
Chisel教程

Chisel简介

- 性质：Chisel是**Scala语言的一个库**，并不是一个全新的语言
- 功能：描述硬件设计
- 特点：
  - 提供了函数式编程、硬件原语等更简便的编程工具
  - 提供了类、结构体等更高层次的抽象
  - 提供了更好的参数化编程思路，减少了verilog中width-1:0的反人类设计

实用资料

- Chisel官方Github仓库：chipsalliance/chisel: Chisel: A Modern Hardware Design Language (github.com)
- Chisel官方文档：Chisel | Chisel (chisel-lang.org)

Chisel原理

运行Scala

Scala的安装

- Scala的运行需要两个重要工具
  - 运行时环境：Java Development Kit（JDK）
  - 构建工具：**SBT**或MILL，安装方法在这里
- 将Firrtl转换为Verilog需要
  - firtool：llvm/circt: Circuit IR Compilers and Tools (github.com)

注：从Chisel v6.0.0开始，firtool无需额外下载，Scala运行时可自动下载并安装适合于当前版本的firtool

Scala文件结构

如果想要运行Scala代码，必须有如下文件夹结构：

```
./
├── src
│   └── main
│       └── scala
│           ├── /* your source files here */
```

处于`src/main/scala`文件夹及其子文件夹下的所有文件，都将被视为“处于项目中”，其中的类互相可见，不存在文件隔离。

Scala的构建与运行

在最外层文件夹运行以下命令，即可构建并执行scala项目：

```
sbt run
```

Scala代码会从继承自App类的对象开始执行，如果有多个类继承自App，则执行前会通过命令行询问你要执行哪个对象：

```
Multiple main classes detected. Select one to run:
 [1] CPU_Main
 [2] MMU_Main

Enter number:
```

这时，在键盘输入`1`，即可开始执行`CPU_Main`这个对象。

Scala运行示例

如果想要使用Chisel库进行硬件开发，我们至少需要两个文件：

- 硬件描述文件：用来编写硬件
- Scala执行文件：用来编写继承自App的对象，用以运行

> 不能把他们写在一起的原因是，构建Verilog的ChiselStage对象包含在circt.stage中，与chisel3.utils库有冲突情况，除非你真的不用chisel3.utils里面丰富的工具，否则在import时就会报错

假设我们已经构建了一个完全符合Chisel规范的ALU硬件，我们可以通过如下代码将其运行并转化为verilog：

```
import chisel3._
import circt.stage.ChiselStage
import chisel3.stage.ChiselOption

object ALU_Main extends App {
    ChiselStage.emitSystemVerilogFile(
        new ALU,
        Array("-td", "build/"),
    )
}
```

`emitSystemVerilogFile`方法就可以将这个ALU转化为verilog代码，并将代码放置到`-td`所指向的文件夹中。

Chisel代码的基本组成

Chisel的引入

如果想使用Chisel编写硬件，你需要在代码开头编写：

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

第一个是必须要有的，第二个是chisel的丰富的硬件原语，建议也一并import进来。

硬件模块：Module类的继承

在verilog中，最基本的硬件功能单元被称为`module`，而在Chisel中，我们可以将一个硬件模块看做一个“类”，这个“类”就是实打实的硬件。Chisel规定，**一切这样的硬件“类”都必须继承自`Module`类，后者是Chisel库中给出的一个既定类。**

> 注：在这里大家可以姑且这么认为，其实硬件类可以继承自很多类，比如RawModule、BlackBox等，但他们使用率极低，且几乎都可以被Module类平替。

例如，如果我希望实现一个ALU，那么我需要写一个这样的类：

```
class ALU extends Module{

}
```

这样的一个Module子类一旦声明，它就会自带两个公有成员：

- clock：时钟信号，属于类Clock
- reset：复位信号，属于类Reset

也就是说，由它生成的ALU模块应该是类似于这样的：

```
module ALU(
    input clock,
    input reset,
    ....
);
    ....
```

Chisel在生成verilog的过程中，会进行**死代码删除**优化。如果你没有使用这两个信号，那么它最终生成的ALU模块中**就不存在这两个信号**，这也符合组合逻辑的定义。

> 注：Scala的类型系统远比verilog复杂，如果你在模块中希望使用reset信号，那么必须使用 `reset.asBool` 来将其转换为布尔值，再使用判断语句进行判断。

输入输出的定义

一个模块必须存在输入输出，这些信号我们更希望能用一个“信号组”将其抽象为结构体以便于维护。Chisel为我们提供了一个`Bundle`类，它更像是结构体，能将多个信号打包起来：

```
class ALU_IO extends Bundle{
    val src1  = Input(UInt(32.W))
    val src2  = Input(UInt(32.W))
    val op    = Input(UInt(5.W))
    val res   = Output(UInt(32.W))
}
```

上述代码中，四个量`src1` `src2` `op` `res`被组装成了一个`Bundle`类，下面我们就对输入输出信号的具体细节进行说明。

输入输出信号需要声明成`val`而不是`var`

事实上Chisel规定，**如果你希望这个信号唯一存在于verilog代码中，那么必须要声明为**`val`，这是因为`var`往往用于硬件逻辑迭代计算。这里举出一个来自verilog的逆天例子：

```
module count_1(
    input [7:0] a,
    output [3:0] nums_1
);
    reg [3:0] count;
    always @(*) begin
        count = 0;
        for(integer i = 0; i < 8; i = i + 1) begin
            count = count + a[i];
        end
    end
    assign nums_1 = count;
endmodule
```

这个模块是为了计数`a`中有多少个1，那么在这里，`a`和`nums_1`都是`val`，而`count`是`var`。

那么如何用Chisel描述这个组合逻辑呢?

```
count = PopCount(a)
```

其实就这么简单，但是为了体现`var`的特点，我们搞个复杂的：

```
var count = UInt(3.W)
for (i <- 0 until 8){
    count = Mux(a(i), count + 1.U, count)
}
```

看不太懂没有关系，这个例子只是为了给大家解释`var`和`val`的区别。这模块似乎在实际应用中不多，因此大部分的情况下，我们就把信号声明为`val`是最稳妥的。

输入的每个信号属于Input类，输出的每个信号属于Output类，

它们的构造函数是一个Chisel数据类型。例如我们拿其中一个例子来看：

```
val src1 = Input(UInt(32.W))
```

这表示`src1`是一个Input类的实例，并且它在参与计算时视作无符号32位整数。有关Chisel数据类型，我们将在后文讲解。

定义好了输入输出的`Bundle`，我们就可以在模块中实例化它，使其成为真正的硬件信号：

```
class ALU extends Module{
    val io = IO(new ALU_IO)
}
```

`io`是`IO`类的一个实例，这个类的构造函数参数是一个继承自`Bundle`类的实例。这时的`src1`信号，已经是`io`实例的一个成员了，因此如果我们希望访问它，需要使用：

```
io.src1
```

Chisel数据类型与基本语法

常用基础数据类型

在输入输出定义好后，我们就需要关注Chisel本身的数据类型了。在编写Chisel时，一定要格外注意**这个变量是“硬件”，还是说只是一个“值”**，更需要注意**这个变量是Chisel中的数据类型，还是Scala中的数据类型。**下面介绍几种常见的数据类型：

UInt：Chisel无符号整数类

最常用的数据类型，这个类需要通过一个`Width`类的实例进行构造，而`Width`类又可以通过Scala的整数类进行构造。例如，我想构造一个32位Chisel无符号整数，可以通过以下方法：

```
UInt(32.W)
```

还可以通过Scala的整数类进行构造：

```
5.U
5.U(3.W)
```

第一种情况下，位宽由编译器自动推断；第二种情况下，位宽已经指定。

如果希望访问`UInt`实例的某些位，可以用下标实现：

```
val a = 31.U
val b = a(1, 0)
```

特别注意，**不能使用下标来修改某一位，对UInt的修改必须整体进行！**

```
val a = Wire(31.U(5.W))
a(0) := 1.U(1.W) // 错误的
```

Bool：Chisel布尔类

第二常用的数据类型，**只有Bool类型的数据类型，才可以对其真假进行判断。**它一般是Chisel判断表达式的结果，但是在某些情况下，我们需要把`UInt(1.W)`的实例`a`转换为布尔类以进行真假判断：

```
a.asBool
```

当然，对于单位宽数据，我们也可以在输入输出中直接使用布尔类进行构造：

```
val is_ok = Input(Bool())
```

SInt：Chisel有符号整数类

SInt和UInt的性质几乎一样，不过我们几乎不把变量声明为SInt，而是经常将UInt转化为SInt执行计算，例如有符号数比较：

```
src1.asSInt < src2.asSInt
```

需要注意的是，以上三个类只是**“值”**，它们并不是verilog中的线网型变量，**不能被赋值。**如果你希望声明一个32位宽的线网，那么你需要再“套一层”，构造一个**线网类**：

```
val a = Wire(UInt(32.W))
```

此时，a的值就可以通过赋值运算符`:=`被组合逻辑改动了，例如：

```
a := 1.U + 2.U
```

注意，这里不可以写作`a := 1 + 2`，因为1和2都是Scala中的数据类型，我们需要将其转换为Chisel的无符号整数数据类型。

常用聚合数据类型

如果你掌握了基础数据类型，那么你已经可以用Chisel表达任何Verilog的变量了。不过，那样怎么才能体现我们Chisel的优势呢？我们需要更强力的伙伴！

Vec：Chisel中的数组类

Vec类可以任意嵌套，用来表达任意长度、任意宽度的数组。它的构造函数需要两个参数：**数组长度**和**元素类型**。例如，我希望创建一个长度为8的32位宽数组：

```
val a = Wire(Vec(8, UInt(32.W)))
```

数组类的嵌套可以如下表示：

```
val b = Wire(Vec(8, Vec(8, UInt(32.W))))
```

如果想访问或者修改 `a` 数组中的某几个元素，可以用下标：

```
val a = Wire(Vec(4, UInt(32.W)))
val b = Wire(Vec(8, UInt(32.W)))
a(0) := b(0)
```

基本上，目前开发时聚合类型只使用Vec类，其它聚合类都属于实验类型。

判断语句

类似于verilog中的行为级描述，Chisel也存在这种“乍一看不好评估延迟”的语法，但也正因为他们的存在，Chisel变得可以像verilog那样编写理解了。

when ... .elsewhen ... .otherwise

由于Scala已经占用了if和else关键词，且在Chisel中这两个关键词也会使用，故为了描述硬件，Chisel引入了新的关键词：

```
when(...){
    ...
}.elsewhen(...){
    ...
}.otherwise{
    ...
}
```

switch

verilog中的case语句可以使用**switch**关键字实现，但需要注意的是，**switch中并不支持default语句**，因此为了防止锁存器，我们需要给予被赋值信号一个默认值：

```
alu_out := DontCare
switch(alu_op){
    is(ALU_ADD) {
        alu_out := src1 + src2
    }
    is(ALU_SUB) {
        alu_out := src1 - src2
    }
    is(ALU_SLT) {
        alu_out := Mux(src1.asSInt < src2.asSInt, 1.U, 0.U)
    }
    is(ALU_SLTU) {
        alu_out := Mux(src1 < src2, 1.U, 0.U)
    }
    is(ALU_NOR) {
        alu_out := ~(src1 | src2)
    }
}
```

位拼接

- `##` 拼接运算符

  拼接运算符可以将其右侧的数值拼接到左侧的低位，例如：

  ```
  val a = b ## c ## d
  ```

  这对应于verilog中的

  ```
  assign a = {b, c, d}
  ```

- `Cat` 硬件原语

  Cat可以传递任意多个参数，它将把从左到右的参数从高到低进行拼接，例如：

  ```
  val a = Cat(b, c, d)
  ```

  这对应于verilog中的

  ```
  assign a = {b, c, d}
  ```

相等与不等判断

由于Scala已经占用了`==`和`!=`运算符，且这两个运算符在Chisel代码中依然有其意义，所以Chisel重新定义了两个判断符：

- `===`相等判断
- `=/=`不等判断

缩位运算

Chisel定义了两种缩位运算，它们都以对象的方法的形式而存在：

- `andR` 缩位与

```
val a = 14.U // 0xfffe
val b = a.andR // b = 0.U
```

- `orR` 缩位或

```
val a = 14.U // 0xfffe
val b = a.orR // b = 1.U
```

寄存器

Chisel提供了丰富的寄存器原语，寄存器也只能使用这些原语进行构建。

Reg

最基础的寄存器，朴实无华，没有任何特点，几乎不用

```
Reg(t: T) // t为类型模板
```

示例：

```
val a = Reg(UInt(32.W))
when(reset.asBool){
    a := 0.U
}.elsewhen(io.en){
    a := io.in
}
io.out := a
```

RegNext

将一个信号打一拍后输出，可以指定复位值：

```
RegNext(next: T, init: T) // 可以不指定init
```

示例：

```
val a = RegNext(io.in, 0.U)
io.out := a
```

坏处就是，这个原语没有办法指定寄存器的使能，自由度略下降

RegInit

指定复位值的寄存器，类型由这个复位值自动推断

```
RegInit(init: T)
```

示例：

```
val a = RegInit(0.U(32.W))
when(io.en){
    a := io.in
}
io.out := a
```

这个原语应用最为广泛，是大家最喜爱的原语之一

ShiftRegister

将一个信号打指定拍后输出，可以指定使能、复位值、打拍数

```
ShiftRegister(in: T, n: Int)
ShiftRegister(in: T, n: Int, en: Bool)
ShiftRegister(in: T, n: Int, resetData: T, en: Bool)
```

示例：

```
val a = ShiftRegister(io.in, 1, 0.U, io.en)
io.out := a
```

段间寄存器最喜欢的一集

Chisel高级原语与基本类方法

组合逻辑硬件原语

什么是硬件原语？硬件原语类似于函数，是Chisel集成好的复杂逻辑实现。这些实现近乎是这些功能在硬件上的最优解。

Mux

```
Mux(cond: Bool, true_value: T, false_value: T)
```

Mux是最基础的硬件原语，也是应用最广泛的原语。它根据判断条件来实现了一个二选一多选器，例如：

```
val a = Mux(is_true, b, c)
```

OHToUInt

```
OHToUInt(value: UInt)
```

OHToUInt可以将输入的独热码转换为二进制编码，位宽自动推断。但若输入为非法独热码，那么结果将是无法预知的。

根据源码，OHToUInt采用了二分查找来找到1在独热码中的位置，并返回这个位置——这无疑是最为高效的算法。

一般情况下，OHToUInt常常用于**返回命中索引的全相连查找**，如果全相连查找还需要获得对应命中单元的其他存储信息，那么用这个原语返回索引，再访问一次存储器并不是什么很好的主意，而是需要用到下文所述的其他方法。

UIntToOH

```
UIntToOH(value: UInt)
```

这个硬件原语将输入转换为对应的独热码，其实底层代码就是`1.U << value`

PriorityEncoder

```
PriorityEncoder(value: UInt)
```

正如其名，这个原语可以将输入的数据进行优先编码，**低位优先**。这个原语优化程度非常有限，更多的时候是为了提供一个更方便使用的优先编码器，而对时序没有太大影响。

PriorityEncoderOH

```
PriorityEncoderOH(value: UInt)
```

这个原语的功能与PriorityEncoder功能类似，只是输出的是独热码。这个硬件原语的时序要略好于PriorityEncoder。

Mux1H

```
Mux1H(oh_code: UInt, info: Vec)
```

独热码多选器，若info总项数为n，且独热码oh_code长度也为n，且第k位是1，则返回info[k]。其本质算法是借助独热码特点，使用了与或式代替传统选择逻辑，减少了50%的逻辑延迟。

上文提到的**获取命中项存储信息的全相连查找**，就可以使用这个硬件原语：

- 首先，使用命中逻辑获取命中信息的独热码，这个比较是并行的；
- 随后，用Mux1H，第一个参数为命中独热码，第二个参数为整个存储单元

如此，返回结果便是命中项存储的所有信息。

MuxLookup

```
MuxLookup(key: UInt, default: T)(mapping: Seq[(UInt, T)])
```

MuxLookup是以查找表的形式进行多选，包含两组参数，第一组参数中key代表查找的索引，default代表若查找不到，那么结果默认返回的值；第二组参数是Scala中的映射表，我们用一个例子来看：

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

当idx为0时，返回a；当idx为1时，返回b，否则返回default。这其实就是switch的简化版，对时序的优化并不大。

PopCount

```
PopCount(value: UInt)
```

这个硬件原语返回输入value存在多少个1，它的底层实现是采用二叉树加法进行1个数的累加。

基本类方法

以下的方法是为了更简便地构建复杂的值或批量连线。

VecInit

VecInit是为Vec创建时赋予初值的对象，其中有两个方法非常重要：

- **VecInit.fill**

  这个方法是为了构建一个由相同的若干值组成的数组，这个方法在批量赋予初值的时候很方便，例如将32个32位寄存器组成的寄存器堆初值全赋值为0：

```
val rf = RegInit(VecInit.fill(32)(0.U(32.W)))
```

- **VecInit.tabulate**

  这个方法更是重量级，可以为赋值的每一项按照一定规律进行复制，例如将第i号寄存器初值赋予为i：

```
val rf = RegInit(VecInit.tabulate(32)(i => i.U))
```

WireDefault

由于在switch或者if-else嵌套中，我们在有些情况下希望某个信号维持一个默认值，而无需再对其额外赋值。这里我们可以用WireDefault来为一个组合信号在声明时赋予一个默认值：

```
val a = WireDefault(0xffff.U(32.W))
```

map

map是每一个可迭代变量都具备的内在方法，不过更多的时候，Bundle数组里运用map方法的机会比较多，用来提取将Bundle中的某一个成员提取出来，重新组成数组：

```
class crat extends Bundle{
    val lr = UInt(5.W)
    val free = Bool()
}
val rat = RegInit(VecInit.fill(32)(new crat))
val free_list = rat.map(_.free)
```

如此，free_list就是由rat中全体free保序组成的新数组，共32项。

take

take方法接受一个参数n，返回当前对象低n个元素（若对象不为聚合类型，则返回其低n位），例如：

```
val rf = RegInit(VecInit.tabulate(32)(i => i.U))
val rf_low_8 = rf.take(8)
```

这里就取出了`rf`的0到7号寄存器的值。

drop

drop与take对应，将当前对象低n个元素去除，例如：

```
val rf = RegInit(VecInit.tabulate(32)(i => i.U))
val rf_high_8 = rf.drop(24)
```

这里舍弃了rf最低24个元素，其结果就是rf的高8个元素。