Hierarchically defines structure and function of circuit.

**Specification**

**Simulation**

Verification: Does the design behave as required with regards to function (and timing, and power consumption)?

**Synthesis**

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.

```verilog
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.

```verilog
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;
}
```
Verilog Issues

- 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
Traditional Hardware Design Process

- specify design
- write Verilog
- verify
- synthesize and layout

- specify design generator
- write generator (python, perl)
- write Verilog
- verify
- synthesize and layout
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 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)
Chisel Hardware Design Process

1. Specify design generator
2. Write generator (chisel+scala)
3. Generate verilog
4. Verify
5. Synthesize and layout
// 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
val buf = new ArrayBuffer[Int]()
buf += 12
val z = buf(0)
val l = buf.length

// List’s
val els = List(1, 2, 3)
val (x, y, z) = (1, 2, 3)
val els2 = x :: y :: y :: Nil
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

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

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

// 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

// 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 = r
}

new Zep(true, 100.0)
object Blimp {
    var numBlimps = 0
    def apply(r: Double) = {
        numBlimps += 1
        new Blimp(r)
    }
}

Blimp.numBlimps
Blimp(10.0)
- shallow copy of object
- user can override method to incorporate parameters
- `this.type` allows precise return types

```scala
class Blimp(r: Double) {
  val rad = r
  override def clone(): this.type = new Blimp(r)
}

val b1 = new Blimp(10)
val b2 = b1.clone()
```
scala> 1 + 2
=> 3
scala> def f (x: Int) = 2 * x
=> (Int) => Int
scala> f(4)
=> 8
class Mux2 extends Module {
    val io = new Bundle{
        val sel = UInt(INPUT, 1)
        val in0 = UInt(INPUT, 1)
        val in1 = UInt(INPUT, 1)
        val out = UInt(OUTPUT, 1)
    }
    io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
<table>
<thead>
<tr>
<th>Literal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UInt(1)</td>
<td>decimal 1-bit literal from Scala Int.</td>
</tr>
<tr>
<td>UInt(&quot;ha&quot;)</td>
<td>hexadecimal 4-bit literal from string.</td>
</tr>
<tr>
<td>UInt(&quot;o12&quot;)</td>
<td>octal 4-bit literal from string.</td>
</tr>
<tr>
<td>UInt(&quot;b1010&quot;)</td>
<td>binary 4-bit literal from string.</td>
</tr>
<tr>
<td>SInt(5)</td>
<td>signed decimal 4-bit literal from Scala Int.</td>
</tr>
<tr>
<td>SInt(-8)</td>
<td>negative decimal 4-bit literal from Scala Int.</td>
</tr>
<tr>
<td>UInt(5)</td>
<td>unsigned decimal 3-bit literal from Scala Int.</td>
</tr>
<tr>
<td>Bool(true)</td>
<td>Bool literals from Scala literals.</td>
</tr>
<tr>
<td>Bool(false)</td>
<td></td>
</tr>
</tbody>
</table>
Literals

UInt("h_dead_beef") // 32-bit literal of type UInt.
UInt(1) // decimal 1-bit literal from Scala Int.
UInt("ha", 8) // hexadecimal 8-bit literal of type UInt.
UInt("o12", 6) // octal 6-bit literal of type UInt.
UInt("b1010", 12) // binary 12-bit literal of type UInt.

SInt(5, 7) // signed decimal 7-bit literal of type SInt.
UInt(5, 8) // unsigned decimal 8-bit literal of type UInt.
UInt(1)
$\text{UInt}(1) + \text{UInt}(2)$
(sel & in1) | (~sel & in0)
val sel = a | b
val out = (sel & in1) | (~sel & in0)
val sel = UInt()
val out = (sel & in1) | (~sel & in0)
sel := a | b
Valid on UInt, SInt, Bool.

```scala
// Bitwise-NOT
val invertedX = ~x

// Bitwise-AND
val hiBits = x & UInt("h_ffff_0000")

// Bitwise-OR
val flagsOut = flagsIn | overflow

// Bitwise-XOR
val flagsOut = flagsIn ^ toggle
```
Valid on UInt and SInt. 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 UInt, SInt, 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 SInt and UInt.

// Logical left shift.
val twoToTheX = SInt(1) << x

// Right shift (logical on UInt & UInt, arithmetic on SInt).
val hiBits = x >> UInt(16)

where logical is a raw shift and arithmetic performs top bit sign extension.
Valid on SInt, UInt, 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, UInt("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: SInt and UInt.

// 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 SInt is a signed fixed-point number represented in two’s complement and UInt is an unsigned fixed-point number.
Valid on Nums: SInt and UInt. 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>$wz = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = x - y$</td>
<td>$wz = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = x &amp; y$</td>
<td>$wz = \min(wx, wy)$</td>
</tr>
<tr>
<td>$z = x \mid y$</td>
<td>$wz = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = \text{Mux}(c, x, y)$</td>
<td>$wz = \max(wx, wy)$</td>
</tr>
<tr>
<td>$z = w \times y$</td>
<td>$wz = wx + wy$</td>
</tr>
<tr>
<td>$z = x \ll n$</td>
<td>$wz = wx + \max\text{Num}(n)$</td>
</tr>
<tr>
<td>$z = x \gg n$</td>
<td>$wz = wx - \min\text{Num}(n)$</td>
</tr>
<tr>
<td>$z = \text{Cat}(x, y)$</td>
<td>$wz = wx + wy$</td>
</tr>
<tr>
<td>$z = \text{Fill}(n, x)$</td>
<td>$wz = wx \times \max\text{Num}(n)$</td>
</tr>
</tbody>
</table>
def mux2 (sel: UInt, in0: UInt, in1: UInt) =
  (sel & in1) | (~sel & in0)

val out = mux2(k,a,b)
```scala
class MyFloat extends Bundle {
  val sign = Bool()
  val exponent = UInt(width = 8)
  val significand = UInt(width = 23)
}
val x = new MyFloat()
val xs = x.sign
```
// Vector of 3 23-bit signed integers.
val myVec = Vec.fill(3) { SInt(width = 23) }

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

// Connect to one vector element chosen at elaboration time.
val sint0 = myVec(0)
val sint1 = myVec(1)
fix1 := data1
myVec(2) := data2
val myVec = Vec.fill(3) { SInt(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 Decoupled extends Bundle {
    val data = UInt(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
}
```
Module

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

```scala
class Mux2 extends Module {
  val io = new Bundle{
    val sel = UInt(INPUT, 1)
    val in0 = UInt(INPUT, 1)
    val in1 = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 1)
  }
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```
Chisel Workflow

Mux2.scala -> scala compiler -> bytecodes

bytecodes -> jvm chisel builder -> Mux2.v

Mux2.v -> jvm verilog backend -> verification

Mux2.v -> jvm cpp backend -> Mux2.cpp

Mux2.cpp -> g++ -> Mux2

Mux2.v -> net list + power, area, and speed estimates
Reg(next = in)
def risingEdge(x: Bool) = x && !Reg(next = x)
def counter(max: UInt) = {
  val x = Reg(init = UInt(0, max.getWidth))
  x := Mux(x === max, UInt(0), x + UInt(1))
  x
}
/ Produce pulse every n cycles.
def pulse(n: UInt) = counter(n - UInt(1)) === UInt(0)

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

// Square wave where each half cycle has given period.
def squareWave(period: UInt) = toggle(pulse(period))
// reset
eval_combinational(true);
assign_next_state(true);

// execution
loop {
  eval_combinational(false);
  assign_next_state(false);
}