使用Chisel做数字设计

Martin Schoeberl

February 14, 2019

These lecture notes, which shall become a book, are an introduction into digital design with the focus on using the hardware construction language Chisel. The approach of this book is to present small to medium sized typical hardware components to explore hardware design with Chisel. We will walk through these examples in detail.

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。

Contents

1 Introduction

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, to enable 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 is targeting two groups of developers: (1) hardware designers fluid in VHDL or Verilog using other languages such as Python, Java, or TCL/TK to generate hardware to move to a single hardware construction language where hardware generation is part of the language and (2) software programmers who are becoming interested in hardware design, e.g., as future chips from Intel will include programmable hardware to speed up programs.

这本书和Chisel的目标群体是两种开发者：

（1）硬件设计者，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。

TODO: Why Chisel

Hardware is now commonly described with a hardware description language. The time of drawing hardware components, even with CAD tools, is definitely 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 a lot of legacies, and have a moving line of what constructions 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 in 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。不要把我理解错了：VHDL和Verilog可以完美描述用来综合成ASIC的硬件部分。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 basics, such as how to build a gate out of CMOS transistors, refer to other digital design books. However, the intention of this book is to teach digital design at an abstraction level that is current practice to describe ASICs or designs targeting FPGAs.[1] As prerequisites for this book we assume basic knowledge of Boolean algebra and the binary number system. Furthermore, some programming experience in any programming language is assumed. No knowledge of Verilog or VHDL is needed. Chisel can be and shall 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.

这本书不是一个泛泛而谈的硬件设计和基础导论。对于一个基础的导论，像是如何使用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

[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.

行数字设计，作为当今描述ASIC或是设计FPGA的例子。作为这本书的前置需求，我们假设你有一些基本的布尔逻辑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（译者认为你最好还是会一些）。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。

TODO: Think about: Chisel is not a large (big?) language, but this is good for hardware description. The basic constructs can be learned/ fit on a page.... The power of the Chisel ecosystem comes from being embedded in Scala when programming hardware generators. Check also Rust book for structure, but better the Scala book.

Chisel itself is not a big language, as it shall be for hardware design. Therefore, this book is not a big book as well. Chisel is for sure smaller than VHDL and Verilog, which carry a lot of legacies. The power of Chisel comes from the embedding of Chisel within Scala, which is itself an expressive language. However, Scala is not the topic of this book. We provide a short section on Scala for hardware designers, but a general introduction to Scala is better served by the textbook by Odersky et al. [**?**].

Chisel用来硬件设计，本身不是一个大的语言。于是，这本书也不是一本大书。Chisel应该是比VHDL和Verilog更小, VHDL/Verilog有很多规则。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。但是，Scala不是本书的话题。我们提供了一个短的章节给硬件设计者，但是Odersky提供了一个更好的Scala导论。

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件描述风格的规则。

This book is optimized for reading on a laptop or on a 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，大多数是维基的文章。

1.1 Installing Chisel and FPGA Tools

Chisel is a Scala library and the easiest way to install Chisel and Scala is with `sbt`, the Scala build tool.

On Mac OS X, with the packet manager Homebrew, `sbt` is installed with:

```
$ brew install sbt
```

For Ubuntu, which is based on Debian, software is 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:

```
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
```

Chisel and Scala can also be installed and used under Windows. For instructions see: Installing sbt on Windows. *TODO: Test Windows version*

To build hardware for an FPGA you need a synthesize tool. The two major FPGA vendors, Intel[2] 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.

1.2 Hello World

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

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

Compiling and executing this short program with `sbt`

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

leads to the expected output of a Hello World program:

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

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

1.3 Chisel Hello World

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

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

1.4 An IDE for Chisel

This book makes no assumptions about your programming environment or editor 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 example here the Unix `ls` command, which lists files of the current folder:

```
$ ls
```

[2] former Altera

```
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
}
```

Figure 1.1: A hardware Hello World in Chisel

That said, an integrated development environment (IDE) with a compiler running in the background can speedup coding. As Chisel is basically a Scala library, all IDEs that support Scala are also good IDEs for Chisel. It is possible in Eclipse and IntelliJ to generate a project from the sbt project configuration in `build.sbt`. You can create an Eclipse project via

```
$ sbt eclipse
```

and import that project in Eclipse. In IntelliJ you can create a new project from existing sources and then select from `build.sbt`.

1.5 Source Access

This book is open source and hosted at GitHub: chisel-book. Small Chisel code snippets are included in the repository of the book. Larger examples are collected 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.

1.6 Further Reading

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

- Digital Design: A Systems Approach, by William J. Dally and R. Curtis Harting, is a modern textbook on digital design.

The official Chisel documentation and further documents are available online:

- Chisel home page, the official starting point to download and learn Chisel.
- The Chisel Tutorial provides a ready setup project containing small exercises with testers and solutions.
- The Chisel Wiki contains a short users guide to Chisel and links to further information.

- The Chisel Testers are in their own repository that contains a Wiki documentation.
- The Generator Bootcamp is a Chisel course focusing on hardware generators, as a Jupyter notebook
- A Chisel Style Guide by Christopher Celio.

1.7 Exercises

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

```
$ 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 to not need to deal with these low-level details. Chisel provides register components and those are connected automatically to `clock` and `reset` (if needed).

We also provide an additional Verilog top level in `verilog/hello_top.v`, which can be used to avoid connecting a reset signal (as FPGA registers usually are set to 0 on FPGA configuration).

The next step is to setup an FPGA project file for the synthesize tool, assign the pins, and compile[3] the Verilog code, and configure the FPGA with the resulting bitfile. We cannot provide the details of these steps. Please consult the manual of your Intel Quartus or 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 it happens that 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.

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

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.

As a more challenging extension to the exercise, generate 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?

[3] 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.

2 Basic Circuits

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。

2.1 Signal Types and Constants

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 different types, an 8-bit `Bits`, 8-bit unsigned integer, and an 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，UInts，和SInt。UInt和SInt是Bits的拓展，并且所有的三种类型代表bits的矢量。UInts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。Chisel使用二补数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)
```

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类型定义。以下表示把scala的整型n转换成Chisel的宽度，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  // defines a UInt constant of 0
-3.S // defines a SInt constant of -3
```

Constants can also be defined with a width, by using the Chisel width type:
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) // An 4-bit constant of 8
```

When 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.

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

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。

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

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

2.2 Combinational Circuits

Chisel uses Boolean algebra operators, as they are defined in C, Java, Scala, and probably several other programming languages, to described combinational circuits. 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使用布尔逻辑操作符，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进行与逻辑，然后把它的结果和c进行或逻辑的电路：

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

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

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

```
val and = a & b // bitwise and
```

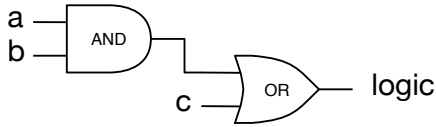

Figure 2.1: Logic for the expression `a & b | c`. The wires can be single bit or multiple bits. The Chisel expression, and the schematics, are the same.

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

在这个例子中我们没有定义类型也没有定义信号logic的宽度。这两个从表达式的type和width中推断而来。标准Chisel的逻辑操作是： 1. 按位与 2. 按位或 3. 按位异或 4. 按位取反（译者：这个和！有区别的，注意）

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.[2]

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。

The full list of operators can be found at builtin operators.

完整的操作列表可以在操作表格看到。

2.2.1 Multiplexer

A multiplexer is a circuit that selects between alternatives. In the most basic form it selects between two alternatives. Figure 2.2 shows such a 2:1 multiplexer, or mux for short. Depending on the value of the select signal (`sel`) signal `y` will represent signal `a` or signale `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 simply built from logic. However, as multiplexing is such a common operation, Chisel provides a 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.

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

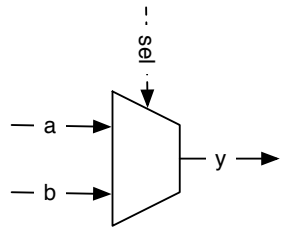

Figure 2.2: A basic 2:1 multiplexor.

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。

2.3 Register for State

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型触发器的集合。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 regBoth = RegNext(d, 0.U)
```

To distinguish from signals representing combinational logic and registers, a common practice is to prefix 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ref。另一个常见的方式，来自Java和Scala，市区使用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。

2.3.1 Counting

Counting is a very basic 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 in 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。

If we want to count N, we start with 0 and count till N-1.
如果我们想要数到N的时候，我们从0开始数到N-1。

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

cntReg := Mux(cntReg === 100.U, 0.U, cntReg + 1.U)
```

2.4 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 the 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）。

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. This 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 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。

3 Build Process and Testing

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

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 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。

3.1 Building your Project with sbt

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，在飞机上进行硬件设计，酷。

3.1.1 Source Organization

`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。

Figure **??** 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`. Optional also 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`.

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

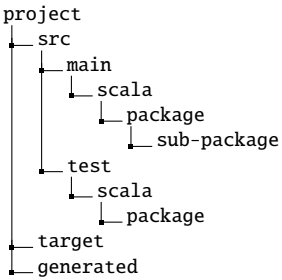

Figure 3.1: Source tree of a Chisel project (using `sbt`)

图3.1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mypack:

```
package mypack

import chisel3._

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

Note that in this example we see the import of the `chisel3` 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

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莫款在一个不同的地方（软件包命名空间），软件包的mypack需要被引用。下划线（ψ*ES*ϒ(*WC*fiffl*s*@@Ψ*mypack*–˜(Θ

```
import mypack._

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

  val abc = new Abc()

}
```

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

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()

}
```

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

}
```

3.1.2 Running sbt

A Chisel project can be compiled and executed with a simple `sbt` command:
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 contains a `main` method, or simpler that extends `App`. If there is more than one such object, all objects are listed and can be selected. You can also directly specify the object that shall be executed as 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.[2] 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:
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。

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

3.2 Testing with Chisel

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

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。

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 simply 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 to 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 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 simulation is advanced with one step (= one clock cycle) with `step(1)`. We can print the values of the outputs with `println()`.

The test is created and run with following tester main:

```
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):

```
[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 a fine 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()`:

```
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.

```
[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 contain an error, produces an error message describing the difference between the expected and actual value. In the following we changed the test bench to expect a 4, which is an error:

```
[info] [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. We will show these possibilities later.

3.2.1 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 following line:

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

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

```
$ sbt test
```

A minimal test (a testing hello world) to just test a Scala Integer addition:

```
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:

```
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:

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

3.3 Exercises

For this exercise we will revisit the blinking LED from chisel-examples and explore the ALU example.

3.3.1 A Minimal Project

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

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

```
import chisel3._
```

Then follows the hardware description, as shown in Figure 1.1. To generate the Verilog description we need an application. A Scala object that `extends APP` is an application that implicitly generates a 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 `HelloWorld.v`.

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

Run the generation of the example manually with

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

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

```
scalaVersion := "2.11.7"

resolvers ++= Seq(
  Resolver.sonatypeRepo("snapshots"),
  Resolver.sonatypeRepo("releases")
)

libraryDependencies += "edu.berkeley.cs" %% "chisel3" % "3.1.2"
```

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

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

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

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

```
make
```

The project also contains a Verilog top level, which wires the reset signal to 0. This works for our example in an FPGA, as registers usually power up 0. 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.

Note that the *Hello World* is the very minimal Chisel project. More realistic projects have their source files

organized and contain testers. The next exercise will explore such a project.

3.3.2 A Testing Exercise

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

Testing within Chisel can really speedup 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 decent pipelined RISC processor may consume about 300 4-bit LUTs and may run around 100 MHz in a low-cost FPGA (Intel Cyclone or Xilinx Spartan).

TODO: Do we need more details here?

3.3.3 A More Complete Project

For this exercise we will use the ALU example from chisel-examples.

TODO: This example uses already quite some Scala and the switch statement. We should probably postpone this for later.

4 Components

A larger digital design is structured into a set of components, often in a hierarchical way. Each component has an interface with input and output wires, usually called ports. These are similar to input and output pins in an IC. Components are connected by wiring up the inputs and outputs. Components may contain subcomponents to build the hierarchy.

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。

Figure **??** shows an example design. Component C has two input ports and two output ports. The component itself is assembled out of two subcomponents: B and C, wich 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.

图4.1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。

The outermost component, which is connected to physical pins in a chip, is called the top-level component.

最外层的组件，连接到芯片的物理引脚，称为上层组件。

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。

4.1 Components in Chisel are Modules

Hardware components are called modules in Chisel. Each module extends the class `Module` and contains a field `io` for the interface. The interface is defined by a `Bundle` that is wrapped into a call to `IO()`.

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。

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.

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

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

  // function of A
}

class CompB extends Module {
```

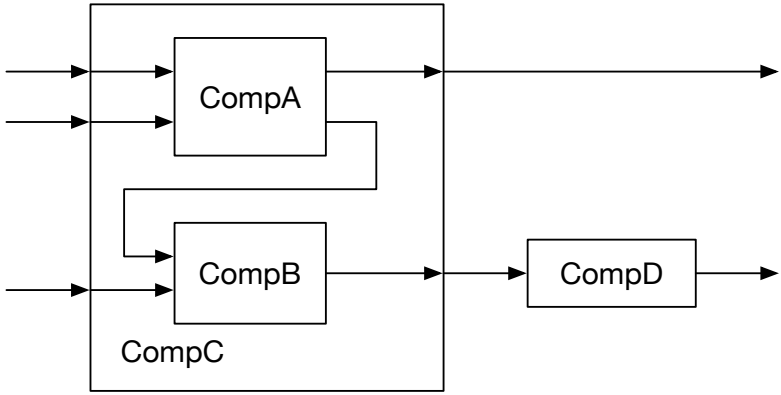

Figure 4.1: A design built of a a hierarchy of components.

```
  val io = IO(new Bundle {
    val in1 = Input(UInt(8.W))
    val in2 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  // function of B
}
```

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`. All ports use an unsigned integer (`UInt`) with a bit width of 8. As this example code is about connecting components and building a hierarchy, we do not show any implementation within the components. The implementation of the component is written at the place where the comments states "function of X". As we have no function associated with those example components, we used generic port names. For a real design use descriptive port names, such as `data`, `valid`, or `ready`.

Component 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 als the connection between an output port of A and an input port of B:

```
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())`.

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`.

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

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

  // function of D
}
```

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

```
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 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 very small components, such an adder, and very large components.

Beginners in hardware design often start with very tiny components. The problem is that design books use very small 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.

The interface to a component is a little bit verbose (with types, names, directions, IO construction). As a

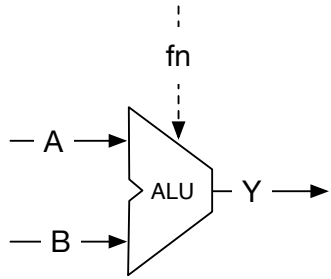

Figure 4.2: An arithmetic logic unit, or ALU for short.

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.

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

4.2 An Arithmetic Logic Unit

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

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.

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

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

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

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

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

```
}
```

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

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

4.3 Lightweight Components with Functions

Modules are the general way to structure your hardware description. However, there is some boiler plate when declaring a module and when instantiating and connecting it. A lightweight way to structure your hardware is to use functions. Scala functions can take Chisel (and Scala) parameters and return generated hardware. As a simple example we generate an adder:

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

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

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

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

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

By calling the function with the function itself as parameter, this generated a two clock cycle delay.

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

Again, this is a too short example to be useful, as `RegNext()` already is that function creating the register for the delay.

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 just collects utility functions.

5 Combinational Building Blocks

In this chapter we explore various combinational circuits, basic building block 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 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。

5.1 Combinational Circuits

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

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（a）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` and 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` has also 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
}
```

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

Chisel也支持一链条的条件（a if elseif else链条）通过.elsewhen

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

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

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.

记住.elsewhen中的"."在scala的链式方法是需要的。 这些".elsewhen"分支可以是任意长的。 但是，如果条件链条取决于一个单个的信号，最好用switch声明，这个会在下边的译码器电路介绍。

For more complex combinational circuits it might be practical to assign a default value to a `Wire`. This can be combined with the wire declaration with `WireDefault`.[1]

对于更复杂的组合电路，可能设给Wire一个默认值更加有效。

```
// TODO: change to WireDefault when 3.2 is out
val w = WireInit(0.U)

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

[1] In the current version of Chisel it is called WireInit, but will change with the release of Chisel 3.2

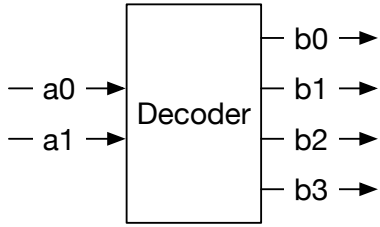

Figure 5.1: A 2-bit to 4-bit decoder.

a	b
00	0001
01	0010
10	0100
11	1000

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

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

有人可能会问，为什么使用when，.elsewhen, 和.otherwise，Scala用的是if，else if和else。 这些声明式对于Scala代码的条件执行，不是用于生成Chisel（复用器）的硬件。 这些Scala条件分支在Chisel中写电路生成器的时候有用处，它会采取变量，并条件化生成不同的硬件部分。

5.2 Decoder

A decoder converts a binary number of n bits to an m-bit signal, where $m \leq 2^n$. The output is one-hot encoded (where exactly one bit is one).

译码器把n位的二进制码转化位m位的信号，这里m小于等于$2^n\Theta/\Delta\Phi 1M1/1\Psi\Theta$

Figure **??** shows a 2-bit to 4-bit decoder. The function of the decoder can be described in a truth table, such as Table **??**.

图5.1表示了一个二位到四位的译码器，译码器的功能可以在真知表中被描述，像是表5.2。

The representation of logic as a truth table can be directly described in Chisel with a `switch` based table. The `switch` statement is not part of the core Chisel language. Therefore, we need to include the elements of the package `chisel.util`.

逻辑的表示像是真值表可以被直接描述成Chisel使用基于switch的表格。 switch声明不是Chisel语言core的部分。 于是，我们需要引用chisel.util的软件包

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

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

下边的代码引用Chisel的switch声明去描述一个译码器:

```
  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声明列出所有色sel信号的可能值，并把译码的值赋值到result信号。注意，即使我们列举了所有可能的输入值，Chisel仍旧需要我们去赋值一个默认值，我们把result设为0。这个赋值从来不会被激发，于是在后端工具被优化掉。这个是用来防止组合电路的非完全赋值的情况（在Chisel是Wire），这样会导致硬件描述语言像是VHDL和Verilog的不想要的锁存器。Chisel不允许非完全赋值。

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

在我们使用无符号整型的信号之前，一个可能更为清晰的编码器电路使用二进制表述：

```
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 examine the table, we see an 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 for 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。

5.3 Encoder

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

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。

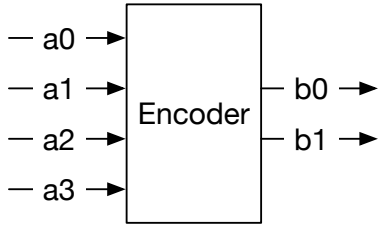

Figure 5.2: A 4-bit to 2-bit encoder.

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

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

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

图5.2表明了一个4位独热码输入到一个二进制输出的编码器，表5.3是编码函数的真值表。但是，编码器只在输入信号是独热码的时候工作。对于其它的输入值，输出是未定义的。所以我们不能描述一个有着未定义的函数作为输出，我们使用默认值赋值，抓取所有的未经定义的输入形式

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

下边的Chisel代码赋值默认00，然后使用switch声明作为合法输入。

```
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}
}
```

5.4 Exercise

Describe a combinational circuit to convert a 4-bit binary input to the encoding of a 7-segment display.

描述一个组合电路，用来转换4位二进制输入作为7位输出的编码。

You can either just 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.

6 Sequential Building Blocks

Sequential circuits a circuits where the output depends on the input *and* a previous value. 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。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。

6.1 Registers

The basic element to build 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. Or 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。

Figure **??** shows the schematic symbol of a register. I contains an input `D` and an output `Q`. Each register also contains in 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 a schematic. The little triangle on the bottom of the box symbolizes the clock input and tells us that this is a register.

图6.1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时钟信号作为输入。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。

In Chisel a registers 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, this is implicitly done by Chisel. 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。

[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.

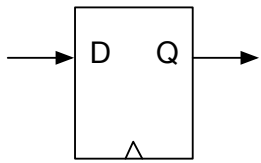

Figure 6.1: A D flip-flop based register.

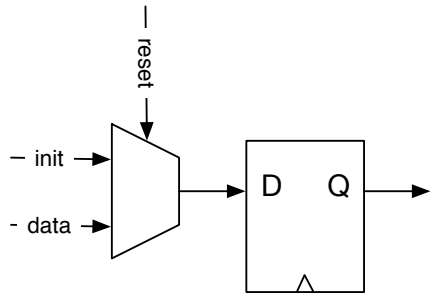

Figure 6.2: A D flip-flop based register with a synchronous reset.

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 the name of the register starting with `reg`. To easily distinguish elements between combinational circuits and sequential circuits it is common practice to prepend the marker `reg` 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
A register can also be initialized on reset. The reset signal is, like 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.

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

regVal := inVal
```

The default implementation of reset in Chisel is a synchronous reset.[2] 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.

Figure **??** 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.

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 waveform or timing diagram.

Figure **??** 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

[2] Support for asynchronous reset is currently under development

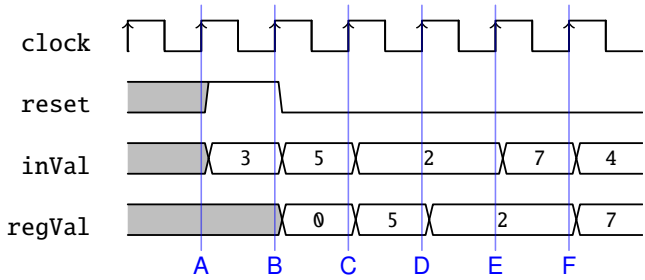

Figure 6.3: A waveform diagram for a register with a reset.

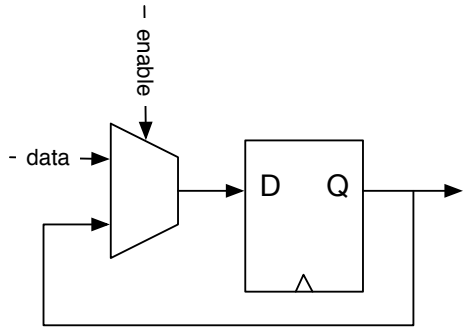

Figure 6.4: A D flip-flop based register with an enable signal.

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.

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

A common design pattern is a register with an enable signal. Only when the enable signal it `true` (high), the register captures the input, otherwise it keeps its old value. This enable behavior can be implemented, similar to the synchronous reset with a multiplexer at the input of the register, where one input is the feedback of the output of the register.

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

Figure **??** 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 forth clock cycle `enable` is low and the register keeps its value (5) at rising edge D.

Oh, and yes, a register with an enable can be described in a few lines of Chisel code with a conditional update:

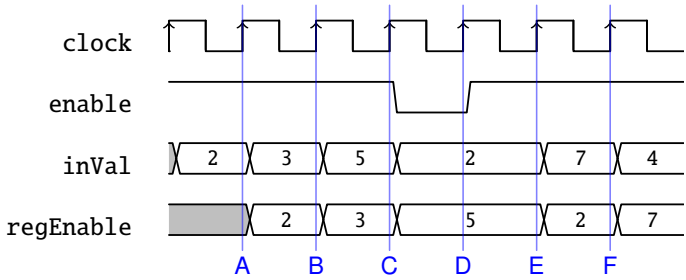

Figure 6.5: A waveform diagram for a register with an enable signal.

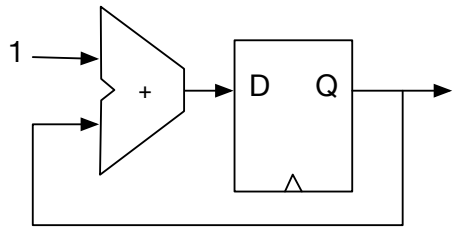

Figure 6.6: An adder and a register results in counter.

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

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

And a register with enable can also be reset.

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

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

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

6.2 Counters

One of the most basic sequential circuit 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 it connected to the input of the register. Figure **??** shows such a free-running counter.

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

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

regCnt := regCnt + 1.U
```

When we want to count events, we simply use a condition to increment the counter.

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

6.2.1 Generating Timing with Counters

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.

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 is exactly one original 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}.

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, `tick` is `true` for a single cycle and the counter is reset to `0`. When we count from `0` to `N - 1`, we generate one logical tick every `N` clock cycles.

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

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

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

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

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.

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

6.3 Stuff

TODO: Notes on what to describe: edge detector, also using a function (e.g., one has more inputs to detect

pressed buttons)

7 Example Components

In this section we explore some medium sizes digital designs, such as a FIFO buffer, which are used as building blocks for larger design. As an example we will design a serial interface (also called UART), which itself will 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。

7.1 FIFO Buffer

To decouple a write (sender) and a reader (receiver) some form of buffering between the writer and the reader is inserted. A common buffer is a first-in, first-out (FIFO) buffer. Figure **??** 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 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 and efficient implementation.

可能有一些不同形式对FIFO的补充：例如，使用片上存储，读取和写入指针或只是一连串的寄存器伴随小的状态寄存器。对于小的缓冲器（最多十多种元素），FIFO通过一些小的寄存器连接到一串缓冲是一个简单高效的写法。

We start by defining the IO signals for the writer and the reader. The size of the data is configurable with `size`.

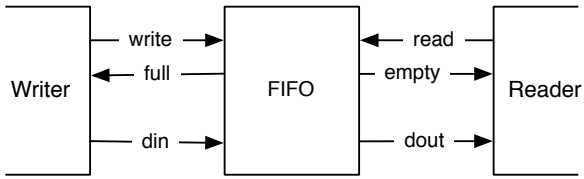

Figure 7.1: A writer, a FIFO buffer, and a reader.

我们从写入和读取的IO信号开始。数据大小可以通过size配置。

```
class WriterIO(size: Int) extends Bundle {
  val din = UInt(INPUT, size)
  val write = Bool(INPUT)
  val full = Bool(OUTPUT)
}

class ReaderIO(size: Int) extends Bundle {
  val dout = UInt(OUTPUT, size)
  val read = Bool(INPUT)
  val empty = Bool(OUTPUT)
}
```

Start on Better Counters

This is straightforward. However, one might ask if this is the most efficient implementation of counting. The comparison against an end value, 10 in our example, involves some combinational circuit in an ASIC. More efficient in an ASIC (and as well as in loops of software programs) is counting down and exit when 0 is reached.

这是直观的。但是，有人可能会问这是否是最有效的计数写法。比较末尾值，在我们的例子里是10，包括一些ASIC的组合电路。更有效的ASIC（并且在很多软件程序中）是向下数，到达0的时候退出。

If we want to count N, we start with N-1 and count till 0.

如果你想要从N开始数，我们从N-1开始，数到0.

TODO: down counting code

With a comparison against 0 we need only a NOR gat to detect the 0 condition. This optimization is valid for an ASIC, but not for an FPGA based on LUTs. With comparison using LUTs, there is no difference comparing against a '0' or '1' bit. In that case, use a better readable counting logic towards the end value.

通过和0比较，我们只需要一个NOR门电路去探测0的条件。这个优化对ASIC是有效的，但是对基于LUT（查找表）的FPGA来说无效。通过使用LUT进行比较，与"0"或"1"比较是没有区别的。那种情况下，使用可读性更好的数到末尾值的计数方式是更好的。

However, there is still one more trick a clever hardware designer can pull off. Counting up or down needed a comparison against all counting bis, so far. What about counting down *one too many*? In that case, the counter becomes negative. Detecting a negative value is simply comparing the most significant bit against 'a'.

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。

If we want to count N, we start with N-2 and count till -1.

如果我们想要数N，我们从N-2开始数到-1。

TODO: Show optimized down counting code.

TODO: discuss >= and so on

8 A Processor Design

As one of the last chapters in this book we present a medium size project: the design, simulation, and testing of a microprocessor. In order to keep this project manageable, we will 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。

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 **??**. 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 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8.1。 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。

8.1 Start with an ALU

A central component of a processor is the arithmetic logic unit, or ALU for short. Therefore, we start with coding of the ALU und 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)
}
```

An ALU has usually two operand inputs (call them `a` and `b`), an operation `op` (or opcode) input to select the function and an output `y`. Figure **??** 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。 图8.1表明ALU。

TODO: draw a nice ALU, see Wikipedia

We first define shorter names for the three inputs. The logic for `res` is computed with a `switch` statement. Therefore, it gets a default assignment of 0.

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.

The switch statement enumerates all operations and assigns the expression accordingly. All operations map directly to a Chisel expression. At the end we assign the result `res` to the ALU output `y`

For the testing we write the ALU function in plain Scala, as shown in Figure **??**.

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 test vector:

```
  // Some interesting corner cases
  val interesting = Array(1, 2, 4, 123, 0, -1, -2, 0x80000000, 0x7fffffff)
```

And testing all functions with those values on both inputs:

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 forth 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 8.1: Leros instruction set.

```
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 = WireInit(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
}
```

Figure 8.1: The Leros ALU.

```
  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
    }
  }
```

Figure 8.2: The Leros ALU function written in Scala.

```
  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:

```
  val randArgs = Seq.fill(100)(scala.util.Random.nextInt)
  test(randArgs)
```

The tests can be run within the Leros project with

```
$ sbt "test:runMain leros.AluTester"
```

and shall produce a success message similar to:

```
[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
```

8.2 Decoding Instructions

From the ALU we work backwards and implement the instruction decoder. However, first we define the instruction encoding in its own Scala class and in 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.

```
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
  // ...
```

TODO: Update code when Leros is more complete, as stuff is missing.

For the decode component we define a `Bundle` for the output, which is later fed partially into the ALU.

```
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`.

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

  val f = WireInit(nop)
  val imm = WireInit(false.B)
  val ena = WireInit(false.B)

  io.dout.exit := false.B
```

The decoding itself is just a large switch statement on part of the instruction that represents the opcode (in Leros for most instructions the upper 8 bits.)

```
  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
  }
  // ...
```

8.3 Assembling Instructions

To write programs for Leros we need an assembler. However, for the very first test we can hard code a few instructions, and put them into a Scala array, which we will use to initialize the instruction memory.

```
val prog = Array[Int](
  0x0903, // addi 0x3
  0x09ff, // -1
  0x0d02, // subi 2
  0x21ab, // ldi 0xab
  0x230f, // and 0x0f
  0x25c3, // or 0xc3
  0x0000
)

def getProgramFix() = prog
```

However, this is a very inefficient approach to 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.

```
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 parse the two possible operands: (1) an integer constant (allowing decimal or hexadecimal notation) and (2) to read a register number.

```
def assemble(prog: String, pass2: Boolean): Array[Int] = {

  val source = Source.fromFile(prog)
  var program = List[Int]()
  var pc = 0

  def toInt(s: String): Int = {
    if (s.startsWith("0x")) {
      Integer.parseInt(s.substring(2), 16)
    } else {
      Integer.parseInt(s)
    }
  }

  def regNumber(s: String): Int = {
    assert(s.startsWith("r"), "Register numbers shall start with \'r\'")
    s.substring(1).toInt
  }
```

Figure **??** shows the core of the assembler for Leros. The core of the assembly function is covered by a Scala `match` expression. *TODO: Some more words on the code.*

8.4 Exercise

This last chapter exercise assignment 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 read the chapter again and read along 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仓库的源代码，运行测试案例，破坏代码，观察测试失败。

Another option is to simply write your own 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 creasy 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 you own processor from scratch. Maybe the demonstration how to build the Leros processor and the needed tools has convinced you that processor design and implementation is no magic art, but 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。

[1] This function does not actually read the source file, but for this discussion we can consider it as the reading function.

```
for (line <- source.getLines()) {
  if (!pass2) println(line)
  val tokens = line.trim.split(" ")
  val Pattern = "(.*:)".r
  val instr = tokens(0) match {
    case "//" => // comment
    case Pattern(l) => if (!pass2) symbols += (l.substring(0, l.length - 1) ->
        pc)
    case "add" => (ADD << 8) + regNumber(tokens(1))
    case "sub" => (SUB << 8) + regNumber(tokens(1))
    case "and" => (AND << 8) + regNumber(tokens(1))
    case "or" => (OR << 8) + regNumber(tokens(1))
    case "xor" => (XOR << 8) + regNumber(tokens(1))
    case "load" => (LD << 8) + regNumber(tokens(1))
    case "addi" => (ADDI << 8) + toInt(tokens(1))
    case "subi" => (SUBI << 8) + toInt(tokens(1))
    case "andi" => (ANDI << 8) + toInt(tokens(1))
    case "ori" => (ORI << 8) + toInt(tokens(1))
    case "xori" => (XORI << 8) + toInt(tokens(1))
    case "shr" => (SHR << 8)
    // ...
    case "" => // println("Empty line")
    case t: String => throw new Exception("Assembler error: unknown
        instruction: " + t)
    case _ => throw new Exception("Assembler error")
```

Figure 8.3: The main part of the Leros assembler.

9 Contributing to 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 setup your environment for Chisel library development and how to contribute to Chisel.

9.1 Setup the Development Environment

Chisel consists of several different repositories, all hosted at the freechips organization at GitHub.

Fork the repository, which you like to contribute, into your personal GitHub account. This can be easily done by pressing the `Fork` button in the GitHub web interface. Then from that fork, clone your fork of the repository. In our example we will change `chisel3` and the clone command for my local fork is:

```
$ git clone git@github.com:schoeberl/chisel3.git
```

To compile Chisel 3 and publish as a local library execute:

```
$ cd chisel3
$ sbt compile
$ sbt publishLocal
```

Watch out during the publish local command for the version string of the published library, which contains the string `SNAPSHOT`. If you use the tester and the published version is not compatible with the Chisel SNAPSHOT, fork and clone the chisel-tester repo as well and publish it local.

To test your changes in Chisel, you probably also want to setup a Chisel project, e.g., by forking/cloning an empty Chisel project, renaming it, and removing the `.git` folder from it.

Change the `build.sbt` to reference the locally published version of Chisel. 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 following Scala option: `"-Xsource:2.11"`. The `build.sbt` should look similar to:

```
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 will pickup the SNAPSHOT version from the server instead of your local published library.)

See also some notes at the Chisel repo.

9.2 Testing

TODO: I need more details here from the Chisel developers on what and how to test.

When you change the Chisel library you should run the Chisel tests. In an `sbt` based project, this is usually run with:

```
$ sbt test
```

Furthermore, if you add functionality to Chisel you should also provide tests for the new features.

9.3 Contribute with a Pull Request

In the Chisel project no developer commits directly to the main repository. A contribution is organized via a pull request from a branch in a forked version of library. For further information see the documentation at GitHub on collaboration with pull requests. The Chisel group started to document contribution guide lines.

9.4 Exercise

Invent a new operator for the `UInt` type, implement it in the Chisel library, and write some usage/test code to explore the operator. It does not need to be a very useful operator, just anything, 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]

As simple as this was, please be not tempted to fork the Chisel project and add you little extensions. Changes and extension shall be coordinated with the main developers. This was just a simple exercise to get you started.

If you are getting bold, you could pick one of the open issues and try to solve it. Then contribute with a pull request to Chisel. However, probably first watch the style of development in Chisel by watching the GitHub repositories. See how changes and pull requests are handled in the Chisel open-source project.

[1] A quick a dirty implementation needs just two lines of Scala code.

10 Chisel 2

This book covers version 3 of Chisel. And 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 how to convert a Chisel 2 project to Chisel 3:

- 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 will provide here some information on Chisel 2 coding for those who have started with Chisel 3.

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:

```
$ 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.

Chisel 2 constructs can be used, to some extend, 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 will not cover it here.

Here are two examples of basic components, the same that have been presented for Chisel 3. A module containing combinational logic:

```
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 the IO definition is *not* wrapped into an `IO()` class. Furthermore, the direction of the different IO ports is defined as part of type definition, in this example as `INPUT` and `OUTPUT` as part of `UInt`. The width is given as the second parameter.

The 8-bit register example in Chisel 2:

```
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
}
```

Here you see a typical register definition with a reset value passed in as an `UInt` to the named parameter `init`. This form is still valid in Chisel 3, but 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)`.

Chisel based testing and Verilog code is generated by calling `chiselMainTest` and `chiselMain`. Both "main" functions take a `String` array for further parameters.

```
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:

```
  val mem = Mem(UInt(width = 8), 256, seqRead = true)
  val rdData = mem(Reg(next = rdAddr))
  when(wrEna) {
```

```
    mem(wrAddr) := wrData
  }
```

11 Summary

Acknowledgment

TODO: Sometimes we received some help. Sometimes external funding.

Source Access

This book is available in open source. The repository also contains slides for a Chisel course and all Chisel examples are included.

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 contains also projects for various popular FPGA boards.

https://github.com/schoeberl/chisel-examples

12 Headings

TODO: Collect the headings here, but only elevate them to chapter level when some writing is going on.

12.1 Introduction

* Why Chisel, what is cool about it * What this book is (and what not) * Overview of the following Chapters

12.2 Basic Circuits

* Combinational expressions (basic gates with logic tables) * Combinational base circuits (chapter 8 in Culler)
* Multiplexer (just the simple one for a start) * Registers
  1. base functions: +, - , and or, register (with reset, with enable)

12.3 Build Process and Testing

* A full example (blinking LED again) * Packages * Source organization (Scala) * Object to generate Verilog
* Testing * sbt

12.4 Components/Modules

12.5 Building Blocks

* building blocks (adder, mux, ALU, counter, memory) * maybe split into combinational and sequential

12.6 Bundles and Vecs (better title needed)

12.7 Medium Complex Circuits (better title needed)

3. small designs (better name): UART, FIFO, PWM, VGA, sigma delta

12.8 State Machines and Data Path

* Is this still relevant? Or is this simple old stuff not in real use anymore?

12.9 Memory

* Vec based ROM with address register * Vec based read/write * All other variations * Escape code in VHDL and Verilog for unsupported memories

12.10 Tips and Tricks (better title needed)

Stuff that saves a little hardware, but might not worse the less readable code
* Counter to -1 * Shared adder and subtractor * Mux with one hot encoding of select 5. little tricks: count down, add/sub

12.11 Scala for Hardware Developers

* Simple Scala (for, if else) * functions for hardware generation * Classes and constructor

12.12 More Complex Testing

* As we know now sone Scala it is time to use the power of Scala for better testing

12.13 Hardware Generation

* More advanced stuff * Table generation (sinus, assembler)

12.14 Leros

* as a more complex design 4. full design(s): processor

12.15 Chisel 2

* Some notes for reading Chisel 2 code * Update Lipsi to Chisel 3 to work on the 2 to 3 documentation

12.16 Chisel Projects

* Projects written in Chisel, as paper reference, some words and a URL.

12.17 Appendix

* Basic digital circuits, e.g., transistor based inverter, half and full adder * With lot of links to Wikipedia

Snippets and Collect Stuff to Describe

12.18 Minor Pitfalls

How to define a 32 bit unsigned constant of 0xffffffff?

Comparing a 32-bit UInt against a large literal results in error message.

val x = UInt(32.W) when (x === 0xFFFFFFFF.U) ...

0xffffffff.U does not work, as Int is a signed integer and the value is -1.

solutions are BigInt or Strings:

From email: However! You can do this with either a String or a BigInt. I think both of the following work:

val x = UInt(32.W) when (x === "hffffffff".U || x === BigInt("ffffffff", 16))

...

from Richard: You can also use a long, 0xffffffffL.U, though this will break at 64 bits. For the String-to-UInt conversion, x also works as a hex radix specifier, eg "xffffffff".U

TODO: Make an example and explain.

Is this below Chisel 2?

This chip was done in Chisel: https://aiyprojects.withgoogle.com/edge-tpu See also https://cloud.google.com/edge-tpu/

Concatenation and subfield access. Cat is in util.

TODO: A collection is Chisel snippets collected along some coding

Initialize a `Vec` from a Scala array:

```
val program = new Array[Bits](3)
program(0) = Bits(0x01, 8)
program(1) = Bits(0x23, 8)
program(2) = Bits(0x16, 8)

val rom = Vec(program)
```

State machine:

```
val fetch :: execute :: load :: Nil = Enum(UInt(), 3)
val stateReg = Reg(init = fetch)

switch(stateReg) {
  is(fetch) {
    stateReg := execute
  }
  is(execute) {
    when(isLoad) {
      stateReg := load
    }.otherwise {
      stateReg := fetch
    }
  }
  is(load) {
```

```
      stateReg := fetch
    }
  }
```

No default assignment possible when declaring a signal! Need an extra one.