Basic Chisel Constructs

Chisel Wire Operators:

// Allocate a as wire of type UInt()
val x = Wire(UInt())
x := y // Connect wire y to wire x

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

Enum generates value literals for enumerations
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:
log2Ceil(in:Int): Int  log2(in) rounded up
log2Floor(in:Int): Int  log2(in) rounded down
isPow2(in:Int): Boolean True if in is a power of 2

Basic Data Types

Constructors:
Bool()
true.B or false.B type, boolean value
UInt(32.W) literal values
UInt() type 32-bit unsigned
77.U or "hdead".U type, width inferred
1.U(16.W) unlimited with forced width
SInt() or SInt(64.W) like UInt
3.S(2.W) signed literals

Bits, UInt, SInt Casts: reinterpret cast except for:
UInt → SInt Zero-extend to SInt

State Elements

Registers retain state until updated
val my_reg = Reg(UInt(32.W))

Flavors
RegInit(7.U(32.W)) reg with initial value 7
RegNext(next_val) update each clock, no init
RegEnable(next, enable) update with enable gate

Updating: assign to latch new value on next clock:
my_reg := next_val

Read-Write Memory provide addressable memories
val my_mem = Mem(n:Int, out:Data)
out memory element type
n memory depth (elements)

Using: access elements by indexing:
val readVal = my_mem(addr:UInt/Int)
for synchronous read: assign output to Reg
mu_mem(addr:UInt/Int) := y

Modules:

Defining: subclass Module with elements, code:

```chisel
class Accum(width: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out = Output(UInt(width.W))
  })
  val sum = new Reg(UInt())
  sum := sum + io.in
  io.out := sum
}
```

Usage: access elements using dot notation:
(code inside a Module is always running)
val my_module = Module(new Accum(32))
my_module.io.in := some_data
val sum := my_module.io.out

Operators:

<table>
<thead>
<tr>
<th>Chisel</th>
<th>Explanation</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>!x</td>
<td>Logical NOT</td>
<td>1</td>
</tr>
<tr>
<td>x &amp; y</td>
<td>Logical AND</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
<td>Logical OR</td>
</tr>
<tr>
<td>x(n)</td>
<td>Extract bit, 0 is LSB</td>
<td>1</td>
</tr>
<tr>
<td>x(n, m)</td>
<td>Extract bitfield</td>
<td>n - m + 1</td>
</tr>
<tr>
<td>x &lt;&lt; y</td>
<td>Dynamic left shift</td>
<td>w(x) + maxVal(y)</td>
</tr>
<tr>
<td>x &gt;&gt; y</td>
<td>Dynamic right shift</td>
<td>w(x) - minVal(y)</td>
</tr>
<tr>
<td>x &lt;&lt; n</td>
<td>Static left shift</td>
<td>w(x) + n</td>
</tr>
<tr>
<td>x &gt;&gt; n</td>
<td>Static right shift</td>
<td>w(x) - n</td>
</tr>
<tr>
<td>Fill(n, x)</td>
<td>Replicate x, n times</td>
<td>n * w(x)</td>
</tr>
<tr>
<td>Cat(x, y)</td>
<td>Concatenate</td>
<td>w(x) + w(y)</td>
</tr>
<tr>
<td>Mux(c, x, y)</td>
<td>If c, then x; else y</td>
<td>max(w(x), w(y))</td>
</tr>
</tbody>
</table>

Uint bit-reduction methods:

<table>
<thead>
<tr>
<th>Chisel</th>
<th>Explanation</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.andR</td>
<td>AND-reduce</td>
<td>1</td>
</tr>
<tr>
<td>x.orR</td>
<td>OR-reduce</td>
<td>1</td>
</tr>
<tr>
<td>x.xorR</td>
<td>XOR-reduce</td>
<td>1</td>
</tr>
</tbody>
</table>

As an example to apply the andR method to an SInt use:
x.asUInt.andR
Hardware Generation

Functions provide block abstractions for code. Scala functions that instantiate or return Chisel types are code generators.

Also: Scala’s if and for can be used to control hardware generation and are equivalent to Verilog generate if/for

```scala
val number = Reg(if(can_be_negative) SInt() else UInt())
```

will create a Register of type SInt or UInt depending on the value of a Scala variable

Aggregate Types

Bundle contains Data types indexed by name

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

Constructor: instantiate Bundle subclass

```scala
val my_bundle = new MyBundle()
```

Inline defining: define a Bundle type:

```scala
val my_bundle = new Bundle {
  val a = Bool()
  val b = UInt(32.W)
}
```

Using: access elements through dot notation:

```scala
val bundleVal = my_bundle.a
```

Vec is an indexable vector of Data types

```scala
val myVec = Vec(elts:Iterable[Data])
```

- elts initial element Data (vector depth inferred)
- gen initial element Data, called once per element

Using: access elements by dynamic or static indexing:

```scala
readVal := myVec(ind:Data/idx:Int)
```

Standard Library: Function Blocks

Stateless:

```scala
PopCount(in:Bits/Seq[Bool]): UInt
Returns number of hot (= 1) bits in in
```

- Reverse(in:UInt): UInt
  Reverses the bit order of in
- UIntToOH(in:UInt, [width:Int]): Bits
  Returns the one-hot encoding of in
  - width (optional, else inferred) output width
- OHToUInt(in:Bits/Seq[Bool]): UInt
  Returns the (in:Bits/Seq[Bool]): UInt
- Counter(n:Int): UInt
  .inc() bumps counter returning true when n reached
  .value returns current value
- PriorityEncoder(in:Bits/Iterable[bool]): UInt
  Returns the position of 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
  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
- PriorityMux(sel:Bits/Iterable[bool],
  in:Iterable[Data]): Data
- PriorityMuxOH(in:Bits): Bits
  Returns the position the hot bit in
- Stateless:
- LFSR16([increment:Bool]): UInt
  16-bit LFSR (to generate pseudorandom numbers)
  - increment (optional, default True) shift on next clock
- ShiftRegister(in:Data, n:Int, [en:Bool]): Data
  Shift register, returns n-cycle delayed input in
  - en (optional, default True) enable
```

Standard Library: Interfaces

DecoupledIO is a Bundle with a ready-valid interface

```scala
Constructor:
DecoupledIO(gen:Data)
```

ValidIO is a Bundle with a valid interface

```scala
Constructor:
Valid(gen:Data)
```

Pipe is a Module delaying input data

```scala
Constructor:
Pipe(enqValid:Bool, enqBits:Data, [latency:Int])
```

Arbiters are Modules connecting multiple producers to one consumer

```scala
Arbiter(gen:Data, n:Int)
```

Arbiters prioritize lower producers

```scala
RRArbiter runs in round-robin order
```

Queue is a Module providing a hardware queue

```scala
Constructor:
Queue(enq:DecoupledIO, entries:Int)
```

Pipe is a Module delaying input data

```scala
Constructor:
Pipe(enqValid:Bool, enqBits:Data, [latency:Int])
```

Queue is a Module providing a hardware queue

```scala
Queue(enq:DecoupledIO, entries:Int)
```

Pipe is a Module delaying input data

```scala
Constructor:
Pipe(enqValid:Bool, enqBits:Data, [latency:Int])
```

Arbiters are Modules connecting multiple producers to one consumer

```scala
Arbiter(gen:Data, n:Int)
```

Arbiters prioritize lower producers

```scala
RRArbiter runs in round-robin order
```

Queue is a Module providing a hardware queue

```scala
Queue(enq:DecoupledIO, entries:Int)
```

Pipe is a Module delaying input data

```scala
Constructor:
Pipe(enqValid:Bool, enqBits:Data, [latency:Int])
```

Arbiters are Modules connecting multiple producers to one consumer

```scala
Arbiter(gen:Data, n:Int)
```

Arbiters prioritize lower producers

```scala
RRArbiter runs in round-robin order
```

Queue is a Module providing a hardware queue

```scala
Queue(enq:DecoupledIO, entries:Int)
```

Pipe is a Module delaying input data

```scala
Constructor:
Pipe(enqValid:Bool, enqBits:Data, [latency:Int])
```

Arbiters are Modules connecting multiple producers to one consumer

```scala
Arbiter(gen:Data, n:Int)
```

Arbiters prioritize lower producers

```scala
RRArbiter runs in round-robin order
```