Advanced Chisel Topics

Jonathan Bachrach

EECS UC Berkeley

February 11, 2016

Today

- so you think you know chisel?

Today

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

Debugging

- assert, printf
- visualization with dot
- vcd dumps
- flags to control visibility

- 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 = UInt(0, 4))
x := x + UInt(1)
assert(x < UInt(10))
```

can be used in when statements:

```
val x = Bits(0x4142)
val s1 = sprintf("%x %s", x, x);
when (c) { printf("%d %s\n", x, s1); }
```

supported format specifiers:

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

VCD Debugging

- use `-vcd` arg to have simulation produce VCD output
- run your compiled C++ emulation app for a number of cycles
  - specifying the number of cycles as a first argument
- can view waveforms with
  - `vcs` – commercial
  - GTKWave – open source
- can hierarchically focus on particular signals
- can view in a variety of formats

Controlling Visibility

- only top level io and registers will be visible every cycle
- use `debug` call to selectively make them visible

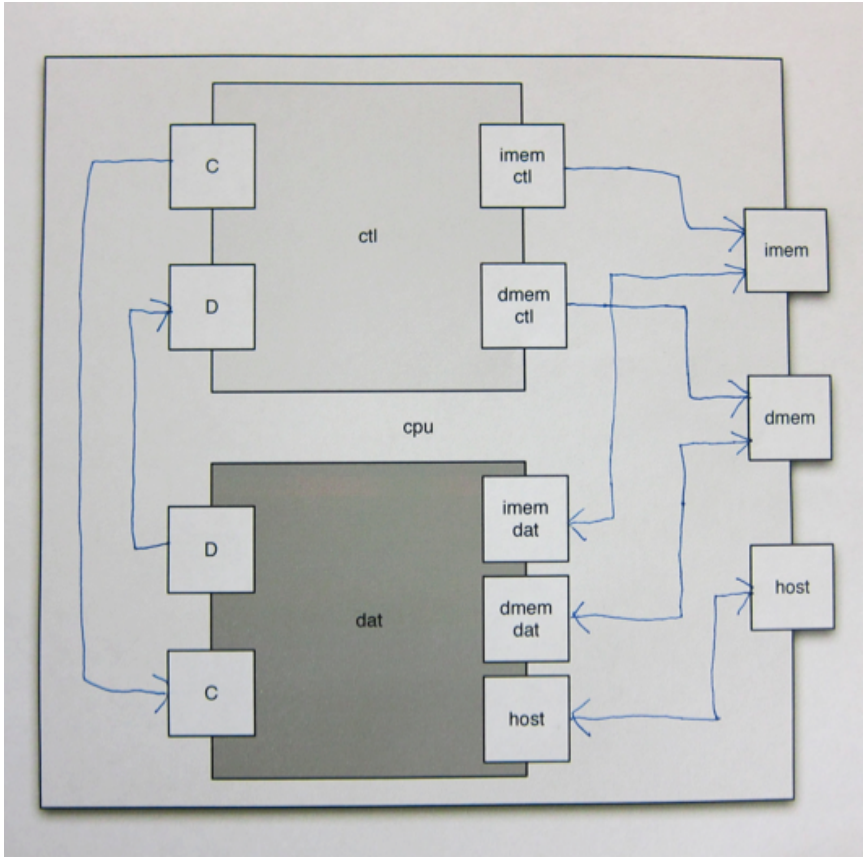
```
val x = debug(UInt(width = 32))
```

- use `-debug` arg to make all signals visible

- bulk connections
- black boxes

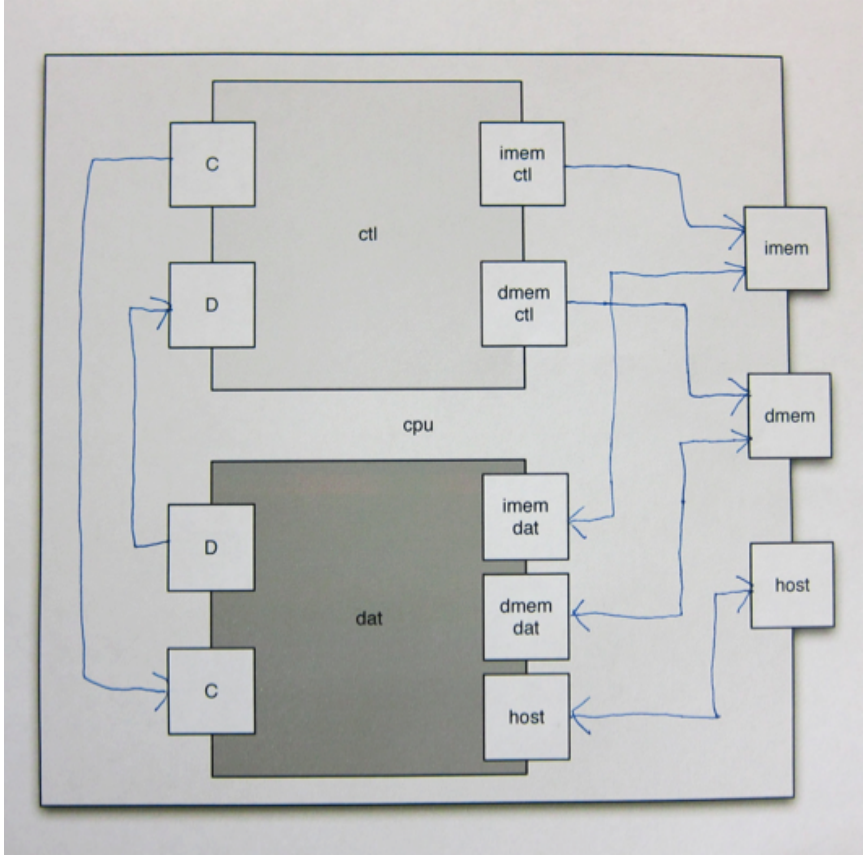
Interface Views


```
class Cpu extends Module {
  val io = new CpuIo()
  val c  = new CtlPath()
  val d  = new DatPath()
  c.io.ctl  <> d.io.ctl
  c.io.dat  <> d.io.dat
  c.io.imem <> io.imem
  d.io.imem <> io.imem
  c.io.dmem <> io.dmem
  d.io.dmem <> io.dmem
  d.io.host <> io.host
}
```



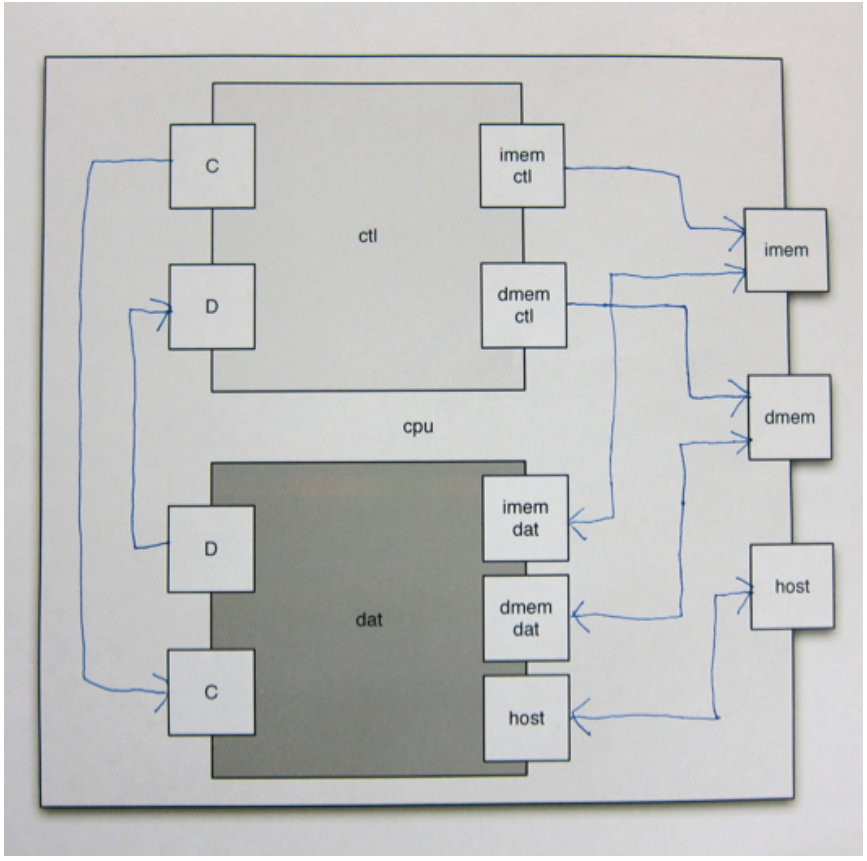
CPU Interfaces


```
class RomIo extends Bundle {
  val isVal = Bool(INPUT)
  val raddr = UInt(INPUT, 32)
  val rdata = Bits(OUTPUT, 32)
}

class RamIo extends RomIo {
  val isWr  = Bool(INPUT)
  val wdata = Bits(INPUT, 32)
}

class CpathIo extends Bundle {
  val imem = RomIo().flip()
  val dmem = RamIo().flip()
  ... }

class DpathIo extends Bundle {
  val imem = RomIo().flip()
  val dmem = RamIo().flip()
  ... }
```



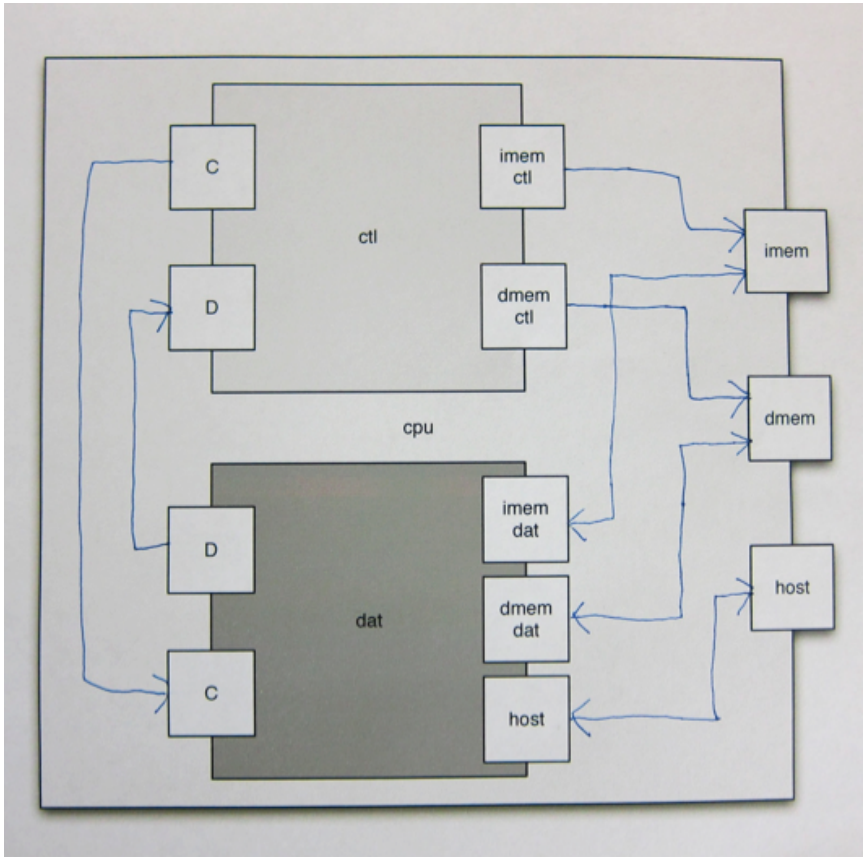
Partial Interface Fulfillment


```
class Cpath extends Module {
  val io = new CpathIo();
  ...
  io.imem.isVal := ...;
  io.dmem.isVal := ...;
  io.dmem.isWr  := ...;
  ...
}

class Dpath extends Module {
  val io = new DpathIo();
  ...
  io.imem.raddr := ...;
  io.dmem.raddr := ...;
  io.dmem.wdata := ...;
  ...
}
```



Multiple Partial Bulk Connections


```
class Cpu extends Module {
  val io = new CpuIo()
  val c  = new CtlPath()
  val d  = new DatPath()
  c.io.ctl  <> d.io.ctl
  c.io.dat  <> d.io.dat
  c.io.imem <> io.imem
  d.io.imem <> io.imem
  c.io.dmem <> io.dmem
  d.io.dmem <> io.dmem
  d.io.host <> io.host
}
```



black boxes

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

```
class RomIo extends Bundle {
  val isVal = Bool(INPUT)
  val raddr = UInt(INPUT,  32)
  val rdata = UInt(OUTPUT, 32)
}

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

- changing names using setName
- adding clocks ...

```
val io = new Bundle{
  val i = UInt(INPUT, 8); val o = UInt(INPUT, 8); }
io.i.setName("i")
io.o.setName("o")
```

- 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 = Bool(true)): 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] = ...
}
```

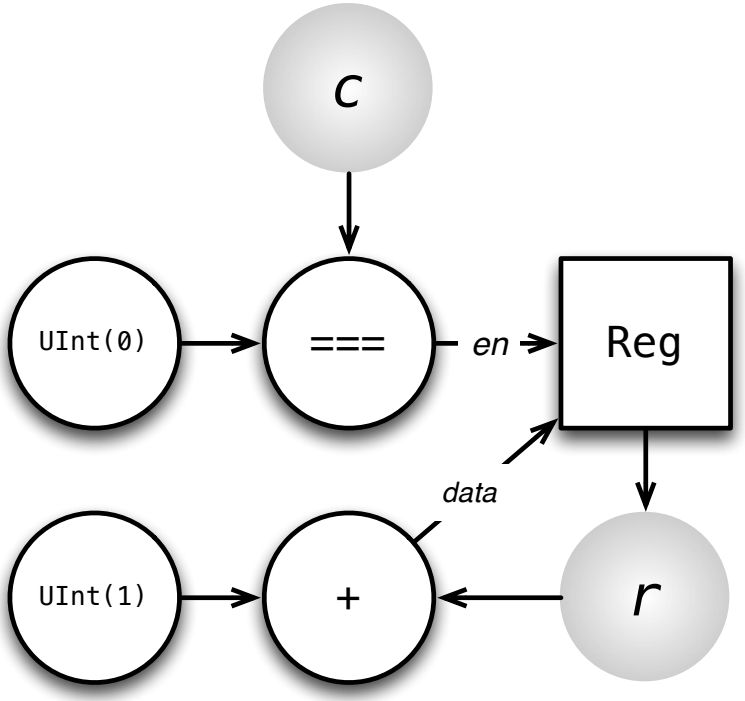
- conditional update rules
- state machines
- reg forms
- counters
- delaying
- examples

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(width = 16) )
when (c === UInt(0) ) {
  r := r + UInt(1)
}
```



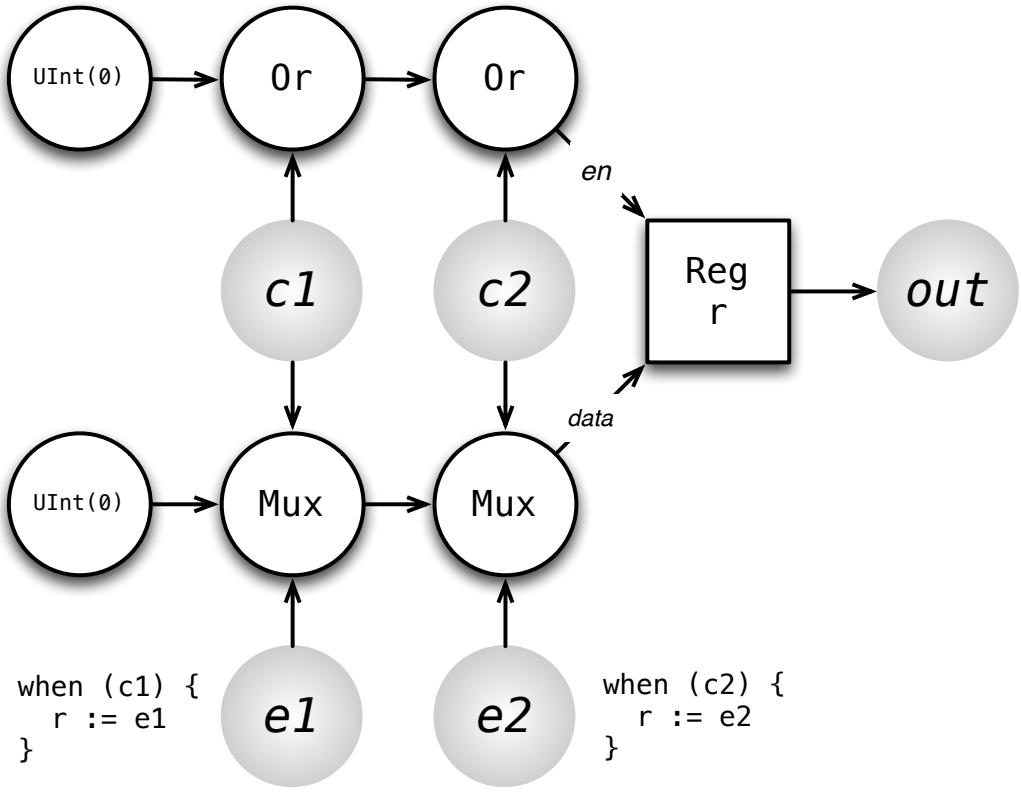
Conditional Updates Priority

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

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



- 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.

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

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

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

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:

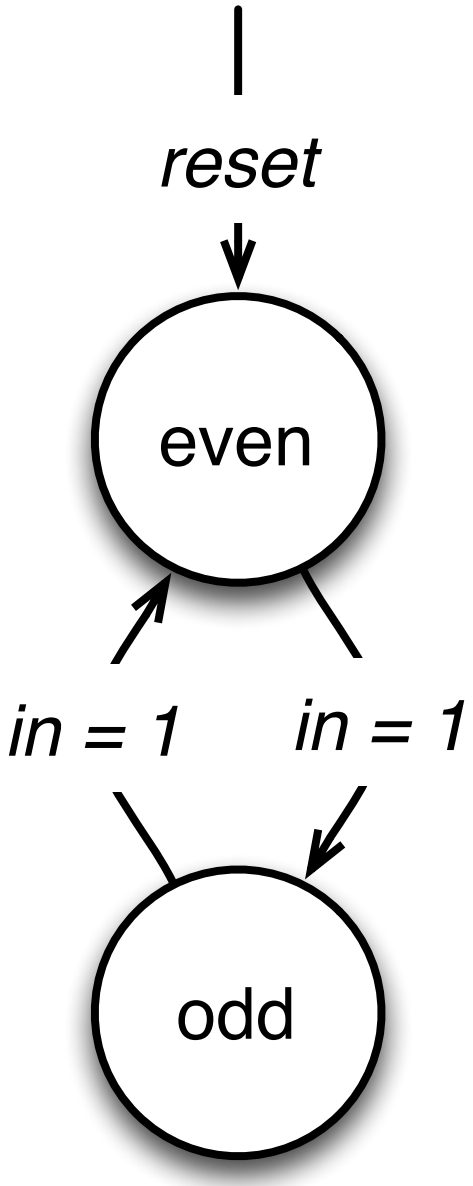
```
when (idx === v1) { u1 }
when (idx === v2) { u2 }
```

```
// create n enum values of given type
def Enum[T <: UInt]
  (n: Int)(type: => T): List[T]
// create enum values of given type and names
def Enum[T <: UInt]
  (l: Symbol *)(type: => T): Map[Symbol, T]
// create enum values of given type and names
def Enum[T <: UInt]
  (l: List[Symbol])(type: => T): Map[Symbol, T]
```

```
val s_even :: s_odd :: Nil = Enum(UInt(), 2)
```

Finite state machines can now be readily defined as follows:

```
class Parity extends Module {
  val io = new Bundle {
    val in  = Bool(INPUT)
    val out = Bool(OUTPUT) }
  val s_even :: s_odd :: Nil = Enum(UInt(), 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)
}
```



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

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

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

```
Latch[T <: Data](in: T, en: Bool = Bool(true)): T
```

```
Delays[T <: Data](in: T, n: Int, en: Bool = Bool(true)): Vec[T]
```

- pipe
- queue
- arbiters

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

```
class ValidIO[+T <: Data](data: T) extends Bundle {
  val valid = Bool(OUTPUT)
  val bits  = data.clone.asOutput
}

class Pipe[T <: Data](type: T, latency: Int = 1) extends Module
```

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

- adds a ready-valid handshaking protocol to any interface

```
class DecoupledIO[+T <: Data](gen: T) extends Bundle {
  val ready = Bool(INPUT)
  val valid = Bool(OUTPUT)
  val bits  = gen.clone.asOutput
  def fire(dummy: Int = 0): Bool = ready && valid
  ...
}

object Decoupled {
  def apply[T <: Data](gen: T): DecoupledIO[T] = new DecoupledIO(gen)
}
```

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

```
class QueueIO[T <: Data](data: T, entries: Int) extends Bundle {
  val enq   = Decoupled(data.clone).flip
  val deq   = Decoupled(data.clone)
  val count = UInt(OUTPUT, log2Up(entries+1))
}

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

```
val q = 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     = Vec.fill(n) { Decoupled(data) }.flip
  val out    = Decoupled( data.clone )
  val chosen = Bits(OUTPUT, log2Up(n))
}

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     = Vec.fill(n) { Decoupled(data) }.flip
  val out    = Decoupled( data.clone )
  val chosen = Bits(OUTPUT, log2Up(n))
}

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
```

- 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
var y = 2
y = 3
```

```
// 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

// List's
val els = List(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)
```

```
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)
```

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

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

```
// 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)
```

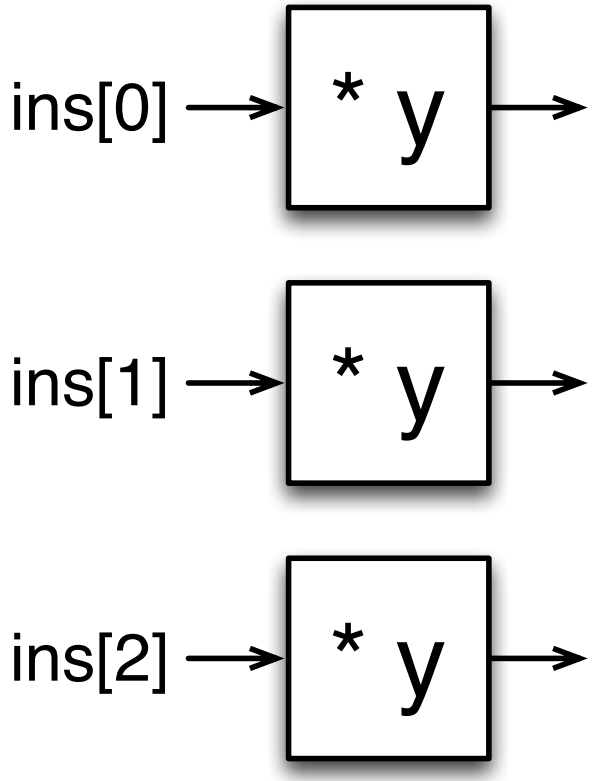
```
class Blimp(r: Double) {
  val rad = r
  println("Another Blimp")
}

new Blimp(10.0)
```

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

new Zep(true, 100.0)
```

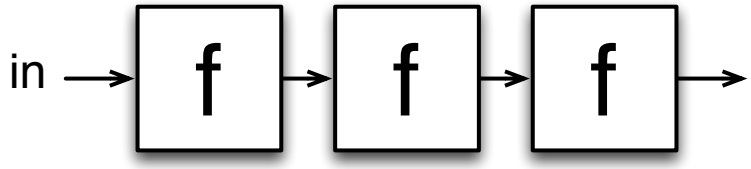
- like Java class methods
- for top level methods

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

Blimp.numBlimps
Blimp(10.0)
```

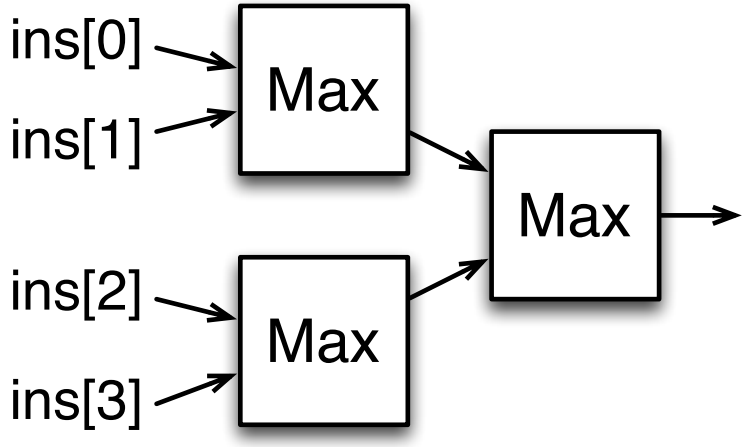
```
Map(ins, x => x * y)
```




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




```
Reduce(ins, Max)
```



	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)`

```
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 = new Bundle{ val x  = SInt(INPUT, 8); val y = SInt(OUTPUT, 8) }
  val h  = Array(SInt(1), SInt(2), SInt(4))
  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(UInt(0))(_ + _)
```

- returns width if available during construction time
- can tie widths together by cloning type

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

Vec Construction


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

  def fill[T <: Data](n: Int)(f: => T): Vec[T]
  def tabulate[T <: Data](n: Int)(f: Int => T): Vec[T]
  def tabulate[T <: Data](n1: Int, n2: Int)(f: (Int, Int) => T): Vec[Vec[T]]
}
```

```
Vec(A, L, M)
Vec.fill(3){ UInt(width = 8) } ====
  Vec(UInt(width = 8), UInt(width = 8), UInt(width = 8))
Vec.tabulate(3){ UInt(_) } ====
  Vec(UInt(0), UInt(1), UInt(2))
val v = Vec.fill(0){ UInt(width = 8) }
for ...
  v += UInt(width = 8)
```

```
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 )
```

- mems vs vecs
- bitvec as uint

aggregate wires

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

all outputs of modules

```
val v2 = Vec.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 === UInt(0) )
```

```
val useRAS = Reg(UInt(width = conf.entries))
...
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
}
```

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

```
val bits      = inPacket.toBits()
val outPacket = (new Packet).fromBits(bits)
```

Cloning

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

```
val r1 = Reg(UInt(width = 16))

val w  = UInt(width = 16)
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(width = Log2Up(n))
  val payload = UInt(width = w)
  override def clone: this.type =
    new Packet(n, w).asInstanceOf[this.type]
}
```

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.

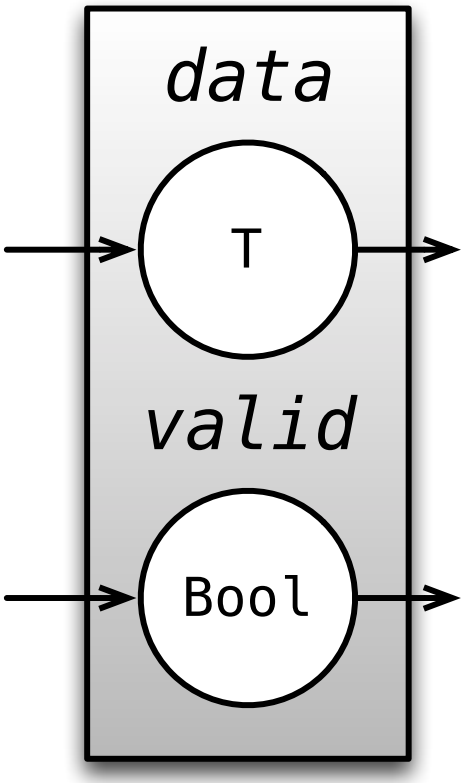
You can think of the type parameter as a way of just constraining the types of the allowable arguments.

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

```
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])
}
```

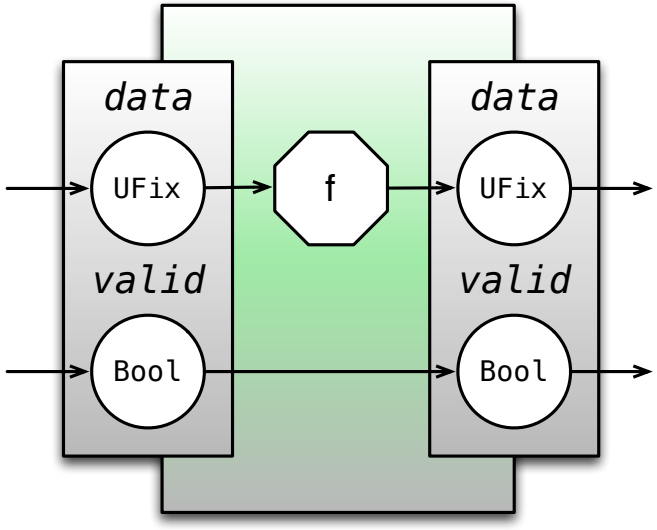
Valid Wrapper


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

class GCD extends Module {
  val io = new Bundle {
    val a   = UInt(INPUT, 16)
    val b   = UInt(INPUT, 16)
    val out = new Valid(UInt(OUTPUT, 16))
  } }
  ...
  io.out.data  := x
  io.out.valid := y === UInt(0)
}
```



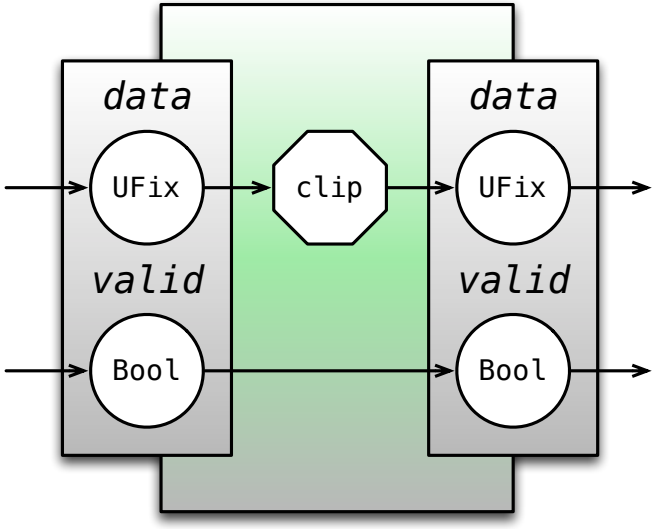
Function Filters


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



- sint
- sfix/ufix
- flo/dbl
- complex
- examples

```
class UInt extends Bits with Num[SInt] ...
  def zext(): SInt = Cat(UInt(0,1), this).toSInt
  def + (b: SInt): SInt = b + this
  def * (b: SInt): SInt = b * this
  def - (b: SInt): SInt = this.zext - b
  def / (b: SInt): SInt = this.zext / b
  def % (b: SInt): SInt = this.zext % b
```

```
class SInt extends Bits with Num[SInt] ...
  def !=  (b: UInt): Bool = this != b.zext
  def >   (b: UInt): Bool = this > b.zext
  def <   (b: UInt): Bool = this < b.zext
  def >=  (b: UInt): Bool = this >= b.zext
  def <=  (b: UInt): Bool = this <= b.zext
  //SInt to UInt arithmetic
  def * (b: UInt): SInt = newBinaryOp(b.zext, "s*u")
  def + (b: UInt): SInt = this + b.zext
  def - (b: UInt): SInt = this - b.zext
  def / (b: UInt): SInt = this / b.zext
  def % (b: UInt): SInt = this % b.zext
  def abs: UInt = Mux(this < SInt(0), (-this).toUInt, this.toUInt)
```

```
  def toSInt(): SInt = chiselCast(this){SInt()};
  def toUInt(): UInt = chiselCast(this){UInt()};
```

```
class UFix(val exp: Int, val raw: UInt) extends Bundle with Num[UFix] ...
  def <<(b: Int): UFix
  def >>(b: Int): UFix
  def + (b: UFix): UFix
  def * (b: UFix): UFix
  ...
}
object UFix {
  def apply(exp: Int, width: Int): UFix
}
class SFix(val exp: Int, val raw: SInt) extends Bundle with Num[SFix] ...
  def <<(b: Int): SFix
  def >>(b: Int): SFix
  def + (b: SFix): SFix
  def - (b: SFix): SFix
  def * (b: SFix): SFix
  def unary_-(): SFix
}
object SFix {
  def apply(exp: Int, width: Int): SFix
}
```

SFix Example


```
class Toy extends Module {
  val io = new Bundle {
    val in0 = SFix(2, 4).asInput
    val in1 = SFix(2, 4).asInput

    val out = SFix(4,16).asOutput
    val oraw = Bits(OUTPUT, width=128)
  }

  val int_result = -io.in0 * (io.in0 + io.in1)

  io.out := int_result
  io.oraw := int_result.raw
}
```

```
class Flo extends Bits with Num[Flo] ...
class Dbl extends Bits with Num[Dbl] ...
```

music example

```
object SinWave {
  def apply(f: Dbl): Dbl = {
    val phase = Reg(init = Dbl(0.0));
    phase := phase + (f * TwoPi() / SampsPerSec());
    Sin(phase)
  }
}

object RingMod {
  def apply(carrier: Dbl, fm: Dbl, index: Dbl): Dbl = {
    ((Dbl(1)-index) * carrier) + (index * carrier * SinWave(fm));
  }
}
```

- simulation – backends C++ and Verilog to come

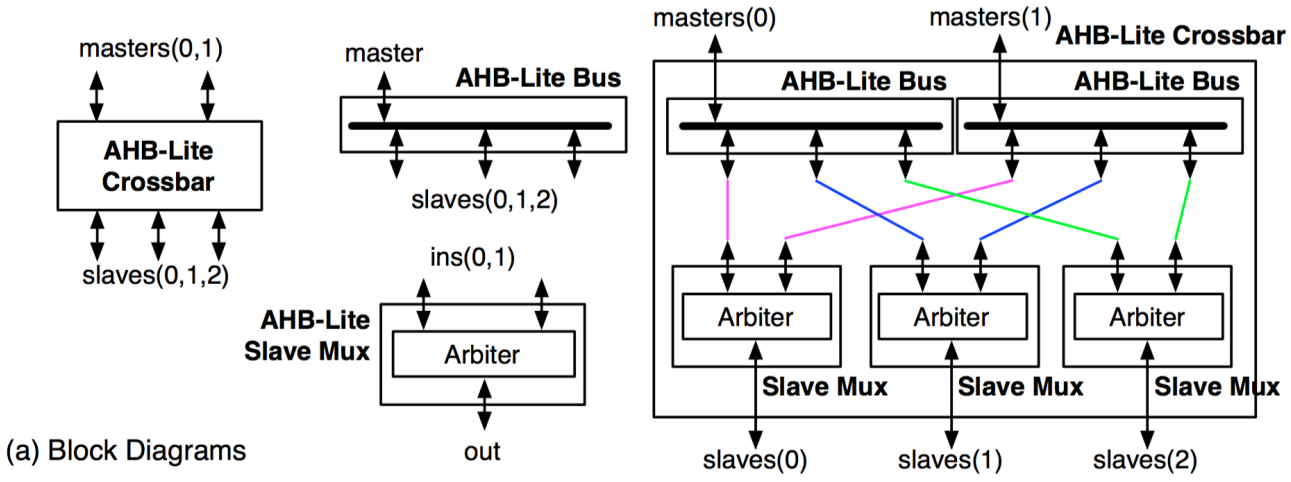
definition

```
class Complex[T <: Data](val re: T, val im: T) extends Bundle {
  override def cloneType: this.type =
    new Complex(re.clone, im.clone, 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 = 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




```
classAHBXbar(nMasters:Int, nSlaves:Int) extends Module {
  val io = new Bundle {
    val masters = Vec(new AHBMasterIO, nMasters).flip
    val slaves = Vec(new AHBSlaveIO, nSlaves).flip
  }
  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)
  } }
```

- csr
- bht
- cam
- complex
- 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 UInt(0)),
  ...
  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       = Bool(INPUT)
  val clear_hit   = Bool(INPUT)
  val tag         = Bits(INPUT, tag_bits)
  val hit         = Bool(OUTPUT)
  val hits        = UInt(OUTPUT, entries)
  val valid_bits  = Bits(OUTPUT, entries)
  val write       = Bool(INPUT)
  val write_tag   = Bits(INPUT, tag_bits)
  val write_addr  = UInt(INPUT, addr_bits)
}
```

```
class RocketCAM(entries: Int, tag_bits: Int) extends Module {
  val addr_bits = ceil(log(entries)/log(2)).toInt
  val io = new CAMIO(entries, addr_bits, tag_bits)
  val cam_tags = Mem(Bits(width = tag_bits), entries)
  val vb_array = Reg(init=Bits(0, entries))
  when (io.write) {
    vb_array := vb_array.bitSet(io.write_addr, Bool(true))
    cam_tags(io.write_addr) := io.write_tag
  }
  when (io.clear) {
    vb_array := Bits(0, entries)
  } .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).toBits // <-- turn into bit vector
  io.hit := io.hits.orR
}
```

abstract data type

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

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

exceptions


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

```
val id_csr_wen = id_raddr1 != UInt(0) || !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=Bits(0, n))
  private var _next = r
  private var ens = Bool(false)
  private def mask(en: Bool, addr: UInt) =
    Mux(en, UInt(1) << addr, UInt(0))
  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)
}
```

- bundle operations
- instruction inheritance
- hardware objects

Bundle Operations


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

  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(width = SZ_VLEN)
  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(width = SZ_VLEN)
  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(width = SZ_BCNT)
  val last = Bool()
}
```

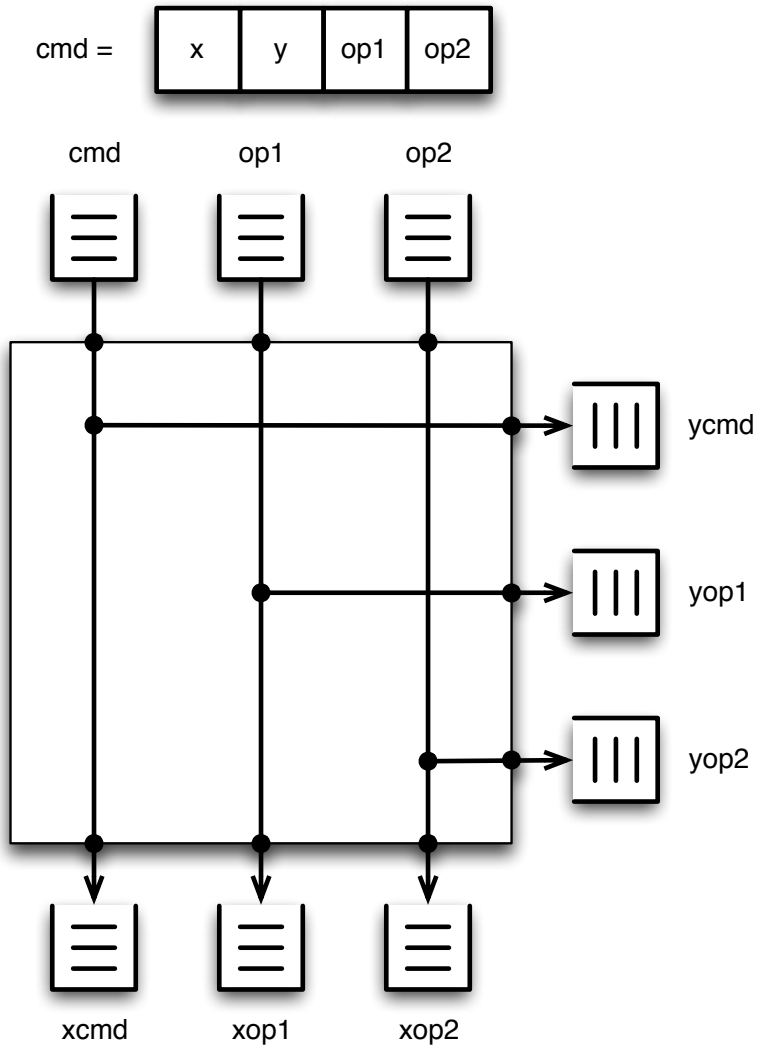
```
class BuildSequencer[T <: Data](n: Int) {
  val valid = Vec.fill(entries){ Reg(init=Bool(false)) }
  val stall = Vec.fill(entries){ Reg(init=Bool(false)) }
  val vlen  = Vec.fill(n){ Reg(UInt(width=SZ_VLEN)) }
  val last  = Vec.fill(n){ Reg(Bool()) }
  val e     = Vec.fill(n){ Reg(new SequencerEntry) }
  val aiw   = Vec.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 << UInt(3))

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

- 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

- 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(_&&_) && (Bool(true) :: 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)
...
```

Functional Memory

- current memories create port for every read/write
- suppose you want only n ports?

introducing funmem

```
val isValidO0 = Module(new FunMem(Bool(), n, 1, 6))
...
isValidI2.write(addr, value, idx)
isValidI0.read(io.dat.i.bits.dst.rid, 0)
```

Functional Memory Interface


```
class RdIO[T <: Data](data: T, depth: Int) extends Bundle {
  val adr = Bits( INPUT,  log2Up(depth) );
  val dat = data.clone.asOutput;
}

class WrIO[T <: Data](data: T, depth: Int) extends Bundle {
  val is  = Bool( INPUT );
  val adr = Bits( INPUT, log2Up(depth) );
  val dat = data.clone.asInput;
}

class FunRdIO[T <: Data](data: T, depth: Int) extends RdIO(data, depth) {
  adr := Bits(0);
  def read(nadr: Bits): T = {
    adr := nadr
    dat
  }
  override def clone = { new FunRdIO(data, depth).asInstanceOf[this.type]; }
}

class FunWrIO[T <: Data](data: T, depth: Int) extends WrIO(data, depth) {
  is  := Bool(false)
  adr := Bits(0)
  dat := data.fromBits(Bits(0))
  def write(nadr: Bits, ndat: T) = {
    is  := Module.current.whenCond // <<< INTERNAL THAT READ CURRENT CONDITION
    adr := nadr
    dat := ndat
  }
  override def clone = { new FunWrIO(data, depth).asInstanceOf[this.type]; }
}
```

Functional Memory Definition


```
class FunMemIO[T <: Data](data: T, depth: Int, numReads: Int, numWrites: Int) extends Bundle {
  val reads  = Vec.fill(numReads){ new FunRdIO(data, depth) }
  val writes = Vec.fill(numWrites){ new FunWrIO(data, depth) }
  val direct = Vec.fill(depth)( data.clone.asOutput )
}

class FunStore[T <: Data](data: T, val depth: Int, numReads: Int, numWrites: Int) extends Module {
  val io = new FunMemIO(data, depth, numReads, numWrites)
}

class FunMem[T <: Data](data: T, val depth: Int, numReads: Int, numWrites: Int) extends Module {
  val io = new FunMemIO(data, depth, numReads, numWrites)
  val mem = Mem(data, depth)
  def read(addr: UInt, idx: Int = 0): T = io.reads(idx).read(addr)
  def write(addr: UInt, data: T, idx: Int = 0) = io.writes(idx).write(addr, data)
  for (read <- io.reads)
    read.dat := mem(read.adr)
  for (write <- io.writes)
    when (write.is) { mem(write.adr) := write.dat }
  for (i <- 0 until depth)
    io.direct(i) := mem(i)
  def read(addr: Int): T = io.direct(addr)
}
```

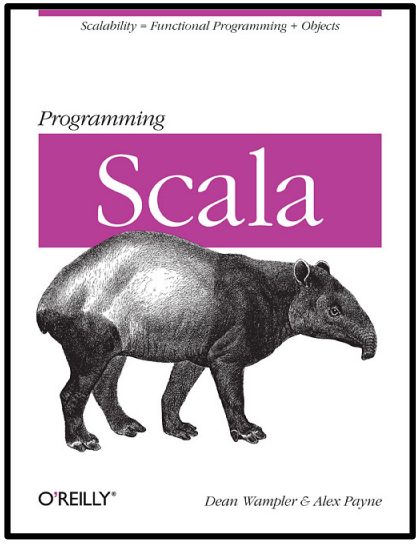
Resources

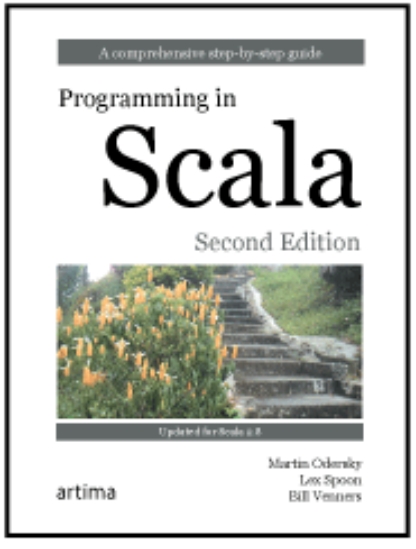
```
git@github.com:ucb-bar/riscv-sodor.git
  git@github.com:ucb-bar/rocket.git
  git@github.com:ucb-bar/uncore.git
  git@github.com:ucb-bar/hwacha.git
git@github.com:ucb-bar/riscv-boom.git
```

Todo

- decoupled – another lecture
- parameters – `git@github.com:ucb-bar/context-dependent-environments.git`
- barcrawl – `git@github.com:ucb-bar/bar-crawl.git`

Scala Resources





Keep in Touch

website	`chisel.eecs.berkeley.edu`
mailing list	`groups.google.com/group/chisel-users`
github	`https://github.com/ucb-bar/chisel/`
features + bugs	`https://github.com/ucb-bar/chisel/issues`
more questions	`stackoverflow.com/quesions/tagged/chisel`
twitter	`#chiselhdl`
me	`jrb@eecs.berkeley.edu`