嵌入式荣誉课第二次讨论

1. Class、Object和trait在Chisel中的使用场景。

class常用来生成硬件电路，其中的abstract class是抽象类，常用来设置一系列的常数

与接口，为后续的子类继承提供了模板。

Object

trait与abstract class的功能很像，可以作为补充。

object是伴生对象，一般有以下三种情况使用，1/类中有相关的常数，2/需要在构造类之前或者之后运行代码。3/需要对一个类进行多种构造。

常常在里面使用工厂方法，来简化一些模块的例化和连线。

1. Chisel本身不支持比特位赋值，Chisel的比特位赋值要如何实现？

通过vec()来实现。

使用scala的集合：

class MyManyElementFir(consts: Seq[UInt], bitWidth: Int) extends Module {

val io = IO(new Bundle {

val in = Input(UInt(bitWidth.W))

val out = Output(UInt(bitWidth.W))

})

val regs = mutable.ArrayBuffer[UInt]()

for(i <- 0 until consts.length) {

if(i == 0) regs += io.in

else regs += RegNext(regs(i - 1), 0.U)

}

val muls = mutable.ArrayBuffer[UInt]()

for(i <- 0 until consts.length) {

muls += regs(i) \* consts(i)

}

val scan = mutable.ArrayBuffer[UInt]()

for(i <- 0 until consts.length) {

if(i == 0) scan += muls(i)

else scan += muls(i) + scan(i - 1)

}

io.out := scan.last

}

ArrayBuffer的+=方法是往最后添加元素，+=:是向前添加元素，last方法是取出最后一个元素。

首先，我们创建一个ArrayBuffer regs，其元素类型是UInt。将输入添加为第一个元素，然后使用RegNext创建寄存器，该寄存器的输入连接到前一个元素regs(i-1)，并将其初始化为0.U。这些寄存器的使用就实现了在regs中保存io.in及其延迟信号：io.in\_delay1,io.in\_delay2,io.in\_delay3。

接下来，我们创建另一个元素类型是UInt的ArrayBuffer muls。muls的每个元素都是一个节点，每个元素都是regs(i)和consts(i)的乘积,也就是公式中的每一项的值。

最后，我们又创建另一个元素类型是UInt的ArrayBuffer scan。scan的作用就是将muls(0)、muls(0)+muls(1)、muls(0)+muls(1)+muls(2)、muls(0)+muls(1)+muls(2)+muls(3)的值依次保存下来，这样最后io.out其实就是scan的最后一个元素。

使用chisel的Vec

class ManyDynamicElementVecFir(length: Int) extends Module {

val io = IO(new Bundle {

val in = Input(UInt(8.W))

val consts = Input(Vec(length, UInt(8.W)))

val out = Output(UInt(8.W))

})

// Reference solution

var regs = VecInit(Seq.fill(length)(0.U(8.W)))

for(i <- 0 until length) {

if(i == 0) regs = VecInit(regs.takeRight(length - 1) :+ io.in)

else regs = VecInit(regs.takeRight(length - 1) :+ RegNext(regs(length - 1)))

}

var muls = VecInit(Seq.fill(length)(0.U(8.W)))

for(i <- 0 until length) {

muls = VecInit(muls.takeRight(length - 1) :+ (regs(i) \* io.consts(i)))

}

var scan = VecInit(Seq.fill(length)(0.U(8.W)))

for(i <- 0 until length) {

if(i == 0) scan = VecInit(scan.takeRight(length - 1) :+ muls(i))

else scan = VecInit(scan.takeRight(length - 1) :+ (muls(i) + scan(length - 1)))

}

io.out := scan(length - 1)

}

使用chisel的Vec和使用scala的ArrayBuffer在代码结构上是差不多的，不同的是后者使用的是Vec，Vec有以下几个添加元素的方法：

+: 表示向前添加元素

:+表示向后添加元素

++ 表示连接两个Vec，并且返回序列的类型和左边相同

++: 也表示连接两个Vec，不过返回序列的类型和右边相同

还需要注意的是，每次添加元素之后，返回的都是一个IndexedSeq，所以如果想要赋值，需要将其转换成Vec。

val regs = mutable.ArrayBuffer[UInt]()

for(i <- 0 until consts.length) {

//这里就是将io.in及其三个延迟值依次放入regs,之所以是延迟值,是因为使用了RegNext

//顺序就是io.in,io.in\_delay1,io.in\_delay2,io.in\_delay3

if(i == 0) regs += io.in

else regs += RegNext(regs(i - 1), 0.U)

}

io.out := (regs zip consts).map { case (a, b) => a \* b}.reduce(\_ + \_)

//regs最后存储的是(io.in,io.in\_delay1,io.in\_delay2,io.in\_delay3)

var regs = VecInit(Seq.fill(length)(0.U(8.W)))

for(i <- 0 until length) {

if(i == 0) regs = VecInit(regs.takeRight(length - 1) :+ io.in)

else regs = VecInit(regs.takeRight(length - 1) :+ RegNext(regs(length - 1)))

}

io.out := (regs zip io.consts).map { case (a, b) => a \* b }.reduce(\_ + \_)

1. Chisel的存储器有哪些类型，以及各自的用法。

Mem and SyncReadMem

Mem是同步写异步读Memory。在请求之后，写操作在时钟的上升沿有效。读取是组合的（请求将在同一时钟周期返回数据）实际上是生成一个寄存器堆。

SyncReadMem同步写同步读都是在上升沿写有效和返回数据。调用真的ram

用fpga时的dram时，要用黑盒，或者行为描述一样？

1. Chisel中多种Mux类型的具体功能和区别。

Mux(sel, in1, in2)”。sel是Bool类型，in1和in2的类型相同，都是Data的任意子类型。当sel为true.B时，返回in1，否则返回in2。

第二种就是针对上述n输入多路选择器的简便写法，形式为“MuxCase(default, Array(c1 -> a, c2 -> b, ...))”，它的展开与嵌套的Mux是一样的。第一个参数是默认情况下返回的结果，第二个参数是一个数组，数组的元素是对偶“(成立条件，被选择的输入)”。MuxCase在chisel3.util包里。Mux(c1, a, Mux(c2, b, Mux(..., default)))

第三种是MuxCase的变体，它相当于把MuxCase的成立条件依次换成从0开始的索引值，就好像一个查找表，其形式为“MuxLookup(idx, default, Array(0.U -> a, 1.U -> b, ...))”。它的展开相当于“MuxCase(default, Array((idx === 0.U) -> a, (idx === 1.U) -> b, ...))”。MuxLookup也在chisel3.util包里。

第四种是chisel3.util包里的独热码多路选择器，它的选择信号是一个独热码。如果零个或多个选择信号有效，则行为未定义。其形式如下：0001/0010/0100/1000

val hotValue = Mux1H(Seq(  
    io.selector(0) -> 2.U,  
    io.selector(1) -> 4.U,  
    io.selector(2) -> 8.U,  
    io.selector(4) -> 11.U  
))

ptioritymux：有优先级的编码器1100/0111之类的onehot码

1. 讲解Scala中列表常用方法（map、zip、zipWithIndex、fill、fold、tabulate、reduce）的功能并给出chisel中的应用场景。

操作 Scala 集合时，一般会进行两类操作：转换操作（transformation ）和行动操作（actions）

第一种操作类型将集合转换为另一个集合，第二种操作类型返回某些类型的值。

Map:

valnumbers= Seq(1,2,3,4,5,6)

1. //List(2, 4, 6, 8, 10, 12)
2. numbers.map(n=**>** n \* 2)
3. valchars= Seq('a','b','c','d')
4. //输出为List(A, B, C, D)
5. chars.map(ch=**>** ch.toUpper)

map  函数的逻辑是遍历集合中的元素并对每个元素调用函数

zip:

**zip方法将两个集合结合在一起**

scala>  List('a,'b,'c).zip(List(1,2,3))

res32: List[(Symbol, Int)] = List(('a,1), ('b,2), ('c,3))

zipWithIndex:

**将元素和下标结合在一起**

cala> List(2,3,4,5).zipWithIndex

res33: List[(Int, Int)] = List((2,0), (3,1), (4,2), (5,3))

fill:cala> val array = Array.fill(5)(3.5)

array: Array[Double] = Array(3.5, 3.5, 3.5, 3.5, 3.5)

如果fill第二个参数只写一个值的话，那么该数组的所有元素都是该值，但是如果第二个参数是一个iterator或者random，那么数组就会被赋值为它们的值。

b. 这里使用了Scala的call-by-name机制

Array.fill()方法的签名为：

def fill[T: ClassTag](n: Int)(elem: => T): Array[T]

包含两个参数列表，第一个为数组元素的个数；第二个参数使用“=> T”表明该参数为call-by-name参数，使用时如同这个参数是一个函数，这个函数返回类型为T的返回值。

所以Array.fill(n)(Module(new FullAdder()).io)的意思：

i. 创建一个数组；

i. 数组元素的个数为n;

i. 创建数组元素的方法为：Module(new FullAdder()).io。即每次创建数组元素时，都调用这段代码；

scala> val array = Array.fill(2)(math.random)

array: Array[Double] = Array(0.2810736748034083, 0.7261142068882558)

fold:

在 Scala 的上下文中，通常可以考虑  foldLeft  和  foldRight 。他们是从不同的方面做同样的工作：

valnumbers=Seq(1,2,3,4,5)

1. //15
2. numbers.foldLeft(0)((res, n) =**>** res + n)

在第一对括号中，我们放一个起始值。 在第二对括号中，我们定义需要对数字序列的每个元素执行的操作。 第一步，n = 0，然后它根据序列元素变化。

val words = Seq("apple", "dog", "table")

1. //13
2. words.foldLeft(0)((resultLength, word) =**>** resultLength + word.length)
3. val words = Seq("apple", "dog", "table")
4. //13
5. words.foldLeft(0)((resultLength, word) =**>** resultLength + word.length)

tabulate：

ist.tabulate(2,3)() // 代表生成一个二维的列表，两行三列

// 假设第二个参数是b

List.tabulate(2,3)(b)

// 这个b是什么呢

for (x1 <- 0 until 2){

for (x2 <- 0 until 3){

b; // x1与x2的计算公式

}

}

// 在第二个列表直接输入b的计算公式, 每个x用“\_”代表

List.tabulate(2,3)(\_\*\_) // 代表x1 \* x2

// 上面得到的列表

List[List[Int]] = List(List(0, 0, 0), List(0, 1, 2))

// 写成详细的计算过程

List[List[Int]] = List(List(0 \* 0, 0 \* 1, 0 \* 2),

List(1 \* 0, 1 \* 1, 1 \* 2))

reduce:

从左边加减，默认调用自己的reduceLeft()，方法。

val a = List(1,2,3,5,6)  
 val a = List(1,2,3,5,6)  
  val b = a.reduce(\_+\_) //求和21，类似a.sum  
  val c = a.reduce(\_-\_)//求差。-19  
  同理：(\_\*\_) (\_/\_)  
  val b = a.reduce(\_+\_) //求和21，类似a.sum  
  val c = a.reduce(\_-\_)//求差。-19  
  同理：(\_\*\_) (\_/\_)

flatten:

**flatten可以把嵌套的结构展开.** **当有一个集合的集合，** 然后你想对这些集合的所有元素进行操作时，就会用到 flatten。

**val**   abcd   **=**   Seq( 'a' ,   'b' ,   'c' ,   'd' )

**val**   efgj   **=**   Seq( 'e' ,   'f' ,   'g' ,   'h' )

**val**   ijkl   **=**   Seq( 'i' ,   'j' ,   'k' ,   'l' )

**val**   mnop   **=**   Seq( 'm' ,   'n' ,   'o' ,   'p' )

**val**   qrst   **=**   Seq( 'q' ,   'r' ,   's' ,   't' )

**val**   uvwx   **=**   Seq( 'u' ,   'v' ,   'w' ,   'x' )

**val**   yz    **=**   Seq( 'y' ,   'z' )

**val**   alphabet   **=**   Seq(abcd, efgj, ijkl, mnop, qrst, uvwx, yz)

// List(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z)

alphabet.flatten

flatmap:

**latMap结合了map和flatten的功能。接收一个可以处理嵌套列表的函数，然后** **把返回结果连接起来**

flatMap使用流程：内部成员进行 **map** **，** **map** **的结果为一个返回** **，然** **后在如果的基础上进行** **flat** **将** **map** **操作后的结果进行合并；可以拆分成map+flatten**

valabcd= Seq('a','b','c','d')

1. //List(A, a, B, b, C, c, D, d)
2. abcd.flatMap(ch=**>** List(ch.toUpper, ch))
3. scala**>** List(List(1,2),List(3,4)).flatMap(x=**>**x.map(x=**>**x\*2))
4. res5: List[Int] = List(2, 4, 6, 8)

foreach forall:

**foreach和map相似，只不过它没有返回值或者说返回值是Unit，foreach只要是为了对参数进行作用。**

**比如** names.foreach{name=>println(name)}

**确保集合中所有元素都要符合某些要求，如果有哪怕一个元素不符合条件，就需要进行一些处理：**

valnumbers=Seq(3,7,2,9,6,5,1,4,2)

* //ture
* numbers.forall(n=**>** n **<** **10**)
* //false
* numbers.forall(n=**>** n **>** 5)  而 forall 函数就是为处理这类需求而创建的。

Filter:

过滤出 集合中符合特定条件的子集

**val**   numbers   **=**   Seq( 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 )

numbers.filter(n   **=** > n   **%**   2   **==**   0 )    //seq(2,4,6,8,10)

**case**   **class**   Book(title **:**   String, pages **:**   Int)

**val**   books   **=**   Seq(

  Book( "Future of Scala developers" ,   85 ),

  Book( "Parallel algorithms" ,   240 ),

  Book( "Object Oriented Programming" ,   130 ),

  Book( "Mobile Development" ,   495 )

)

books.filter(book   **=** > book.pages > **=**   120 )

partition:

**将一个集合按一定的规则拆分成两个新的集合？比如，我们把某个集合拆分成偶数集和奇数集，partition 函数可以帮我们做到这一点：**

**val**   numbers   **=**   Seq( 3 ,   7 ,   2 ,   9 ,   6 ,   5 ,   1 ,   4 ,   2 )

//(List(2, 6, 4, 2), List(3, 7, 9, 5, 1))

numbers. **partition** (n   **=** > n   **%**   2   **==**   0 )

// (List(Book(Parallel algorithms,240), Book(Object Oriented Programming,130)),

     List(Book(Future of Scala developers,85), Book(Mobile Development,495)))

**case**   **class**   Book(title **:**   String, pages **:**   Int)

**val**   books   **=**   Seq(

  Book( "Future of Scala developers" ,   85 ),

  Book( "Parallel algorithms" ,   240 ),

  Book( "Object Oriented Programming" ,   130 ),

  Book( "Mobile Development" ,   495 )

)

**val** boo= books.partition(book =>book.pages %  2 == 0 )

  println (boo)

valnumbers=Seq(11,2,5,1,6,3,9)

* numbers.max//11
* numbers.min//1

caseclassBook(title:String, pages:Int)

* valbooks=Seq(
* Book("Future of Scala developers",85),
* Book("Parallel algorithms",240),
* Book("Object Oriented Programming",130),
* Book("Mobile Development",495)
* )
* //Book(Mobile Development,495)
* books.maxBy(book=**>** book.pages)
* //Book(Future of Scala developers,85)
* books.minBy(book=**>** book.pages)

**10.欧拉图函数（Euler Diagram函数）**

    差集、交集和并集。以下示例能很好地解释 Euler Diagram 函数：

|  |
| --- |
| **val**   num 1   **=**   Seq( 1 ,   2 ,   3 ,   4 ,   5 ,   6 )  **val**   num 2   **=**   Seq( 4 ,   5 ,   6 ,   7 ,   8 ,   9 )  //List(1, 2, 3)  num 1 .diff(num 2 )  //List(4, 5, 6)  num 1 .intersect(num 2 )  //List(1, 2, 3, 4, 5, 6, 4, 5, 6, 7, 8, 9)  num 1 .union(num 2 ) |

上述示例中的  union 保留了重复的元素。如果我们不需要重复怎么办？这时可以使用  distinct 函数：

//List(1, 2, 3, 4, 5, 6, 7, 8, 9)

num1.union(num2).distinct

1. Abstract class在硬件设计中能发挥什么作用？

中的abstract class是抽象类，常用来设置一系列的常数

与接口，为后续的子类继承提供了模板。

1. Chisel中的DecoupledIO和ValidIO的使用场景(通常结合Flipped一起使用)，连接符“<>”的作用。

定义一个模块前一定要先定义好端口。整个端口列表是由方法“IO[T <: Data](iodef: T)”来定义的，通常其参数是一个Bundle类型的对象，而且引用的字段名称必须是“io”。

因为端口存在方向，所以还需要方法“Input[T <: Data](source: T)”和“Output[T <: Data](source: T)”来为每个端口表明具体的方向。

注意，“Input[T <: Data](source: T)”和“Output[T <: Data](source: T)”仅仅是复制它们的参数，所以不能是已经被硬件类型包裹的数据类型。

目前Chisel还不支持双向端口inout，只能通过黑盒里的Analog端口来模拟外部[Verilog](https://so.csdn.net/so/search?q=Verilog&spm=1001.2101.3001.7020)的双向端口。

对于两个相连的模块，可能存在大量同名但方向相反的端口。仅仅为了翻转方向而不得不重写一遍端口显得费时费力，所以Chisel提供了“Flipped[T <: Data](source: T)”方法，可以把参数里所有的输入转输出，输出转输入。如果是黑盒里的Analog端口，则仍是双向的。

class MyIO extends Bundle {

val in = Input(Vec(5, UInt(32.W)))

val out = Output(UInt(32.W))

}

......

val io = IO(new MyIO) // in是输入，out是输出

......

val io = IO(Flipped(new MyIO)) // out是输入，in是输出

翻转方向的端口列表通常配合整体连接符号“<>”使用。该操作符会把左右两边的端口列表里所有同名的端口进行连接，而且同一级的端口方向必须是输入连输出、输出连输入，父级和子级的端口方向则是输入连输入、输出连输出。注意，方向必须按这个规则匹配，而且不能存在端口名字、数量、类型不同的情况。这样就省去了大量连线的代码。

class MyIO extends Bundle {

val in = Input(Vec(5, UInt(32.W)))

val out = Output(UInt(32.W))

}

......

val io = IO(new Bundle {

val x = new MyIO

val y = Flipped(new MyIO)

})

io.x <> io.y // 相当于 io.y.in := io.x.in; io.x.out := io.y.out

4、动态修改端口

方法①：使用可选字段

Chisel通过引入Scala的Boolean参数、可选值以及if语句可以创建出可选的端口，在例化该模块时可以通过控制Boolean入参来生成不同的端口。例如：

class HalfFullAdder(val hasCarry: Boolean) extends Module {

val io = IO(new Bundle {

val a = Input(UInt(1.W))

val b = Input(UInt(1.W))

val carryIn = if (hasCarry) Some(Input(UInt(1.W))) else None

val s = Output(UInt(1.W))

val carryOut = Output(UInt(1.W))

})

val sum = io.a +& io.b +& io.carryIn.getOrElse(0.U)

io.s := sum(0)

io.carryOut := sum(1)

}

注意，端口应该包装成可选值，这样不需要端口时就能用对象None代替，编译出来的Verilog就不会生成这个端口。在给可选端口赋值时，应该先用可选值的get方法把端口解放出来。这里也体现了可选值语法的便利性。

方法②：使用Zero-Width

chisel允许数据的位宽为0。位宽为0的IO会从生成的Verilog中去除，当你试图使用位宽为0的wire时，你会得到一个常数0。如果0是一个合理的默认值，那么更推荐使用该方法。如下例所示：

class HalfFullAdder(val hasCarry: Boolean) extends Module {

val io = IO(new Bundle {

val a = Input(UInt(1.W))

val b = Input(UInt(1.W))

val carryIn = Input(if (hasCarry) UInt(1.W) else UInt(0.W))

val s = Output(UInt(1.W))

val carryOut = Output(UInt(1.W))

})

val sum = io.a +& io.b +& io.carryIn

io.s := sum(0)

io.carryOut := sum(1)

}

注2：

如果端口定义成多维vec，那么在生成的verilog代码中，会将其展开为位宽相同、名字不同的端口。

还有一点需要注意，这样定义的模块会继承一个字段“clock”，类型是Clock，它表示全局时钟，在整个模块内都可见。对于组合逻辑，是用不上它的，而时序逻辑虽然需要这个时钟，但也不用显式声明。

还有一个继承的字段“reset”，类型是Reset，表示全局复位信号，在整个模块内可见。对于需要复位的时序元件，也可以不用显式使用该字段。

如果确实需要用到全局时钟和复位，则可以通过它们的字段名称来使用，但要注意类型是否匹配，经常需要“reset.toBool”这样的语句把Reset类型转换成Bool类型用于控制。隐式的全局时钟和复位端口只有在生成Verilog代码时才能看到。

## Decoupled: 标准Ready-Valid接口

DecoupledIO是Chisel提供的一个标准接口，它提供了一个用于数据传输的Ready-Valid界面。

* 发送方（数据源）：控制bits和valid
* 接收方：控制ready，当其准备好接收数据的时候，拉高ready

Chisel提供了一个DecoupledIO接口。

val myChiselData = UInt(8.W)

val myDecoupled = Decoupled(myChiselData)

val io = IO(new Bundle {

val in = Flipped(Decoupled(UInt(8.W)))

val out = Decoupled(UInt(8.W))

})

val queue = Queue(io.in, 2) // 2-element queue

io.out <> queue

### 实例：仲裁器

据预先设定好的优先级，仲裁器将数据从DecoupledIO源路由到DecoupledIO目的。

仲裁器分为

* 普通仲裁器：优先允许Index较低的请求
* 轮询仲裁器：轮询各个请求，优先级相等

val io = IO(new Bundle{

val x = ValidIO(new exampleIO) //ValidIO为端口添加一个Valid信号

val y = Flipped(ValidIO(new exampleIO)) //Flipped翻转端口，此时a,b均为Input

})

1. Chisel中BlackBox的用法，与普通Module的区别。

所以设计人员在当前版本下无法实现的功能，就需要用Verilog来实现。在这种情况下，可以使用Chisel的BlackBox功能，它的作用就是向Chisel代码提供了用Verilog设计的电路的接口，使得Chisel层面的代码可以通过模块的端口来进行交互。

假设有一个外部的Verilog模块，它的端口列表声明如下：

module Dut ( input [31: 0] a, input clk, input reset, output [3: 0] b );

按照Verilog的语法，它的例化代码应该是这样的：

Dut u0 ( .a(u0\_a), .clk(u0\_clk), .reset(u0\_reset), .b(u0\_b) );

倘若把这个Verilog模块声明成普通的Chisel模块，然后直接例化使用，那么例化的Verilog代码就会变成：

Dut u0 ( .io\_a(io\_u0\_a), .io\_clk(io\_u0\_clk), .io\_reset(io\_u0\_reset), .io\_b(i

如果定义Dut类时，不是继承自Module，而是继承自chisel3.BlackBox，则允许只有端口定义，也只需要端口定义。此外，在别的模块里例化黑盒时，编译器不会给黑盒的端口名加上“io\_”，连接的线网名变成引用黑盒的变量名(u0)与黑盒端口名(a，clk，reset，b)的组合。例如：

// blackbox.scala

package test

import chisel3.\_

class Dut extends BlackBox {

val io = IO(new Bundle {

val a = Input(UInt(32.W))

val clk = Input(Clock())

val reset = Input(Bool())

val b = Output(UInt(4.W))

})

}

class UseDut extends Module {

val io = IO(new Bundle {

val toDut\_a = Input(UInt(32.W))

val toDut\_b = Output(UInt(4.W))

})

val u0 = Module(new Dut)

u0.io.a := io.toDut\_a

u0.io.clk := clock

u0.io.reset := reset

io.toDut\_b := u0.io.b

}

object UseDutTest extends App {

chisel3.Driver.execute(args, () => new UseDut)

}

它对应生成的Verilog代码为：

// UseDut.v

module UseDut(

input clock,

input reset,

input [31:0] io\_toDut\_a,

output [3:0] io\_toDut\_b

);

wire [31:0] u0\_a; // @[blackbox.scala 20:18]

wire u0\_clk; // @[blackbox.scala 20:18]

wire u0\_reset; // @[blackbox.scala 20:18]

wire [3:0] u0\_b; // @[blackbox.scala 20:18]

Dut u0 ( // @[blackbox.scala 20:18]

.a(u0\_a),

.clk(u0\_clk),

.reset(u0\_reset),

.b(u0\_b)

);

assign io\_toDut\_b = u0\_b; // @[blackbox.scala 25:14]

assign u0\_a = io\_toDut\_a; // @[blackbox.scala 22:11]

assign u0\_clk = clock; // @[blackbox.scala 23:13]

assign u0\_reset = reset; // @[blackbox.scala 24:15]

endmodule

可以看到，例化黑盒生成的Verilog代码，完全符合Verilog例化模块的语法规则。通过黑盒导入Verilog模块的端口列表给Chisel模块使用，然后把Chisel代码转换成Verilog，把它与导入的Verilog一同传递给EDA工具使用。

BlackBox的构造方法可以接收一个Map[String, Param]类型的参数，这会使得例化外部的Verilog模块时具有配置模块(“#(参数配置)”)的功能。

映射的键固定是字符串类型，它对应Verilog里声明的参数名；映射的值对应传入的配置参数，可以是字符串，也可以是整数和浮点数。

虽然值的类型是Param，这是一个Chisel的印章类，但是单例对象chisel3.experimental里定义了相应的隐式转换，可以把BigInt、Int、Long、Double和String转换成对应的Param类型。例如把上例修改成：

mport chisel3.experimental.\_

class Dut extends BlackBox(Map("DATA\_WIDTH" -> 32,

"MODE" -> "Sequential",

"RESET" -> "Asynchronous")) {

val io = IO(new Bundle {

val a = Input(UInt(32.W))

val clk = Input(Clock())

val reset = Input(Bool())

val b = Output(UInt(4.W))

})

}

对应的Verilog就变成了：

...

Dut #(.DATA\_WIDTH(32), .MODE("Sequential"), .RESET("Asynchronous")) u0 ( // @[blackbox.scala 23:18]

.a(u0\_a),

.clk(u0\_clk),

.reset(u0\_reset),

.b(u0\_b)

);

### 四、inout端口

Chisel目前只支持在黑盒中引入Verilog的inout端口。Bundle中使用 “Analog(位宽)”声明Analog类型的端口，经过编译后变成Verilog的inout端口。模块里的端口可以声明成Analog类型，但只能用于与黑盒连接，不能在Chisel代码中进行读写，也即只能使用<>或者对每个端口依次赋值进行连接操作。

因为是双向端口，所以不需要用Input或Output指明方向，但是可以用Flipped来翻转，也就不会影响整个Bundle的翻转。使用前，要先用“chisel3.experimental.\_”进行导入。

// inout.scala

package test

import chisel3.\_

import chisel3.util.\_

import chisel3.experimental.\_

class InoutIO extends Bundle {

val a = Analog(16.W)

val b = Input(UInt(16.W))

val sel = Input(Bool())

val c = Output(UInt(16.W))

}

class InoutPort extends BlackBox with HasBlackBoxInline {

val io = IO(new InoutIO)

setInline("InoutPort.v",

"""

|module InoutPort( inout [15:0] a,

| input [15:0] b,

| input sel,

| output [15:0] c);

| assign a = sel ? 'bz : b;

| assign c = sel ? a : 'bz;

|endmodule

""".stripMargin)

}

class MakeInout extends Module {

val io = IO(new InoutIO)

val m = Module(new InoutPort)

m.io <> io

}

object InoutGen extends App {

chisel3.Driver.execute(args, () => new MakeInout)

}

对应的Verilog为：

// MakeInout.v

module MakeInout(

input clock,

input reset,

inout [15:0] io\_a,

input [15:0] io\_b,

input io\_sel,

output [15:0] io\_c

);

wire [15:0] m\_b; // @[inout.scala 32:17]

wire m\_sel; // @[inout.scala 32:17]

wire [15:0] m\_c; // @[inout.scala 32:17]

InoutPort m ( // @[inout.scala 32:17]

.a(io\_a),

.b(m\_b),

.sel(m\_sel),

.c(m\_c)

);

assign io\_c = m\_c; // @[inout.scala 34:8]

assign m\_b = io\_b; // @[inout.scala 34:8]

assign m\_sel = io\_sel; // @[inout.scala 34:8]

endmodule