Chisel 3 Documentation

Table of contents

Release Notes

With the 3.0.0 release of Chisel3 on 11/24/17, release notes have been moved to the GitHub repository.

Home

Welcome to the Chisel 3 wiki!

If you are completely new to Chisel, check out A Short Users Guide to Chisel.

Chisel is constantly being improved. See the latest Release Notes.

For migrating from Chisel 2 to Chisel 3, check out Chisel3 vs Chisel2.

The ScalaDoc for Chisel3 is available at the API tab on the Chisel web site.

For useful design patterns, see the Cookbook.

For cool new features on the leading and bleeding edge, see Experimental Features.

If you're developing a Chisel library, see Developers.

Interested in the Future of Chisel, see Governance.

Other interesting pages:

- Frequently Asked Questions
- The new Experimental Interval Type
- Printing in Chisel
- How to use IntelliJ to run chisel3
- Tips and Tricks
- How to publish a version
- Developers' Troubleshooting Guide
- Chisel Toolchain
- Useful SBT commands
- Chisel Cheatsheet for Hardware Engineers
- Scala Things You Should Know
- Scala-land vs. Chisel-land
- CS250 a Chisel+Scala Primer
- Unconnected Wires
- Chisel Memories
- Chisel Developer Stuff
- Debugging with the Interpreter
  - Debugging with the Interpreter-REPL
  - Using the REPL to debug a problem Module
  - VCD Files and the Repl

Frequently Asked Questions

Where should I start if I want to learn Chisel?

We recommend the Chisel Bootcamp for getting started with Chisel.

How do I do ... (e.g. like that in Verilog) in Chisel?

See the cookbook.

How can I contribute to Chisel?

A good to place to start is to fill out the How Can I Contribute Form.

What is the difference between release and master branches?

We have two main branches for each main Chisel project:

- `master`
- `release`

`master` is the main development branch and it is updated frequently (often several times a day). Although we endeavour to keep the `master` branches in sync, they may drift out of sync for a day or two. We do not publish the `master` branches. If you wish to use them, you need to clone the GitHub repositories and use `sbt publishLocal` to make them available on your local machine.

The `release` branches are updated less often (currently bi-weekly) and we try to guarantee they are in sync. We publish these to Sonatype/Maven on a bi-weekly basis.

In general, you can not mix `release` and `master` branches and assume they will work.

The default branches for the user-facing repositories (chisel-template and chisel-tutorial) are the `release` branches - these should always *just work* for new users as they use the `release` branches of chisel projects.

If you want to use something more current than the `release` branch, you should `git checkout master` for all the chisel repos you intend to use, then `sbt publishLocal` them in this order:

- firrtl
- firrtl-interpreter
- chisel3
- chisel-testers

Then, if you're working with the user-facing repositories:

- chisel-tutorial
- chisel-template

Since this is a substantial amount of work (with no guarantee of success), unless you are actively involved in Chisel development, we encourage you to stick with the `release` branches and their respective dependencies.

Why DecoupledIO instead of ReadyValidIO?

There are multiple kinds of Ready/Valid interfaces that impose varying restrictions on the producers and consumers. Chisel currently provides the following:

- DecoupledIO - No guarantees
- IrrevocableIO - Producer promises to not change the value of 'bits' after a cycle where 'valid' is high and 'ready' is low. Additionally, once 'valid' is raised it will never be lowered until after 'ready' has also been raised.

Why do I have to wrap module instantiations in `Module(...)`?

In short: Limitations of Scala

Chisel Modules are written by defining a Scala class and implementing its constructor. As elaboration runs, Chisel constructs a hardware AST from these Modules. The compiler needs hooks to run before and after the actual construction of the Module object. In Scala, superclasses are fully initialized before subclasses, so by extending Module, Chisel has the ability to run some initialization code before the user's Module is constructed. However, there is no such hook to run after the Module object is initialized. By wrapping Module instantiations in the Module object's apply method (ie. `Module(...)`), Chisel is able to perform post-initialization actions. There is a proposed solution, so eventually this requirement will be lifted, but for now, wrap those Modules!

Why Chisel?

Borrowed from Chisel Introduction

> We were motivated to develop a new hardware language by years of struggle with existing hardware description languages in our research projects and hardware design courses. *Verilog* and *VHDL* were developed as hardware *simulation* languages, and only later did they become a basis for hardware *synthesis*. Much of the semantics of these languages are not appropriate for hardware synthesis and, in fact, many constructs are simply not synthesizable. Other constructs are non-intuitive in how they map to hardware implementations, or their use can accidently lead to highly inefficient hardware structures. While it is possible to use a subset of these languages and still get acceptable results, they nonetheless present a cluttered and confusing specification model, particularly in an instructional setting.

> However, our strongest motivation for developing a new hardware language is our desire to change the way that electronic system design takes place. We believe that it is important to not only teach students how to design circuits, but also to teach them how to design *circuit generators* ---programs that automatically generate designs from a high-level set of design parameters and constraints. Through circuit generators, we hope to leverage the hard work of design experts and raise the level of design abstraction for everyone. To express flexible and scalable circuit construction, circuit generators must employ sophisticated programming techniques to make decisions concerning how to best customize their output circuits according to high-level parameter values and constraints. While Verilog and VHDL include some primitive constructs for programmatic circuit generation, they lack the powerful facilities present in modern programming languages, such as object-oriented programming, type inference, support for functional programming, and reflection.

> Instead of building a new hardware design language from scratch, we chose to embed hardware construction primitives within an existing language. We picked Scala not only because it includes the programming features we feel are important for building circuit generators, but because it was specifically developed as a base for domain-specific languages.

Does Chisel support X and Z logic values

Chisel does not directly support Verilog logic values `x` *unknown* and `z` *high-impedance*. There are a number of reasons to want to avoid these values. See:The Dangers of Living With An X and Malicious LUT: A stealthy FPGA Trojan injected and triggered by the design flow. Chisel has it's own eco-system of unit and functional testers that limit the need for `x` and `z` and their omission simplify language implementation, design, and testing. The circuits created by chisel do not preclude developers from using `x` and `z` in downstream toolchains as they see fit.

Get me Verilog

I wrote a module and I want to see the Verilog; what do I do?

Here's a simple hello world module in a file HelloWorld.scala.

```
package intro
import chisel3._
class HelloWorld extends Module {
  val io = IO(new Bundle{})
  printf("hello world\n")
}
```

Add the following

```
object HelloWorld extends App {
  chisel3.Driver.execute(args, () => new HelloWorld)
  // Alternate version if there are no args
  // chisel3.Driver.execute(Array[String](), () => new HelloWorld)
}
```

Now you can get some Verilog. Start sbt:

```
bash> sbt
> run-main intro.HelloWorld
[info] Running examples.HelloWorld
[info] [0.004] Elaborating design...
[info] [0.100] Done elaborating.
[success] Total time: 1 s, completed Jan 12, 2017 6:24:03 PM
```

or as a one-liner:

```
bash> sbt 'runMain intro.HelloWorld'
```

After either of the above there will be a HelloWorld.v file in the current directory.

You can see additional options with

```
bash> sbt 'runMain intro.HelloWorld --help'
```

This will return a comprehensive usage line with available options.

For example to place the output in a directory name buildstuff use

```
bash> sbt 'runMain intro.HelloWorld --target-dir buildstuff --top-name HelloWorld'
```

Alternatively, you can also use the sbt console to invoke the Verilog driver:

```
$ sbt
> console
[info] Starting scala interpreter...
[info]
Welcome to Scala 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_121).
Type in expressions for evaluation. Or try :help.
scala> chisel3.Driver.execute(Array[String](), () => new HelloWorld)
chisel3.Driver.execute(Array[String](), () => new HelloWorld)
[info] [0.014] Elaborating design...
[info] [0.306] Done elaborating.
Total FIRRTL Compile Time: 838.8 ms
res3: chisel3.ChiselExecutionResult = [...]
```

As before, there should be a HelloWorld.v file in the current directory.

Get me FIRRTL

If for some reason you don't want the Verilog (e.g. maybe you want to run some custom transformations before exporting to Verilog), then use something along these lines (replace Multiplier with your module):

```
package intro

import chisel3._
import java.io.File

object Main extends App {
  val f = new File("Multiplier.fir")
  chisel3.Driver.dumpFirrtl(chisel3.Driver.elaborate(() => new Multiplier), Option(f))
}
```

Run it with:

```
sbt 'runMain intro.Main'
```

Alternatively, you can also use the sbt console to invoke the FIRRTL driver directly (replace HelloWorld with your module name):

```
$ sbt
> console
[info] Starting scala interpreter...
[info]
Welcome to Scala 2.11.11 (OpenJDK 64-Bit Server VM, Java 1.8.0_151).
Type in expressions for evaluation. Or try :help.
scala> chisel3.Driver.dumpFirrtl(chisel3.Driver.elaborate(() => new HelloWorld), Option(new java.io.File("output.fir")))
chisel3.Driver.dumpFirrtl(chisel3.Driver.elaborate(() => new HelloWorld), Option(new java.io.File("output.fir")))
[info] [0.000] Elaborating design...
[info] [0.001] Done elaborating.
res3: java.io.File = output.fir
```

Why doesn't Chisel tell me which wires aren't connected?

As of commit c313e13 it can! Please visit the wiki page Unconnected Wires for details.

What does `Reference ... is not fully initialized.` mean?

It means that you have unconnected wires in your design which could be an indication of a design bug.

In Chisel2 compatibility mode (`NotStrict` compile options), chisel generates firrtl code that disables firrtl's initialized wire checks. In pure chisel3 (`Strict` compile options), the generated firrtl code does not contain these disablers (`is invalid`). Output wires that are not driven (not connected) are reported by firrtl as `not fully initialized`. Please visit the wiki page Unconnected Wires for details on solving the problem.

Cookbook

Welcome to the Chisel cookbook. This cookbook is still in early stages. If you have any requests or examples to share, please file an issue and let us know!

Please note that these examples make use of Chisel's scala-style printing.

Converting Chisel Types to/from UInt

How do I create a UInt from an instance of a Bundle?

Call asUInt on the Bundle instance.

```
// Example
class MyBundle extends Bundle {
  val foo = UInt(4.W)
  val bar = UInt(4.W)
}
val bundle = Wire(new MyBundle)
bundle.foo := 0xc.U
bundle.bar := 0x3.U
val uint = bundle.asUInt
printf(p"$uint") // 195

// Test
assert(uint === 0xc3.U)
```

How do I create a Bundle from a UInt?

On an instance of the Bundle, call the method fromBits with the UInt as the argument

```
// Example
class MyBundle extends Bundle {
  val foo = UInt(4.W)
  val bar = UInt(4.W)
}
val uint = 0xb4.U
val bundle = (new MyBundle).fromBits(uint)
printf(p"$bundle") // Bundle(foo -> 11, bar -> 4)

// Test
assert(bundle.foo === 0xb.U)
assert(bundle.bar === 0x4.U)
```

How do I create a Vec of Bools from a UInt?

Use the builtin function chisel3.core.Bits.toBools to create a Scala Seq of Bool, then wrap the resulting Seq in Vec(...)

```
// Example
val uint = 0xc.U
val vec = Vec(uint.toBools)
printf(p"$vec") // Vec(0, 0, 1, 1)

// Test
assert(vec(0) === false.B)
assert(vec(1) === false.B)
assert(vec(2) === true.B)
assert(vec(3) === true.B)
```

How do I create a UInt from a Vec of Bool?

Use the builtin function asUInt

```
// Example
val vec = Vec(true.B, false.B, true.B, true.B)
val uint = vec.asUInt
printf(p"$uint") // 13

/* Test
 *
 * (remember leftmost Bool in Vec is low order bit)
 */
assert(0xd.U === uint)
```

Vectors and Registers

How do I create a Vector of Registers?

Rule! Use Reg of Vec not Vec of Reg!

You create a Reg of type Vec. Because Vecs are a *type* (like `UInt`, `Bool`) rather than a *value*, we must bind the Vec to some concrete *value*.

How do I create a Reg of type Vec?

For information, please see the API documentation (https://chisel.eecs.berkeley.edu/api/index.html#chisel3.core.Vec)

```
// Reg of Vec of 32-bit UInts without initialization
val regOfVec = Reg(Vec(4, UInt(32.W)))
regOfVec(0) := 123.U // a couple of assignments
regOfVec(2) := regOfVec(0)

// Reg of Vec of 32-bit UInts initialized to zero
//   Note that Seq.fill constructs 4 32-bit UInt literals with the value 0
//   VecInit(...) then constructs a Wire of these literals
//   The Reg is then initialized to the value of the Wire (which gives it the same type)
val initRegOfVec = RegInit(VecInit(Seq.fill(4)(0.U(32.W))))

// Simple test (cycle comes from superclass)
when (cycle === 2.U) { assert(regOfVec(2) === 123.U) }
for (elt <- initRegOfVec) { assert(elt === 0.U) }
```

How do I create a finite state machine?

Use Chisel Enum to construct the states and switch & is to construct the FSM control logic.

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

class DetectTwoOnes extends Module {
  val io = IO(new Bundle {
    val in = Input(Bool())
    val out = Output(Bool())
  })

  val sNone :: sOne1 :: sTwo1s :: Nil = Enum(3)
  val state = RegInit(sNone)

  io.out := (state === sTwo1s)

  switch (state) {
    is (sNone) {
      when (io.in) {
        state := sOne1
      }
    }
    is (sOne1) {
      when (io.in) {
        state := sTwo1s
      } .otherwise {
        state := sNone
      }
    }
    is (sTwo1s) {
      when (!io.in) {
        state := sNone
      }
    }
  }
}
```

The `is` statement can take multiple conditions e.g. `is (sTwo1s, sOne1) { ... }`.

How do I unpack a value ("reverse concatenation") like in Verilog?

```
wire [1:0] a;
wire [3:0] b;
wire [2:0] c;
wire [8:0] z = [...];
assign {a,b,c} = z;
```

Unpacking often corresponds to reinterpreting an unstructured data type as a structured data type. Frequently, this structured type is used prolifically in the design, and has been declared as in the following example:

```
class MyBundle extends Bundle {
  val a = UInt(2.W)
  val b = UInt(4.W)
  val c = UInt(3.W)
}
```

The easiest way to accomplish this in Chisel would be:

```
val z = Wire(UInt(9.W))
// z := ...
val unpacked = z.asTypeOf(new MyBundle)
unpacked.a
unpacked.b
unpacked.c
```

If you **really** need to do this for a one-off case (Think thrice! It is likely you can better structure the code using bundles), then rocket-chip has a Split utility which can accomplish this.

How do I do subword assignment (assign to some bits in a UInt)?

Example:

```
class TestModule extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(10.W))
    val bit = Input(Bool())
    val out = Output(UInt(10.W))
  })
  io.out(0) := io.bit
}
```

Chisel3 does not support subword assignment. We find that this type of thing can usually be better expressed with aggregate/structured types: Bundles and Vecs.

If you must express it this way, you can blast your UInt to a Vec of Bools and back:

```
class TestModule extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(10.W))
    val bit = Input(Bool())
    val out = Output(UInt(10.W))
  })
  val bools = VecInit(io.in.toBools)
  bools(0) := io.bit
  io.out := bools.asUInt
}
```

How can I dynamically set/parametrize the name of a module?

You can override the `desiredName` function. This works with normal Chisel modules and `BlackBox`es. Example:

```
class Coffee extends BlackBox {
    val io = IO(new Bundle {
        val I = Input(UInt(32.W))
        val O = Output(UInt(32.W))
    })
    override def desiredName = "Tea"
}
class Salt extends Module {
    val io = IO(new Bundle {})
    val drink = Module(new Coffee)
    override def desiredName = "SodiumMonochloride"
}
```

Elaborating the Chisel module `Salt` yields:

```
module SodiumMonochloride(
  input   clock,
  input   reset
);
  wire [31:0] drink_O;
  wire [31:0] drink_I;
  Tea drink (
    .O(drink_O),
    .I(drink_I)
  );
  assign drink_I = 32'h0;
endmodule
```

How do I create an optional I/O?

The following example is a module which includes the optional port `out2` only if the given parameter is `true`.

```
class ModuleWithOptionalIOs(flag: Boolean) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(12.W))
    val out = Output(UInt(12.W))
    val out2 = if (flag) Some(Output(UInt(12.W))) else None
  })

  io.out := io.in
  if (flag) {
    io.out2.get := io.in
  }
}
```

How do I get Chisel to name signals properly in blocks like when/withClockAndReset?

To get Chisel to name signals (wires and registers) declared inside of blocks like `when`, `withClockAndReset`, etc, use the `chiselName` annotation as shown below:

```
import chisel3._
import chisel3.experimental.chiselName

@chiselName
class TestMod extends Module {
  val io = IO(new Bundle {
    val a = Input(Bool())
    val b = Output(UInt(4.W))
  })
  when (io.a) {
    val innerReg = RegInit(5.U(4.W))
    innerReg := innerReg + 1.U
    io.b := innerReg
  } .otherwise {
    io.b := 10.U
  }
}
```

Note that you will need to add the following line to your project's `build.sbt` file.

```
addCompilerPlugin("org.scalamacros" % "paradise" % "2.1.0" cross CrossVersion.full)
```

Without `chiselName`, Chisel is not able to name `innerReg` correctly (notice the `_T`):

```
module TestMod(
  input        clock,
  input        reset,
  input        io_a,
  output [3:0] io_b
);
  reg [3:0] _T;
  wire [3:0] _T_2;
  assign _T_2 = _T + 4'h1;
  assign io_b = io_a ? _T : 4'ha;
  always @(posedge clock) begin
    if (reset) begin
      _T <= 4'h5;
    end else begin
      _T <= _T_2;
    end
  end
endmodule
```

In contrast, Chisel is able to name `innerReg` correctly with `chiselName`:

```
module TestMod(
  input        clock,
  input        reset,
  input        io_a,
  output [3:0] io_b
);
  reg [3:0] innerReg;
  wire [3:0] _T_1;
  assign _T_1 = innerReg + 4'h1;
  assign io_b = io_a ? innerReg : 4'ha;
  always @(posedge clock) begin
    if (reset) begin
      innerReg <= 4'h5;
    end else begin
      innerReg <= _T_1;
    end
  end
endmodule
```

Troubleshooting

This page is a starting point for recording common and not so common problems in developing with Chisel3. In particular, those situations where there is a work around that will keep you going.

`type mismatch` specifying width/value of a `UInt`/`SInt`

I have some old code that used to work correctly in chisel2 (and still does if I use the `import Chisel._` compatibility layer) but causes a `type mismatch` error in straight chisel3:

```
class TestBlock extends Module {
  val io = IO(new Bundle {
    val output = Output(UInt(width=3))
  })
}
```

produces

```
type mismatch;
[error]  found   : Int(3)
[error]  required: chisel3.internal.firrtl.Width
[error]         val output = Output(UInt(width=3))
```

The single argument, multi-function object/constructors from chisel2 have been removed from chisel3. It was felt these were too prone to error and made it difficult to diagnose error conditions in chisel3 code.

In chisel3, the single argument to the `UInt`/`SInt` object/constructor specifies the *width* and must be a `Width` type. Although there are no automatic conversions from `Int` to `Width`, an `Int` may be converted to a `Width` by applying the `W` method to an `Int`. In chisel3, the above code becomes:

```
class TestBlock extends Module {
  val io = IO(new Bundle {
    val output = Output(UInt(3.W))
  })
}
```

`UInt`/`SInt` literals may be created from an `Int` with the application of either the `U` or `S` method.

```
UInt(42)
```

in chisel2, becomes

```
42.U
```

in chisel3

A literal with a specific width is created by calling the `U` or `S` method with a `W` argument. Use:

```
1.S(8.W)
```

to create an 8-bit wide (signed) literal with value 1.

A Short Users Guide to Chisel

This is a brief guide to the Chisel hardware description language. To get started, we recommend looking at this document and the Chisel-tutorial

Introduction

This document is a tutorial introduction to *Chisel* (Constructing Hardware In a Scala Embedded Language). Chisel is a hardware construction language embedded in the high-level programming language Scala. At some point we will provide a proper reference manual, in addition to more tutorial examples. In the meantime, this document along with a lot of trial and error should set you on your way to using Chisel. *Chisel is really only a set of special class definitions, predefined objects, and usage conventions within Scala, so when you write Chisel you are actually writing a Scala program that constructs a hardware graph.* However, for the tutorial we don't presume that you understand how to program in Scala. We will point out necessary Scala features through the Chisel examples we give, and significant hardware designs can be completed using only the material contained herein. But as you gain experience and want to make your code simpler or more reusable, you will find it important to leverage the underlying power of the Scala language. We recommend you consult one of the excellent Scala books to become more expert in Scala programming.

> Through the tutorial, we format commentary on our design choices as in this paragraph. You should be able to skip the commentary sections and still fully understand how to use Chisel, but we hope you'll find them interesting.

> We were motivated to develop a new hardware language by years of struggle with existing hardware description languages in our research projects and hardware design courses. *Verilog* and *VHDL* were developed as hardware *simulation* languages, and only later did they become a basis for hardware *synthesis*. Much of the semantics of these languages are not appropriate for hardware synthesis and, in fact, many constructs are simply not synthesizable. Other constructs are non-intuitive in how they map to hardware implementations, or their use can accidently lead to highly inefficient hardware structures. While it is possible to use a subset of these languages and still get acceptable results, they nonetheless present a cluttered and confusing specification model, particularly in an instructional setting.

> However, our strongest motivation for developing a new hardware language is our desire to change the way that electronic system design takes place. We believe that it is important to not only teach students how to design circuits, but also to teach them how to design *circuit generators* ---programs that automatically generate designs from a high-level set of design parameters and constraints. Through circuit generators, we hope to leverage the hard work of design experts and raise the level of design abstraction for everyone. To express flexible and scalable circuit construction, circuit generators must employ sophisticated programming techniques to make decisions concerning how to best customize their output circuits according to high-level parameter values and constraints. While Verilog and VHDL include some primitive constructs for programmatic circuit generation, they lack the powerful facilities present in modern programming languages, such as object-oriented programming, type inference, support for functional programming, and reflection.

> Instead of building a new hardware design language from scratch, we chose to embed hardware construction primitives within an existing language. We picked Scala not only because it includes the programming features we feel are important for building circuit generators, but because it was specifically developed as a base for domain-specific languages.

Next (Hardware Expressible in Chisel)

Hardware Expressible in Chisel

While Chisel focuses on binary logic, Chisel can support analog and tri-state wires with the `Analog` type - see Datatypes in Chisel.

We focus on binary logic designs as they constitute the vast majority of designs in practice. Tri-state logic are poorly supported standard industry flows and require special/controlled hard macros in order to be done.

Prev (Chisel Introduction) Next (Datatypes in Chisel)

Datatypes in Chisel

Chisel datatypes are used to specify the type of values held in state elements or flowing on wires. While hardware designs ultimately operate on vectors of binary digits, other more abstract representations for values allow clearer specifications and help the tools generate more optimal circuits. In Chisel, a raw collection of bits is represented by the `Bits` type. Signed and unsigned integers are considered subsets of fixed-point numbers and are represented by types `SInt` and `UInt` respectively. Signed fixed-point numbers, including integers, are represented using two's-complement format. Boolean values are represented as type `Bool`. Note that these types are distinct from Scala's builtin types such as `Int` or `Boolean`.

> There is a new experimental type **Interval** which gives the developer more control of the type by allowing the definition of an IntervalRange. See: Experimental - Interval Type

Additionally, Chisel defines `Bundles` for making collections of values with named fields (similar to `structs` in other languages), and `Vecs` for indexable collections of values.

Bundles and Vecs will be covered later.

Constant or literal values are expressed using Scala integers or strings passed to constructors for the types:

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

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

8.U(4.W) // 4-bit unsigned decimal, value 8.
-152.S(32.W) // 32-bit signed decimal, value -152.

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

Underscores can be used as separators in long string literals to aid readability, but are ignored when creating the value, e.g.:

```
"h_dead_beef".U   // 32-bit lit of type UInt
```

By default, the Chisel compiler will size each constant to the minimum number of bits required to hold the constant, including a sign bit for signed types. Bit widths can also be specified explicitly on literals, as shown below. Note that (`.W` is used to cast a Scala Int to a Chisel width)

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

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

For literals of type `UInt`, the value is zero-extended to the desired bit width. For literals of type `SInt`, the value is sign-extended to fill the desired bit width. If the given bit width is too small to hold the argument value, then a Chisel error is generated.

> We are working on a more concise literal syntax for Chisel using symbolic prefix operators, but are stymied by the limitations of Scala operator overloading and have not yet settled on a syntax that is actually more readable than constructors taking strings.

> We have also considered allowing Scala literals to be automatically converted to Chisel types, but this can cause type ambiguity and requires an additional import.

> The SInt and UInt types will also later support an optional exponent field to allow Chisel to automatically produce optimized fixed-point arithmetic circuits.

Casting

We can also cast types in Chisel:

```
val sint = 3.S(4.W)             // 4-bit SInt

val uint = sint.asUInt          // cast SInt to UInt
uint.asSInt                     // cast UInt to SInt
```

NOTE: `asUInt`/`asSInt` with an explicit width can **not** be used to cast (convert) between Chisel datatypes. No width parameter is accepted, as Chisel will automatically pad or truncate as required when the objects are connected.

We can also perform casts on clocks, though you should be careful about this, since clocking (especially in ASIC) requires special attention:

```
val bool: Bool = false.B        // always-low wire
val clock = bool.asClock        // always-low clock

clock.asUInt                    // convert clock to UInt (width 1)
clock.asUInt.asBool             // convert clock to Bool (Chisel 3.2+)
clock.asUInt.toBool             // convert clock to Bool (Chisel 3.0 and 3.1 only)
```

Analog/BlackBox type

(Experimental, Chisel 3.1+)

Chisel supports an `Analog` type (equivalent to Verilog `inout`) that can be used to support arbitrary nets in Chisel. This includes analog wires, tri-state/bi-directional wires, and power nets (with appropriate annotations).

`Analog` is an undirectioned type, and so it is possible to connect multiple `Analog` nets together using the `attach` operator. It is possible to connect an `Analog` **once** using `<>` but illegal to do it more than once.

```
val a = IO(Analog(1.W))
val b = IO(Analog(1.W))
val c = IO(Analog(1.W))

// Legal
attach(a, b)
attach(a, c)

// Legal
a <> b

// Illegal - connects 'a' multiple times
a <> b
a <> c
```

Prev (Hardware Expressible in Chisel) Next (Combinational Circuits)

Combinational Circuits

A circuit is represented as a graph of nodes in Chisel. Each node is a hardware operator that has zero or more inputs and that drives one output. A literal, introduced above, is a degenerate kind of node that has no inputs and drives a constant value on its output. One way to create and wire together nodes is using textual expressions. For example, we can express a simple combinational logic circuit using the following expression:

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

The syntax should look familiar, with `&` and `|` representing bitwise-AND and -OR respectively, and `~` representing bitwise-NOT. The names `a` through `d` represent named wires of some (unspecified) width.

Any simple expression can be converted directly into a circuit tree, with named wires at the leaves and operators forming the internal nodes. The final circuit output of the expression is taken from the operator at the root of the tree, in this example, the bitwise-OR.

Simple expressions can build circuits in the shape of trees, but to construct circuits in the shape of arbitrary directed acyclic graphs (DAGs), we need to describe fan-out. In Chisel, we do this by naming a wire that holds a subexpression that we can then reference multiple times in subsequent expressions. We name a wire in Chisel by declaring a variable. For example, consider the select expression, which is used twice in the following multiplexer description:

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

The keyword `val` is part of Scala, and is used to name variables that have values that won't change. It is used here to name the Chisel wire, `sel`, holding the output of the first bitwise-OR operator so that the output can be used multiple times in the second expression.

Wires

Chisel also supports wires as hardware nodes to which one can assign values or connect other nodes.

```
val myNode = Wire(UInt(8.W))
when (isReady) {
  myNode := 255.U
} .otherwise {
  myNode := 0.U
}
```

```
val myNode = Wire(UInt(8.W))
when (input > 128.U) {
  myNode := 255.U
} .elsewhen (input > 64.U) {
  myNode := 1.U
} .otherwise {
  myNode := 0.U
}
```

Note that the last connection to a Wire takes effect. For example, the following two Chisel circuits are equivalent:

```
val myNode = Wire(UInt(8.W))
myNode := 10.U
myNode := 0.U
```

```
val myNode = Wire(UInt(8.W))
myNode := 0.U
```

Prev (Datatypes in Chisel) Next (Builtin Operators)

Builtin Operators

List of operators

Chisel defines a set of hardware operators:

Operation	Explanation		
Bitwise operators	**Valid on:** SInt, UInt, Bool		
`val invertedX = ~x`	Bitwise NOT		
`val hiBits = x & "h_ffff_0000".U`	Bitwise AND		
`val flagsOut = flagsIn \	overflow`	Bitwise OR	
`val flagsOut = flagsIn ^ toggle`	Bitwise XOR		
Bitwise reductions.	**Valid on:** SInt and UInt. Returns Bool.		
`val allSet = x.andR`	AND reduction		
`val anySet = x.orR`	OR reduction		
`val parity = x.xorR`	XOR reduction		
Equality comparison.	**Valid on:** SInt, UInt, and Bool. Returns Bool.		
`val equ = x === y`	Equality		
`val neq = x =/= y`	Inequality		
Shifts	**Valid on:** SInt and UInt		
`val twoToTheX = 1.S << x`	Logical shift left		
`val hiBits = x >> 16.U`	Right shift (logical on UInt and arithmetic on SInt).		
Bitfield manipulation	**Valid on:** SInt, UInt, and Bool.		
`val xLSB = x(0)`	Extract single bit, LSB has index 0.		
`val xTopNibble = x(15, 12)`	Extract bit field from end to start bit position.		
`val usDebt = Fill(3, "hA".U)`	Replicate a bit string multiple times.		
`val float = Cat(sign, exponent, mantissa)`	Concatenates bit fields, with first argument on left.		
Logical Operations	**Valid on:** Bool		
`val sleep = !busy`	Logical NOT		
`val hit = tagMatch && valid`	Logical AND		
`val stall = src1busy \	\	src2busy`	Logical OR
`val out = Mux(sel, inTrue, inFalse)`	Two-input mux where sel is a Bool		
Arithmetic operations	**Valid on Nums:** SInt and UInt.		
`val sum = a + b` *or* `val sum = a +% b`	Addition (without width expansion)		
`val sum = a +& b`	Addition (with width expansion)		
`val diff = a - b` *or* `val diff = a -% b`	Subtraction (without width expansion)		
`val diff = a -& b`	Subtraction (with width expansion)		
`val prod = a * b`	Multiplication		
`val div = a / b`	Division		

Operation	Explanation
`val mod = a % b`	Modulus
Arithmetic comparisons	**Valid on Nums:** SInt and UInt. Returns Bool.
`val gt = a > b`	Greater than
`val gte = a >= b`	Greater than or equal
`val lt = a < b`	Less than
`val lte = a <= b`	Less than or equal

> Our choice of operator names was constrained by the Scala language. We have to use triple equals `===` for equality and `=/=` for inequality to allow the native Scala equals operator to remain usable.

Prev (Combinational Circuits) Next (Width Inference)

Width Inference

Chisel provides bit width inference to reduce design effort. Users are encouraged to manually specify widths of ports and registers to prevent any surprises, but otherwise unspecified widths will be inferred by the Firrtl compiler.

For all circuit components declared with unspecified widths, the FIRRTL compiler will infer the minimum possible width that maintains the legality of all its incoming connections. Implicit here is that inference is done in a right to left fashion in the sense of an assignment statement in chisel, i.e. from the left hand side from the right hand side. If a component has no incoming connections, and the width is unspecified, then an error is thrown to indicate that the width could not be inferred.

For module input ports with unspecified widths, the inferred width is the minimum possible width that maintains the legality of all incoming connections to all instantiations of the module. The width of a ground-typed multiplexor expression is the maximum of its two corresponding input widths. For multiplexing aggregate-typed expressions, the resulting widths of each leaf subelement is the maximum of its corresponding two input leaf subelement widths. The width of a conditionally valid expression is the width of its input expression. For the full formal description see the Firrtl Spec.

Hardware operators have output widths as defined by the following set of rules:

operation	bit width
`z = x + y` *or* `z = x +% y`	`w(z) = max(w(x), w(y))`
`z = x +& y`	`w(z) = max(w(x), w(y)) + 1`
`z = x - y` *or* `z = x -% y`	`w(z) = max(w(x), w(y))`
`z = x -& y`	`w(z) = max(w(x), w(y)) + 1`
`z = x & y`	`w(z) = max(w(x), w(y))`
`z = Mux(c, x, y)`	`w(z) = max(w(x), w(y))`
`z = w * y`	`w(z) = w(x) + w(y)`
`z = x << n`	`w(z) = w(x) + maxNum(n)`
`z = x >> n`	`w(z) = w(x) - minNum(n)`
`z = Cat(x, y)`	`w(z) = w(x) + w(y)`
`z = Fill(n, x)`	`w(z) = w(x) * maxNum(n)`

> where for instance `w(z)` is the bit width of wire `z`, and the `&` rule applies to all bitwise logical operations.

Given a path of connections that begins with an unspecified width element (most commonly a top-level input), then the compiler will throw an exception indicating a certain width was uninferrable.

A common "gotcha" comes from truncating addition and subtraction with the operators `+` and `-`. Users who want the result to maintain the full, expanded precision of the addition or subtraction should use the expanding operators `+&` and `-&`.

> The default truncating operation comes from Chisel's history as a microprocessor design language.

Prev (Builtin Operators) Next (Functional Abstraction)

Functional Abstraction

We can define functions to factor out a repeated piece of logic that we later reuse multiple times in a design. For example, we can wrap up our earlier example of a simple combinational logic block as follows:

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

where `clb` is the function which takes `a`, `b`, `c`, `d` as arguments and returns a wire to the output of a boolean circuit. The `def` keyword is part of Scala and introduces a function definition, with each argument followed by a colon then its type, and the function return type given after the colon following the argument list. The equals (`=})`sign separates the function argument list from the function definition.

We can then use the block in another circuit as follows:

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

We will later describe many powerful ways to use functions to construct hardware using Scala's functional programming support.

Prev (Width Inference) Next (Bundles and Vecs)

Bundles and Vecs

`Bundle` and `Vec` are classes that allow the user to expand the set of Chisel datatypes with aggregates of other types.

Bundles group together several named fields of potentially different types into a coherent unit, much like a `struct` in C. Users define their own bundles by defining a class as a subclass of `Bundle`.

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

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

> Currently, there is no way to create a bundle literal like `8.U` for `UInt`s. Therefore, in order to create literals for bundles, we must declare a wire of that bundle type, and then assign values to it. We are working on a way to declare bundle literals without requiring the creation of a Wire node and assigning to it.

```
// Floating point constant.
val floatConst = Wire(new MyFloat)
floatConst.sign := true.B
floatConst.exponent := 10.U
floatConst.significand := 128.U
```

A Scala convention is to capitalize the name of new classes and we suggest you follow that convention in Chisel too.

Vecs create an indexable vector of elements, and are constructed as follows:

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

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

(Note that we specify the number followed by the type of the `Vec` elements. We also specifiy the width of the `SInt`)

The set of primitive classes (`SInt`, `UInt`, and `Bool`) plus the aggregate classes (`Bundles` and `Vec`s) all inherit from a common superclass, `Data`. Every object that ultimately inherits from `Data` can be represented as a bit vector in a hardware design.

Bundles and Vecs can be arbitrarily nested to build complex data structures:

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

Note that the builtin Chisel primitive and aggregate classes do not require the `new` when creating an instance, whereas new user datatypes will. A Scala `apply` constructor can be defined so that a user datatype also does not require `new`, as described in Function Constructor.

Flipping Bundles

The `Flipped()` function recursively flips all elements in a Bundle/Record. This is very useful for building bidirectional interfaces that connect to each other (e.g. `Decoupled`). See below for an example.

```
import chisel3.experimental.RawModule
class MyBundle extends Bundle {
  val a = Input(Bool())
  val b = Output(Bool())
}
class MyModule extends RawModule {
  // Normal instantiation of the bundle
  // 'a' is an Input and 'b' is an Output
  val normalBundle = IO(new MyBundle)
  normalBundle.b := normalBundle.a

  // Flipped recursively flips the direction of all Bundle fields
  // Now 'a' is an Output and 'b' is an Input
  val flippedBundle = IO(Flipped(new MyBundle))
  flippedBundle.a := flippedBundle.b
}
```

This generates the following Verilog:

```
module MyModule( // @[:@3.2]
  input   normalBundle_a, // @[:@4.4]
  output  normalBundle_b, // @[:@4.4]
  output  flippedBundle_a, // @[:@5.4]
  input   flippedBundle_b // @[:@5.4]
);
  assign normalBundle_b = normalBundle_a;
  assign flippedBundle_a = flippedBundle_b;
endmodule
```

MixedVec

(Chisel 3.2+)

All elements of a `Vec` must be of the same type. If we want to create a Vec where the elements have different types, we can use a MixedVec:

```
class MyModule extends Module {
  val io = IO(new Bundle {
    val x = Input(UInt(3.W))
    val y = Input(UInt(10.W))
    val vec = Output(MixedVec(UInt(3.W), UInt(10.W)))
  })
  io.vec(0) := io.x
  io.vec(1) := io.y
}
```

We can also programmatically create the types in a MixedVec:

```
class MyModule(x: Int, y: Int) extends Module {
  val io = IO(new Bundle {
    val vec = Input(MixedVec((x to y) map { i => UInt(i.W) }))
    // ...
  })
  // ...rest of the module goes here...
}
```

A note on `cloneType`

Since Chisel is built on top of Scala and the JVM, it needs to know how to construct copies of bundles for various purposes (creating wires, IOs, etc). If you have a parametrized bundle and Chisel can't automatically figure out how to clone your bundle, you will need to create a custom `cloneType` method in your bundle. Most of the time, this is as simple as `override def cloneType = (new YourBundleHere(...)).asInstanceOf[this.type]`.

Note that in the vast majority of cases, **this is not required** as Chisel can figure out how to clone most bundles automatically.

Here is an example of a parametrized bundle (`ExampleBundle`) that features a custom `cloneType`.

```
class ExampleBundle(a: Int, b: Int) extends Bundle {
    val foo = UInt(a.W)
    val bar = UInt(b.W)
    override def cloneType = (new ExampleBundle(a, b)).asInstanceOf[this.type]
}

class ExampleBundleModule(btype: ExampleBundle) extends Module {
    val io = IO(new Bundle {
        val out = Output(UInt(32.W))
        val b = Input(chiselTypeOf(btype))
    })
    io.out := io.b.foo + io.b.bar
}

class Top extends Module {
    val io = IO(new Bundle {
        val out = Output(UInt(32.W))
        val in = Input(UInt(17.W))
    })
    val x = Wire(new ExampleBundle(31, 17))
    x := DontCare
    val m = Module(new ExampleBundleModule(x))
    m.io.b.foo := io.in
    m.io.b.bar := io.in
    io.out := m.io.out
}
```

Generally cloneType can be automatically defined if all arguments to the Bundle are vals e.g.

```
class MyBundle(val width: Int) extends Bundle {
   val field = UInt(width.W)
   // ...
}
```

The only caveat is if you are passing something of type Data as a "generator" parameter, in which case you should make it a `private val`.

For example, consider the following Bundle:

```
class RegisterWriteIO[T <: Data](gen: T) extends Bundle {
  val request  = Flipped(Decoupled(gen))
  val response = Irrevocable(Bool()) // ignore .bits

  override def cloneType = new RegisterWriteIO(gen).asInstanceOf[this.type]
}
```

We can make this this infer cloneType by making `gen` private since it is a "type parameter":

```
class RegisterWriteIO[T <: Data](private val gen: T) extends Bundle {
  val request  = Flipped(Decoupled(gen))
  val response = Irrevocable(Bool()) // ignore .bits
}
```

Prev (Functional Abstraction) Next (Ports)

Ports

Ports are used as interfaces to hardware components. A port is simply any `Data` object that has directions assigned to its members.

Chisel provides port constructors to allow a direction to be added (input or output) to an object at construction time. Primitive port constructors wrap the type of the port in `Input` or `Output`.

An example port declaration is as follows:

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

After defining `Decoupled`, it becomes a new type that can be used as needed for module interfaces or for named collections of wires.

By folding directions into the object declarations, Chisel is able to provide powerful wiring constructs described later.

Inspecting Module ports

(Chisel 3.2+)

Chisel 3.2+ introduces an API `DataMirror.modulePorts` which can be used to inspect the IOs of any Chisel module, including MultiIOModules, RawModules, and BlackBoxes.

Here is an example of how to use this API:

```
import chisel3.experimental.DataMirror

class Adder extends MultiIOModule {
  val a = IO(Input(UInt(8.W)))
  val b = IO(Input(UInt(8.W)))
  val c = IO(Output(UInt(8.W)))
  c := a +& b
}

class Test extends MultiIOModule {
  val adder = Module(new Adder)
  // for debug only
  adder.a := DontCare
  adder.b := DontCare

  // Inspect ports of adder
  // Prints something like this
  /**
    * Found port clock: Clock(IO clock in Adder)
    * Found port reset: Bool(IO reset in Adder)
    * Found port a: UInt<8>(IO a in Adder)
    * Found port b: UInt<8>(IO b in Adder)
    * Found port c: UInt<8>(IO c in Adder)
    */
  DataMirror.modulePorts(adder).foreach { case (name, port) => {
    println(s"Found port $name: $port")
  }}
}

chisel3.Driver.execute(Array[String](), () => new Test)
```

Prev (Bundles and Vecs) Next (Modules)

Modules

Chisel *modules* are very similar to Verilog *modules* in defining a hierarchical structure in the generated circuit.

The hierarchical module namespace is accessible in downstream tools to aid in debugging and physical layout. A user-defined module is defined as a *class* which:

- inherits from `Module`,
- contains an interface wrapped in a Module's `IO()` method and stored in a port field named `io`, and
- wires together subcircuits in its constructor.

As an example, consider defining your own two-input multiplexer as a module:

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

The wiring interface to a module is a collection of ports in the form of a `Bundle`. The interface to the module is defined through a field named `io`. For `Mux2`, `io` is defined as a bundle with four fields, one for each multiplexer port.

The `:=` assignment operator, used here in the body of the definition, is a special operator in Chisel that wires the input of left-hand side to the output of the right-hand side.

Module Hierarchy

We can now construct circuit hierarchies, where we build larger modules out of smaller sub-modules. For example, we can build a 4-input multiplexer module in terms of the `Mux2` module by wiring together three 2-input multiplexers:

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

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

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

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

We again define the module interface as `io` and wire up the inputs and outputs. In this case, we create three `Mux2` children modules, using the `Module` constructor function and the Scala `new` keyword to create a new object. We then wire them up to one another and to the ports of the `Mux4` interface.

Prev(Ports) Next(BlackBoxes)

BlackBoxes

Chisel provides two constructs, `ExtModule` and `BlackBox` for implementing externally defined modules. This construct is useful for hardware constructs that cannot be described in Chisel and for connecting to FPGA or other IP not defined in Chisel.

Modules defined as a `ExtModule` or `BlackBox` will be instantiated in the generated Verilog, but no code will be generated to define the behavior of module.

The preferred `ExtModule` works much like a `MultiIOMOdule`. Port naming is entirely under the developers control, there is no required IO Bundle, but one can be used manually if desired. Clock and Reset Ports must be explicitly declared.

Ports declared in the required `val io = IO()` bundle will be generated with the requested name (ie. no preceding `io_`). This can be confusing at times. Like `ExtMOdule`, `BlackBox` has no implicit clock and reset.

In general, in the following discussion. BlackBox and ExtModule behave almost identically except for the port naming.

Parameterization

This is an experimental feature and is subject to API change

Verilog parameters can be passed as an argument to the BlackBox constructor.

For example, consider instantiating a Xilinx differential clock buffer (IBUFDS) in a Chisel design:

```
import chisel3._
import chisel3.util._
import chisel3.experimental._ // To enable experimental features

class IBUFDS extends BlackBox(Map("DIFF_TERM" -> "TRUE",
                                  "IOSTANDARD" -> "DEFAULT")) {
  val io = IO(new Bundle {
    val O = Output(Clock())
    val I = Input(Clock())
    val IB = Input(Clock())
  })
}
```

In the Chisel-generated Verilog code, `IBUFDS` will be instantiated as:

```
IBUFDS #(.DIFF_TERM("TRUE"), .IOSTANDARD("DEFAULT")) ibufds (
  .IB(ibufds_IB),
  .I(ibufds_I),
  .O(ibufds_O)
);
```

Providing Implementations for Blackboxes

The Chisel Execution Harness (see Running Stuff) provides the following ways of delivering the code underlying the blackbox. Consider the following blackbox that adds two real numbers together. The numbers are represented in chisel3 as 64-bit unsigned integers.

```
class BlackBoxRealAdd extends BlackBox {
  val io = IO(new Bundle() {
    val in1 = Input(UInt(64.W))
    val in2 = Input(UInt(64.W))
    val out = Output(UInt(64.W))
  })
}
```

The implementation is described by the following verilog

```
module BlackBoxRealAdd(
    input  [63:0] in1,
    input  [63:0] in2,
    output reg [63:0] out
);
  always @* begin
  out <= $realtobits($bitstoreal(in1) + $bitstoreal(in2));
  end
endmodule
```

Blackboxes with Verilog in a Resource File

In order to deliver the verilog snippet above to the backend simulator, chisel3 provides the following tools based on the chisel/firrtl annotation system. Add the trait `HasBlackBoxResource` to the declaration, and then call a function in the body to say where the system can find the verilog. The Module now looks like

```
class BlackBoxRealAdd extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle() {
    val in1 = Input(UInt(64.W))
    val in2 = Input(UInt(64.W))
    val out = Output(UInt(64.W))
  })
  setResource("/real_math.v")
}
```

The verilog snippet above gets put into a resource file names `real_math.v`. What is a resource file? It comes from a java convention of keeping files in a project that are automatically included in library distributions. In a typical chisel3 project, see chisel-template, this would be a directory in the source hierarchy

```
src/main/resources/real_math.v
```

Blackboxes with In-line Verilog

It is also possible to place this verilog directly in the scala source. Instead of `HasBlackBoxResource` use `HasBlackBoxInline` and instead of `setResource` use `setInline`. The code will look like this.

```
class BlackBoxRealAdd extends BlackBox with HasBlackBoxInline {
  val io = IO(new Bundle() {
    val in1 = Input(UInt(64.W))
    val in2 = Input(UInt(64.W))
    val out = Output(UInt(64.W))
  })
  setInline("BlackBoxRealAdd.v",
    s"""
      |module BlackBoxRealAdd(
      |    input  [15:0] in1,
      |    input  [15:0] in2,
      |    output [15:0] out
      |);
      |always @* begin
      |  out <= $realtobits($bitstoreal(in1) + $bitstoreal(in2));
      |end
      |endmodule
    """.stripMargin)
}
```

This technique will copy the inline verilog into the target directory under the name `BlackBoxRealAdd.v`

Under the Hood

This mechanism of delivering verilog content to the testing backends is implemented via chisel/firrtl annotations. The two methods, inline and resource, are two kinds of annotations that are created via the `setInline` and `setResource` methods calls. Those annotations are passed through to the chisel-testers which in turn passes them on to firrtl. The default firrtl verilog compilers have a pass that detects the annotations and moves the files or inline test into the build directory. For each unique file added, the transform adds a line to a file black_box_verilog_files.f, this file is added to the command line constructed for verilator or vcs to inform them where to look. The dsptools project is a good example of using this feature to build a real number simulation tester based on black boxes.

Treadle and the firrtl-interpreter (Scala based simulators)

Both Treadle and the firrtl interpreter use a separate system that allows users to construct scala implementations of the black boxes. The Scala implementation code built into a BlackBoxFactory which is passed down to the simulators by the execution harness. Once again the dsptools project uses this mechanism and is a good place to look at it.

> It is planned that the BlackBoxFactory will be replaced by integration with the annotation based blackbox methods stuff soon.

Prev(Modules) Next(State Elements)

State Elements

The simplest form of state element supported by Chisel is a positive edge-triggered register, which can be instantiated as:

```
val reg = RegNext(in)
```

This circuit has an output that is a copy of the input signal `in` delayed by one clock cycle. Note that we do not have to specify the type of Reg as it will be automatically inferred from its input when instantiated in this way. In the current version of Chisel, clock and reset are global signals that are implicitly included where needed.

Note that registers which do not specify an initial value will not change value upon toggling the reset signal.

Using registers, we can quickly define a number of useful circuit constructs. For example, a rising-edge detector that takes a boolean signal in and outputs true when the current value is true and the previous value is false is given by:

```
def risingedge(x: Bool) = x && !RegNext(x)
```

Counters are an important sequential circuit. To construct an up-counter that counts up to a maximum value, max, then wraps around back to zero (i.e., modulo max+1), we write:

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

The counter register is created in the counter function with a reset value of 0 (with width large enough to hold max), to which the register will be initialized when the global reset for the circuit is asserted. The := assignment to x in counter wires an update combinational circuit which increments the counter value unless it hits the max at which point it wraps back to zero. Note that when x appears on the right-hand side of an assignment, its output is referenced, whereas when on the left-hand side, its input is referenced. Counters can be used to build a number of useful sequential circuits. For example, we can build a pulse generator by outputting true when a counter reaches zero:

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

A square-wave generator can then be toggled by the pulse train, toggling between true and false on each pulse:

```
// Flip internal state when input true.
def toggle(p: Bool) = {
  val x = RegInit(false.B)
  x := Mux(p, !x, x)
  x
}
// Square wave of a given period.
def squareWave(period: UInt) = toggle(pulse(period/2))
```

Prev(BlackBoxes) Next (Memories)

Memories

Chisel provides facilities for creating both read only and read/write memories.

ROM

Users can define read only memories with a `Vec`:

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

where `inits` is a sequence of initial `Data` literals that initialize the ROM. For example, users cancreate a small ROM initialized to 1, 2, 4, 8 and loop through all values using a counter as an address generator as follows:

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

We can create an *n* value sine lookup table using a ROM initialized as follows:

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

where `amp` is used to scale the fixpoint values stored in the ROM.

Mem

Memories are given special treatment in Chisel since hardware implementations of memory vary greatly. For example, FPGA memories are instantiated quite differently from ASIC memories. Chisel defines a memory abstraction that can map to either simple Verilog behavioural descriptions or to instances of memory modules that are available from external memory generators provided by foundry or IP vendors.

Chisel supports random-access memories via the `Mem` construct. Writes to `Mem`s are **combinational/asynchronous-read, sequential/synchronous-write**. These `Mem`s will likely be synthesized to register banks, since most SRAMs in modern technologies (FPGA, ASIC) tend to no longer support combinational (asynchronous) reads.

Chisel also has a construct called `SyncReadMem` for **sequential/synchronous-read, sequential/synchronous-write** memories. These `SyncReadMem`s will likely be synthesized to technology SRAMs (as opposed to register banks).

Ports into Mems are created by applying a `UInt` index. A 1024-entry register file with one write port and one sequential/synchronous read port might be expressed as follows:

```
val width:Int = 32
val addr = Wire(UInt(width.W))
val dataIn = Wire(UInt(width.W))
val dataOut = Wire(UInt(width.W))
val enable = Wire(Bool())

// assign data...

// Create a synchronous-read, synchronous-write memory (like in FPGAs).
val mem = SyncReadMem(1024, UInt(width.W))
// Create one write port and one read port.
mem.write(addr, dataIn)
dataOut := mem.read(addr, enable)
```

Creating an asynchronous-read version of the above simply involves replacing `SyncReadMem` with just `Mem`.

Chisel can also infer other features such as single ports and masks directly with Mem.

Single-ported SRAMs can be inferred when the same port is both read-from and written to:

```
val width:Int = 32
val enable = Input(Bool())
val write = Input(Bool())
val addr  = Input(UInt(10.W))
val dataIn = Input(UInt(width.W))
val dataOut = Output(UInt(width.W))


val mem = SyncReadMem(2048, UInt(32.W))

dataOut := DontCare
when(enable) {
  val rdwrPort = mem(addr)
  when (write) { rdwrPort := dataIn }
  .otherwise { dataOut := rdwrPort }
}
```

(The `DontCare` is there to make Chisel's unconnected wire detection aware that reading while writing is undefined.)

If the same `Mem` address is both written and sequentially read on the same clock edge, or if a sequential read enable is cleared, then the read data is undefined.

Masks

Chisel memories also support write masks for subword writes. Chisel will infer masks if the data type of the memory is a vector. To infer a mask, specify the `mask` argument of the `write` function which creates write ports. A given masked length is written if the corresponding mask bit is set. For example, in the example below, if the 0th bit of mask is true, it will write the lower 8 bits of the corresponding address.

```
val dataOut = Wire(Vec(4, UInt(8.W)))
val dataIn = Wire(Vec(4, UInt(8.W)))
val mask = Wire(Vec(4, Bool()))
val enable = Wire(Bool())
val readAddr = Wire(UInt(10.W))
val writeAddr = Wire(UInt(10.W))

// ... assign values ...

// Create a 32-bit wide memory that is byte-masked.
val mem = SyncReadMem(1024, Vec(4, UInt(8.W)))
// Create one masked write port and one read port.
mem.write(writeAddr, dataIn, mask)
dataOut := mem.read(readAddr, enable)
```

Prev(State Elements) Next(Interfaces \& Bulk Connections)

Interfaces & Bulk Connections

Interfaces & Bulk Connections

For more sophisticated modules it is often useful to define and instantiate interface classes while defining the IO for a module. First and foremost, interface classes promote reuse allowing users to capture once and for all common interfaces in a useful form.

Secondly, interfaces allow users to dramatically reduce wiring by supporting bulk connections between producer and consumer modules. Finally, users can make changes in large interfaces in one place reducing the number of updates required when adding or removing pieces of the interface.

Note that Chisel has some built-in standard interface which should be used whenever possible for interoperability (e.g. Decoupled).

Ports: Subclasses & Nesting

As we saw earlier, users can define their own interfaces by defining a class that subclasses Bundle. For example, a user could define a simple link for hand-shaking data as follows:

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

We can then extend SimpleLink by adding parity bits using bundle inheritance:

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

In general, users can organize their interfaces into hierarchies using inheritance.

From there we can define a filter interface by nesting two PLinks into a new FilterIO bundle:

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

where flip recursively changes the direction of a bundle, changing input to output and output to input.

We can now define a filter by defining a filter class extending module:

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

where the io field contains FilterIO.

Bundle Vectors

Beyond single elements, vectors of elements form richer hierarchical interfaces. For example, in order to create a crossbar with a vector of inputs, producing a vector of outputs, and selected by a UInt input, we utilize the Vec constructor:

```
import chisel3.util.log2Ceil
class CrossbarIo(n: Int) extends Bundle {
  val in = Vec(n, Flipped(new PLink))
  val sel = Input(UInt(log2Ceil(n).W))
  val out = Vec(n, new PLink)
}
```

where Vec takes a size as the first argument and a block returning a port as the second argument.

Bulk Connections

We can now compose two filters into a filter block as follows:

```
class Block extends Module {
  val io = IO(new FilterIO)
  val f1 = Module(new Filter)
  val f2 = Module(new Filter)
  f1.io.x <> io.x
  f1.io.y <> f2.io.x
  f2.io.y <> io.y
}
```

where <> bulk connects interfaces of opposite gender between sibling modules or interfaces of the same gender between parent/child modules.

Bulk connections connect leaf ports of the same name to each other. If the names do not match or are missing, Chisel does not generate a connection.

Caution: bulk connections should only be used with **directioned elements** (like IOs), and is not magical (e.g. connecting two wires isn't supported since Chisel can't necessarily figure out the directions automatically chisel3#603).

The standard ready-valid interface (ReadyValidIO / Decoupled)

Chisel provides a standard interface for ready-valid interfaces.

Usually, we use the utility function `Decoupled()` to turn any type into a ready-valid interface rather than directly using ReadyValidIO.

- `Decoupled(...)` creates a producer / output ready-valid interface (i.e. bits is an output).
- `Flipped(Decoupled(...))` creates a consumer / input ready-valid interface (i.e. bits is an input).

Take a look at the following example Chisel code to better understand exactly what is generated:

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

/**
  * Using Decoupled(...) creates a producer interface.
  * i.e. it has bits as an output.
  * This produces the following ports:
  *   input         io_readyValid_ready,
  *   output        io_readyValid_valid,
  *   output [31:0] io_readyValid_bits
  */
class ProducingData extends Module {
  val io = IO(new Bundle {
    val readyValid = Decoupled(UInt(32.W))
  })
  // do something with io.readyValid.ready
  io.readyValid.valid := true.B
  io.readyValid.bits := 5.U
}

/**
  * Using Flipped(Decoupled(...)) creates a consumer interface.
  * i.e. it has bits as an input.
  * This produces the following ports:
  *   output        io_readyValid_ready,
  *   input         io_readyValid_valid,
  *   input  [31:0] io_readyValid_bits
  */
class ConsumingData extends Module {
  val io = IO(new Bundle {
    val readyValid = Flipped(Decoupled(UInt(32.W)))
  })
  io.readyValid.ready := false.B
  // do something with io.readyValid.valid
  // do something with io.readyValid.bits
}
```

Prev(Memories) Next(Functional Module Creation)

Functional Module Creation

Objects in Scala have a pre-existing creation function (method) called `apply`. When an object is used as value in an expression (which basically means that the constructor was called), this method determines the returned value. When dealing with hardware modules, one would expect the module output to be representative of the hardware module's functionality. Therefore, we would sometimes like the module output to be the value returned when using the object as a value in an expression. Since hardware modules are represented as Scala objects, this can be done by defining the object's `apply` method to return the module's output. This can be referred to as creating a functional interface for module construction. If we apply this on the standard mux2 example, we would to return the mux2 output ports when we used mux2 in an expression. Implementing this requires building a constructor that takes multiplexer inputs as parameters and returns the multiplexer output:

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

As we can see in the code example, we defined the `apply` method to take the Mux2 inputs as the method parameters, and return the Mux2 output as the function's return value. By defining modules in this way, it is easier to later implement larger and more complex version of this regular module. For example, we previously implemented Mux4 like this:

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

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

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

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

However, by using the creation function we redefined for Mux2, we can now use the Mux2 outputs as values of the modules themselves when writing the Mux4 output expression:

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

This allows to write more intuitively readable hardware connection descriptions, which are similar to software expression evaluation.

Prev(Interfaces \& Bulk Connections) Next (Muxes and Input Selection)

Muxes and Input Selection

Selecting inputs is very useful in hardware description, and therefore Chisel provides several built-in generic input-selection implementations.

Mux

The first one is `Mux`. This is a 2-input selector. Unlike the `Mux2` example which was presented previously, the built-in `Mux` allows the inputs (`in0` and `in1`) to be any datatype as long as they are the same subclass of `Data`.

by using the functional module creation feature presented in the previous section, we can create multi-input selector in a simple way:

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

MuxCase

However, this is not necessary since Chisel also provides the built-in `MuxCase`, which implements that exact feature. `MuxCase` is an n-way `Mux`, which can be used as follows:

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

Where each selection dependency is represented as a tuple in a Scala array [condition -> selected_input_port].

MuxLookup

Chisel also provides `MuxLookup` which is an n-way indexed multiplexer:

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

This is the same as a `MuxCase`, where the conditions are all index based selection:

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

Note that the conditions/cases/selectors (eg. c1, c2) must be in parentheses.

Mux1H

Another `Mux` utility is `Mux1H` that takes a sequence of selectors and values and returns the value associated with the one selector that is set. If zero or multiple selectors are set the behavior is undefined. For example:

```
  val hotValue = chisel3.util.oneHotMux(Seq(
    io.selector(0) -> 2.U,
    io.selector(1) -> 4.U,
    io.selector(2) -> 8.U,
    io.selector(4) -> 11.U,
  ))
```

`oneHotMux` whenever possible generates *Firrtl* that is readily optimizable as low depth and/or tree. This optimization is not possible when the values are of type `FixedPoint` or an aggregate type that contains `FixedPoint`s and results instead as a simple `Mux` tree. This behavior could be sub-optimal. As `FixedPoint` is still *experimental* this behavior may change in the future.

Prev(Functional Module Creation) Next(Polymorphism and Parameterization)

Polymorphism and Parameterization

This section is advanced and can be skipped at first reading.

Scala is a strongly typed language and uses parameterized types to specify generic functions and classes. In this section, we show how Chisel users can define their own reusable functions and classes using parameterized classes.

Parameterized Functions

Earlier we defined `Mux2` on `Bool`, but now we show how we can define a generic multiplexer function. We define this function as taking a boolean condition and con and alt arguments (corresponding to then and else expressions) of type `T`:

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

where `T` is required to be a subclass of `Bits`. Scala ensures that in each usage of `Mux`, it can find a common superclass of the actual con and alt argument types, otherwise it causes a Scala compilation type error. For example,

```
Mux(c, UInt(10), UInt(11))
```

yields a `UInt` wire because the `con` and `alt` arguments are each of type `UInt`.

Parameterized Classes

Like parameterized functions, we can also parameterize classes to make them more reusable. For instance, we can generalize the Filter class to use any kind of link. We do so by parameterizing the `FilterIO` class and defining the constructor to take a single argument `gen` of type `T` as below.

```
class FilterIO[T <: Data](gen: T) extends Bundle {
  val x = Input(gen)
  val y = Output(gen)
}
```

We can now define `Filter` by defining a module class that also takes a link type constructor argument and passes it through to the `FilterIO` interface constructor:

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

We can now define a `PLink`-based `Filter` as follows:

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

A generic FIFO could be defined as follows:

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

class Fifo[T <: Data](gen: T, n: Int) extends Module {
  val io = IO(new Bundle {
    val enqVal = Input(Bool())
    val enqRdy = Output(Bool())
    val deqVal = Output(Bool())
    val deqRdy = Input(Bool())
    val enqDat = Input(gen)
    val deqDat = Output(gen)
  })
  val enqPtr     = RegInit(0.asUInt(sizeof(n).W))
  val deqPtr     = RegInit(0.asUInt(sizeof(n).W))
  val isFull     = RegInit(false.B)
  val doEnq      = io.enqRdy && io.enqVal
  val doDeq      = io.deqRdy && io.deqVal
  val isEmpty    = !isFull && (enqPtr === deqPtr)
  val deqPtrInc  = deqPtr + 1.U
  val enqPtrInc  = enqPtr + 1.U
  val isFullNext = Mux(doEnq && ~doDeq && (enqPtrInc === deqPtr),
                       true.B, Mux(doDeq && isFull, false.B,
                       isFull))
  enqPtr := Mux(doEnq, enqPtrInc, enqPtr)
  deqPtr := Mux(doDeq, deqPtrInc, deqPtr)
  isFull := isFullNext
  val ram = Mem(n)
  when (doEnq) {
    ram(enqPtr) := io.enqDat
  }
  io.enqRdy := !isFull
  io.deqVal := !isEmpty
  ram(deqPtr) <> io.deqDat
}
```

An Fifo with 8 elements of type DataBundle could then be instantiated as:

```
val fifo = Module(new Fifo(new DataBundle, 8))
```

It is also possible to define a generic decoupled (ready/valid) interface:

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

This template can then be used to add a handshaking protocol to any set of signals:

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

The FIFO interface can be now be simplified as follows:

```
class Fifo[T <: Data](data: T, n: Int) extends Module {
  val io = IO(new Bundle {
    val enq = Flipped(new DecoupledIO(data))
    val deq = new DecoupledIO(data)
  })
  ...
}
```

Parametrization based on Modules

You can also parametrize modules based on other modules rather than just types. The following is an example of a module parametrized by other modules as opposed to e.g. types.

```
import chisel3.experimental.{BaseModule, RawModule}

// Provides a more specific interface since generic Module
// provides no compile-time information on generic module's IOs.
trait MyAdder {
    def in1: UInt
    def in2: UInt
    def out: UInt
}

class Mod1 extends RawModule with MyAdder {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 + in2
}

class Mod2 extends RawModule with MyAdder {
    val in1 = IO(Input(UInt(8.W)))
    val in2 = IO(Input(UInt(8.W)))
    val out = IO(Output(UInt(8.W)))
    out := in1 - in2
}

class X[T <: BaseModule with MyAdder](genT: => T) extends Module {
    val io = IO(new Bundle {
        val in1 = Input(UInt(8.W))
        val in2 = Input(UInt(8.W))
        val out = Output(UInt(8.W))
    })
    val subMod = Module(genT)
    io.out := subMod.out
    subMod.in1 := io.in1
    subMod.in2 := io.in2
}

println(chisel3.Driver.emitVerilog(new X(new Mod1)))
println(chisel3.Driver.emitVerilog(new X(new Mod2)))
```

Prev(Muxes and Input Selection) Next(Multiple Clock Domains)

Multiple Clock Domains

Chisel 3 supports multiple clock domains as follows.

Note that in order to cross clock domains safely, you will need appropriate synchronization logic (such as an asynchronous FIFO). You can use the AsyncQueue library to do this easily.

```
class MultiClockModule extends Module {
  val io = IO(new Bundle {
    val clockB = Input(Clock())
    val resetB = Input(Bool())
    val stuff = Input(Bool())
  })

  // This register is clocked against the module clock.
  val regClock = RegNext(io.stuff)

  withClockAndReset (io.clockB, io.resetB) {
    // In this withClock scope, all synchronous elements are clocked against io.clockB.
    // Reset for flops in this domain is using the explicitly provided reset io.resetB.

    // This register is clocked against io.clockB.
    val regClockB = RegNext(io.stuff)
  }

  // This register is also clocked against the module clock.
  val regClock2 = RegNext(io.stuff)
}
```

You can also instantiate modules in another clock domain:

```
class MultiClockModule extends Module {
  val io = IO(new Bundle {
    val clockB = Input(Clock())
    val resetB = Input(Bool())
    val stuff = Input(Bool())
  })
  val clockB_child = withClockAndReset(io.clockB, io.resetB) { Module(new ChildModule) }
  clockB_child.io.in := io.stuff
}
```

If you only want to connect your clock to a new clock domain and use the regular implicit reset signal, you can use `withClock(clock)` instead of `withClockAndReset`.

```
class MultiClockModule extends Module {
  val io = IO(new Bundle {
    val clockB = Input(Clock())
    val stuff = Input(Bool())
  })

  // This register is clocked against the module clock.
  val regClock = RegNext(io.stuff)

  withClock (io.clockB) {
    // In this withClock scope, all synchronous elements are clocked against io.clockB.

    // This register is clocked against io.clockB, but uses implict reset from the parent context.
    val regClockB = RegNext(io.stuff)
  }

  // This register is also clocked against the module clock.
  val regClock2 = RegNext(io.stuff)
}

// Instantiate module in another clock domain with implicit reset.
class MultiClockModule extends Module {
  val io = IO(new Bundle {
    val clockB = Input(Clock())
    val stuff = Input(Bool())
  })
  val clockB_child = withClock(io.clockB) { Module(new ChildModule) }
  clockB_child.io.in := io.stuff
}
```

Prev(Polymorphism and Parameterization) Next(Chisel3 vs Chisel2)

Printing in Chisel

Chisel provides the `printf` function for debugging purposes. It comes in two flavors:

- Scala-style
- C-style

Scala-style

Chisel also supports printf in a style similar to Scala's String Interpolation. Chisel provides a custom string interpolator `p` which can be used as follows:

```
val myUInt = 33.U
printf(p"myUInt = $myUInt") // myUInt = 33
```

Simple formatting

Other formats are available as follows:

```
// Hexadecimal
printf(p"myUInt = 0x${Hexadecimal(myUInt)}") // myUInt = 0x21
// Binary
printf(p"myUInt = ${Binary(myUInt)}") // myUInt = 100001
// Character
printf(p"myUInt = ${Character(myUInt)}") // myUInt = !
```

We recognize that the format specifiers are verbose, so we are working on a more concise syntax.

Aggregate data-types

Chisel provides default custom "pretty-printing" for Vecs and Bundles. The default printing of a Vec is similar to printing a Seq or List in Scala while printing a Bundle is similar to printing a Scala Map.

```
val myVec = Vec(5.U, 10.U, 13.U)
printf(p"myVec = $myVec") // myVec = Vec(5, 10, 13)

val myBundle = Wire(new Bundle {
  val foo = UInt()
  val bar = UInt()
})
myBundle.foo := 3.U
myBundle.bar := 11.U
printf(p"myBundle = $myBundle") // myBundle = Bundle(a -> 3, b -> 11)
```

Custom Printing

Chisel also provides the ability to specify *custom* printing for user-defined Bundles.

```
class Message extends Bundle {
  val valid = Bool()
  val addr = UInt(32.W)
  val length = UInt(4.W)
  val data = UInt(64.W)
  override def toPrintable: Printable = {
    val char = Mux(valid, 'v'.U, '-'.U)
    p"Message:\n" +
    p"  valid  : ${Character(char)}\n" +
    p"  addr   : 0x${Hexadecimal(addr)}\n" +
    p"  length : $length\n" +
    p"  data   : 0x${Hexadecimal(data)}\n"
  }
}

val myMessage = Wire(new Message)
myMessage.valid := true.B
myMessage.addr := "h1234".U
myMessage.length := 10.U
myMessage.data := "hdeadbeef".U

printf(p"$myMessage")
```

Which prints the following:

```
Message:
  valid  : v
  addr   : 0x00001234
  length : 10
  data   : 0x00000000deadbeef
```

Notice the use of `+` between `p` interpolated "strings". The results of `p` interpolation can be concatenated by using the `+` operator. For more information, please see the documentation

C-Style

Chisel provides `printf` in a similar style to its C namesake. It accepts a double-quoted format string and a variable number of arguments which will then be printed on rising clock edges. Chisel supports the following format specifiers:

Format Specifier	Meaning
`%d`	decimal number
`%x`	hexadecimal number
`%b`	binary number
`%c`	8-bit ASCII character
`%%`	literal percent

It also supports a small set of escape characters:

Escape Character	Meaning
`\n`	newline
`\t`	tab
`\"`	literal double quote
`\'`	literal single quote
`\\`	literal backslash

Note that single quotes do not require escaping, but are legal to escape.

Thus printf can be used in a way very similar to how it is used in C:

```
val myUInt = 32.U
printf("myUInt = %d", myUInt) // myUInt = 32
```

Annotations: Extending Chisel and Firrtl

Annotations are used to mark modules and their elements in a way that can be accessed by Firrtl transformation passes. Custom passes and the annotations that guide their behavior extend the circuit generation capabilities of Chisel/Firrtl. This article focuses on the approach to building a library that contains Annotations and Transforms. We will walk through src/test/scala/chiselTests/AnnotatingDiamondSpec.scala to see the basic concepts.

Imports

We need a few basic imports to reference the components we need. The chisel3 is a standard

```
import chisel3._
import chisel3.internal.InstanceId
import firrtl.{CircuitForm, CircuitState, LowForm, Transform}
import firrtl.annotations.{Annotation, ModuleName, Named}
```

Write a transform

This is an identity transform that returns whatever it is passed without alteration. See Writing Firrtl Transforms for the gory details on writing transforms that actually do something.

```
class IdentityTransform extends Transform {
  override def inputForm: CircuitForm = LowForm
  override def outputForm: CircuitForm = LowForm

  override def execute(state: CircuitState): CircuitState = {
    getMyAnnotations(state) match {
      case Nil => state
      case myAnnotations =>
        // Use annotations contained in the myAnnotations list to modify state
        // and return that modified state.
        state
    }
  }
}
```

This creates a transform that operates on low Firrtl (LowForm) and returns low Firrtl. `getMyAnnotations` returns a list of annotations for your pass. This example does nothing with those annotations.

Create an Annotation Factory

The following creates an annotation that is connected to your transform, note the `classOf[IdentityTransform]`. The unapply is a convenience method for extracting information from your annotation by using the Scala `match` operator.

```
object IdentityAnnotation {
  def apply(target: Named, value: String): Annotation = Annotation(target, classOf[IdentityTransform], value)

  def unapply(a: Annotation): Option[(Named, String)] = a match {
    case Annotation(named, t, value) if t == classOf[IdentityTransform] => Some((named, value))
    case _ => None
  }
}
```

> note `target: Named` identifies a firrtl circuit component. Annotations can refer to specific elements of a Module such as registers or wires, or can point to a Module in the case of some more generic transformation.

Create an Annotator

An Annotator is a trait that only be applied to a Module. It provides an abstraction layer over the underlying Chisel annotation system. In this example, the `identify` annotator takes an kind of circuit component reference (i.e. `InstanceId`) and packages it with `value` to be available in the firrtl pass. The library writer could place restrictions on the type of component and value.

> The `value` passed to the Annotator does not have to be a string, but it must serializable into a string for the `value` parameter of the `ChiselAnnotation` being created.

```
trait IdentityAnnotator {
  self: Module =>
  def identify(component: InstanceId, value: String): Unit = {
    annotate(ChiselAnnotation(component, classOf[IdentityTransform], value))
  }
}
```

Using the Annotator

Here is a module that uses our `IdentityAnnotator`

```
class ModC(widthC: Int) extends Module with IdentityAnnotator {
  val io = IO(new Bundle {
    val in = Input(UInt(widthC.W))
    val out = Output(UInt(widthC.W))
  })
  io.out := io.in

  identify(this, s"ModC($widthC)")

  identify(io.out, s"ModC(ignore param)")
}
```

There are several things to note here. ModC includes the `with IdentityAnnotator` which adds the identity method to it. The `identify(this, s"ModC($widthC)")` annotates an instance of ModC as it is created. It value annotations includes the `widthC` parameter to the constructor. In this case that could be used to distinguish transformation behavior between different instances of ModC. The `identify(io.out, s"ModC(ignore param)")` annotates io.out but with a fixed string. In contrast the previous annotation, multiple instances of ModC would have result in a single `io.out` annotation here.

Unconnected Wires

The Invalidate API (#645) adds support to chisel for reporting unconnected wires as errors.

Prior to this pull request, chisel automatically generated a firrtl `is invalid` for `Module IO()`, and each `Wire()` definition. This made it difficult to detect cases where output signals were never driven. Chisel now supports a `DontCare` element, which may be connected to an output signal, indicating that that signal is intentionally not driven. Unless a signal is driven by hardware or connected to a `DontCare`, Firrtl will complain with a "not fully initialized" error.

API

Output signals may be connected to DontCare, generating a `is invalid` when the corresponding firrtl is emitted.

```
io.out.debug := true.B
io.out.debugOption := DontCare
```

This indicates that the signal `io.out.debugOption` is intentionally not driven and firrtl should not issue a "not fully initialized" error for this signal.

This can be applied to aggregates as well as individual signals:

```
{
  ...
  val nElements = 5
  val io = IO(new Bundle {
    val outs = Output(Vec(nElements, Bool()))
  })
  io.outs <> DontCare
  ...
}

class TrivialInterface extends Bundle {
  val in  = Input(Bool())
  val out = Output(Bool())
}

{
  ...
  val io = IO(new TrivialInterface)
  io <> DontCare
  ...
}
```

This feature is controlled by `CompileOptions.explicitInvalidate` and is set to `false` in `NotStrict` (Chisel2 compatibility mode), and `true` in `Strict` mode.

You can selectively enable this for Chisel2 compatibility mode by providing your own explicit `compileOptions`, either for a group of Modules (via inheritance):

```
abstract class ExplicitInvalidateModule extends Module()(chisel3.core.ExplicitCompileOptions.NotStrict.copy(explicitInvalidate = true))
```

or on a per-Module basis:

```
class MyModule extends Module {
  override val compileOptions = chisel3.core.ExplicitCompileOptions.NotStrict.copy(explicitInvalidate = true)
  ...
}
```

Or conversely, disable this stricter checking (which is now the default in pure chisel3):

```
abstract class ImplicitInvalidateModule extends Module()(chisel3.core.ExplicitCompileOptions.Strict.copy(explicitInvalidate = false))
```

or on a per-Module basis:

```
class MyModule extends Module {
  override val compileOptions = chisel3.core.ExplicitCompileOptions.Strict.copy(explicitInvalidate = false)
  ...
}
```

Please see the corresponding API tests for examples.

Determining the unconnected element

I have an interface with 42 wires. Which one of them is unconnected?

The firrtl error message should contain something like:

```
firrtl.passes.CheckInitialization$RefNotInitializedException:  @[:@6.4] : [module Router]  Reference io is not fully initialized.
   @[Decoupled.scala 38:19:@48.12] : node _GEN_23 = mux(and(UInt<1>("h1"), eq(UInt<2>("h3"), _T_84)), _GEN_2, VOID) @[Decoupled.scala 38:19:@4
8.12]
   @[Router.scala 78:30:@44.10] : node _GEN_36 = mux(_GEN_0.ready, _GEN_23, VOID) @[Router.scala 78:30:@44.10]
   @[Router.scala 75:26:@39.8] : node _GEN_54 = mux(io.in.valid, _GEN_36, VOID) @[Router.scala 75:26:@39.8]
   @[Router.scala 70:50:@27.6] : node _GEN_76 = mux(io.load_routing_table_request.valid, VOID, _GEN_54) @[Router.scala 70:50:@27.6]
   @[Router.scala 65:85:@19.4] : node _GEN_102 = mux(_T_62, VOID, _GEN_76) @[Router.scala 65:85:@19.4]
   : io.outs[3].bits.body <= _GEN_102
```

The first line is the initial error report. Successive lines, indented and beginning with source line information indicate connections involving the problematic signal. Unfortunately, if these are `when` conditions involving muxes, they may be difficult to decipher. The last line of the group, indented and beginning with a `:` should indicate the uninitialized signal component. This example (from the Router tutorial) was produced when the output queue bits were not initialized. The old code was:

```
  io.outs.foreach { out => out.noenq() }
```

which initialized the queue's `valid` bit, but did not initialize the actual output values. The fix was:

```
  io.outs.foreach { out =>
    out.bits := 0.U.asTypeOf(out.bits)
    out.noenq()
  }
```

Acknowledgements

Many people have helped out in the design of Chisel, and we thank them for their patience, bravery, and belief in a better way. Many Berkeley EECS students in the Isis group gave weekly feedback as the design evolved including but not limited to Yunsup Lee, Andrew Waterman, Scott Beamer, Chris Celio, etc. Yunsup Lee gave us feedback in response to the first RISC-V implementation, called TrainWreck, translated from Verilog to Chisel. Andrew Waterman and Yunsup Lee helped us get our Verilog backend up and running and Chisel TrainWreck running on an FPGA. Brian Richards was the first actual Chisel user, first translating (with Huy Vo) John Hauser's FPU Verilog code to Chisel, and later implementing generic memory blocks. Brian gave many invaluable comments on the design and brought a vast experience in hardware design and design tools. Chris Batten shared his fast multiword C++ template library that inspired our fast emulation library. Huy Vo became our undergraduate research assistant and was the first to actually assist in the Chisel implementation. We appreciate all the EECS students who participated in the Chisel bootcamp and proposed and worked on hardware design projects all of which pushed the Chisel envelope. We appreciate the work that James Martin and Alex Williams did in writing and translating network and memory controllers and non-blocking caches. Finally, Chisel's functional programming and bit-width inference ideas were inspired by earlier work on a hardware description language called *Gel* designed in collaboration with Dany Qumsiyeh and Mark Tobenkin.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avizienis, Wawrzynek, J., Asanovic, K.
Chisel: Constructing Hardware in a Scala Embedded Language.
in DAC '12.
Bachrach, J., Qumsiyeh, D., Tobenkin, M.
Hardware Scripting in Gel.
in Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th.

Prev(Chisel3 vs Chisel2)

Chisel3 vs Chisel2

Chisel3 vs Chisel2

Chisel2 Migration

For those moving from Chisel2, there were some backwards incompatible changes and your RTL needs to be modified to work with Chisel3. The required modifications are:

- Wire declaration style:

  ```
  val wire = UInt(width = 15)
  ```

  becomes (in Chisel3):

  ```
  val wire = Wire(UInt(15.W))
  ```

- I/O declaration style:

  ```
  val done = Bool(OUTPUT)
  ```

  becomes (in Chisel3):

  ```
  val wire = Output(Bool())
  ```

- Sequential memories:

  ```
  val addr = Reg(UInt())
  val mem = Mem(UInt(8.W), 1024, seqRead = true)
  val dout = when(enable) { mem(addr) }
  ```

  becomes (in Chisel3):

  ```
  val addr = UInt()
  val mem = SyncReadMem(1024, UInt(8.W))
  val dout = mem.read(addr, enable)
  ```

  Notice the address register is now internal to the SyncReadMem(), but the data will still return on the subsequent cycle.

- Generating Verilog with

  ```
  object Hello {
    def main(args: Array[String]): Unit = {
      chiselMain(Array("--backend", "v"), () => Module(new Hello()))
    }
  }
  ```

  becomes (in Chisel3):

  ```
  object Hello {
    def main(args: Array[String]): Unit = {
      chisel3.Driver.execute(Array[String](), () => new Hello())
    }
  }
  ```

- Package changes:
  - Chisel.log2Ceil -> chisel3.util.log2Ceil
  - BitPat
  - Decoupled is also in chisel3.util

Please refer to the Chisel3 compatibility section for instructions on preparing your Chisel2 designs for Chisel3.

Deprecated Usage

- `Vec(Reg)` should be replaced with `Reg(Vec)`,
- type-only vals (no associated data) must be wrapped in a `Wire()` if they will be the destination of a wiring operation (":=" or " < >"),
- masked bit patterns ('b??') should be created using BitPat(), not UInt() or Bits(),
- the `clone` method required for parameterized Bundles has been renamed `cloneType`,
- the con and alt inputs to a Mux must be type-compatible - both signed or both unsigned,
- bulk-connection to a node that has been procedurally assigned-to is illegal,
- `!=` is deprecated, use `=/=` instead,
- use `SyncReadMem(...)` instead of `Mem(..., seqRead)`,
- use `SyncReadMem(n:Int, out: => T)` instead of `SyncReadMem(out: => T, n:Int)`,
- use `SyncReadMem(...)` instead of `SeqMem(...)`,
- use `Mem(n:Int, t:T)` instead of `Mem(out:T, n:Int)`,
- use `Vec(n:Int, gen: => T)` instead of `Vec(gen: => T, n:Int)`,
- module io's must be wrapped in `IO()`.
- The methods `asInput`, `asOutput`, and `flip` should be replaced by the `Input()`, `Output()`, and `Flipped()` object apply methods.

Unsupported constructs

- `Mem(..., orderedWrites)` is no longer supported,
- masked writes are only supported for `Mem[Vec[_]]`,
- Chisel3 Vecs must all have the same type, unlike with Chisel2. Use `MixedVec` (see Bundles and Vecs) for Vecs where the elements are of different types.
- connections between `UInt` and `SInt` are illegal.
- the `Node` class and object no longer exist (the class should have been private in Chisel2)
- `debug()` has been removed (use `dontTouch` instead)
- `printf()` is defined in the Chisel object and produces simulation printf()'s. To use the Scala `Predef.printf()`, you need to qualify it with `Predef`.
- In Chisel2, bulk-connects `<>` with unconnected source components do not update connections from the unconnected components.
    - In Chisel3, bulk-connects strictly adhere to last connection semantics and unconnected OUTPUTs will be connected to INPUTs resulting in the assignment of random values to those inputs.
- In Chisel3, adding hardware inside `BlackBox` for simulation is no longer supported. (#289)
- `ChiselError` is gone
    - Change `ChiselError.error("error msg")` to `throw new Error("error msg")`
    - Change `ChiselError.info("info msg")` to `println("info msg")`
- In Chisel3, subword assignments are not supported. Alternative constructions exist in Chisel3.

Further changes

- The clock signal was renamed from `clk` to `clock` in Chisel3.
- Changed `getWidth()` to `getWidth`

Packaging

Chisel3 is implemented as several packages. The core DSL is differentiated from utility or library classes and objects, testers, and interpreters. The prime components of the Chisel3 front end (the DSL and library objects) are:

- coreMacros - source locators provide Chisel line numbers for `firrtl` detected errors,
- chiselFrontend - main DSL components,
- chisel3 - compiler driver, interface packages, compatibility layer.

Due to the wonders of `sbt`, you need only declare a dependency on the chisel3 package, and the others will be downloaded as required.

The `firrtl` compiler is distributed as a separate package, and release versions will also be downloaded automatically as required. If you choose to integrate the compiler into your own toolchain, or you're working with the development (master) branch of chisel3, you should clone the firrtl repo and follow the instructions for installing the `firrtl` compiler.

The testers in Chisel3 are distributed as a separate package. If you intend to use them in your tests, you will either need to clone the chisel-testers repo or declare a dependency on the published version of the package. See the `build.sbt` file in either the chisel-template or chisel-tutorial repos for examples of the latter.

Simulation

Chisel2 was capable of directly generating a `C++` simulation from the Chisel code, or a harness for use with a `vcs` simulation. Chisel3 relies on verilator to generate the `C++` simulation from the Verilog output of `firrtl`. See the Chisel3 README for directions on installing `verilator`.

Compile Options and Front End Checks (Strict vs. NotStrict)

Chisel3 introduces a formal specification for hardware circuit graphs: FIRRTL, and Chisel3 itself (the Scala library implementing the Chisel DSL), is a relatively thin front end that generates FIRRTL. Since the `firrtl` parser needs to validate FIRRTL input, most of the checks that were performed in Chisel2 were eliminated from the initial Chisel3 front end (the DRY principle).

However, this does impact the ability to provide detailed messages for error conditions that could be detected in the Chisel3 front end. The decision was made to optionally enable stricter error checking (for connections and the use of raw types versus hardware objects), based on specific imports. This allows designs to move from less strict front end checks (largely compatible with Chisel2), to stricter checking, on a file by file basis, by adjusting specific import statements.

```
import chisel3.core.ExplicitCompileOptions.Strict
```

enables stricter connection and usage checks, while

```
import chisel3.core.ExplicitCompileOptions.NotStrict
```

defers these checks to the `firrtl` compiler.

By default, the `Chisel` compatibility layer, invoked by:

```
import Chisel._
```

implicitly defines the compile options as `chisel3.core.ExplicitCompileOptions.NotStrict`

whereas the Chisel3 package, invoked by:

```
import chisel3._
```

implicitly defines the compile options as `chisel3.core.ExplicitCompileOptions.Strict`

Again, these implicit compile options definitions may be overridden by explicit imports.

Currently, the specific error checks (found in CompileOptions.scala) are:

```
trait CompileOptions {
  // Should Bundle connections require a strict match of fields.
  // If true and the same fields aren't present in both source and sink, a MissingFieldException,
  // MissingLeftFieldException, or MissingRightFieldException will be thrown.
  val connectFieldsMustMatch: Boolean
  // When creating an object that takes a type argument, the argument must be unbound (a pure type).
  val declaredTypeMustBeUnbound: Boolean
  // Module IOs should be wrapped in an IO() to define their bindings before the reset of the module is defined.
  val requireIOWrap: Boolean
  // If a connection operator fails, don't try the connection with the operands (source and sink) reversed.
  val dontTryConnectionsSwapped: Boolean
  // If connection directionality is not explicit, do not use heuristics to attempt to determine it.
  val dontAssumeDirectionality: Boolean
  // Issue a deprecation warning if Data.{flip, asInput,asOutput} is used
  // instead of Flipped, Input, or Output.
  val deprecateOldDirectionMethods: Boolean
  // Check that referenced Data have actually been declared.
  val checkSynthesizable: Boolean
}
```

`chisel3.core.ExplicitCompileOptions.Strict` sets all CompileOptions fields to true and `chisel3.core.ExplicitCompileOptions.NotStrict` sets them all to false. Clients are free to define their own settings for these options. Examples may be found in the test CompileOptionsSpec

Prev(Multiple Clock Domains) Next(Acknowledgements)

Experimental Features

Chisel has a number of new features that are worth checking out. This page is an informal list of these features and projects.

Interval Type

See Experimental - Interval Types

FixedPoint

FixedPoint numbers are basic *Data* type along side of UInt, SInt, etc. Most common math and logic operations are supported. Chisel allows both the width and binary point to be inferred by the Firrtl compiler which can simplify circuit descriptions. See: FixedPoint and [FixedPointSpec]blob/master/src/test/scala/chiselTests/FixedPointSpec.scala)

Module Variants

The standard Chisel *Module* requires a `val io = IO(...)`, the experimental package introduces several new ways of defining Modules

- BaseModule: no contents, instantiable
- BlackBox extends BaseModule
- UserDefinedModule extends BaseModule: this module can contain Chisel RTL. No default clock or reset lines. No default IO. - User should be able to specify non-io ports, ideally multiple of them.
- ImplicitModule extends UserModule: has clock, reset, and io, essentially current Chisel Module.
- RawModule: will be the user-facing version of UserDefinedModule
- Module: type-aliases to ImplicitModule, the user-facing version of ImplicitModule.

Bundle Literals

Chisel 3.2 introduces an experimental mechanism for Bundle literals in #820, but this feature is largely incomplete and not ready for user code yet. The following is provided as documentation for library writers who want to take a stab at using this mechanism for their library's bundles.

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

  // Bundle literal constructor code, which will be auto-generated using macro annotations in
  // the future.
  import chisel3.core.BundleLitBinding
  import chisel3.internal.firrtl.{ULit, Width}

  // Full bundle literal constructor
  def Lit(aVal: UInt, bVal: Bool): MyBundle = {
    val clone = cloneType
    clone.selfBind(BundleLitBinding(Map(
      clone.a -> litArgOfBits(aVal),
      clone.b -> litArgOfBits(bVal)
    )))
    clone
  }

  // Partial bundle literal constructor
  def Lit(aVal: UInt): MyBundle = {
    val clone = cloneType
    clone.selfBind(BundleLitBinding(Map(
      clone.a -> litArgOfBits(aVal)
    )))
    clone
  }
}
```

Example usage:

```scala
val outsideBundleLit = (new MyBundle).Lit(42.U, true.B)
```

Developers

Developer Documentation

Tips and tricks for Chisel developers.

- Embedding Chisel as an sbt subproject
- Test Coverage

sbt subproject

Chisel as an sbt subproject

In order to use the constructs defined in the Chisel3 library, those definitions must be made available to the Scala compiler at the time a project dependent on them is compiled. For sbt-based builds there are fundamentally two ways to do this:

- provide a library dependency on the published Chisel3 jars via sbt's `libraryDependencies` setting,
- clone the Chisel3 git repository and include the source code as a subproject of a dependent project.

The former of these two approaches is used by the chisel-tutorial project. It is the simplest approach and assumes you do not require tight control over Chisel3 source code and are content with the published release versions of Chisel3.

The latter approach should be used by Chisel3 projects that require finer control over Chisel3 source code.

It's hard to predict in advance the future requirements of a project, and it would be advantageous to be able to switch between the two approaches relatively easily. In order to accomplish this, we provide the `sbt-chisel-dep` plugin that allows the developer to concisely specify Chisel3 subproject dependencies and switch between subproject and library dependency support based on the presence of a directory (or symbolic link) in the root of the dependent project.

The chisel-template project uses this plugin to support switching between either dependency (subproject or library). By default, the chisel-template project does not contain a chisel3 subproject directory, and hence, uses a library dependency on chisel3 (and related Chisel3 projects). However, if you clone the chisel3 GitHub project from the root directory of the chisel-template project, creating a chisel3 subdirectory, the `sbt-chisel-dep` plugin will take note of the chisel3 project subdirectory, and provide an sbt subproject dependency in place of the library dependency.

Checkout the README for the `sbt-chisel-dep` project for instructions on its usage.

Example versions of the build.sbt and specification of the sbt-chisel-dep plugin are available from the skeleton branch of the chisel-template repository.

Test Coverage

Test Coverage Setup

Chisel's sbt build instructions contain the requisite plug-in (sbt-scoverage) for generating test coverage information. Please see the sbt-scoverage web page for details on the plug-in. The tests themselves are found in `src/test/scala`.

Generating A Test Coverage Report

Use the following sequence of sbt commands to generate a test coverage report:

```
sbt clean coverage test
sbt coverageReport
```

The coverage reports should be found in `target/scala-x.yy/scoverage-report/{scoverage.xml,index.html}` where `x.yy` corresponds to the version of Scala used to compile Firrtl and the tests. `scoverage.xml` is useful if you want to analyze the results programmatically. `index.html` is designed for navigation with a web browser, allowing one to drill down to invidual statements covered (or not) by the tests.