Standard Design Methodology

Hierarchically defines structure and function of circuit.

**Specification**

- Verification: Does the design behave as required with regards to function (and timing, and power consumption)?
- Maps design to resources of implementation platform (FGPA or ASIC).
- Design circuits graphically
- Used commonly until approximately 2002
- Schematics are intuitive
- Labor intensive to produce (especially readable ones).
- Requires a special editor tool
- Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow on large designs
**Structural Description:**
connections of components with a nearly one-to-one correspondence to schematic diagram.

```plaintext
Decoder(output x0,x1,x2,x3; input a,b) {
    wire abar, bbar;
    inv(bbar, b);
    inv(abar, a);
    and(x0, abar, bbar);
    and(x1, abar, b );
    and(x2, a, bbar);
    and(x3, a, b ) ;
}
```

**Behavioral Description:** use high-level constructs (similar to conventional programming) to describe the circuit function.

```plaintext
Decoder(output x0,x1,x2,x3; input a,b) {
    case [a b]
        00: [x0 x1 x2 x3] = 0x1;
        01: [x0 x1 x2 x3] = 0x2;
        10: [x0 x1 x2 x3] = 0x4;
        11: [x0 x1 x2 x3] = 0x8;
    endcase;
}
```
Originally invented for simulation

Many constructs don’t synthesize: ex: deassign, timing constructs

Others lead to mysterious results: for-loops

Difficult to understand synthesis implications of procedural assignments (always blocks), and blocking versus non-blocking assignments

In common use, most users ignore much of the language and stick to a very strict style

Very weak meta programming support for creating circuit generators

Various hacks around this over the years, ex: embedded TCL scripting

VHDL has much the same issues
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!
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)
}
Literals

Bits(1)       // decimal 1-bit literal from Scala Int.
Bits("ha")   // hexadecimal 4-bit literal from string.
Bits("012")  // 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.
```
Literal Node Construction

UFix(1)
UFix(1) + UFix(1)

UFix(1) +
UFix(2) +
(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, and UFix. Returns Bool.

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

where reduction applies the operation to all the bits.
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.

// 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>$w_z = \max(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = x - y$</td>
<td>$w_z = \max(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = x &amp; y$</td>
<td>$w_z = \min(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = x \mid y$</td>
<td>$w_z = \max(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = \text{Mux}(c, x, y)$</td>
<td>$w_z = \max(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = w \ast y$</td>
<td>$w_z = w_x + \max(w_x, w_y)$</td>
</tr>
<tr>
<td>$z = x \ll n$</td>
<td>$w_z = w_x + \max(w_x, maxNum(n))$</td>
</tr>
<tr>
<td>$z = x \gg n$</td>
<td>$w_z = w_x - \min(w_x, minNum(n))$</td>
</tr>
<tr>
<td>$z = \text{Cat}(x, y)$</td>
<td>$w_z = w_x + w_y$</td>
</tr>
<tr>
<td>$z = \text{Fill}(n, x)$</td>
<td>$w_z = w_x \ast \max(w_x, maxNum(n))$</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 `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)
}
```
Chisel Workflow

Mux2.scala -> scala compiler -> bytecodes -> jvm chisel builder

Mux2.v -> dc -> jvm verilog backend

Mux2.v -> vcs -> verification

Mux2.v -> jvm cpp backend

Mux2.v -> exec

Mux2.cpp -> g++ -> Mux2

Mux2.cpp -> net list + power, area, and speed estimates
Reg(in)
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
}
// 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))