单比特的输入和输出。 输出逻辑是下个逻辑的一部分，在从*zero*到*one*的转换过程，输出被设定为*true.B*。 否则的话，默认赋值是按照(*false.B*)。

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 is 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.

如果问到如果一个成熟的FSM是不是一个最好的边沿检测电路的最好方法， 特别地，像我们看到过chisel单行代码解决了同样的问题。 硬件消耗是同样的。两个解决方案需要一个单D触发器用于表示状态。 FSM的组合逻辑可能有点复杂，随着状态改变，需要依赖于现有状态和输入值。 对于这个函数，单行代码是易写和易读的，这个更为重要。 于是，单行代码是更好的解决方式。

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

我们已经使用了这个例子区表明了一个更为聪明的方式去编写Mealy FSM。 FSM应当在更为复杂的三个或是更多状态的电路被使用。

10.3 Moore对比Mealy

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

为了表明Moore和Mealy FSM的区别，我们重做了一遍使用Moore FSM进行边沿检测。

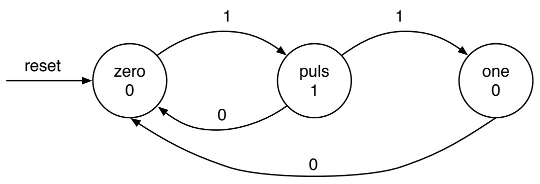

Figure 10.6: The state diagram of the rising edge detector as Moore FSM.

Figure 10.6 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 two states in the Mealy version. The state *puls* is needed to produce the single-cycle puls. The FSM stays in state *puls* 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.

图片 10.6 表明了使用Moore FSM的状态图用于上升沿检测。 第一个事情是去注意到Moore FSM需要三个状态，相比较于Mealy版本需要两个状态。 *puls*状态需要去产生单一周期的脉冲。 FSM处于状态*puls*只是一个周期，然后返回到开始状态*zero*或是状态*one*， 等待输入再次变为0。 我们表明了在状态转移箭头下的输入条件，和在圆圈代表的状态下FSM的输出。

```
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
  val zero :: puls :: one :: Nil = Enum(3)

  // 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 10.2: Rising edge detection with a Moore FSM

10.2 shows the Moore version of the rising edge detection circuit. Is uses double the number of D flip-flops than the Mealy or direct coded version. The resulting next state logic is therefore also larger than the Mealy or direct coded version.

10.2表明了Moore版本的上升沿检测电路。 它使用双倍的D触发器， 比Mealy或是直接的版本。 转移逻辑于是比Mealy或直接版本的更大。

Figure 10.7 shows the waveform of a Mealy and a Moore version of the rising edge 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, where the Mealy output is usually less than a clock cycle.

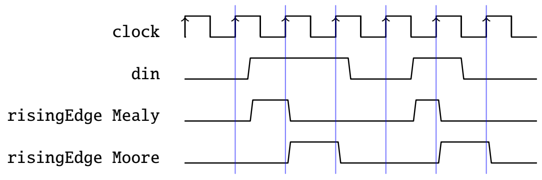

Figure 10.7: Mealy and a Moore FSM waveform for rising edge detection.

图片 10.7表明了Mealy和Moore版本的上升沿检测FSM。你可以看到Mealy输出紧紧跟随上升边沿， 当Moore输出在时钟tick后上升。 我们也可以看到Moore输出是一个时钟周期的宽度，而Mealy输出一般小于一个时钟周期。

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 trouble 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.

从以上的例子，你可以发现Mealy FSM是更好的， 因为它需要更少的状态（和逻辑），并比Moore FSM反应更快。 但是，在一个Mealy机器种，组合电路可能在更大规模的决定上制造麻烦。 首先，具有一串通信FSM（看下一章），这个组合通路可以很长。 其次，如果FSM的通信形成一个圆圈，会造成组合回馈，在同步设计造成错误。 由于具有状态寄存器的组合通路的一个切割形成Moore FSM， 上述的问题在Moore FSM通信不存在。

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 cycle is of utmost importance. Small circuits such as the rising edge detection, which are practically Mealy machines, are fine as well.

总结来说，Moore FSM在状态机间通信的组合是更好的，他们比Mealy FSM更加稳定。 使用Mealy FSM只是当关注在相同周期的反应下更为重要。 小的电路，像是上升沿检测， 像是实际上的Mealy机，也是可以的。

10.4 练习

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.

在这个章节，你看到很多小的FSM例子。现在是时候写一些真实的FSM代码了。 选取一点难的例子，并去补充一个FSM，写一个测试。

A classic example for a FSM is a traffic light controller. 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). To make this example a little bit more interesting, consider a priority road. 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.

一个经典FSM的例子是交通灯控制器(看14.3章)。 一个交通灯控制器需要去确定从红到绿的切换，有一个状态是两条路的中间没有信号灯（红色或是橘色）。 为了使这个例子变得更有趣一些，想象一个主干道。 副路需要有两个车辆检测器（在交叉的两个入口）。 当检查到车辆的时候，在副路切换到绿灯，然后切换回主干道切换回绿灯。

Chapter 11

状态机通信

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 FSMs output is another FSMs input, and the FSM watches the output of the other FSM. When we split a large FSM into simpler ones, this is called factoring FSMs. However, often communicating FSMs are directly designed from the specification, as often a single FSM would be infeasible large.

一个问题是，经常很复杂，去使用单一FSM去描述一个电路。 在那种情况下，这个问题可以被分为两个或是更小更简单的FSM。 这些FSM然后使用信号去通信。 一个FSM的输出是另一个FSM的输入，然后这个FSM观察了另一个FSM的输出。 当我们把一个大型FSM为简单的FSM，这个被称为分解FSM。 但是，经常来说，直接根据要求在FSM之间进行通信， 作为一个简单的FSM会是非常大的。

11.1 一个灯光闪烁器的例子

To discuss communicating FSMs, we use an example from [**?**, Chapter 17], the light flasher. The light flasher has one input start and one output light. The specification of the light flasher is 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.

为了讨论FSM的通信，我们使用十七章的一个例子，灯光闪烁器/ 灯光闪烁器有一个输入start和一个输出light。 灯光闪烁器像是如下:

- 当start 在一个周期内, 闪烁序列开始
- 一个序列闪烁三次。
- 当light变为*on*为6个周期，light变为*off*四个周期，在一次闪烁中。
- 在序列后，FSM变为light *off*，等待下一个开始。

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.

FSM用于直接补充[2]有27个状态: 一个开始值是等待输入，3×6个状态，对于前三个*on*的状态，并且2×4个状态，对于*off*状态。 我们不放出这个简单的灯光闪烁器的代码。

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 11.1 shows the composition of the two FSMs.

这个问题可以通过更优雅地分解大型FSM到两个小的FSM: 主FSM补充了闪烁逻辑，和计时器FSM补充了等待。 图片 11.1表明了两个FSM的组成。

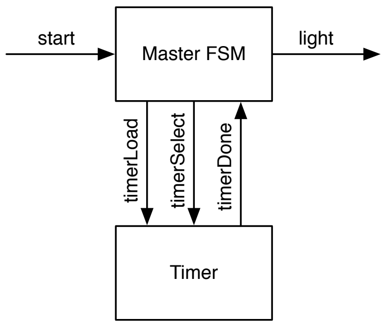

Figure 11.1: The light flasher split into a Master FSM and a Timer FSM.

The timer FSM counts down for 6 or 4 clock cycles to produce the desired timing. The timer specification is as follows:

计时器FSM从6或4向下数产生想要的延迟， 时序的定义如下:

- when *timerLoad* is asserted, the timer loads a value into the down counter, independent of the state;
- *timerSelect* selects between 5 or 3 for the load;
- *timerDone* is asserted when the counter completed the countdown and remains asserted;
- otherwise, the timer counts down.
- 当 *timerLoad*被插入，计时器读入数值到向下计数器， 独立于状态。
- *timerSelect* 选择5或3，用于读取。
- *timerDone* 信号被插入，当计数器完成了向下计数，并保持插入。
- 否则，计时器向下数。

Following code shows the timer FSM of the light flasher:
以下的代码表明了FSM的时序用于亮灯的闪光:

[1]The state diagram is shown in [**?**, p. 376].
[2]The state diagram is shown in [**?**, p. 376].

```
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 11.1: flasher timer

Listing 11.2 shows the master FSM.
11.2 表明了主FSM.

```
val off :: flash1 :: space1 :: flash2 :: space2 :: flash3 :: Nil = Enum(6)
  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())

  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) {
      timerSelect := false.B
      light := true.B
      when (timerDone) { stateReg := space2 }
    }
```

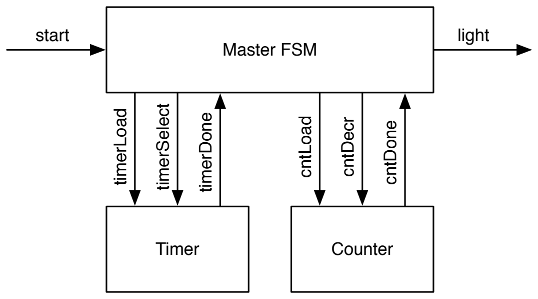

Figure 11.2: The light flasher split into a Master FSM, a Timer FSM, and a Counter FSM.

```
    is (space2) {
      when (timerDone) { stateReg := flash3 }
    }
    is (flash3) {
      timerSelect := false.B
      light := true.B
      when (timerDone) { stateReg := off }
    }
  }
```

Listing 11.2: flasher fsm

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.

具有一个主FSM和一个计数器的解决方案始终在主FSM有代码的冗余。 状态flash1， flash2, 和flash3完成着同样的函数， 还有状态space1和codespace2。 我们可以其它flash装进第二个计数器。 然后主FSM被缩减到三个状态： off, flash, 和 space。

Figure 11.2 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.

图片 11.2 表明了主FSM的设计和两个计数的FSM。 一个FSM用于计数*on*和*off*之间的时钟周期长度； 第二个FSM去计数剩余的闪烁。

Following code shows the down counter FSM:

以下代码表明FSM的向下计数器：

```

val cntReg = RegInit(0.U)
cntDone := cntReg === 0.U

// Down counter FSM
when(cntLoad) { cntReg := 2.U }
when(cntDecr) { cntReg := cntReg - 1.U }
//- end

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
    }
  }

  io.light := light
}
```

Listing 11.3: flasher2 counter

Note, that the counter is loaded with 2 for 3 flashes, as it counts the *remaining* flashes and is decremented in state space when the timer is done. Listing 11.4 shows the master FSM for the double refactored flasher. 注意到，计数器读2的时候，是闪烁三次的，因为它计数剩下的闪烁，当计时器结束的时候减少状态量。 11.4表明主FSM用于双倍闪烁。

```
val off :: flash :: space :: Nil = Enum(3)
  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 11.4: 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.

除了有一个缩减到三个状态的主FSM，我们目前的解决方案也是一个更好的可配置方案。如果我们想要改变*on*或是*off*的闪烁数量间隔， 没有FSM需要被改变。

In this section, we have explored communicating circuits, especially FSM, that only exchange control signals. However, circuits can also exchange data. 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.

在这个部分，我们已经探索了电路间通信，特别是FSM，那个只是交换控制信号。但是，电路也交换数据。 对于调度数据的交换，我们使用握手信号。下个部分描述了对于控制流数据交换的单向ready-valid界面。

11.2 具有数据通路的状态机

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 datapath (FSMD). The state machine controls the datapath, and the datapath performs the computation. The FSM input is the input from the environment and the input from the datapath. The data from the environment is fed into the datapath, and the data output comes from the datapath. Figure 11.3 shows an example of the combination of the FSM with the datapath.

我们的一个状态机通信的例子是具有数据通路的状态机。这个组合被称为具有数据通路的有限状态机(FSMD)。 这个状态机控制着数据通路，数据通路完成计算。这个FSM数据是来自环境的输入和来自数据通路的输入。 来自于环境的数据输入到数据通路，输出来自数据通路。Figure 11.3表明一个具有 数据通路的FSM组合的例子。

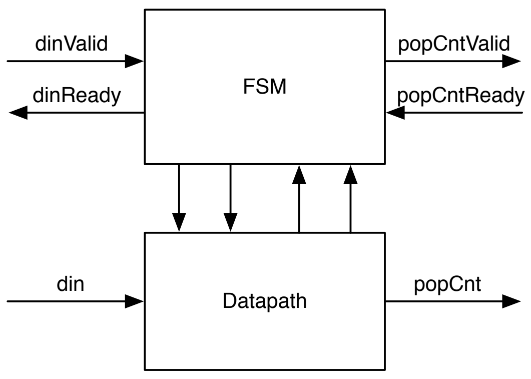

Figure 11.3: A state machine with a datapath.

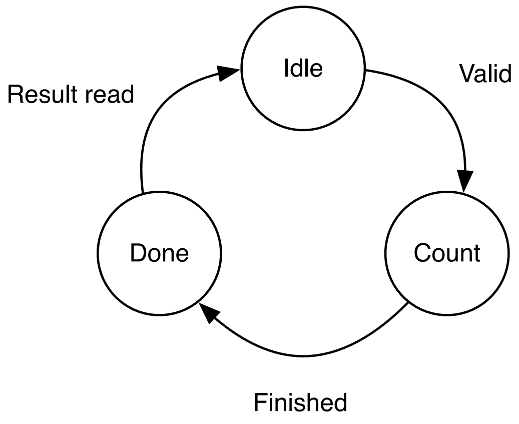

Figure 11.4: State diagram for the popcount FSM.

11.2.1 位一计数的例子

The FSMD shown in Figure 11.3 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.

FSMD在图片 11.3作为一个例子，去计算位1计数， 也被称为汉明重量。 汉明重量是一串符号中非零符号的个数。 对于二进制字符，是1的数量。

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 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.

位1计数单元包含数据输入*din*和结果输出*popCount*，连接到数据通路。 对于输入和输出，我们使用ready-valid握手信号。 当数据是可用的，出现valid。当接收器可以接受数据，出现ready。 当信号都出现，开始传输信号。握手信号传给FSM。 fsm使用控制信号传入数据通路，通过来自数据通路的状态信号和控制信号。

As a next step, we can design the FSM, starting with a state diagram, shown in Figure 11.4. We start in state *Idle*, where the FSM waits for input. When data arrives, signaled with a valid signal, the FSM advances to state *Load* to load a shift register. The FSM proceeds to the next state *Count*, there the number of '1's is counted sequentially. We use a shift register, an adder, an accumulator register, and a down counter to perform the computation. When the down counter reaches zero, we are finished and the FSM moves to state *Done*. There the FSM signals with a valid signal that the popcount value is ready to be consumed. On a ready signal from the receiver, the FSM moves back to the *Idle* state, ready to compute the next popcount.

作为下一步，我们可以设计FSM, 从状态图开始，像是Figure 11.4的那样。 我们从*Idle*状态开始，这里FSM等待输入。当数据到来的时候，数据伴随着有效信号，FSM进入到*Load*状态， 开始读取移位寄存器。FSM进行到下一个状态*Count*，这里'1'按照顺序读取。 我们使用一个移位寄存器，一个加法器，一个累加寄存器，和一个向下计数器去完成计算。 当向下计数器到达了0，我们完成了，FSM移动到状态*Done*。 这里具有valid的信号的FSM，它的位1计数开始被读取。 接收到接收器的ready信号以后，FSM返回到*Idle*状态，开始计算下一个位1计数。

The top level component, shown in Listing 11.5 instantiates the FSM and the datapath components and connects them with bulk connections.

顶层部分，像是11.5初始化了FSM和数据通路的部分，并且使用整体连线的方式进行连线。

```
class PopCount 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 11.5: The top level of the popcount circuit

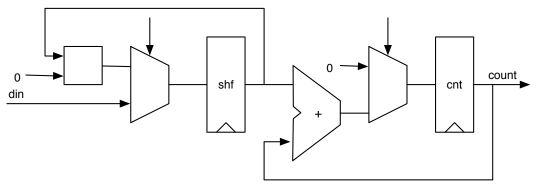

Figure 11.5: Datapath for the popcount circuit.

Figure 11.5 shows the datapath for the popcount circuit. The data is loaded into the *shf* register. On the load also the *cnt* register is reset to 0. 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 FSW switches back to *Idle*.

11.5表明了位1计数电路的数据通路。 数据读入*shf*寄存器。在读取端，*cnt*被置为0。 为了计算1的数量，*shf*是向右移动，每个周期最低位添加到*cnt*。 计数器没有出现在这个图片里，向下数，直到所有的二进制位移动到最低位。 当计数器到达零的时候，计数器结束了。 FSM切换到

状态0，位1计数完成了。FSM切换到状态*Done*，并且通过*popCntReady*声明信号结果。当开始读出结果的时候，通过查看*popCntValid*, FSM切换回*Idle*。

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.

对于load信号，regData寄存器作为输入被读取，regPopCount寄存器重置为0，计数器寄存器regCount设为设为需要移位的数量。

```
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 11.6: Datapath of the popcount circuit

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

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

Listing 11.7: The FSM of the popcount circuit

Otherwise, the regData register is 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. Listing 11.6 shows the Chisel code for the datapath of the popcount circuit.

另一方面，regData向右移动，最低位的二进制位通过regData寄存器添加到regPopCount寄存器，计数器向下计数直到为0。当计数器数到0，输出包含位1计数。 11.6 表明了用于位1计数的数据通路的chisel代码。

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 valid, the FSM switches to state done and waits till the popcount is read (signaled by popCntReady). ListingDatapath of the popcount circuit shows the code of the FSM.

FSM从状态idle开始。一旦输入(dinValid)有效，它切换到状态count，等到数据通路完成了计数。当位1计数的结果是有效的，FSM切换到状态done，等到位1计数开始读（通过popCntReady）。 The FSM of the popcount circuit 表明了FSM的代码。

11.3 Ready-Valid 界面

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 the output data using valid and ready signals.

在子系统间通信可以通过数据移动和控制流的握手信号进行归纳。 在位1计数的例子，我们已经看到了使用到valid-ready信号的输入和输出数据的握手界面。

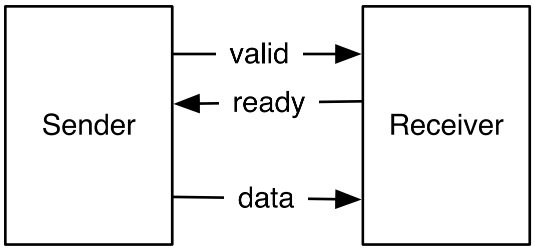

Figure 11.6: The ready-valid flow control.

The ready-valid interface is a simple flow control interface consisting of *data* and a *valid* signal at the sender side and a *ready* signal at the receiver side (see Figure 11.6). 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.

ready-valid界面是一个简单的控制流界面，包含了*data*和*valid*，在发送端，*ready*在接收端（参见 11.6）。发送端声明*valid*，当*data*是有效的。当接收端开始接收一个字长的数据，它声明了*ready*。当两边信号*valid*和*ready*全部被声明的时候，数据传输开始。如果两边的信号有一边没有被声明，没有传输发生。

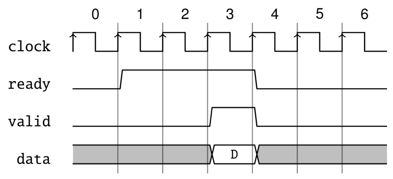

Figure 11.7: Data transfer with a ready-valid interface, early ready

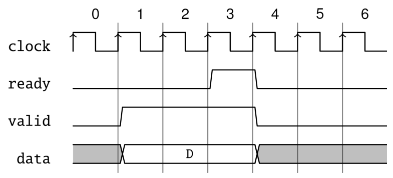

Figure 11.8: Data transfer with a ready-valid interface, late ready

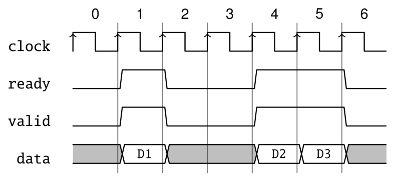

Figure 11.9: Single cycle ready/valid and back-to-back transfers

Figure 11.7 shows a timing diagram of the ready-valid transaction where the receiver signals ready (from clock cycle 1 on) before the sender has data. The data transfer happens in clock cycle 3. From clock cycle 4 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.

图片11.7表示一个ready-valid交易的时间图，这里接收器信号ready（从时钟周期1开始）在发送者具有数据前。数据在时钟周期3开始传输。从时钟周期4开始，发送者没有数据，或是接收器没有准备好下一个传输。当接收者在每个周期能够接受数据，这被称为“随时就绪”的界面，ready可以被硬件设为true。

Figure 11.8 shows a timing diagram of the ready-valid transaction where the sender signals valid (from clock cycle 1 on) before the receiver is ready. The data transfer happens in clock cycle 3. From clock cycle 4 on neither the sender has data nor the receiver is ready for the next transfer. Similar to the “always ready” interface we can envision and always valid interface. However, in that case the data will probably not change on signaling ready and we would simply drop the handshake signals.

图片11.8表示一个ready-valid交易，这里发送信号valid（从时钟周期1开始）在接收器准备好之前。数据传输在时钟周期3开始。从时钟周期4开始，发送者没有信号，或是接收者没有准备好下一个传输。类似于“时刻就绪”的界面，我们可以想象一个时刻有效的界面。但是，在那种情况下，数据可能不会根据ready发生改变，所以我们可能会取消握手信号。

Figure 11.9 shows further variations of the ready-valid interface. In clock cycle 1 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 4 and 5 with the transfer of D2 and D3

图片11.9表示了更多ready-valid接口。在时钟周期1，两个信号（ready和 valid都是高位，在一

个单一周期，然后数据传输D1发生。数据可以背对背传播（在每个周期）像是时钟周期4和5的传输D2和D3。

To make this interface composable neither ready not valid is allowed to depend combinational on the other signal. As this interface is so common, Chisel defines the DecoupledIO bundle, similar to the following:

为了让这个界面可以组成，ready或valid都不允许相互组合性依赖。因为这个接口比较常用，Chisel定义了DecoupledIO线束，类似于以下：

```
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线束是可参数化的，通过配置data的类型。Chisel定义的接口，使用数据的bits域。

One question remains if the ready or valid may be de-asserted after being active and *no* data transfer has happened. For example a receiver might be ready for some time and not receiving 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 clock cycles and becoming non-valid without a data transfer. 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.

一个问题是ready或是valid在全部有效以后是否可能是置零的，这样会没有数据发生传播。例如，接收者可能会就绪一些时间，并没有收到数据，但是由于其它的一些时间，可能没有就绪。相同的道理发生在发送端，在时钟数据有效，然后数据无效，没有数据传输。无论这个行为是否呗允许，这个不是ready-valid接口要讲的，但是它需要被接口使用方式上具体定义。

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:

当使用DecoupledIO类的时候，Chisel没有在ready和valid的信号设置前置条件。但是IrrevocableIO类放置了以下的条件，在接收方：

> 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.
>
> 一个具体的ReadyValidIO的子类，当valid是高位，ready是低位，保证不会在bits数值改变的一个周期后改变。也就是说，一旦valid升高，它就不会变低，直到下一个ready也升高。

Note that this is a convention that cannot be enforced by using the class IrrevocableIO.
注意这个是一个习惯，并不能被IrrevocableIO类强制规范。

AXI 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 that once ready or valid is asserted it is not allowed to get de-asserted until the data transfer happened.

AXI使用对于下列总线的元素使用ready-valid界面：读地址，读数据，写地址，和写数据。AXI现置了界面，一旦ready或是valid高位，它不允许变成低位，直到下一个数据传输发生。

Chapter 12

硬件生成器

The strength of Chisel is that it allows us to write so-called hardware generators. With older hardware description languages, such as VHDL and Verilog, we usually use another language, e.g., Java or Python, to generate hardware. The author has often written small Java programs to generate VHDL tables. In Chisel, the full power of Scala (and Java libraries) is available at hardware construction. Therefore, we can write our hardware generators in the same language and execute them as part of the Chisel circuit generation.

chisel有力的一点使它允许我们去写所谓的硬件生成器。 有了旧的硬件描述语言，类似VHDL或是Verilog， 我们一般使用其它语言，像是Java或是Python去生成硬件。 作者经常编写小型java程序去生成vhdl网表。 在chisel，scala（和java libraries）的完整力量在硬件构建上是可行的。 于是，我们可以使用相同语言编写硬件生成器，并去执行它们，作为chisel电路的生成。

12.1 一点Scala的内容

This subsection gives a very brief introduction into Scala. It should be enough to write hardware generators for Chisel. For an in-depth introduction into Scala I recommend the textbook by Odersky et al. [?].

这部分很简单地介绍一下Scala。对于写Chisel硬件生成器应该是足够的。对于一个有深度的Scala介绍，我推荐Odersky的 [?]书。

Scala has two types of variables: vals and vars. A val gives an expression a name and cannot be reassigned a value. This following snippet shows the definition of of an integer value called zero. If we try to reassign a value to zero, we get a compile error.

Scala有两类变量：val 和 var。val提供了一个命名的表达式，单手不能被重新赋值。下边的部分表示了一个被称为zero的整型数值。如果我们试图去重新赋值给zero，我们会得到编译错误。

```
// A value is a constant
val zero = 0
// No new assignment is possible
// The following will not compile
zero = 3
```

In Chisel we use vals only to name hardware components. Note that the := operator is a Chisel operator and not a Scala operator.

在Chisel，我们使用val只是命名硬件部分。注意到:=操作符是一个Chisel操作符，而不是Scala操作符。

Scala also provides the more classic version of a variable as var. Following code defines an integer variable and reassigns it a new value:

Scala也提供常见的变量var版本。以下代码规定了一个整型变量，并重新赋值：

```scala
// 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 it to use it to name a hardware *component*.
我们会需要 var 去编写硬件 生成器, 但是从来不需要使用它去命名一个硬件 组成。

You may have wandered what type those variables have. As we assigned an integer constant in the above example, the type of the variable is *inferred*; it is a Scala Int type. In most cases the Scala compiler is able to infer the type. However, if we are in the mood of being more explicit, we can explicitly state the type as follows:

你可能会纠结于这些变量是什么类型的。像我们在上述例子里把整型常量进行赋值，变量的类型是被推断的；它是一个Scala的Int类型。 在大多数类型，Scala编译器能够推断一个类型。但是，如果我们想要更加具体的形式，我们可以显性声明这些类型如下：

```scala
val number: Int = 42
```

Simple loops are written as follows: 简单的循环如下编写：

```scala
// 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.
我们使用一个循环用作电路生成器。以下的循环连接了移位寄存器的每一位。

```scala
val shiftReg = RegInit(0.U(8.W))

shiftReg(0) := inVal

for (i <- 1 until 8) {
  shiftReg(i) := shiftReg(i-1)
}
```

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.

条件使用if和else。注意到这个条件被Scala的runtime在电路生成的时候被评价。这个建构并不生成一个复用器。

```scala
for (i <- 0 until 10) {
  if (i%2 == 0) {
    println(i + " is even")
  } else {
    println(i + " is odd")
  }
}
```

12.2 使用参数配置

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.

chisel组成部分和函数可以通过参数被设置。 参数可以像是整数常量一样简单，但是也可以称为chisel硬件类型。

12.2.1 简单参数

The basic 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. Following example 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.

最基本的参数化电路的方式是去定义一个单元宽度作为参数。参数可以被传入作为chisel的构建器。 以下例子是那个模块的玩具例子，去实现一个可以参数化单元宽度的加法器。 这个参数宽度*n*是作为组成部分的参数(scala类型的 *Int*)传入构建器， 可以在IO捆束使用。

```scala
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:
加法器的参数化版本可以像如下被创造:

```scala
  val add8 = Module(new ParamAdder(8))
  val add16 = Module(new ParamAdder(16))
```

12.2.2 使用类型参数的函数

Having the bit width as a configuration parameter is just the starting point for hardware generators. A very flexible configuration is the usage of types. That feature allows for Chisel to provide a multiplexer (*Mux*) that can accept any types for the multiplexing. To show how to use types for the configuration, we build a multiplexer that accepts arbitrary types. Following function defines the multiplexer:

有了位宽作为参数，这是硬件生成器的开始点。一个非常复杂的设置是使用类型。 这个功能允许你使用chisel使用任何类型的输入作为复用器(*Mux*)。 为了表明如何使用参数的类型，我们搭建了一个可以接受任何类型的复用器。 以下函数定义了复用器:

```
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* set is *Data* or a subclass of *Data*. *Data* is the root of the Chisel type system.

chisel允许使用类型参数化函数，在我们的例子里是去使用chisel的类型。在方括号的表达式*[T <: Data]* 定义了一个参数变量*T*集合是*Data*或是*Data*的一个子集。 *Data*是chisel类型系统的根。

Our multiplexer function has three parameters: the boolean condition, one parameter for the true path, and one parameter for the false path. Both path parameters are of type *T*, an information that is provided at function call. The function itself is straight forward: we define a wire with the default value of *fPath* and change the value is 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.

我们的复用器函数有三个参数，布尔条件，一个参数用于真的通路，和一个参数用于假的通路。 两个参数属于类型*T*，一个信息在函数呼叫的时候被提供。 函数本身是直接的: 我们定义一个wire，具有默认值是*fPath*，并去改变这个值。 这个条件是一个常见的复用器函数。 在函数的末尾，我们返回复用器硬件。

We can use our multiplexer function with simple types such as *UInt*:

我们可以使用复用器函数，具有简单的类型类似于*UInt*:

```
  val resA = myMux(selA, 5.U, 10.U)
```

The types of the two multiplexer paths need to be the same. Following wrong usage of the multiplexer results in a runtime error:
两个复用器线路的类型需要一致。 以下复用器的错误使用会返回一个运行时间错误:

```
  val resErr = myMux(selA, 5.U, 10.S)
```

We define our type as a *Bundle* with two fields: 我们定义了我们的*Bundle*类型，打包进两个域:

```
class ComplexIO extends Bundle {
  val d = UInt(10.W)
  val b = Bool()
}
```

We can define *Bundle* constants by first creating a *Wire* and then setting the subfields. Then we can use our parameterized multiplexer with this complex type.
我们可以定义*Bundle*常量，通过首先创造一个*Wire*，然后设置子域。 然后我们可以通过这个复杂类型，来使用我们定制的复用器。

```scala
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 mulitplexer with a complex type
val resB = myMux(selB, tVal, fVal)
```

In our initial design of the function, we used *WireInit* 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. Following function shows the alternative way to code the multiplexer.

在我们最开始的函数设计，我们使用*WireInit*用于创建一个具有*T*的默认类型的wire。 如果我们想要创建一个只有chisel类型的weire，没有告知默认值，我们可以使用*fPath.cloneType* 去访问chisel的类型。 以下的函数表明了另一个方式去编写复用器。

```scala
def myMuxAlt[T <: Data](sel: Bool, tPath: T, fPath: T): T = {

  val ret = Wire(fPath.cloneType)
  ret := fPath
  when (sel) {
    ret := tPath
  }
  ret
}
```

12.2.3 具有类型参数的模块

We can also parameterize modules with Chisel types. Let us 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 *parametrize* 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. Additionally, in this example, we also make the number of router ports configurable.

我们也可以参数化模块，使用chisel类型。 让我们假定，我们想要设计一个noc芯片，在处理核心和数据之间移动。 但是，我们不想要在路由界面上硬编码数据类型；我们想要对其进行*parametrize*。 类似于函数的参数，我们对模块的构造器添加参数*T*。更多地，我们需要有那个类型的构造参数。 额外地，在这个例子，我们也使路由的端口数量变得可以设置。

```scala
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))
  })
```

To use our router, we first need to define the data type we want to route, e.g., as a Chisel *Bundle*:
为了使用我们的路由，我们首先需要去定义我们想要路由的数据类型，例如， 像是chisel的*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:
我们创造了一个路由，通过传入一个自定义的bundle的实例 并且给路由的构建器传入端口数量：

```
val router = Module(new NocRouter(new Payload, 2))
```

12.2.4 参数化的捆束

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:

在路由的例子，我们使用两个不同的矢量域作为路由的输入，一个用于地址，另一个用于数据，这个是参数化的。 一个更加优雅的解法是使用自参数化的*Bundle*，类似于:

```
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 it is used in a *Vec*, this public field is in the way. A solution (workaround) to this issue is to make the parameter field private:

*Bundle*具有类型*T*，是chisel的*Data*的子类型。 在bundle，我们定义了一个*data*域，通过对其类型呼叫*cloneType*。 但是，当我们使用构建器参数，这个参数称为类的公共域。当chisel需要复制这个*Bundle*的类型， 例如，当在*Vec*内部使用的情况下，这个公共域行不通。 这个的解决方式是把参数变为私有域。

```
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 通过新的*Bundle*，我们可以定义我们的路由端口

```
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))
  })
```

and instantiate that router with a *Port* that takes a *Payload* as a parameter:
使用*Port*去实例化我们的路由，采取*Payload*作为一个参数：

```
val router = Module(new NocRouter2(new Port(new Payload), 2))
```

12.3 生成组合逻辑

In Chisel, we can easily generate logic by creating a logic table with a Chisel *Vec* from a Scala *Array*. We might have data in a file, that we can read in during hardware generation time for the logic table. Listing*file reader* 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.

在Chisel，我们可以简单生成逻辑，通过使用来自继承自scala*Array*的chisel*Vec*创建的逻辑表。我们可能会在文件内有数据，便于我们在硬件生成逻辑表的时候读数据。 *file reader*表明了如何使用scala的scala标准library的*Source*类， 读出文件"data.txt"。这个包含了用于文字表示的整型常量。

```
  val table = VecInit(array.map(_.U(8.W)))
```

A Scala *Array* can be implicitly converted to a 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.
scala的*Array*可以隐式转换到序列(*Seq*)， 这个支持map函数*map*。 *map* 产生了一个函数，序列的每一个元素返回了一序列的函数返回值。 在我们的函数_.*U(8.W)*表示来自scala 序列的每一个*Int*，作为_，完成了 从scala*Int*到chisel*UInt*的表达式，具有8位的大小。 chisel的对象*VecInit*创建了一个chisel*Vec*，来自序列*Seq*的chisel类型。

```
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))
```

```scala
  })

  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 into the Chisel type Vec
  val table = VecInit(array.map(_.U(8.W)))

  // use the table
  io.data := table(io.address)
}
```

Listing 12.1: 读取文本文件生成逻辑表

We can use the full power of Scala to generate our logic (tables). E.g., generate a table of fixpoint constants to represent a trigonometric function, compute constants for digital filters, or writing a small 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.

我们可以使用scala的全部能力，去生成我们的逻辑表。例如，生成一个表格的定点类型常量，去表示一个三角函数， 计算数字滤波器的常量，或是编写一个小的scala汇编器用来生成chisel编写的微处理器。所有的这些函数在同一个代码库（同一个语言）并且可以在硬件生成的过程中被执行。

A classic example is the conversion of a binary number into a binary-coded decimal (BCD) representation. BCD is used to represent a number in a decimal format using 4 bits fo 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.

一个经典的例子是二进制数转换为binary-coded decimal (BCD)表达方式。BCD用来生成十进制格式，每个十进制的数使用4二进制位。例如，十进制*13*在二进制*1101*，在BCD里作为1和3的二进制*00010011*。BCD允许显示十进制数，是更为用户友好的数字表达形式，相比较于十六进制。

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. The key part of the conversion is just two lines.

我们可以编写一个Java程序用来计算表格并转换二进制到BCD。那个Java程序打印出的VHDL代码可以在一个项目中被引用。Java程序大概是100行代码；大多数代码是生成VHDL字符串。最核心的转换只是两行。

With Chisel, we can compute this table directly as part of the hardware generation. Listing 12.2 shows the table generation for the binary to BCD conversion.

有了chisel，我们可以直接计算这个表格作为硬件生成的一部分。12.2表明了这个表格的生成，作为二进制到BCD的转换。

```scala
import chisel3._

class BcdTable 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)

  // Convert binary to BCD
  for (i <- 0 to 99) {
    array(i) = ((i/10)<<4) + i%10
  }

  val table = VecInit(array.map(_.U(8.W)))
  io.data := table(io.address)
}
```

Listing 12.2: BcdTable

12.4 使用继承

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 out into a parent class. We explore how to use inheritance with an example.

Chisel是一个面对对象语言。一个硬件部分，chisel的*Module*是一个scala的类。于是，我们可以使用继承，去把一个共有的行为归类为父类。我们用以下例子来探索如何使用继承。

In Section 8.2 we have explored different forms of counters, which may be used for a low-frequency tick generation. Let us assume we want to explore those different versions, e.g., to compare their resource requirement. We start with an abstract class to define the ticking interface:

8.2 我们探索了不同类型的计数器，这个可以生成低频下的tick。我们假设，我们想要探索不同版本，例如，去比较其它资源需求。我们开始使用一个抽象类去定义tick的界面。

```scala
abstract class Ticker(n: Int) extends Module {
  val io = IO(new Bundle{
    val tick = Output(Bool())
  })
}
```

Listing 12.3 shows a first implementation of that abstract class with a counter, counting up, for the tick generation.
12.3表明那个抽象类型，具有一个计数器，向上计数，产生tick的第一个编写方式。

```scala
class UpTicker(n: Int) extends Ticker(n) {

  val N = (n-1).U

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

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

  io.tick := cntReg === N
}
```

Listing 12.3: 使用计数器生成tick

We can test all different versions of our *ticker* logic with a single test bench. We *just* need to define the test bench to accept subtypes of *Ticker*. *ticker test* 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.

我们可以测试所有不同的*ticker*逻辑，使用一个简单的测试平台。 我们emph只是需要去定义测试平台去接收*Ticker*的子类型。 *ticker test*表明了用于测试器的chisel代码。 *TickerTester*有一些参数：(1)类型参数*[T <: Ticker]*去接受一个*Ticker* 或是任何一个类型去继承*Ticker*。 (2) 接受测试的设计，作为类型*T*或是其子类型。 (3) 对于每个tick，我们期待的时钟周期。 测试器等待第一个tick发生的时间（开始的时间可能根据不同的做法有所不同），然后检查tick每n时钟周期重复。

```
import chisel3.iotesters.PeekPokeTester
import org.scalatest._

class TickerTester[T <: Ticker](dut: T, n: Int) extends PeekPokeTester(dut: T) {

  // -1 is the notion that we have not yet seen the first tick
  var count = -1
  for (i <- 0 to n * 3) {
    if (count > 0) {
      expect(dut.io.tick, 0)
    }
    if (count == 0) {
      expect(dut.io.tick, 1)
    }
    val t = peek(dut.io.tick)
    // On a tick we reset the tester counter to N-1,
    // otherwise we decrement the tester counter
    if (t == 1) {
      count = n-1
    } else {
      count -= 1
    }

    step(1)
  }
}
```

Listing 12.4: 用于测试不同版本ticker的测试器

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 12.5 shows the tick generation with a counter counting down to 0. Listing 12.6 shows the nerd version of counting down to -1 to use less hardware by avoiding the comparator.

第一个，简单的ticker的做法，我们可以测试测试器本身，可能是使用一些println作为debug手段。当我们对于这个简单的ticker感到有信心，并且测试器是正确的情况下，我们可以继续探索两个ticker的版本。12.5表明tick生成器，通过向下计数，数到0。列表12.6表明了nerd版本的向下数到-1，使用更少的硬件，通过规避比较器。

```scala
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 12.5: 使用向下计数器生成tick

```scala
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 12.6: 使用向下数到-1的计数生成

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 12.7 shows the specification. We run only the ticker tests with:

我们可以测试所有的3个ticker的版本，通过使用ScalaTest的要求，创建不同版本的ticker，并把它们传入testbench本身。 12.7表明要求。 我们使用ticker测试，通过:

```
sbt "testOnly TickerSpec"
```

```
class TickerSpec extends FlatSpec with Matchers {

  "UpTicker 5" should "pass" in {
    chisel3.iotesters.Driver(() => new UpTicker(5)) { c =>
      new TickerTester(c, 5)
    } should be (true)
  }

  "DownTicker 7" should "pass" in {
    chisel3.iotesters.Driver(() => new DownTicker(7)) { c =>
      new TickerTester(c, 7)
    } should be (true)
  }

  "NerdTicker 11" should "pass" in {
    chisel3.iotesters.Driver(() => new NerdTicker(11)) { c =>
      new TickerTester(c, 11)
    } should be (true)
  }
}
```

Listing 12.7: ScalaTest specifications for the ticker tests

12.5 使用函数式编程做硬件生成

Scala supports functional programming, so does Chisel then. We can use functions to represent hardware and combine those hardware components with functional programming (by using a so-called “higher-order function”). Let us start with a simple example, the sum of a vector:

Scala支持函数式编程，所以Chisel也支持。我们可以使用函数去表示硬件，并结合这些硬件部分，使用函数式编程（通过使用所谓的“高阶函数”）。 我们开始使用一个简单的例子，向量的加和:

```
def add(a: UInt, b:UInt) = a + b
val sum = vec.reduce(add)
```

First we define the hardware for the adder in function add. The vector is located in vec. The Scala reduce() method combines all elements of a collection with a binary operation, producing a single value. The reduce() method reduces the sequence starting from the left. It takes the first two elements and performs the operation. The result is then combined with the next element, until a single result is left.
首先我们定义加法硬件，使用函数add。向量位于vec。Scala的reduce()方法使用位数操作合并了这个集合的元素，产生一个单一值。reduce()方法从左开始reduce。这需要前边两个元素来完成操作。数值然后和下一个合并，直到剩下一个单一结果。

The function to combine to elements is provided as parameter to reduce, in our case add, which returns an adder. The resulting hardware is a chain of adders computing the sum of the elements of vector vec.

合并的函数作为参数输送给reduce，在我们的例子是add，返回了一个加法器。这个产生的硬件是一连串的加法器，计算一个向量vec的加和。

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.

除了定义这个(简单)的add函数，我们可以提供加法，作为匿名函数，并且使用Scala的通配符 "_" 去表示这两个操作数。

```
val sum = vec.reduce(_ + _)
```

With this one liner we have generated the chain of adders. For the sum function a chain is not the ideal configuration, 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 generated a tree of adders:
有了这一行代码，我们生成了一串加法器。对于加法函数，一个链条并不是理想的配置，一个树会有一个更短的组合性延迟。如果我们不信任综合工具去放置我们的加法链，我们可以使用Chisel的reduceTree方法去生成一个加法器的树：

```
val sum = vec.reduceTree(_ + _)
```

Chapter 13

示例设计

In this section, we explore some small size digital designs, such as a FIFO buffer, which are used as building blocks for a larger design. As another example, we design a serial interface (also called UART), which itself may use the FIFO buffer.

在这个部分，我们探索一些中等大小的设计，例如FIFO缓冲，这个是用于大型设计中的部分。作为一个例子，我们会设计一个串行接口（也称UART），它本身会使用FIFO缓冲器。

13.1 FIFO 缓冲器

We can decouple a write (sender) and a reader (receiver) by a buffer between the writer and reader. A common buffer is a first-in, first-out (FIFO) buffer. Figure 13.1 shows a writer, the FIFO, and a reader. Data is put into the FIFO by the writer on *din* with an active *write* signal. Data is read from the the FIFO by the reader on *dout* with an active *read* signal.

为了分离一个写入（发送者）和一个读取（接收者），在写和读取之间插入一些形式的缓冲。一个常见的缓冲是一个先进先出（FIFO）缓冲器。图7.1表明了一个写入，一个FIFO，和一个读取。数据通过din写入，和一个激活write的信号。信号从FIFO读取通过读取dout，和一个激活read的信号。

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 lost. In other words, the signals empty and full serve as handshake signals

FIFO初始是空的，通过empty信号表明空着。从一个空的FIFO读取一般是没有被定义的。当数据写入，并从来没有从FIFO读取，会变得full。写入一个full的FIFO经常被忽略，数据随之丢失。换句话说，empty和full充当握手信号。

Several different implementations of a FIFO are possible: E.g., using on-chip memory and read and

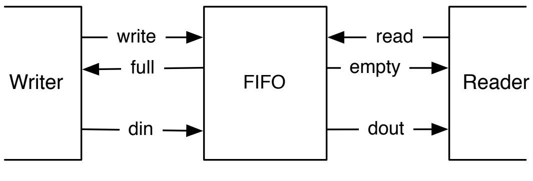

Figure 13.1: A writer, a FIFO buffer, and a reader.

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 with a low resource requirement. The code of the bubble FIFO is available in the chisel-examples repository.[1]

可能一个FIFO有多个表现形式：例如，使用片上存储和读写指针，或只是一连串寄存器，有一个简单的状态机。对于小的缓存（最多是10个元素），一个FIFO被单独寄存器组织起来，这些寄存器连接到一串缓存，这个是一个简单，低资源的要求。冒泡FIFO的代码在chisel-examples文件仓库。[2]

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 side.

我们开始定义IO信号用于写入和读出端。 数据的大小通过size设置。 写入数据是din，写入信号是write。 full作为flow control 在写入端。

```
class WriterIO(size: Int) extends Bundle {
  val write = Input(Bool())
  val full = Output(Bool())
  val din = Input(UInt(size.W))
}
```

Listing 13.1: bubble fifo writer io

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.

读出端提供了dout作为数据，读出端使用read初始化。 empty信号负责控制读出端的信号控制。

Listing 13.2 shows a single buffer. The buffer has a enqueueing port enq of type WriterIO and a dequeueing port deq of type ReaderIO. The state elements of the buffer is 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.

13.2表明了一个单个缓存。缓存有一个进入端口enq，具有WriterIO的类型， 和一个输出端口deq，具有ReaderIO的类型。缓存的状态元素是一个寄存器， 含有数据（dataReg和一个状态寄存器用于一个简单的FSM(stateReg)。 FSM只有两个状态：当状态是empty或是full。 如果缓存位于empty，写入会寄存输入数据，并且改变到full的状态。 如果缓存位于full，读出会倒出数据，并且改变为empty状态。 IO端口full和empty代表缓存的写入和读取的状态）

```
class FifoRegister(size: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })
```

[1]For completeness, the Chisel book repository contains a copy of the FIFO code as well.
[2]为了完整性，这个Chisel书的仓库也包含了一份FIFO的代码。

```

  val empty :: full :: Nil = Enum(2)
  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 13.2: 单级的冒泡FIFO

Listing13.3 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 an depth stages bubble FIFO out of depth FifoRegisters. We crate the stages by filling them into a Scala Array. The Scala array has no hardware meaning, it *just* provides us with a container to have references to the created buffers. In a Scala for loop we connect the individual buffers. The first buffers 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.

13.3表明了一个完整的FIFO。完整的FIFO具有一样的IO端口，像是单独的FIFO缓存。BubbleFifo具有数据字长的size参数，和缓存的级数depth。我们可以搭建一个depth的阶段用于缓冲FIFO，来自FifoRegister的深度depth。我们通过把它们装填入scala的Array。scala的array没有任何硬件意义，它只是提供给我们一个容器，去有创造缓存的依据。在scala的for循环，我们连接了单独的缓存。在第一个buffer，读入端接入完整的FIFO端口，最后一个缓存的读出端连接到完整FIFO的读出端。

```
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 13.3: 一个由一串冒泡FIFO串组成的FIFO

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 simple, and a good solution when the data rate is considerable slower than the clock rate, e.g., as a decouple buffer for a serial port, which is presented in the next section.

这个展示的想法是连接独立的缓冲, 去补充一个FIFO队列, 称为冒泡FIFO, 因为数据从队列冒泡。这个简单的解法，在数据率比时钟慢的情况下是好的，例如，用于序列端口的去耦缓冲，在下一部分展示。

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 2 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 13.3.

但是，当数据绿达到时钟频率，冒泡FIFO有两个限制：(1)因为每个缓存需要在*empty*和*full*摆动，这意味着最大吞吐量是每个字长两个时钟周期。(2)数据需要通过整个FIFO进行冒泡，于是，从输入到输出的延迟至少是buffer的数量。我会展示另外的FIFO的可能做法Section 13.3。

13.2 一个串口端口

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, least significant bit first, and then one or two stop bits (1). When no data is transmitted, the output is 1. Figure 13.2 shows the timing diagram of one byte transmitted.

一个串口(经常被称为UART 或是RS-232)是其中一个最简单的方法去连接你的电脑和FPGA板子。像名字暗示的那样，数据串行传输。一个8位的字节按照以下传输：从开始的0位开始，8位的数据，从最低位开始，然后是每次1到2个比特。当没有数据传输的时候，输出是1. 图片 13.2表明了一个字节传输的时序图。

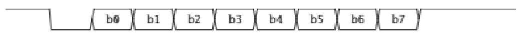

Figure 13.2: 被UART传输的一个字节

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.

我们设计我们的UART以一个模块的方式，每个模块实现最小功能。我们呈现一个发送器(TX), 接收端(RX), 一个缓存，然后是使用这些基本的单元。

First, we need an interface, a port definition. For the UART design, we use a ready/valid handshake interface, with the direction as seen from the transmitter.

首先，我们需要一个界面，一个端口定义。从UART的定义，我们使用ready/valid握手界面，从发送端观察。

```scala
class Channel extends Bundle {
  val data = Input(Bits(8.W))
  val ready = Output(Bool())
  val valid = Input(Bool())
}
```

Listing 13.4: uart channel

The convention of a ready/valid interface is that the data is transferred when both *ready* and *valid* are asserted.

ready/valid界面的常见做法是数据当*ready*和*valid*高位的时候进行传输。

```scala
class Tx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
    val channel = new Channel()
  })

  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 := Cat(1.U, shift(9, 0))
      bitsReg := bitsReg - 1.U
    }.otherwise {
      when(io.channel.valid) {
        // two stop bits, data, one start bit
        shiftReg := Cat(Cat(3.U, io.channel.data), 0.U)
        bitsReg := 11.U
      }.otherwise {
        shiftReg := 0x7ff.U
      }
    }

  }.otherwise {
    cntReg := cntReg - 1.U
  }
}
```

Listing 13.5: uart tx

Listing **??** 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 correct timing, we compute a constant for by computing the time in clock cycles for one serial bit.

?? 表明一个准系统串行发送器(Tx)。 IO端口是txd端口，这里串行数据进行发送，Channel是发送器用来接受串行和发送的数据。 为了产生正确的时序，我们计算一个常量，通过计算一个串行二进制位的时钟周期。

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. No additional state register of FSM is needed, all state is encoded in those three registers.

我们使用三个寄存器： (1) 移动数据的寄存器(串行化)(shiftReg)。 (2) 一个计数器去生成正确的波特率(cntReg)，和 (3) 一个计数器用于计算需要被移动走二进制位的数量。 不需要额外的FSM状态寄存器，所有的状态在这三个寄存器编码。

Counter cntReg is continuously running (counting down to 0 and reset to 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.

计数器cntReg持续在运行(向下数到0，数到0以后重置到开始值)。 所有的行为只有当cntReg是0的时候结束。当我们搭建一个最小的发送器， 我们只有移位寄存器去保存数据。于是，这个通道只有当cntReg是ready的， 没有剩下的二进制位需要被移走。

The IO port txd is directly connected to the least significant bit of the shift register.

IO端口txd直接连接到移位寄存器的最小位。

When there are more bits to shift out (bitsReg =/= 0.U), we shift the bits to the right and fill with 1 from the top (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 valid port). 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.

当存在更多二进制位去移动(bitsReg =/= 0.U)， 我们移动二进制位到右边，并把它从顶部填充1（发送器的闲置） 。 如果没有更多的二进制位需要被移位，我们检查一下通道是否包含数据 (通过valid端口标志信号)。 如果是这样的，留做移动的二进制字符串通过首位(0), 8位数据，和两个停止位(1)。 于是， 二进制位的总数被设为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. Accepting new data only when cntReg is 0 also means that the ready flag is also de-asserted when there would be space in the shift register. However, we do not want to add this “complexity” to the transmitter but delegate it to a buffer.

这是一个非常小的发射器，没有额外的缓存，可以接受一个新的字符，当且仅当移位寄存器是空的， 并且时钟周期是cntReg为0的情况下。 当且仅当在cntReg为0的情况下接收新的数据也意味着 当移位寄存器没有空位的时候，ready符号也会被取消。 但是，我们不想把这个复杂性添加到发射器，而是把它作为缓冲器。

```
class Buffer extends Module {
  val io = IO(new Bundle {
    val in = new Channel()
    val out = Flipped(new Channel())
  })

  val empty :: full :: Nil = Enum(2)
  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.data
      stateReg := full
    }
  }.otherwise { // full
    when(io.out.ready) {
      stateReg := empty
    }
  }
  io.out.data := dataReg
}
```

Listing 13.6: 一个具有ready/valid界面的单字节缓冲

Listing 13.6 shows a single byte buffer, similar to the FIFO register for the bubble FIFO. The input port is a Channel interface, and the output is the Channel interface with flipped directions. The buffer contains the minimal state machine to indicate empty or full. The buffer driven handshake signals (in.ready and out.valid depend on the state register.

13.6表明了一个单字节的缓冲器， 类似于FIFO寄存器，用于冒泡FIFO。输入端口是Channel界面， 输出是翻转的Channel界面。 缓冲包含最小状态机，去表明是否empty还是full。被驱动的握手信号(in.ready和out.valid取决于状态寄存器)。

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.

当状态为empty的时候，数据输入是valid， 当我们寄存数据，状态切换到状态full。 当状态切换到full，下方的接收器ready， 向下数据传输开始， 我们切换回状态empty。

```
class BufferedTx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
    val channel = new Channel()
  })
  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 13.7: 一个具有额外缓存的发送器

With that buffer we can extend our bare-bone transmitter. Listing 13.7 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.

有了缓冲器，我们可以拓展我们的发射器原型。 13.7表明了发射器的组合Tx和一个单独的缓冲器在前边。 这个缓冲器现在变为这样一个问题，Tx只有在单时钟周期是ready的。 我们现在把这个问题变为缓冲器的问题。 单字长缓冲器的拓展，对于一个真实FIFO，可以简单被完成， 并且不需要在发射器或是单字节缓冲器做出改变。

```
class Rx(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val rxd = Input(Bits(1.W))
    val channel = Flipped(new Channel())
  })

  val BIT_CNT = ((frequency + baudRate / 2) / baudRate - 1).U
  val START_CNT = ((3 * frequency / 2 + baudRate / 2) / baudRate - 1).U

  // Sync in the asynchronous RX data
  // Reset to 1 to not start reading after a reset
  val rxReg = RegNext(RegNext(io.rxd, 1.U), 1.U)

  val shiftReg = RegInit('A'.U(8.W))
  val cntReg = RegInit(0.U(20.W))
  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
    shiftReg := Cat(rxReg, shiftReg >> 1)
    bitsReg := bitsReg - 1.U
    // the last shifted in
    when(bitsReg === 1.U) {
      valReg := true.B
    }
  // wait 1.5 bits after falling edge of start
  }.elsewhen(rxReg === 0.U) {
    cntReg := START_CNT
    bitsReg := 8.U
  }

  when(valReg && io.channel.ready) {
    valReg := false.B
  }

  io.channel.data := shiftReg
  io.channel.valid := valReg
}
```

Listing 13.8: 用于串口的接收器

Listing 13.8 shows the code for the receiver (Rx). A receiver is a little 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 shifts in the bits every bit time. You can observe these two waiting times as START_CNT and BIT_CNT. For both times, the same counter (cntReg) is used. After 8 bits are shifted in, valReg signals an available byte

13.8 表明了接收器(Rx)的代码。 接收器有点需要技巧，因为它需要去重新构建串行数据的时序。 接收端等待开始位的下降沿。 从那个时刻，接收端以每次1.5二进制位的速度等待，到它本身的位置0。 然后它每次移位一个二进制位。你可以观察这两个等待时间, 通过START_CNT和BIT_CNT。 对于这两个时间，使用相同的计数器(cntReg)。 在8位全部移动进入，valReg标志着字节有效。

```
class Sender(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(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.data := text(cntReg)
  tx.io.channel.valid := cntReg =/= len

  when(tx.io.channel.ready && cntReg =/= len) {
    cntReg := cntReg + 1.U
  }
}
```

Listing 13.9: 通过串口发送“Hello World!"

Listing 13.9 shows the usage of the serial port transmitter by sending a friendly message out. We define the message in a Scala string (msg) and converting 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 functions return values. If the function literal shall have only one argument, as it is 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 VecInint 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.

13.9表明了串行端口的使用，通过发送一个友善的信息。 我们在scala字符串定义消息(msg)并把它转换为chisel的一个Vec的UInt。 scala字符串是一串序列，支持map方法。map方法作用于函数字面量的参数，把这个函数应用于每个元素，并且搭建一串函数，返回数值。如果函数字面量应该有一个参数，作为这个例子， 这个参数可以通过_表示。我们的函数字面量呼叫chisel方法.U去传唤scala的Char 变为chiselUInt。 这个序列然后传递到VecInint去构建chisel的Vec。 我们使用计数器cntReg索引向量text，去提供单一的字符到具有缓存的发送器。每收到ready信号，我们每增加计数器，直到完整的字符串被发送。 发送方保持valid为真，直到最后的字符被发送。

```scala
class Echo(frequency: Int, baudRate: Int) extends Module {
  val io = IO(new Bundle {
    val txd = Output(Bits(1.W))
    val rxd = Input(Bits(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 13.10: 在串口显示数据

Listing 13.10 shows the usage of the receiver and the transmitter by connecting them together. This connection generates an Echo circuit where each received character is sent back (echoed).

13.10 表明了接收端和发送端的使用，通过将它们相连。 这个连接产生了Echo电路，这里每个接受到的字符都被返回（打印echo）。

13.3 设计FIFO中的变量

.

In this section we will implement different variations of a FIFO queue. To make these implementations interchangeable we will use inheritance, as introduced in Section 12.4.

在这个部分，我们会补充一个FIFO序列的不同变量。 为了补充这些可互换量，我们会使用继承， 像是在12.4介绍的那样。

13.3.1 参数化FIFO

We define an *abstract* FIFO class:

我们定义一个*abstract*的FIFO类型：

```scala
abstract class Fifo[T <: Data](gen: T, depth: Int) extends Module {
  val io = IO(new FifoIO(gen))

  assert(depth > 0, "Number of buffer elements needs to be larger than 0")
}
```

Listing 13.11: fifo abstract

In Section 13.1 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 (e.g., we write into the FIFO when it is not full. However, we can generalize this handshaking to the so called ready-valid interface. E.g, 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:[3]
在 13.1我们定义了我们的具有常见名称的界面类型， 例如write, full, din, read, empty和dout。 这样的缓存输入和输出包含了数据和两套握手信号(例如，当不是full的时候， 我们write入FIFO。) 但是，我们可以产生这个握手信号，作为所谓的ready-valid界面。 例如，我们可以产生一个元素（写入FIFO），当FIFO是ready。 我们使用valid作为写入端的信号。 因为这个ready-valid界面是很常见的，chisel提供了一个DecoupledIO界面，定义如下：[4]

```
  class DecoupledIO[T <: Data](gen: T) {
    val ready = Input(Bool())
    val valid = Output(Bool())
    val bits  = Output(gen)
  }
```

Listing 13.12: fifo decoupled

With the DecoupledIO interface we define the interface for our FIFOs: a FifoIO with an enq enqueue and a deq dequeue port consisting of ready-valid interfaces.
随着我们定义的用于我们的FIFO的DecoupledIO界面： 一个FifoIO，具有一个enq发送队列和一个deq接收队列，组成了ready-valid界面。
The DecoupledIO interface is defined from the writer's (producer's) view point. Therefore, enqueue port of the FIFO needs to flip the signal directions.
DecoupledIO界面通过写入端的角度（或是产生端）的角度。 于是，FIFO的输入端需要去翻转信号的方向。

```
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 for different FIFO implementations optimized for different parameters (speed, area, power, or just simplicity).

通过抽象的类和一个接口，我们可以定义不同的FIFO，用于不同的目标（速度，面积，功耗，或是简单）。

[3]This is a simplification, as DecoupledIO actually extends an abstract class.
[4]这是一个简化，因为DecoupledIO实际上拓展了一个抽象类型。

13.3.2 重新设计冒泡FIFO

We can redefine our bubble FIFO from Section 13.1 using standard ready-valid interfaces and being parametrizable with a Chisel data type.

我们可以从13.1重新定义我们的冒泡FIFO，使用标准的ready-valid界面，并且可以通过Chisel数据类型参数化。

```scala
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 13.13: 一个具有ready-valid界面的冒泡FIFO

Listing 13.13 shows the refactored bubble FIFO with ready-valid interface. Note what we put the Buffer component inside from BubbleFifo as private class. This helper class is only needed for this component and therefore we hide it and avoid polluting the name space. The buffer class has also been simplified. Instead of an FSM we use only a single bit, fullReg, to note the state of the buffer: full or empty.

13.13表示了经过重新参数化的具有ready-valid界面的冒泡FIFO。注意到我们放到Buffer的部分是BubbleFifo作为一个私有类的内部的。这个帮助类只是对这个部分是需要的，于是我们隐藏，并且防止污染命名空间。缓存类也被简单化了。除了我们用于单一位的, fullReg, 用来表示

缓存的状态：满或是空。

The bubble FIFO is simply, 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 two clock cycles per word.

冒泡FIFO是简单的，易于理解，使用最少资源的。但是，因为每个缓存级需要在空和满之间摆荡，最大的FIFO带宽是每两个时钟一个字。

One could consider to look 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.

你可以考虑在缓存使用两个界面，当生产者valid和消费者的ready置高位的时候，接收新的字。 但是，这个引入了组合性路径，从消费者的握手到生产者的握手，这个违反了ready-valid协议的定义。

13.3.3 Double Buffer FIFO

One solution is stay ready even when the buffer register if full. To be able 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 the buffer is full, new data is stored in the shadow register and ready is de-asserted. 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.

一个解决方式是保持ready，甚至缓存寄存器是满的情况下。为了从生产者能够接受新的数据字，当消费者不ready的时候，我们需要第二个缓存。我们称为影子寄存器。当缓存是满的情况下，新的数据保存在影子寄存器，并且ready置零。当消费者重新变为ready，数据从数据寄存器传输到消费者，并且从影子寄存器传输到数据到数据寄存器。

```scala
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))

    val empty :: one :: two :: Nil = Enum(3)
    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 13.14: 一个双重缓存的FIFO

Listing 13.14 shows the double buffer. 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 shadowReg. 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 has valid data when it is in state one or two.

13.14表示了双重缓存。每个缓存可以保存两个元素，我们只需要一半的缓存元素(depth/2)。DoubleBuffer包含两个寄存器，dataReg和shadowReg。消费者从shadowReg被服务。双重缓存有三个状态：empty，one，和two，这些信号表示了双重缓存被填满的状态。 缓存是ready去接收新的数据，当它在状态empty或是one。具有有效数据的状态是one或是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 de-asserts 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 fo clock cycles for the same buffer capacity. Similar the fall through latency is half of the bubble FIFO.

如果我们全速运行FIFO，并且消费者一直是ready，双重缓存的稳定状态是one。只有当消费者ready取消以后，队列满了，缓存进入状态two。 但是，相比于单一冒泡FIFO，一个队列的重启只需要半个时钟周期，对于相同的缓存容量。 类似地，延迟也是bubble FIFO的一半。

13.3.4 具有寄存存储器的FIFO

When you come with a software engineering background you may have been wondering that we built hardware queues out of many small individual small buffer elements, all executing in parallel and handshaking with upstream and downstream elements. For small buffers this is probably the most efficient implementation.

如果你来自软件工程背景，你可能疑惑于我们既然搭建了很多小的单独的缓存元素，所有的事物并行运行，并且和上流下流元素握手。对于小的缓存，这个可能是最有效率的做法。

A queue in software is usually used by a sequential code in a single thread. Or as 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, the are set back to the begin 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.

一个软件的队列经常在一个线程被作为一序列的代码使用，或者作为一个队列，去耦合一个生产者和消费者的线程。 在这个设置，一个固定大小的FIFO队列，经常被写为循环缓存。 两个指针指到读写部分，在一个队列的存储器一段。当指针指到了存储器的尾端，指针回到存储器的开始。两个指针的区别是队列中的数字，当两个指针指到了相同的地址，这个队列可能是满的，或是空的。为了区别空或是满，我们需要另外一个flag信号。

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 13.15 shows a FIFO queue implemented with memory and read and write pointers.

我们也可以补充这个基于存储器的FIFO的硬件。对于小的队列，我们可以使用一个寄存器文件（例如，一个Reg(Vec())）。13.15表明了一个通过存储器读写指针补充的FIFO队列。

```
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)

  when (io.enq.valid && !fullReg) {
    memReg(writePtr) := io.enq.bits
    emptyReg := false.B
    fullReg := nextWrite === readPtr
    incrWrite := true.B
  }

  when (io.deq.ready && !emptyReg) {
    fullReg := false.B
    emptyReg := nextRead === writePtr
    incrRead := true.B
  }

  io.deq.bits := memReg(readPtr)
  io.enq.ready := !fullReg
  io.deq.valid := !emptyReg
}
```

Listing 13.15: 一个基于寄存存储器的FIFO

As there are two pointers that behave the same, being 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 of 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.

因为有两个指针的行为是一样的，向上数的型位，并把它们在缓存尾端重新来过，我们定义一个函数counter，用来补充这些轮转的计数器。通过log2Ceil(depth).W我们计算计数器的位长。下一个值是增加一，或是重来到0。寄存器只有在输入incr是true.B的情况下增加。

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 so called *tuple*, which is simply a container to hold more than one value. The syntax to create such a duple is simply wrapping the comma separated values in parentheses:

更多地，因为我们也需要下一个可能的值（增加或回到0），我们要通过counter函数返回数值。在Scala我们可以返回一个被称为*tuple*的类型，这个只是一个容器，去包含不止一个数值。用来创建这样一个tuple的语法是简单包裹，使用逗号外加括号:

```
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 signal to increment the read and write pointer and create the read and write pointers with the function counter. When both pointer are equal, the buffer is either empty or full. We define two flags to for the notion of empty and full.

对于存储器，我们创建一个寄存器的向量(Reg(Vec(depth, gen))，类型是Chisel类型gen。我们定义两个信号去读写指针，创建读和写的指针，使用函数counter。当两个指针是相同的，缓存可能是空或满的。我们定义两个flag信号，用于记录空和满。

When the producer asserts valid and the FIFO is not full we: (1) write into the buffer, (2) ensure emptyReg is de-asserted, (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.

当生产者的valid置一，并且FIFO不是满的情况下，我们：（1）写入缓存，（2）确保emptyReg置零，（3）标记缓存是满的，如果写指针能够跟上读指针，在下一个周期循环（比较现在的读指针，和下一个写指针），（4）发出信号，让写指针增加。

When the consumer is ready and the FIFO is not empty we: (1) ensure that the fullReg is de-asserted, (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.

当消费者是ready状态，FIFO不是空的，我们：（1）确保fullReg置零，（2）如果下个周期，读取指针能够跟上写指针，标记空的缓存，（3）发信号，让读取指针增加。

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.

FIFO的输出是读指针地址的存储器元素。 ready和valid的flag信号只是简单从full和empty的flag信号得来的。

13.3.5 使用片上存储的FIFO

The last version of the FIFO used a register files 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 13.16 shows a FIFO using a synchronous memory for storage.

最后的FIFO版本使用寄存器文件去表示存储器，这个对于小的FIFO是好的解决方法。 对于更大的FIFO，最好使用片上存储。 13.16表示一个使用同步的存储器，用于保存数据。

```
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)

  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 idle :: valid :: full :: Nil = Enum(3)
  val stateReg = RegInit(idle)
  val shadowReg = Reg(gen)

  when (io.enq.valid && !fullReg) {
   mem.write(writePtr, io.enq.bits)
```

```
    emptyReg := false.B
    fullReg := nextWrite === readPtr
    incrWrite := true.B
  }

  val data = mem.read(readPtr)

  // Handling of the one cycle memory latency
  // with an additional output register
  switch(stateReg) {
    is(idle) {
      when(!emptyReg) {
        stateReg := valid
        fullReg := false.B
        emptyReg := nextRead === writePtr
        incrRead := true.B
      }
    }
    is(valid) {
      when(io.deq.ready) {
        when(!emptyReg) {
          stateReg := valid
          fullReg := false.B
          emptyReg := nextRead === writePtr
          incrRead := true.B
        } otherwise {
          stateReg := idle
        }
      } otherwise {
        shadowReg := data
        stateReg := full
      }

    }
    is(full) {
      when(io.deq.ready) {
        when(!emptyReg) {
          stateReg := valid
          fullReg := false.B
          emptyReg := nextRead === writePtr
          incrRead := true.B
        } otherwise {
          stateReg := idle
        }

      }
    }
  }

  io.deq.bits := Mux(stateReg === valid, data, shadowReg)
  io.enq.ready := !fullReg
  io.deq.valid := stateReg === valid || stateReg === full
```

```
}
```
Listing 13.16: 使用片上存储的FIFO

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.

处理读取和写入的指针和寄存存储FIFO是类似的。但是，一个同步片上的存储器在下一个时钟周期传达结果，读寄存器文件的行为是在同周期完成的。

Therefore, we need some additional FSM and a shadow register to handle this latency. We read the memory out and provide the value of the top of the queue to the output port. If that value is not consumed, we need to store it in the shadow register shadowReg while reading the next value from the memory. The state machine consists of three states to represent: (1) an empty FIFO, (2) a valid data read out from the memory, and (3) head of the queue in the shadow register and valid data (the next element) from the memory.

于是，我们需要一些额外地FSM和一个影子寄存器去处理这个延迟。我们从存储器读出，并提供队列的顶端数值给输出。如果那个数值没有被消费，我们需要把它存入影子寄存器shadowReg的同时，从存储器读入下一个值。状态机包括三种状态去表达：（1）一个空的FIFO，（2）一个有效数值从存储器读出，（3）影子寄存器的头部和来自存储器的有效数据（下一个元素）。

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 13.17 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.

存储器为基本FIFO可以有效地在队列中保持更多的数据，和更短的延时。在最后的设计，输出的FIFO可能直接来自存储器读出。如果这个数据通路在设计的关键路径，我们可以简单地通过合并两个FIFO，流水线我们的设计。13.17表示了这样一个组合。对于存储器为基础的FIFO，我们增加一个简单的级，双重缓存的FIFO，去解耦存储器从输出的读出路径。

```
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 13.17: 使用双重缓存合并一个存储器为基础的FIFO

13.4 练习

This exercise section is a little bit longer as it contains two exercises: (1) exploring the bubble FIFO and implement a different FIFO design; and (2) exploring the UART and extending it. Source code for both exercises is included in the chisel-examples repository.

练习部分有点长，它包含两个练习：(1)探索冒泡FIFO，并补充一个不同的FIFO设计；(2)探索UART并去拓展它。 两个练习的源码在chisel-examples仓库。

13.4.1 探索冒泡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 FifoTester 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 starts GTKWave for waveform viewing. Explore the tester and the generated waveform.

FIFO也包括一个测试器，测试器包括读写操作，并生成波形文件 value change dump (VCD) 格式。 VCD也可以在波形观察器上查看，例如 GTKWave。 查看 FifoTester 的文件夹. 这个代码库也包含了 *Makefile* 用来运行例子, 例如FIFO例子 只是输入:

```
$ make fifo
```

这个命令会编译FIFO，运行测试，并且从GTKWave开始查看波形。 请查看测试器和生成的波形。

In the first cycles, 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.

在第一个周期中，测试器会写一个单词。我们可以观察到 波形该字如何通过FIFO冒泡，因此 名称*bubble FIFO*。这种冒泡也意味着 数据字通过FIFO的等待时间等于FIFO的深度。

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 side.

下一个，是测试填充FIFO直到满，随之是一个简单的读。注意到空word如何从读到写端冒泡。 当一个冒泡FIFO是满的，它需要一个buffer深度的延迟用来读，去影响写端。

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.

测试的末尾包含一个循环，试图去写入和读取，在最大的速度。 我们可以看到冒泡FIFO在最高带宽下运行，这个是每周期两个word。 一个缓冲阶段任何时候都可以在空和满状态下转递单个word。

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 end to the reader end, and (3) a full FIFO needs n clock cycles for the restart.

一个冒泡FIFO是简单的，对于小的缓存的资源要求低。 主要的缺点是n级冒泡FIFO是：(1)最大吞吐量是每周期两个时钟周期生成一个字， (2)一个字长需要旅行n时钟周期从写入端到读出端， (3)一个完整的FIFO需要n时钟周期重新开始

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.

这些缺点可以通过环形缓冲器 来解决。 环形缓冲可以通过存储器，读取和写入指针来解决。

Implement a FIFO as a circular buffer with four elements, using the same interface, and explore the different behavior with the tester. For an initial implementation of the circular buffer use, as a shortcut, a vector of registers (*Reg(Vec(4, UInt(size.W)))*).

补充一个FIFO作为一个环形缓冲使用四种元素，使用相同的界面，并探索测试器下不同的行文。 对于一个最初的环形缓冲器的做法，作为一个快捷方式， 是一个向量的寄存器(*Reg(Vec(4, UInt(size.W)))*)。

13.4.2 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:

```
$ 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 then configure the FPGA. After configuration, you should see a greeting message in the terminal.

对于UART的例子，你需要一个具有串口的FPGA板子和带有串口的电脑(经常是通过USB连接)。 在FPGA板和电脑连接串口线。开始一个terminal程序，例如，windows下的hyperterm， 或是linux下的gtkterm

```
$ gtkterm &
```

设置你的端口去选择正确设备，使用一个USB UART，这个经常是类似/dev/ttyUSB0。 把波特率设为114200，没有奇偶校验位或是控制流(握手信号)。 通过以下的命令行，你可以创造verilog代码用于UART:

```
$ make uart
```

然后使用你的综合工具去综合设计。 仓库包含了一个Quartus项目用于DE2-115的FPGA板子。 通过Quartus点击运行按钮去综合设计，然后设置FPGA。 经过设置，你应该在terminal看到一个欢迎的消息。

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.

拓展闪光LED的例子，使用一个UART，并去给串口线写0和1，当LED是关闭或是开启的时候。 使用BufferedTx，作为Sender的例子。

With the slow output of characters (two per second), you can write the data to the UART transmit register and can ignore the read/valid handshake. Extend the example by writing repeated numbers 0-9 as fast as the baud rate allows. In this case, you have to extend your state machine to poll the UART status to check if the transmit buffer is free.

有了字符的缓慢输出（每秒钟2个），你可以给UART传输寄存器写入数据，并且可以忽略read/valid握手信号。 通过重复写入数字0-9，尽波特率的最大允许速率，拓展这个例子。 在这个情况下，你需要拓展状态机去轮询UART状态，用来查看传输缓存是否空闲。

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.

这个例子代码包含了只有一个缓冲用于Tx。欢迎添加你写好的FIFO，用于发送和接收的缓冲。

13.4.3 探索FIFO

Write a simple FIFO with 4 buffer elements in dedicated registers. Use 2-bit read and write counters, which can just just overflow. As a further simplification consider the situation when the read and write pointers are equal as empty FIFO. This means you can maximally store 3 elements. This simplification avoids the counter function from the example in Listing 13.15 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 form the pointer values alone. How much simpler is this design?

写一个简单的FIFO，具有4个缓冲元素，使用特定的寄存器。 使用2位读和写计数器，这个可能会刚刚溢出。作为更多的简化，考虑当写和读指针等于空FIFO的情况。这意味着你可以最多存储3个元素。这个简化避开了来自13.15的例子中的计数器函数会出现的处理空或是满状态，使用同一个指针值。我们不需要空或满的flag，因为这个可以通过指针值得到。这个设计可以有多简化?

The presented different FIFO designs have different design tradeoffs relative to 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 chisel-examples. Where are the sweet spots for FIFOs of 4 words, 16 words, and 256 words?

目前不同的FIFO设计有不同的折衷，相对于以下考虑：（1）最大吞吐量（2）穿透延迟（3）资源需求和（4）最大时钟频率。探索所有的FIFO变化，通过在FPGA中综合观察面积不同；源代码在这里chisel-examples。 对于4字，16字，256个字，这些的最佳点在哪里?

Chapter 14

设计一个处理器

As one of the last chapters in this book, we present a medium size 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 and is available in open source at `https://github.com/leros-dev/leros`.

作为本书的一个最后一章，我们讲述了一个中等大小的项目：设计，仿真和测试一个微处理器。为了使这个项目科管理，我们会设计一个简单的累加器机器。这个处理器被称为Leros，在开源 `https://github.com/leros-dev/leros` 可以使用。

We would like to mention that this is an advanced example and some computer architecture knowledge is needed to follow the presented code examples.

我们想要说这是一个高阶的例子，一些计算机架构需要遵守所示代码例子。

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. In that table 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.

Leros被设计为一个简单的，但是仍配备一个C编译器。该指令的描述占一页，见表格 14.1。在那个表格，A代表累加器，OC使程序技术其，i使一个立即数（0到255），Rn是一个寄存器n（0到255），o是一个相对于PC的分支偏置，并且AR是一个地址寄存器用于存储器访问。

14.1 从ALU开始

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 an *Enum* to represent the different operations of the ALU:

处理器的一个中心组成部分是逻辑运算单元，或者ALU。于是，我们从ALU和testbench开始编写。首先，我们定义一个Enum代表ALU的不同操作：

```
object Types {
  val nop :: add :: sub :: and :: or :: xor :: ld :: shr :: Nil = Enum(8)
}
```

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 in Rn
ldaddr	AR = A	Load address register AR with A
loadind	A = mem[AR+(i << 2)]	Load a word from memory into A
loadindbu	A = mem[AR+i]$_{7-0}$	Load a byte unsigned from memory into A
storeind	mem[AR+(i << 2)] = A	Store A into memory
storeindb	mem[AR+i]$_{7-0}$ = A	Store a byte 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	scall A	System call (simulation hook)

Table 14.1: Leros instruction set.

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 14.1 shows the ALU.

ALU一般有两个操作数输入（称为a和b），一个操作op（或是op码）作为输入用来选择函数，和一个输出y。 14.1表明ALU。

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.

我们首先定义一个短的输入名字。res的逻辑通过switch声明得到。于是，它得到一个默认值为0. switch声明列举了所有操作，并分别赋值。所有的操作直接对应到一个Chisel表达式。

In the end, we assign the result res to the ALU output y

结尾，我们给res结果赋值ALU输出y。

```
class Alu(size: Int) extends Module {
  val io = IO(new Bundle {
    val op = Input(UInt(3.W))
    val a = Input(SInt(size.W))
    val b = Input(SInt(size.W))
    val y = Output(SInt(size.W))
  })

  val op = io.op
  val a = io.a
  val b = io.b
  val res = WireDefault(0.S(size.W))

  switch(op) {
    is(add) {
      res := a + b
    }
    is(sub) {
      res := a - b
    }
    is(and) {
      res := a & b
    }
    is(or) {
      res := a | b
    }
    is(xor) {
      res := a ^ b
    }
    is (shr) {
      // the following does NOT result in an unsigned shift
      // res := (a.asUInt >> 1).asSInt
      // work around
      res := (a >> 1) & 0x7fffffff.S
    }
    is(ld) {
      res := b
    }
  }

  io.y := res
}
```

Listing 14.1: The Leros ALU

For the testing, we write the ALU function in plain Scala, as shown in Listing 14.2.
为了测试，我们使用普通Scala函数编写了ALU函数，如下所示14.2。

```
def alu(a: Int, b: Int, op: Int): Int = {

  op match {
    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 14.2: Leros ALU的Scala表示

While this duplication of hardware written in Chisel by a 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:
使用Scala把这个Chisel硬件重写一遍不会检测出spec的错误；它只是一些合理性检验。我们使用一些边角情况作为测试向量。
We test all functions with those values on both inputs: 我们把这些值赋予输入，测试了所有的函数：

```
def test(values: Seq[Int]) = {
  for (fun <- add to shr) {
    for (a <- values) {
      for (b <- values) {
        poke(dut.io.op, fun)
        poke(dut.io.a, a)
        poke(dut.io.b, b)
        step(1)
        expect(dut.io.y, alu(a, b, fun.toInt))
      }
    }
  }
}
```

Full, exhaustive testing for 32-bit arguments is not possible, which was the reason we selected some corner cases as input values. Beside testing against corner cases, it is also useful to test against random inputs:
全部，尽可能地去列举32位的参数是不可能的，这是为什么我们选取一些边角情况作为输入值。除了测试一些边角情况，测试一些随机值也是有用的：

```
val randArgs = Seq.fill(100)(scala.util.Random.nextInt)
test(randArgs)
```

You can run the tests within the Leros project with
你可以在Leros项目中运行测试

```
$ sbt "test:runMain leros.AluTester"
```

and shall produce a success message similar to:
应该会生成一个成功的消息像是如下：

```
[info] [0.001] SEED 1544507337402
test Alu Success: 70567 tests passed in 70572 cycles taking
3.845715 seconds
[info] [3.825] RAN 70567 CYCLES PASSED
```

14.2 译码指令

From the ALU, we work backward and implement the instruction decoder. 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 of Leros.

从ALU，我们向后退一步，补充指令译码器。 但是，首先，我们在它本身的scala类定义指令和一个*shared*的拓展包。 我们想要在leros的硬件，leros的汇编器，和leros的指令集模拟器之间分享编码常量。

```scala
package leros.shared {

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
  // ...
```

For the decode component, we define a *Bundle* for the output, which is later fed partially into the ALU.
对于译码单元，我们定义了一个*Bundle*用于输出， 这个以后用于传给ALU。

```scala
class DecodeOut extends Bundle {
  val ena = Bool()
  val func = UInt()
  val exit = Bool()
}
```

Decode takes as input an 8-bit opcode and delivers the decoded signals as output. Those driving signals are assigned a default value with WireInit.
译码单元采取8位操作码作为输入，并把译码后的信号作为输出。 这些驱动信号被赋予一个默认值，通过WireInit。

```scala
class Decode() extends Module {
  val io = IO(new Bundle {
    val din = Input(UInt(8.W))
    val dout = Output(new DecodeOut)
  })

  val f = WireDefault(nop)
  val imm = WireDefault(false.B)
  val ena = WireDefault(false.B)

  io.dout.exit := false.B
```

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.)
译码本身是一个大的跳转声明，根据指令对应的操作码（在Leros, 指令大多根据前8位）。

```
switch(io.din) {
  is(ADD.U) {
    f := add
    ena := true.B
  }
  is(ADDI.U) {
    f := add
    imm := true.B
    ena := true.B
  }
  is(SUB.U) {
    f := sub
    ena := true.B
  }
  is(SUBI.U) {
    f := sub
    imm := true.B
    ena := true.B
  }
  is(SHR.U) {
    f := shr
    ena := true.B
  }
```

14.3 汇编指令

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.

要为Leros编写程序，我们需要一个汇编程序。但是，首先 测试，我们可以对一些指令进行硬编码，然后将它们放入Scala数组中， 我们用来初始化指令存储器。

```
val prog = Array[Int](
  0x0903, // addi 0x3
  0x09ff, // -1
  0x0d02, // subi 2
```

```scala
    0x21ab, // ldi 0xab
    0x230f, // and 0x0f
    0x25c3, // or 0xc3
    0x0000
  )

  def getProgramFix() = prog
  //- end

  //- start leros_asm_call
  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)
  }
```

However, this is a very inefficient approach to test 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. For branch destinations, we need a symbol table, which we collect in a Map. A classic assembler runs in two passes: (1) collect the values for the symbol table and (2) assemble the program with the symbols collected in the first pass. Therefore, we call assemble twice with a parameter to indicate which pass it is. 但是，这个是一个很困难的方法去测试处理器。 使用一个表达性的语言，像是scala编写一个汇编器，不是一个大的项目。 于是，我们写一个简单的汇编器用于leros，使用了不到100行的代码。 我们定义一个函数getProgram，呼叫汇编器。 对于跳转的目的地，我们需要一个符号的表格，这个在Map中被收集。 一个经典的汇编器在两个步骤： (1)收集元素表格的值 (2)使用从第一步收集来的符号汇编程序。 于是，我们呼叫assemble两次，并使用一个参数去表明是哪个步骤。

```scala
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 reading in the source file[1] and defining two helper functions to

[1]This function does not actually read the source file, but for this discussion we can consider it as the reading function.

parse the two possible operands: (1) an integer constant (allowing decimal or hexadecimal notation) and (2) to read a register number.

assemble 函数从读取源文件开始[2]， 并定义两个辅助函数去传入两个可能的操作数： （1） 一个整型常量（允许十进制和十六进制表示） （2）读入寄存器数。

```scala
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 14.3 shows the core of the assembler for Leros. A Scala match expression covers the core of the assembly function.

14.3 表明了汇编器的主要部分。 scala的match表达式覆盖了函数核心部分。

```scala
  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)
      // ...
```

[2]这个函数实际上不读取源文件， 但是本次讨论中我们可以认为它读取函数

```
      case "" => // println("Empty line")
      case t: String => throw new Exception("Assembler error: unknown instruction: " + t)
      case _ => throw new Exception("Assembler error")
```

Listing 14.3: Leros汇编器主要部分

14.4 练习

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.

最后章节的练习作业是非常自由的。你在学习Chisel旅程的末尾，并且准备解决有趣的设计难题。

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.

其中一个选择是再次读本章节，并且读Leros repository的源代码，运行测试案例，破坏代码，观察测试失败。

Another option is to write your implementation of Leros. The implementation in the repository is just one possible organization of a pipeline. You could write a Chisel simulation version of Leros with just a single pipeline stage, or go crazy and superpipeline Leros for the highest possible clocking frequency.

另一个选择是简单写一个你对Leros的个人补充。 这个仓库的版本只是一种流水线的可能形式。 你可以写一个Chisel仿真版本的Keros使用只是一个单流水线，或者走向疯狂，尝试深度流水线的Leros，以达到最高的时钟频率的可能性。

A third option is to design your processor from scratch. Maybe the demonstration of how to build the Leros processor and the needed tools has convinced you that processor design and implementation is no magic art, but the engineering that can be very joyful.

第三个选择是从零设计你的处理器。 可能这个关于如何搭建Leros处理器的演示和所需要的工具已经让你感到信服： 这个处理器设计和补充不再是魔法，而可以是好玩的工程。

Chapter 15

贡献chisel

Chisel is an open-source project under constant development and improvement. Therefore, you can also contribute to the project. Here we describe how to set up your environment for Chisel library development and how to contribute to Chisel.

Chisel是一个开源项目，在不断地开发和改进。所以，你也可以对项目做贡献。这里我们描述了如何去设置你的Chisel library开发环境， 以及如何对Chisel做贡献。

15.1 安装开发环境

Chisel consists of several different repositories; all hosted at the freechips organization at GitHub.

Chisel包括了多个不同的仓库；所有在的freechips organization at GitHub这里。

Fork the repository, which you 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. In our example, we change chisel3, and the clone command for my local fork is:

分支你想要做贡献的仓库，到你的个人Github账户上， 你可以分支这个仓库，使用*Fork*按钮，在GitHub网页界面上。 然后从那个分支，clone你的repo的fork。在我们的例子，我们切换到 *chisel3*，clone命令到我的本地分支是：

```
$ git clone git@github.com:schoeberl/chisel3.git
```

To compile Chisel 3 and publish as a local library execute:

为了编译Chisel3，和发布作为本地library，执行：

```
$ 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.

注意到在本地发布library的版本的命令行，包含了SNAPSHOT。 如果你使用测试器，发布的版本和Chisel SNAPSHOT不兼容， 要把它chisel-tester 也克隆下来，并且本地发布。

To test your changes in Chisel, you probably also want to set up a Chisel project, e.g., by forking/-cloning an empty Chisel project, renaming it, and removing the .git folder from it.

为了测试你对chisel做出的改变，你可能也想要建立一个chisel对象， 例如，通过分支或是克隆空chisel项目, 重命名，并且移除来自其中的.git。

Change the build.sbt to reference the locally published version of Chisel. Furthermore, at the time of this writing, the head of Chisel source uses Scala 2.12, but Scala 2.12 has troubles with anonymous bundles. Therefore, you need to add the following Scala option: "-Xsource:2.11". The build.sbt should look similar to:

改变build.sbt去引用本地版本的chisel。 更多地，在我写作的时刻，chisel源的头部使用scala 2.12, 但是scala 2.12在使用 匿名捆束会出现问题。 于是，你需要添加以下在scala的设置："-Xsource:2.11"。 build.sbt应该看起来像是如下：

```
scalaVersion := "2.12.6"

scalacOptions := Seq("-Xsource:2.11")

resolvers ++= Seq(
  Resolver.sonatypeRepo("snapshots"),
  Resolver.sonatypeRepo("releases")
)

libraryDependencies +=
  "edu.berkeley.cs" %% "chisel3" % "3.2-SNAPSHOT"
libraryDependencies +=
  "edu.berkeley.cs" %% "chisel-iotesters" % "1.3-SNAPSHOT"
```

Compile your Chisel test application and take a close look if it picks up the local published version of the Chisel library (there is also a SNAPSHOT version published, so if, e.g., 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.)

编译你的chisel测试应用，并且仔细看一下，它是否采用本地发布的chisel library(也存在一个发布的SNAPSHOT版本， 所以如果，例如，scala版本在你的chisel library和你的应用代码是有所不同的， 它会从服务器采取SNAPSHOT版本，而不是你本地发布的library)。

See also some notes at the Chisel repo.

也看一下一些chisel笔记.

15.2 测试

When you change the Chisel library, you should run the Chisel tests. In an sbt based project, this is usually run with:

更改Chisel库时，应运行Chisel测试。 在基于sbt的项目中，通常使用以下命令运行：

```
$ sbt test
```

Furthermore, if you add functionality to Chisel, you should also provide tests for the new features.

此外，如果您向Chisel添加功能，还应该提供针对新功能的测试。

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 at GitHub on collaboration with pull requests. The Chisel group started to document contribution guidelines.

在chisel项目中，没有开发者直接对主仓库进行更新。一次更新，是通过pull request从library的分支发动更新请求的。对于更多的信息，参考github的文档collaboration with pull requests。chisel小组开始在contribution guidelines进行文档。

15.3 练习

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, e.g., a ? operator that delivers the lefthand side if it is different from 0 otherwise the righthand side. Sounds like a multiplexer, right? How many lines of code did you need to add?[1]

创造一个用于UInt类型的新操作符，把它写入chisel library，并写下一些使用和测试代码去探索新的操作符。它不一定是一个有用的操作符；只是无论是否有用都很好，例如，?操作符表示操作数的左手侧是否为零，否则就返回右手侧。听起来像是复用器，对吗？你需要添加多少行代码？[2]。

As simple as this was, please be not tempted to fork the Chisel project and add your little extensions. Changes and extension shall be coordinated with the main developers. This exercise was just a simple exercise to get you started.

尽管简单，请不要试图分支Chisel项目，并添加您的小扩展。变更和扩展应与 主要开发商。这项练习只是使您入门的简单练习。

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.

如果你变得有经验，你可以选取其中一个open issues并且试图去解决它。然后通过更新请求做出贡献。但是，可能你要先去关注chisel开发的样式，通过点击github的watch。这样你就会观察和对于pr的处理方式，在chisel开源项目中。

[1]A quick and dirty implementation needs just two lines of Scala code.

[2]一个快速和直接的补充只需要两行scala代码

Chapter 16

总结

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 to simple uses of Scala. A next logical step is to learn a few basics of Scala and apply them to your Chisel project.

本书介绍了使用 硬件构造语言Chisel。 我们已经看到了几种简单到中型的数字电路在Chisel中描述。 Chisel嵌入在Scala中，因此继承了强大的功能 Scala的抽象。 由于本书旨在作为介绍，因此我们限制了 我们的示例简单介绍了Scala的用法。 下一步的逻辑步骤是学习Scala的一些基础知识并将其应用 到您的Chisel项目中。

I would be happy to receive feedback on the book, as I will further improve it and will publish new editions. You can contact me at `mailto:masca@dtu.dk`, or with an issue request on the GitHub repository. I am also happily accepting pull requests for the book repository for any fixes and improvements.

能收到反馈，我会非常高兴的，并且我会进一步加强，并推出新版本。 你可以通过`mailto:masca@dtu.dk`联系我，或是在github提出issue。 我会很高兴接受书目录的pull request，用于任何的更正和改进。

I would be happy to receive feedback on the book, as I will further improve it and will publish new editions. You can contact me at `mailto:masca@dtu.dk`, or with an issue request on the GitHub repository. I am also happily accepting pull requests for the book repository for any fixes and improvements.

能收到反馈，我会非常高兴的，并且我会进一步加强，并推出新版本。 你可以通过`mailto:masca@dtu.dk`联系我，或是在github提出issue。 我会很高兴接受书目录的pull request，用于任何的更正和改进。

访问资源

This book is available in open source. The repository also contains slides for a Chisel course and all Chisel examples: `https://github.com/schoeberl/chisel-book`

这本书是开源的。 这个目录也包含了其它的chisel课程的演示内容，以及所有的chisel例子：
`https://github.com/schoeberl/chisel-book`

A collection of medium-sized examples, which most 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`

在这本书受到最多的引用的一系列中等大小的例子，也是开源的。这个系列也包含了用于各种不同的FPGA例子的项目：`https://github.com/schoeberl/chisel-examples`

Appendix A

chisel项目

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 are aware of that use Chisel and are in open source.

尚未在许多项目中使用Chisel。因此，开源的Chisel代码 学习语言和编码风格的情况很少。这里我们列出了几个项目 我们知道使用Chisel是开源的。

Rocket Chip is a RISC-V [?] processor-complex generator that comprises the Rocket microarchitecture and TileLink interconnect generators. Originally developed at UC Berkeley as the first chip-scale Chisel project [?], Rocket Chip is now commercially supported by SiFive.

Rocket Chip 是RISC-V [?] 复杂处理器生成器， 包括rocket微处理器和TileLink连接生成器。 最早在UC Berkeley被开发出来，作为芯片大小的Chisel项目 [?]， Rocket芯片现在被SiFive商业支持。

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.

Sodor 是一套RISC-V的搭建，教育专用。 它包含1，2，3和5级流水线的搭建。所有的处理器使用一个简单的暂存存储器，被指令读取，数据读取，和debug端口程序所共享。 Sodor主要是用来仿真。

Patmos is an implementation of a processor optimized for real-time systems [?]. The Patmos repository includes several multicore communication architectures, such as a time-predictable memory arbiter [?], a network-on-chip [?] a shared scratchpad memory with an ownership [?]. At the time of this writing, Patmos is still described in Chisel 2.

Patmos 是一个为实时系统而补充的处理器 [?]. Patmos仓库包括多核通信的架构，例如时间可预测的存储器arbiter [?] , 一个片上网络 [?] 一个有版权的共享网络的存储器 [?]. 在我写这本书的时候，Patmos仍然使用Chisel2进行描述。

FlexPRET is an implementation of a precision timed architecture [?]. FlexPRET implements the RISC-V instruction set and has been updated to Chisel 3.1.

FlexPRET 是一个实时系统架构的补充 [?]. FlexPRET补充了RISC-V指令集架构，并且升级到了Chisel3.1。

Lipsi is a tiny processor intended for utility functions on a system-on-chip [?]. 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 the productivity of Chisel/Scala. 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.

Lipsi 是一个小型处理器，用于工具性函数，在system-on-chip [?]。 作为Lipsi的代码库是很小的，它可以作为使用chisel设计处理器的起始点。Lipsi也可以作为chisel生产力的代表。它花费了我14个小时去使用chisel描述硬件， 在fpga上运行它，使用scala编写汇编，写了一个Lipsi指令集模拟器作为协仿真，并写了一些测试例子用于Lipsi汇编。

OpenSoC Fabric is an open-source NoC generator written in Chisel [?]. It is intended to provide a system-on-chip for large-scale design exploration. The NoC itself is a state-of-the-art design with wormhole routing, credits for flow control, and virtual channels. OpenSoC Fabric is still using Chisel 2.

OpenSoC Fabric 是一个开源NoC生成器，通过chisel [?]进行编写。 它是用来提供一个片上系统的大型设计探索。NoC本身是使用虫孔路由的当代设计，为控制流，和虚拟通道创造贡献。 OpenSoC Fabric仍然使用chisel2。

DANA is a neural network accelerator that integrates with the RISC-V Rocket processor using the Rocket Custom Coprocessor (RoCC) interface [?]. DANA supports inference and learning.

DANA 是一个神经网络加速器，整合了RISCV Rocket处理器， 使用Rocket Custom Coprocessor(RoCC)接口 [?]。DANA支持推理和学习。

Chiselwatt is an implementation of the POWER Open ISA. It includes instruction to run Micropython.

Chiselwatt 是一个POWER Open ISA的版本。它包括运行Micropython的指南。

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.

如果您知道使用Chisel的开源项目，请给我留言 因此我可以将其包含在本书的未来版本中。

Appendix B

Chisel 2

This book covers version 3 of Chisel. Moreover, Chisel 3 is recommended for new designs. However, there is still Chisel 2 code out in the wild, which has not yet been converted to Chisel 3. There is documentation available on how to convert a Chisel 2 project to Chisel 3:

本书涵盖了Chisel的第3版。此外，Chisel3建议用于新设计。 但是，仍然有大量的Chisel2代码，尚未转换 到Chisel3。有有关如何将Chisel2项目转换为Chisel2的文档。 Chisel3：

- Chisel2 vs. Chisel3 and
- Towards Chisel 3

However, you might get involved in a project that still uses Chisel 2, e.g., the Patmos [?] processor. Therefore, we provide here some information on Chisel 2 coding for those who have started with Chisel 3.

但是，你可能加入一个仍旧使用chisel2的项目， 例如，Patmos [?]处理器。 于是，我们在这里提供一些chisel2的信息，为这些是以chisel3为起点的。

First, all documentation on Chisel 2 has been removed from the web sites belonging to Chisel. We have rescued those PDF documents and put them on GitHub at `https://github.com/schoeberl/chisel2-doc`. You can use the Chisel 2 tutorial by switching to the Chisel 2 branch:

首先，所有chisel2的文档官网上被移除了。我们保留了这些文档，并把它放在github上，在`https://github.com/schoeberl/chisel2-doc`。 你可以使用chisel2教程，通过切换到chisel2的分支：

```
$ git clone https://github.com/ucb-bar/chisel-tutorial.git
$ cd chisel-tutorial
$ git checkout chisel2
```

The main visible difference between Chisel 3 and 2 are the definitions of constants, bundles for IO, wires, memories, and probably older forms of register definitions.

Chisel3和2之间的主要可见区别是 常量，用于IO的捆束，电线，存储器以及可能是较旧形式的寄存器 定义。

Chisel 2 constructs can be used, to some extent, in a Chisel 3 project by using the compatibility layer using as package Chisel instead of chisel3. However, using this compatibility layer should only be used in a transition phase. Therefore, we do not cover it here.

chisel2构建出的模块，在某种程度上，在chisel3项目中使用兼容层使用像是Chisel拓展包，而不是Chisel3拓展包。 无论如何，使用兼容层应该只用来转换的过度。于是，我们在这里不讨论。

Here are two examples of basic components, the same that have been presented for Chisel 3. A module containing combinational logic:

有两个基础构成的例子，在Chisel3的版本里同样被演示过。一个包含组合逻辑的模块：

```
import Chisel._

class Logic extends Module {
  val io = new Bundle {
    val a = UInt(INPUT, 1)
    val b = UInt(INPUT, 1)
    val c = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 1)
  }

  io.out := io.a & io.b | io.c
}
```

Note that the Bundle for the IO definition is *not* wrapped into an IO() class. Furthermore, the direction of the different IO ports is defined as part of the type definition, in this example as INPUT and OUTPUT as part of UInt. The width is given as the second parameter.

注意到这个Bundle用来定义IO是不被包裹进IO()类。更多的， 不同的IO端口的方向作为类型的定义， 在这个例子里作为INPUT和OUTPUT，作为UInt的一部分。宽度作为第二参数提供。

```

import Chisel._

class Register extends Module {
  val io = new Bundle {
    val in = UInt(INPUT, 8)
    val out = UInt(OUTPUT, 8)
  }

  val reg = Reg(init = UInt(0, 8))
  reg := io.in

  io.out := reg
}
```

Listing B.1: Chisel2的8位寄存器例子

Here you see a typical register definition with a reset value passed in as a UInt to the named parameter init. This form is still valid in Chisel 3, but the usage of RegInit and RegNext is recommended for new Chisel 3 designs. Note also here the constant definition of an 8-bit wide 0 as UInt(0, 8).

这里你看到一个伴随着复位值的典型寄存器定义，作为UInt在init下。这个形式在Chisel 3是仍旧有效的， 但是使用RegInit和RegNext是被推荐的，在新的Chisel3设计中。注意到这里，8位常量的0被定义为UInt(0, 8)。

Chisel based testing C++ code and Verilog code are generated by calling chiselMainTest and chiselMain. Both “main” functions take a String array for further parameters.

以Chisel为基础的C++和Verilog的代码，是通过chiselMainTest和chiselMain生成的。 两个“main”函数采用String作为更多的参数。

```
import Chisel._

class LogicTester(c: Logic) extends Tester(c) {

  poke(c.io.a, 1)
  poke(c.io.b, 0)
  poke(c.io.c, 1)
  step(1)
  expect(c.io.out, 1)
}

object LogicTester {
  def main(args: Array[String]): Unit = {
    chiselMainTest(Array("--genHarness", "--test",
      "--backend", "c",
      "--compile", "--targetDir", "generated"),
      () => Module(new Logic())) {
        c => new LogicTester(c)
      }
  }
}
```

```
import Chisel._

object LogicHardware {
  def main(args: Array[String]): Unit = {
    chiselMain(Array("--backend", "v"), () => Module(new Logic()))
  }
}
```

A memory with sequential registered read and write ports is defined in Chisel 2 as:

一个拥有时序寄存器的读写寄存器端口，在Chisel2被定义为：

```
  val mem = Mem(UInt(width = 8), 256, seqRead = true)
  val rdData = mem(Reg(next = rdAddr))
  when(wrEna) {
    mem(wrAddr) := wrData
  }
```

Appendix C

简称

Hardware designers and computer engineers like to use acronyms. However, it needs time to get used to them. Here is a list of common terms related to digital design and computer architecture.

硬件设计者和计算机工程师喜欢简称。但是，需要一些时间去适应。这是一个常用数字设计和计算机架构的术语列表。

ADC analog-to-digital converter 模拟-数字转换器

ALU arithmetic and logic unit 运算和逻辑单元

ASIC application-specific integrated circuit 专用集成电路

CFG control flow graph 控制流图

Chisel constructing hardware in a Scala embedded language 在Scala嵌入的语言建构硬件

CISC complex instruction set computer 复杂指令集计算机

CPI clock cycles per instruction 时钟周期每指令

CRC cyclic redundancy check 循环空余检查

DAC digital-to-analog converter 数字-模拟转换器

DFF D flip-flop, data flip-flop D触发器，数据触发器

DMA direct memory access 直接访问存储器

DRAM dynamic random access memory 动态随机访问存储器

EMC electromagnetic compatibility 电磁兼容性

ESD electrostatic discharge 电荷释放

FF flip-flop 触发器

FIFO first-in, first-out 先进先出

FPGA field-programmable gate array 可编程逻辑门

HDL	hardware description language	硬件描述语言
HLS	high-level synthesis	高阶语言综合
IC	instruction count	指令计数
IDE	integrated development environment	集成开发环境
ILP	instruction level parallelism	指令级别平行
IO	input/output	输入/输出
ISA	instruction set architecture	指令集架构
JDK	Java development kit	Java开发工具
JIT	just-In-time	及时
JVM	Java virtual machine	Java虚拟机
LC	logic cell	逻辑单元
LRU	least-recently used	最近不常用的
MMIO	memory-mapped IO	存储器映射的输入输出
MUX	multiplexer	复用器
OO	object oriented	面向对象的
OOO	out-of order	乱序
OS	operating system	操作系统
RISC	reduced instruction set computer	精简指令集计算机
SDRAM	synchronous DRAM	同步DRAM
SRAM	static random access memory	静态随机访问存储器
TOS	top-of stack	栈顶部
UART	universal asynchronous receiver/transmitter	集总异步收发
VHDL	VHSIC hardware description language	超速硬件描述语言
VHSIC	very high speed integrated circuit	超快集成电路
WCET	Worst-Case Execution Time	最坏执行时间