Constructing Hardware In Scala Embedded Language

- Embed a hardware-description language in Scala, using Scala’s extension facilities
- Chisel is just a set of class definitions in Scala and when you write a Chisel program you are actually writing a Scala program
- A hardware module is just a data structure in Scala
- Clean simple set of design construction primitives for RTL design
- Full power of Scala for writing hardware generators
- Different output routines can generate different types of output (C, FPGA-Verilog, ASIC-Verilog) from same hardware representation
- Can be extended above with domain specific languages (such as declarative cache coherence specifications)
- Can be extended below with new backends (such as quantum)
- Open source with lots of libraries
- Only 5200 lines of code in current version!
Chisel Workflow

Chisel Program

Scala / JVM

C++ Code

FPGA Verilog

ASIC Verilog

C++ Compiler

FPGA Tools

ASIC Tools

C++ Simulator

FPGA Emulation

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

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

// List’s
val els = List(1, 2, 3)
val a :: b :: c :: Nil = els
val m = els.length
val tbl = new Array[Int](256)

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

// loop of each sequence element
for (e <- tbl)
    tbl(i) += e

// nested loop
for (i <- 0 until 16; j <- 0 until 16)
    tbl(j*16 + i) = i

// create second table with doubled elements
val tbl2 = for (i <- 0 until 16) yield tbl(i)*2
// 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)
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
scala> 1 + 2
=> 3
scala> def f (x: Int) = 2 * x
=> (Int) => Int
scala> f(4)
=> 8
class Mux2 extends Component {
  val io = new Bundle{
    val sel = Bits(INPUT, 1)
    val in0 = Bits(INPUT, 1)
    val in1 = Bits(INPUT, 1)
    val out = Bits(OUTPUT, 1)
  }
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
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)
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("012", 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.
UFix(1)
\[ \text{UFix}(1) + \text{UFix}(1) \]
(sel & in1) | (~sel & in0)
val sel = a | b
val out = (sel & in1) | (~sel & in0)
val sel = Bits()
val out = (sel & in1) | (~sel & in0)

sel := a | b
Valid on Bits, Fix, UFix, Bool.

```scala
// 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
```
Valid on Bits, Fix, UFix, and Bool. Returns Bool.

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

where `===` is used instead of `==` to avoid collision with Scala.
Valid on Bits, Fix, and UFix.

```scala
// Logical left shift.
val twoToTheX = Fix(1) << x

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

where logical is a raw shift and arithmetic performs top bit sign extension.
Valid on Bits, Fix, UFix, and Bool.

```scala
// 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)
```
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)
Valid on Nums: Fix and UFix.

```scala
// 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
```

where Fix is a signed fixed-point number represented in two’s complement and UFix is an unsigned fixed-point number.
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
<table>
<thead>
<tr>
<th>operation</th>
<th>bit width</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z = x + y$</td>
<td>$w_z = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = x - y$</td>
<td>$w_z = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = x &amp; y$</td>
<td>$w_z = \min(wx, wy)$</td>
</tr>
<tr>
<td>$z = x</td>
<td>y$</td>
</tr>
<tr>
<td>$z = \text{Mux}(c, x, y)$</td>
<td>$w_z = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = w * y$</td>
<td>$w_z = wx + wy$</td>
</tr>
<tr>
<td>$z = x &lt;&lt; n$</td>
<td>$w_z = wx + \maxNum(n)$</td>
</tr>
<tr>
<td>$z = x &gt;&gt; n$</td>
<td>$w_z = wx - \minNum(n)$</td>
</tr>
<tr>
<td>$z = \text{Cat}(x, y)$</td>
<td>$w_z = wx + wy$</td>
</tr>
<tr>
<td>$z = \text{Fill}(n, x)$</td>
<td>$w_z = wx * \maxNum(n)$</td>
</tr>
</tbody>
</table>
def mux2 (sel: Bits, in0: Bits, in1: Bits) =
  (sel & in1) | (~sel & in0)
val out = mux2(k,a,b)
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 3 23-bit signed integers.
val myVec = Vec(3) { Fix(width = 23) }

- can be used as Scala sequences
- can also be nested into Chisel Bundles
val myVec = Vec(3) { Fix(width = 23) }

// Connect to one vector element chosen at elaboration time.
val fix0 = myVec(0)
val fix1 = myVec(1)
fix1 := data1
myVec(2) := data2
val myVec = Vec(3) { Fix(width = 23) }

// Connect to one vector element chosen at runtime.
val out0 = myVec(addr0)
val out1 = myVec(addr1)
myVec(addr2) := data2
Data object with directions assigned to its members

```scala
class FIFOIO extends Bundle {
  val data = Bits(INPUT, 32)
  val valid = Bool(OUTPUT)
  val ready = Bool(INPUT)
}
```

Direction assigned at instantiation time

```scala
class ScaleIO extends Bundle {
  val in = new MyFloat().asInput
  val scale = new MyFloat().asInput
  val out = new MyFloat().asOutput
}
```
- inherits from Component,
- contains an interface stored in a port field named \( \text{io} \), and
- wires together subcircuits in its constructor.

```scala
class Mux2 extends Component {
  val io = new Bundle{
    val sel = Bits(INPUT, 1)
    val in0 = Bits(INPUT, 1)
    val in1 = Bits(INPUT, 1)
    val out = Bits(OUTPUT, 1)
  }
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```
Mux2.scala → scala compiler → bytecodes → jvm chisel builder → Mux2.v

Mux2.v → jvm verilog backend → verification

Mux2.v → jvm cpp backend → net list + power, area, and speed estimates

Mux2.v → g++ → Mux2
def risingEdge(x: Bool) = x && !Reg(x)
def counter(max: UFix) = {
  val x = Reg(resetVal = UFix(0, max.getWidth))
  x := Mux(x == max, UFix(0), x + UFix(1))
  x
}
What is Chisel?

- Chisel is just a set of class definitions in Scala and when you write a Chisel program you are actually writing a Scala program,
- Chisel programs produce and manipulate a data structure in Scala using a convenient textural language layered on top of Scala,
- Chisel makes it possible to create powerful and reusable hardware components using modern programming language concepts, and
- the same Chisel description can generate different types of output.
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() { 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:**

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>r</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>r</td>
<td>r unchanged</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>c2 takes precedence over c1</td>
</tr>
</tbody>
</table>
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(){ Fix(3) }
s := Reg(){ 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:

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

r updated in c2 block, s updated using default
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

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

which is the same as:

```javascript
when (idx === v1) { u1 }
when (idx === v2) { u2 }
```
Conditional updates also work for

- wires but must have defaults and
- for memory reads and writes as we’ll see soon...

For wires, we can do conditional updates as follows:

```scala
val w = Bits(width = 32)
w := Bits(0) // default value
when (c1) { w := Bits(1) }
when (c2) { w := Bits(2) }
```

which is the same as

```scala
val w = Bits(width = 32)
when (Bool(true)) { w := Bits(0) } // default value
when (c1) { w := Bits(1) }
when (c2) { w := Bits(2) }
```
Finite state machines can now be readily defined as follows:

```scala
class Parity extends Component {
  val io = new Bundle {
    val in  = Bool(INPUT)
    val out = Bool(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) 
}
```

where `Enum(2){ UFix() }` creates a list of two `UFix()` literals.
val d = Array(UFix(1), UFix(2), UFix(4), UFix(8))
val m = ROM(d){ UFix(width = 32) }
val r = m(counter(UFix(3)))
class Mul extends Component {
    val io = new Bundle {
        val x = UFix(INPUT, 4)
        val y = UFix(INPUT, 4)
        val z = UFix(OUTPUT, 8) }

    val muls = new Array[UFix](256)
    for (x <- 0 until 16; y <- 0 until 16)
        muls((x << 4) | y) = x * y

    val tbl = ROM(muls){ UFix(8) }

    io.z := tbl((io.x << 4) | io.y)
}
RAM is supported using the `Mem` construct

```scala
val m = Mem(32){ Bits(width = 32) }
```

where
- writes to Mems are positive-edge-triggered
- reads are either combinational or positive-edge-triggered
- ports are created by applying a `UFix` index
val regs = Mem(32){ Bits(width = 32) }
when (wrEn) {
    regs(wrAddr) := wrData
}
val iDat = regs(iAddr)
val mDat = regs(mAddr)
Sequential read ports are inferred when:
- optional parameter `seqRead` is set and
- a reg is assigned to the output of a MemRead

```scala
val ram1r1w = Mem(1024, seqRead = true) { Bits(width = 32) }
val dOut = Reg() { Bits() }
when (wrEn) { ram1r1w(wrAddr) := wrData }
when (rdEn) { dOut := ram1r1w(rdAddr) }
```
Single-ported SRAMs can be inferred when the read and write conditions are mutually exclusive in the same when chain.

```java
val ram1p = Mem(1024, seqRead = true) { Bits(width = 32) }
val dOut  = Reg() { Bits() }
when (wrEn) { ram1p(wrAddr) := wrData }
  .elsewhen (rdEn) { dOut := ram1p(rdAddr) }
```
Suppose we want to break computation into a series of filters (ala Unix):

where data is fed though with an additional \texttt{valid} signal to say whether data has \textbf{not} been filtered.
We can define a pass through filter component by defining a filter class extending component:

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

where the `io` field contains `FilterIO`. 
Suppose we want to write a small and odd filter. We could write these out by hand:

class SmallFilter extends Component {
  val io = new FilterIO()
  io.out.data := io.in.data
  io.out.valid := io.in.valid && (io.in.data < 10)
}

class OddFilter extends Component {
  val io = new FilterIO()
  io.out.data := io.in.data
  io.out.valid := io.in.valid && (io.in.data & 1)
}
class PipeIO extends Bundle {
    val data = UFix(OUTPUT, 16)
    val valid = Bool(OUTPUT)
}
From there we can define a filter interface by nesting two PipeIOs into a new FilterIO bundle:

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

where `flip` recursively changes the “gender” of a bundle, changing input to output and output to input.
We can now compose two filters into a filter block as follows:

```
class SmallOdds extends Component {
  val io = new FilterIO()
  val smalls = new SmallFilter()
  val odds = new OddFilter()

  smalls.io.in <> io.in
  smalls.io.out <> odds.io.in
  odds.io.out <> io.out
}
```

where <> bulk connects interfaces. Note that:

- bulk connections recursively pattern match names between left and right hand sides finally connecting leaf ports to each other, and
- after all connections are made and the circuit is being elaborated, Chisel warns users if ports have other than exactly one connection to them.
read **Chisel Tutorial**

come prepared with
- one paragraph summary with the good and the bad
- three questions

install chisel using chisel installation guide on chisel.eecs.berkeley.edu

check out course website
http://inst.eecs.berkeley.edu/ cs294-88/sp13/