

Chisel Bootcamp

Jonathan Bachrach

EECS UC Berkeley

June 8, 2012

Logging into EC2

```
ssh xxx@yyy  
password: bootcamp
```

Using Screen

to prevent lossage of state if disconnected ... when you first log in, type

```
screen
```

when you log back into the instance, type

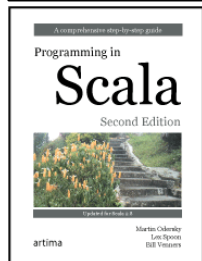
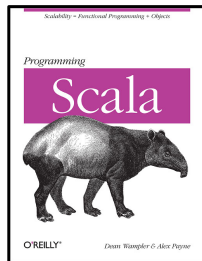
```
screen -r
```

Getting the Latest

```
cd chisel  
git pull
```

The Scala Programming Language

- Compiled to JVM
 - Good performance
 - Great Java interoperability
 - Mature debugging, execution environments
- Object Oriented
 - Factory Objects, Classes
 - Traits, overloading etc
- Functional
 - Higher order functions
 - Anonymous functions
 - Currying etc
- Extensible
 - Domain Specific Languages (DSLs)



Scala Functional

```
def f (x: Int) = 2 * x
```

```
def g (xs: List[Int]) = xs.map(f)
```

Scala Object Oriented

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

```
Blimp.numBlimps  
Blimp(10.0)
```

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

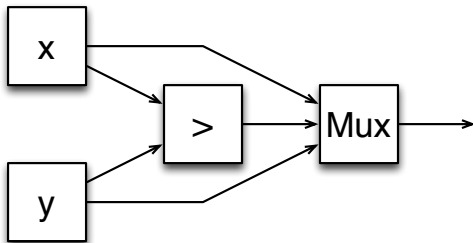
```
class Zep(r: Double) extends Blimp(r)
```

Scala Console

```
scala  
1 + 2  
def f (x: Int) = 2 * x  
f(4)
```

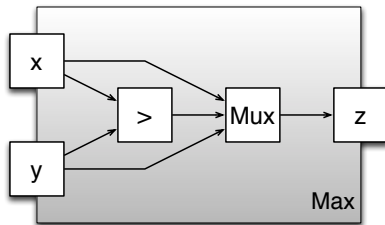

Algebraic Graph Construction

$\text{Mux}(x > y, x, y)$



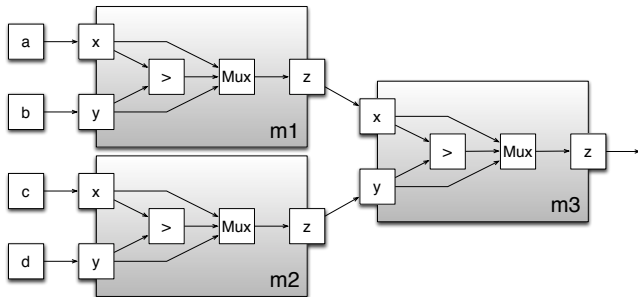
Creating Component

```
class Max2 extends Component {  
  val io = new Bundle {  
    val x = UFix(width = 8).asInput  
    val y = UFix(width = 8).asInput  
    val z = UFix(width = 8).asOutput }  
  io.z := Mux(io.x > io.y, io.x, io.y)  
}
```



Connecting Components

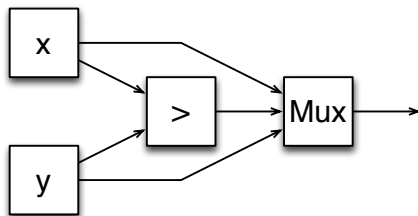
```
val m1 = new Max2()  
m1.io.x := a  
m1.io.y := b  
val m2 = new Max2()  
m2.io.x := c  
m2.io.y := d  
val m3 = new Max2()  
m3.io.x := m1.io.z  
m3.io.y := m2.io.z
```



Defining Construction Functions

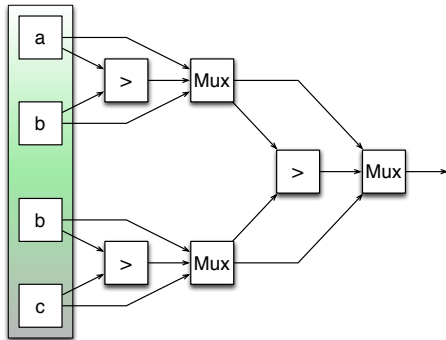
```
def Max2 = Mux(x > y, x, y)
```

```
Max2(x, y)
```



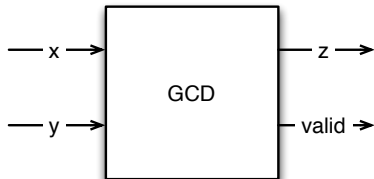
Functional Construction

Reduce(Array(a, b, c, d), Max2)



Example

```
class GCD extends Component {  
  val io = new Bundle {  
    val a      = UFix(16, INPUT)  
    val b      = UFix(16, INPUT)  
    val z      = UFix(16, OUTPUT)  
    val valid  = Bool(OUTPUT) }  
  val x = Reg(resetVal = io.a)  
  val y = Reg(resetVal = io.b)  
  when (x > y) {  
    x := x - y  
  } .otherwise {  
    y := y - x  
  }  
  io.z      := x  
  io.valid := y === UFix(0)  
}
```



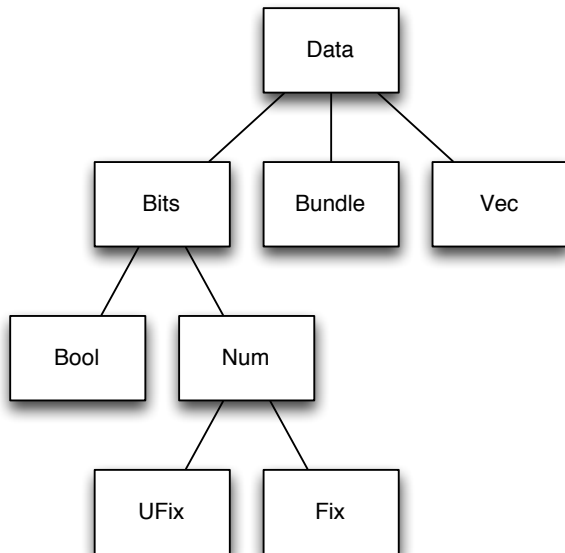
Literals

```
Bits(1)           // decimal 1-bit literal from Scala Int.  
Bits("ha")       // hexadecimal 4-bit literal from string.  
Bits("o12")      // octal 4-bit literal from string.  
Bits("b1010")    // binary 4-bit literal from string.  
  
Fix(5)           // signed decimal 4-bit literal from Scala Int.  
Fix(-8)          // negative decimal 4-bit literal from Scala Int.  
UFix(5)          // unsigned decimal 3-bit literal from Scala Int.  
  
Bool(true)       // Bool literals from Scala literals.  
Bool(false)
```

Literals

```
Bits("h_dead_beef") // 32-bit literal of type Bits.  
Bits(1)              // decimal 1-bit literal from Scala Int.  
Bits("ha", 8)        // hexadecimal 8-bit literal of type Bits.  
Bits("o12", 6)       // octal 6-bit literal of type Bits.  
Bits("b1010", 12)    // binary 12-bit literal of type Bits.  
  
Fix(5, 7)            // signed decimal 7-bit literal of type Fix.  
UFix(5, 8)           // unsigned decimal 8-bit literal of type UFix.
```


Type Hierarchy

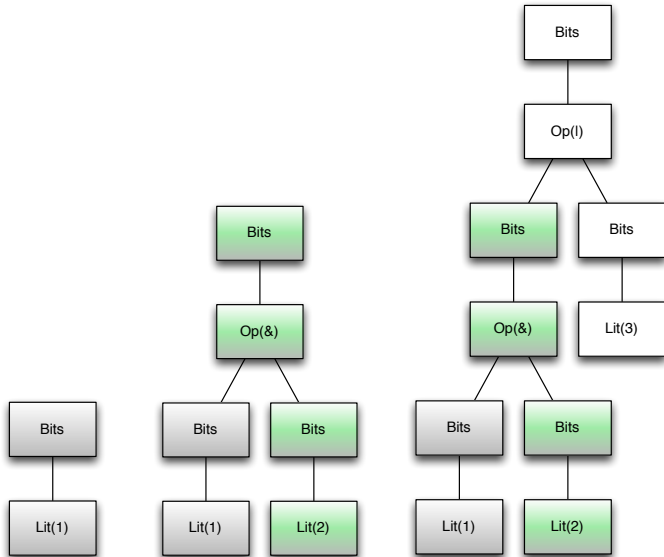


Combinational Circuits

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

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

Building Graphs



`a = Bits(1)`

`b = a & Bits(2)`

`b | Bits(3)`

Bitwise operators

Valid on Bits, Fix, UFix, Bool.

```
// Bitwise-NOT
val invertedX = ~x
// Bitwise-AND
val hiBits    = x & Bits("h_ffff_0000")
// Bitwise-OR
val flagsOut  = flagsIn | overflow
// Bitwise-XOR
val flagsOut  = flagsIn ^ toggle
```

Bitwise reductions

Valid on Bits, Fix, and UFix. Returns Bool.

```
// AND-reduction
val allSet = andR(x)
// OR-reduction
val anySet = orR(x)
// XOR-reduction
val parity = xorR(x)
```

Equality comparison

Valid on Bits, Fix, UFix, and Bool. Returns Bool.

```
// Equality
val equ = x === y
// Inequality
val neq = x != y
```

Valid on Bits, Fix, and UFix.

```
// Logical left shift.  
val twoToTheX = Fix(1) << x  
// Right shift (logical on Bits & UFix, arithmetic on Fix).  
val hiBits    = x >> UFix(16)
```

Bitfield manipulation

Valid on Bits, Fix, UFix, and Bool.

```
// Extract single bit, LSB has index 0.  
val xLSB      = x(0)  
// Extract bit field  from end to start bit pos.  
val xTopNibble = x(15,12)  
// Replicate a bit string multiple times.  
val usDebt     = Fill(3, Bits("hA"))  
// Concatenates bit fields, w/ first arg on left  
val float      = Cat(sgn,exp,man)
```


Logical Operations

Valid on Bools.

```
// Logical NOT.  
val sleep = !busy  
// Logical AND.  
val hit    = tagMatch && valid  
// Logical OR.  
val stall  = src1busy || src2busy  
// Two-input mux where sel is a Bool.  
val out    = Mux(sel, inTrue, inFalse)
```

Arithmetic operations

Valid on Nums: Fix and UFix.

```
// Addition.  
val sum = a + b  
// Subtraction.  
val diff = a - b  
// Multiplication.  
val prod = a * b  
// Division.  
val div = a / b  
// Modulus  
val mod = a % b
```

Arithmetic comparisons

Valid on Nums: Fix and UFix. Returns Bool.

```
// Greater than.  
val gt  = a > b  
// Greater than or equal.  
val gte = a >= b  
// Less than.  
val lt  = a < b  
// Less than or equal.  
val lte = a <= b
```

Bitwidth Inference

operation

bit width

$z = x + y$

$wz = \max(wx, wy)$

$z = x - y$

$wz = \max(wx, wy)$

$z = x \& y$

$wz = \max(wx, wy)$

$z = \text{Mux}(c, x, y)$

$wz = \max(wx, wy)$

$z = w * y$

$wz = wx + wy$

$z = x \ll n$

$wz = wx + \text{maxNum}(n)$

$z = x \gg n$

$wz = wx - \text{minNum}(n)$

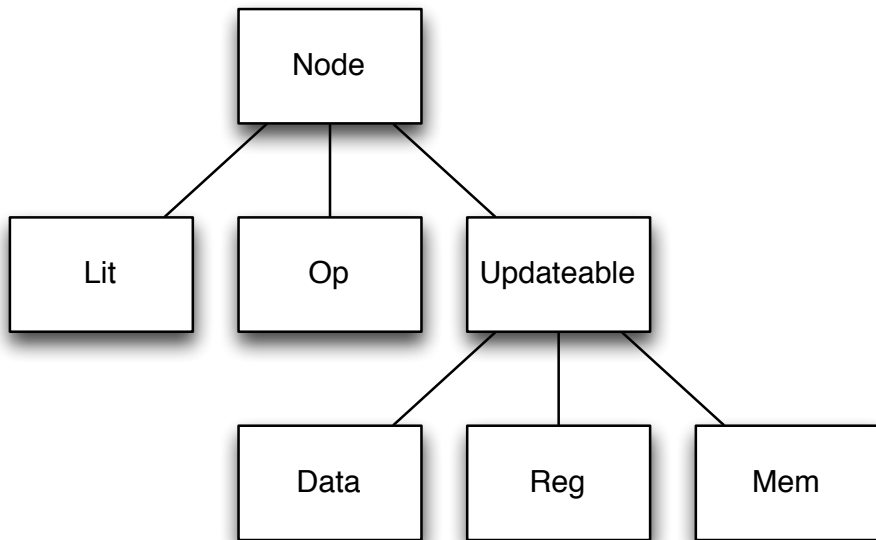
$z = \text{Cat}(x, y)$

$wz = wx + wy$

$z = \text{Fill}(n, x)$

$wz = wx * \text{maxNum}(n)$

Node Class Hierarchy



src	<i># chisel scala source code</i>
csrc	<i># chisel emulator source code</i>
doc	<i># documentation</i>
www	<i># web sources</i>
tutorial	<i># tutorial project</i>
tutorial/src	<i># tutorial source code</i>
tutorial/sbt	<i># tutorial sbt files</i>
tutorial/emulator	<i># tutorial emulator build products</i>
tutorial/emulator/Makefile	<i># Makefile for emulator products</i>
tutorial/verilog	<i># tutorial verilog build products</i>
tutorial/verilog/Makefile	<i># Makefile for verilog products</i>

Combinational – tutorial/src/combinational.scala

```
package Tutorial

import Chisel._

class Combinational extends Component {
  val io = new Bundle {
    val x = UFix(16, INPUT)
    val y = UFix(16, INPUT)
    val z = UFix(16, OUTPUT)
  }
  io.z := io.x + io.y
}
```

Tutorial Main – tutorial/src/tutorial.scala

```
package Tutorial

import Chisel._

object Tutorial {
  def main(args: Array[String]) = {
    val args = args.slice(1, args.length)
    args(0) match {
      case "combinational" =>
        chiselMain(args, () => new Combinational())
      ...
    }
  }
}
```


Emitting C++ and Compiling

```
def main(args: Array[String]) = {  
  val args = args.slice(1, args.length)  
  args(0) match {  
    case "combinational" => chiselMain(args, () => new Combinational())  
  }  
}
```

get into sbt directory:

```
cd $CHISEL/tutorial/sbt
```

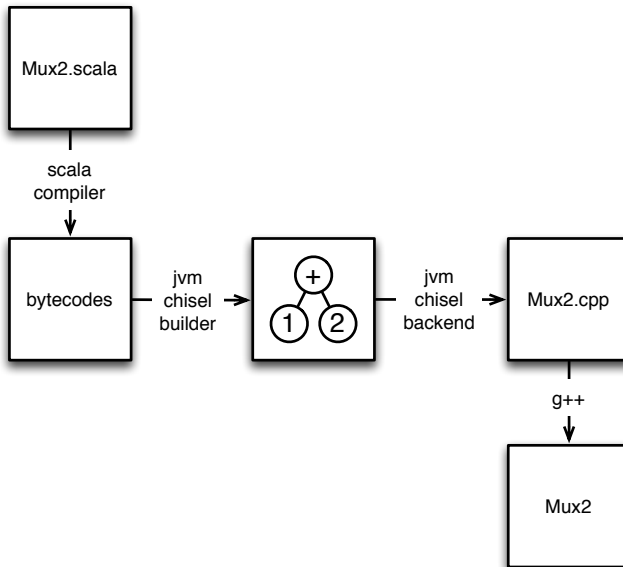
emit C++ and compile tutorial/emulator/Combination-* to produce combinational circuit app named tutorial/emulator/Combination:

```
sbt  
sbt> project tutorial  
sbt> compile  
sbt> run Combinational --backend c --targetDir ../emulator --compile --genHarness  
sbt> exit
```

or on one line

```
sbt "project tutorial" "run Combinational --backend c --targetDir ../emulator --compile --genHarness"
```

Chisel Workflow



Defining a Tester

```
package Tutorial
import Chisel._
import scala.collection.mutable.HashMap
import scala.util.Random

class Combinational extends Component {
  val io = new Bundle {
    val x = UFix(16, INPUT)
    val y = UFix(16, INPUT)
    val z = UFix(16, OUTPUT) }
  io.z := io.x + io.y
}

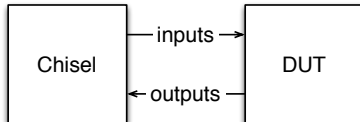
class CombinationalTests(c: Combinational)
  extends Tester(c, Array(c.io)) {
  defTests {
    var allGood = true
    val vars = new HashMap[Node, Node]()
    val rnd = new Random()
    val maxInt = 1 << 16
    for (i <- 0 until 10) {
      vars.clear()
      val x = rnd.nextInt(maxInt)
      val y = rnd.nextInt(maxInt)
      vars(c.io.x) = UFix(x)
      vars(c.io.y) = UFix(y)
      vars(c.io.z) = UFix((x + y) & (maxInt - 1))
      allGood = step(vars) && allGood
    }
    allGood
  } }
}
```

```
class Tester[T <: Component]
  (val c: T, val testNodes: Array[Node])

def defTests(body: => Boolean)

def step(vars: HashMap[Node, Node]): Boolean
```

- user subclasses **Tester** defining DUT and **testNodes** and tests in **defTests** body
- **vars** is mapping from **testNodes** to literals, called bindings
- **step** runs test with given bindings, where var values for input ports are sent to DUT, DUT computes next outputs, and DUT sends next outputs to Chisel
- finally **step** compares received values against var values for and returns false if any comparisons fail output ports



Binding Tester to Component

```
object chiselMainTest {  
  def apply[T <: Component]  
    (args: Array[String], comp: () => T)(tester: T => Tester[T]): T  
}
```

and used as follows:

```
chiselMainTest(args + "--test", () => new Combinational()){  
  c => new CombinationalTests(c)  
}
```

Running Tests Examples

```
cd $CHISEL/tutorial/sbt  
sbt "project tutorial" "run Combinational ... --compile --test"  
...  
PASSED
```

or through makefile

```
cd $CHISEL/tutorial/emulator  
make combinational  
...  
PASSED
```

Functional Abstraction

```
def clb(a: Bits, b: Bits, c: Bits, d: Bits) =  
  (a & b) | (~c & d)  
  
val out = clb(a,b,c,d)
```

Functional Scala

```
class Functional extends Component {  
  def clb(a: Bits, b: Bits, c: Bits, d: Bits) =  
    (a & b) | (~c & d)  
  val io = new Bundle {  
    val x = Bits(16, INPUT)  
    val y = Bits(16, INPUT)  
    val z = Bits(16, OUTPUT)  
  }  
  io.z := clb(io.x, io.y, io.x, io.y)  
}
```

Bundles

```
class MyFloat extends Bundle {  
  val sign      = Bool()  
  val exponent  = UFix(width = 8)  
  val significand = UFix(width = 23)  
}  
  
val x = new MyFloat()  
val xs = x.sign
```



```
// Vector of 5 23-bit signed integers.
val myVec = Vec(5) { Fix(width = 23) }

// Connect to one static element of vector.
val reg3 = myVec(3)
reg3      := data3
myVec(4)  := data4

// Connect to one dynamic element of vector.
val reg    = myVec(addr)
reg        := data1
myVec(addr2) := data2
```

Data object with directions assigned to its members

```
class FIFOInput extends Bundle {  
  val ready = Bool(OUTPUT)  
  val bits  = Bits(32, INPUT)  
  val valid = Bool(INPUT)  
}
```

Direction assigned at instantiation time

```
class ScaleIO extends Bundle {  
  val in    = new MyFloat().asInput  
  val scale = new MyFloat().asInput  
  val out   = new MyFloat().asOutput  
}
```

Component

- inherits from Component,
- contains an interface stored in a port field named `io`, and
- wires together subcircuits in its constructor.

```
class Mux2 extends Component {  
  val io = new Bundle{  
    val sel = Bits(1, INPUT)  
    val in0 = Bits(1, INPUT)  
    val in1 = Bits(1, INPUT)  
    val out = Bits(1, OUTPUT)  
  }  
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)  
}
```

chiselMain Command Line Arguments

```
sbt
sbt> project tutorial
sbt> compile           // compiles Chisel Scala code
sbt> run Combinational // produces C++ files
sbt> run Combinational --compile // produces C++ files and compiles
sbt> run Combinational --test    // produces C++ files, compiles, tests
sbt> exit
```

with a complete set of command line arguments being:

--targetDir	target pathname prefix
--genHarness	generate harness file for C++
--debug	put all wires in C++ class file
--compile	compiles generated C++
--test	runs tests using C++ app
--backend v	generate verilog
--backend c	generate C++ (default)
--vcd	enable vcd dumping

Creating Verilog

```
cd $CHISEL/tutorial/sbt; sbt "project tutorial" "run Mux2 --backend v ..."
```

or through makefile:

```
cd $CHISEL/tutorial/verilog; make Mux2
```

producing Mux2.v:

```
module Mux2(  
    input io_sel,  
    input io_in0,  
    input io_in1,  
    output io_out);  
  
    wire T0;  
    wire T1;  
    wire T2;  
    wire T3;  
  
    assign io_out = T0;  
    assign T0 = T3 | T1;  
    assign T1 = T2 & io_in0;  
    assign T2 = ~ io_sel;  
    assign T3 = io_sel & io_in1;  
endmodule
```

Producing VCD

```
cd $CHISEL/tutorial/sbt  
sbt "project tutorial" "run Mux2 --backend c ... --vcd --compile --test"
```

which then produces

```
$CHISEL/tutorial/emulator/Mux2.vcd
```

which you can view with a vcd viewer

Component Hierarchy Problem – Mux4.scala

- child components stored in fields of parent
- now write 4-to-1 mux out of 3 2-to-1 mux's

```
class Mux4 extends Component {  
  val io = new Bundle {  
    val in0 = Bits(1, INPUT)  
    val in1 = Bits(1, INPUT)  
    val in2 = Bits(1, INPUT)  
    val in3 = Bits(1, INPUT)  
    val sel = Bits(2, INPUT)  
    val out = Bits(1, OUTPUT)  
  }  
  val m0 = new Mux2()  
  m0.io.sel := io.sel(0); m0.io.in0 := io.in0; m0.io.in1 := io.in1  
  
  // flush this out ...  
  
  io.out := io.in0 & io.in1 & io.in2 & io.in3 & io.sel  
}
```

State Elements

```
Reg(in)
```

```
def risingEdge(x: Bool) = x && !Reg(x)
```


Counter

```
def wrapAround(n: UFix, max: UFix) =  
  Mux(n > max, UFix(0), n)  
  
def counter(max: UFix) = {  
  val x = Reg(resetVal = UFix(0, max.getWidth))  
  x := wrapAround(x + UFix(1), max)  
  x  
}
```

Sequential Circuits

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

```
// Flip internal state when input true.  
def toggle(p: Bool) = {  
  val x = Reg(resetVal = Bool(false))  
  x := Mux(p, !x, x)  
  x  
}
```

```
// Square wave where each half cycle has given period.  
def squareWave(period: UFix) = toggle(pulse(period))
```

Sequential Circuit Problem – Accumulator.scala

- write sequential circuit that counts trues

```
class Accumulator extends Component {  
  val io = new Bundle {  
    val in  = Bool(INPUT)  
    val out = UFix(8, OUTPUT)  
  }  
  
  // flush this out ...  
  
  io.out := UFix(0)  
}
```

Forward Declarations

```
val pcPlus4      = UFix()  
val branchTarget = UFix()  
val pcNext       = Mux(io.ctl.pcSel, branchTarget, pcPlus4)  
val pcReg        = Reg(data = pcNext, resetVal = UFix(0, 32))  
pcPlus4          := pcReg + UFix(4)  
...  
branchTarget     := addOut
```

Conditional Updates

```
val r = Reg() { UFix(16) }  
when (c === UFix(0) ) {  
    r := r + UFix(1)  
}
```

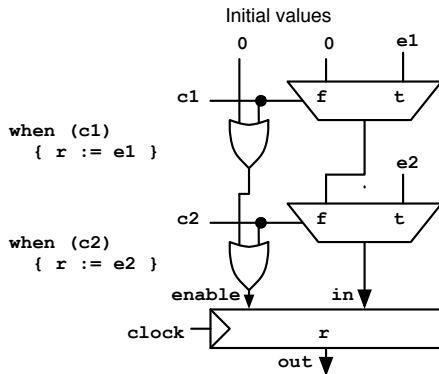
Conditional Updates Priority

```
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.

Targetting Multiple Registers

```
r := Fix(3)
s := Fix(3)
when (c1) { r := Fix(1); s := Fix(1) }
when (c2) { r := Fix(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 }
```

Finite State Machines

```
class Parity extends Component {  
  val io = new Bundle {  
    val in  = Bool(dir = INPUT)  
    val out = Bool(dir = OUTPUT) }  
  val s_even :: s_odd :: Nil = Enum(2){ UFix() }  
  val state = Reg(resetVal = s_even)  
  when (io.in) {  
    when (state === s_even) { state := s_odd }  
    .otherwise               { state := s_even }  
  }  
  io.out := (state === s_odd)  
}
```

FSM Problem – VendingMachine.scala

- write vending machine which needs accepts 20 cents or more before raising valid high

```
class VendingMachine extends Component {  
  val io = new Bundle {  
    val nickel = Bool(dir = INPUT)  
    val dime   = Bool(dir = INPUT)  
    val valid   = Bool(dir = OUTPUT) }  
  val sIdle :: s5 :: s10 :: s15 :: s0k :: Nil = Enum(5){ UFix() }  
  val state = Reg(resetVal = sIdle)  
  
  // flush this out ...  
  
  io.valid := (state == s0k)  
}
```

```
def Vec[T <: Data](elts: Seq[T])(data: => T): Vec[T]  
def Vec[T <: Data](elts: T*)(data: => T): Vec[T]
```

```
val i = Array(UFix(1), UFix(2), UFix(4), UFix(8))  
val m = Vec(i){ UFix(width = 32) }  
val r = m(counter(UFix(3)))
```

Multiplication Lookup Table Problem – Mul.scala

- write 16x16 multiplication table using Vec

```
class Mul extends Component {  
  val io = new Bundle {  
    val x    = UFix(4, INPUT)  
    val y    = UFix(4, INPUT)  
    val z    = UFix(8, OUTPUT)  
  }  
  val muls = new ArrayBuffer[UFix]()  
  
  // flush this out ...  
  
  io.z := UFix(0)  
}
```

Memories

```
def object Mem {  
  def apply[T <: Data](n: Int, resetVal: T = null)(type: => T): Mem  
}  
  
class Mem[T <: Data]  
  (val n: Int, val resetVal: T, val inits: Seq[T], type: () => T)  
  extends Updateable {  
    def apply(addr: UFix): T  
  }
```


Register File

```
val regs = Mem(32){ Bits(width = 32) }  
when (wr_en) {  
  regs(wr_addr) := wr_data  
}  
val idat = regs(iaddr)  
val mdat = regs(maddr)
```

Table Problem — Memo.scala

- write read/write table using Mem

```
class Memo extends Component {  
  val io = new Bundle {  
    val isWr      = Bool(INPUT)  
    val wrAddr    = UFix(8, INPUT)  
    val wrData    = UFix(8, INPUT)  
    val isRd      = Bool(INPUT)  
    val rdAddr    = UFix(8, INPUT)  
    val rdData    = UFix(8, OUTPUT)  
  }  
  val mem = Mem(256){ UFix(width = 8) }  
  
  // flush this out ...  
  
  io.rdData := UFix(0)  
}
```

Port Classes, Subclasses, and Nesting

```
class LinkIO extends Bundle {  
  val data = Bits(16, OUTPUT)  
  val valid = Bool(OUTPUT)  
}
```

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

```
class PLinkIO extends LinkIO {  
  val parity = Bits(5, OUTPUT)  
}
```

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

Filter Example

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

```
class FilterIO extends Bundle {  
  val in  = new LinkIO().flip  
  val out = new LinkIO()  
}
```

where flip recursively changes the “gender” of a bundle, changing input to output and output to input.

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

```
class Filter extends Component {  
  val io = new FilterIO()  
  io.out.valid := io.in.valid  
  io.out.data  := io.in.data  
}
```

where the io field contains FilterIO.

Even Filter Problem – Filter.scala

- write filter that filters out even numbers

```
class Filter extends Component {  
  val io = new FilterIO()  
  
  // flush this out ...  
  
  io.out.valid := Bool(true)  
  io.out.data  := Bits(0)  
}
```

Testing Decoupled Circuits

```
class GCDTests(c: GCD) extends Tester(c, Array(c.io)) {  
  defTests {  
    val (a, b, z) = (64, 48, 16)  
    val svars = new HashMap[Node, Node]()  
    val ovars = new HashMap[Node, Node]()  
    var t = 0  
    do {  
      svars(c.io.a) = UFix(a)  
      svars(c.io.b) = UFix(b)  
      step(svars, ovars)  
      t += 1  
    } while (t <= 1 || ovars(c.io.v).litValue() == 0)  
    ovars(c.io.z).litValue() == z  
  }  
}
```

Bundle Vectors

```
class CrossbarIO extends Bundle {  
  val in  = Vec(2){ new LinkIO() }  
  val sel = UFix(2, INPUT)  
  val out = Vec(2){ new LinkIO() }  
}
```

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

Bundle Vectors Cloning

```
class CrossbarIO(n: Int) extends Bundle {  
  val in  = Vec(n){ new LinkIO() }  
  val sel = UFix(log2Up(n), INPUT)  
  val out = Vec(n){ new LinkIO() }  
  override def clone() = (new CrossbarIO(n)).asInstanceOf[this.type]  
}
```

where `clone` definition fixes cloning, by incorporating the crossbar construction argument `n` in cloning.

Bulk Connections

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

```
class Block extends Component {  
  val io = new FilterIO()  
  val f1 = new Filter()  
  val f2 = new Filter()  
  
  f1.io.in  <> io.in  
  f1.io.out <> f2.io.in  
  f2.io.out <> io.out  
}
```

where <> bulk connects interfaces.

- Bulk connections connect leaf ports of the same name to each other.
- After all connections are made and the circuit is being elaborated, Chisel warns users if ports have other than exactly one connection to them.

Parameterized Functions

```
def Mux[T <: Bits](c: Bool, con: T, alt: T): T  
  
Mux(c, UFix(10), UFix(11))
```

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

Parameterized Signal Processing

$$y[t] = \sum_j w_j * x_j[t - j] \quad (1)$$

```
def innerProductFIR[T <: Num] (w: Seq[T], x: T) = {  
  val delays = Range(0, w.length).map(i => w(i) * delay(x, i))  
  delays.foldRight(_ + _)  
}  
  
def delay[T <: Bits](x: T, n: Int): T =  
  if (n == 0) x else Reg(delay(x, n - 1))
```

Parameterized Classes

```
class FilterIO[T <: Data]() (data: => T) extends Bundle {  
  val in  = data.asInput.flip  
  val out = data.asOutput  
}  
  
class Filter[T <: Data]() (data: => T) extends Component {  
  val io = (new FilterIO()) { data }  
  ...  
}
```

Parameterized Classes Continued

```
class FIFOIO[T <: Data]() (data: => T) extends Bundle {  
  val ready = Bool(INPUT)  
  val valid = Bool(OUTPUT)  
  val bits  = data.asOutput  
}  
  
class PipeIO[+T <: Data]() (data: => T) extends Bundle {  
  val valid = Bool(OUTPUT)  
  val bits  = data.asOutput  
}
```

Decoupled GCD Problem – RealGCD.scala

```
class RealGCDInput extends Bundle {  
  val a = Bits(width = 16)  
  val b = Bits(width = 16)  
}  
  
class RealGCD extends Component {  
  val io = new Bundle {  
    val in  = new FIFOIO()( new RealGCDInput() ).flip()  
    val out = new PipeIO()( Bits(width = 16) )  
  }  
  
  // flush this out ...  
}
```

Object Oriented FIFOIO

```
class EnqIO[T <: Data]() (data: => T) extends FIFOIO()(data) {  
  def enq(dat: T): T = { valid := Bool(true); data := dat; dat }  
  valid := Bool(false);  
  for (io <- data.flatten.map(x => x._2))  
    io := UFix(0, io.getWidth());  
}
```

```
class DeqIO[T <: Data]() (data: => T) extends FIFOIO()(data) {  
  flip()  
  ready := Bool(false);  
  def deq(b: Boolean = false): T = { ready := Bool(true); data }  
}
```

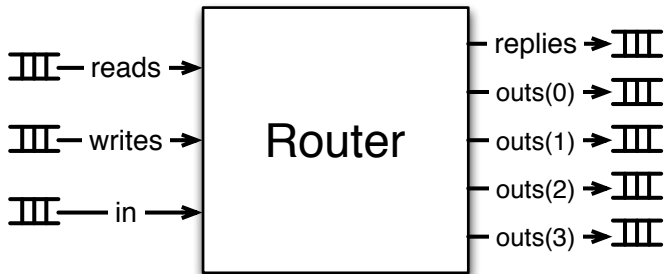
```
class Filter[T <: Data]() (data: => T) extends Component {  
  val io = new Bundle {  
    val in  = (new DeqIO()){ data }  
    val out = (new EnqIO()){ data }  
  }  
  when (io.in.valid && io.out.ready) {  
    io.out.enq(io.in.deq())  
  }  
}
```

Router Interface

```
class ReadCmd extends Bundle {  
  val addr = UFix(width = 32)  
}  
  
class WriteCmd extends ReadCmd {  
  val data = UFix(width = 32)  
}  
  
class Packet extends Bundle {  
  val header = UFix(width = 8)  
  val body   = Bits(width = 64)  
}  
  
class RouterIO(n: Int) extends Bundle {  
  override def clone = new RouterIO(n).asInstanceOf[this.type]  
  val reads  = (new DeqIO()){ new ReadCmd() }  
  val replies = (new EnqIO()){ UFix(width = 8) }  
  val writes = (new DeqIO()){ new WriteCmd() }  
  val in     = (new DeqIO()){ new Packet() }  
  val outs   = Vec(n){ (new EnqIO()){ new Packet() } }  
}
```


Router Problem – Router.scala

```
class Router extends Component {  
  val depth = 32;  
  val n      = 4;  
  val io     = new RouterIO(n);  
  val tbl    = Mem(depth){ UFix(width = sizeof(n)) };  
  
  // flush it out ...  
}
```



- git
- sbt
- project directory structure
- project file
- installation

```
cd ${HOME}
git clone git@github.com:ucb-bar/chisel.git
export CHISEL=${HOME}/chisel
git pull
git status
git log
git add filename
git commit -m "comment"
git push
```

```
cd ${CHISEL}/tutorial/sbt  
sbt  
project tutorial  
compile  
run  
console
```

Project Directory Structure

```
chisel/  
  tutorial/  
  src/  
gpu/  
  chisel -> ../chisel  
  sbt/  
    project/build.scala # edit this as shown below  
    chisel/src/main/scala -> $CHISEL/src  
    gpu/src/main/scala -> ../../../../src  
  src/ # your source files go here  
    gpu.scala  
  emulator/ # your C++ target can go here
```

Project File

```
import sbt._
import Keys._

object BuildSettings {
  val buildOrganization = "edu.berkeley.cs"
  val buildVersion = "1.1"
  val buildScalaVersion = "2.9.2"

  val buildSettings = Defaults.defaultSettings ++ Seq (
    organization := buildOrganization,
    version      := buildVersion,
    scalaVersion := buildScalaVersion
  )
}

object ChiselBuild extends Build {
  import BuildSettings._

  lazy val chisel =
    Project("chisel", file("chisel"),
      settings = buildSettings)
  lazy val gpu =
    Project("gpu", file("gpu"), settings = buildSettings)
    dependsOn(chisel)
}
```

Installation

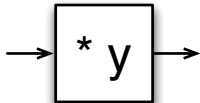
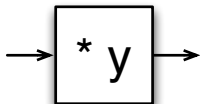
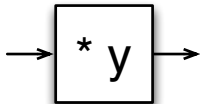
- on mac install:
 - XCODE console tools
 - <http://www.macports.org>
- on windows install:
 - cygwin
- everywhere install:
 - git
 - g++
 - java
- everywhere
 - `export $CHISEL=...`
 - `git clone https://github.com/ucb-bar/chisel.git`

Projects Ideas

audio processing	Echo.scala
image processing	Darken.scala
risc processor	Risc.scala
game of life	Life.scala
router	Router.scala
map/reduce	<i>see next slide</i>
network	
fft	
cryptography	
serial multiplier	
pong	

Functional Composition

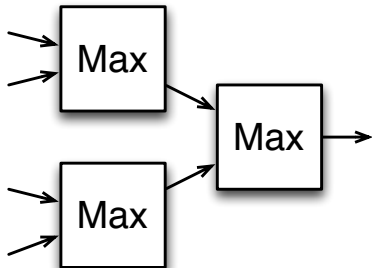
Map(*ins*, $x \Rightarrow x * y$)



Chain(*n*, *in*, $x \Rightarrow f(x)$)



Reduce(*data*, Max)



Keep in Touch

website	<code>chisel.eecs.berkeley.edu</code>
mailing list	<code>groups.google.com/group/chisel-users</code>
github	<code>https://github.com/ucb-bar/chisel/</code>
me	<code>jrb@pobox.com</code>

Thanks

- **Arrangements** – Roxana and Tammy
- **EC2 configuration** – Henry Cook and Michael Armbrust
- **Bootcamp Materials** – Huy Vo and Andrew Waterman
- **Dry Runs** – Danny, Quan, and Albert
- **Funding** – Department of Energy