Chisel Manual

Jonathan Bachrach, Krste Asanović, Huy Vo
EECS Department, UC Berkeley
{jrb|krste|huytbvo}@eecs.berkeley.edu

October 26, 2012

1 Introduction

This document is a manual for *Chisel* (Constructing Hardware In a Scala Embedded Language). Chisel is a hardware construction language embedded in the high-level programming language Scala. A separate Chisel tutorial document provides a gentle introduction to using Chisel, and should be read first. This manual provides a comprehensive overview and specification of the Chisel language, which is really only a set of special class definitions, predefined objects, and usage conventions within Scala. When you write a Chisel program you are actually writing a Scala program. In this manual, we presume that you already understand the basics of Scala. If you are unfamiliar with Scala, we recommend you consult one of the excellent Scala books ([3], [2]).

2 Nodes

Any hardware design in Chisel is ultimately represented by a graph of node objects. User code in Chisel generate this graph of nodes, which is then passed to the Chisel backends to be translated into Verilog or C++ code. Nodes are defined as follows:

```
class Node {
  // name assigned by user or from introspection
  var name: String = ""
  // incoming graph edges
  def inputs: ArrayBuffer[Node]
  // outgoing graph edges
  def consumers: ArrayBuffer[Node]
  // node specific width inference
  def inferWidth: Int
  // get width immediately inferrable
  def getWidth: Int
  // get first raw node
  def getRawNode: Node
  // convert to raw bits
  def toBits: Bits
  // convert to raw bits
  def fromBits(x: Bits): this.type
  // return lit value if inferrable else null
  def litOf: Lit
  // return value of lit if litOf is non null
  def litValue(default: BigInt = BigInt(-1)):
      BigInt
}
```

The uppermost levels of the node class hierarchy are shown in Figure 1. The basic categories are:

Lit – constants or literals,

Op – logical or arithmetic operations,

Updateable – conditionally updated nodes,

Data – typed wires or ports,

Reg – positive-edge-triggered registers, and

Mem – memories.

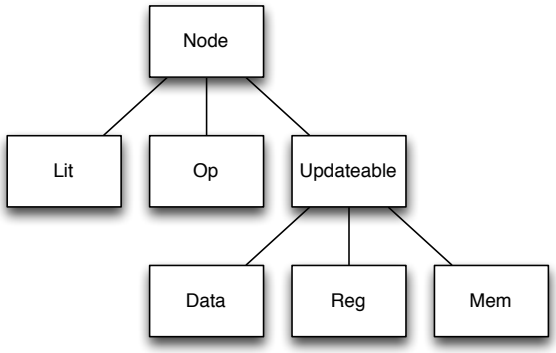

Figure 1: Node hierarchy.

3 Lits

Raw literals are represented as `Lit` nodes defined as follows:

```
class Lit extends Node {
  // original value
  val inputVal: BigInt
}
```

Raw literals contain a collection of bits. Users do not create raw literals directly, but instead use type constructors defined in Section 5.

4 Ops

Raw operations are represented as `Op` nodes defined as follows:

```
class Op extends Node {
  // op name used during emission
  val op: String
}
```

Ops compute a combinational function of their inputs.

5 Types

A Chisel graph representing a hardware design contains *raw* and *type* nodes. The Chisel type system is maintained separately from the underlying Scala type system, and so type nodes are interspersed between raw nodes to allow Chisel to check and respond to Chisel types. Chisel type nodes are erased before the hardware design is translated into C++ or Verilog. The `getRawNode` operator defined in the base Node class, skips type nodes and returns the first raw node found. Figure 2 shows the built-in Chisel type hierarchy, with `Data` as the topmost node.

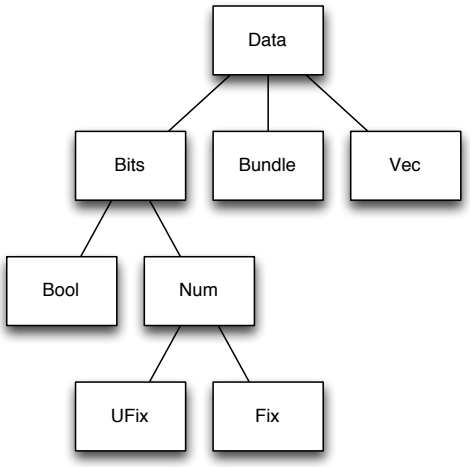

Figure 2: Chisel type hierarchy.

Built-in scalar types include `Bits`, `Bool`, `Fix`, and `UFix` and built-in aggregate types `Bundle` and `Vec` allow the user to expand the set of Chisel datatypes with collections of other types.

`Data` itself is a node:

```
abstract class Data extends Node {
  override def clone(): this.type =
    this.getClass.newInstance.
      asInstanceOf[this.type]
  // simple conversions
  def toFix: Fix
  def toUFix: UFix
  def toBool: Bool
  def toBits: Bits
  // flatten out to leaves of tree
  def flatten: Array[(String, Data)]
  // port direction if leaf
  def dir: PortDir
  // change dir to OUTPUT
  def asOutput: this.type
  // change dir to INPUT
  def asInput: this.type
  // change polarity of dir
  def flip: this.type
  // assign to input
  def :=[T <: Data](t: T)
  // bulk assign to input
  def <>(t: Data)
}
```

The Data class has methods for converting between types and for delegating port methods to its single input. We will discuss ports in Section 10. Finally, users can override the `clone` method in their own type nodes (e.g., bundles) in order to reflect construction parameters that are necessary for cloning.

Data nodes can be used for four purposes:

- **types** – `UFix(width = 8)` – record intermediate types in the graph specifying at minimum bitwidth (described in this section),
- **wires** – `UFix(width = 8)` – serve as forward declarations of data allowing future conditional updates (described in Section 6),
- **ports** – `UFix(dir = OUTPUT, width = 8)` – are specialized wires defining component interfaces, and additionally specify direction (described in Section 10), and
- **literals** – `UFix(1)` or `UFix(1, 8)` – can be constructed using type object constructors specifying their value and optional width.

5.1 Bits

In Chisel, a raw collection of bits is represented by the `Bits` type defined as follows:

```
object Bits {
  def apply(dir: PortDir = null,
            width: Int = -1): Bits
  // create literal from BigInt or Int
  def apply(value: BigInt, width: Int = -1): Bits
  // create literal from String using
  // base_char digit+ string format
  def apply(value: String, width: Int = -1): Bits
}

class Bits extends Data with Updateable {
  // bitwise-not
  def unary_~(): Bits
  // bitwise-and
  def &  (b: Bits): Bits
  // bitwise-or
  def |  (b: Bits): Bits
  // bitwise-xor
  def ^  (b: Bits): Bits
  // and-reduction
  def andR(): Bool
  // or-reduction
  def orR():  Bool
  // xor-reduction
  def xorR():  Bool
  // logical NOT
  def unary_!(): Bool
  // logical AND
  def && (b: Bool): Bool
  // logical OR
  def || (b: Bool): Bool
  // equality
  def ===(b: Bits): Bool
  // inequality
  def != (b: Bits): Bool
  // logical left shift
  def << (b: UFix): Bits
  // logical right shift
  def >> (b: UFix): Bits
  // concatenate
  def ## (b: Bits): Bits
  // extract single bit, LSB is 0
  def apply(x: Int): Bits
  // extract bit field from end to start bit pos
  def apply(hi: Int, lo: Int): Bits
}

def Cat[T <: Data](elt: T, elts: T*): Bits
```

Bits has methods for simple bit operations. Note that `##` is binary concatenation, while `Cat` is an n-ary concatentation. To avoid colliding with Scala's builtin `==`, Chisel's bitwise comparison is named `===`.

A field of width `n` can be created from a single bit using `Fill`:

```
def Fill(n: Int, field: Bits): Bits
```

and two inputs can be selected using `Mux`:

```
def Mux[T <: Data](sel: Bits, cons: T, alt: T): T
```

Constant or literal values are expressed using Scala integers or strings passed to constructors for the types:

```
Bits(1)       // decimal 1-bit lit from Scala Int.
Bits("ha")    // hex 4-bit lit from string.
Bits("o12")   // octal 4-bit lit from string.
Bits("b1010") // binary 4-bit lit from string.
```

producing a `Lit` as shown in the leftmost subfigure of Figure 3.

Operations return an actual operator node with a type node combining the input type nodes. See Figure 3 for successively more complicated examples.

5.2 Bools

Boolean values are represented as `Bools`:

```
object Bool {
  def apply(dir: PortDir = null): Bool
  // create literal
  def apply(value: Boolean): Bool
}

class Bool extends Bits
```

`Bool` is equivalent to `Bits(width = 1)`.

5.3 Nums

`Num` is a type node which defines arithmetic operations:

```
class Num extends Bits {
  // Negation
  def unary_-(): Bits
  // Addition
  def +(b: Num): Num
  // Subtraction
  def -(b: Num): Num
  // Multiplication
  def *(b: Num): Num
  // Greater than
  def >(b: Num): Bool
  // Less than
  def <(b: Num): Bool
  // Less than or equal
  def <=(b: Num): Bool
```

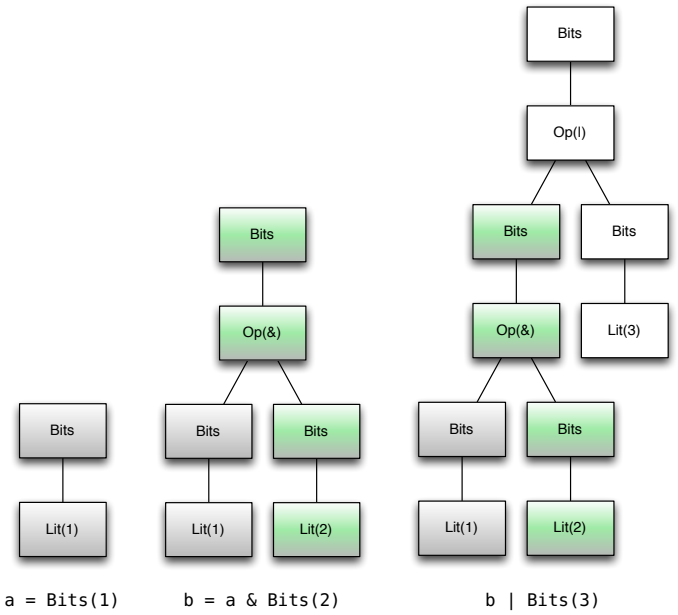

Figure 3: Chisel Op/Lit graphs constructed with algebraic expressions showing the insertion of type nodes.

```
  // Greater than or equal
  def >=(b: Num): Bool
}
```

Signed and unsigned integers are considered subsets of fixed-point numbers and are represented by types `Fix` and `UFix` respectively:

```
object Fix {
  def apply (dir: PortDir = null,
             width: Int = -1): Fix
  // create literal
  def apply (value: BigInt, width: Int = -1): Fix
  def apply (value: String, width: Int = -1): Fix
}

class Fix extends Num

object UFix {
  def apply(dir: PortDir = null,
            width: Int = -1): UFix
  // create literal
  def apply(value: BigInt, width: Int = -1): UFix
  def apply(value: String, width: Int = -1): UFix
}

class UFix extends Num {
  // arithmetic right shift
  override def >> (b: UFix): Fix
}
```

Signed fixed-point numbers, including integers, are represented using two's-complement format.

5.4 Bundles

Bundles group together several named fields of potentially different types into a coherent unit, much like a `struct` in C:

```
class Bundle extends Data {
  // shallow named bundle elements
  def elements: ArrayBuffer[(String, Data)]
}
```

The name and type of each element in a Bundle can be obtained with the `elements` method, and the `flatten` method returns the elements at the leaves for nested aggregates. Users can define new bundles by subclassing `Bundle` as follows:

```
class MyFloat extends Bundle {
  val sign        = Bool()
  val exponent    = Bits(width = 8)
  val significand = Bits(width = 23)
}
```

Elements are accessed using Scala field access:

```
val x  = new MyFloat()
val xs = x.sign
```

The names given to a bundle's elements when they are emitted by a C++ or Verilog backend are obtained from their bundle field names, using Scala introspection.

5.5 Vecs

Vecs create an indexable vector of elements:

```
object Vec {
  def apply[T <: Data](n: Int)(type: => T): Vec[T]
  def apply[T <: Data](elts: Seq[T])(type: => T):
      Vec[T]
  def apply[T <: Data](elts: T*)(type: => T):
      Vec[T]
}

class Vec[T <: Data](n: Int, val type: () => T)
    extends Data {
  def apply(idx: UFix): T
  def apply(idx: Int): T
}
```

with `n` elements of type defined with the `gen` thunk. Users can access elements statically with an `Int` index or dynamically using a `UFix` index, where dynamic access creates a virtual type node (representing a read "port") that records the read using the given address. In either case, users can wire to the result of a read as follows:

```
v(a) := d
```

Read-only memories can be expressed as Vecs of literals:

```
val rom = Vec(UFix(3), UFix(7), UFix(4), UFix(0))
    { UFix(width=3) }
val dout = rom(addr)
```

5.6 Bit Width Inference

Users are required to set bit widths of ports and registers, but otherwise, bit widths on nodes are automatically inferred unless set manually by the user (using `Extract` or `Cat`). The bit-width inference engine starts from the graph's input ports and calculates node output bit widths from their respective input bit widths according to the following set of rules:

operation	bit width
`z = x + y`	`wz = max(wx, wy)`
`z = x - y`	`wz = max(wx, wy)`
`z = x & y`	`wz = max(wx, wy)`
`z = Mux(c, x, y)`	`wz = max(wx, wy)`
`z = w * y`	`wz = wx + wy`
`z = x << n`	`wz = wx + maxNum(n)`
`z = x >> n`	`wz = wx - minNum(n)`
`z = Cat(x, y)`	`wz = wx + wy`
`z = Fill(n, x)`	`wz = wx * maxNum(n)`

where for instance wz is the bit width of wire z, and the & rule applies to all bitwise logical operations.

The bit-width inference process continues until no bit width changes. Except for right shifts by known constant amounts, the bit-width inference rules specify output bit widths that are never smaller than the input bit widths, and thus, output bit widths either grow or stay the same. Furthermore, the width of a register must be specified by the user either explicitly or from the bitwidth of the reset value. From these two requirements, we can show that the bit-width inference process will converge to a fixpoint.

```
Shouldn't & return bitwidth that is min() of inputs?
```

6 Updateables

When describing the operation of wire and state nodes, it is often useful to give the specification as a series of conditional updates to the output value and to spread out these updates across several separate statements. For example, the output of a Data node can be referenced immediately, but its input can be set later. `Updateable` represents a conditionally updateable node, which accumulates accesses to the node and which can later generate muxes to combine these accesses in the circuit.

```
abstract class Updateable extends Node {
  // conditional reads
  def reads: Queue[(Bool, UFix)]
  // conditional writes
  def writes: Queue[(Bool, UFix, Node)]
  // gen mux integrating all conditional writes
  def genMuxes(default: Node)
  override def := (x: Node): this.type
}
```

Chisel provides conditional update rules in the form of the `when` construct to support this style of sequential logic description:

```
object when {
  def apply(cond: Bool)(block: => Unit): when
}

class when (prevCond: Bool) {
  def elsewhen (cond: Bool)(block: => Unit): when
  def otherwise (block: => Unit): Unit
}
```

`when` manipulates a global condition stack with dynamic scope. Therefore, `when` creates a new condition that is in force across function calls. For example:

```
def updateWhen (c: Bool, d: Data) =
  when (c) { r := d }
```

```
when (a) {
  updateWhen(b, x)
}
```

is the same as:

```
when (a) {
  when (b) { r := x }
}
```

Chisel provides some syntactic sugar for other common forms of conditional updates:

```
def unless(c: Bool)(block: => Unit) =
  when (!c) { block }
```

and

```
def otherwise(block: => Unit) =
  when (Bool(true)) { block }
```

We introduce the `switch` statement for conditional updates involving a series of comparisons against a common key:

```
def switch(c: Bits)(block: => Unit): Unit

def is(v: Bits)(block: => Unit)
```

7 Forward Declarations

Purely combinational circuits are not allowed to have cycles between nodes, and Chisel will report an error if such a cycle is detected. Because they do not have cycles, legal combinational circuits can always be constructed in a feed-forward manner, by adding new nodes whose inputs are derived from nodes that have already been defined. Sequential circuits naturally have feedback between nodes, and so it is sometimes necessary to reference an output wire before the producing node has been defined. Because Scala evaluates program statements sequentially, we have allowed data nodes to serve as a wire providing a declaration of a node that can be used immediately, but whose input will be set later. For example, in a simple CPU, we need to define the `pcPlus4` and `brTarget` wires so they can be referenced before definition:

```
val pcPlus4  = UFix()
val brTarget = UFix()
val pcNext   = Mux(pcSel, brTarget, pcPlus4)
val pcReg    = Reg(pcNext)
pcPlus4     := pcReg + UFix(4)
...
brTarget    := addOut
```

The wiring operator `:=` is used to wire up the connection after `pcReg` and `addOut` are defined. After all assignments are made and the circuit is being elaborated, it is an error if a forward declaration is unassigned.

8 Regs

The simplest form of state element supported by Chisel is a positive-edge-triggered register defined as follows:

```
object Reg {
  def apply[T <: Data]
        (data: T, resetVal: T = null): T
  def apply[T <: Data] (resetVal: T): T
  def apply[T <: Data] ()(type: => T): T
}

class Reg extends Updateable
```

where it can be constructed as follows:

```
val r1 = Reg(io.in)
val r2 = Reg(resetVal = UFix(1, 8))
val r3 = Reg(data = io.in, resetVal = UFix(1))
val r4 = Reg(){ UFix(width = 8) }
```

where `resetVal` is the value a reg takes on when implicit `reset` is `Bool(true)`.

9 Mems

Chisel supports random-access memories via the Mem construct. Writes to Mems are positive-edge-triggered and reads are either combinational or positive-edge-triggered.

```
object Mem {
  def apply[T <: Data](depth: Int,
          seqRead: Boolean = false)
        (type: => T): Mem
}

class Mem[T <: Data](depth: Int,
      seqRead: Boolean = false,
      type: () => T)
    extends Updateable {
  def apply(idx: UFix): T
}
```

Ports into Mems are created by applying a `UFix` index. A 32-entry register file with one write port and two combinational read ports might be expressed as follows:

```
val rf = Mem(32) { UFix(width = 64) }
```

```
when (wen) { rf(waddr) := wdata }
val dout1 = rf(waddr1)
val dout2 = rf(waddr2)
```

If the optional parameter seqRead is set, Chisel will attempt to infer sequential read ports when a Reg is assigned the output of a Mem. A one-read, one-write SRAM might be described as follows:

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

Single-ported SRAMs can be inferred when the read and write conditions are mutually exclusive in the same `when` chain:

```
val ram1p =
  Mem(1024, seqRead = true) { Bits(width = 32) }
val dout = Reg() { Bits() }
when (wen) { ram1p(waddr) := wdata }
.elsewhen (ren) { dout := ram1p(raddr) }
```

If the same Mem address is both written and sequentially read on the same clock edge, or if a sequential read enable is cleared, then the read data is implementation-defined.

Mem also supports write masks for subword writes. A given bit is written if the corresponding mask bit is set.

```
val ram = Mem(256) { Bits(width = 32) }
when (wen) { ram.write(waddr, wdata, wmask) }
```

10 Ports

Ports are `Data` derived nodes used as interfaces to hardware components. A port is a directional version of a primitive `Data` object. Port directions are defined as follows:

```
trait PortDir
object INPUT  extends PortDir
object OUTPUT extends PortDir
```

Aggregate ports can be recursively constructed using either a vec or bundle with instances of `Ports` as leaves.

11 Components

In Chisel, *components* are very similar to *modules* in Verilog, defining a hierarchical structure in the generated circuit. The hierarchical component namespace is accessible in downstream tools to aid in debugging and physical layout. A user-defined component is defined as a *class* which:

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

Users write their own components by subclassing Component which is defined as follows:

```
abstract class Component {
  val io: Bundle
  var name: String = ""
  def compileV: Unit
  def compileC: Unit
}
```

and defining their own `io` field. For example, to define a two input mux, we would define a component as follows:

```
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)
}
```

The `:=` assignment operator, used in the body of a component definition, is a special operator in Chisel that wires the input of left-hand side to the output of the right-hand side. It is typically used to connect an output port to its definition.

The `<>` operator bulk connects interfaces of opposite gender between sibling components or interfaces of same gender between parent/child components. Bulk connections connect leaf ports using pathname matching. Connections are only made if one of the ports is non-null, allowing users to repeatedly bulk-connect partially filled interfaces. After all connections are made and the circuit is being elaborated, Chisel warns users if ports have other than exactly one connection to them.

The names given to the nodes and subcomponents stored in a component when they are emitted by a C++ or Verilog backend are obtained from their component field names, using Scala introspection.

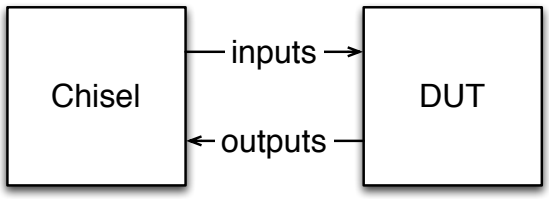

Figure 4: DUT run using a Tester object in Scala with stdin and stdout connected

12 BlackBox

Black boxes allow users to define interfaces to circuits defined outside of Chisel. The user defines:

- a component as a subclass of `BlackBox` and
- an `io` field with the interface.

For example, one could define a simple ROM blackbox as:

```
class RomIo extends Bundle {
  val isVal = Bool(INPUT)
  val raddr = UFix(INPUT,  32)
  val rdata = Bits(OUTPUT, 32)
}

class Rom extends BlackBox {
  val io = new RomIo()
}
```

13 Main and Testing

In order to construct a circuit, the user calls `chiselMain` from their top level `main` function:

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

which when run creates C++ files named *component_name*`.cpp` and *component_name*`.h` in the directory specified with `-targetDir` *dir_name* argument.

Testing is a crucial part of circuit design, and thus in Chisel we provide a mechanism for testing circuits by providing test vectors within Scala using subclasses of the `Tester` class:

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

which binds a tester to a component, and specifies the nodes to test, and finally allows users to write a `defTests` function. The definition for `defTests` is:

```
def defTests(body: => Boolean)
```

where `testNodes` are the graph nodes that will be input to and output from the DUT, and where users write calls to `step` in the body. Users connect tester instances to components using:

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

When `-test` is given as an argument to `chiselMain`, a tester instance runs the Design Under Test (DUT) in a separate process with stdin and stdout connected so that inputs can be sent to the DUT and outputs can received from the DUT as shown in Figure 4.

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

where `vars` is a mapping of test nodes to literals, with assignments to input nodes being sent to the DUT and assignments to non-input nodes being interpreted as expected values. `test` first sends inputs specifed in vars, steps the DUT and then either compares expected values from vars or sets vars for test nodes without entries in `vars`. The following is an example for defining tests for `Mux2`:

```
class Mux2Tests(c: Mux2)
    extends Tester(c, Array(c.io)) {
  defTests {
    var allGood = true
    val n = pow(2, 3).toInt
    val vars = new HashMap[Node, Node]()
    for (s <- 0 until 2) {
    for (i0 <- 0 until 2) {
    for (i1 <- 0 until 2) {
      vars(c.io.sel) = Bits(s)
      vars(c.io.in1) = Bits(i1)
      vars(c.io.in0) = Bits(i0)
      vars(c.io.out) = Bits(if (s == 1) i1 else
          i0)
      allGood &&= step(vars)
    } } }
    allGood
  }
}
```

and the following shows how it is invoked:

```
chiselMainTest(args + "--test", () => new Mux2()){
  c => new Mux2Tests(c)
}
```

Finally, command arguments for `chiselMain*` are as follows:

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

14 C++ Emulator

The C++ emulator is based on a fast multiword library using C++ templates. A single word is defined by `val_t` as follows:

```
typedef uint64_t val_t;
typedef int64_t sval_t;
typedef uint32_t half_val_t;
```

and multiwords are defined by `dat_t` as follows:

```
template <int w>
class dat_t {
 public:
  const static int n_words;
  inline int width ( void );
  inline int n_words_of ( void );
  inline bool to_bool ( void );
  inline val_t lo_word ( void );
  inline unsigned long to_ulong ( void );
  std::string to_str ();
  static dat_t<w> rand();
  dat_t<w> ();
template <int sw>
  dat_t<w> (const dat_t<sw>& src);
  dat_t<w> (const dat_t<w>& src);
  dat_t<w> (val_t val);
template <int sw>
  dat_t<w> mask(dat_t<sw> fill, int n);
template <int dw>
  dat_t<dw> mask(int n);
template <int n>
  dat_t<n> mask(void);
  dat_t<w> operator + ( dat_t<w> o );
  dat_t<w> operator - ( dat_t<w> o );
  dat_t<w> operator - ( );
  dat_t<w+w> operator * ( dat_t<w> o );
  dat_t<w+w> fix_times_fix( dat_t<w> o );
  dat_t<w+w> ufix_times_fix( dat_t<w> o );
  dat_t<w+w> fix_times_ufix( dat_t<w> o );
  dat_t<1> operator < ( dat_t<w> o );
  dat_t<1> operator > ( dat_t<w> o );
  dat_t<1> operator >= ( dat_t<w> o );
  dat_t<1> operator <= ( dat_t<w> o );
  dat_t<1> gt ( dat_t<w> o );
  dat_t<1> gte ( dat_t<w> o );
  dat_t<1> lt ( dat_t<w> o );
  dat_t<1> lte ( dat_t<w> o );
  dat_t<w> operator ^ ( dat_t<w> o );
  dat_t<w> operator & ( dat_t<w> o );
  dat_t<w> operator | ( dat_t<w> o );
  dat_t<w> operator ~ ( void );
  dat_t<1> operator ! ( void );
  dat_t<1> operator && ( dat_t<1> o );
  dat_t<1> operator || ( dat_t<1> o );
  dat_t<1> operator == ( dat_t<w> o );
  dat_t<1> operator == ( datz_t<w> o );
  dat_t<1> operator != ( dat_t<w> o );
  dat_t<w> operator << ( int amount );
  dat_t<w> operator << ( dat_t<w> o );
  dat_t<w> operator >> ( int amount );
  dat_t<w> operator >> ( dat_t<w> o );
  dat_t<w> rsha ( dat_t<w> o);
  dat_t<w>& operator = ( dat_t<w> o );
  dat_t<w> fill_bit(val_t bit);
  dat_t<w> fill_byte
    (val_t byte, int nb, int n);
template <int dw, int n>
  dat_t<dw> fill( void );
template <int dw, int nw>
  dat_t<dw> fill( dat_t<nw> n );
template <int dw>
  dat_t<dw> extract();
template <int dw>
  dat_t<dw> extract(val_t e, val_t s);
template <int dw, int iwe, int iws>
  dat_t<dw> extract
    (dat_t<iwe> e, dat_t<iws> s);
template <int sw>
  dat_t<w> inject
    (dat_t<sw> src, val_t e, val_t s);
template <int sw, int iwe, int iws>
  dat_t<w> inject
    (dat_t<sw> src,
     dat_t<iwe> e, dat_t<iws> s);
template <int dw>
  dat_t<dw> log2();
  dat_t<1> bit(val_t b);
  val_t msb();
template <int iw>
  dat_t<1> bit(dat_t<iw> b)
}


template <int w, int sw>
  dat_t<w> DAT(dat_t<sw> dat);
template <int w>
  dat_t<w> LIT(val_t value);
template <int w> dat_t<w>
  mux ( dat_t<1> t, dat_t<w> c, dat_t<w> a )
```

where `w` is the bit width parameter.

The Chisel compiler compiles top level components into a single flattened `mod_t` class that can be

created and executed:

```
class mod_t {
 public:
  // initialize component
  virtual void init (void) { };
  // compute all combinational logic
  virtual void clock_lo (dat_t<1> reset) { };
  // commit state updates
  virtual void clock_hi (dat_t<1> reset) { };
  // print printer specd node values to stdout
  virtual void print (FILE* f) { };
  // scan scanner specd node values from stdin
  virtual bool scan (FILE* f) { return true; };
  // dump vcd file
  virtual void dump (FILE* f, int t) { };
};
```

Either the Chisel compiler can create a harness or the user can write a harness themselves. The following is an example of a harness for a CPU component:

```
#include "cpu.h"

int main (int argc, char* argv[]) {
  cpu_t* c = new cpu_t();
  int lim = (argc > 1) ? atoi(argv[1]) : -1;
  c->init();
  for (int t = 0; lim < 0 || t < lim; t++) {
    dat_t<1> reset = LIT<1>(t == 0);
    if (!c->scan(stdin)) break;
    c->clock_lo(reset);
    c->clock_hi(reset);
    c->print(stdout);
  }
}
```

15 Verilog

Chisel generates Verilog when the `-v` argument is passed into `chiselMain`. For example, from SBT, the following

```
run --v
```

would produce a single Verilog file named *component-name*`.v` in the target directory. The file will contain one module per component defined as subcomponents of the top level component created in `chiselMain`. Modules with the same interface and body are cached and reused.

16 Extra Stuff

```
def ListLookup[T <: Bits]
  (addr: Bits, default: List[T],
   mapping: Array[(Bits, List[T])]): List[T]

def Lookup[T <: Data]
  (addr: Bits, default: T,
   mapping: Seq[(Bits, T)]): T

// n-way multiplexor
def MuxCase[T <: Data]
  (default: T, mapping: Seq[(Bool, T)]): T

// n-way indexed multiplexer:
def MuxLookup[S <: Bits, T <: Data]
  (key: S, default: T, mapping: Seq[(S, T)]): T


// create n enum values of given type
def Enum[T <: Bits]
  (n: Int)(type: => T): List[T]
```

17 Standard Library

17.1 Math

```
// Returns the log base 2 of the input
// Scala Integer rounded up
def log2Up(in: Int): Int

// Returns the log base 2 of the input
// Scala Integer rounded down
def log2Down(in: Int): Int

// Returns true if the input Scala Integer
// is a power of 2
def isPow2(in: Int): Boolean

// linear feedback shift register
def LFSR16(increment: Bool = Bool(true)): Bits
```

17.2 Sequential

```
// Returns the n-cycle delayed version
// of the input signal
def ShiftRegister[T <: Data](n: Int, in: T): T

def Counter(cond: Bool, n: Int) = {
  val c = Reg(resetVal = UFix(0, log2Up(n)))
  val wrap = c === UFix(n-1)
  when (cond) {
    c := Mux(Bool(!isPow2(n)) && wrap, UFix(0), c
        + UFix(1))
  }
  (c, wrap && cond)
}
```

17.3 Bits

```
// Returns the number of bits set in the
// input signal. Causes an exception if
// the input is wider than 32 bits.
def PopCount(in: Bits): Bits

// Returns the reverse the input signal
def Reverse(in: Bits): Bits

// returns the one hot encoding of
// the input UFix
def UFixToOH(in: UFix, width: Int): Bits

// does the inverse of UFixToOH
def OHToUFix(in: Bits): UFix
def OHToUFix(in: Seq[Bool]): UFix

// Builds a Mux tree out of the input
// signal vector using a one hot encoded
// select signal. Returns the output of
// the Mux tree
def Mux1H[T <: Data](sel: Bits, in: Vec[T]): T
def Mux1H[T <: Data](sel: Vec[Bool], in: Vec[T]):
    T

// Builds a Mux tree under the
// assumption that multiple
// select signals can be enabled.
// Priority is given to the first
// select signal. Returns the output
// of the Mux tree.
def PriorityMux[T <: Data](sel: Bits, in:
    Seq[T]): T
def PriorityMux[T <: Data](sel: Seq[Bits], in:
    Seq[T]): T

// Returns the bit position of the
// trailing 1 in the input vector with
// the assumption that multiple bits of
// the input bit vector can be set
def PriorityEncoder(in: Bits): UFix
def PriorityEncoder(in: Seq[Bool]): UFix

// Returns the bit position of the
// trailing 1 in the input vector with
// the assumption that only one bit in
// the input vector can be set
def PriorityEncoderOH(in: Bits): UFix
def PriorityEncoderOH(in: Seq[Boo]): UFix
```

17.4 Decoupled

```
// Adds a ready-valid handshaking
// protocol to any interface. The
// standard used is that the
// consumer uses the flipped
// interface.
class FIFOIO[+T <: Data]()(data: => T)
    extends Bundle {
  val ready = Bool(INPUT)
  val valid = Bool(OUTPUT)
  val bits  = data.asOutput
}

// Adds a valid protocol to any
// interface. The standard used is
// that the consumer uses the
// fliped interface.
class PipeIO[+T <: Data]()(data: => T)
    extends Bundle {
  val valid = Bool(OUTPUT)
  val bits = data.asOutput
}

// Hardware module that is used to
// sequence n producers into 1 consumer.
// Priority is given to lower
// producer
// Example usage:
//    val arb = new Arbiter(2){ Bits() }
//    arb.io.in(0) <> producer0.io.out
//    arb.io.in(1) <> producer1.io.out
//    consumer.io.in <> arb.io.out
class Arbiter[T <: Data](n: Int)(data: => T)
    extends Component

// Hardware module that is used to
// sequence n producers into 1 consumer.
// Producers are chosen in round robin
// order
// Example usage:
//    val arb = new RRArbiter(2){ Bits() }
//    arb.io.in(0) <> producer0.io.out
//    arb.io.in(1) <> producer1.io.out
//    consumer.io.in <> arb.io.out
class RRArbiter[T <: Data](n: Int)(data: => T)
    extends Component

// Generic hardware queue. Required
// parameter entries controls the
// depth of the queues. The width of
// the queue is determined from the
// inputs.
// Example usage:
//    val q = new Queue(16){ Bits() }
//    q.io.enq <> producer.io.out
//    consumer.io.in <> q.io.deq
class Queue[T <: Data]
    (entries: Int,
     pipe: Boolean = false,
     flow: Boolean = false
     flushable: Boolean = false)
    (data: => T) extends Component

// A hardware module that delays data
// coming down the pipeline by the
// number of cycles set by the
// latency parameter. Functionality
```

```
// is similar to ShiftRegister but
// this exposes a Pipe interface.
// Example usage:
//    val pipe = new Pipe(){ Bits() }
//    pipe.io.enq <> produce.io.out
//    consumer.io.in <> pipe.io.deq
class Pipe[T <: Data]
    (latency: Int = 1)
    (data: => T) extends Component
```

References

[1] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, Wawrzynek, J., Asanović *Chisel: Constructing Hardware in a Scala Embedded Language* in DAC '12.

[2] Odersky, M., Spoon, L., Venners, B. *Programming in Scala* by Artima.

[3] Payne, A., Wampler, D. *Programming Scala* by O'Reilly books.