def class MemReq extends Bundle {
  val cmd = UInt(width = 2)
  val mtype = UInt(width = 2)
  val tag = UInt(width = 9)
  val addr = UInt(width = 64)
  val data = UInt(width = 64)
}

def class MemResp extends Bundle {
  val cmd = UInt(width = 2)
  val tag = UInt(width = 9)
  val mtype = UInt(width = 2)
  val data = UInt(width = 64)
}

def class OpReq extends Bundle {
  val code = new RoccInst()
  val a = UInt(width = 64)
  val b = UInt(width = 64)
}

def class OpResp extends Bundle {
  val idx = UInt(width = 5)
  val data = UInt(width = 64)
}

def class RoccIO extends Bundle {
  val busy = Bool(OUTPUT)
  val isInstr = Bool(OUTPUT)
  val memReq = Decoupled(new MemReq).flip
  val memResp = Decoupled(new MemResp)
  val opReq = Decoupled(new OpReq)
  val opResp = Decoupled(new OpResp).flip
}
- tags on write commands,
- responses to write commands,
- must keep busy asserted until all reads and writes have completed, and
- memory system has single port with accelerator having priority
What is Chisel?

- Chisel is just a set of class definitions in Scala and when you write a Chisel program you are actually writing a Scala program,
- Chisel programs produce and manipulate a data structure in Scala using a convenient textural language layered on top of Scala,
- Chisel makes it possible to create powerful and reusable hardware modules using modern programming language concepts, and
- the same Chisel description can generate different types of output
Today

- conditional updates on wires, registers, and memories,
- give you perspective,
- roms and rams,
- abstraction through object orientation and functional programming,
- present how to make hierarchical modules,
- teach you how to make reusable modules,
- show you to even more powerful construction techniques.
- introduce you to the standard library
When describing state operations, we could simply wire register inputs to combinational logic blocks, but it is often more convenient:

- to specify when updates to registers will occur and
- to specify these updates spread across several separate statements

```scala
val r = Reg(UInt(width = 16))
when (c === UInt(0)) {
  r := r + UInt(1)
}
```
Conditional Updates Priority

```plaintext
when (c1) { r := Bits(1) }
when (c2) { r := Bits(2) }
```

### Conditional Update Order:

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>r</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- r unchanged
- c2 takes precedence over c1
Each `when` statement adds another level of data mux and ORs the predicate into the enable chain and

the compiler effectively adds the termination values to the end of the chain automatically.
### Targetting Multiple Registers

```plaintext
r := Reg( init = UInt(3) )
s := Reg( init = UInt(3) )
when (c1) { r := UInt(1); s := UInt(1) }
when (c2) { r := UInt(2) }
```

leads to $r$ and $s$ being updated according to the following truth table:

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>r</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

$r$ updated in $c2$ block, $s$ updated using default
Conditional Update Nesting

when (a) { when (b) { body } }

which is the same as:

when (a && b) { body }
when (c1) { u1 }
  .elsewhen (c2) { u2 }
  .otherwise { ud }

which is the same as:

when (c1) { u1 }
when (!c1 && c2) { u2 }
when (!(c1 || c2)) { ud }
Switch Statement

```javascript
switch(idx) {
    is(v1) { u1 }
    is(v2) { u2 }
}
```

which is the same as:

```javascript
when (idx === v1) { u1 }
when (idx === v2) { u2 }
```
Conditional updates also work for
- wires but must have defaults and
- for memory reads and writes as we’ll see soon...

For wires, we can do conditional updates as follows:

```scala
val w = Bits(width = 32)
w := Bits(0)  // default value
when (c1)    { w := Bits(1) }
when (c2)    { w := Bits(2) }
```

which is the same as

```scala
val w = Bits(width = 32)
when (Bool(true)) { w := Bits(0) }  // default value
when (c1)    { w := Bits(1) }
when (c2)    { w := Bits(2) }
```
Enums can be defined to create a list of increasing nums.

```scala
object Enum {
  def apply[T <: UInt](type: T, n: Int): List[T] = ...
}

val s_even :: s_odd :: Nil = Enum(UInt(), 2)
```
Finite state machines can now be readily defined as follows:

```scala
class Parity extends Module {
  val io = new Bundle {
    val in  = Bool(INPUT)
    val out = Bool(OUTPUT) }
  val s_even :: s_odd :: Nil = Enum(UInt(), 2)
  val state = Reg(resetVal = s_even)
  when (io.in) {
    when (state === s_even) { state := s_odd }
    .otherwise { state := s_even }
  }
  io.out := (state === s_odd)
}
```

The diagram illustrates the state transitions of the parity machine.
What is Chisel?

- **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*
Functions are provided to create and connect circuit components:

- `make_register()`
- `make_adder()`
- `make_multiplexor()`
- `make_wire(name)`
- `make_constant(name)`
- `connect(input, output)`
- ...
```c
int main (){
    // Create Components
    Wire* reset = make_wire("reset");
    Reg* counter = make_register("counter");
    Adder* adder = make_adder();
    Multiplexor* multiplexor = make_multiplexor();
    Constant* one = make_constant(1);
    Constant* zero = make_constant(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);
}
```
What if Chisel was a C Library?

```c
int main (){
    // Create Components
    Wire* reset = make_wire("reset");
    Reg* counter = make_register("counter");
    Adder* adder = make_adder();
    Multiplexor* multiplexor = make_multiplexor();
    Constant* one = make_constant(1);
    Constant* zero = make_constant(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 C to programmatically generate hardware
- can use full power of C (loops, arrays, conditionals, ...)

![Diagram of hardware components with connections]
int main (){  // Create Components  Wire* reset = make_wire("reset");  Reg* counter = make_register("counter");  Adder* adder = make_adder();  Multiplexor* multiplexor = make_multiplexor();  Constant* one = make_constant(1);  Constant* zero = make_constant(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 C is pretty Verbose, how can we do better?
```c
int main (){  
    // Create Components  
    reset    := make_wire("reset");  
    counter  := make_register("counter");  
    adder    := make_adder();  
    multiplexor := make_multiplexor();  
    one      := make_constant(1);  
    zero     := make_constant(0);  

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

    // Produce Verilog  
    generate_verilog(counter);  
}
int main (){  
  // Create Components  
  reset := make_wire("reset");
  counter := make_register("counter");
  multiplexor := make_multiplexor();
  one := make_constant(1);
  zero := make_constant(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);
}
```c
int main (){
    // Create Components
    reset := make_wire("reset");
    counter := make_register("counter");
    one := make_constant(1);
    zero := make_constant(0);

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

    // Produce Verilog
    generate_verilog(counter);
}
```
int main (){
    // Create Components
    reset := make_wire("reset");
    counter := make_register("counter");

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

    // Produce Verilog
    generate_verilog(counter);
}

reset
+ 0

1

0

+
int main (){  
    // Create Components  
    reset := make_wire("reset");  
    counter := make_register("counter");  
    
    // Connect Components  
    connect(counter.in,  
             make_multiplexor(reset,  
                              make_constant(0),  
                              make_constant(1) + counter.out));  
    
    // Produce Verilog  
    generate_verilog(counter);  
}
int main (){
    // Create Components
    reset := make_wire("reset");
    counter := make_register("counter");

    // Connect Components
    counter.in <-
        make_multiplexor(reset,
                        make_constant(0),
                        make_constant(1) + counter.out);

    // Produce Verilog
    generate_verilog(counter);
}
```c
int main (){
    // Create Components
    reset := make_wire("reset");
    counter := make_register("counter");

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }

    // Produce Verilog
    generate_verilog(counter);
}
```
int main (){
  // Create Components
  reset := make_wire();
  counter := make_register();

  // Connect Components
  multiplexing(reset) {
    counter.in <- make_constant(0);
  } else {
    counter.in <- make_constant(1) + counter.out;
  }

  // Produce Verilog
  generate_verilog(counter);
}
make_counter(reset) {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main (){
    // Create Components
    reset := make_wire();
    counter := make_counter(reset);

    // Produce Verilog
    generate_verilog(counter);
}
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main (){
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := make_counter();
    }
    // Produce Verilog
    generate_verilog(counter);
}
Looks “Behavioral” but ...

```cpp
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main (){
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := 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
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main (){  
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := make_counter(reset);
    }

    // Produce Verilog
    generate_verilog(counter);
}
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main (){
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := make_counter(reset);
    }

    // Produce Verilog
    generate_verilog(counter);
}
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }

    return counter;
}

int main() {
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := make_counter(reset);
    }

    // Produce Verilog
    generate_verilog(counter);
}
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main (){
    // Create Components
    withReset(reset) {
        counter := make_counter(reset);
    }

    // Produce Verilog
    generate_verilog(counter);
}
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }

    return counter;
}

int main (){
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := make_counter();
    }

    // Produce Verilog
    generate_verilog(counter);
}
```java
make_counter() {
    counter := make_register();

    // Connect Components
    multiplexing(reset) {
        counter.in <- make_constant(0);
    } else {
        counter.in <- make_constant(1) + counter.out;
    }
    return counter;
}

int main() {
    // Create Components
    reset := make_wire();
    withReset(reset) {
        counter := make_counter(reset);
    }

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

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

Even Better with

- Object Orientation
- Powerful Macros
Well formed Chisel graphs are synthesizable.

- Use small number of basic nodes
  - simple semantics
  - easy to synthesize
- During construction check that
  - types, directions and widths match
  - there are no combinational loops

- If it passes these checks then it’s synthesizable
behavioral – high level language compiled to verilog
  ■ examples: C, Lime, CHP, Esterel, BlueSpec
simulation – simulation language with synthesizable subset
  ■ examples: Verilog, System Verilog, SystemC, myHDL
construction – programmatically construct circuits
  ■ examples: Chisel, Lava
## Hardware Language Comparisons

<table>
<thead>
<tr>
<th>name</th>
<th>example</th>
<th>pros</th>
<th>cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>behavioral</td>
<td>C-to-gates</td>
<td>high level</td>
<td>unpredictable QoR</td>
</tr>
<tr>
<td>simulation</td>
<td>Verilog</td>
<td>flexible</td>
<td>synthesizable? + low abstraction</td>
</tr>
<tr>
<td>construction</td>
<td>Chisel</td>
<td>metaprogramming + predictable QoR + synthesizable QoR</td>
<td>two levels + blemishes</td>
</tr>
</tbody>
</table>

**Chisel**
- is **not** Scala to Verilog
- produces circuits that are synthesizable by construction
- permits simulation by driving synthesized design
val d = Array(UInt(1), UInt(2), UInt(4), UInt(8))
val m = ROM(UInt(width = 32), d)
val r = m(counter(UInt(3)))
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 Array[UInt](256)
    for (x <- 0 until 16; y <- 0 until 16)
        muls((x << 4) | y) = UInt(x * y)

    val tbl = ROM(UInt(8), muls)

    io.z := tbl((io.x << 4) | io.y)
}
RAM is supported using the `Mem` construct

```scala
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
val regs = Mem(Bits(width = 32), 32)
when (wrEn) {
    regs(wrAddr) := wrData
}
val iDat = regs(iAddr)
val mDat = regs(mAddr)
Sequential read ports are inferred when:

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

```scala
val ramlr1lw = Mem(UInt(width = 32), 1024, seqRead = true)
val reg_raddr = Reg(UInt())
when (wen) { ramlr1lw(waddr) := wdata }
when (ren) { reg_raddr := raddr }
val rdata = ramlr1lw(reg_raddr)
```
Single-ported SRAMs can be inferred when the read and write conditions are mutually exclusive in the same when chain.

```scala
val ram1p = Mem(UInt(width = 32), 1024, seqRead = true)
val reg_raddr = Reg(UInt())
when (wen) { ram1p(waddr) := wdata }
.elsewhen (ren) { reg_raddr := raddr }
val rdata = ram1p(reg_raddr)
```
Mem also supports write masks for subword writes.

- A given bit is written if the corresponding mask bit is set.

```scala
val ram = Mem(UInt(width = 32), 256)
when (wen) { ram.write(waddr, wdata, wmask) }
```
Congratulations, you have all that you need at this point to write Chisel programs! You can write RTL, define modules (even with recursive data types), and wire them together.

- In order to attain true hardware description power though, you need to be able to write reusable RTL, modules and interfaces.
- This will allow you to both use and write generic module libraries and more quickly explore design space.
- To do this, we will use modern programming techniques such as:
  - object orientation,
  - functional programming,
  - parameterized types
- You will be greatly rewarded for your efforts!
First we need to learn about parameterized types in Scala. We can define a generic `Mux` function as taking a boolean condition and `con` and `alt` arguments (corresponding to then and else expressions) of type `T` as follows:

```scala
def Mux[T <: Data](c: Bool, con: T, alt: T): T = ...
```

where

- `T` is required to be a subclass of `Data` and
- the type of `con` and `alt` are required to match.

You can think of the type parameter as a way of just constraining the types of the allowable arguments.
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)
}
class Valid[T <: Data](dtype: T) extends Bundle {
    val data = dtype.clone
    val valid = Bool()
    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)
    }
}
abstract class Filter[T <: Data](dtype: T) extends Module {
    val io = new Bundle {
        val in = new Valid(dtype).asInput
        val out = new Valid(dtype).asOutput
    }
}

class FunctionFilter[T <: Data](f: T => T, dtype: T) extends Filter(dtype) {
    io.out.valid := io.in.valid
    io.out := f(io.in)
}
def clippingFilter[T <: Num](limit: Int, dtype: T) =
  new FunctionFilter(min(limit, max(-limit, _)), dtype)
def shiftingFilter[T <: Num](shift: Int, dtype: T) =
    new FunctionFilter(_ >> shift, dtype)
class ChainedFilter[T <: Num](dtype: T) extends Filter(dtype) = {
  val shift = new ShiftFilter(2, dtype)
  val clipper = new ClippingFilter(1 << 7, dtype)
  io.in <> shift.io.in
  shift.io.out <> clipper.io.in
  clipper.io.out <> io.out
}

---

**Chained Filter**
Functional Composition

Map(ins, x => x * y)

```
ins[0] → * y
```

```
ins[1] → * y
```

```
ins[2] → * y
```

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

```
in → f → f → f
```

Reduce(ins, Max)

```
ins[0] → Max
```

```
ins[1] → Max
```

```
```

```
```
def delays[T <: Data](x: T, n: Int): List[T] = 
  if (n <= 1) List(x) else x :: taps(Reg(next = x), n-1)

def FIR[T <: Num](hs: Seq[T], x: T): T =
  (hs, delays(x, hs.length)).zipped.map( _ * _ ).reduce( _ + _ )

class TstFIR extends Module {
  val io = new Bundle{ val x = SInt(INPUT, 8); val y = SInt(OUTPUT, 8) }
  val h   = Array(SInt(1), SInt(2), SInt(4))
  io.y := FIR(h, io.x)
}

\[
y[n] = \sum_{k=0}^{N-1} x[n - k]h[k]
\]
class Cpu extends Module {
    val io = new CpuIo()
    val c = new CtlPath()
    val d = new DatPath()
    c.io.ctl <> d.io.ctl
    c.io.dat <> d.io.dat
    c.io.imem <> io.imem
    d.io.imem <> io.imem
    c.io.dmem <> io.dmem
    d.io.dmem <> io.dmem
    d.io.host <> io.host
}
class RomIo extends Bundle {
    val isVal = Bool(INPUT)
    val raddr = UInt(INPUT, 32)
    val rdata = Bits(OUTPUT, 32)
}

class RamIo extends RomIo {
    val isWr = Bool(INPUT)
    val wdata = Bits(INPUT, 32)
}

class CpathIo extends Bundle {
    val imem = RomIo().flip()
    val dmem = RamIo().flip()
    ...
}

class DpathIo extends Bundle {
    val imem = RomIo().flip()
    val dmem = RamIo().flip()
    ...
}
class Cpath extends Module {
  val io = new CpathIo();
  ...
  io.imem.isVal := ...;
  io.dmem.isVal := ...;
  io.dmem.isWr := ...;
  ...
}

class Dpath extends Module {
  val io = new DpathIo();
  ...
  io.imem.raddr := ...;
  io.dmem.raddr := ...;
  io.dmem.wdata := ...;
  ...
}
class Cpu extends Module {
    val io = new CpuIo()
    val c = new CtlPath()
    val d = new DatPath()
    c.io.ctl <> d(io.ctl)
    c.io.dat <> d.io.dat
    c.io.imem <> io.imem
    d.io.imem <> io.imem
    c.io.dmem <> io.dmem
    d.io.dmem <> io.dmem
    d.io.host <> io.host
}
Resources

- Scala books
- chisel.eecs.berkeley.edu
- Chisel writings
  - Chisel tutorial
  - Chisel manual
  - Chisel DAC-2012 paper
- Chisel examples on github
  - Sodor Processors
  - Floating Point Unit
  - Rocket Processor
  - Hwacha Vector Unit
- Basic Utils
- Vecs
- Queues
- Arbiters
- Structural Memory
Bits Properties

```scala
object log2Up {
    def apply(in: Int): Int = if(in == 1) 1 else ceil(log(in)/log(2)).toInt
}

object log2Down {
    def apply(x : Int): Int = if (x == 1) 1 else floor(log(x)/log(2.0)).toInt
}

object isPow2 {
    def apply(in: Int): Boolean = in > 0 && ((in & (in-1)) == 0)
}

object PopCount {
    def apply(in: Seq[Boolean]): UInt = ...
    def apply(in: Bits): UInt = ...
}
```
LFSR16 – random number generator
Reverse – reverse order of bits
FillInterleaved – space out booleans into uint

```scala
object LFSR16 {
  def apply(increment: Bool = Bool(true)): UInt = ...
}
object Reverse {
  def apply(in: UInt): UInt = ...
}
object FillInterleaved {
  def apply(n: Int, in: Bits): UInt = ...
}
```
- n cycle delayed version of input signal

```scala
object ShiftRegister {
  def apply[T <: Data](in: T, n: Int, en: Bool = Bool(true)): T = ... 
}
```

- enable driven counter with parameterized wrapping

```scala
object Counter {
  def apply(cond: Bool, n: Int): (UInt, Bool) = ... 
}
```
Priority Encoding Functions

- UIntToOH – returns one hot encoding of input int
- OHToUInt – returns int version of one hot encoding input
- Mux1H – builds mux tree of input vector using a one hot encoded select signal

```scala
object UIntToOH {
  def apply(in: UInt, width: Int = -1): Bits = ...
}

object OHToUInt {
  def apply(in: Seq[Bool]): UInt = ...
}

object Mux1H {
  def apply[T <: Data](sel: Vec[Bool], in: Vec[T]): T = ...
}
```
Priority Mux Function

- PriorityMux – build mux tree allow multiple select signals with priority given to first select signal

```scala
object PriorityMux {
  def apply[T <: Bits](in: Seq[(Bool, T))): T = ...  
  def apply[T <: Bits](sel: Seq[Bool], in: Seq[T]): T = ...  
  def apply[T <: Bits](sel: Bits, in: Seq[T]): T = ...  
}
```
Priority Encoding Functions

- PriorityEncoder – 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.
- PriorityEncoderOH – 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.

```scala
object PriorityEncoder {
  def apply(in: Seq[Bool]): UInt = ...
  def apply(in: Bits): UInt = ...
}

object PriorityEncoderOH {
  def apply(in: Bits): UInt = ...
  def apply(in: Seq[Bool]): Seq[UInt] = ...
}
```
Vec Construction

object Vec {
  def apply[T <: Data](elts: Seq[T]): Vec[T]
  def apply[T <: Data](elts: Vec[T]): Vec[T]
  def apply[T <: Data](elt0: T, elts: T*): Vec[T]

  def fill[T <: Data](n: Int)(f: => T): Vec[T]
  def tabulate[T <: Data](n: Int)(f: Int => T): Vec[T]
  def tabulate[T <: Data](n1: Int, n2: Int)(f: (Int, Int) => T): Vec[Vec[T]]
}

Vec(A, L, M)
Vec.fill(3){ UInt(width = 8) } ====
  Vec(UInt(width = 8), UInt(width = 8), UInt(width = 8))
Vec.tabulate(3){ UInt(_) } ====
  Vec(UInt(0), UInt(1), UInt(2))
val v = Vec.fill(0){ UInt(width = 8) }
for ...
  v += UInt(width = 8)
class Vec[T <: Data](val gen: () => T)
    extends Data with Cloneable with BufferProxy[T] {
    ...
    def forall(p: T => Bool): Bool
    def exists(p: T => Bool): Bool
    def contains(x: T): Bool
    def count(p: T => Bool): UInt
    def indexWhere(p: T => Bool): UInt
    def lastIndexWhere(p: T => Bool): UInt
}
Queues

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

```scala
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
```
- delays data coming down pipeline by latency cycles
- similar to ShiftRegister but exposes Pipe interface

```scala
class PipeIO[+T <: Data](data: T) extends Bundle {
  val valid = Bool(OUTPUT)
  val bits = data.clone.asOutput
}

class Pipe[T <: Data](type: T, latency: Int = 1) extends Module

val pipe = new Pipe(UInt())
pipe.io.enq <> produce.io.out
consumer.io.in <> pipe.io.deq
```
- sequences $n$ producers into 1 consumer
- priority is given to lower producer

```scala
class ArbiterIO[T <: Data](data: T, n: Int) extends Bundle {
  val in = Vec.fill(n) { Decoupled(data) }.flip
  val out = Decoupled(data.clone)
  val chosen = Bits(OUTPUT, log2Up(n))
}

class Arbiter[T <: Data](type: T, n: Int) extends Module

val arb = new Arbiter(UInt(), 2)
arb.io.in(0) <> producer0.io.out
arb.io.in(1) <> producer1.io.out
consumer.io.in <> arb.io.out
```
- sequences \( n \) producers into 1 consumer
- producers are chosen in round robin order

```scala
class ArbiterIO[T <: Data](data: T, n: Int) extends Bundle {
  val in = Vec.fill(n) { Decoupled(data) }.flip
  val out = Decoupled(data.clone)
  val chosen = Bits(OUTPUT, log2Up(n))
}

class RRArbiter[T <: Data](type: T, n: Int) extends Module
  val arb = new RRArbiter(UInt(), 2)
  arb.io.in(0) <> producer0.io.out
  arb.io.in(1) <> producer1.io.out
  consumer.io.in <> arb.io.out
```
Structured Memory

- Defined number of ports
- Port specific accesses

```scala
class FunMem[T <: Data]
  (data: T, depth: Int, numReads: Int, numWrites: Int) {
    ...
    def read(addr: UInt, idx: Int = 0): T = ...
    def write(addr: UInt, data: T, idx: Int = 0) = ...
    ...
  }

val cellDats = new FunMem(Bits(width = DATA_WIDTH), NUM_CELLS, 1, 1)
when (isWrite0) {
  cellDats.write(ca0, dat0, 0)
}
when (isWrite1) {
  cellDats.write(ca1, dat1, 1)
}
... cellDats.read(ca, 0) ...
... cellDats.read(ca, 1) ...
```
“What is Chisel?” based on slides from Patrick Li