Generators

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

September 30, 2025

Overview

- Functions with more outputs
- Solution to the min/max lab
- Type conversions
- Parameters (simple and types)
- More Scala for generators

Functions with Multiple Outputs

- We use functions to generate hardware
- The return value is the *output* of that *module*
- A function usually has a single return value
- Use a Scala tuple for more output *ports*

```
def compare(a: UInt, b: UInt) = {
  val equ = a === b
  val gt = a > b
  (equ, gt)
}
```

Functions with More Outputs

- Access the two output wires with the `._n` syntax.

```
val cmp = compare(inA, inB)
val equResult = cmp._1
val gtResult = cmp._2
```

- Or directly decompose the tuple

```
val (equ, gt) = compare(inA, inB)
```

- Functions can be declared as part of a `Module`
- Better place them into a Scala object collecting utility functions
- Functions can serve as ligthweight modules

Lab two Weeks Ago

- Write a search for the minimum circuit (with `treeReduce()`)
- Add the generation of the index of the minimum value
- This was a problem formulated by Microchip

Functional Generation

- Anonymous functions, called *function literal*

```
(param) => function body
```

- A function for a minimum search

```
val min = vec.reduceTree((x, y) => Mux(x <
    y, x, y))
```

- This was a very short exercise - let us extend this

Minimal Function with Index

- Was the example for Tjark's heap sort

```
class Two extends Bundle {
  val v = UInt(w.W)
  val idx = UInt(8.W)
}

val vecTwo = Wire(Vec(n, new Two()))
for (i <- 0 until n) {
  vecTwo(i).v := vec(i)
  vecTwo(i).idx := i.U
}

val res = vecTwo.reduceTree((x, y) =>
    Mux(x.v < y.v, x, y))
```

- We need an extra bundle to hold both values
- A for loop is not so functional

We Can Use Tuples and zipWithIndex

- `zipWithIndex` transforms the original sequence to a sequence of tuples with second element is the index
- Use `map` to translate from Scala `Int` to Chisel `UInt`
- `reduce` does the minimum function, actually called 2 times (we could optimize this)

```
val resFun = vec.zipWithIndex
  .map((x) => (x._1, x._2.U))
  .reduce((x, y) => (Mux(x._1 < y._1, x._1,
     y._1), Mux(x._1 < y._1, x._2, y._2)))
```

- The result is a Scala `Vector` and not a Chisel `Vec`
- No `reduceTree` available on a Scala `Vector`

Solution with a Vec

- This results in a Chisel `Vec`

```
val scalaVector = vec.zipWithIndex
  .map((x) => MixedVecInit(x._1,
      x._2.U(8.W)))
val resFun2 = VecInit(scalaVector)
  .reduceTree((x, y) => Mux(x(0) < y(0), x,
      y))

val minVal = resFun2(0)
val minIdx = resFun2(1)
```

- `MixedVecInit` is like a `Bundle`, but indexable
- We should add a `reduceTree` to the Scala sequence version (`TraversableOnce`)

Type Conversion

- Sometimes we would like to see a value in different types
- All types represent a collection of bits
- Example to package 4 bytes into a 32-bit `UInt`

```
val vec = Wire(Vec(4, UInt(8.W)))
val word = vec.asUInt
```

- And converting it back to a `Vec`

```
val vec2 = word.asTypeOf(Vec(4, UInt(8.W)))
```

Type Conversion

- Convert a `Bundle` to a `UInt`

```
class MyBundle extends Bundle {
  val a = UInt(8.W)
  val b = UInt(16.W)
}

val bundle = Wire(new MyBundle)
val word2 = bundle.asUInt
```

- A `UInt` can be converted (back) to a bundle

```
val bundle2 = word2.asTypeOf(new MyBundle)
```

- Initialize to 0 on a conversion

```
val bundle3 = 0.U.asTypeOf(new MyBundle)
```

Simple Parameters

- Simplest way is bit width
- You have seen this, also in Verilog or VHDL

```
class ParamAdder(n: Int) extends Module {
  val io = IO(new Bundle{
    val a = Input(UInt(n.W))
    val b = Input(UInt(n.W))
    val c = Output(UInt(n.W))
  })

  io.c := io.a + io.b
}
```

- A bit more interesting is using case classes for parameters

```
case class Config(txDepth: Int, rxDepth: Int,
   width: Int)
```

Case Classes

- Reading the immutable fields

```
val param = Config(4, 2, 16)

println("The width is " + param.width)
```

- Adding checking code to case classes

```
case class SaveConf(txDepth: Int, rxDepth:
    Int, width: Int) {

  assert(txDepth > 0 && rxDepth > 0 && width >
      0, "parameters must be larger than 0")
}
```

Module with Type Parameters

- Assume a network-on-chip
- Moves data between processing cores
- We want to *parameterize* that data type
- Add a type parameter `T` to the Module constructor

```
class NocRouter[T <: Data](dt: T, n: Int)
    extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val address = Input(Vec(n, UInt(8.W)))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

Use that Router

- Define data type we want to route with

```
class Payload extends Bundle {
  val data = UInt(16.W)
  val flag = Bool()
}
```

- Pass an instance of that bundle to the constructor of the router

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

Parameterized Bundles

- We still have vectors of addresses and the payload
- We want to parametrize a Bundle

```
class Port[T <: Data](dt: T) extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

- `dt` is the type parameter, which we use for `cloneType`
- However, it is a public field, in the way for using in a `Vec`

Parameterized Bundles

- As a fix (workaround) make it private

```
class Port[T <: Data](private val dt: T)
    extends Bundle {
  val address = UInt(8.W)
  val data = dt.cloneType
}
```

- Define our router ports

```
class NocRouter2[T <: Data](dt: T, n: Int)
    extends Module {
  val io =IO(new Bundle {
    val inPort = Input(Vec(n, dt))
    val outPort = Output(Vec(n, dt))
  })

  // Route the payload according to the address
  // ...
```

Using Parameterized Bundles

- Instantiate that router with a `Port` that takes a `Payload` as a parameter

```
val router = Module(new NocRouter2(new
    Port(new Payload), 2))
```

Scala Option

- Scala's `Option[T]` is a wrapper around type T
- Potential non-existence
- Is either `Some(x)` or `None`

```
val opt: Option[Int] = Some(123)
if (o.isDefined)
  println(o.get)
else
  println("None")
```

Optional Ports

- IO ports may depend on configuration
- In Scala, this is represented as an `Option`
- Return a value wrapped in `Some` or represent the missing value as a `None`
- Could be used for debugging

```
dut.io.dbgPort.get(4).expect(123.U)
```

Example: Register File

```
class RegisterFile(debug: Boolean) extends Module {
  val io = IO(new Bundle {
    val rs1 = Input(UInt(5.W))
    val rs2 = Input(UInt(5.W))
...
    val rs2Val = Output(UInt(32.W))
    val dbgPort = if (debug)
      Some(Output(Vec(32, UInt(32.W)))) else None
  })
  val regfile =
     RegInit(VecInit(Seq.fill(32)(0.U(32.W))))
  io.rs1Val := regfile(io.rs1)
  io.rs2Val := regfile(io.rs2)
  when(io.wrEna) {
    regfile(io.rd) := io.wrData
  }
  if (debug) {
    io.dbgPort.get := regfile
  }}
```

Scala `tabulate`

- More general than `fill`
- Produce a new collection by calling an anonymous function
- Index is the single argument
- Can use the _ wildcat

```
Seq.fill(5)(0)
Seq.tabulate(5)(i => i * i)
Seq.tabulate(5)(_ + 1)
```

Scala's `apply()` Method

- Create a new instance without `new`

```
val p = Person("Jope Hacker")
```

- is translated during compilation to

```
val p = Person.apply("Jope Hacker")
```

- `apply()` is also used as access function for arrays
- No special syntax for arrays

Scala's `apply()` Method

- A *companion object* in Scala is an object
  - Declared in the same file as a class,
  - and has the same name as the class
- Add an `apply()` method to the companion object

```
class Person {
    var name = ""
}

object Person {
    def apply(name: String): Person = {
        var p = new Person
        p.name = name
        p
    }
}
```

Factory Methods

- Simpler component creation and use
- Usage similar to built-in components, such as `Mux`

```
val myAdder = Adder(x, y)
```

- A little bit more work on the component side
- Define an `apply` method on the companion object that returns the component (output port)

```
object Adder {
  def apply(a: UInt, b: UInt) = {
    val adder = Module(new Adder)
    adder.io.a := a
    adder.io.b := b
    adder.io.result
  }
}
```

Summary

- We use Scala to write hardware generators
- Get started with your project
- Next week: guest lecture by Emad on testing and CI