Basic Chisel Constructs

Chisel Wire Operators:
```scala
val x = Uint()  // Allocate a as wire of type Uint()
x := y  // Assign (connect) wire y to wire x
x <> y  // Connect x and y, wire directionality is automatically inferred
```

When executes blocks conditionally by `Bool`, and is equivalent to Verilog if:
```scala
when(condition1) {
  // run if condition1 true and skip rest
} .elsewhen(condition2) {
  // run if condition2 true and skip rest
}.unless(condition3) {
  // run if condition3 false and skip rest
}.otherwise {
  // run if none of the above ran
}
```

Switch executes blocks conditionally by data
```scala
switch(x) {
  is(value1) {
    // run if x === value1
  } is(value2) {
    // run if x === value2
  }
}
```

Enum generates value literals for enumerations
```scala
val s1::s2:: ... ::sn::Nil
  = Enum(nodeType:Uint, n:Int)
s1, s2, ..., sn will be created as nodeType literals with distinct values
nodeType type of s1, s2, ..., sn
n element count
```

Math Helpers:
```scala
log2Up(in:Int): Int  // log2(in) rounded up
log2Down(in:Int): Int  // log2(in) rounded down
isPow2(in:Int): Boolean True if in is a power of 2
```

### Basic Data Types

<table>
<thead>
<tr>
<th>Chisel</th>
<th>Explanation</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x(n)</code></td>
<td>Extract bit, 0 is LSB</td>
<td>1</td>
</tr>
<tr>
<td><code>x(n, m)</code></td>
<td>Extract bitfield</td>
<td>n - m + 1</td>
</tr>
<tr>
<td><code>x &lt;&lt; y</code></td>
<td>Dynamic left shift</td>
<td><code>w(x) + maxVal(y)</code></td>
</tr>
<tr>
<td><code>x &gt;&gt; y</code></td>
<td>Dynamic right shift</td>
<td><code>w(x) - minVal(y)</code></td>
</tr>
<tr>
<td><code>x &amp; y</code></td>
<td>Static left shift</td>
<td><code>w(x) + n</code></td>
</tr>
<tr>
<td><code>x &gt;&gt; n</code></td>
<td>Static right shift</td>
<td><code>w(x) - n</code></td>
</tr>
<tr>
<td><code>Fill(n, x)</code></td>
<td>Replicate x, n times</td>
<td><code>n * w(x)</code></td>
</tr>
<tr>
<td><code>Cat(x, y)</code></td>
<td>Concatenate bits</td>
<td><code>w(x) + w(y)</code></td>
</tr>
<tr>
<td><code>Max(c, x, y)</code></td>
<td>If c, then x; else y</td>
<td><code>max(w(x), w(y))</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chisel</th>
<th>Explanation</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!x</code></td>
<td>Logical NOT</td>
<td>1</td>
</tr>
<tr>
<td><code>x &amp; y</code></td>
<td>Logical AND</td>
<td>1</td>
</tr>
<tr>
<td>`x</td>
<td>y`</td>
<td>Logical OR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chisel</th>
<th>Explanation</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x(n)</code></td>
<td>Extract bit, 0 is LSB</td>
<td>1</td>
</tr>
<tr>
<td><code>x(n, m)</code></td>
<td>Extract bitfield</td>
<td>n - m + 1</td>
</tr>
<tr>
<td><code>w(x)</code></td>
<td>wire directionality</td>
<td><code>w(x) + maxVal(y)</code></td>
</tr>
<tr>
<td><code>w(x)</code></td>
<td>wire directionality</td>
<td><code>w(x) - minVal(y)</code></td>
</tr>
<tr>
<td><code>w(x)</code></td>
<td>wire directionality</td>
<td><code>w(x) + n</code></td>
</tr>
<tr>
<td><code>w(x)</code></td>
<td>wire directionality</td>
<td><code>w(x) - n</code></td>
</tr>
<tr>
<td><code>w(x)</code></td>
<td>wire directionality</td>
<td><code>n * w(x)</code></td>
</tr>
<tr>
<td><code>w(x)</code></td>
<td>wire directionality</td>
<td><code>w(x) + w(y)</code></td>
</tr>
</tbody>
</table>

### State Elements

Registers retain state until updated
```scala
val my_reg = Reg([outType:Data], [next:Data], [init:Data])
```

outType (optional) register type (or inferred)
next (optional) update value every clock
init (optional) initialization value on reset

Upgrading: assign to latch new value on next clock:
```scala
my_reg := next_val
```

The last update (lexically, per clock) runs

### Read-Write Memory

Provide addressable memories
```scala
val my_mem = Mem(out:Data, n:Int, seqRead:Boolean)
```

out memory element type
n memory depth (elements)
seqRead only update reads on clock edge

Using: access elements by indexing:
```scala
val readVal = my_mem(addr:Uint/Int)
```

for synchronous read: assign output to Reg
```scala
mu_mem(addr:Uint/Int) := y
```

### Modules

Defining: subclass `Module` with elements, code:
```scala
class Accum(width: Int) extends Module {
  val io = new Bundle {
    val in = Uint(INPUT, width)
    val out = Uint(OUTPUT, width)
  }
  val sum = new Reg(UInt())
  sum := sum + io.in
  io.out := sum
}
```

Usage: access elements using dot notation:
```scala
val my_module = Module(new Accum(32))
my_module.io.in := some_data
val sum := my_module.io.out
```

### Functions

Provide block abstractions for code
Defining: write Scala functions with Chisel code:
```scala
def Adder(op_a : UInt, op_b : UInt): UInt = {
  op_a + op_b
}
```

Usage: hardware is instantiated when called:
```scala
val sum := Adder(UInt(1), some_data)
```

If/For can be used to control hardware generation and is equivalent to Verilog `generate if/for`
Aggregate Types

Bundle contains Data types indexed by name

Defining: subclass Bundle, define components:

```scala
class MyBundle extends Bundle {
  val a = Bool()
  val b = UInt(width = 32)
}
```

Constructor: instantiate Bundle subclass:
val my_bundle = new MyBundle()

Inline defining: define a Bundle type:
```scala
val my_bundle = new Bundle {
  class MyBundle extends Bundle {
    val a = Bool()
    val b = UInt(width = 32)
  }
}
```

Using: access elements through dot notation:
```scala
val bundleVal = my_bundle.a
my_bundle.a := Bool(true)
```

Vec is an indexable vector of Data types
```scala
val myVec = Vec(els:Iterable[Data])
val myVec = Vec.fill(n:Int) {gen:Data}
```

Vect is an indexable vector of Data types
```scala
val myVec = Vec(els:Iterable[Data])
elts initial element Data (vector depth inferred)
val myVec = Vec.fill(n:Int) {gen:Data}
n vector depth (elements)
gen initial element Data, called once per element
```

Using: access elements by dynamic or static indexing:
```scala
readVal := myVec(ind:Data/idx:Int)
```

Stateful:
```scala
PriorityEncoderOH(in:Bits/Iterable[Bool]): UInt
Returns the position the least significant 1 in in
PriorityEncoderOH(in:Bits): UInt
Returns the position of the hot bit in in
Mux1H(in:Iterable[(Data, Bool)]: Data
Mux1H(sel:Bits/Iterable[Bool],
in:Iterable[Data]): Data
PriorityMux(in:Iterable[Bool, Bits]): Bits
PriorityMux(sel:Bits/Iterable[Bool],
in:Iterable[Bits]): Bits
A mux tree with either a one-hot select or multiple
selects (where the first inputs are prioritized)
in iterable of combined input and select (Bool, Bits)
tuples or just mux input Bits
sel select signals or bitvector, one per input
```

Standard Library: Function Blocks

DecoupledIO is a Bundle with a ready-valid interface
```scala
Decoupled(gen:Data)
gen Chisel Data to wrap ready-valid protocol around
```

Interface:
```scala
.ready ready Bool
.valid valid Bool
.bits data
```

ValidIO is a Bundle with a valid interface
```scala
Valid(gen:Data)
gen Chisel Data to wrap valid protocol around
```

Interface:
```scala
.ready ready Bool
.valid valid Bool
.bits data
```

Queue is a Module providing a hardware queue
```scala
Queue(enq:DecoupledIO, entries:Int)
enq DecoupledIO source for the queue
entries size of queue
```

Interface:
```scala
.enq DecoupledIO source (flipped)
.deq DecoupledIO sink
.count UInt count of elements in the queue
```

Pipe is a Module delaying input data
```scala
Pipe(enqValid: Bool, enqBits: Data, [latency:Int])
enqValid input data, valid component
enqBits input data, data component
enq input data as ValidIO
```

Arbiters are Modules connecting multiple producers
```scala
Arbiter(gen:Data, n:Int)
gen data type
n number of producers
```

Interface:
```scala
.enq ValidIO source (flipped)
.deq ValidIO sink
```

Tester is a class with functions for testing Modules,
connecting and communicating with a simulator:
```scala
reset(n:Int) reset the DUT for n (default 1) clocks
step(n:Int) steps the DUT for n clocks
```

Pipe is a Module delaying input data
```scala
Pipe(enqValid: Bool, enqBits: Data, [latency:Int])
enqValid input data, valid component
enqBits input data, data component
enq input data as ValidIO
```

Arbiters are Modules connecting multiple producers
to one consumer
Arbiter prioritizes lower producers
RRArbiter runs in round-robin order
```scala
Arbiter(gen:Data, n:Int)
gen data type
n number of producers
```

Interface:
```scala
.enq ValidIO source (flipped)
.deq ValidIO sink
```

Tester is a class with functions for testing Modules,
connecting and communicating with a simulator:
```scala
reset(n:Int) reset the DUT for n (default 1) clocks
step(n:Int) steps the DUT for n clocks
```

Pipe is a Module delaying input data
```scala
Pipe(enqValid: Bool, enqBits: Data, [latency:Int])
enqValid input data, valid component
enqBits input data, data component
enq input data as ValidIO
```

Arbiters are Modules connecting multiple producers
to one consumer
Arbiter prioritizes lower producers
RRArbiter runs in round-robin order
```scala
Arbiter(gen:Data, n:Int)
gen data type
n number of producers
```

Interface:
```scala
.enq ValidIO source (flipped)
.deq ValidIO sink
```

Tester is a class with functions for testing Modules,
connecting and communicating with a simulator:
```scala
reset(n:Int) reset the DUT for n (default 1) clocks
step(n:Int) steps the DUT for n clocks
```

Pipe is a Module delaying input data
```scala
Pipe(enqValid: Bool, enqBits: Data, [latency:Int])
enqValid input data, valid component
enqBits input data, data component
enq input data as ValidIO
```

Arbiters are Modules connecting multiple producers
to one consumer
Arbiter prioritizes lower producers
RRArbiter runs in round-robin order
```scala
Arbiter(gen:Data, n:Int)
gen data type
n number of producers
```

Interface:
```scala
.enq ValidIO source (flipped)
.deq ValidIO sink
```