Chisel 3.0 Tutorial (Beta)

Jonathan Bachrach, Krste Asanović, John Wawrzynek
EECS Department, UC Berkeley
{jrb|krste|johnw}@eecs.berkeley.edu

Translator: Sand River
liwei.ma@gmail.com

April 20, 2017

1 引言

本文档是 *Chisel* (Constructing Hardware In a Scala Embedded Language) 的介绍性教程。Chisel 是一种嵌入在高阶编程语言 Scala 中用来构造硬件的语言。在未来的某个时候我们将提供更适合的参考手册，引入更多的教程示例。在这之前，本文档虽然有一些尝试和错误，但也应该可以带你开始使用 Chisel。Chisel 实际上只是一组特殊的用 Scala 事先定义的类、对象和使用惯例，所以当写一份 Chisel 程序的时候，你实际上在写一份 Scala 程序。我们会在讲述我们的 Chisel 示例的时候指明必要的 Scala 语法特性，根据本文档给出的材料应该就可以完成相当有意义的硬件设计。但是如果你还想体验和设计出简洁和高复用的代码，你会发现借用 Scala 语言潜在的优势非常重要。我们建议你翻阅一些优秀的 Scala 书籍以便在使用 Scala 编程的时候更加专业。

Chisel 现在还处于早期阶段，你有可能会遇到一些实现缺陷，甚至有可能遇到方案设计问题。然而我们正在积极的修正和改进这种语言，欢迎提交缺陷报告和改进建议。即使在它的早期阶段，我们希望 Chisel 可以帮助设计者在构建容易复用和维护的设计中提高生产力。

在本教程中这样版式的段落里，我们评述设计该语言时的各种选择。你可以跳过这些评述章节而完全不会影响理解如何使用 *Chisel*，但是我们希望你能发现这些章节的有趣之处。

在数年的研究项目和硬件教学实践中一直和现有的硬件描述语言做斗争，我们非常有一种开发新的硬件语言的冲动。*Verilog* 和 *VHDL* 在设计之初只是硬件**模拟**语言，仅仅在后来才成为硬件**综合**的基石。这些语言的很多语法都不适合硬件综合，实际上很多语法概念完全不是可综合的。另一些语法概念在如何映射到硬件实现方面是非常不直观的，或者一不小心就会导致非常低效的电路结构。使用这些语言的一个子集生成合意的结果未尝不可，然而它们表达的规范模型毕竟是杂乱和混淆的，在教学过程中这些现象尤其明显。

而且，我们开发新的硬件语言的强烈冲动来自于我们对改变现有的电子系统的设计方法的渴望。我们坚信，教会学生如何设计电路固然重要，更加重要的是教会他们如何设计**电路生成器** — 能够根据一组高阶的设计参数和约束自动生成电路的程序。通过电路生成器，我们期望可以借用设计专家的才干提升每个工程师的设计抽象水平。为了表达灵活而可扩展的电路构造，电路生成器必须利用精致的编程技巧，在考虑如何更好的定制产生的电路的时候就可以做出更好的设计选择，以满足高阶的参数值和限制条件。虽然 *Verilog* 和 *VHDL* 也包括了一些构造原语可以实现电路的程序化生成，但还是缺少现代编程语言中的强大的设施，诸如面向对象的编程，类型推断，函数式编程和反省机制。

我们选择把硬件构造原语嵌入在一个现有的语言中，而不是从头构建一个新的硬件设计语言。我们选择 *Scala* 不仅因为它包括了一些编程特性，我们认为对构造电路生成器非常重要，而且因为它是专门为成为领域专用语言的基础而开发的。

2 Chisel 可表达的硬件

本版本的 Chisel 只支持二值逻辑，不支持三态信号。

我们聚焦在二值逻辑设计是因为其囊括实际应用中绝大部分的设计。在当前 *Chisel* 语言中我们省略了三态逻辑的支持，因为无论如何它在工业流

程中的支持都很差劲，在受控的硬件宏块外也很难被可靠的使用。

3 Chisel 的数据类型

Chisel 的数据类型用来表明在状态单元上保存或者信号线上流动的数值的类型。虽然硬件设计最终都是在操作二值数码的向量，然而更抽象的数值表示方法容许定义更清晰的规范，帮助工具生成更加优化的电路。在 Chisel 中，`Bits` 类型表示多个比特的简单集合。有符号和无符号整数是定点数字的子集，分别用 `SInt` 类型和 `UInt` 类型表示。有符号定点数字，包括整数，使用二进制补码形式表示。布尔数值用 `Bool` 类型表示。请注意这些类型明显区别于 Scala 内建的类型，比如 `Int` 和 `Boolean`。

常量和字面数值表示成 Scala 整数或者附有对应类型构造器的字符串：

```
1.U       // decimal 1-bit lit from Scala Int.
"ha".U    // hexadecimal 4-bit lit from string.
"o12".U   // octal 4-bit lit from string.
"b1010".U // binary 4-bit lit from string.

5.S    // signed decimal 4-bit lit from Scala Int.
-8.S   // negative decimal 4-bit lit from Scala Int.
5.U    // unsigned decimal 3-bit lit from Scala Int.

true.B // Bool lits from Scala lits.
false.B
```

Chisel 编译器默认使用最小的位宽来保存常量，有符号类型会包括一个符号比特。字面数字也可以显式地指明位宽，如下所示：

```
"ha".U(8.W)     // hexadecimal 8-bit lit of type UInt
"o12".U(6.W)    // octal 6-bit lit of type UInt
"b1010".U(12.W) // binary 12-bit lit of type UInt

5.S(7.W) // signed decimal 7-bit lit of type SInt
5.U(8.W) // unsigned decimal 8-bit lit of type UInt
```

`UInt` 类型的字面常量的数值用零扩展到目标位宽。`SInt` 类型的字面常量的数值用符号为扩展到目标位宽。如果给出的位宽太小不能保存参量数值，Chisel 会报错。

我们正在想办法设计更精确的 *Chisel* 字面常量的语法，比如使用符号化的前缀操作符。但是受阻于 *Scala* 操作符重载的限制，理想的更可读的语法还没能实现，现在只能使用字符串构造字面常量。

我们也考虑过允许 *Scala* 字面常量自动的转换到 *Chisel* 类型，但这会导致含混的语法，需要追加额外的导入。

`SInt` 和 `UInt` 类型以后会支持可选的指数域，这样 *Chisel* 就可以自动的产生优化的定点算术电路。

4 组合电路

电路在 Chisel 中表示成节点间的图。每个节点是一个硬件操作符，拥有零个或多个输入，驱动一个输出。前面所述的字面常量是退化的节点，没有输入，输出一个常量。一种常见的创建和连接节点方法便是使用文本表达式。例如，使用下面的表达式我们可以表达一个简单的组合逻辑电路：

```
(a & b) | (~c & d)
```

这些语法看起来应该很熟悉，`&` 表示按位与，`|` 表示按位或，`~` 表示按位非。从 `a` 到 `d` 的这些名称表示一定位宽（尚未指明）的有确定名称的信号线。

任何简单的表达式都可以直接转换成一个树状电路，命了名的信号线是叶子，操作符是内部节点。在树的根节点操作符上获得表达式最终电路的输出，这个操作符在本例中就是那个位或。

简单的表达式可以构造树形的电路，但是为了构造任意有向无环图 (DAG) 形状的电路，我们必须能够描述电路扇出。在 Chisel 中，为了实现扇出，我们可以用命名的信号线来捕获一个子表达式，然后在后续的表达式中多次引用这个信号线。在 Chisel 中，我们通过声明一个变量来命名一个信号线。例如，在下面这个多路选择描述中，这个选择表达式被使用了两次：

```
val sel = a | b
val out = (sel & in1) | (~sel & in0)
```

关键字 `val` 来源于 Scala，是给那些值不更改的变量命名的。这里被用来给 Chisel 的信号线命名，信号线 `sel` 捕获了第一个位或操作符的输出，便可以在第二个表达式中重复使用它。

5 内建操作符

Chisel 为内建数据类型定义了硬件操作符，如表 1 所示。

5.1 位宽推断

用户需要为端口和寄存器设置位宽，然而信号线上的位宽也可以被自动的推断出来，除非已经被用户手工

Example	Explanation
Bitwise operators. Valid on SInt, UInt, Bool.	
`val invertedX = ~x`	Bitwise NOT
`val hiBits = x & "h_ffff_0000".U`	Bitwise AND
`val flagsOut = flagsIn | overflow`	Bitwise OR
`val flagsOut = flagsIn ^ toggle`	Bitwise XOR
Bitwise reductions. Valid on SInt and UInt. Returns Bool.	
`val allSet = andR(x)`	AND reduction
`val anySet = orR(x)`	OR reduction
`val parity = xorR(x)`	XOR reduction
Equality comparison. Valid on SInt, UInt, and Bool. Returns Bool.	
`val equ = x === y`	Equality
`val neq = x =/= y`	Inequality
Shifts. Valid on SInt and UInt.	
`val twoToTheX = 1.S << x`	Logical left shift.
`val hiBits = x >> 16.U`	Right shift (logical on UInt and& arithmetic on SInt).
Bitfield manipulation. Valid on SInt, UInt, and Bool.	
`val xLSB = x(0)`	Extract single bit, LSB has index 0.
`val xTopNibble = x(15,12)`	Extract bit field from end to start bit position.
`val usDebt = Fill(3, "hA".U)`	Replicate a bit string multiple times.
`val float = Cat(sign,exponent,mantissa)`	Concatenates bit fields, with first argument on left.
Logical operations. Valid on Bools.	
`val sleep = !busy`	Logical NOT
`val hit = tagMatch && valid`	Logical AND
`val stall = src1busy || src2busy`	Logical OR
`val out = Mux(sel, inTrue, inFalse)`	Two-input mux where sel is a Bool
Arithmetic operations. Valid on Nums: SInt and UInt.	
`val sum = a + b`	Addition
`val diff = a - b`	Subtraction
`val prod = a * b`	Multiplication
`val div = a / b`	Division
`val mod = a % b`	Modulus
Arithmetic comparisons. Valid on Nums: SInt and UInt. Returns Bool.	
`val gt = a > b`	Greater than
`val gte = a >= b`	Greater than or equal
`val lt = a < b`	Less than
`val lte = a <= b`	Less than or equal

Table 1: Chisel operators on builtin data types.

设置。位宽推断引擎从图的输入端口开始，根据下面的规则集从节点相应的输入的位宽计算其输出的位宽。

operation	bit width
`z = x + y`	`wz = max(wx, wy)`
`z = x - y`	`wz = max(wx, wy)`
`z = x & y`	`wz = min(wx, wy)`
`z = Mux(c, x, y)`	`wz = max(wx, wy)`
`z = w * y`	`wz = wx + wy`
`z = x << n`	`wz = wx + maxNum(n)`
`z = x >> n`	`wz = wx - minNum(n)`
`z = Cat(x, y)`	`wz = wx + wy`
`z = Fill(n, x)`	`wz = wx * maxNum(n)`

这里以 wz 为例，是信号线 z 的位宽，并且 & 操作的推断规则适用于所有的按位逻辑操作。

位宽推断程序会一直进行到没有位宽再变化为止。除非操作是右移已知的常量位宽，位宽推断规则规定输出的位宽永远不会小于输入的位宽，因此输出的位宽跟输入相比或者增加或者保持。另外，用户必须显式地设置寄存器的位宽，或者指定为其复位数值的位宽，或者指定为其下一状态的相关参数。根据这两项规定，我们可以认定位宽推断程序最终会收敛到一个定点数。

由于操作符名字的选择受限于 *Scala* 语言，我们不得不使用三个等号 `===` 来表示相等，用 `=/=` 来表示不相等。这样 *Scala* 原有的判断相等的操作符可以继续按原意使用。

我们也在考虑增加新的操作符，可以强迫输出位宽等于两个输入的较大的位宽。

6 函数式抽象

我们可以把重复使用的一段逻辑定义成函数，然后在一个设计中复用多次。例如，我们可以把前面例子中的那个简单的组合逻辑块包装成如下的函数：

```
def clb(a: UInt, b: UInt, c: UInt, d: UInt): UInt =
  (a & b) | (~c & d)
```

此处 `clb` 就是那个函数，它输入 `a, b, c, d` 作为参数，返回一个信号线，这个信号线连接在一个布尔电路的输出信号上。关键字 `def` 来自于 Scala 语言，用来定义一个函数。在函数定义中，每个参数跟着它的数据类型，用冒号隔开，在参数列表的后面跟着函数的返回类型，同样也是用冒号隔开。等号 (=) 分隔开函数的参数列表和它的定义段落。

这样我们就可以像这样在其他电路中使用这个函数块了：

```
val out = clb(a,b,c,d)
```

在后文中，我们将讲述运用 Scala 的函数式编程的支持使用函数来构造硬件的各种方法。

7 绑裹和向量

绑裹类型（`Bundle`）和向量类型（`Vec`）是事先定义的 Scala 类，用户可以使用它们通过聚合其他的数据类型来扩展 Chisel 的数据类型。绑裹类型把若干命名的可以是不同类型的的域集合在一起变成一个连贯清晰的单元，这非常像 C 语言中的 `struct`。用户通过从 `Bundle` 衍生一个子类就可以定义自己的绑裹类型：

```
class MyFloat extends Bundle {
  val sign        = Bool()
  val exponent    = UInt(8.W)
  val significand = UInt(23.W)
}

val x  = new MyFloat()
val xs = x.sign
```

Scala 语言有一个传统，新定义的类的名字需要大写首字母，我们建议你在 Chisel 语言中也遵守这个传统。这里的 `W` 是事先定义的一个方法，把 Scala 的 `Int` 转换成 Chisel 的 `Width`，用以指明数据类型的位宽。

向量类型创建了一组可以索引的元素，可依照如下示例构建：

```
// Vector of 5 23-bit signed integers.
val myVec = Vec(5, SInt(23.W))

// Connect to one element of vector.
val reg3  = myVec(3)
```

（请注意，在小括号内我们必须指明向量 `Vec` 元素的数据类型，所以我们必须传递位宽参数给的 `SInt` 构造器。）

数据类型的基础类（`SInt`, `UInt` 和 `Bool`）和聚合类（`Bundles` 和 `Vecs`）都是从共同的超类 `Data` 继承而来的。每一个从 `Data` 继承而来的对象在硬件设计中最终都可以表示成比特向量。

绑裹类型和向量类型可以任意相互嵌套构造复杂的数据结构：

```
class BigBundle extends Bundle {
 // Vector of 5 23-bit signed integers.
 val myVec = Vec(5, SInt(23.W))
 val flag  = Bool()
 // Previously defined bundle.
 val f     = new MyFloat()
}
```

请注意，在创建一个实例的时候，Chisel 内建基础类和聚合类不需要使用 new 关键字，但是用户自定义的数据类型就必须使用。使用 Scala apply 构造器，用户自定义的数据类型也可以省略 new 关键字，详见第 14 节。

8 端口

端口是连接硬件组件的接口。端口可以是任何一种属于 Data 类的对象，其成员被赋予了信号方向。

Chisel 提供了端口构造器，允许在对象构造的时候增加方向（输入或者输出）。这只需要很简单的在对象外边包裹上 Input() 或者 Output() 函数。

```
class Decoupled extends Bundle {
  val ready = Output(Bool())
  val data  = Input(UInt(32.W))
  val valid = Input(Bool())
}
```

这样定义的 Decoupled 是一种新的数据类型，在需要时可以用于模块接口或者有名称的信号线集合。

也可以在例化的时候指定一个对象的方向，例如：

```
class ScaleIO extends Bundle {
  val in    = new MyFloat().asInput
  val scale = new MyFloat().asInput
  val out   = new MyFloat().asOutput
}
```

这两个方法 asInput 和 asOutput 把要求的方向强制赋予数据对象的所有成员。

后面将介绍，通过把方向包裹在对象的声明表达式上，Chisel 可以提供强大的信号线连接构造方法。

9 模块

在定义生成电路的层级化结构方面，Chisel 的模块与 Verilog 的模块非常相似。后续的工具可以访问的层级化的模块命名空间以辅助功能调试和物理布局布线。用户定义的模块是一个类：

- 继承自 Module，
- 包含一个用函数 IO() 包裹的接口，并且保存在一个名为 io 的端口域内。
- 在它的构造器内连接它所有的子电路。

例如，考虑把你自己的双输入多路选择器定义成一个模块：

```
class Mux2 extends Module {
  val io = IO(new Bundle{
    val sel = Input(UInt(1.W))
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val out = Output(UInt(1.W))
  })
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```

模块的连线接口是一组端口绑裹在一起形成的 Bundle。接入模块的接口被命名成 io 的域。对这个 Mux2 模块而言，io 是包含四个域的一个绑裹，每一个域都是这个多路选择器的一个端口。

这里用在定义主体里的赋值操作符 := 是 Chisel 定义的特殊的操作符，连接右手边电路的输出到左手边电路的输入。

9.1 模块层级

用小的子模块构建大的模块，我们现在可以构造电路层级。连接三个双输入的多路选择器，我们可以用模块 Mux2 来建造四输入的多路选择器模块，例如：

```
class Mux4 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val in2 = Input(UInt(1.W))
    val in3 = Input(UInt(1.W))
    val sel = Input(UInt(2.W))
    val out = Output(UInt(1.W))
  })
  val m0 = Module(new Mux2())
  m0.io.sel := io.sel(0)
  m0.io.in0 := io.in0; m0.io.in1 := io.in1

  val m1 = Module(new Mux2())
  m1.io.sel := io.sel(0)
  m1.io.in0 := io.in2; m1.io.in1 := io.in3

  val m3 = Module(new Mux2())
  m3.io.sel := io.sel(1)
  m3.io.in0 := m0.io.out; m3.io.in1 := m1.io.out

  io.out := m3.io.out
}
```

同样地，我们定义了名为 io 的模块接口，并且连接输入和输出。使用 Module 的构造函数和 Scala 的 new 关键字可以创建一个模块对象，在本例中，我们这样创建了三个 Mux2 子模块。然后我们把它们相互连接并和 Mux4 接口里的端口连接起来。

10 运行示例

现在我们已经定义了模块，下面讨论如何运行和测试一个电路。

测试实则是电路设计中极其关键的组成部分，因而我们在 Chisel 中提供可强大的测试机制，可以在 Scala 中提供测试向量来测试电路。测试器被绑定到一个模块，用户可以使用如下的调试协议定义测试集，并通过调用测试器的方法来测试电路。可以特别指出的是，用户可以使用如下测试设施：

- 置数操作 (poke)：可以在输入端口和状态电路上放置数值，
- 单步操作 (step)：可以让电路往前运行一个时间单位，
- 窥探操作 (peek)：可以窥探端口和状态电路的数值，
- 期待操作 (expect)：可以用期望的参数来比较读取的电路数值。

Chisel 生成 Firrtl 中间表示 (IR)，Firrtl 可以直接解释执行，也可以转换成 Verilog，然后在用 Verilator 生成 C++ 仿真器。

例如，在下面的样本程序中，

```
class Mux2Tests(c: Mux2, b: Option[TesterBackend] = None)
     extends PeekPokeTester(c, _backend=b) {
  val n = pow(2, 3).toInt
  for (s <- 0 until 2) {
    for (i0 <- 0 until 2) {
      for (i1 <- 0 until 2) {
        poke(c.io.sel, s)
        poke(c.io.in1, i1)
        poke(c.io.in0, i0)
        step(1)
        expect(c.io.out, (if (s == 1) i1 else i0))
      }
    }
  }
}
```

使用 poke 方法，Mux2 的每个输入被赋予适当的数值。在这个特定的例子中，我们用强硬编码的方式设置输入到某些已知的数值，然后检查输出是否是对应的已知数值，这样就可以测试 Mux2 的功能。为了实现这些测试逻辑，在每次循环中我们产生被测模块需要的恰当的输入数值，然后告诉仿真把这些数值放在我们正在测试的设备 c 的输入上，向前运行电路一个时钟周期，然后用期望的结果来检查输出。单步运行将更新寄存器和寄存器驱动的组合电路，因而是非常必要的。对于纯组合电路路径而言，置数操作本身就足够更新所有与被置数的输入相连接的组合电路路径。

最后，通过调用函数 runPeekPokeTester 来激活一个测试器。

```
def main(args: Array[String]): Unit = {
  runPeekPokeTester(() => new GCD()){
    (c,b) => new GCDTests(c,b)}
}
```

这个例子将通过 Firrtl 解释器来仿真运行 GCDTests 定义关于 GCD 的测试集。通过下面的方法，我们也可以用 Verilator 生成 C++ 仿真器来仿真 GCD 模块：

```
def main(args: Array[String]): Unit = {
  runPeekPokeTester(() => new GCD(), "verilator"){
    (c,b) => new GCDTests(c,b)}
}
```

11 状态元件

Chisel 支持的最简单形式的状态元件就是正沿触发的寄存器，其例化方法如下：

```
val reg = Reg(next = in)
```

这个电路的输出是输入信号 in 被延时一个时钟周期的拷贝。请注意，我们并不需要指定 Reg 的数据类型，因为当用这种方式例化寄存器的时候，它的类型可以从输入那里自动推断出来。在 Chisel 当前的版本，时钟和复位信号是全局信号，会被自动插入到需要的地方。

使用寄存器，我们马上就可以定义实用的电路构造了。例如，上升沿检测器可以检测一个布尔信号的上升跳变，当输入信号的当前值为真而上一个时钟周期的值为假，检测器就会输出真。下面的电路给出了上升沿检测器的设计：

```
def risingedge(x: Bool) = x && !Reg(next = x)
```

计数器是非常重要的时序电路。一个向上计数的计数器，在数到最大值 max 后会返回到零（即对 max+1 取模），如下方法可以实现这样的计数器：

```
def counter(max: UInt) = {
  val x = Reg(init = 0.U(max.getWidth.W))
  x := Mux(x === max, 0.U, x + 1.U)
  x
}
```

计数寄存器在函数 counter 中被创建出来，复位值是 0（位宽恰好足够放入 max）。复位值是当电路全局的复位信号有效的时候寄存器被初始化的值。函数 counter 中的给 x 赋值的符号 := 把一个组合电路连接到寄存器为其更新状态。这个组合电路一直增加计数器的数值到 max 的时候才重新返回到零。请注意 x 出现在赋值语句的右手边时，调用的是它的输出端，而当它出现在左手边时，调用的是它的输入端。

应用计数器可以构建很多有用的时序电路。例如，我们可以构建一个脉冲发生器，在计数器返回到零的时候输出真：

```
// Produce pulse every n cycles.
def pulse(n: UInt) = counter(n - 1.U) === 0.U
```

利用在脉冲序列，反复在真和假之间反转，就可以构造方波发生器：

```
// Flip internal state when input true.
def toggle(p: Bool) = {
  val x = Reg(init = false.B)
  x := Mux(p, !x, x)
  x
}

// Square wave of a given period.
def squareWave(period: UInt) = toggle(pulse(period/2))
```

11.1 前向声明

纯组合电路在节点之间不能存在环，如果检测出组合逻辑环，Chisel 会报错。组合电路因为没有环，就一定可以用前馈的方式构造，新增加的节点的输入是前面定义好的节点输出。时序电路天然地存在节点之间的反馈，因此在有的时候需要在某些节点被定义之前引用它的输出信号线。因为 Scala 顺序执行程序语句，我们允许事先提供数据节点的声明，充当信号线，可以马上使用，但它的输入却在稍后给出。例如，在下面的简单的 CPU 中，我们需要事先声明 pcPlus4 和 brTarget 信号线，这样它们可以在定义之前被使用。

```
val pcPlus4  = UInt()
val brTarget = UInt()
val pcNext   = Mux(io.ctrl.pcSel, brTarget, pcPlus4)
val pcReg    = Reg(next = pcNext, init = 0.U(32.W))
pcPlus4     := pcReg + 4.U
...
brTarget    := addOut
```

在 pcReg 和 addOut 被定义之后，连线操作符 := 连接了相关的信号。

11.2 条件更新

在前面应用寄存器的例子中，我们只是简单地把组合逻辑块的输出连接到寄存器的输入。在描述状态元件的操作时，在分立的多个语句中指出寄存器需要更新的条件是非常方便的。Chisel 提供了条件更新规则来支持这种类型的时序逻辑描述，这就是 when 形式的构造语法。例如，

```
val r = Reg(init = 0.U(16.W))
when (cond) {
  r := r + 1.U
}
```

在这个例子中寄存器 r 只有当 cond 是 true 的时候才在当前时钟周期结束的时候更新状态。when 语句的输入参数是一个返回 Bool 类型的预测电路表达式。更新操作位于 when 后面的语句块中，只能包括操作符 := 表示的赋值语句，简单的表达式和用 val 命名的信号线定义。

在一串条件更新语句中，最后一个条件为真的更新取得优先权。例如

```
when (c1) { r := 1.U }
when (c2) { r := 2.U }
```

本例中寄存器 r 根据下面的真值表更新状态：

c1	c2	r	
0	0	r	r unchanged
0	1	2	
1	0	1	
1	1	2	c2 takes precedence over c1

图 1 表明每一个条件更新语句都可以被看作是在寄存器的输入前面插入了一个多路选择器，根据 when 语句的条件，或者选择更新表达式或者选择上一个输入。而且这些条件或在一起成为使能信号接入到寄存器的加载使能信号上。编译器在逻辑或门链的开始放置初始值，这样如果在本时钟周期内没有条件更新被激活，寄存器的加载使能端就被置为无效，寄存器的值也就不会发生改变。

Chisel 为条件更新的一些普遍的形式提供了一些语法糖。unless 语句与 when 语句类似，但是在它们生效的条件相反。换言之，

```
unless (c) { body }
```

等效于

```
when (!c) { body }
```

更新语句块可以操作多个寄存器，而且这组寄存器多个不同的相互重叠的子集可以出现在多个更新语

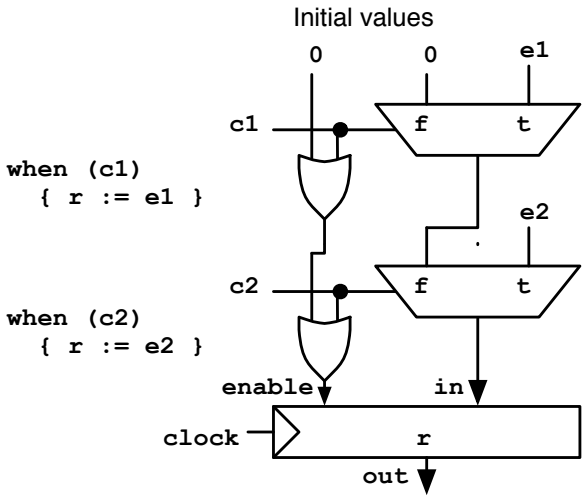

Figure 1: 本例中一串条件更新语句构造的等价电路。每一个 when 语句增加新的一级数据多路选择器，它们的条件用逻辑或门串在一起组成寄存器的使能信号。Chisel 编译器会在逻辑或门链的结尾自动加入终止数值。

句块中。只有包裹了寄存器更新语句块的条件才会影响这个寄存器的后续值。同样的原则也适用于组合逻辑电路中（更新一个 Wire 类型的信号线）。需要注意的是，所有组合电路的信号线需要一个默认的值。例如，

```
r := 3.S; s := 3.S
when (c1)   { r := 1.S; s := 1.S }
when (c2)   { r := 2.S }
```

这段代码中 r 和 s 根据如下的真值表更新状态：

```
c1 c2  r  s
0   0  3  3
0   1  2  3 // r updated in c2 block, s at top level.
1   0  1  1
1   1  2  1
```

我们也在考虑增加另一种条件更新的形式，只允许单个更新块发生作用。这些原子更新操作将会非常像 *Bluespec* 中的受保护的原子操作。

条件更新构造表达式块可以嵌套，某一表达式块在其所有嵌套在外边的条件都满足的时候才会被执行。例如

```
when (a) { when (b) { body } }
```

等效于

```
when (a && b) { body }
```

多个条件可以用 when, .elsewhen, .otherwise 串在一起，就像 Scala 语言中的 if, else if 和 else。

```
when (c1) { u1 }
.elsewhen (c2) { u2 }
.otherwise { ud }
```

等效于

```
when (c1) { u1 }
when (!c1 && c2) { u2 }
when (!(c1 || c2)) { ud }
```

当处理针对一个共同的键值一系列比较的时候，我们还为条件更新引入了 switch 语法。例如，

```
switch(idx) {
 is(v1) { u1 }
 is(v2) { u2 }
}
```

等效于：

```
when (idx === v1) { u1 }
.elsewhen (idx === v2) { u2 }
```

Wire 类型表示的是组合逻辑电路的输出信号线，Chisel 也允许条件更新语句式用于 Wire 类型的数据，这样复杂的组合逻辑表达式就可以递增地构建出来。Chisel 要求组合电路输出的更新条件必须是完全的，如果组合电路输出没有默认的更新规则，Chisel 会报错。

在 *Verilog* 中，如果组合逻辑块的程序性的规范是不完全的，一个锁存器就被静悄悄的产生了，这导致了大量的令人抓狂的缺陷。

为了确定一组更新条件是否覆盖了所有的可能性，在 *Chisel* 的编译器内加入更多的分析本是可行的。但是现在的版本中，我们需要在 Wire 类型的条件更新链条的一开始加入一个条件为真的默认更新规则。

11.3 有限状态机

在数字电路设计中，一种常见的时序电路类型就是状态机 (FSM)。奇偶发生器就是 FSM 的一个简单例子：

```
class Parity extends Module {
  val io = IO(new Bundle {
    val in  = Input(Bool())
    val out = Output(Bool()) })
  val s_even :: s_odd :: Nil = Enum(2)
  val state  = Reg(init = s_even)
  when (io.in) {
    when (state === s_even) { state := s_odd  }
    when (state === s_odd)  { state := s_even }
```

```
  }
  io.out := (state === s_odd)
}
```

这里 `Enum(2)` 生成两个 `UInt` 类型的字面常量。当输入信号 `in` 为真的时候，状态才会被更新。值得特别提及的是，FSM 所有的构造机制都建立在寄存器、信号线和条件更新之上。

下面给出了一个稍微复杂点儿的 FSM 例子，这是为自动售货机如何收钱而设计的一个电路：

```
class VendingMachine extends Module {
  val io = IO(new Bundle {
    val nickel = Input(Bool())
    val dime   = Input(Bool())
    val valid    = Output(Bool()) })
  val s_idle :: s_5 :: s_10 :: s_15 :: s_ok :: Nil =
      Enum(5)
  val state = Reg(init = s_idle)
  when (state === s_idle) {
    when (io.nickel) { state := s_5 }
    when (io.dime)   { state := s_10 }
  }
  when (state === s_5) {
    when (io.nickel) { state := s_10 }
    when (io.dime)   { state := s_15 }
  }
  when (state === s_10) {
    when (io.nickel) { state := s_15 }
    when (io.dime)   { state := s_ok }
  }
  when (state === s_15) {
    when (io.nickel) { state := s_ok }
    when (io.dime)   { state := s_ok }
  }
  when (state === s_ok) {
    state := s_idle
  }
  io.valid := (state === s_ok)
}
```

下面再给出用 `switch` 语句设计的自动售货机的 FSM：

```
class VendingMachine extends Module {
  val io = IO(new Bundle {
    val nickel = Input(Bool())
    val dime   = Input(Bool())
    val valid  = Output(Bool())
  })
  val s_idle :: s_5 :: s_10 :: s_15 :: s_ok :: Nil =
      Enum(5)
  val state = Reg(init = s_idle)

  switch (state) {
    is (s_idle) {
      when (io.nickel) { state := s_5 }
      when (io.dime) { state := s_10 }
    }
    is (s_5) {
      when (io.nickel) { state := s_10 }
      when (io.dime) { state := s_15 }
    }
    is (s_10) {
      when (io.nickel) { state := s_15 }
      when (io.dime) { state := s_ok }
    }
    is (s_15) {
      when (io.nickel) { state := s_ok }
      when (io.dime) { state := s_ok }
    }
    is (s_ok) {
      state := s_idle
    }
  }
  io.valid := (state === s_ok)
}
```

12 存储器

Chisel 提供了创建只读存储器和可读写存储器的机制。

12.1 只读存储器 (ROM)

用户可以使用 `Vec` 类型来定义只读存储器，如示例：

```
Vec(inits: Seq[T])
Vec(elt0: T, elts: T*)
```

这里 `inits` 是一串初始 `Data` 类型的字面常量，用来初始化 ROM。用户可以创建一个小 ROM，初始化成 `1, 2, 4, 8`，并且用一个计数器作为地址生成器来遍历它所有的值，如下例所示：

```
val m = Vec(Array(1.U, 2.U, 4.U, 8.U))
val r = m(counter(UInt(m.length.W)))
```

我们可以 ROM 创建一个有 `n` 个入口的 sine 函数的查找表，如下例所示：

```
def sinTable (amp: Double, n: Int) = {
  val times =
    Range(0, n, 1).map(i => (i*2*Pi)/(n.toDouble-1) - Pi)
  val inits =
    times.map(t => SInt(round(amp * sin(t)), width = 32))
  Vec(inits)
}
def sinWave (amp: Double, n: Int) =
  sinTable(amp, n)(counter(UInt(n.W))
```

这里的 `amp` 用来缩放 ROM 中保存的定点数值。

12.2 可读写存储器 (Mem)

可读写存储器有很多种硬件实现方法，比如，FPGA 的存储器就和 ASIC 的实例化方法就非常不同，因而 Chisel 给予存储器很特别的待遇。Chisel 定义了一种抽象的存储器，既可以映射到简单的 Verilog 行为描述，也可以映射到代工厂或 IP 供货商给出的存储器模块的实例。

Chisel 通过 `Mem` 构造器来支持随机访问的存储器。写入存储器是正沿触发的行为，读取存储器可以是组合逻辑输出，也可以是正沿触发的行为。[1]

```
val rf = Mem(32, UInt(64.W))
when (wen) { rf(waddr) := wdata }
val dout1 = rf(waddr1)
val dout2 = rf(waddr2)
```

在可选参数 `seqRead` 被设置成有效的时候，如果读地址是一个 `Reg`，Chisel 就会尝试推断时序读端口。单个读端口、单个写端口的 SRAM 可以如下设计：

```
val ram1r1w =
  Mem(1024, UInt(32.W))
val reg_raddr = Reg(UInt())
when (wen) { ram1r1w(waddr) := wdata }
when (ren) { reg_raddr := raddr }
val rdata = ram1r1w(reg_raddr)
```

如果读取和写入被放在在同一个 `when` 语句中，并且它们的条件是互斥的，那么就可以推断这是一个单端口 SRAM。

```
val ram1p = Mem(1024, UInt(32.W))
val reg_raddr = Reg(UInt())
when (wen) { ram1p(waddr) := wdata }
.elsewhen (ren) { reg_raddr := raddr }
val rdata = ram1p(reg_raddr)
```

如果在同一个时钟周期，同时写入和读取相同的 `Mem` 地址，那么读出的数据是未定义的。如果时序读取的使能是无效的，那么读出的数据也是未定义的。

`Mem` 也支持写入模具来实现部分写入操作。只有当对应的写入模具的比特有效的时候，给定的比特才会被写入存储器。

```
val ram = Mem(256, UInt(32.W))
when (wen) { ram.write(waddr, wdata, wmask) }
```

[1] 现在的 FPGA 工艺已经不（再）支持组合逻辑（异步）读取存储器了。读地址需要用寄存器锁住。

13 接口和成批连接

对于复杂的电路模块来说，定义和例化接口类是非常有效的设计模块 IO 的方法。首先最重要的，接口类允许用户用有效的形式化的方法抓住那些能够设计一次复用多次的接口，鼓励他们复用设计。再者，接口支持在生产者和消费者模块之间使用成批连接，允许用户大幅节省连线语句。最后，用户可以在管理大接口变动的时候只在一处做更改，在增加减少接口子域的时候减少了更改的位置数量。

13.1 端口：子类和嵌套

如我们之前讨论过的，用户可以从 `Bundle` 继承子类来定义自己的接口。例如，用户可以像这样定一个简单的链路来传输握手数据：

```
class SimpleLink extends Bundle {
  val data  = Output(UInt(16.W))
  val valid = Output(Bool())
}
```

我们下面可以用继承的方法增加奇偶位来扩展 `SimpleLink`：

```
class PLink extends SimpleLink {
  val parity = Output(UInt(5.W))
}
```

一般来说，用户可以用继承的方法组织出层级化的接口。

现在，只要把两个 `PLink` 套叠在一起形成新的绑裹 `FilterIO`，我们就可以定义过滤器接口了：

```
class FilterIO extends Bundle {
  val x = new PLink().flip
  val y = new PLink()
}
```

这里 `flip` 逐级递归地反转一个绑裹内部成员的“性别”，把输入变成输出，输出变成输入。

现在我们可以从模块继承一个过滤器类来定义过滤器了：

```
class Filter extends Module {
  val io = IO(new FilterIO())
  ...
}
```

这里 `FilterIO` 构成了模块的 `io` 域。

13.2 向量绑裹

元素的向量超过单个元素，可以形成更加丰富的接口层级。例如，根据一个 `UInt` 输入从一组输入生成一组输入的交换器可以通过 `Vec` 构造器实现：

```
class CrossbarIo(n: Int) extends Bundle {
  val in  = Vec(n, new PLink().flip())
  val sel = Input(UInt(sizeof(n).W))
  val out = Vec(n, new PLink())
}
```

这里 `Vec` 的第一个参数规定了数量，第二个参数是能产生端口的语句块。

13.3 成批连接

现在我们来用两个过滤来组成一个过滤器丛，如下例所示：

```
class Block extends Module {
  val io = IO(new FilterIO())
  val f1 = Module(new Filter())
  val f2 = Module(new Filter())

  f1.io.x <> io.x
  f1.io.y <> f2.io.x
  f2.io.y <> io.y
}
```

这里 `<>` 就是成批连接的操作符，它可以把两个子模块的相对方向的端口连接起来，也可以把父模块和子模块的相同方向的端口连接起来。成批连接能够把相同名字的子端口相互连接起来。当所有的连接都成功之后，电路就开始详细展开了。如果有端口被连接了多个端口，Chisel 就会报告警告给用户。

13.4 接口的视域

如图 2 所示，考虑一个简化的 CPU，只包括控制、数据子模块和和主控、存储器接口。在这个 CPU 中我们看到，控制通路和数据通路都只和指令和数据存储器接口的一部分连在一起。Chisel 允许用户局部地填充接口。用户先这样定义只读存储器和可读写存储器的完整的接口：

```
class RomIo extends Bundle {
  val isVal = Input(Bool())
  val raddr = Input(UInt(32.W))
  val rdata = Output(UInt(32.W))
}

class RamIo extends RomIo {
  val isWr  = Input(Bool())
  val wdata = Input(UInt(32.W))
```

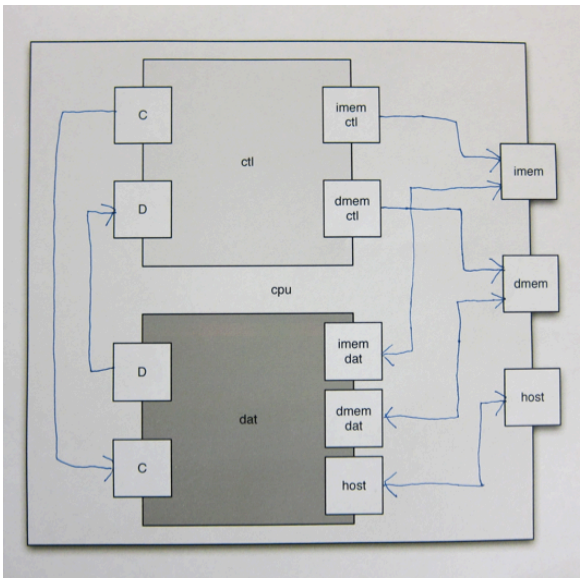

Figure 2: 包含控制、数据子模块和和主控、存储器接口的简化的 CPU。

```
}
```

现在控制通路可以用这些接口来定义自己的接口。

```
class CpathIo extends Bundle {
  val imem = RomIo().flip()
  val dmem = RamIo().flip()
  ...
}
```

控制通路和数据通路模块可以部分地连接这些端口，如示：

```
class Cpath extends Module {
  val io = IO(new CpathIo())
  ...
  io.imem.isVal := ...
  io.dmem.isVal := ...
  io.dmem.isWr  := ...
  ...
}

class Dpath extends Module {
  val io = IO(new DpathIo())
  ...
  io.imem.raddr := ...
  io.dmem.raddr := ...
  io.dmem.wdata := ...
  ...
}
```

现在我们可以在 CPU 模块中按照我们设想的那样使用成批连接了：

```
class Cpu extends Module {
  val io = IO(new CpuIo())
  val c  = Module(new CtlPath())
  val d  = Module(new DatPath())
  c.io.ctl  <> d.io.ctl
  c.io.dat  <> d.io.dat
  c.io.imem <> io.imem
  d.io.imem <> io.imem
  c.io.dmem <> io.dmem
  d.io.dmem <> io.dmem
  d.io.host <> io.host
}
```

重复的成批连接分别在局部连接了控制通路和数据通路的接口，最终把 CPU 的接口完全的连接起来。

14 函数式的模块创建

为模块构造设计一个函数式的接口也是非常有用的。例如，我们可以设计一个构造器，它以多路选择器模块作为输入，以更复杂的多路选择器模块作为输出：

```
object Mux2 {
  def apply (sel: UInt, in0: UInt, in1: UInt) = {
    val m = new Mux2()
    m.io.in0 := in0
    m.io.in1 := in1
    m.io.sel := sel
    m.io.out
  }
}
```

这里 `object Mux2` 是在模块类 `Mux2` 的基础上创建的一个 Scala 单例对象，`apply` 定义了一个方法可以创建一个 `Mux2` 的实例。有了这个 `Mux2` 创建函数，`Mux4` 的设计规范就显著的简单了。

```
class Mux4 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val in2 = Input(UInt(1.W))
    val in3 = Input(UInt(1.W))
    val sel = Input(UInt(2.W))
    val out = Output(UInt(1.W))
  })
  io.out := Mux2(io.sel(1),
                 Mux2(io.sel(0), io.in0, io.in1),
                 Mux2(io.sel(0), io.in2, io.in3))
}
```

从输入中做多路选择是如此的重要，所以 Chisel 把这个功能内建了，这就是 `Mux`。然而，不同于上面设计的 `Mux2`，Chisel 内建的版本允许 `in0` 和 `in1` 是任何数据类型，只要它们都属于继承于 `Data` 的相同子类。在第 15 节，我们会看到如何自己定义这样的高阶版本。

```
MuxCase(default, Array(c1 -> a, c2 -> b, ...))
```

此处的每个条件 / 数值的被表示成 Scala 数组中的三元组，`MuxCase` 可以被翻译成下面的 `Mux` 表达式。

```
Mux(c1, a, Mux(c2, b, Mux(..., default)))
```

Chisel 还提供了 `MuxLookup`，这是一个 n 路的多路选择器：

```
MuxLookup(idx, default,
          Array(0.U -> a, 1.U -> b, ...))
```

它可以用 `MuxCase` 重新写成下面的形式：

```
MuxCase(default,
        Array((idx === 0.U) -> a,
              (idx === 1.U) -> b, ...))
```

请注意，这些条件（如 c1, c2）必须用小括号括起来。

15 多态和参数化设计

Scala 是强类型的语言，可以用参数化的数据类型来定义一般化的函数和类。在本节，我们会展示 Chisel 的用户如何能够用参数化的类来设计他们自己的可复用的函数和类。

本节是教程的高阶部分，初次阅读可以跳过。

15.1 参数化的函数

前面我们使用 `Bool` 类型定义了 `Mux2`，现在我们来看一下如何定一个一般化的多路选择器函数。我们以布尔类型的条件和 `T` 类型的数据 con 和 alt 为参数来定义这个一般化的函数（con 和 alt 分别表示 then 和 else 表达式）。

```
def Mux[T <: Bits](c: Bool, con: T, alt: T): T { ... }
```

这里的 `T` 必须是 `Bits` 类型的子类。Scala 在调用 `Mux` 的时候会确保它能够找到实际的 con 和 alt 的参数类型共同的超类，否则它会报告编译时期的类型错误。例如，

```
Mux(c, 10.U, 11.U)
```

将生成一个 `UInt` 类型的信号线，这是因为此处的 `con` 和 `alt` 参数都属于 `UInt` 类型。

现在，我们来展示参数化函数的一个更加进阶的例子，为所有的 Chisel `Num` 类型定一个基于内积的 FIR 数字滤波器。内积 FIR 滤波器的数学定义是：

$$y[t] = \sum_j w_j * x_j[t-j] \tag{1}$$

这里的 x 是输入信号，w 是权值向量。在 Chisel 中，这个滤波器可以这样描述：

```
def delays[T <: Data](x: T, n: Int): List[T] =
  if (n <= 1) List(x) else x :: Delays(RegNext(x), n-1)

def FIR[T <: Data with Num[T]](ws: Seq[T], x: T): T =
  (ws, Delays(x, ws.length)).zipped.
    map( _ * _ ).reduce( _ + _ )
```

这里，`delays` 函数创建了一个对它的输入逐级增加延时的列表。`reduce` 可以使用二输入合成函数 `f` 够构造约简电路。在本例中，`reduce` 创建了一个求和电路。最后，只要是定义了乘法和加法的 Chisel `Num` 类型，`FIR` 函数都可以在上面工作。

15.2 参数化的类

与参数化的函数类似，我们也可以设计参数化的类来使它们更容易复用。例如，我们可以泛化过滤器类到任意一种链路。我们可以这样参数化 `FilterIO` 类，在它的构造函数体内使用零参数的泛化类型的构造函数，如下例所示：

```
class FilterIO[T <: Data](type: T) extends Bundle {
  val x = type.asInput.flip
  val y = type.asOutput
}
```

现在，我们可以设计 `Filter` 模块类，它也使用链路的泛化类型的构造器作为参数，再把它传递给 `FilterIO` 接口的构造器。

```
class Filter[T <: Data](type: T) extends Module {
  val io = IO(new FilterIO(type))
  ...
}
```

现在，我们可以基于 `PLink` 来定义 `Filter`：

```
val f = Module(new Filter(new PLink()))
```

如图 3 所示的一般化的 FIFO 可以这样用：

```
class DataBundle extends Bundle {
  val A = UInt(32.W)
  val B = UInt(32.W)
}
```

```
object FifoDemo {
  def apply () = new Fifo(new DataBundle, 32)
}
```

```
class Fifo[T <: Data] (type: T, n: Int)
    extends Module {
  val io = IO(new Bundle {
    val enq_val = Input(Bool())
    val enq_rdy = Output(Bool())
    val deq_val = Output(Bool())
    val deq_rdy = Input(Bool())
    val enq_dat = type.asInput
    val deq_dat = type.asOutput
  })
  val enq_ptr      = Reg(init = 0.U(sizeof(n).W))
  val deq_ptr      = Reg(init = 0.U(sizeof(n).W))
  val is_full      = Reg(init = false.B)
  val do_enq       = io.enq_rdy && io.enq_val
  val do_deq       = io.deq_rdy && io.deq_val
  val is_empty     = !is_full && (enq_ptr === deq_ptr)
  val deq_ptr_inc  = deq_ptr + 1.U
  val enq_ptr_inc  = enq_ptr + 1.U
  val is_full_next =
    Mux(do_enq && ~do_deq && (enq_ptr_inc === deq_ptr),
        true.B,
        Mux(do_deq && is_full, false.B, is_full))
  enq_ptr := Mux(do_enq, enq_ptr_inc, enq_ptr)
  deq_ptr := Mux(do_deq, deq_ptr_inc, deq_ptr)
  is_full := is_full_next
  val ram = Mem(n)
  when (do_enq) {
    ram(enq_ptr) := io.enq_dat
  }
  io.enq_rdy := !is_full
  io.deq_val := !is_empty
  ram(deq_ptr) <> io.deq_dat
}
```

Figure 3: 参数化的 FIFO 示例。

一般化的解耦合接口可以这样定义：

```
class DecoupledIO[T <: Data](data: T)
    extends Bundle {
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = data.cloneType.asOutput
}
```

这个模板就可以为任意的信号集合增加握手协议了。

```
class DecoupledDemo
  extends DecoupledIO()( new DataBundle )
```

图 3 中的 FIFO 接口现在可以简化成：

```
class Fifo[T <: Data] (data: T, n: Int)
```

```
    extends Module {
  val io = IO(new Bundle {
    val enq = new DecoupledIO( data ).flip()
    val deq = new DecoupledIO( data )
  })
  ...
}
```

16 多时钟域

Chisel 3.0 现在还不能支持多个时钟域，但是马上就将增加这样的支持。

17 鸣谢

很多人帮助过 Chisel 的设计，我们非常感谢他们的耐心、勇敢和对更好方法的信念。在 Chisel 的设计演进中，很多 Berkeley EECS Isis 组的同学们给出了每周反馈，包括但不限于 Yunsup Lee, Andrew Waterman, Scott Beamer 和 Chris Celio。Yunsup Lee 对第一版名为 TrainWreck 的 RISC-V 实现给了我们反馈。Andrew Waterman 和 Yunsup Lee 帮助我们建立 Verilog 后端，并且把 Chisel 版本的 TrainWreck 跑在了 FPGA 上。Brian Richards 是第一个 Chisel 的用户，（和 Huy Vo 一起）首先把 John Hauser 的 Verilog 版本的 FPU 翻译到 Chisel，后来又实现了一般化的内存模块。Brian 还给出了很多有价值的设计建议，带来了大量的硬件设计和设计工具方面的经验。Chris Batten 分享了他的快速多字段 C++ 模板库，启发了我们对快速仿真库的设计。Huy Vo 成为了我们本科研究助理，最先实际地帮助到 Chisel 的实现。我们真挚的感激所有加入 Chisel 训练营的 EECS 的同学们，他们建议并努力实现了很多硬件设计项目，这些工作都扩充了 Chisel 的内容。我们也很感激 James Martin 和 Alex Williams 在开发网络和存储器控制器和非阻塞缓存方面所做的工作。最后我们要感谢的是，Chisel 的函数式编程和位宽推断想法受到了 Gel [2] 语言早期工作的启发，Gel 是与 Dany Qumsiyeh 和 Mark Tobenkin 合作完成的一种硬件描述语言。

References

[1] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, Wawrzynek, J., Asanović *Chisel: Constructing Hardware in a Scala Embedded Language*. in DAC '12.

[2] Bachrach, J., Qumsiyeh, D., Tobenkin, M. *Hardware Scripting in Gel*. in Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th.

A 翻译对照表

Bitwidth	位宽
Build	构建
Boolean	布尔的
Buildin	内建
Bundle	绑裹
Bug	缺陷
Construct (n.)	概念，构造
Construct (v.)	构造，构建
Construction	构造
Constructor	构造器，构造函数
Create	创建
Debug	调试
Digit	数码
Element	单元，元素
Fan-out	扇出
Import	导入
Interface	接口
Instantation	例化
Instance	实例
Literal	字面，字面常量
Module	模块
Port	端口
Reflection	反省
Structure	结构
Vec	向量
Wire	信号线