Chisel Bootcamp

Jonathan Bachrach

EECS UC Berkeley

March 11, 2015

Goals for Bootcamp

- get you started with Chisel
- get a basic working knowledge of Chisel
- learn how to think in Chisel
- know where to get more information

```
-Install VirtualBox
-File->Import appliance, chisel-vm.ova
-Start
-Login (username: chisel-bootcamp, password: chisel)
-GTKWave, Emacs, etc. all installed
```

Development Tool Installation

- **MacOSX**:
  - Install XCODE, including console tools.
- **Linux**:
  - To prepare your Linux platform for Chisel, you will need to install the following packages:
    - `g++`
    - `openjdk-7-jre`
  - using
    - `sudo apt-get install`

```
git clone https://github.com/ucb-bar/chisel-tutorial.git
cd chisel-tutorial
```

Chisel Tutorial Contents

```
chisel-tutorial/
  Makefile
  examples/    # Contains chisel examples
    Makefile
    build.sbt # Contains project description
    FullAdder.scala ...
  problems/    # Contains skeletal files for tutorial problems
    Makefile
    build.sbt # Contains project description
    Accumulator.scala ...
  solutions/   # Contains solutions to problems
    Makefile
    build.sbt # Contains project description
    Counter.scala ...
```

Test It


```
cd $TUT_DIR
make
```

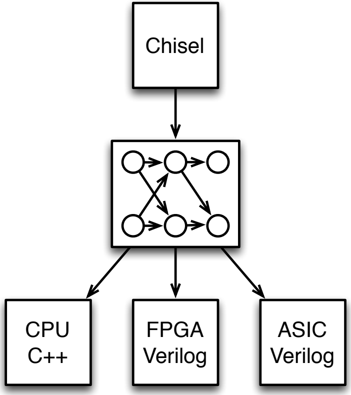
If your system is set up correctly, you should see a messsage `[success]` followed by the total time of the run, and date and time of completion.

`chisel.eecs.berkeley.edu/chisel-bootcamp.pdf`

Chisel

- A hardware construction language
  - "synthesizable by construction"
  - creates graph representing hardware
- Embedded within Scala language to leverage mindshare and language design
- Best of hardware and software design ideas
- Multiple targets
  - Simulation and synthesis
  - Memory IP is target-specific
- **Not** Scala app -> Verilog arch

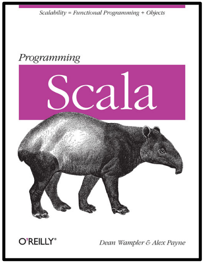
single source

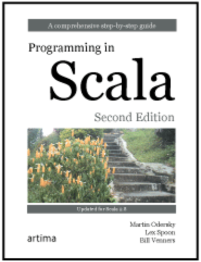


multiple targets

The Scala Programming Language

- Object Oriented
  - Factory Objects, Classes
  - Traits, overloading etc
  - Strongly typed with type inference
- Functional
  - Higher order functions
  - Anonymous functions
  - Currying etc
- Extensible
  - Domain Specific Languages (DSLs)
- Compiled to JVM
  - Good performance
  - Great Java interoperability
  - Mature debugging, execution environments
- Growing Popularity
  - Twitter
  - many Universities





```scala
// constant
val x = 1
val (x, y) = (1, 2)

// variable
var y = 2
y = 3
```

```scala
// 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)
```

```scala
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
```

```scala
import scala.collection.mutable.HashSet

val keys = new HashSet[Int]()
keys += 1
keys += 5
keys.size -> 2
keys.contains(2) -> false
```

```scala
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)
```

Scala Functions


```
// 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)
```

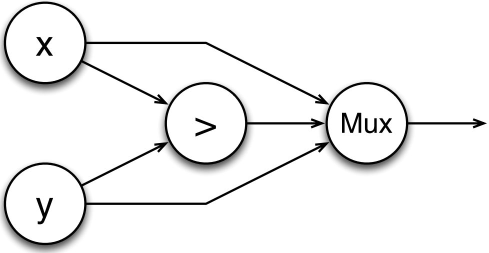
```scala
class Blimp(r: Double) {
  val rad = r
  println("Another Blimp")
}

new Blimp(10.0)
```

```scala
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

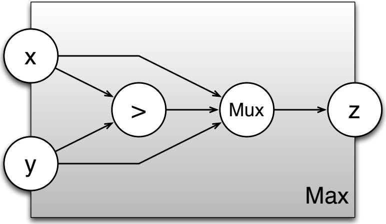
```
object Blimp {
  var numBlimps = 0
  def apply(r: Double) = {
    numBlimps += 1
    new Blimp(r)
  }
}

Blimp.numBlimps
Blimp(10.0)
```

```
Mux(x > y, x, y)
```



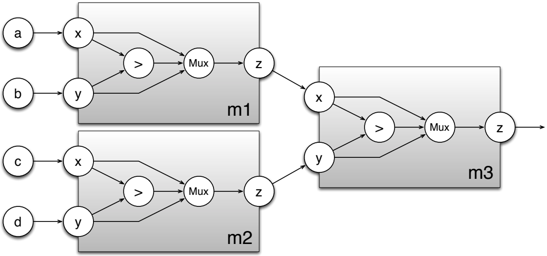
Creating Module


```
class Max2 extends Module {
  val io = new Bundle {
    val x = UInt(INPUT, 8)
    val y = UInt(INPUT, 8)
    val z = UInt(OUTPUT, 8) }
  io.z := Mux(io.x > io.y, io.x, io.y)
}
```



Connecting Modules

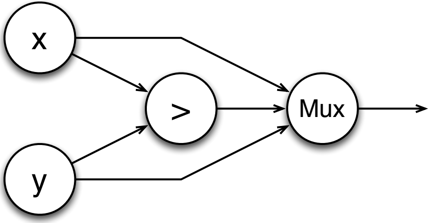

```
val m1 =
  Module(new Max2())
m1.io.x := a
m1.io.y := b
val m2 =
  Module(new Max2())
m2.io.x := c
m2.io.y := d
val m3 =
  Module(new Max2())
m3.io.x := m1.io.z
m3.io.y := m2.io.z
```



Defining Construction Functions

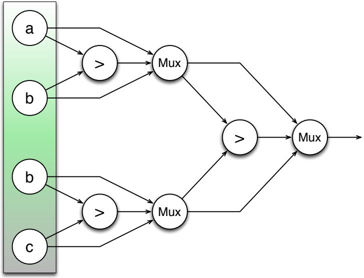
```
def Max2(x, y) = Mux(x > y, x, y)
```

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



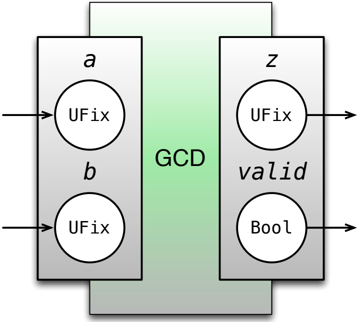
Functional Construction


```
class MaxN(n: Int, w: Int) extends Module {
  val io = new Bundle {
    val in  = Vec.fill(n){ UInt(INPUT, w) }
    val out = UInt(OUTPUT, w)
  }
  io.out := io.in.reduceLeft(Max2)
}
```



Example

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



Running the Chisel Simulation

```
cd ~/chisel-tutorial/examples
make GCD.out
```

```
...
PASSED
[success] Total time: 2 s, completed Feb 28, 2013 8:14:37 PM
```

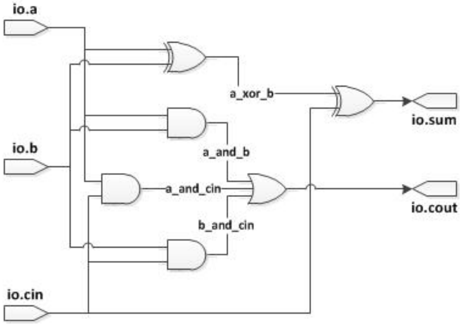
```
cd ~/chisel-tutorial/examples
make GCD.v
```

The Verilog source is roughly divided into three parts:

1. Module declaration with input and outputs
2. Temporary wire and register declaration used for holding intermediate values
3. Register assignments in `always @ (posedge clk)`

FullAdder – Type Inference

```
class FullAdder extends Module {
  val io = new Bundle {
    val a    = UInt(INPUT, 1)
    val b    = UInt(INPUT, 1)
    val cin  = UInt(INPUT, 1)
    val sum  = UInt(OUTPUT, 1)
    val cout = UInt(OUTPUT, 1)
  }
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b   = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout :=
    a_and_b | b_and_cin | a_and_cin
}
```



FullAdder Verilog – Width Inference 1

```scala
class FullAdder extends Module {
  val io = new Bundle {
    val a    = UInt(INPUT, 1)
    val b    = UInt(INPUT, 1)
    val cin  = UInt(INPUT, 1)
    val sum  = UInt(OUTPUT, 1)
    val cout = UInt(OUTPUT, 1)
  }
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b   = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout :=
    a_and_b | b_and_cin | a_and_cin
}
```

```verilog
module FullAdder(
    input  io_a,
    input  io_b,
    input  io_cin,
    output io_sum,
    output io_cout);
  wire T0;
  wire a_and_cin;
  wire T1;
  wire b_and_cin;
  wire a_and_b;
  wire T2;
  wire a_xor_b;

  assign io_cout = T0;
  assign T0 = T1 | a_and_cin;
  assign a_and_cin = io_a & io_cin;
  assign T1 = a_and_b | b_and_cin;
  assign b_and_cin = io_b & io_cin;
  assign a_and_b = io_a & io_b;
  assign io_sum = T2;
  assign T2 = a_xor_b ^ io_cin;
  assign a_xor_b = io_a ^ io_b;
endmodule
```

FullAdder2 Verilog – Width Inference 2


```
class FullAdder2 extends Module {
  val io = new Bundle {
    val a    = UInt(INPUT, 2)
    val b    = UInt(INPUT, 2)
    val cin  = UInt(INPUT, 2)
    val sum  = UInt(OUTPUT, 2)
    val cout = UInt(OUTPUT, 2)
  }
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b   = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout :=
    a_and_b | b_and_cin | a_and_cin
}
```

```
module FullAdder(
    input [1:0] io_a,
    input [1:0] io_b,
    input [1:0] io_cin,
    output[1:0] io_sum,
    output[1:0] io_cout);
  wire[1:0] T0;
  wire[1:0] a_and_cin;
  wire[1:0] T1;
  wire[1:0] b_and_cin;
  wire[1:0] a_and_b;
  wire[1:0] T2;
  wire[1:0] a_xor_b;

  assign io_cout = T0;
  assign T0 = T1 | a_and_cin;
  assign a_and_cin = io_a & io_cin;
  assign T1 = a_and_b | b_and_cin;
  assign b_and_cin = io_b & io_cin;
  assign a_and_b = io_a & io_b;
  assign io_sum = T2;
  assign T2 = a_xor_b ^ io_cin;
  assign a_xor_b = io_a ^ io_b;
endmodule
```

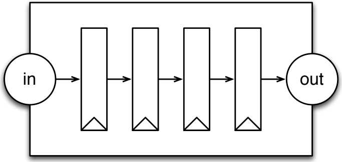
Using Registers


```
// clock the new reg value on every cycle
val y = io.x
val z = Reg(next = y)
```

```
// clock the new reg value when the condition a > b
val x = Reg(UInt())
when (a > b) { x := y }
.elsewhen (b > a) { x := z }
.otherwise { x := w }
```

Unconditional Register Update

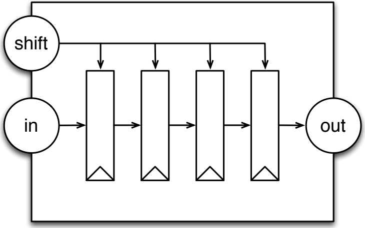
```
class ShiftRegister extends Module {
  val io = new Bundle {
    val in  = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 1)
  }
  val r0 = Reg(next = io.in)
  val r1 = Reg(next = r0)
  val r2 = Reg(next = r1)
  val r3 = Reg(next = r2)
  io.out := r3
}
```




```
module ShiftRegister(input clk, input reset,
    input  io_in,
    output io_out);

  reg[0:0] r3;
  reg[0:0] r2;
  reg[0:0] r1;
  reg[0:0] r0;

  assign io_out = r3;
  always @(posedge clk) begin
    r3 <= r2;
    r2 <= r1;
    r1 <= r0;
    r0 <= io_in;
  end
endmodule
```

Conditional Register Update


```
class ShiftRegister extends Module {
  val io = new Bundle {
    val in    = UInt(INPUT, 1)
    val shift = Bool(INPUT)
    val out   = UInt(OUTPUT, 1)
  }

  val r0 = Reg(UInt())
  val r1 = Reg(UInt())
  val r2 = Reg(UInt())
  val r3 = Reg(UInt())

  when (io.shift) {
    r0 := io.in
    r1 := r0
    r2 := r1
    r3 := r2
  }
  io.out := r3
}
```



Conditional Register Update with Reset


```
class EnableShiftRegister extends Module {
  val io = new Bundle {
    val in    = UInt(INPUT, 1)
    val shift = Bool(INPUT)
    val out   = UInt(OUTPUT, 1)
  }
  // Register reset to zero
  val r0 = Reg(init = UInt(0, 1))
  val r1 = Reg(init = UInt(0, 1))
  val r2 = Reg(init = UInt(0, 1))
  val r3 = Reg(init = UInt(0, 1))
  when (io.shift) {
    r0 := io.in
    r1 := r0
    r2 := r1
    r3 := r2
  }
  io.out := r3
}
```

inferred width

```
UInt(1)       // decimal 1-bit literal from Scala Int.
UInt("ha")    // hexadecimal 4-bit literal from string.
UInt("o12")   // octal 4-bit literal from string.
UInt("b1010") // binary 4-bit literal from string.
```

specified widths

```
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.
UInt(5, 8)          // unsigned decimal 8-bit literal of type UInt.
```

- write sequential circuit that sums `in` values
- in `chisel-tutorial/problems/Accumulator.scala`
- run `make Accumulator.out` until passing

```
class Accumulator extends Module {
  val io = new Bundle {
    val in  = UInt(INPUT, 1)
    val out = UInt(OUTPUT, 8)
  }

  // flush this out ...

  io.out := UInt(0)
}
```

UInt Operations and Conditional Assignment


```
class BasicALU extends Module {
  val io = new Bundle {
    val a      = UInt(INPUT, 4)
    val b      = UInt(INPUT, 4)
    val opcode = UInt(INPUT, 4)
    val output = UInt(OUTPUT, 4)
  }
  io.output := UInt(0)
  when (io.opcode === UInt(0)) {
    io.output := io.a                   // pass A
  } .elsewhen (io.opcode === UInt(1)) {
    io.output := io.b                   // pass B
  } .elsewhen (io.opcode === UInt(2)) {
    io.output := io.a + UInt(1)         // inc A by 1
  } .elsewhen (io.opcode === UInt(3)) {
    io.output := io.a - UInt(1)         // dec B by 1
  } .elsewhen (io.opcode === UInt(4)) {
    io.output := io.a + UInt(4)         // inc A by 4
  } .elsewhen (io.opcode === UInt(5)) {
    io.output := io.a - UInt(4)         // dec A by 4
  } .elsewhen (io.opcode === UInt(6)) {
    io.output := io.a + io.b            // add A and B
  } .elsewhen (io.opcode === UInt(7)) {
    io.output := io.a - io.b            // sub B from A
  } .elsewhen (io.opcode === UInt(8)) {
    io.output := (io.a < io.b)          // set on A < B
  } .otherwise {
    io.output := (io.a === io.b)        // set on A == B
  }
}
```

- wire `io.output` defaulted to 0 and then
- conditionally reassigned to based on opcode
- unlike registers, wires are required to be defaulted
- wires also allow forward declarations

UInt Operations

Symbol	Operation	Output Type
+	Add	UInt
-	Subtract	UInt
*	Multiply	UInt
/	UInt Divide	UInt
%	Modulo	UInt
~	Bitwise Negation	UInt
^	Bitwise XOR	UInt
&	Bitwise AND	UInt
|	Bitwise OR	Bool
===	Equal	Bool
!=	Not Equal	Bool
>	Greater	Bool
<	Less	Bool
>=	Greater or Equal	Bool
<=	Less or Equal	Bool

```
// extracts the x through y bits of value
val x_to_y = value(x, y)
```

```
// extract the x-th bit from value
val x_of_value = value(x)
```

```
class ByteSelector extends Module {
  val io = new Bundle {
    val in     = UInt(INPUT, 32)
    val offset = UInt(INPUT, 2)
    val out    = UInt(OUTPUT, 8)
  }
  io.out := UInt(0, width = 8)
  when (io.offset === UInt(0)) {
    io.out := io.in(7,0)   // pull out lowest byte
  } .elsewhen (io.offset === UInt(1)) {
    io.out := io.in(15,8)  // pull out second byte
  } .elsewhen (io.offset === UInt(2)) {
    io.out := io.in(23,16) // pull out third byte
  } .otherwise {
    io.out := io.in(31,24) // pull out highest byte
  }
}
```

You concatenating bits using `Cat`:

```
val A   = UInt(width = 32)
val B   = UInt(width = 32)
val bus = Cat(A, B) // concatenate A and B
```

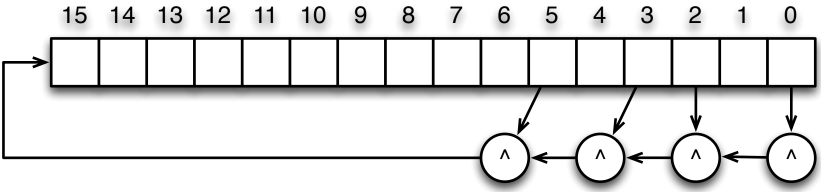
and replicate bits using `Fill`:

```
// Replicate a bit string multiple times.
val usDebt = Fill(3, UInt("hA"))
```

LFSR16 – `problems/lfsr16.scala`

```
class LFSR16 extends Module {
  val io = new Bundle {
    val inc = Bool(INPUT)
    val out = UInt(OUTPUT, 16)
  }
  // ...
  io.out := UInt(0)
}
```

- `reg`, `cat`, `extract`, `^`
- init reg to 1
- updates when `inc` asserted



UInt Bit Inference


```scala
class HiLoMultiplier()
    extends Module {
  val io = new Bundle {
    val A  = UInt(INPUT, 16)
    val B  = UInt(INPUT, 16)
    val Hi = UInt(OUTPUT, 16)
    val Lo = UInt(OUTPUT, 16)
  }
  val mult = io.A * io.B
  io.Lo := mult(15, 0)
  io.Hi := mult(31, 16)
}
```

```verilog
module HiLoMultiplier(
    input [15:0] io_A,
    input [15:0] io_B,
    output[15:0] io_Hi,
    output[15:0] io_Lo);

  wire[15:0] T0;
  wire[31:0] mult; // inferred as 32 bits
  wire[15:0] T1;

  assign io_Lo = T0;
  assign T0 = mult[4'hf:1'h0];
  assign mult = io_A * io_B;
  assign io_Hi = T1;
  assign T1 = mult[5'h1f:5'h10];
endmodule
```

Bit Inference Rules

<table>
<tr><th>Operation</th><th>Result Bit Width</th></tr>
<tr><td><code>Z = X + Y</code></td><td>max(Width(X), Width(Y))</td></tr>
<tr><td><code>Z = X - Y</code></td><td>max(Width(X), Width(Y))</td></tr>
<tr><td><code>Z = X & Y</code></td><td>min(Width(X), Width(Y))</td></tr>
<tr><td><code>Z = X | Y</code></td><td>max(Width(X), Width(Y))</td></tr>
<tr><td><code>Z = X ^ Y</code></td><td>max(Width(X), Width(Y))</td></tr>
<tr><td><code>Z = ~X</code></td><td>Width(X)</td></tr>
<tr><td><code>Z = Mux(C, X, Y)</code></td><td>max(Width(X), Width (Y))</td></tr>
<tr><td><code>Z = X * Y</code></td><td>Width(X) + Width(Y)</td></tr>
<tr><td><code>Z = X << n</code></td><td>Width(X) + n</td></tr>
<tr><td><code>Z = X >> n</code></td><td>Width(X) - n</td></tr>
<tr><td><code>Z = Cat(X, Y)</code></td><td>Width(X) + Width(Y)</td></tr>
<tr><td><code>Z = Fill(n, x)</code></td><td>Width(X) + n</td></tr>
</table>

Bool Type

The Chisel Bool is used to represent the result of logical expressions:

```
val change = io.a === io.b // change gets Bool type
when (change) { // execute if change is true
 ...
}
```

You can instantiate a Bool value like this:

```
val true_value  = Bool(true)
val false_value = Bool(false)
```

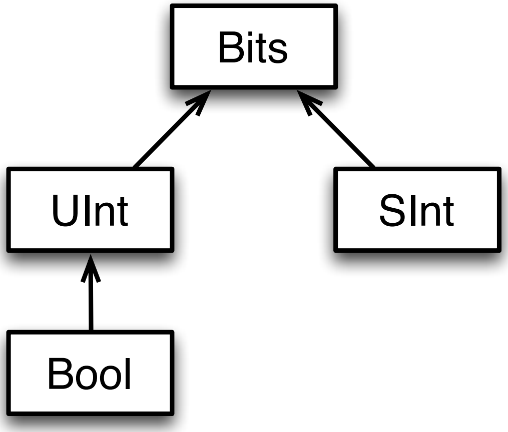
You can cast an UInt to a Bool as follows:

```
val bit = UInt(width = 1) ...
when (bit.toBool) { ... }
```

You can use a Bool as an UInt:

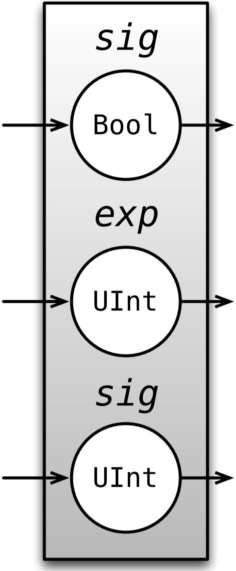
```
val bit = UInt(width = 1) ...
bit := a > b
```

- `SInt` is a signed integer type



Bundles

```
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
```

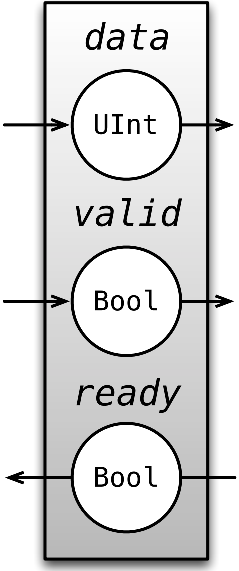


Data object with directions assigned to its members

```
class Decoupled extends Bundle {
  val data  = UInt(INPUT, 32)
  val valid = Bool(OUTPUT)
  val ready = Bool(INPUT)
}
```

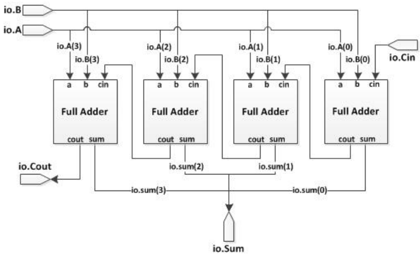
Direction assigned at instantiation time

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



Instantiating Modules

```
// A 4-bit adder with carry in and carry out
class Adder4 extends Module {
  val io = new Bundle {
    val A    = UInt(INPUT, 4)
    val B    = UInt(INPUT, 4)
    val Cin  = UInt(INPUT, 1)
    val Sum  = UInt(OUTPUT, 4)
    val Cout = UInt(OUTPUT, 1)
  }
  // Adder for bit 0
  val Adder0 = Module(new FullAdder())
  Adder0.io.a   := io.A(0)
  Adder0.io.b   := io.B(0)
  Adder0.io.cin := io.Cin
  val s0 = Adder0.io.sum
  // Adder for bit 1
  val Adder1 = Module(new FullAdder())
  Adder1.io.a   := io.A(1)
  Adder1.io.b   := io.B(1)
  Adder1.io.cin := Adder0.io.cout
  val s1 = Cat(Adder1.io.sum, s0)
  ...
  // Adder for bit 3
  val Adder3 = Module(new FullAdder())
  Adder3.io.a   := io.A(3)
  Adder3.io.b   := io.B(3)
  Adder3.io.cin := Adder2.io.cout
  io.Sum  := Cat(Adder3.io.sum, s2)
  io.Cout := Adder3.io.cout
}
```



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

constructing vecs

```
val myVec1 = Vec.fill( <number of elements> ) { <data type> }
val myVec2 = Vec(<elt0>, <elt1>, ...)
```

creating a vec of wires

```
val ufix5_vec10 = Vec.fill(10) { UInt(width = 5) }
```

creating a vec of regs

```
val reg_vec32 = Vec.fill(32){ Reg() }
```

writing

```
reg_vec32(1) := UInt(0)
```

reading

```
val reg5 = reg_vec(5)
```

- add loadability to shift register
- change interface to use vec's

```
class VecShiftRegister extends Module {
  val io = new Bundle {
    val ins   = Vec.fill(4){ UInt(INPUT, 1) }
    val load  = Bool(INPUT)
    val shift = Bool(INPUT)
    val out   = UInt(OUTPUT, 4)
  }
  val delays = Vec.fill(4){ Reg(UInt(width = 4)) }
  when ( ... ) {
  // fill in here ...
  } .elsewhen (io.shift) {
    ...
  }
  io.out := delays(3)
}
```

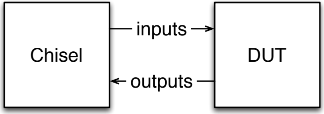
Defining a Tester

```
package Tutorial
import Chisel._

class ByteSelector extends Module {
  val io = new Bundle {
    val in     = UInt(INPUT, 32)
    val offset = UInt(INPUT, 2)
    val out    = UInt(OUTPUT, 8)
  }
  io.out := UInt(0, width=8)
  ...
}

class BSTests(c: ByteSelector) extends Tester(c) {
  val test_in = 12345678
  for (t <- 0 until 4) {
    poke(c.io.in, test_in)
    poke(c.io.offset, t)
    step(1)
    val ref_out = (test_in >> (t * 8)) & 0xFF
    expect(c.io.out, ref_out)
  }
}
```

```
class Tester[T <: Module]
    (val c: T, val isTrace: Boolean = true) {
  var t: Int
  var ok: Boolean
  val rnd: Random
  def reset(n: Int = 1)
  def step(n: Int): Int
  def peek(data: Aggregate): Array[BigInt]
  def peekAt(data: Mem[T], index: Int)
  def peek(data: Bits): BigInt
  def int(x: Boolean): BigInt
  def int(x: Int): BigInt
  def int(x: Bits): BigInt
  def poke(data: Aggregate, x: Array[BigInt])
  def pokeAt(data: Mem[T], index: Int, x: BigInt)
  def poke(data: Bits, x: BigInt)
  def expect (good: Boolean, msg: String): Boolean
  def expect (data: Bits, target: BigInt): Boolean
}
```

which binds a tester to a module and allows users to write tests using the given debug protocol. In particular, users utilize:

- `poke` to set input port and state values,
- `step` to execute the circuit one time unit,
- `peek` to read port and state values, and
- `expect` to compare peeked circuit values to expected arguments.



Simulation Debug Output

```
> cd chisel-tutorial/examples
> make ByteSelector.out
STARTING ../emulator/problems/ByteSelector
---
INPUTS
  INPUT(ByteSelector__io_in.ByteSelector) = 12345678
  INPUT(ByteSelector__io_offset.ByteSelector) = 0
OUTPUTS
  READ OUTPUT(ByteSelector__io_out.ByteSelector) = 78
  EXPECTED: OUTPUT(ByteSelector__io_out.ByteSelector) = 78
  SUCCESS
---
INPUTS
  INPUT(ByteSelector__io_in.ByteSelector) = 12345678
  INPUT(ByteSelector__io_offset.ByteSelector) = 1
OUTPUTS
  READ OUTPUT(ByteSelector__io_out.ByteSelector) = 97
  EXPECTED: OUTPUT(ByteSelector__io_out.ByteSelector) = 97
  SUCCESS
---
...
---
INPUTS
  INPUT(ByteSelector__io_in.ByteSelector) = 12345678
  INPUT(ByteSelector__io_offset.ByteSelector) = 3
OUTPUTS
  READ OUTPUT(ByteSelector__io_out.ByteSelector) = 0
  EXPECTED: OUTPUT(ByteSelector__io_out.ByteSelector) = 0
  SUCCESS
PASSED   // Final pass assertion
[success] Total time: 26 s, ...
```

Testbench Ingredients

In particular, users utilize:

- `poke` to set input port and state values,
- `step` to execute the circuit one time unit,
- `peek` to read port and state values, and
- `expect` to compare peeked circuit values to expected arguments.

- write a testbench for MaxN

```
class MaxN(val n: Int, val w: Int)
    extends Module {

  def Max2(x: UInt, y: UInt) =
    Mux(x > y, x, y)

  val io = new Bundle {
    val ins = Vec.fill(n){ UInt(INPUT, w) }
    val out = UInt(OUTPUT, w)
  }
  io.out := io.ins.reduceLeft(Max2)
}
```

```
// returns random int in 0..lim-1
val x = rnd.nextInt(lim)
```

```
class MaxNTests(c: MaxN) extends
     Tester(c) {
  for (i <- 0 until 10) {
    for (j <- 0 until c.n) {
      // FILL THIS IN HERE
      poke(c.io.ins(0), 0)
    }
    // FILL THIS IN HERE
    step(1)
    expect(c.io.out, 1)
  }
}
```

```
class MemorySearch extends Module {
  val io = new Bundle {
    val target  = UInt(INPUT,  4)
    val en      = Bool(INPUT)
    val address = UInt(OUTPUT, 3)
    val done    = Bool(OUTPUT)
  }
  val index  = Reg(init = UInt(0, width = 3))
  val list   = Vec(UInt(0), UInt(4), UInt(15), UInt(14),
                   UInt(2), UInt(5), UInt(13)){ UInt(width = 4) }
  val memVal = list(index)
  val done   = !io.en && ((memVal === io.target) || (index === UInt(7)))
  when (io.en) {
    index := UInt(0)
  } .elsewhen (done === Bool(false)) {
    index := index + UInt(1)
  }
  io.done    := done
  io.address := index
}
```

RAM is supported using the `Mem` construct

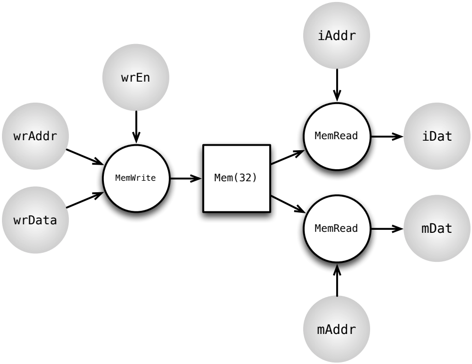
```
val m = Mem(Bits(width = 32), 32)
```

where

- writes to Mems are positive-edge-triggered
- reads are either combinational or positive-edge-triggered
- ports are created by applying a `UInt` index

32-entry Register File

```
val regs = Mem(Bits(width = 32), 32)
when (wrEn) {
  regs(wrAddr) := wrData
}
val iDat = regs(iAddr)
val mDat = regs(mAddr)
```

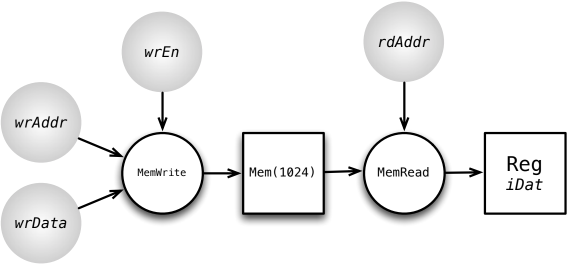


```
class DynamicMemorySearch extends Module {
  val io = new Bundle {
    val isWr   = Bool(INPUT)
    val wrAddr = UInt(INPUT,  3)
    val data   = UInt(INPUT,  4)
    val en     = Bool(INPUT)
    val target = UInt(OUTPUT, 3)
    val done   = Bool(OUTPUT)
  }
  val index  = Reg(init = UInt(0, width = 3))
  val memVal = ...
  val done = !io.en && ((memVal === io.data) || (index === UInt(7)))
  // ...
  when (io.en) {
    index := UInt(0)
  } .elsewhen (done === Bool(false)) {
    index := index + UInt(1)
  }
  io.done   := done
  io.target := index
}
```

Sequential read ports are inferred when:

- optional parameter `seqRead` is set and
- read address is a reg

```
val ram1r1w = Mem(UInt(width = 32), 1024, seqRead = true)
val reg_raddr = Reg(UInt())
when (wen) { ram1r1w(waddr) := wdata }
when (ren) { reg_raddr := raddr }
val rdata = ram1r1w(reg_raddr)
```

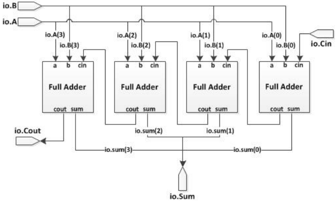


Stack


```
class Stack(depth: Int) extends Module {
  val io = new Bundle {
    val dataIn  = UInt(INPUT,  32)
    val dataOut = UInt(OUTPUT, 32)
    val push    = Bool(INPUT)
    val pop     = Bool(INPUT)
    val en      = Bool(INPUT)
  }
  // declare the memory for the stack
  val stack_mem = Mem(UInt(width = 32), depth, seqRead = false)
  val sp = Reg(init = UInt(0, width = log2Up(depth)))
  val dataOut = Reg(init = UInt(0, width = 32))
  // Push condition - make sure stack isn't full
  when(io.en && io.push && (sp != UInt(depth-1))) {
    stack_mem(sp + UInt(1)) := io.dataIn
    sp := sp + UInt(1)
  }
  // Pop condition - make sure the stack isn't empty
  .elsewhen(io.en && io.pop && (sp > UInt(0))) {
    sp := sp - UInt(1)
  }
  when(io.en) {
    dataOut := stack_mem(sp)
  }
  io.dataOut := dataOut
}
```

Scripting Hardware Generation

```
// A n-bit adder with carry in and carry out
class Adder(n: Int) extends Module {
  val io = new Bundle {
    val A    = UInt(INPUT, n)
    val B    = UInt(INPUT, n)
    val Cin  = UInt(INPUT, 1)
    val Sum  = UInt(OUTPUT, n)
    val Cout = UInt(OUTPUT, 1)
  }
  // create a vector of FullAdders
  val FAs   = Vec.fill(n){ Module(new FullAdder()).io }
  val carry = Vec.fill(n+1){ UInt(width = 1) }
  val sum   = Vec.fill(n){ Bool() }

  // first carry is the top level carry in
  carry(0) := io.Cin

  // wire up the ports of the full adders
  for(i <- 0 until n) {
    FAs(i).a    := io.A(i)
    FAs(i).b    := io.B(i)
    FAs(i).cin := carry(i)
    carry(i+1) := FAs(i).cout
    sum(i)     := FAs(i).sum.toBool()
  }
  io.Sum  := sum.toBits().toUInt()
  io.Cout := carry(n)
}
```



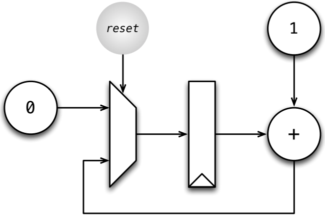
- **Abstractly**: Chisel is a framework for *programmatically* generating circuitry.
- **Less Abstractly**: Chisel is a software library for creating and connecting circuit components to form a circuit graph.
- **Concretely**: Chisel is a DSL embedded in Scala for creating and connecting circuit components, with tools for simulation and translation to Verilog.

** based on slides by my PhD student Patrick Li*

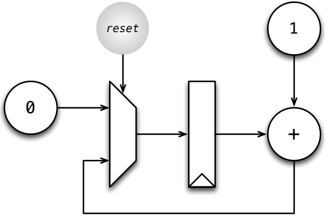
- Classes are provided for circuit components:
  - `Register(name)`
  - `Adder()`
  - `Multiplexor()`
  - `Wire(name)`
  - `Constant(value)`

  and `new` used to construct components and `connect` used to wire them together:
  - `new Register(name)`
  - ...
  - `connect(input, output)`

```
def main(args: Array[String]) = {
  // Create Components
  val reset       = new Wire("reset");
  val counter     = new Register("counter");
  val adder       = new Adder();
  val multiplexor = new Multiplexor();
  val one         = new UInt(1);
  val zero        = new UInt(0);

  // Connect Components
  connect(multiplexor.choice, reset);
  connect(multiplexor.in_a, zero.out);
  connect(multiplexor.in_b, adder.out);
  connect(counter.in, multiplexor.out);
  connect(adder.in_a, counter.out);
  connect(adder.in_b, one.out);

  // Produce Verilog
  generate_verilog(counter);
}
```



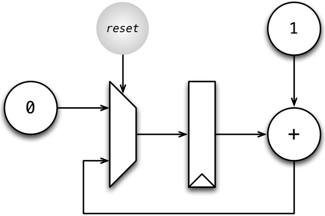
```scala
def main(args: Array[String]) = {
  // Create Components
  val reset       = new Wire("reset");
  val counter     = new Register("counter");
  val adder       = new Adder();
  val multiplexor = new Multiplexor();
  val one         = new UInt(1);
  val zero        = new UInt(0);

  // Connect Components
  connect(multiplexor.choice, reset);
  connect(multiplexor.in_a, zero.out);
  connect(multiplexor.in_b, adder.out);
  connect(counter.in, multiplexor.out);
  connect(adder.in_a, counter.out);
  connect(adder.in_b, one.out);

  // Produce Verilog
  generate_verilog(counter);
}
```



- using Scala to programmatically generate hardware
- can use full power of Scala (loops, arrays, conditionals, ...)

What if Chisel was a Scala Library?


```
def main(args: Array[String]) = {
  // Create Components
  val reset       = new Wire("reset");
  val counter     = new Register("counter");
  val adder       = new Adder();
  val multiplexor = new Multiplexor();
  val one         = new UInt(1);
  val zero        = new UInt(0);

  // Connect Components
  connect(multiplexor.choice, reset);
  connect(multiplexor.in_a, zero.out);
  connect(multiplexor.in_b, adder.out);
  connect(counter.in, multiplexor.out);
  connect(adder.in_a, counter.out);
  connect(adder.in_b, one.out);

  // Produce Verilog
  generate_verilog(counter);
}
```



- but Scala is pretty Verbose, how can we do better?

Functional Composition of Adder


```
def main(args: Array[String]) = {
  // Create Components
  val reset       = new Wire("reset");
  val counter     = new Register("counter");
  val adder       = new Adder();
  val multiplexor = new Multiplexor();
  val one         = new UInt(1);
  val zero        = new UInt(0);

  // Connect Components
  connect(multiplexor.choice, reset);
  connect(multiplexor.in_a, zero.out);
  connect(multiplexor.in_b, adder.out);
  connect(counter.in, multiplexor.out);
  connect(adder.in_a, counter.out);
  connect(adder.in_b, one.out);

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset       = new Wire("reset");
  val counter     = new Register("counter");
  val multiplexor = new Multiplexor();
  val one         = new UInt(1);
  val zero        = new UInt(0);

  // Connect Components
  connect(multiplexor.choice, reset);
  connect(multiplexor.in_a, zero.out);
  connect(multiplexor.in_b,
          make_adder(one.out, counter.out));
  connect(counter.in, multiplexor.out);

  // Produce Verilog
  generate_verilog(counter);
}
```

Functional Composition of Multiplexor


```
def main(args: Array[String]) = {
  // Create Components
  val reset       = new Wire("reset");
  val counter     = new Register("counter");
  val multiplexor = new Multiplexor();
  val one         = new UInt(1);
  val zero        = new UInt(0);

  // Connect Components
  connect(multiplexor.choice, reset);
  connect(multiplexor.in_a, zero.out);
  connect(multiplexor.in_b,
          make_adder(one.out, counter.out));
  connect(counter.in, multiplexor.out);

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");
  val one     = new UInt(1);
  val zero    = new UInt(0);

  // Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      zero.out
      make_adder(one.out, counter.out)));

  // Produce Verilog
  generate_verilog(counter);
}
```

Functional Composition of UInt Creation


```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");
  val one     = new UInt(1);
  val zero    = new UInt(0);

  // Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      zero.out
      make_adder(one.out, counter.out)));

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0),
      make_adder(UInt(1), counter.out)));

  // Produce Verilog
  generate_verilog(counter);
}
```

Overload Addition Operator

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0),
      make_adder(UInt(1), counter.out)));

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out));

  // Produce Verilog
  generate_verilog(counter);
}
```

operator overloading

Introduce Connect Infix Operator

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  connect(counter.in,
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out));

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  counter.in :=
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out);

  // Produce Verilog
  generate_verilog(counter);
}
```

operator overloading

Automatically Create Multiplexors


```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  counter.in :=
    make_multiplexor(reset,
      UInt(0),
      UInt(1) + counter.out);

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }

  // Produce Verilog
  generate_verilog(counter);
}
```

dynamic scoping, closures, object-orientation

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire("reset");
  val counter = new Register("counter");

  // Connect Components
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }

  // Produce Verilog
  generate_verilog(counter);
}
```

$\Rightarrow$

```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire();
  val counter = new Register();

  // Connect Components
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }

  // Produce Verilog
  generate_verilog(counter);
}
```

introspection

Abstract Counter


```
def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire();
  val counter = new Register();

  // Connect Components
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }

  // Produce Verilog
  generate_verilog(counter);
}
```

⇒

```
def make_counter(reset: Boolean) = {
  val counter = new Register();
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }
  counter
}

def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire();
  val counter = make_counter(reset);

  // Produce Verilog
  generate_verilog(counter);
}
```

```
def make_counter(reset: Boolean) = {
  val counter = new Register();
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }
  counter
}

def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire();
  val counter = make_counter(reset);

  // Produce Verilog
  generate_verilog(counter);
}
```

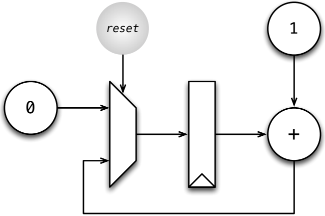
$\Rightarrow$

```
def make_counter() = {
  val counter = new Register();
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }
  counter
}

def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire();
  val counter =
    withReset(reset) {
      make_counter(reset);
    }
  // Produce Verilog
  generate_verilog(counter);
}
```

dynamic scoping

Looks “Behavioral” but ...


```
def make_counter() = {
  val counter = new Register();
  when (reset) {
    counter.in := UInt(0);
  } .otherwise {
    counter.in := UInt(1) + counter.out;
  }
  counter
}

def main(args: Array[String]) = {
  // Create Components
  val reset   = new Wire();
  val counter =
    withReset(reset) {
      make_counter(reset);
    }
  // Produce Verilog
  generate_verilog(counter);
}
```



- every construct actually creates a concrete circuit
- know cost of everything
- layered and can choose level of abstraction

functional programming

Crucial

- Type Inference
- Infix Operator Overloading
- Lightweight Closures
- Dynamic Scoping
- Introspection (or Simple Macros)
- Functional Programming

Even Better with

- Object Orientation
- Powerful Macros

- Graceful Introduction of PL Concepts
- Orthogonal Concepts Mixed in Serendipitous Ways
- Seamless Integration of Domain
- Strong Performance Model

Exploring Stanza by Patrick Li – `www.lbstanza.org`

- Gradual Typing
- Multimethod System
- Macro System

Mul Lookup Table Problem – `Mul.scala`

- write 16x16 multiplication table using `Vec`

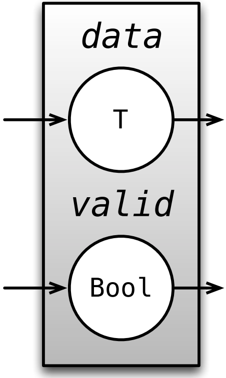
```
class Mul extends Module {
  val io = new Bundle {
    val x   = UInt(INPUT, 4)
    val y   = UInt(INPUT, 4)
    val z   = UInt(OUTPUT, 8)
  }
  val muls = new ArrayBuffer[UInt]()

  // flush this out ...

  io.z := UInt(0)
}
```

hint:

```
val tab = Vec(muls)
io.z := tab(Cat(io.x, io.y))
```

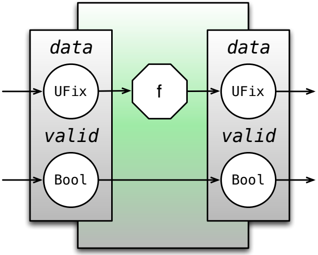
Valid Wrapper

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

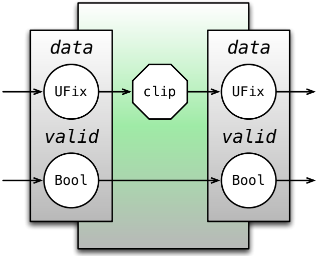


Function Filters

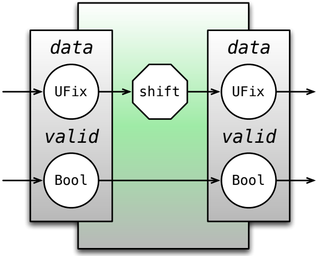

```
abstract class Filter[T <: Data](dtype: T) extends Module {
  val io = new Bundle {
    val in  = Valid(dtype).asInput
    val out = Valid(dtype).asOutput
} }

class FunctionFilter[T <: Data](dtype: T, f: T => T) extends Filter(dtype) {
  io.out.valid := io.in.valid
  io.out.bits  := f(io.in)
}
```



```
def clippingFilter[T <: Bits](limit: Int, dtype: T) =
  new FunctionFilter(dtype, x => min(limit, max(-limit, x)))
```



```
def shiftingFilter[T <: Bits](shift: Int, dtype: T) =
  new FunctionFilter(dtype, x => x >> shift)
```

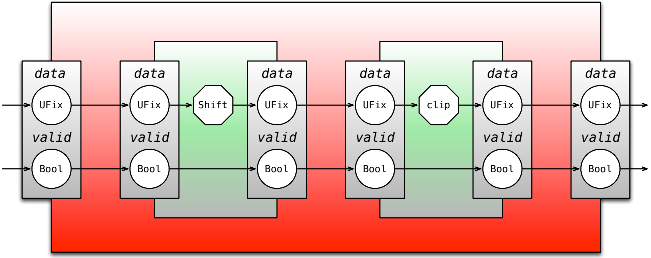


- using ovars for outputs
- need to check outputs directly using `litValue`

```
class GCDTests(c: GCD) extends Tester(c) {
  val (a, b, z) = (64, 48, 16)
  do {
    poke(c.io.a, a)
    poke(c.io.b, b)
    step(1)
  } while (t <= 1 || peek(c.io.v) == 0)
  expect(c.io.z, z)
}
```

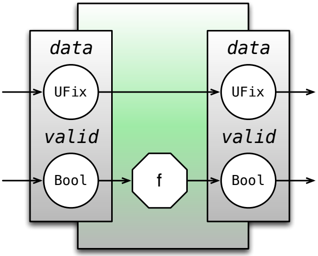
Chained Filter


```
class ChainedFilter[T <: Num](dtype: T) extends Filter(dtype) = {
  val shift   = Module(new ShiftFilter(2, dtype))
  val clipper = Module(new ClippingFilter(1 << 7, dtype))
  io.in          <> shift.io.in
  shift.io.out   <> clipper.io.in
  clipper.io.out <> io.out
}
```



Predicate Filter


```
class PredicateFilter[T <: Data](dtype: T, f: T => Bool)
    extends Filter(dtype) {
  io.out.valid := io.in.valid && f(io.in.bits)
  io.out.bits  := io.in.bits
}
```

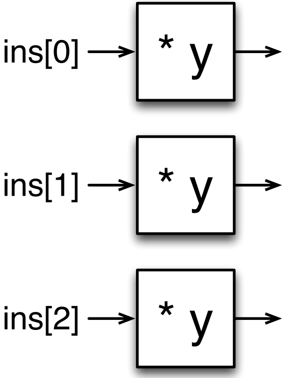


- write filter that lets only even single digit numbers through

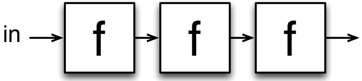
```
object SingleFilter {
  def apply[T <: UInt](dtype: T) = // FILL IN FUNCTION BELOW
    Module(new PredicateFilter(dtype, (x: T) => Bool(false)))
}

object EvenFilter {
  def apply[T <: UInt](dtype: T) = // FILL IN FUNCTION BELOW
    Module(new PredicateFilter(dtype, (x: T) => Bool(false)))
}

class SingleEvenFilter[T <: UInt](dtype: T) extends Filter(dtype) {
  // FILL IN CONSTRUCTION AND WIRING
  io.out := UInt(0)
}
```

Functional Composition

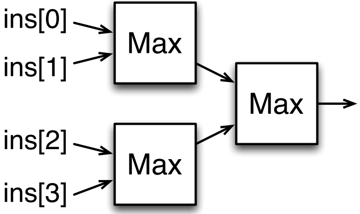
`Map(ins, x => x * y)`



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



`Reduce(ins, Max)`



```
object FloDelays {
  def apply(x: Flo, n: Int): List[Flo] =
    if (n <= 1) List(x) else x :: FloDelays(RegNext(x), n-1)
}
object FloFIR {
  def apply(ws: Seq[Flo], x: T): T =
    (ws, FloDelays(x, ws.length)).zipped.map( _ * _ ).reduce( _ + _ )
}
class FIR extends Module {
  val io = new Bundle { val x = Flo(INPUT); val z = Flo(OUTPUT) }
  val ws = Array(Flo(0.25), Flo(0.75))
  io.z  := FloFIR(ws, io.x)
}
```

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k]$$

```
object Delays {
  def apply[U <: Data](x: U, n: Int): List[U] =
    if (n <= 1) List(x) else x :: Delays(RegNext(x), n-1)
}
object GenFIR {
  def apply[T <: Data with Num[T]](ws: Seq[T], x: T): T =
    (ws, Delays(x, ws.length)).zipped.map( _ * _ ).reduce( _ + _ )
}
class FIR extends Module {
  val io = new Bundle { val x = Flo(INPUT); val z = Flo(OUTPUT) }
  val ws = Array(Flo(0.25), Flo(0.75))
  io.z  := GenFIR(ws, io.x)
}
```

$$y[n] = \sum_{k=0}^{N-1} x[n-k]h[k]$$

Bits Properities	`log2Up`, `log2Down`, `isPow2`, `PopCount`
Numeric Utilities	`LFSR16`, `Reverse`, `FillInterleaved`
Stateful Functions	`ShiftRegister`, `Counter`
Priority Encoding Functions	`UIntToOH`, `OHToUInt`, `Mux1H`
Priority Encoders	`PriorityEncoder`, `PriorityEncoderOH`
Vec Construction	`Vec.fill`, `Vec.tabulate`
Vec Functional	`forall`, `exists`, `contains`, ...
Queues and Pipes	`Decoupled`, `Queue`, `Valid`, `Pipe`
Arbiters	`ArbiterIO`, `Arbiter`, `RRArbiter`

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

```
class QueueIO[T <: Data](type: T, entries: Int) extends Bundle {
  val enq   = Decoupled(data.clone).flip
  val deq   = Decoupled(data.clone)
  val count = UFix(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
```

Multiple Clock Domains

Clocks are first class and take a name argument:

```
class Clock (val name: String) extends Node {
}
```

and when constructed define a clock at top-level with the given name:

```
val clkA = new Clock("A")
```

There is a builtin implicit clock that state elements use by default:

```
class Module {
  def clock(): Clock
  def reset(): Bool
  ...
}
```

The clock for state elements and modules can be specified:

```
Reg(... explClock: Clock = clock())
Mem(... explClock: Clock = clock())
Module(... explClock: Clock = clock())
```

For example, a register can be created in a different clock domain as follows:

```
val reg = Reg(UInt(), explClock = clock2)
```

The most general technique to send data between domains is using an asynchronous queue:

```
class AsyncQueue[T <: Data]
    (dataType: T, depth: Int, enq_clk: Clock, deq_clock: Clock) extends Module
```

Using these queues, we can then move a signalA from clock domains clockA to signalB in clockB:

```
val queue = new AsyncQueue(Uint(width = 32), 2, clockA, clockB)
fifo.enq.bits := signalA
signalB       := fifo.deq.bits
fifo.valid    := condA
fifo.ready    := condB
...
```

```
class MultiClockDomain extends Module {
  val io = new Bundle {
    val start = Bool(INPUT)
    val sum = Decoupled(UInt(OUTPUT))
  }
  val fastClock = new Clock()
  val slowClock = new Clock()
  ...
}

class MultiClockDomainTests(c: MultiClockDomain)
    extends Tester(c, Array(c.io)) {
  val clocks = new HashMap[Clock, Int]
  clocks(Module.implicitClock) = 2
  clocks(c.fastClock) = 4
  clocks(c.slowClock) = 6
  setClocks(clocks)
  ...
}
```

directory structure

```
Hello/
  build.sbt   # scala configuration file
  Hello.scala # your source file
```

Writing Your Source File

```
package Hello
import Chisel._

class Hello extends Module {
  val io = new Bundle {
    val out = UInt(OUTPUT, 8) }
  io.out := UInt(33)
}

class HelloTests(c: Hello) extends Tester(c) {
  step(1)
  expect(c.io.out, 33)
}

object Hello {
  def main(args: Array[String]): Unit = {
    val args = Array("--backend", "c", "--genHarness", "--compile", "--test")
    chiselMainTest(args, () => Module(new Hello())) {
      c => new HelloTests(c) }
} }
```

```
scalaVersion := "2.10.2"

addSbtPlugin("com.github.scct" % "sbt-scct" % "0.2")

libraryDependencies +=
  "edu.berkeley.cs" %% "chisel" % "latest.release"
```

Producing C++

```
sbt run "--backend c"
```

Producing Verilog

```
sbt run "--backend v"
```

Running the Chisel Tests

```
sbt run "--backend c --compile --test --genHarness"
```

chiselMain(Test) Command Line Arguments

```
sbt
sbt> compile          // compiles Chisel Scala code
sbt> run              // compile and run Chisel Scala Code
sbt> run --backend c // produces C++ files
sbt> exit
```

with a complete set of command line arguments being:

`--backend v`	generate verilog
`--backend c`	generate C++ (default)
`--vcd`	enable vcd dumping
`--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

Make Your Own Project

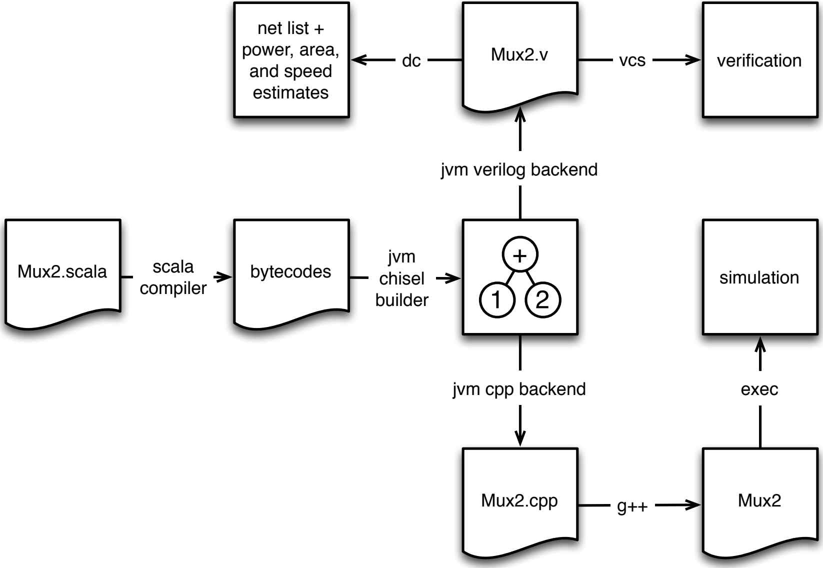
set hello project up

```
cd ~
mkdir hello
cp -r ~/chisel-tutorial/hello/* hello
cd hello
sbt run
```

make a change

- make output a function of an new input

Chisel Workflow



- during simulation
  - `printf` prings the formatted string to the console on rising clock edges
  - `sprintf` returns the formatted string as a bit vector
- format specifiers are
  - `%b` – binary number
  - `%d` – decimal number
  - `%x` – hexidecimal number
  - `%e` – floating point number in scientific notation
  - `%s` – string consisting of a sequence of 8-bit extended ASCII chars
  - `%%` – specifies a literal

the following prints the line `"0x4142 16706 AB"` on cycles when `c` is true:

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

- 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))
```

Installation

- on mac install:
  - XCODE console tools
- on windows install:
  - cygwin
- everywhere install:
  - git
  - g++ version 4.0 or later
  - java
- everywhere
  - git clone https://github.com/ucb-bar/chisel-tutorial.git

`https://chisel.eecs.berkeley.edu/documentation.html`

getting started `getting-started.pdf`
tutorial `tutorial.pdf`
manual `manual.pdf`

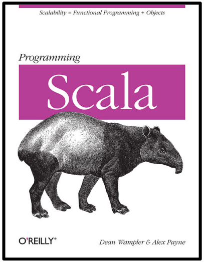
`https://github.com/ucb-bar/chisel/`

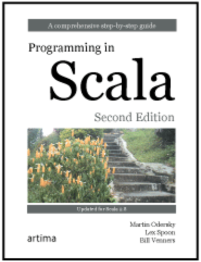
setup `readme.md`
utils `src/main/scala/ChiselUtils.scala`

`https://chisel.eecs.berkeley.edu/download.html`

sodor `https://github.com/ucb-bar/riscv-sodor/`

Scala Resources





audio processing	`Echo.scala`
image processing	`Darken.scala`
risc processor	`Risc.scala`
game of life	`Life.scala`
router	`Router.scala`
map/reduce	`FIR.scala`
network	
decoupled filter	
cryptography	
serial multiplier	
pong	

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`

- **Arrangements** – HPCA folks
- **USB Sticks** – Albert Magyar + Jim Lawson
- **Bootcamp Materials** – JB, Vincent Lee, Stephen Twigg, Huy Vo
- **Funding** – Department of Energy, Department of Defense, StarNet, C-Far, LBNL, Intel, Google, LG, Nvidia, Samsung, Oracle, Huawei