Chisel Bootcamp

https://github.com/freechipsproject/chisel-bootcamp

MODULE 0

* This tutorial *does not* yet cover SBT, build systems, backend flows for FPGA or ASIC processes, or analog circuits.*

Chisel : a Berkeley hardware construction DSL written in Scala.

LINK online execution on Jupiter: https://mybinder.org/v2/gh/freechipsproject/chisel-bootcamp/master

What you'll learn

- Why hardware designs are better expressed as generators, not instances I don't understand
- Basics and some advanced features of Scala, a modern programming language
- Basics and some advanced features of Chisel, a hardware description language embedded in Scala
- How to write unit tests for Chisel designs
- Basic introduction to some useful features in Chisel libraries, including dsptools and rocketchip.

*"The **benefits of Chisel** are in how you use it, not in the language itself. If you decide to write instances instead of generators, you will see fewer advantages of Chisel over Verilog. But if you take the time to learn how to write generators, then the power of Chisel will become apparent and you will realize you can never go back to writing Verilog."*

Instalation:

Windows:

Install the chisel-bootcamp repo.

Download the chisel-bootcamp as a zip file (or use a Windows git client) and unpack it in a directory you have access to. Ideally, you should put it in a path that has no spaces.

Não entendo como fazer

Install the customization script by moving `chisel-bootcamp/source/custom.js` to `%HOMEDRIVE%%HOMEPATH%\.jupyter\custom\custom.js`. If you already have a custom.js file, append this script to it.

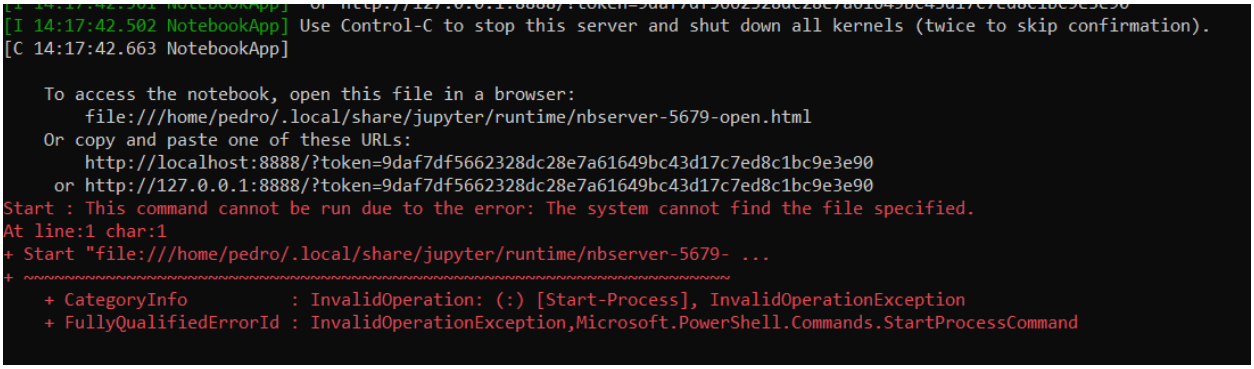
- - - - - - - - - - -

Cadence AWS Setup

Should I do this??

If you don't know what is Cadence AWS, or don't have access to Cadence AWS, skip this section.

Ubuntu from Windows:



Ubuntu: is okay

DEMO

to download and imports the dependencies needed for the demo.

```
val path = System.getProperty("user.dir") + "/source/load-ivy.sc"
interp.load.module(ammonite.ops.Path(java.nio.file.FileSystems.getDefault().getPath(path)))
```

Não entendo bem

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}
```

Not begging stuff a voltar a isto mais tarde.

1: Introduction to Scala

-Instanciar variáveis

```
var numberOfKittens = 6
val kittensPerHouse = 101
val alphabet = "abcdefghijklmnopqrstuvwxyz"
var done = false
```

-fazer print

```
println(alphabet)
done = true
```

```
abcdefghijklmnopqrstuvwxyz
```

val type can't be changed. Não se usa vírgulas

-Condições (if, for) implementadas da mesma forma.

"if" TRUE retorna um valor definido pela segunda linha.

```
for (i <- 0 to 7) { print(i + " ") }
println()
```

```
0 1 2 3 4 5 6 7
```

Use `until` instead of `to` for iterating from 0 to 6 (7 is r

```
for (i <- 0 until 7) { print(i + " ") }
println()
```

```
0 1 2 3 4 5 6
```

Add a `by` to increment by some fixed amount. The follow

```
for(i <- 0 to 10 by 2) { print(i + " ") }
println()
```

```
0 2 4 6 8 10
```

```
val randomList = List(scala.util
var listSum = 0
for (value <- randomList) {
  listSum += value
}
println("sum is " + listSum)
```

```
sum is 432274534
```

```
val likelyCharactersSet = if (alphabet.length == 26)
    "english"
else
    "not english"

println(likelyCharactersSet)
```

```
english
likelyCharactersSet: String = "english"
```

-declaração funções

```
// Simple scaling function with an input argument, e.g., times2(3) returns 6
// Curly braces can be omitted for short one-line functions.
def times2(x: Int): Int = 2 * x

// More complicated function
def distance(x: Int, y: Int, returnPositive: Boolean): Int = {
    val xy = x * y
    if (returnPositive) xy.abs else -xy.abs
}
```

```
defined function times2
defined function distance
```

def "nome_função" ("nome_variável": tipo_variável): tipo_variável_de_retorno da_função = {corpo_da_função}

Permite overloading de funcões

Permite definição de funções dentro de funções e recursividade (chamar a própria função)

-Listas

```
val x = 7
val y = 14
val list1 = List(1, 2, 3)
val list2 = x :: y :: y :: Nil       // An alternate notation for assembling a list

val list3 = list1 ++ list2           // Appends the second list to the first list
val m = list2.length
val s = list2.size

val headOfList = list1.head          // Gets the first element of the list
val restOfList = list1.tail          // Get a new list with first element removed

val third = list1(2)                 // Gets the third element of a list (0-indexed)
```

-Class & Instances of Classes

```
// WrapCounter counts up to a max value bas
class WrapCounter(counterBits: Int) {...}
```

```
val x = new WrapCounter(2)
```

Module 2

2.1 - First Chisel Module

-Setup

```
val path = System.getProperty("user.dir") + "/source/load-ivy.sc"
interp.load.module(ammonite.ops.Path(java.nio.file.FileSystems.getDefault().getPath(path)))
```

```
import chisel3._
import chisel3.util._
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}
```

-Chisel Code: Declare a new module definition

`Module`, `Passthrough`, that has one 4-bit input, `in`, and one 4-bit output, `out`. The module combinationally connects `in` and `out`, so `in` drives `out`.

```
// Chisel Code: Declare a new module definition
class Passthrough extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(4.W))
    val out = Output(UInt(4.W))
  })
  io.out := io.in
}
```

`Module` is a built-in Chisel class that all hardware modules must extend.

We **declare all our input and output ports in a special `io` `val`**. It must be called `io` and be an `IO` object or instance, which requires something of the form `IO(_instantiated_bundle_)`.

We declare a **new hardware struct type (Bundle)** that contains some named signals **`in` and `out` with directions Input and Output**, respectively.

`UInt(4.W)` → signal's hardware type, it is an unsigned integer of width 4.

`io.out := io.in` → We **connect our input port to our output port**, such that io.in *drives* io.out. Note that the ":=" operator is a *Chisel* operator that indicates that the **right-hand signal drives the left-hand signal**.

Converter para Verilog:

```
println(getVerilog(new Passthrough))
```

Devo testar o Código Verilog??

-Example module Generator (Chisel Code)

Mesma classe, mas tamanho do módulo é definido por parâmetro. No longer describes a single Module, but instead describes a family of modules parameterized.

```
class PassthroughGenerator(width: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out = Output(UInt(width.W))
  })
  io.out := io.in
}

// Let's now generate modules with different widths
println(getVerilog(new PassthroughGenerator(10)))
println(getVerilog(new PassthroughGenerator(20)))
```

```
Total FIRRTL Compile Time: 32.3 ms
module PassthroughGenerator(
  input        clock,
  input        reset,
  input  [9:0] io_in,
  output [9:0] io_out
);
  assign io_out = io_in; // @[cmd4.sc 6:10]
endmodule
```

```
Total FIRRTL Compile Time: 31.9 ms
module PassthroughGenerator(
  input         clock,
  input         reset,
  input  [19:0] io_in,
  output [19:0] io_out
);
  assign io_out = io_in; // @[cmd4.sc 6:10]
endmodule
```

-Testing:

Para os nossos módulos hardare, é necessário criar o código de teste *"Tester"*

Chisel has **built-in test features for testing** that you will explore throughout this bootcamp. The following example is a Chisel test harness that passes values to an instance of Passthrough's input port in and checks that the same value is seen on the output port out.

```
val testResult = Driver(() => new Passthrough()) {
  c => new PeekPokeTester(c) {
    poke(c.io.in, 0)     // Set our input to value 0
    expect(c.io.out, 0)  // Assert that the output correctly has 0
    poke(c.io.in, 1)     // Set our input to value 1
    expect(c.io.out, 1)  // Assert that the output correctly has 1
    poke(c.io.in, 2)     // Set our input to value 2
    expect(c.io.out, 2)  // Assert that the output correctly has 2
  }
}
assert(testResult)   // Scala Code: if testResult == false, will throw an erro
println("SUCCESS!!") // Scala Code: if we get here, our tests passed!
```

To set an input, we call **poke**.To check an output, we call **expect**.If we don't want to compare the output to an expected value we can **peek** the output instead. If all expect statements are true, then will return true.

// Viewing the firrtl for debugging É importante ver???

println(getFirrtl(new Passthro

Extra Note: `println` is a built-in Scala function that prints to the console. It **cannot** be used to print during circuit simulation because the generated circuit is FIRRTL or Verilog- not Scala.

-Printing

In [12]:
```
class PrintingModule extends Module {
    val io = IO(new Bundle {
        val in = Input(UInt(4.W))
        val out = Output(UInt(4.W))
    })
    io.out := 5.U / 2.U

    printf("Print during simulation: Input is %d\n", io.in)
    // chisel printf has its own string interpolator too
    printf(p"Print during simulation: IO is $io\n")

    println(s"Print during generation: Input is ${io.in}")
}

class PrintingModuleTester(c: PrintingModule) extends PeekPokeTester(c) {
    poke(c.io.in, 3)
    step(5) // circuit will print

    println(s"Print during testing: Output is ${peek(c.io.out)}")
}
chisel3.iotesters.Driver( () => new PrintingModule ) { c => new PrintingModuleTester(c) }
```

```
[info] [0.000] Elaborating design...
Print during generation: Input is UInt<4>(IO in unelaborated PrintingModule)
[info] [0.044] Done elaborating.
Total FIRRTL Compile Time: 32.2 ms
Total FIRRTL Compile Time: 28.6 ms
End of dependency graph
Circuit state created
[info] [0.000] SEED 1609839877412
Print during simulation: Input is 3
Print during simulation: IO is AnonymousBundle(in -> 3, out -> 2)
Print during simulation: Input is 3
Print during simulation: IO is AnonymousBundle(in -> 3, out -> 2)
Print during simulation: Input is 3
Print during simulation: IO is AnonymousBundle(in -> 3, out -> 2)
Print during simulation: Input is 3
Print during simulation: IO is AnonymousBundle(in -> 3, out -> 2)
Print during simulation: Input is 3
Print during simulation: IO is AnonymousBundle(in -> 3, out -> 2)
[info] [0.006] Print during testing: Output is 2
test PrintingModule Success: 0 tests passed in 10 cycles taking 0.008834 seconds
[info] [0.006] RAN 5 CYCLES PASSED
```

Out[12]:
```
defined class PrintingModule
defined class PrintingModuleTester
```

2.2: Combinational Logic

Scala Int: “val two = 1 + 1”

Chisel UInt: “ val utwo = 1.U + 1.U”

Adding type Chisel to type Scala results in error: “val error = 1 + 1.U”

Permite operadores de adição, subtração, multiplicação e divisão:

```
io.out_add := 1.U + 4.U
io.out_sub := 2.U - 1.U
io.out_mul := 4.U * 2.U
```

Módulo para Operações e o Tester:

```
class MyOperators extends Module {
  val io = IO(new Bundle {
    val in      = Input(UInt(4.W))
    val out_add = Output(UInt(4.W))
    val out_sub = Output(UInt(4.W))
    val out_mul = Output(UInt(4.W))
  })

  io.out_add := 1.U + 4.U
  io.out_sub := 2.U - 1.U
  io.out_mul := 4.U * 2.U
  io.out_mul := 4.U / 2.U
}
```

```
class MyOperatorsTester(c: MyOperators) extends PeekPokeTester(c) {
  expect(c.io.out_add, 5)
  expect(c.io.out_sub, 1)
  expect(c.io.out_mul, 8)
}
assert(Driver(() => new MyOperators) {c => new MyOperatorsTester(c)})
println("SUCCESS!!")
```

Tem operando de Mux e Concatenação:

```
val s = true.B
io.out_mux := Mux(s, 3.U, 0.U) // should return 3.U, since s is true
io.out_cat := Cat(2.U, 1.U)    // concatenates 2 (b10) with 1 (b1) to give 5
```

Mux: visto s é True, resultado é 3. 10 concatenado com 1 é 101 resulta em 5.

2.3: Control Flow

It is possible to issue multiple connect statements to the same component. When this happens, the last statement wins.

```
io.out := 3.U
io.out := 4.U
```

when, elsewhen, and otherwise:

```
when(someBooleanCondition) {
  // things to do when true
}.elsewhen(someOtherBooleanCondition) {
  // things to do on this condition
}.otherwise {
  // things to do if none of th boolean conditions are true
}
```

No exercicio, porque usar os WIRES em vez de novas VARIÀVEIS ????

2.4: Sequential Logic

Register: A `Reg` holds its output value until the rising edge of its clock, at which time it takes on the value of its input.

"Val register = Reg(UInt(12.W))"

Between calls to `poke()` and `expect`, there is a call to `step(1)`. This tells the test harness to tick the clock once, which will cause the register to pass its input to its output.

Note:

- The module has an input for clock (and reset) that you didn't add- this is the implicit clock
- `register` is updated on `posedge clock`

One important note is that Chisel distinguishes between types (like `UInt`) and hardware nodes (like the literal `2.U`, or the output of `myReg`). While

```
val myReg = Reg(UInt(2.W))
```

is legal because a Reg needs a data type as a model,

```
val myReg = Reg(2.U)
```

is an error because `2.U` is already a hardware node and can't be used as a model.

Object RegNext:

```
// register bitwidth is inferred from io.out
io.out := RegNext(io.in + 1.U)
```

RegInit: The way to create a register that resets to a given value is with `RegInit`.

For instance, a 12-bit register initialized to zero can be created with the following. Both versions below are valid and do the same thing:

```
val myReg = RegInit(UInt(12.W), 0.U)
val myReg = RegInit(0.U(12.W))
```

Explicit clock and reset

Chisel modules have a default clock and reset that are implicitly used by every register created inside them.

Clocks and resets can be overridden separately or together with *withClock() {}, withReset() {}, and withClockAndReset() {}.*

Reset is always synchronous and of type Bool. Clocks have their own type in Chisel (Clock). Bools can be converted to Clocks by calling *asClock().*

Chisel-testers do not currently have complete support for multi-clock designs.

2.5: FIR Filter

Or, a signals definition:

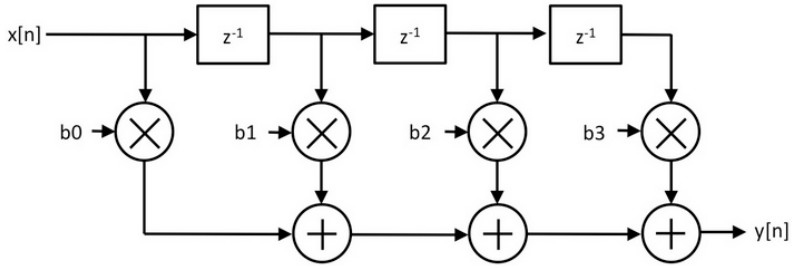
$y[n]=b0x[n]+b1x[n-1]+b2x[n-2]+...$

$y[n]$ is the output signal at time n

$x[n]$ is the input signal

bi are the filter coefficients or impulse response

$n-1$, $n-2$, ... are time n delayed by 1, 2, ... cycles



Module:

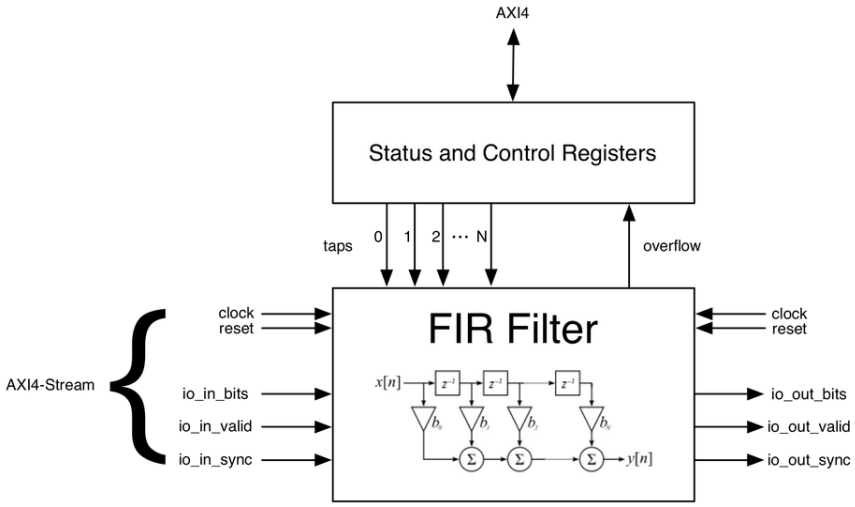
```
class My4ElementFir(b0: Int, b1: Int, b2: Int, b3: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  val state_b1 = RegNext(io.in, 0.U)
  val state_b2 = RegNext(state_b1, 0.U)
  val state_b3 = RegNext(state_b2, 0.U)
  io.out := (io.in * b0.U(8.W)) + (state_b1 * b1.U(8.W)) + (state_b2 * b2.U(8.W)) + (state_b3 * b3.U(8.W))
}
```

(Nao muito relevante)

DspBlock: A Digital Signal Processing Block has:

- AXI-4 Stream input and output
- Memory-mapped status and control (in this example, AXI4)



`DspBlock`s use diplomatic interfaces from rocket. (diplomacy interface – definição prioridades)

2.5: ChiselTest

	iotesters	ChiselTest
poke	poke(c.io.in1, 6)	c.io.in1.poke(6.U)
peek	peek(c.io.out1)	c.io.out1.peek(6.U)
expect	expect(c.io.out1, 6)	c.io.out1.expect(6.U)
step	step(1)	c.io.clock.step(1)
initiate	Driver.execute(...) { c =>	test(...) { c =>

```
val testResult = Driver(() => new Passthrough()) {
  c => new PeekPokeTester(c) {
    poke(c.io.in, 0)      // Set our input to value 0
    expect(c.io.out, 0)   // Assert that the output co
    poke(c.io.in, 1)      // Set our input to value 1
    expect(c.io.out, 1)   // Assert that the output co
    poke(c.io.in, 2)      // Set our input to value 2
    expect(c.io.out, 2)   // Assert that the output co
  }
}
assert(testResult)    // Scala Code: if testResult == 
println("SUCCESS!!") // Scala Code: if we get here, o
```

```
test(new PassthroughGenerator(16)) { c =>
    c.io.in.poke(0.U)      // Set our input
    c.io.out.expect(0.U)   // Assert that t
    c.io.in.poke(1.U)      // Set our input
    c.io.out.expect(1.U)   // Assert that t
    c.io.in.poke(2.U)      // Set our input
    c.io.out.expect(2.U)   // Assert that t
}
```

QUEUE

```
case class QueueModule[T <: Data](ioType: T, entries: Int) extends MultiIOModule {
  val in = IO(Flipped(Decoupled(ioType)))
  val out = IO(Decoupled(ioType))
  out <> Queue(in, entries)
}
```

`Decoupled` takes a chisel data type and provides it with `ready` and `valid` signals.

The `QueueModule` passes through data whose type is determined by `ioType`. There are `entries` state elements inside the `QueueModule` meaning it can hold that many elements before it exerts backpressure.

NÂO CONSIGO EXECUTAR O CODIGO

EnqueueNow and expectDequeueNow

method	description
enqueueNow	Add (enqueue) one element to a `Decoupled` input interface
expectDequeueNow	Removes (dequeues) one element from a `Decoupled` output interface

There is some required boiler plate `initSource`, `setSourceClock`, etc that is necessary to ensure that the `ready` and `valid` fields are all initialized correctly at the beginning of the test.

```
test(QueueModule(UInt(9.W), entries = 200)) { c =>
    // Example testsequence showing the use and behavior of Queue
    c.in.initSource()
    c.in.setSourceClock(c.clock)
    c.out.initSink()
    c.out.setSinkClock(c.clock)

    val testVector = Seq.tabulate(200){ i => i.U }

    testVector.zip(testVector).foreach { case (in, out) =>
      c.in.enqueueNow(in)
      c.out.expectDequeueNow(out)
    }
}
```

EnqueueSeq and DequeueSeq

method	description
enqueueSeq	Continues to add (enqueue) elements from the `Seq` to a `Decoupled` input interface, one at a time, until the sequence is exhausted
expectDequeueSeq	Removes (dequeues) elements from a `Decoupled` output interface, one at a time, and compares each one to the next element of the `Seq`

```
test(QueueModule(UInt(9.W), entries = 200)) { c =>
    // Example testsequence showing the use and behavior of Queue
    c.in.initSource()
    c.in.setSourceClock(c.clock)
    c.out.initSink()
    c.out.setSinkClock(c.clock)

    val testVector = Seq.tabulate(100){ i => i.U }

    c.in.enqueueSeq(testVector)
    c.out.expectDequeueSeq(testVector)
}
```

Fork and Join

method	description
fork	launches a concurrent code block, additional forks can be run concurrently to this one via the .fork appended to end of the code block of the preceeding fork
join	re-unites multiple related forks back into the calling thread

Running sections of a unit test concurrently.

The threads created by fork are run in a deterministic order

Example:

```
test(QueueModule(UInt(9.W), entries = 200)) { c =>
    // Example testsequence showing the use and behavior of Queue
    c.in.initSource()
    c.in.setSourceClock(c.clock)
    c.out.initSink()
    c.out.setSinkClock(c.clock)

    val testVector = Seq.tabulate(300){ i => i.U }

    fork {
        c.in.enqueueSeq(testVector)
    }.fork {
        c.out.expectDequeueSeq(testVector)
    }.join()
}
```

Module 3

3.1 - Generators: Parameters

Generators are programs that take some circuit parameters and produce a circuit description.

Example: Parameterized Scala Object

```
class ParameterizedScalaObject(param1: Int, param2: String) {
  println(s"I have parameters: param1 = $param1 and param2 = $param2")
}
val obj1 = new ParameterizedScalaObject(4,     "Hello")
val obj2 = new ParameterizedScalaObject(4 + 2, "World")
```

Example: Parameterized Chisel Object

```
class ParameterizedWidthAdder(in0Width: Int, in1Width: Int, sumWidth: Int) extends Module {
  require(in0Width >= 0)
  require(in21Width >= 0)
  require(sumWidth >= 0)
  val io = IO(new Bundle {
    val in0 = Input(UInt(in0Width.W))
    val in1 = Input(UInt(in1Width.W))
    val sum = Output(UInt(sumWidth.W))
  })
  // a +& b includes the carry, a + b does not
  io.sum := io.in0 +& io.in1
}

println(getVerilog(new ParameterizedWidthAdder(1, 4, 6)))
```

`require(...)`: These are pre-elaboration assertions, which are useful when your generator only works with certain parameterizations.

There is a separate construct for simulation-time assertions called `assert(...)`.

Option and Default Arguments

Example: Erroneous Map Index Call

In the following example, we have a map containing several key/value pairs. If we try to access a missing key/value pair, then we get a runtime error:

```
val map = Map("a" -> 1)
val a = map("a")
println(a)
val b = map("b")
println(b)
```

```
1
java.util.NoSuchElementException: key not found: b
```

Example: Getting Uncertain Indices

However, `Map` provides another way to access a key's value, through the **get** method. Using this returns a value of abstract class `Option`. `Option` has two subclasses, `Some` and `None`.

```
val map = Map("a" -> 1)
val a = map.get("a")
println(a)
val b = map.get("b")
println(b)
```

```
Some(1)
None

map: Map[String, Int] = Map("a" -> 1)
a: Option[Int] = Some(1)
b: Option[Int] = None
```

`Option` is extremely important because it lets users use a match statement to check Scala types and values.

Example: Get Or Else!

`Option` has a `get` method, which errors if called on `None`.

```
val some = Some(1)
val none = None
println(some.get)           // Returns 1
// println(none.get)        // Errors!
println(some.getOrElse(2)) // Returns 1
println(none.getOrElse(2)) // Returns 2
```

```
1
1
2

some: Some[Int] = Some(1)
none: None.type = None
```

Example: Optional Reset

Sometimes, a parameter doesn't have a good default value. `Option` can be used with a default value of `None` in these situations.

```
class DelayBy1(resetValue: Option[UInt] = None) extends Module {
    val io = IO(new Bundle {
        val in  = Input( UInt(16.W))
        val out = Output(UInt(16.W))
    })
    val reg = if (resetValue.isDefined) { // resetValue = Some(number)
        RegInit(resetValue.get)
    } else { //resetValue = None
        Reg(UInt())
    }
    reg := io.in
    io.out := reg
}
```

If `resetValue = None`, which is the default, the register will have no reset value and be initialized to garbage. This avoids the common but ugly case of using values outside the normal range to indicate "none", like using -1 as the reset value to indicate that this register is not reset.

Match/Case Statements : something like a C *switch* statement

```
// y is an integer variable defined somewhere else in the code
val y = 7
/// ...
val x = y match {
  case 0 => "zero" // One common syntax, preferred if fits in one line
  case 1 =>        // Another common syntax, preferred if does not fit in one line.
      "one"        // Note the code block continues until the next case
  case 2 => {      // Another syntax, but curly braces are not required
      "two"
  }
  case _ => "many" // _ is a wildcard that matches all values
}
println("y is " + x)
```

```
y is many
```

```
y: Int = 7
x: String = "many"
```

The use of underscore as a wildcard, to handle any value not found.

Example: Multiple Value Matching

Multiple variables can be matched at the same time.

```
def animalType(biggerThanBreadBox: Boolean, meanAsCanBe: Boolean): String = {
  (biggerThanBreadBox, meanAsCanBe) match {
    case (true, true) => "wolverine"
    case (true, false) => "elephant"
    case (false, true) => "shrew"
    case (false, false) => "puppy"
  }
}
println(animalType(true, true))
```

Example: Type Matching

The types of all objects are known during runtime. We can use **match statements** to use this type information to dictate control flow:

```
val sequence = Seq("a", 1, 0.0)
sequence.foreach { x =>
  x match {
    case s: String => println(s"$x is a String")
    case s: Int    => println(s"$x is an Int")
    case s: Double => println(s"$x is a Double")
    case _ => println(s"$x is an unknown type!")
  }
}
```

Example: Multiple Type Matching

*Note that you **must** use an _ when matching.*

```
val sequence = Seq("a", 1, 0.0)
sequence.foreach { x =>
  x match {
    case _: Int | _: Double => println(s"$x is a number!")
    case _ => println(s"$x is an unknown type!")
  }
}
```

```
a is an unknown type!
1 is a number!
0.0 is a number!
```

```
sequence: Seq[Any] = List("a", 1, 0.0)
```

Example: Optional Reset Matching

`DelayBy1` module with the match construct instead of `if/else`.

```
class DelayBy1(resetValue: Option[UInt] = None) extends Module {
  val io = IO(new Bundle {
    val in  = Input( UInt(16.W))
    val out = Output(UInt(16.W))
  })
  val reg = resetValue match {
    case Some(r) => RegInit(r)
    case None    => Reg(UInt())
  }
  reg := io.in
  io.out := reg
}
```

I/O Optional Fields

Sometimes we want IOs to be optionally included or excluded.

getOrElse()

```
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)
```

Zero-Wire

```
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
```

Implicit Arguments

used to reduce boilerplate code.

In a given scope, **there can only be one implicit value of a given type**.

When we call **the function with implicit arguments**, we either omit the implicit argument list (letting the compiler find it for us), or explicitly provide an argument (which can be different than the implicit value).

implement logging in a Chisel generator. Não percebi o Còdigo complicado

Example: Implicit Conversion

In the following example, we have two classes, `Animal` and `Human`. `Animal` has a `species` field, but `Human` does not. However, by implementing an implicit conversion, we can call `species` on a `Human`.

```
class Animal(val name: String, val species: String)
class Human(val name: String)
implicit def human2animal(h: Human): Animal = new Animal(h.name, "Homo sapiens")
val me = new Human("Adam")
println(me.species)
```

```
Homo sapiens
```

Generally, implicits can make your code confusing. First try inheritance, traits, or method overloading.

3.2 - Generators: Collections

Golden Model:

Checking that the two different methods, hardware and software, are in sync at each step.

Testar o circuito FIR para N coeficientes.

Primeiro passo criar o **GOLDEN MODEL**. (scala)

Depois testar o Golden Model.

Depois testar o Modelo Hardware dando aos dois modelos os mesmos valores e *"expect"* que *out* seja igual ao resultado do *Golden Model*

Array mutável de registos

```
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)
}
```

Consts é variável parametro de numero de constantes/registos

Vec => "Vectors"

Instanciar conjunto de constantes como inputs, através um vetor dinâmico:

```
class MyManyDynamicElementVecFir(length: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
    val consts = Input(Vec(length, UInt(8.W)))
  })
```

Instanciar cadeia de registos através desse Vetor de constantes:

```
// Reference solution
val regs = RegInit(VecInit(Seq.fill(length - 1)(0.U(8.W))))
for(i <- 0 until length - 1) {
    if(i == 0) regs(i) := io.in
    else       regs(i) := regs(i - 1)
}
```

Registos são a nossa memória. Ao fazermos um registo de registos temos um Vetor endereçável.

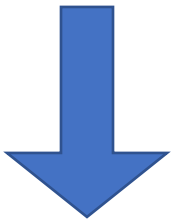
3.3: Higher-Order Functions

Redução do código de Convolução do FIR

```
val muls = Wire(Vec(length, UInt(8.W)))
for(i <- 0 until length) {
  if(i == 0) muls(i) := io.in * io.consts(i)
  else       muls(i) := regs(i - 1) * io.consts(i)
}

val scan = Wire(Vec(length, UInt(8.W)))
for(i <- 0 until length) {
  if(i == 0) scan(i) := muls(i)
  else scan(i) := muls(i) + scan(i - 1)
}

io.out := scan(length - 1)
```



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

- `(taps zip io.consts)` takes two lists, `taps` and `io.consts`, and combines them into one list where each element is a tuple of the elements at the inputs at the corresponding position. Concretely, its value would be `[(taps(0), io.consts(0)), (taps(1), io.consts(1)), ..., (taps(n), io.consts(n))]`.

- `.map { case (a, b) => a * b }` takes a tuple of two elements returns their product. to the elements of the list, and returns the result. In this case, the result is equivalent to `muls` in the verbose example, and has the value `[taps(0) * io.consts(0), taps(1) * io.consts(1), ..., taps(n) * io.consts(n)]`.

- Finally, `.reduce(_ + _)` also applies the addition of elements to elements of the list. However, it takes two arguments: the first is the current accumulation, and the second is the list element. The result would then be, assuming left-to-right traversal, `(((muls(0) + muls(1)) + muls(2)) + ...) + muls(n)`, with the result of deeper-nested parentheses evaluated first. This is the output of the convolution.

Functions as Arguments

Functions like *map* and *reduce* are called higher-order functions: they are functions that take functions as arguments.

Ways of specifying functions

- **"_ + _" :** underscore (_) to refer to each argument.
- "`(a, b) => a + b`"
- "`case (a, b) => a * b`"

Example: Map

`List[A].map` has type signature `map[B](f: (A) ⇒ B): List[B`

Think of types A and B as `Int`s or `UInt`s, meaning they could be software or hardware types.

```
println(List(1, 2, 3, 4).map(x => x + 1))  // explicit argument list in function
//List(2, 3, 4, 5)
println(List(1, 2, 3, 4).map(_ + 1))  // equivalent to the above, but implicit arguments
//List(2, 3, 4, 5)
println(List(1, 2, 3, 4).map(_.toString + "a"))// the output element type can be different from the input element type
//List(1a, 2a, 3a, 4a)
println(List((1, 5), (2, 6), (3, 7), (4, 8)).map { case (x, y) => x*y })//this unpacks a tuple, use of curly braces
//List(5, 12, 21, 32)

// Related: Scala has a syntax for constructing lists of sequential numbers
println(0 to 10)  // to is inclusive , the end point is part of the result
//Range 0 to 10
println(0 until 10)  // until is exclusive at the end, the end point is not part of the result
//Range 0 until 10

// Those largely behave like lists, and can be useful for generating indices:
val myList = List("a", "b", "c", "d")
println((0 until 4).map(myList(_)))
//Vector(a, b, c, d
```

Example: zipWithIndex

`List.zipWithIndex` has type signature `zipWithIndex: List[(A, Int)]`.

So `List("a", "b", "c", "d").zipWithIndex` would return `List(("a", 0), ("b", 1), ("c", 2), ("d", 3))`

Example: Reduce

`List[A].map` has type signature similar to `reduce(op: (A, A) ⇒ A): A`

Example: Fold

`List[A].fold` is very similar to reduce, except that you can specify the initial accumulation value.

```
println(List(1, 2, 3, 4).fold(1)(_ + _))  //11
```

"decoupled" interface: **'valid'** indicates that the producer has put valid data in **'bits'**, and **'ready'** indicates that the consumer is ready to accept the data this cycle. No requirements are placed on the signaling of ready or valid.

```
val io = IO(new Bundle {
  val in = Flipped(Decoupled(UInt(8.W)))
  val out = Decoupled(UInt(8.W))
})
```

```
val io = IO(new Bundle {
  val in = new Bundle {
    val valid = Input(Bool())
    val ready = Output(Bool())
    val bits  = Input(UInt(8.W))
  }
  val out = new Bundle {
    val valid = Output(Bool())
    val ready = Input(Bool())
    val bits  = Output(UInt(8.W))
  }
}
```

3.4: Functional Programming

We can assign a function to a `val` and pass it to classes, objects, or other functions as an argument.

Passar funções para map()

```
val plus1 = (x: Int) => x + 1
val times2 = (x: Int) => x * 2

// pass it to map, a list function
val myList = List(1, 2, 5, 9)
val myListPlus = myList.map(plus1)//List(2, 3, 6, 10)
val myListTimes = myList.map(times2)//List(2, 4, 10, 18)
```

Recursividade

```
def opN(x: Int, n: Int, op: Int => Int): Int = {
  if (n <= 0) { x }
  else { opN(op(x), n-1, op) }
}
```

Example: Functions vs. Objects

both x and y call the nextInt function, but x is evaluated immediately and y is a function

```
// both x and y call the nextInt function, but x is evaluated immediately and y is a function
val x = Random.nextInt
def y = Random.nextInt
```

.Reduce()

```
abc.reduceLeft(add)
// op: A + B = AB
// op: AB + C = ABC
// res: String = ABC
```

.Fold()

```
abc.foldLeft("z")(add)
// op: z + A = zA
// op: zA + B = zAB
// op: zAB + C = zABC
// res: String = zABC
```

.Scan()

```
abc.scanLeft("z")(add)
// op: z + A = zA
// op: zA + B = zAB
// op: zAB + C = zABC
// res: List[String] = List(z, zA, zAB, zABC)
```

FixedPoint variable - The two most common classes of **fixed-point types** are decimal and binary.

Como fazer operação aritmética com dois conjuntos de variáveis e reduzir numa só:

Exemplo (multiplicação a com b, e sumatório dos resultados)

```
val mac = io.in.zip(io.weights).map{ case(a: FixedPoint, b: FixedPoint) => a*b}.reduce(_+_)
```

the type **`Int => String`**, is equivalent to the type **`Function1[Int,String]`** i.e. a function that takes an argument of type **`Int`** and returns a **`String`**.

```
<= // Less-than-or-equals comparison of BigDecimals
```

3.5: Object Oriented Programming

Abstract Classes

The same, They can define many unimplemented values that subclasses must implement. Any object can only directly inherit from one parent abstract class.

Traits

Very similar to abstract classes but:

-A class can inherit from multiple traits -a trait cannot have constructor parameters

```
trait HasFunction {
  def myFunction(i: Int): Int
}
trait HasValue {
  val myValue: String
  val myOtherValue = 100
}
class MyClass extends HasFunction with HasValue {
  override def myFunction(i: Int): Int = i + 1
  val myValue = "Hello World!"
}
```

Objects

Static classes

```
object MyObject {
  def hi: String = "Hello World!"
  def apply(msg: String) = msg
}
println(MyObject.hi)
```

Companion Objects

When a class and an object share the same name and defined in the same file, the object is called a **companion object**. When you use `new` before the class/object name, it will instantiate the class. If you don't use `new`, it will reference the object:

```
object Lion {
    def roar(): Unit = println("I'M AN OBJECT!")
}
class Lion {
    def roar(): Unit = println("I'M A CLASS!")
}
new Lion().roar()
Lion.roar()
```

Companion objects are usually used for the following reasons:

1. to contain constants related to the class
2. to execute code before/after the class constructor
3. to create multiple constructors for a class

```
val myModule = Module(new MyModule)
```

you are calling the **Module companion object**, so Chisel can run background code before and after instantiating `MyModule`.

Case Classes

- Allows external access to the class parameters
- Eliminates the need to use new when instantiating the class
- Automatically creates an unapply method that supplies access to all of the class Parameters.
- Cannot be subclassed from

```
class Nail(length: Int) // Regular class
val nail = new Nail(10) // Requires the `new` keyword
// println(nail.length) // Illegal! Class constructor parameters are not by default externally visible

class Screw(val threadSpace: Int) // By using the `val` keyword, threadSpace is now externally visible
val screw = new Screw(2)          // Requires the `new` keyword
println(screw.threadSpace)

case class Staple(isClosed: Boolean) // Case class constructor parameters are, by default, externally visible
val staple = Staple(false)           // No `new` keyword required
println(staple.isClosed)
```

3.6: Generators: Types

Static Types

All objects in Scala have a type

```
println(10.getClass)
println(10.0.getClass)
println("ten".getClass)
```

```
int
double
class java.lang.String
```

recommended that you **define input and output types for all function declarations**.

Scala vs Chisel Types

is legal because `0.U` is of type `UInt` (a Chisel type), whereas

```
val a = Wire(UInt(4.W))
a := 0.U
```

is illegal because 0 is type `Int` (a Scala type).

```
val a = Wire(UInt(4.W))
a := 0
```

This is also true of `Bool`, a Chisel type which is distinct from `Boolean`.

```
val bool = Wire(Bool())
val boolean: Boolean = false
// legal
when (bool) { ... }
if (boolean) { ... }
// illegal
if (bool) { ... }
when (boolean) { ... }
```

If you make a mistake and mix up `UInt` and `Int` or `Bool` and `Boolean`, the Scala compiler will generally catch it for you.

Type Casting in Chisel

The most general is **`asTypeOf()`**, which is shown below. Some chisel objects also define **`asUInt()`** and **`asSInt()`** as well as some others.

SEMPRE QUE COLOCO DATA COMO TIPO "DATA" DÁ ERRO!!!????

Unapply

Give match statements the ability to both match on types and **extract values** from those types during the matching.

It allows doing:

```
case class Something(a: String, b: Int)
val a = Something("A", 3)
a match {
    case Something("A", value) => value
    case Something(str, 3)     => 0
}
```

Finally, you can directly embed condition checking into match statements, as demonstrated by the third of these equivalent examples:

```
case SomeGeneratorParameters(_, sw, false) => sw * 2
case s@SomeGeneratorParameters(_, sw, false) => s.sw * 2
case s: SomeGeneratorParameters if s.pipelineMe => s.sw * 2
```

All these syntaxes are enabled by a Scala unapply method contained in a class's companion object.

Partial Funcitons

a partial function may not have a value for a particular input. This can be tested with `isDefinedAt(...)`.

```
// Defined for 1, 2, 3, etc.
val partialFunc1: PartialFunction[Int, String] = {
  case i if (i + 1) % 3 == 0 => "Something"
}
printAndAssert("partialFunc1.isDefinedAt(2)", partialFunc1.isDefinedAt(2), true)
```

Type Generics

also known as polymorphism. Classes can be polymorphic in their types.

Functions can also be polymorphic in their input or output types.

Note that the `=> T` syntax encodes an anonymous function

```scala
def time[T](block: => T): T = {
    val t0 = System.nanoTime()
    val result = block
    val t1 = System.nanoTime()
    val timeMillis = (t1 - t0) / 1000000.0
    println(s"Block took $timeMillis milliseconds!")
    result
}
```

Type Generics with Typeclasse

Example: DspComplex

```scala
class DspComplex[T <: Data:Ring](val real: T, val imag: T) extends Bundle { ... }
```

`DspComplex`
is a type-generic container. That means the real and imaginary parts of a complex number can be any type as long as they satisfy the type constraints, given by: `T <: Data : Ring`.

`T <: Data`
means `T` is a subtype of `chisel3.Data`, the base type for Chisel objects. This means that `DspComplex` only works for objects that are Chisel types and not arbitrary Scala types.

`T : Ring`
means that a Ring typeclass implementation for `T` exists. `Ring` typeclasses define `+` and `*` operators as well as additive and multiplicative identities. **dsptools** defines typeclasses for commonly used Chisel types.

Module 4:

4.1 Introduction to FIRRTL

What is FIRRTL?

Instead of directly emitting Verilog, Chisel emits an intermediate representation called FIRRTL, which represents the elaborated (parameter-resolved) RTL instance.

The FIRRTL AST

FIRRTL representation can be serialized as a String, but internally, it is a data structure called an AST (abstract syntax tree).

This datastructure is a tree of nodes, where one node can contain children nodes. There are no cycles in this datastructure.

FIRRTL Node Descriptions

Circuit – is the root node of any Firrtl datastructure

FirrtlNode Declaration

```
Circuit(info: Info, modules: Seq[DefModule], main: String)
```

Concrete Syntax

```
circuit Adder:
  ... //List of modules
```

In-memory Representation

```
Circuit(NoInfo, Seq(...), "Adder")
```

Modules - are the unit of modularity within Firrtl . Declaring an instance of a module has its own concrete syntax and AST representation).

Each Module has a name, and a list of ports, and a body containing its implementation.

FirrtlNode declaration

```
Module(info: Info, name: String, ports: Seq[Port], body: Stmt) extends DefModule
```

Concrete Syntax

```
module Adder:
  ... // list of ports
  ... // statements
```

In-memory representation

```
Module(NoInfo, "Adder", Seq(...), )
```

Port - defines part of a Module's io, and has a name, direction (input or output), and type.

FirrtlNode Declaration

```
class Port(info: Info, name: String, direction: Direction, tpe: Type)
```

Concrete Syntax

```
input x: UInt
```

In-memory representation

```
Port(NoInfo, "x", INPUT, UIntType(UnknownWidth))
```

Statement - is used to describe the components within a module and how they interact.

A wire declaration - containing a name and type. It can be both a source (connected *from*) and a sink (connected *to").

FirrtlNode declaration

```
DefWire(info: Info, name: String, tpe: Type)
```

Concrete syntax

```
wire w: UInt
```

In-memory Representation

```
DefWire(NoInfo, "w", UIntType(UnknownWidth))
```

Register declaration - containing a name, type, clock signal, reset signal, and reset value.

FirrtlNode declaration

```
DefRegister(info: Info, name: String, tpe: Type, clock: Expression, reset: Expression, init: Expression)
```

Connection - represents a directioned connection from a source to a sink

FirrtlNode declaration

```
Connect(info: Info, loc: Expression, expr: Expression)
```

Expression - represent references to declared components or logical and arithmetic operations.

Reference - A reference to a declared component, such as a wire, register, or port. It has a name and type field. Note that it does not contain a pointer to the actual declaration, but instead just contains the name as a String.

FirrtlNode declaration

```
Reference(name: String, tpe: Type)
```

DoPrim - an anonymous primitive operation, such as Add, Sub, or And, Or, or subword-selection (Bits). The type of operation is indicated by the op: PrimOp field. Note that the number of required arguments and constants are determined by the op.

FirrtlNode declaration

```
DoPrim(op: PrimOp, args: Seq[Expression], consts: Seq[BigInt], tpe: Type)
```

4.2: FIRRTL AST Traversal

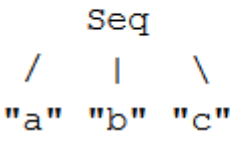
The IR datastructure is a tree, where each IR node can have some number of children nodes (which in turn can have more children nodes, etc.). IR nodes without children are called leaves.

Different IR nodes can have different children types.

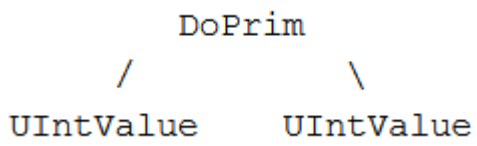
Node	Children
Circuit	DefModule
DefModule	Port, Statement
Port	Type, Direction
Statement	Statement, Expression, Type
Expression	Expression, Type
Type	Type, Width
Width	
Direction	

Seq.map

```
val s = Seq("a", "b", "c")
```



Firrtl's map – (1 +1)



Pre-order traversal

To traverse a Firrtl tree, we use `map` to write recursive functions which visit every child of every node we care about.

```
def f(regNames: mutable.HashSet[String]())(s: Statement): Statement = s match {
  // If register, add name to regNames
  case r: DefRegister =>
    regNames += r.name
    r // Return argument unchanged (ok because DefRegister has no Statement children)
  // If not, apply f(regNames) to all children Statement
  case _ => s map f(regNames) // Note that f(regNames) is of type Statement=>Statement
}
```

Post-order traversal¶

```
def f(regNames: mutable.HashSet[String]())(s: Statement): Statement =
  // Not we immediately recurse to the children nodes, then match
  s map f(regName) match {
    // If register, add name to regNames
    case r: DefRegister =>
      regNames += r.name
      r // Return argument unchanged (ok because DefRegister has no Statement children)
    // If not, return s
    case _ => s // Note that all Statement children of s have had f(regNames) already applied
  }
```

Module 4.3: Common Pass Idioms

Adding statements

We first need to traverse the AST to every Statement and Expression. Then, when we see a DoPrim, we need to add a new DefNode to the module's body and insert a reference to that DefNode in place of the DoPrim.

```
o <= add(x, add(y, z))
```

```
node GEN_1 = add(y, z)
o <= add(x, GEN_1)
```

```
object Splitter extends Pass {
  def name = "Splitter!"
  /** Run splitM on every module **/
  def run(c: Circuit): Circuit = c.copy(modules = c.modules map(splitM(_)))
  /** Run splitS on the body of every module **/
  def splitM(m: DefModule): DefModule = m map splitS(Namespace(m))
  /** Run splitE on all children Expressions.
    * If stmts contain extra statements, return a Block containing them and
    *    the new statement; otherwise, return the new statement. */
  def splitS(namespace: Namespace)(s: Statement): Statement = {
    val block = mutable.ArrayBuffer[Statement]()
    s match {
      case s: HasInfo =>
        val newStmt = s map splitE(block, namespace, s.info)
        block.length match {
          case 0 => newStmt
          case _ => Block(block.toSeq :+ newStmt)
        }
      case s => s map splitS(namespace)
    }
  /** Run splitE on all children expressions.
    * If e is a DoPrim, add a new DefNode to block and return reference to
    * the DefNode; otherwise return e.*/
  def splitE(block: mutable.ArrayBuffer[Statement], namespace: Namespace,
             info: Info)(e: Expression): Expression = e map splitE(block, namespace, info) match {
    case e: DoPrim =>
      val newName = namespace.newTemp
      block += DefNode(info, newName, e)
      Ref(newName, e.tpe)
    case _ => e
  }
}
```

Deleting statements

```
node y = UInt(1)
o <= add(x, y)
```

```
o <= add(x, UInt(1))
```

We first need to traverse the AST to every Statement and Expression. Then, when we see a DefNode pointing to a Literal, we need to store it into a hashmap and return an EmptyStmt (thus deleting that DefNode). Then, whenever we see a reference to the deleted DefNode, we must insert the corresponding Literal.

```
object Inliner extends Pass {
  def name = "Inliner!"
  /** Run inlineM on every module **/
  def run(c: Circuit): Circuit = c.copy(modules = c.modules map(inlineM(_)))
  /** Run inlineS on the body of every module **/
  def inlineM(m: DefModule): DefModule = m map inlineS(mutable.HashMap[String, Expression]())
  /** Run inlineE on all children Expressions, and then run inlineS on children statements.
    * If statement is a DefNode containing a literal, update values and
    *   return EmptyStmt; otherwise return statement. */
  def inlineS(values: mutable.HashMap[String, Expression])(s: Statement): Statement =
    s map inlineE(values) map inlineS(values) match {
      case d: DefNode => d.value match {
        case l: Literal =>
          values(d.name) = l
          EmptyStmt
        case _ => d
      }
      case o => o
    }
  /** If e is a reference whose name is contained in values,
    *   return values(e.name); otherwise run inlineE on all
    *   children expressions.*/
  def inlineE(values: mutable.HashMap[String, Expression])(e: Expression): Expression = e match {
    case e: Ref if values.contains(e.name) => values(e.name)
    case _ => e map inlineE(values)
  }
}
```

Module 4.4: A FIRRTL Transform Example

Counting Adders Per Module

To visit all Firrtl IR nodes in a circuit, we write functions that recursively walk down this tree. To record statistics, we will pass along a Ledger class and use it when we come across an add op:

….

Now, let's define a FIRRTL Transform that walks the circuit and updates our `Ledger` whenever it comes across an adder (`DoPrim` with op argument `Add`).

(Percorrer toda a árvore desde Módulos → statement → Expression e aqui fazer match case para encontrar o adder)

Qual o objetivo do FIRRTL? Analisar os circuitos construídos?

Chisel3 Cheat Sheet

Version 0.5 (beta): September 5, 2019

Notation In This Document:
For Functions and Constructors:
Arguments given as `kwd:type` (name and type(s))
Arguments in brackets (`[...]`) are optional.
For Operators:
`c`, `x`, `y` are Chisel `Data`; `n`, `m` are Scala `Int`
`w(x)`, `w(y)` are the widths of `x`, `y` (respectively)
`minVal(x)`, `maxVal(x)` are the minimum or maximum possible values of `x`

Basic Chisel Constructs

Chisel Wire Operators:

```
// Allocate a as wire of type UInt()
val x = Wire(UInt())
x := y  // Connect wire y to wire x
```

When executes blocks conditionally by `Bool`, and is equivalent to Verilog `if`

```
when(condition1) {
  // run if condition1 true and skip rest
} .elsewhen(condition2) {
  // run if condition2 true and skip rest
} .otherwise {
  // run if none of the above ran
}
```

Switch executes blocks conditionally by data

```
switch(x) {
  is(value1) {
    // run if x === value1
  }
  is(value2) {
    // run if x === value2
  }
}
```

Enum generates value literals for enumerations

```
val s1::s2:: ... ::sn::Nil
    = Enum(nodeType:UInt, n:Int)
```

`s1`, `s2`, ..., `sn` will be created as `nodeType` literals with distinct values
`nodeType` type of `s1`, `s2`, ..., `sn`
`n` element count

Math Helpers:

`log2Ceil(in:Int): Int` log_2(`in`) rounded up
`log2Floor(in:Int): Int` log_2(`in`) rounded down
`isPow2(in:Int): Boolean` `True` if `in` is a power of 2

Basic Data Types

Constructors:

`Bool()`	type, boolean value
`true.B` or `false.B`	literal values
`UInt(32.W)`	type 32-bit unsigned
`UInt()`	type, width inferred
`77.U` or `"hdead".U`	unsigned literals
`1.U(16.W)`	literal with forced width
`SInt()` or `SInt(64.W)`	like UInt
`-3.S` or `"h-44".S`	signed literals
`3.S(2.W)`	signed 2-bits wide *value -1*

Bits, UInt, SInt Casts: reinterpret cast except for:

`UInt` → `SInt`	Zero-extend to SInt

State Elements

Registers retain state until updated

```
val my_reg = Reg(UInt(32.W))
```

Flavors

`RegInit(7.U(32.w))`	reg with initial value 7
`RegNext(next_val)`	update each clock, no init
`RegEnable(next, enable)`	update, with enable gate

Updating: assign to latch new value on next clock:

```
my_reg := next_val
```

Read-Write Memory provide addressable memories

```
val my_mem = Mem(n:Int, out:Data)
```

`out` memory element type
`n` memory depth (elements)

Using: access elements by indexing:

```
val readVal = my_mem(addr:UInt/Int)
```

for synchronous read: assign output to `Reg`

```
mu_mem(addr:UInt/Int) := y
```

Modules

Defining: subclass `Module` with elements, code:

```
class Accum(width:Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out  = Output(UInt(width.W))
  })
  val sum = Reg(UInt())
  sum := sum + io.in
  io.out := sum
}
```

Usage: access elements using dot notation:
(code inside a `Module` is always running)

```
val my_module = Module(new Accum(32))
my_module.io.in := some_data
val sum := my_module.io.out
```

Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	`1`
`x && y`	Logical AND	`1`
`x || y`	Logical OR	`1`
`x(n)`	Extract bit, 0 is LSB	`1`
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y)`
`x >> y`	Dynamic right shift	`w(x) - minVal(y)`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate `x`, `n` times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If `c`, then `x`; else `y`	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	`1`
`x != y`	Inequality	`1`
`x =/= y`	Inequality	`1`
`x + y`	Addition	`max(w(x),w(y))`
`x +% y`	Addition	`max(w(x),w(y))`
`x +& y`	Addition	`max(w(x),w(y))+1`
`x - y`	Subtraction	`max(w(x),w(y))`
`x -% y`	Subtraction	`max(w(x),w(y))`
`x -& y`	Subtraction	`max(w(x),w(y))+1`
`x * y`	Multiplication	`w(x)+w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y)-1)`
`x > y`	Greater than	`1`
`x >= y`	Greater than or equal	`1`
`x < y`	Less than	`1`
`x <= y`	Less than or equal	`1`
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

UInt bit-reduction methods:

Chisel	Explanation	Width
`x.andR`	AND-reduce	`1`
`x.orR`	OR-reduce	`1`
`x.xorR`	XOR-reduce	`1`

As an example to apply the andR method to an SInt use x.asUInt.andR

Hardware Generation

Functions provide block abstractions for code. Scala functions that instantiate or return Chisel types are code generators.

Also: Scala's `if` and `for` can be used to control hardware generation and are equivalent to Verilog `generate if/for`

```
val number = Reg(if(can_be_negative) SInt()
                 else UInt())
```

will create a Register of type SInt or UInt depending on the value of a Scala variable

Aggregate Types

Bundle contains `Data` types indexed by name
Defining: subclass `Bundle`, define components:

```
class MyBundle extends Bundle {
  val a = Bool()
  val b = UInt(32.W)
}
```

Constructor: instantiate `Bundle` subclass:
`val my_bundle = new MyBundle()`
Inline defining: define a `Bundle` type:

```
val my_bundle = new Bundle {
  val a = Bool()
  val b = UInt(32.W)
}
```

Using: access elements through dot notation:
`val bundleVal = my_bundle.a`
`my_bundle.a := Bool(true)`

Vec is an indexable vector of `Data` types
`val myVec = Vec(elts:Iterable[Data])`
`elts` initial element `Data` (vector depth inferred)
`val myVec = Vec.fill(n:Int) {gen:Data}`
`n` vector depth (elements)
`gen` initial element `Data`, called once per element
Using: access elements by dynamic or static indexing:
`readVal := myVec(ind:Data/idx:Int)`
`myVec(ind:Data/idx:Int) := writeVal`
Functions: (`T` is the `Vec` element's type)

`.forall(p:T=>Bool): Bool`	AND-reduce `p` on all elts
`.exists(p:T=>Bool): Bool`	OR-reduce `p` on all elts
`.contains(x:T): Bool`	`True` if this contains `x`
`.count(p:T=>Bool): UInt`	count elts where `p` is `True`
`.indexWhere(p:T=>Bool): UInt`	
`.lastIndexWhere(p:T=>Bool): UInt`	
`.onlyIndexWhere(p:T=>Bool): UInt`	

Standard Library: Function Blocks

Stateless:
`PopCount(in:Bits/Seq[Bool]): UInt`
Returns number of hot (= 1) bits in `in`
`Reverse(in:UInt): UInt`
Reverses the bit order of `in`
`UIntToOH(in:UInt, [width:Int]): Bits`
Returns the one-hot encoding of `in`
`width` *(optional, else inferred)* output width
`OHToUInt(in:Bits/Seq[Bool]): UInt`
Returns the `UInt` representation of one-hot `in`
`Counter(n:Int]): UInt`
.inc() bumps counter returning true when `n` reached
.value returns current value
`PriorityEncoder(in:Bits/Iterable[Bool]): UInt`
Returns the position the least significant `1` in `in`
`PriorityEncoderOH(in:Bits): UInt`
Returns the position of the hot bit in `in`
`Mux1H(in:Iterable[(Data, Bool]): Data`
`Mux1H(sel:Bits/Iterable[Bool], in:Iterable[Data]): Data`
`PriorityMux(in:Iterable[(Bool, Bits]): Bits`
`PriorityMux(sel:Bits/Iterable[Bool], in:Iterable[Bits]): Bits`
A mux tree with either a one-hot select or multiple selects (where the first inputs are prioritized)
`in` iterable of combined input and select `(Bool, Bits)` tuples or just mux input `Bits`
`sel` select signals or bitvector, one per input

Stateful:
`LFSR16([increment:Bool]): UInt`
16-bit LFSR (to generate pseudorandom numbers)
`increment` *(optional, default True)* shift on next clock
`ShiftRegister(in:Data, n:Int, [en:Bool]): Data`
Shift register, returns `n`-cycle delayed input `in`
`en` *(optional, default True)* enable

Standard Library: Interfaces

DecoupledIO is a `Bundle` with a ready-valid interface
Constructor:
`Decoupled(gen:Data)`
`gen` Chisel `Data` to wrap ready-valid protocol around
Interface:
(in) `.ready` ready `Bool`
(out) `.valid` valid `Bool`
(out) `.bits` data

ValidIO is a `Bundle` with a valid interface
Constructor:
`Valid(gen:Data)`
`gen` Chisel `Data` to wrap valid protocol around
Interface:
(out) `.valid` valid `Bool`
(out) `.bits` data

Queue is a `Module` providing a hardware queue
Constructor:
`Queue(enq:DecoupledIO, entries:Int)`
`enq` `DecoupledIO` source for the queue
`entries` size of queue
Interface:
`.io.enq` `DecoupledIO` source (flipped)
`.io.deq` `DecoupledIO` sink
`.io.count` `UInt` count of elements in the queue

Pipe is a `Module` delaying input data
Constructor:
`Pipe(enqValid:Bool, enqBits:Data, [latency:Int])`
`Pipe(enq:ValidIO, [latency:Int])`
`enqValid` input data, valid component
`enqBits` input data, data component
`enq` input data as `ValidIO`
`latency` *(optional, default 1)* cycles to delay data by
Interface:
`.io.enq` `ValidIO` source (flipped)
`.io.deq` `ValidIO` sink

Arbiters are `Modules` connecting multiple producers to one consumer
`Arbiter` prioritizes lower producers
`RRArbiter` runs in round-robin order
Constructor:
`Arbiter(gen:Data, n:Int)`
`gen` data type
`n` number of producers
Interface:
`.io.in` `Vec` of `DecoupledIO` inputs (flipped)
`.io.out` `DecoupledIO` output
`.io.chosen` `UInt` input index on `.io.out`, does not imply output is valid

EXERCISES

2.2

Create a Chisel module that implements the multiply accumulate function, `(A*B)+C`, and passes the testbench.

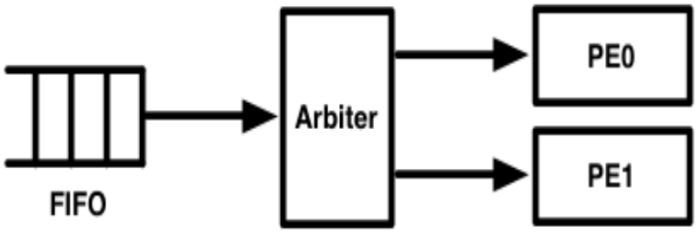
```
class MACTester(c: MAC) extends PeekPokeTester(c) {
  val cycles = 100
  import scala.util.Random
  for (i <- 0 until cycles) {
    val in_a = Random.nextInt(16)
    val in_b = Random.nextInt(16)
    val in_c = Random.nextInt(16)
    poke(c.io.in_a, in_a)
    poke(c.io.in_b, in_b)
    poke(c.io.in_c, in_c)
    expect(c.io.out, in_a*in_b+in_c)
  }
}
assert(Driver(() => new MAC) {c => new MACTester(c)})
println("SUCCESS!!")
```

```
Class MAC extends Module {
        val io = IO(new Bundle {
                val in_a = Input(UInt(4.W) )
                val in_b = Input(UInt(4.W) )
                val in_c = Input(UInt(4.W) )
                val out = output(UInt(8.W) )
        } )
        io.out := (io.in_a * io.in_b) + io.in_c
}
```

Arbiter

The following circuit arbitrates data coming from a FIFO into two parallel processing units. The FIFO and processing elements (PEs) communicate with ready-valid interfaces. Construct the arbiter to send data to whichever PE is ready to receive data, prioritizing PE0 if both are ready to receive data.

Remember that the arbiter should tell the FIFO that it's ready to receive data when at least one of the PEs can receive data. Also, wait for a PE to assert that it's ready before asserting that the data are valid. You will likely need binary operators to complete this exercise.




```
io.fifo_ready := io.pe0_ready || io.pe1_ready
  io.pe0_valid := io.pe0_ready && io.fifo_valid
  io.pe1_valid := io.pe1_ready && io.fifo_valid && !io.pe0_ready
  io.pe0_data := io.fifo_data
  io.pe1_data := io.fifo_data
```

Parameterized Adder

Construct a parameterized adder that can either saturate the output when overflow occurs, or truncate the results (i.e. wrap around).

`val sum = io.in_a +& io.in_b` Desta forma, "sum" terá tamanho de "in_a"+1 caso exceda a width

```
val sum = io.in_a +& io.in_b
if (saturate) {
 io.out := Mux(sum > 15.U, 15.U, sum)
} else {
 io.out := sum
}
```

2.3 - Shift Register

3.1 – Mealy Machine

3.2 – 32-bit RISC-V Processor (Register File)

```
class RegisterFile(readPorts: Int) extends Module {
    require(readPorts >= 0)
    val io = IO(new Bundle {
        val wen   = Input(Bool())
        val waddr = Input(UInt(5.W))
        val wdata = Input(UInt(32.W))
        val raddr = Input(Vec(readPorts, UInt(5.W)))
        val rdata = Output(Vec(readPorts, UInt(32.W)))
    })

    // A Register of a vector of UInts
   val registos = RegInit(VecInit(Seq.fill(32)(0.U(32.W))))

    //sendo chisel_Bool usar "when" em vez de "if"
    when(io.wen){registos(io.waddr) := io.wdata}
    for(i<- 0 until readPorts){
        when(io.raddr(i) === 0.U){
            io.rdata(i) := 0.U
        }
        .otherwise{
            io.rdata(i) := registos(io.raddr(i))
        }
    }


}
```

3.3 – Decoupled Arbiter

```
val io = IO(new Bundle {
  val in = Flipped(Decoupled(UInt(8.W)))
  val out = Decoupled(UInt(8.W))
})
```

```
val io = IO(new Bundle {
  val in = new Bundle {
    val valid = Input(Bool())
    val ready = Output(Bool())
    val bits  = Input(UInt(8.W))
  }
  val out = new Bundle {
    val valid = Output(Bool())
    val ready = Input(Bool())
    val bits  = Output(UInt(8.W))
  }
}
```

3.4 – Scan/Fold/Reduce

```
val exList = List(1, 5, 7, 100)

// write a custom function to add two numbers, then use reduce to find the sum of all values in exList
def add(a: Int, b: Int): Int = a + b
val sum = exList.reduce(add)

// find the sum of exList using an anonymous function (hint: you've seen this before!)
val anon_sum = exList.reduce(_ + _)

// find the moving average of exList from right to left using scan; make the result (ma2) a list of doubles
def avg(a: Int, b: Double): Double = (a + b)/2.0
val ma2 = exList.scanRight(0.0)(avg)
```

Exercise: Neural Network Neuron

In this exercise, you will implement different activation functions and pass them as an argument to your neuron generator.

The parameter `inputs` gives the number of inputs. The parameter `act` is a function that implements the logic of the activation function. We'll make the inputs and outputs 16-bit fixed point values with 8 fractional bits.

```
class Neuron(inputs: Int, act: FixedPoint => FixedPoint) extends Module {
  val io = IO(new Bundle {
    val in      = Input(Vec(inputs, FixedPoint(16.W, 8.BP)))
    val weights = Input(Vec(inputs, FixedPoint(16.W, 8.BP)))
    val out     = Output(FixedPoint(16.W, 8.BP))
  })

  val mac = io.in.zip(io.weights).map{ case(a: FixedPoint, b: FixedPoint) => a*b}.reduce(_+_)
  io.out := act(mac)
}
```

Implement these two functions below.

$$step(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \qquad relu(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

```
val Step: FixedPoint => FixedPoint = x => Mux( x <= 0.F(8.BP), 0.F(8.BP), 1.F(8.BP))
val ReLU: FixedPoint => FixedPoint = x => Mux( x <= 0.F(8.BP), 0.F(8.BP), x)
```

3.4 – EXERCICIO DO NEURON! NÃO ENTENDO O PORQUE DA DIFERENÇA DE VALORES NO INPUT WEIGHTS.

```
// test our Neuron
Driver(() => new Neuron(2, Step)) {
  c => new PeekPokeTester(c) {

    val inputs = Seq(Seq(-1, -1), Seq(-1, 1), Seq(1, -1), Seq(1, 1))

    // make this a sequence of two values
    val weights = Seq(1.0, 1.0)

    // push data through our Neuron and check the result (AND gate)
    reset(5)
    for (i <- inputs) {
      pokeFixedPoint(c.io.in(0), i(0))
      pokeFixedPoint(c.io.in(1), i(1))
      pokeFixedPoint(c.io.weights(0), weights(0))
      pokeFixedPoint(c.io.weights(1), weights(1))
      expectFixedPoint(c.io.out, if (i(0) + i(1) > 0) 1 else 0, "ERROR")
      step(1)
    }
  }
}
```

3.5 – Gray Encoder and Decorder, Gray Counter

Bibliography

Contributors

- Stevo Bailey (stevo@berkeley.edu)
- Adam Izraelevitz (adamiz@berkeley.edu)
- Richard Lin (richard.lin@berkeley.edu)
- Chick Markley (chick@berkeley.edu)
- Paul Rigge (rigge@berkeley.edu)
- Edward Wang (edwardw@berkeley.edu)