# Chisel Breakdown 3

Jack Koenig, SiFive
27 August 2022

# Purpose of this Talk

- Teach current and potential **developers** of the Chisel codebase (or related projects) about the current implementation of Chisel (v3.5.4)
  - Users do **not** need to know most of this
- Primary focus is on the “weird stuff”
- I’m assuming moderate knowledge of using Chisel and Scala
- I apologize if this talk seems to jump a lot, there is too much to cover
- This is not a “Chisel Update”, please see my talk at the CHIPS Alliance Workshop

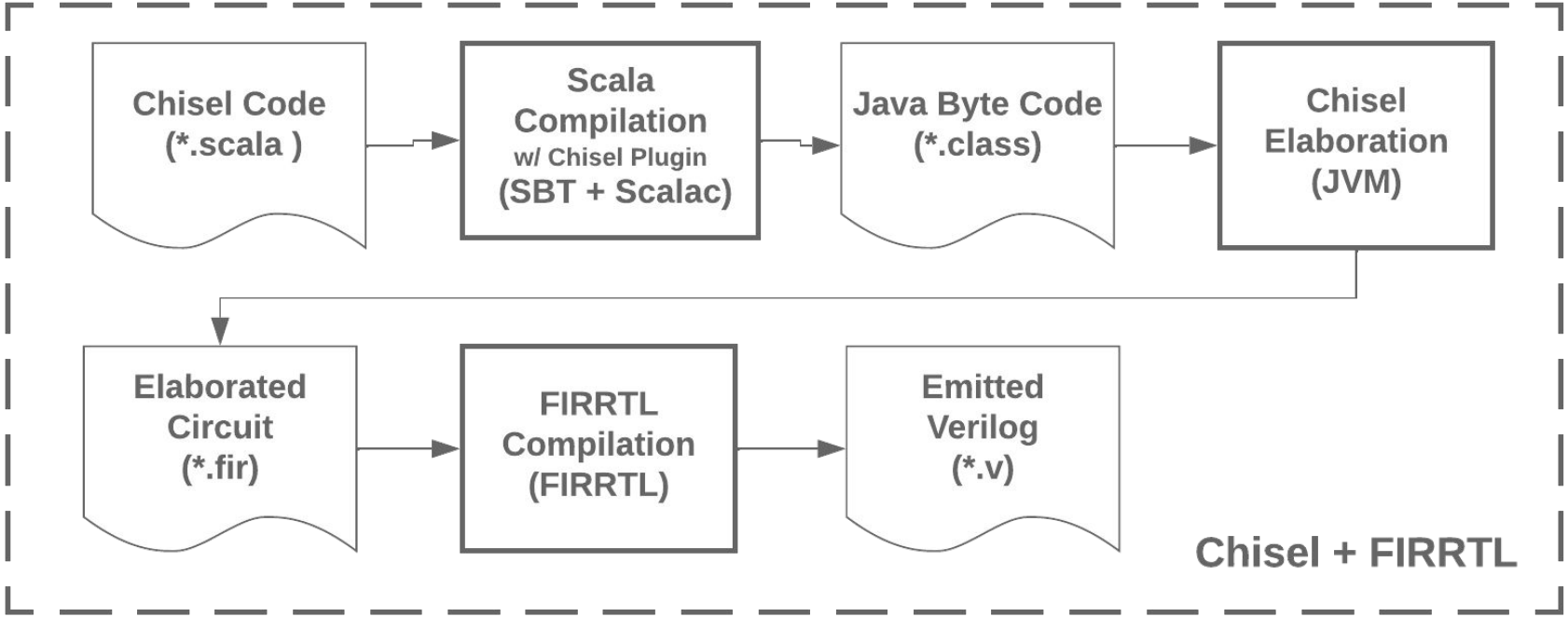
# What Is Chisel?  (from many past talks)

- Hardware Construction Language Embedded in Scala
- NOT high-level synthesis (HLS)
- Domain Specific Language for digital design
- Register, Mux, Wire are objects, use Scala to productively put them together
  - Parameterized types
  - Object-oriented programming
  - Functional programming
  - Static Typing
- Embedded in Scala, meta-programming language and the actual hardware construction language are the same
- **Intended as a platform upon which to build higher-level abstractions**

# What Is Chisel… Really?

- A library of Scala objects and classes for representing hardware
  - Types: eg. UInt, SInt, Vec, Bundle, Clock, AsyncReset
  - Hardware: eg. Reg, Wire, IO, Mux, Mem
  - Structure: eg. Module, when
- A hardware graph
  - Chisel IR (not to be confused with CHIRRTL or FIRRTL)
  - Chisel functions construct IR nodes and **mutate global data structures**
  - Chisel IR can be Emitted or Converted to CHIRRTL
- A Scala compiler plugin (new in v3.4, required in v3.5)
  - Naming
  - AutoCloneType2 (new in v3.4.3)
  - GenBundleElements (new in v3.5.1)
- A lean **frontend** for FIRRTL
  - Some information is not available

# Chisel To Verilog



# Where are errors caught?

- Scala compilation
  - Eg. Cannot call + on a Bundle
- Chisel elaboration (Scala runtime)
  - Eg. Binding exceptions, bad connections
- FIRRTL compilation
  - Eg. Initialization, invalid reset, combinational loop detection
- Verilog compilation
  - Usually a bug in Chisel/FIRRTL
- Simulation
  - Logical errors

# Simple Elaboration Example

```
val wire = Wire(UInt(8.W))
wire := foo & bar
```

```
DefWire(_, UInt(8.W))
DefPrim(_, _wire_T, BitAndOp, Seq(foo, bar))
Connect(_, wire, _wire_T)
```

In ChiselIR, these are called **Commands**
Stored in the containing Module (called **Component** in ChiselIR)

# Binding (Chisel's type system)

- Chisel has its own type system that exists at *elaboration time*
- The ***Scala Type*** of a ***Chisel Type*** and a ***Chisel Hardware Value*** are the same
- Chisel elaboration differentiates via ***Binding***

```
val template: UInt = UInt(8.W)      // This object is a Chisel Type
val wire: UInt = Wire(template)     // This object is a Chisel Value
template.getClass == wire.getClass // true
template === wire                   // Binding exception, template is not hardware
```

# Call By-Name

How does Chisel know the difference between the following?

```
val mod  = new MyModule // Error: attempted to instantiate a Module without
val bad  = Module(mod)             wrapping it in Module().

val good = Module(new MyModule)


object Module {
  def apply[T <: BaseModule](bc: => T): T = {
    // Setup
    val module: T = bc // bc is a function invoked here
    // Teardown
}
```

# Chisel Project Structure

- **plugin**
  - Scalac compiler plugin
- **macros**
- **core**
  - Bulk of Chisel implementation
- **chisel3** (src/main/scala)
  - Stage/Phase/main
  - util
- test (src/test/scala)
  - This is where tests/examples go
- docs
  - Documentation (using mdoc)
- stdlib
  - Future standard library
- integration-tests
  - Tests that use chiseltest

*We use subprojects because a Scala macro cannot be used in the same compilation unit within which it was defined*

# macros/

- Contains macros used by Chisel
- RuntimeDeprecated (@runtimeDeprecated(...))
- SourceInfoTransform
  - Despite the name, does **not** provide source locators
  - Allows for chaining apply (bit extraction) after Chisel functions
  - More on this later
- Literal Bit Extract Warning
  - 3.U(8) vs. 3.U(8.W) vs. myUInt(8) -- new warning in v3.5.4
- RangeTransform (range"...")
  - Interval Types
- ChiselName (@chiselName)
  - Removed in v3.6.0

# core/

- Definitions of user-facing Chisel classes and objects
  - Eg. Module, Data, Wire, when, printf
- CompileOptions (macro and materializer)
  - Materializer defined in core/ to ensure we only materialize in user-code
- internal/
  - Builder
  - firrtl/{IR, Converter} — aka ChiselIR
  - SourceInfo (macro and materializer)
    - Creates source locators
    - Defined in core/ to ensure we only materialize in user-code

# chisel3 (src/main/scala/chisel3/)

- stage/ChiselStage.scala (and friends)
  - This is where you **invoke** Chisel
  - Used for constructing custom compiler flows
- compatibility.scala
  - import Chisel._
- util
  - Decoupled, Queue, Arbiter, log2Ceil
- aop
  - Aspect-oriented programming library

# plugin/

- Scalac compiler plugin — runs as part of **Scala compilation**
- ChiselComponent
  - Name signals
- BundleComponent
  - Autoclonetype2
  - GenBundleElements

# docs/ - Documentation That Works

- docs/src
  - Run with:  sbt docs/mdoc
  - Results in: docs/generated
- Uses mdoc to make sure code examples are **compiled** and **run**
- Can use result of running code examples in emitted mdoc
  - Eg. show Verilog emitted by example

# docs/ - Documentation That Works

## Bundle Literals

Bundle literals can be constructed via an experimental import:

```
import chisel3._
import chisel3.experimental.BundleLiterals._

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

class Example extends RawModule {
  val out = IO(Output(new MyBundle))
  out := (new MyBundle).Lit(_.a -> 8.U, _.b -> true.B)
}
```

```
module Example(
  output [7:0] out_a,
  output       out_b
);
  assign out_a = 8'h8; // @[experimental-features.md 22:7]
  assign out_b = 1'h1; // @[experimental-features.md 22:7]
endmodule
```

```
### Bundle Literals <a name="bundle-literals"></a>

Bundle literals can be constructed via an experimental import:

```scala mdoc
import chisel3._
import chisel3.experimental.BundleLiterals._

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

class Example extends RawModule {
  val out = IO(Output(new MyBundle))
  out := (new MyBundle).Lit(_.a -> 8.U, _.b -> true.B)
}
```

```scala mdoc:verilog
chisel3.stage.ChiselStage.emitVerilog(new Example)
```
```

# ChiselStage

- src/main/scala/chisel3/stage/ChiselStage.scala
- User-facing API for **invoking** Chisel
- Used by ChiselMain.main
  - Command-line interface
- Lots of utility methods for generating Chirrtl, FIRRTL, Verilog
- class ChiselStage vs. object ChiselStage
  - The **class** writes files and is used for custom flows
  - The **object** does not write files, used for little examples / test
  - This distinction isn't very clear, object ChiselStage to be replaced
- Simplified API
  - chisel3.getVerilogString
  - chisel3.emitVerilog

# Builder

- core/src/main/scala/chisel3/internal/Builder.scala
- chisel3.internal.Builder
- Internal “runtime”
- Internal entry point into Chisel elaboration
- Interface for calls into DynamicContext and ChiselContext
  - Contain global mutable state needed during elaboration
  - Wraps each in scala.util.DynamicVariable
    - Makes global state **thread local**
    - Allows multiple invocations of [single-threaded] Chisel elaboration
- All calls mutating global state must go through the Builder
  - Eg. currentModule, pushOp, error

# CompileOptions (Compatibility Mode)

- Options to enable Chisel2-like semantics within Chisel3
- Essentially a set of options that are implicitly created by importing **chisel3._** or **Chisel._**
- Implicitly passed to Chisel functions and alter behavior
  - chisel3 uses **ExplicitCompileOptions.Strict**
  - Compatibility mode uses **ExplicitCompileOptions.NotStrict**
- It is possible (but ill-advised) to mix and match specific options
  - See src/test/scala/chiselTests/CompileOptionsTest.scala
- Finally deprecated in Chisel 3.6.0

# Source Locators

Recall that **SourceInfoTransform** is defined in macros while **SourceInfo** is defined in core

From Bits.scala (core)

```
final def & (that: Bool): Bool = macro SourceInfoTransform.thatArg
def do_& (that: Bool)(implicit sourceInfo: SourceInfo, compileOptions: CompileOptions): Bool
```

All SourceInfoTransform does is replace invocations of **&** in user code with **do_&**

Why is this useful?

# Source Locators

```
final def & (that: Bool): Bool = macro SourceInfoTransform.thatArg
def do_& (that: Bool)(implicit sourceInfo: SourceInfo, compileOptions: CompileOptions): Bool
```

Allows chaining apply (bit extract) after &

```
(myWire & 0xf0.U)(5)
myWire.do_&(0xf0.U)(SourceInfo("File.scala", 20, 8), compileOptions)(5)
```

Without **SourceInfoTransform**, Scala would look for SourceInfo and CompileOptions in the second parameter list

# Chisel Naming Plugin

The compiler plugin inspects your Chisel source code, finding Data objects that are assigned to vals and inserting both naming and prefixing

```
class Example extends RawModule {
  val a, b, c  = IO(Input(UInt(8.W)))
  val out      = IO(Output(UInt()))
  def func() = {
    val x = a + b
    val y = x - 3.U
    y & 0xcf.U
  }
  val result = func() | 0x8.U
  out := result
}
```

# Chisel Naming Plugin

```
def func() = {
  val x = a + b
  val y = x - 3.U
  y & 0xcf.U
}
val result = func() | 0x8.U
out := result
```

```
wire [7:0] result_x = a + b;
wire [7:0] result_y = result_x - 8'h3;
wire [7:0] _result_T = result_y & 8'hcf;
assign out = _result_T | 8'h8;
```

# Chisel Naming Plugin

```
def func() = {
  // val x = a + b
  val x = autoNameRecursively("x")(prefix("x")(a + b))
  // val y = x - 3.U
  val y = autoNameRecursively("y")(prefix("y")(x - 3.U))
  y & 0xcf.U
}
// val result = func() | 0x8.U
val result = autoNameRecursively("result")(prefix("result")(func() | 0x8.U))
out := result // := also prefixes based on the LHS
```

# Chisel Naming Plugin

What happens when there are multiple possible names?

```
val foo = {
  val bar = Wire(UInt(8.W))
  bar := 0.U
  bar
}
val fizz = foo
```

Answer: First name in outer-most scope wins

The name of the wire will be “foo”, but how?

# Chisel Naming Plugin

By-name parameters :)

```
val foo = autoNameRecursively("foo")({
  val bar = autoNameRecursively("bar")(Wire(UInt(8.W)))
  bar := 0.U
  bar
})
val fizz = autoNameRecursively("fizz")(foo)

object autoNameRecursively {
  def apply[T <: Any](name: String)(nameMe: => T): T =
    val currentId = internal.Builder.getCurrentId
    val result = nameMe
    if (result._id > currentId) { result.name(nameMe) }
    result
}
```

# CloneType - What & Why

Because Chisel Types are immutable objects, we need to create fresh instances of them in order to create hardware, add directions, etc.

```
val gen = new Bundle { val foo = Input(UInt(8.W)) }
val io = IO(gen)   // We can use gen as a type template
val w = Wire(gen)  // multiple times
w <> io
```

We need the ability to create copies of any Data object

# CloneType

- Implementation detail for getting a copy of a Chisel Type
- Sealed internal types have internal implementation (obviously)
- User-defined types (Bundles and Records) need cloneType
  - Usually very formulaic, just want to pass the same arguments to the constructor

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

  override def cloneType: this.type =
    new MyBundle(w).asInstanceOf[this.type]
}
```

# Plugin AutoCloneType (aka autoclonetype2)

- Replaced original version which was a beautiful reflection mess-terpiece
  - See extra slides for juicy details
- Improvements in new version
  - Works for non-val parameters
  - Works for inner classes
  - **Much** faster
- Introduced in Chisel v3.4.3 as **opt-in**, mandatory in Chisel v3.5.0+

# Plugin AutoCloneType (aka autoclonetype2)

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

  // Generated by the compiler plugin
  override protected def _cloneTypeImpl: Bundle = new MyBundle(w)
  override protected def _usingPlugin: Boolean = true
}

abstract class Bundle {
  override def cloneType: this.type = {
    val clone = _cloneTypeImpl.asInstanceOf[this.type]
    checkClone(clone)
    clone
  }
}
```

# Plugin GenBundleElements

- Replaces old Java runtime reflection implementation of Bundle.elements
- Runs **much** faster than old reflective version
  - 20-30% speedup in Chisel elaboration
- Introduced in Chisel v3.5.1 as **opt-in**, mandatory in Chisel v3.6
  - Technically a backwards incompatible change if you inherit from a Bundle that switches to using genBundleElements
  - `scalacOptions += "-P:chiselplugin:genBundleElements"`

# Plugin GenBundleElements

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

  // Generated by the compiler plugin
  override protected def _elementsImpl: Iterable[(String, Any)] = Vector("field" -> field)
  override protected def _usingPlugin: Boolean = true
}

abstract class Bundle {
  override lazy val elements: SeqMap[String, Data] = {
    VectorMap(_elementsImpl.map {
      case (name, data: Data) =>
        ...
    }: _*)
  }
}
```

# Questions?

# Extra / Old Slides

- **Note: Many of the following slides were copied without update from the previous versions of this talk (given April 2019 and June 2021)**

# Aspects

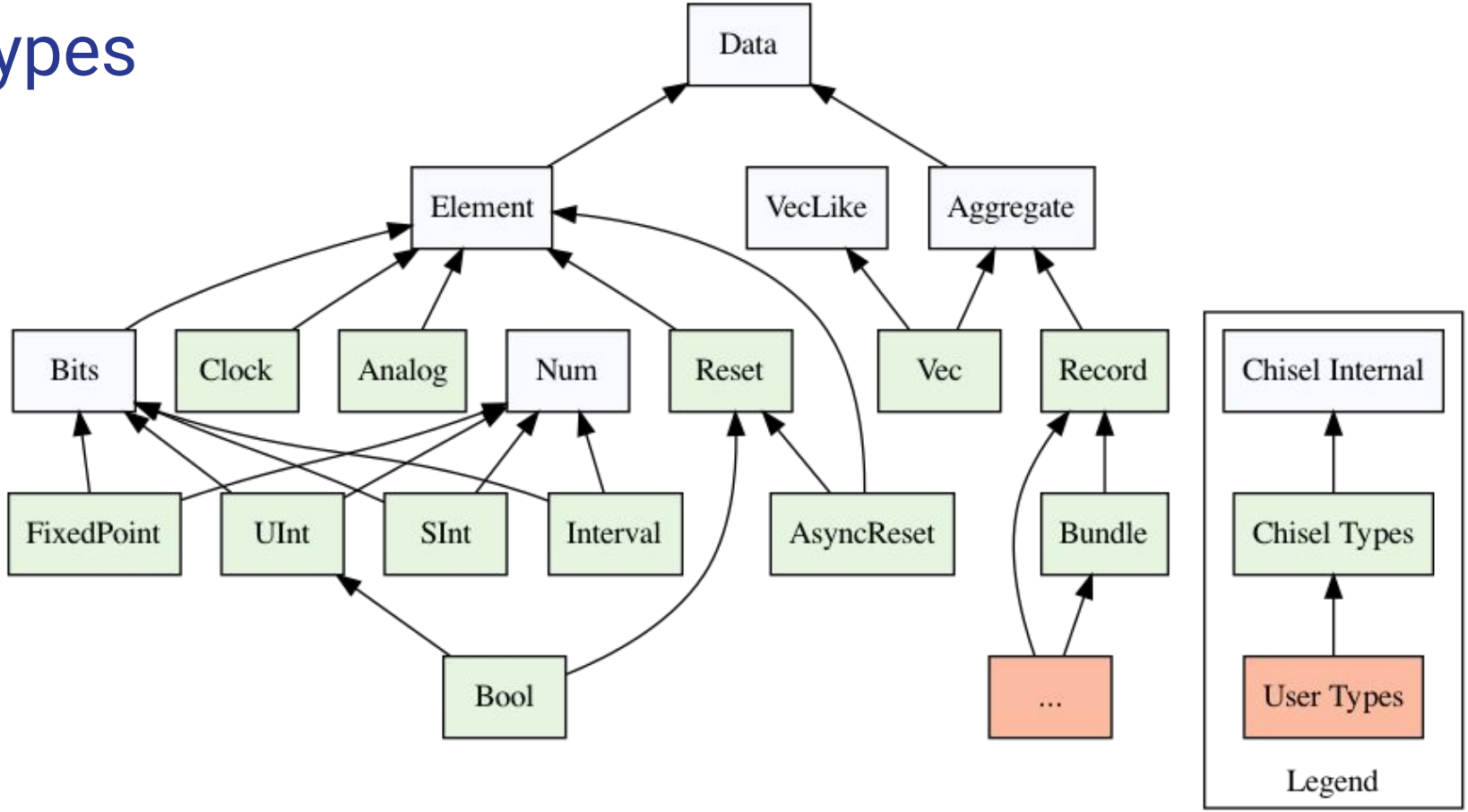
- Runs in AspectPhase after Chisel elaboration (but before FIRRTL)
- Provides Selectors for traversing the elaborated Chisel objects (eg. Modules)
- The sole result of executing an aspect are additional annotations
- These annotations can be used by FIRRTL transforms to
  - Insert additional hardware
  - Generate verification collateral
  - Generate physical design collateral
- Type-safe collateral generation framework for Chisel

# Reflective Naming (removed in 3.6!)

- Recall that the body of a Scala class is the primary constructor
- By default, Chisel names via Java reflection which allows Chisel to inspect **public fields** of a given object

```
class MyModule extends Module {
  val io = IO(...)            // nameable
  val wire = Wire(UInt(8.W)) // nameable
  def func() = {
    val reg = Reg(UInt(8.W)  // NOT nameable by reflection
    io.out := reg + wire
  }
}
```

# Types


Data
Element
VecLike
Aggregate
Bits
Clock
Analog
Num
Reset
Vec
Record
FixedPoint
UInt
SInt
Interval
AsyncReset
Bundle
Bool
...
Chisel Internal
Chisel Types
User Types
Legend

# Widths

- Because Chisel is a lean frontend on Chisel, widths are often not known
- “Why is .getWidth failing, it worked and all I changed was this one small thing!”

```
val x = io.in * io.in
println(x.getWidth)     // 16
val y = WireInit(x * x)
println(y.getWidth)     // 32
val z = Wire(UInt())
z := y * y
println(z.getWidth)     // Exception
```

# Why do we have to wrap Modules in Module(...)

- We need to insert hooks before and **after** Module construction
- For example, to set and unset the **current module**
  - The current module must be known so that we know within which module given hardware lives

# DynamicContext

- Container of Chisel elaboration global state
- Examples:
  - Module namespace
  - Module definitions
  - Current module
  - Current implicit clock and reset
  - Errors (Recoverable ones that can be aggregated and reported later)

# ChiselContext

- Mutable global state that can appear outside of elaboration
- Enables instantiating Chisel types (UInt, Bundle, etc.) outside of Chisel elaboration (eg. in chiseltest)
- Examples:
  - Id generation
  - Prefix Stack

# Record vs. Bundle

- Bundle is like a struct
- Record is the super class of Bundle for programmatic generation of elements
- Requires the user to implement **cloneType** and **elements**

```
val elements: ListMap[String, Data]
```

Bundles use reflection to find and generate these elements

```
class MyBundle {
 val foo = UInt(8.W)
}
```

# IntelliJ/IDE Tips

- Run **sbt compile** on the command-line **before** loading the project in or exporting project to IntelliJ
- Historically, IntelliJ projects do not generically support code generation
- Thus code generation in Chisel and FIRRTL SBT configuration does not work
- If you modify any of the code generation (BuildInfo, ANTLR, ProtoBuf), rerun sbt compile on the command-line
- Some day all of this will be fixed by the Scala Build Server Protocol
  - see Bloop

# @chiselName

@chiselName is a macro that inspects the body of any class or object and finds Chisel Data elements that get assigned to a val which it then names

```
@chiselName
class MyModule extends Module {
  val io = IO(...)            // nameable
  val wire = Wire(UInt(8.W)) // nameable
  def func() = {
    val reg = Reg(UInt(8.W)  // nameable by @chiselName
    io.out := reg + wire
  }
}
```

# Auto-CloneType

- A beautiful reflection mess-terpiece
- Automatically collects arguments to default constructor and constructs new instances with them
- Arguments must be vals (fields of the class) so that reflection can find them
- Suggests making arguments vals if they are not

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

What about inner classes?

# Auto-CloneType (Inner Classes)

- Also known as Path-dependent Types
- The **type** of the inner class is dependent on the **instance** of the outer class
- The instance of the outer class is an implicit argument to the constructor
- For normal inner classes, Scala reflection provides the outer object

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

# Auto-CloneType (Inner Classes)

- Let’s expand that previous example

```
class MyModule(width: Int) extends Module { self =>
  class MyBundle(outer: MyModule) extends Bundle {
    val field = UInt(outer.width.W)
  }
  val io = IO(Output(new MyBundle(self)))
}
```

# Auto-CloneType (Anonymous Inner Classes)

- Unfortunately, for **anonymous** inner classes, Scala reflection does **not** know the outer object
- Fortunately, Chisel knows the current Module, so for anonymous Bundles we can just guess that the outer object is the current Module

```
class MyModule(width: Int) extends Module {
  val io = IO(new Bundle {
    val field = Output(UInt(width.W))
  })
}
```

# Auto-CloneType (Nested Bundles)

- What about outer Bundles?

```
class MyOuterBundle extends Bundle {
  val foo = new Bundle {
    val bar = UInt(8.W)
  }
}
```

# Auto-CloneType (Nested Bundles)

- Unfortunately, since Bundles are not wrapped in some function call (like Module(...)), we don’t know the “current Bundle”
- Instead, upon construction of Bundles we collect stack traces so that we can guess the “outer Bundle”
- Yep, I told you it was a mess-terpiece

# A word on package.scala

```
package object chisel3 { ...
```

In Scala, all defs and vals must be defined in an object or class. This lets us define defs and vals that are included in **import chisel3._**

Why is this useful? For example, type aliasing:

```
@deprecated("MultiIOModule is now just Module", "Chisel 3.5")
type MultiIOModule = chisel3.Module
```

We used to use this for all types, but type aliasing results in bad API docs. Used extensively for `import Chisel._`

# First, some FAQ

Q: Why multiple projects?
A: Scala macros cannot be used in the same project where they were defined

Q: Why {project}/src/main/scala?
A: This is where SBT expects source files, this comes from Maven

# Chisel Simulation Flow