Advanced Chisel Topics

Jonathan Bachrach, **Adam Izraelevitz**, Jack Koenig

EECS UC Berkeley

January 31, 2017

Today

- I’m not Jonathan Bachrach
- Fourth-year PhD student
- I work on the FIRRTL compiler
- Hopefully we can end class early
- Please, interrupt with questions and I’ll do my best to answer them
- Stack overflow!

Today

- so you think you know chisel?

Today

- Debugging
- Modules
- Combinational
- Sequential
- Decoupled
- Scripting
- Vec (and Mems)
- Types
- Object-Oriented Programming

Debugging

- assert, printf

assert

- simulation time assertions are provided by `assert` construct
- if assert arguments false on rising edge then
  - an error is printed and
  - simulation terminates

the following will terminate after 10 clock cycles:

```
val x = Reg(init = 0.U(4.W))
x := x + 1.U
assert(x < 10.U)
```

printf

can be used in when statements:

```
val x = "0x4142".U
when (c) { printf("%d\n", x); }
```

supported format specifiers:

- `%d` decimal
- `%b` binary
- `%h` hexadecimal
- `%s` string
- `%%` literal percent

printf cont.

also supports Scala-like string interpolation:

```
val x = Vec(1.U, 2.U, 3.U, 4.U)
printf(p"Value of x = $x\n" )
// Value of x = Vec(1, 2, 3, 4)
```

For more information, see the wiki page:
https://github.com/ucb-bar/chisel3/wiki/Printing%20in%20Chisel

advanced modules

- black boxes

black boxes

allow users to define interfaces to circuits defined outside of chisel:

```
class RomIo extends Bundle {
  val isVal = Input(Bool())
  val raddr = Input(UInt(32.W))
  val rdata = Output(UInt(32.W))
}

class Rom extends BlackBox {
  val io = IO(new RomIo())
}
```

- names will not contain IO in emitted code

```
val io = IO(new Bundle{
  val i = Input(UInt(8.W)); val o = Input(UInt(8.W)) })
io.i // will become i, not io_i
io.o // will become o, not io_o
```

advanced combinational

- bits properties
- bits functions
- priority encoding functions
- priority mux functions

Bits Properties

```
object log2Up {
  def apply(in: Int): Int = if(in == 1) 1 else ceil(log(in)/log(2)).toInt
}

object log2Down {
  def apply(x : Int): Int = if (x == 1) 1 else floor(log(x)/log(2.0)).toInt
}

object isPow2 {
  def apply(in: Int): Boolean = in > 0 && ((in & (in-1)) == 0)
}

object PopCount {
  def apply(in: Iterable[Bool]): UInt = ...
  def apply(in: Bits): UInt = ...
}
```

Bits Functions

- LFSR16 – random number generator
- Reverse – reverse order of bits
- FillInterleaved – space out booleans into uint

```
object LFSR16 {
  def apply(increment: Bool = true.B): UInt = ...
}
object Reverse {
  def apply(in: UInt): UInt = ...
}
object FillInterleaved {
  def apply(n: Int, in: Bits): UInt = ...
}
```

Priority Encoding Functions

- UIntToOH – returns one hot encoding of input int
- OHToUInt – returns int version of one hot encoding input
- Mux1H – builds mux tree of input vector using a one hot encoded select signal

```
object UIntToOH {
  def apply(in: Bits, width: Int = -1): Bits = ...
}
object OHToUInt {
  def apply(in: Iterable[Bool]): UInt = ...
}
object Mux1H {
  def apply[T <: Data](sel: Iterable[Bool], in: Iterable[T]): T = ...
  def apply[T <: Data](sel: Bits, in: Iterable[T]): T = ...
  def apply[T <: Data](sel: Bits, in: Bits): T = ...
  def apply[T <: Data](sel: Iterable[Bool], in: Bits): T = ...
  def apply[T <: Data](sel: Iterable[(Bool, T)]): T = ...
}
```

Priority Mux Function

- PriorityMux – build mux tree allow multiple select signals with priority given to first select signal

```
object PriorityMux {
  def apply[T <: Bits](in: Iterable[(Bool, T)]): T = ...
  def apply[T <: Bits](sel: Iterable[Bool], in: Iterable[T]): T = ...
  def apply[T <: Bits](sel: Bits, in: Iterable[T]): T = ...
}
```

Priority Encoding Functions

- PriorityEncoder – returns the bit position of the trailing 1 in the input vector with the assumption that multiple bits of the input bit vector can be set
- PriorityEncoderOH – returns the bit position of the trailing 1 in the input vector with the assumption that only one bit in the input vector can be set.

```
object PriorityEncoder {
  def apply(in: Iterable[Bool]): UInt = ...
  def apply(in: Bits): UInt = ...
}

object PriorityEncoderOH {
  def apply(in: Bits): UInt = ...
  def apply(in: Iterable[Bool]): Iterable[UInt] = ...
}
```

advanced sequential

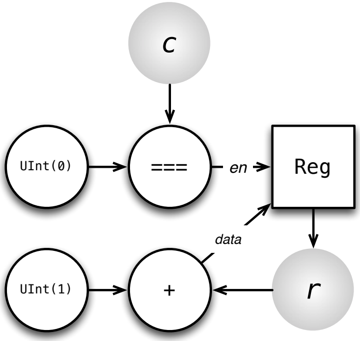
- conditional update rules
- state machines
- reg forms
- counters
- delaying
- examples

Conditional Updates

When describing state operations, we could simply wire register inputs to combinational logic blocks, but it is often more convenient:

- to specify when updates to registers will occur and
- to specify these updates spread across several separate statements

```
val r = Reg( UInt(16.W) )
when (c === 0.U ) {
  r := r + 1.U
}
```



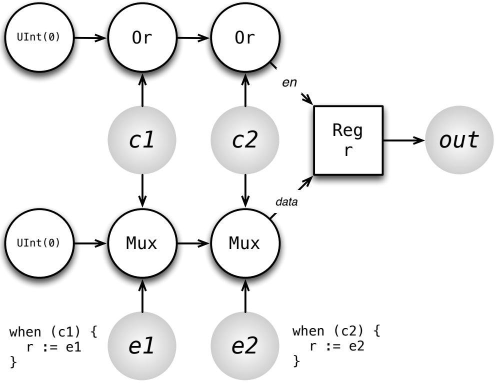
Conditional Updates Priority

```
when (c1) { r := 1.U }
when (c2) { r := 2.U }
```

Conditional Update Order:

c1	c2	r	
0	0	r	r unchanged
0	1	2	
1	0	1	
1	1	2	c2 takes precedence over c1

Conditional Update Synthesized Hardware



```
when (c1) {
  r := e1
}
```

```
when (c2) {
  r := e2
}
```

- Each `when` statement adds another level of data mux and ORs the predicate into the enable chain and
- the compiler effectively adds the termination values to the end of the chain automatically.

Targetting Multiple Registers

```
r := Reg( init = 3.U )
s := Reg( init = 3.U )
when (c1) { r := 1.U; s := 1.U }
when (c2) { r := 2.U }
```

leads to `r` and `s` being updated according to the following truth table:

`c1`	`c2`	`r`	`s`	
0	0	3	3	
0	1	2	3	
1	0	1	1	
1	1	2	1	`r` updated in `c2` block, `s` updated using default

Conditional Update Nesting

```
when (a) { when (b) { body } }
```

which is the same as:

```
when (a && b) { body }
```

Conditional Update Chaining

```
when (c1) { u1 }
.elsewhen (c2) { u2 }
.otherwise { ud }
```

which is the same as:

```
when (c1) { u1 }
when (!c1 && c2) { u2 }
when (!(c1 || c2)) { ud }
```

Switch Statement

```
switch(idx) {
  is(v1) { u1 }
  is(v2) { u2 }
}
```

which is the same as:

```
when (idx === v1) { u1 }
when (idx === v2) { u2 }
```

enums

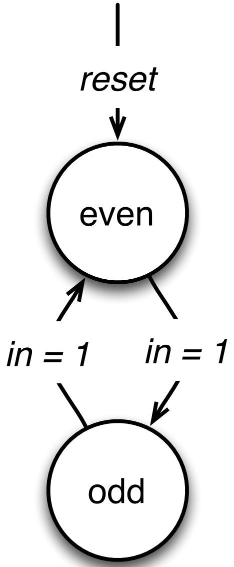
```
object Enum {
  // create n enum values
  def apply (n: Int): Seq[UInt]
}
```

```
// create enum values of given names
val s_even :: s_odd :: Nil = Enum(2)
```

Finite State Machines

Finite state machines can now be readily defined as follows:

```
class Parity extends Module {
  val io = IO(new Bundle {
    val in  = Input(Bool())
    val out = Output(Bool())
  })
  val s_even :: s_odd :: Nil = Enum(2)
  val state  = Reg(init = s_even)
  when (io.in) {
    when (state === s_even) { state := s_odd  }
    .otherwise              { state := s_even }
  }
  io.out := (state === s_odd)
}
```



counters

```
class Counter(n: Int) {
  def value: UInt
  def inc(): Bool
}

Counter(n: Int)
Counter(cond: Bool, n: Int): (UInt, Bool)
```

delaying

```
ShiftRegister[T <: Data](in: T, n: Int, en: Bool = true.B): T
```

decoupled

- pipe
- queue
- arbiters

ValidIO

- A Bundle containing data and a signal determining if it is valid

```
class Valid[+T <: Data](gen: T) extends Bundle
{
  val valid = Output(Bool())
  val bits  = Output(gen.chiselCloneType)
  def fire(): Bool = valid
  override def cloneType: this.type = Valid(gen).asInstanceOf[this.type]
}

/** Adds a valid protocol to any interface */
object Valid {
  def apply[T <: Data](gen: T): Valid[T] = new Valid(gen)
}
```

Pipes

- delays data coming down pipeline by `latency` cycles
- similar to `ShiftRegister` but exposes Pipe interface
- requires a ValidIO interface

```
class Pipe[T <: Data](type: T, latency: Int = 1) extends Module
{
  class PipeIO extends Bundle {
    val enq = Input(Valid(gen))
    val deq = Output(Valid(gen))
  }
  val io = IO(new PipeIO)
  io.deq <> Pipe(io.enq, latency)
}
```

```
val pipe = new Pipe(UInt())
pipe.io.enq <> produce.io.out
consumer.io.in <> pipe.io.deq
```

ReadyValidIOs

```
abstract class ReadyValidIO[+T <: Data](gen: T) extends Bundle
{
  val ready = Input(Bool())
  val valid = Output(Bool())
  val bits  = Output(gen.chiselCloneType)
}
```

```
// Concrete class of ReadyValidIO
class DecoupledIO[+T <: Data](gen: T) extends ReadyValidIO[T](gen) { ... }
```

```
// Concrete class of ReadyValidIO
// Once ’valid’ is raised it won’t lower until after ’ready’ is raised
class IrrevocableIO[+T <: Data](gen: T) extends ReadyValidIO[T](gen) { ... }
```

Queues

- Required parameter `entries` controls depth
- The width is determined from the inputs.

```
class QueueIO[T <: Data](data: T, entries: Int) extends Bundle {
  val enq   = Flip(Decoupled(data.cloneType))
  val deq   = Decoupled(data.cloneType)
  val count = Output((log2Up(entries+1).W))
}

class Queue[T <: Data]
    (type: T, entries: Int,
     pipe: Boolean = false,
     flow: Boolean = false)
    extends Module
```

```
val q = Module(new Queue(UInt(), 16))
q.io.enq <> producer.io.out
consumer.io.in <> q.io.deq
```

Fixed Priority Arbiter

- sequences n producers into 1 consumer
- priority is given to lower producer

```
class ArbiterIO[T <: Data](data: T, n: Int) extends Bundle {
  val in     = Flip(Vec(Seq.fill(n) { Decoupled(data) }) )
  val out    = Decoupled( data.cloneType )
  val chosen = Output(Bits(log2Up(n).W))
}

class Arbiter[T <: Data](type: T, n: Int) extends Module
```

```
val arb = new Arbiter(UInt(), 2)
arb.io.in(0) <> producer0.io.out
arb.io.in(1) <> producer1.io.out
consumer.io.in <> arb.io.out
```

Round Robin Arbiter

- sequences n producers into 1 consumer
- producers are chosen in round robin order

```
class ArbiterIO[T <: Data](data: T, n: Int) extends Bundle {
  val in     = Flip(Vec(Seq.fill(n) { Decoupled(data) }) )
  val out    = Decoupled( data.cloneType )
  val chosen = Output(Bits(log2Up(n).W))
}

class RRArbiter[T <: Data](type: T, n: Int) extends Module
```

```
val arb = new RRArbiter(UInt(), 2)
arb.io.in(0) <> producer0.io.out
arb.io.in(1) <> producer1.io.out
consumer.io.in <> arb.io.out
```

advanced scripting

- functional programming: map, zip, fold
- datastructures: arrayBuffers maps and sets
- getwidth and widths

Scala Bindings

```scala
// constant
val x = 1
val (x, y) = (1, 2)

// variable
// generally don’t use var’s
var y = 2
y = 3
```

Scala Collections

```
// Array’s
val tbl = new Array[Int](256)
tbl(0) = 32
val y = tbl(0)
val n = tbl.length

// ArrayBuffer’s
import scala.collection.mutable.ArrayBuffer
val buf = new ArrayBuffer[Int]()
buf += 12
val z = buf(0)
val l = buf.length

// Seq’s
val els = Seq(1, 2, 3)
val els2 = x :: y :: y :: Nil
val a :: b :: c :: Nil = els
val m = els.length

// Tuple’s
val (x, y, z) = (1, 2, 3)
```

Scala Maps and Sets

```
import scala.collection.mutable.HashMap

val vars = new HashMap[String, Int]()
vars("a") = 1
vars("b") = 2
vars.size
vars.contains("c")
vars.getOrElse("c", -1)
vars.keys
vars.values
```

```
import scala.collection.mutable.HashSet

val keys = new HashSet[Int]()
keys += 1
keys += 5
keys.size -> 2
keys.contains(2) -> false
```

Scala Iteration

```
val tbl = new Array[Int](256)

// loop over all indices
for (i <- 0 until tbl.length)
  tbl(i) = i

// loop of each sequence element
val tbl2 = new ArrayBuffer[Int]
for (e <- tbl)
  tbl2 += 2*e

// loop over hashmap key / values
for ((x, y) <- vars)
  println("K " + x + " V " + y)
```

Scala Functions

```scala
// simple scaling function, e.g., x2(3) => 6
def x2 (x: Int) = 2 * x
```

```scala
// more complicated function with statements
def f (x: Int, y: Int) = {
  val xy = x + y;
  if (x < y) xy else -xy
}
```

Scala Functional

```
// simple scaling function, e.g., x2(3) => 6
def x2 (x: Int) = 2 * x
```

```
// produce list of 2 * elements, e.g., x2list(List(1, 2, 3)) => List(2, 4, 6)
def x2list (xs: List[Int]) = xs.map(x2)
```

```
// simple addition function, e.g., add(1, 2) => 3
def add (x: Int, y: Int) = x + y
```

```
// sum all elements using pairwise reduction, e.g., sum(List(1, 2, 3)) => 6
def sum (xs: List[Int]) = xs.foldLeft(0)(add)
```

Scala Object Oriented

```
class Blimp(r: Double) {
  val rad = r
  println("Another Blimp")
}

new Blimp(10.0)
```

```
class Zep(h: Boolean, r: Double) extends Blimp(r) {
  val isHydrogen = h
}

new Zep(true, 100.0)
```

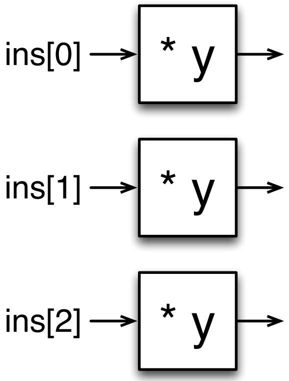
Scala Singleton Objects

- like Java class methods
- for top level methods

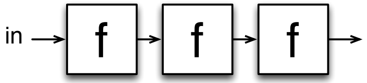
```
object Blimp {
  var numBlimps = 0
  def apply(r: Double) = {
    numBlimps += 1
    new Blimp(r)
  }
}

Blimp.numBlimps
Blimp(10.0)
```

Functional Composition

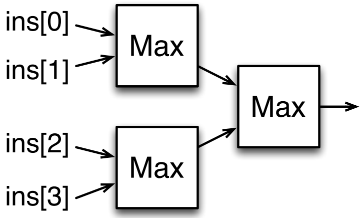
`Map(ins, x => x * y)`



`Chain(n, in, x => f(x))`



`Reduce(ins, Max)`



Functional Programming Review

	idiom	result
A	`(1,2,3) map { n => n + 1 }`	`(2,3,4)`
B	`(1,2,3) zip (a,b,c)`	`((1,a),(2,b),(3,c))`
C	`((1,a),(2,b),(3,c)) map { case (left,right) => left }`	`(1,2,3)`
D	`(1,2,3) foreach { n => print(n) }`	`123`
E	`for (x <- 1 to 3; y <- 1 to 3) yield (x,y)`	`(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)`

Generator

```
def delays[T <: Data](x: T, n: Int): List[T] =
  if (n <= 1) List(x) else x :: delay(Reg(next = x), n-1)

def FIR[T <: Data with Num[T]](hs: Iterable[T], x: T): T =
  (hs, delays(x, hs.length)).zipped.map( _ * _ ).reduce( _ + _ )

class TstFIR extends Module {
  val io = IO(new Bundle{ val x  = Input(SInt(8.W)); val y =
      Output(SInt(8.W)) })
  val h  = Array(1.S, 2.S, 4.S)
  io.y  := FIR(h, io.x)
}
```

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k]$$

getWidth

```
def getWidth(x: Data): Int
```

```
def PopCount(in: Bits): UInt =
  ((0 until in.getWidth).map(in(_))).foldLeft(0.U)(_ + _)
```

- returns width if available during Chisel elaboration
- can tie widths together by cloning type

vec uses

- creation: fill, tabulate
- functional: map, reduce, forall, etc
- bitvec, andR, orR, assignments

Vec Construction

```
object Vec {
  def apply[T <: Data](elts: Seq[T]): Vec[T]
  def apply[T <: Data](elt0: T, elts: T*): Vec[T]

  def tabulate[T <: Data](n: Int)(f: Int => T): Vec[T]
}
```

```
Vec(Seq.fill(3){ UInt(8.W) }) ====
  Vec(UInt(8.W), UInt(8.W), UInt(8.W))
Vec.tabulate(3){ _.U } ====
  Vec(0.U, 1.U, 2.U)
val v = Vec(Seq.fill(0){ UInt(8.W) })
for ...
  v += UInt(8.W)
```

Functional Vec

```
class Vec[T <: Data](val gen: () => T)
    extends Data with Cloneable with BufferProxy[T] {
  ...
  def forall(p: T => Bool): Bool
  def exists(p: T => Bool): Bool
  def contains(x: T): Bool
  def count(p: T => Bool): UInt

  def indexWhere(p: T => Bool): UInt
  def lastIndexWhere(p: T => Bool): UInt
}
```

```
Vec(K, L, M).contains(x) ==== ( x === K || x === L || x === M )
```

advanced mems

- mems vs vecs
- bitvec as uint

mems vs vecs

aggregate wires

```
val v1 = Vec(Seq.fill(n){ UInt(n.W) } )
```

all outputs of modules

```
val v2 = Vec(Seq.fill(n){ (new Core).io } )
```

separate init on each reg

```
val v3 = Vec.tabulate(n)( (i) => Reg(init = UInt(i)) )
```

all element access and map/reduce ops

```
val k = v2.indexWhere( (x) => x === 0.U )
```

bitvec as uint/bits

```
val useRAS = Reg(UInt(conf.entries.W))
...
useRAS(waddr) := update.bits.isReturn
...
val hits = tagMatch(io.req, pageHit)
...
val doPeek = Mux1H(hits, useRAS)
io.resp.valid := hits.orR
```

Useful methods

```
class ... Bits ... { ...
  def andR(): Bool
  def orR(): Bool
  def xorR(): Bool
  def apply(index: Int)
  def toBools: Vec[Bool]
  def bitSet(off: UInt, dat: UInt)
... }
object andR {
  def apply(x: Bits): Bool = x.andR
}
object orR {
  def apply(x: Bits): Bool = x.orR
}
object xorR {
  def apply(x: Bits): Bool = x.xorR
}
```

advanced types

- conversion
- clonining
- type parameterization
- defining your own types

type conversion

```
val myUInt      = inPacket.asUInt()
val outPacket = (new Packet).fromBits(myUInt) // likely to be deprecated soon
```

Cloning

- cloning is a shallow copy
- under the hood chisel data types cloned

```
val r1 = Reg(UInt(16.W))

val w  = UInt(16.W)
val r2 = Reg(w) // w cloned here
```

- when defining `Data` if use parameters need to clone

```
class Packet(n: Int, w: Int) extends Bundle {
  val address = UInt(Log2Up(n).W)
  val payload = UInt(w.W)
  override def cloneType: this.type =
    new Packet(n, w).asInstanceOf[this.type]
}
```

Parameterized Types in Scala

First we need to learn about parameterized types in Scala. We can define a generic `Mux` function as taking a boolean condition and `con` and `alt` arguments (corresponding to then and else expressions) of type `T` as follows:

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

where

- `T` is required to be a subclass of `Data` and
- the type of `con` and `alt` are required to match.

You can think of the type parameter as a way of just constraining the types of the allowable arguments.

defining your own types

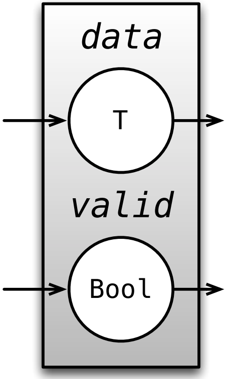
- num trait
- valid wrapper
- filter
- subclassing bundle – complex

num trait

```
abstract trait Num[T <: Data] {
  // def << (b: T): T;
  // def >> (b: T): T;
  def unary_-(): T;
  def +  (b: T): T;
  def *  (b: T): T;
  def /  (b: T): T;
  def %  (b: T): T;
  def -  (b: T): T;
  def <  (b: T): Bool;
  def <= (b: T): Bool;
  def >  (b: T): Bool;
  def >= (b: T): Bool;

  def min(b: T): T = Mux(this < b, this.asInstanceOf[T], b)
  def max(b: T): T = Mux(this < b, b, this.asInstanceOf[T])
}
```

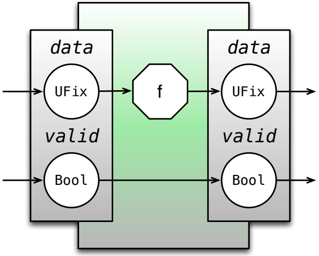
Valid Wrapper

```
class Valid[T <: Data](dtype: T) extends Bundle {
  val data  = dtype.cloneType
  val valid = Bool()
  override def cloneType = new Valid(dtype)
}

class GCD extends Module {
  val io = IO(new Bundle {
    val a   = Input(UInt(16.W))
    val b   = Input(UInt(16.W))
    val out = Valid(Output(UInt(16.W)))
  })
  ...
  io.out.data  := x
  io.out.valid := y === 0.U
}
```



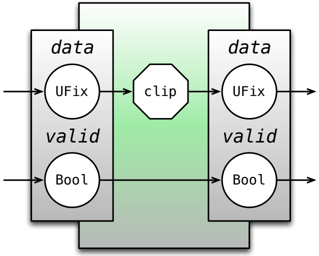
Function Filters

```
abstract class Filter[T <: Data](dtype: T) extends Module {
  val io = IO(new Bundle {
    val in  = Valid(dtype).asInput
    val out = Valid(dtype).asOutput
  })
}

class FunctionFilter[T <: Data](f: T => T, dtype: T) extends Filter(dtype) {
  io.out.valid := io.in.valid
  io.out       := f(io.in)
}
```



Clipping Filter

```
def clippingFilter[T <: Num](limit: Int, dtype: T) =
  new FunctionFilter(min(limit, max(-limit, _)), dtype)
```



complex

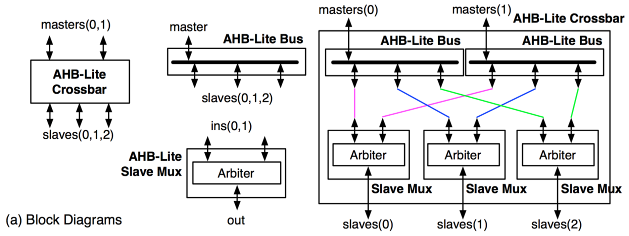
definition

```
class Complex[T <: Data](val re: T, val im: T) extends Bundle {
  override def cloneType: this.type =
    new Complex(re.cloneType, im.cloneType, dir).asInstanceOf[this.type]
  def + (other: T): T
  def - (other: T): T
  def * (other: T): T
  ...
}
```

defining

```
class ComplexMulAdd[T <: Bits](data: T) extends Module {
  val io = IO(new Bundle {
    val a = new Complex(data, data).asInput
    val b = new Complex(data, data).asInput
    val c = new Complex(data, data).asInput
    val o = new Complex(data, data).asOutput
  })
  io.o := io.a * io.b + io.c
}
```

AHB Example by Yunsup Lee



(a) Block Diagrams

```
classAHBXbar(nMasters:Int, nSlaves:Int) extends Module {
  val io = IO(new Bundle {
    val masters = Flip(Vec(new AHBMasterIO, nMasters))
    val slaves = Flip(Vec(new AHBSlaveIO, nSlaves))
  })
  val buses = List.fill(nMasters){Module(new AHBBus(nSlaves))}
  val muxes = List.fill(nSlaves){Module(new AHBSlaveMux(nMasters))}

  (buses.map(b => b.io.master) zip io.masters) foreach { case (b, m) => b <> m }
  (muxes.map(m => m.io.out) zip io.slaves ) foreach { case (x, s) => x <> s }
  for (m <- 0 until nMasters; s <- 0 until nSlaves) yield {
    buses(m).io.slaves(s) <> muxes(s).io.ins(m)
  } }
```

Rocket Examples by Andrew Waterman

- csr
- cam
- bht
- exceptions
- scoreboard

csr example

```
object CSRs {
  val fflags = 0x1
  ...
  val cycle = 0xc00
  ...
  val all = {
    val res = collection.mutable.ArrayBuffer[Int]()
    res += fflags
    ...
  }
}
val addr = Mux(cpu_req_valid, io.rw.addr, host_pcr_bits.addr | 0x500)
val decoded_addr = {
  // one hot encoding for each csr address
  val map = for ((v, i) <- CSRs.all.zipWithIndex)
    yield v -> UInt(BigInt(1) << i)
  val out = ROM(map)(addr)
  Map((CSRs.all zip out.toBools):_*)
}
val read_mapping = collection.mutable.LinkedHashMap[Int,Bits](
  CSRs.fflags -> (if (!conf.fpu.isEmpty) reg_fflags else 0.U),
  ...
  CSRs.cycle -> reg_time,
  ...
}
for (i <- 0 until reg_uarch_counters.size)
  read_mapping += (CSRs.uarch0 + i) -> reg_uarch_counters(i)

io.rw.rdata := Mux1H(for ((k, v) <- read_mapping) yield decoded_addr(k) -> v)
```

cam example

```
class CAMIO(entries: Int, addr_bits: Int, tag_bits: Int) extends Bundle {
  val clear       = Input(Bool())
  val clear_hit   = Input(Bool())
  val tag         = Input(Bits(tag_bits.W))
  val hit         = Output(Bool())
  val hits        = Output(UInt(entries.W))
  val valid_bits  = Output(Bits(entries.W))
  val write       = Input(Bool())
  val write_tag   = Input(Bits(tag_bits.W))
  val write_addr  = Input(UInt(addr_bits.W))
}
```

```
class RocketCAM(entries: Int, tag_bits: Int) extends Module {
  val addr_bits = ceil(log(entries)/log(2)).toInt
  val io = IO(new CAMIO(entries, addr_bits, tag_bits))
  val cam_tags = Mem(entries, Bits(tag_bits.W))
  val vb_array = Reg(init=0.U(entries.W))
  when (io.write) {
    vb_array := vb_array.bitSet(io.write_addr, true.B)
    cam_tags(io.write_addr) := io.write_tag
  }
  when (io.clear) {
    vb_array := 0.U(entries.W)
  } .elsewhen (io.clear_hit) {
    vb_array := vb_array & ~io.hits
  }
  val hits = (0 until entries).map(i => vb_array(i) && cam_tags(i) === io.tag) // <-- functional check
  io.valid_bits := vb_array
  io.hits := Vec(hits).asUInt // <-- turn into bit vector
  io.hit := io.hits.orR
}
```

bht example

abstract data type

```
class BHTResp(implicit conf: BTBConfig) extends Bundle {
  val index = UInt(log2Up(conf.nbht).max(1).W)
  val value = UInt(2.W)
}

class BHT(implicit conf: BTBConfig) {
  def get(addr: UInt): BHTResp = {
    val res = new BHTResp
    res.index := addr(log2Up(conf.nbht)+1,2) ^ history
    res.value := table(res.index)
    res
  }
  def update(d: BHTResp, taken: Bool): Unit = {
    table(d.index) := Cat(taken, (d.value(1) & d.value(0)) | ((d.value(1) | d.value(0)) & taken))
    history := Cat(taken, history(log2Up(conf.nbht)-1,1))
  }

  private val table = Mem(UInt(2.W), conf.nbht) // <-- private
  val history = Reg(UInt(log2Up(conf.nbht).W))
}
```

exceptions

```
val isLegalCSR = Vec.tabulate(1 << id_csr_addr.getWidth)(i => Bool(legal_csrs contains i))
```

```
val id_csr_wen = id_raddr1 =/= 0.U || !Vec(CSR.S, CSR.C).contains(id_csr)
```

```
  def checkExceptions(x: Iterable[(Bool, UInt)]) =
    (x.map(_._1).reduce(_||_), PriorityMux(x))        // <-- cool functional formulation

  val (id_xcpt, id_cause) = checkExceptions(List(
    (id_interrupt,                                    id_interrupt_cause),
    (io.imem.resp.bits.xcpt_ma,                       UInt(Causes.misaligned_fetch)),
    (io.imem.resp.bits.xcpt_if,                       UInt(Causes.fault_fetch)),
    (!id_int_val || id_csr_invalid,                   UInt(Causes.illegal_instruction)),
    (id_csr_privileged,                               UInt(Causes.privileged_instruction)),
    (id_sret && !io.dpath.status.s,                   UInt(Causes.privileged_instruction)),
    ((id_fp_val || id_csr_fp) && !io.dpath.status.ef, UInt(Causes.fp_disabled)),
    (id_syscall,                                      UInt(Causes.syscall)),
    (id_rocc_val && !io.dpath.status.er,              UInt(Causes.accelerator_disabled))))
```

scoreboard example

```
class Scoreboard(n: Int) {
  def set(en: Bool, addr: UInt): Unit   = update(en, _next |  mask(en, addr))
  def clear(en: Bool, addr: UInt): Unit = update(en, _next & ~mask(en, addr))
  def read(addr: UInt): Bool = r(addr)
  def readBypassed(addr: UInt): Bool = _next(addr)

  private val r = Reg(init=0.U(n.W))
  private var _next = r
  private var ens = false.B
  private def mask(en: Bool, addr: UInt) =
    Mux(en, 1.U << addr, 0.U)
  private def update(en: Bool, update: UInt) = {
    _next = update
    ens = ens || en
    when (ens) { r := _next }
  }
}
```

```
val sboard = new Scoreboard(32)
sboard.clear(io.dpath.ll_wen, io.dpath.ll_waddr)

val id_stall_fpu = if (!conf.fpu.isEmpty) {
  val fp_sboard = new Scoreboard(32)
  fp_sboard.set((wb_dcache_miss && wb_reg_fp_wen || io.fpu.sboard_set) && !replay_wb, io.dpath.wb_waddr)
}
```

Hwacha Examples by Yunsup Lee

- bundle operations
- instruction inheritance
- hardware objects

Bundle Operations

```
class VIUFn extends Bundle {
  val t0 = Bits(SZ_BMUXSEL.W)
  val t1 = Bits(SZ_BMUXSEL.W)
  val dw = Bits(SZ_DW.W)
  val fp = Bits(SZ_FP.W)
  val op = Bits(SZ_VIU_OP.W)

  def rtype() = t0 === ML
  def itype() = t0 === MR
  def rs1() = rtype() || itype()
  def rs2() = rtype()
  def wptr_sel(wptr0: Bits, wptr1: Bits, wptr2: Bits) =
    Mux(rtype(), wptr2, Mux(itype(), wptr1, wptr0)).toUInt
}
```

Instruction Inheritance

```
class DecodedInstruction extends Bundle {
  val utidx = UInt(SZ_VLEN.W)
  val fn = new Bundle {
    val viu = new VIUFn
    val vau0 = new VAU0Fn
    val vau1 = new VAU1Fn
    val vau2 = new VAU2Fn
    val vmu  = new VMULaneFn
  }
  val reg = new DecodedRegister
  val imm = new DecodedImmediate
}

class IssueOp extends DecodedInstruction {
  val vlen = UInt(SZ_VLEN.W)
  val active = new Bundle {
    val viu = Bool()
    val vau0 = Bool()
    ...
  }
  val sel = new Bundle {
    val vau1 = Bool()
    val vau2 = Bool()
  }
  ...
}
```

```
class VFU extends Bundle {
  val viu = Bool()
  val vau0 = Bool()
  val vault = Bool()
  val vaulf = Bool()
  ...
}
class SequencerEntry extends DecodedInstruction {
  val active = new VFU
}
class SequencerOp extends SequencerEntry {
  val cnt = Bits(SZ_BCNT.W)
  val last = Bool()
}
```

Sequencer: Object-Oriented Programming

```
class BuildSequencer[T <: Data](n: Int) {
  val valid = Vec(Seq.fill(entries){ Reg(init=false.B) })
  val stall = Vec(Seq.fill(entries){ Reg(init=false.B) })
  val vlen  = Vec(Seq.fill(n){ Reg(UInt(SZ_VLEN.W)) })
  val last  = Vec(Seq.fill(n){ Reg(Bool()) })
  val e     = Vec(Seq.fill(n){ Reg(new SequencerEntry) })
  val aiw   = Vec(Seq.fill(n){ Reg(new AIWUpdateEntry) })

  ...
  def islast(slot: UInt) = {
    val ret = Bool()
    ret := vlen(slot) <= io.cfg.bcnt
    when (turbo_capable(slot)) {
      when (io.cfg.prec == PREC_SINGLE) { ... }
      ...
    }
  }
  def next_addr_base(slot: UInt) =
    e(slot).imm.imm + (e(slot).imm.stride << 3.U)

  def vgu_val(slot: UInt) = valid(slot) && e(slot).active.vgu
  ...
  def active(slot: UInt) =
    alu_active(slot) || mem_active(slot)
  ...
}
```

Object Oriented Benefits

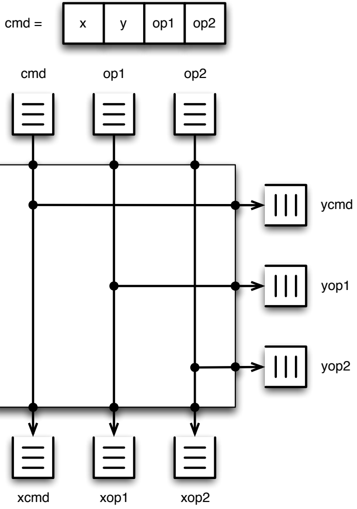
- The Scala compiler starts throwing type errors even before getting to chisel compiler
- Code gets easier to understand
  - less bugs when working in a team
  - self documenting
- no more underscores in IOs everything becomes a dot.
- modularizes code, gets rid of replication

Ready/Valid Signals

- must avoid defining ready/valid in terms of each other
- ends up being ever growing combinational expression
- can define `fire` function to do work

Complex Ready / Valid Example

```
cmd.ready =
  (!cmd.x | xcmd.ready) &
  (!cmd.y | ycmd.ready) &
  (!cmd.op1 | (op1.valid &
     (!cmd.x | xop1.ready) &
     (!cmd.y | yop1.ready))) &
  (!cmd.op2 | (op2.valid &
     (!cmd.x | xop2.ready) &
     (!cmd.y | yop2.ready)))
xcmd.valid =
  cmd.valid & cmd.x &
  (!cmd.op1 | (op1.valid & xop1.ready)) &
  (!cmd.op2 | (op2.valid & xop2.ready)) &
  (!cmd.y | (ycmd.ready &
     (!cmd.op1 | (op1.valid & yop1.ready)) &
     (!cmd.op2 | (op2.valid & yop2.ready))))
ycmd.valid = ...
```



Ready/Valid Signals

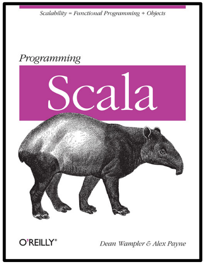
```
def fire(exclude: Bool, include: Bool*) = {
  val rvs = Array(
    !stall, io.vf.active,
    io.imem.resp.valid,
    mask_issue_ready,
    mask_deck_op_ready,
    mask_vmu_cmd_ready,
    mask_aiw_cntb_ready)
    rvs.filter(_ =/= exclude).reduce(_&&_) && (true.B :: include.toList).reduce(_&&_)
}

io.imem.resp.ready := fire(io.imem.resp.valid)
io.vcmdq.cnt.ready := fire(null, deq_vcmdq_cnt)
io.op.valid := fire(mask_issue_ready, issue_op)
io.deckop.valid := fire(mask_deck_op_ready, enq_deck_op)
io.vmu.issue.cmd.valid := fire(mask_vmu_cmd_ready, enq_vmu_cmd)
io.aiw.issue.enq.cntb.valid := fire(mask_aiw_cntb_ready, enq_aiw_cntb)
io.aiw.issue.marklast := fire(null, decode_stop)
io.aiw.issue.update.numcnt.valid := fire(null, issue_op)
io.vf.stop := fire(null, decode_stop)
...
```

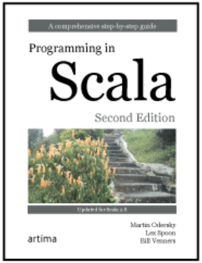
Chisel Resources and To Do

Existing Chisel Codebases

```
git@github.com:ucb-bar/rocket-chip.git
git@github.com:ucb-bar/riscv-sodor.git
git@github.com:ucb-bar/rocket.git
git@github.com:ucb-bar/uncore.git
git@github.com:ucb-bar/hwacha.git
git@github.com:ucb-bar/riscv-boom.git
```

Scala Resources


Scalability = Functional Programming + Objects
Programming
Scala
O'REILLY
Dean Wampler & Alex Payne



Keep In Touch

website	`chisel.eecs.berkeley.edu`
mailing list	`groups.google.com/group/chisel-users`
github	`https://github.com/ucb-bar/chisel3/`
wiki	`https://github.com/ucb-bar/chisel3/wiki`
features + bugs	`https://github.com/ucb-bar/chisel3/issues`
more questions	`stackoverflow.com/quesions/tagged/chisel`
jonathan	`jrb@eecs.berkeley.edu`
adam	`adamiz@eecs.berkeley.edu`