Chisel @ CS250 – Part II – Lecture 03

Jonathan Bachrach

EECS UC Berkeley

October 25, 2012

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 components using modern programming language concepts, and
- the same Chisel description can generate different types of output

Today

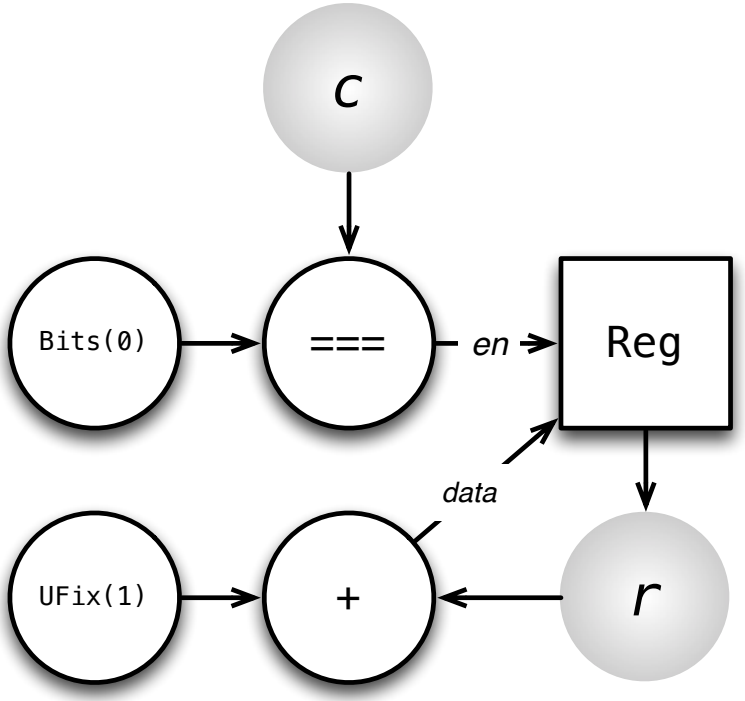
- finish out state operations,
- present how to make hierarchical components,
- teach you how to make reusable components,
- introduce you to even more powerful construction techniques.

Conditional Updates

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

```
val r = Reg() { UFix(16) }
when (c === UFix(0) ) {
  r := r + UFix(1)
}
```



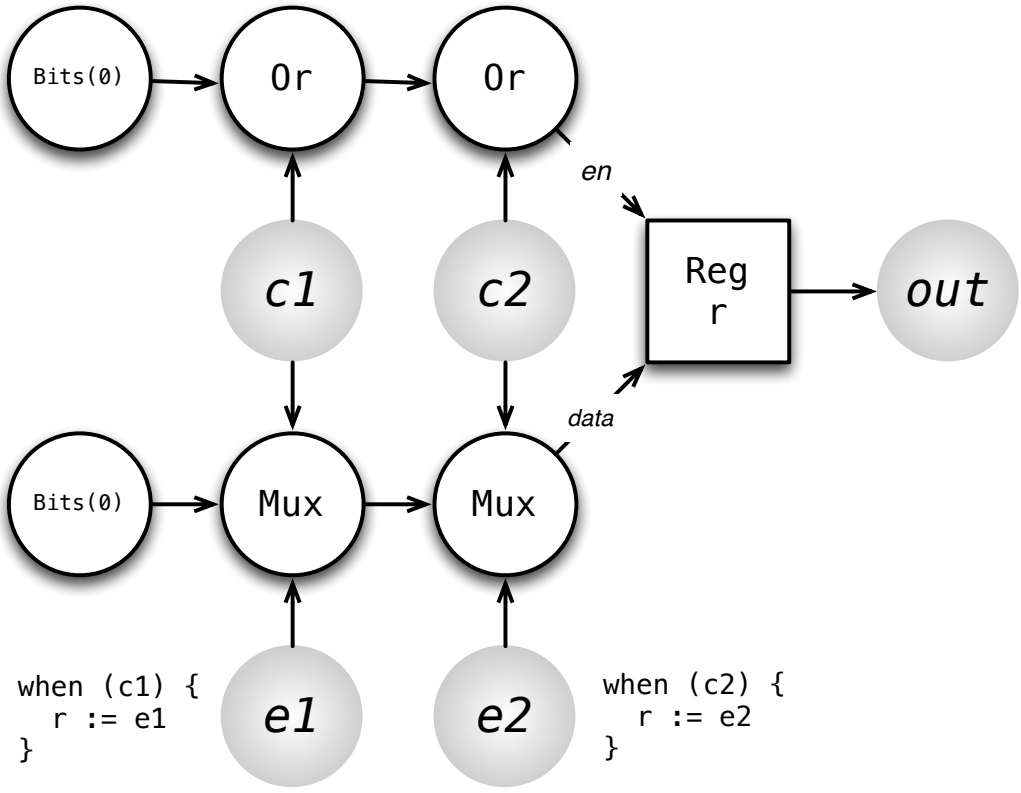
Conditional Updates Priority


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

Conditional Update Order:

c1	c2	r	
0	0	r	r unchanged
0	1	2	
1	0	1	
1	1	2	c2 takes precedence over c1

Conditional Update Synthesized Hardware



- 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.

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

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

c1	c2	r	s	
0	0	3	3	`r` updated in `c2` block, `s` updated using default
0	1	2	3	
1	0	1	1	
1	1	2	1	

Conditional Update Nesting


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

which is the same as:

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

Conditional Update Chaining


```
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


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

which is the same as:

```
when (idx === v1) { u1 }
when (idx === v2) { u2 }
```

Conditional Updates Everywhere

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:

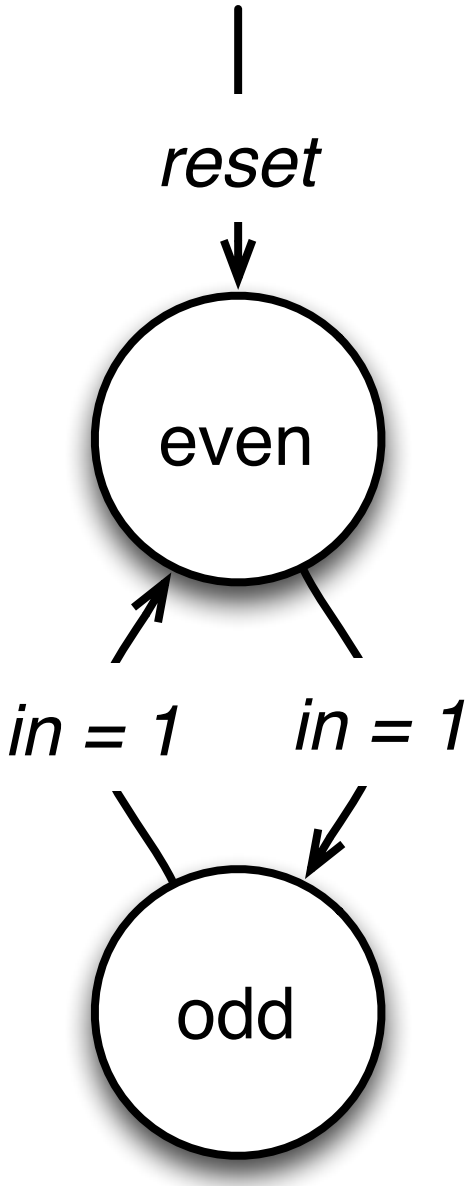
```
val w = Bits(width = 32)
w := Bits(0)                          // default value
when (c1)          { w := Bits(1) }
when (c2)          { w := Bits(2) }
```

which is the same as

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

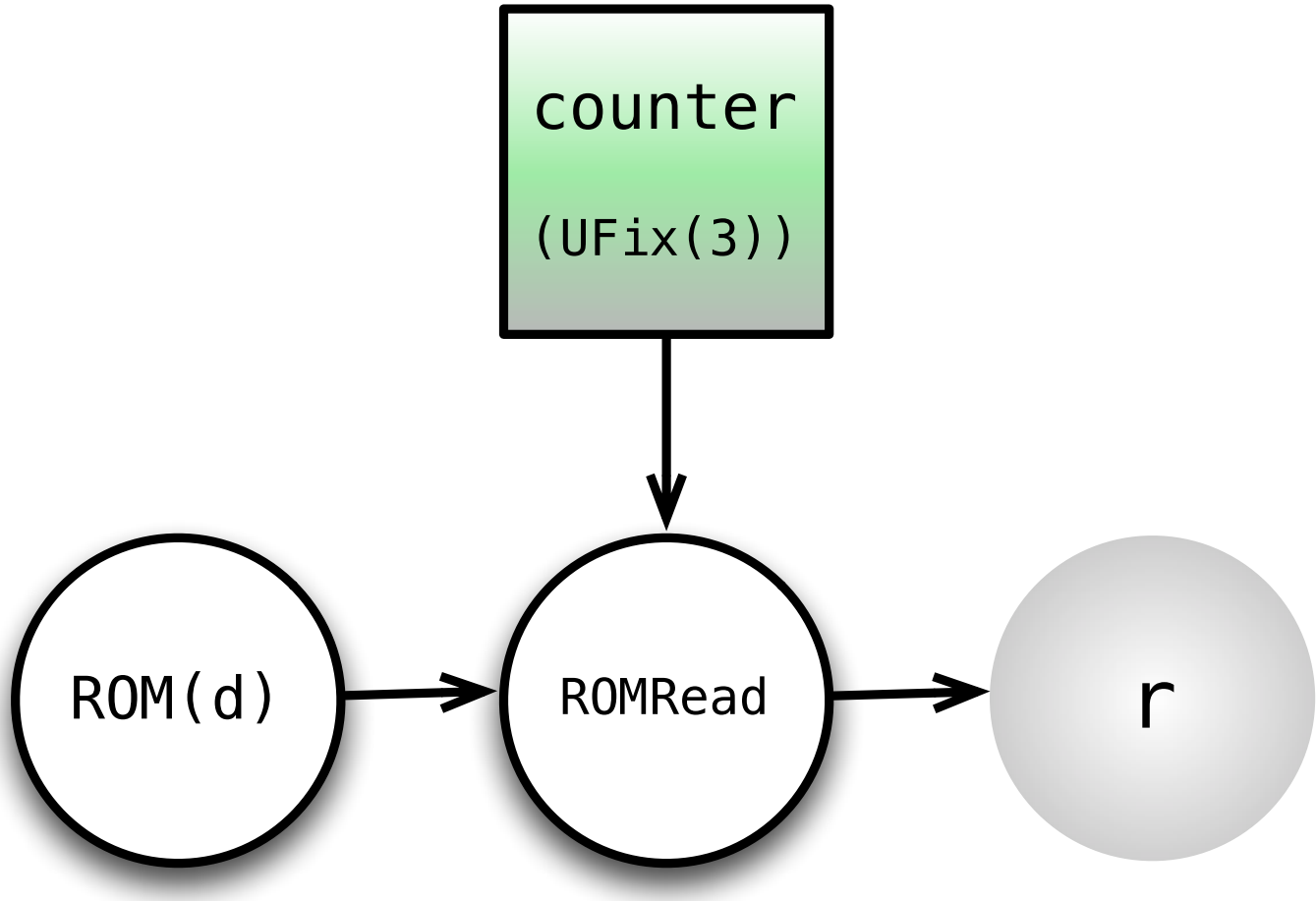
Finite state machines can now be readily defined as follows:

```
class Parity extends Component {
  val io = new Bundle {
    val in  = Bool(INPUT)
    val out = Bool(OUTPUT) }
  val s_even :: s_odd :: Nil = Enum(2){ UFix() }
  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)
}
```



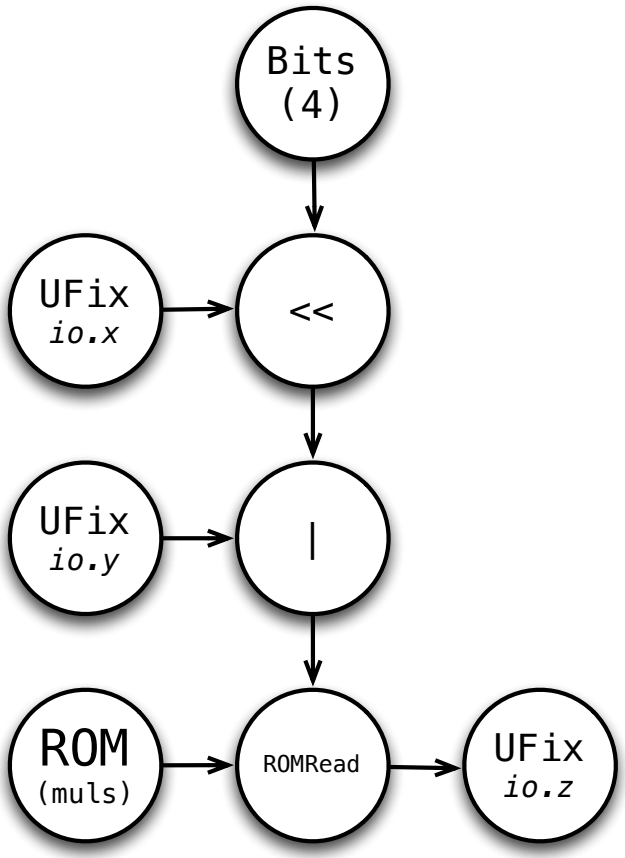
where `Enum(2){ UFix() }` creates a list of two `UFix()` literals.

```
val d = Array(UFix(1), UFix(2), UFix(4), UFix(8))
val m = ROM(d){ UFix(width = 32) }
val r = m(counter(UFix(3)))
```



Mul Lookup Table


```
class Mul extends Component {
  val io = new Bundle {
    val x   = UFix(INPUT, 4)
    val y   = UFix(INPUT, 4)
    val z   = UFix(OUTPUT, 8) }

  val muls = new Array[UFix](256)
  for (x <- 0 until 16; y <- 0 until 16)
    muls((x << 4) | y) = x * y

  val tbl = ROM(muls){ UFix(8) }

  io.z := tbl((io.x << 4) | io.y)
}
```



RAM is supported using the `Mem` construct

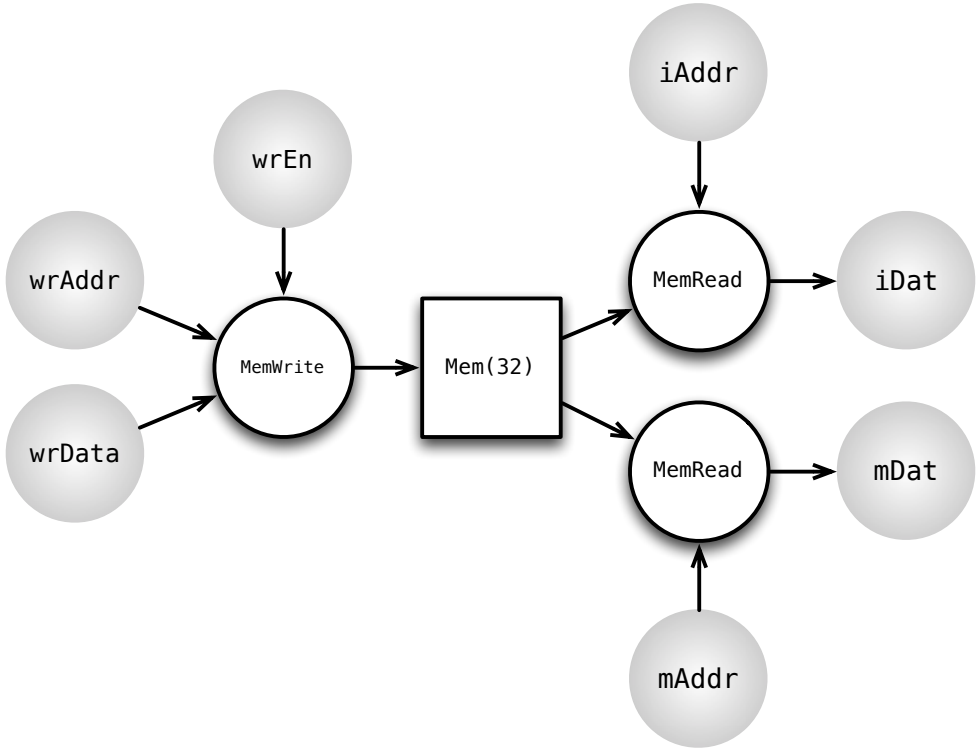
```
val m = Mem(32){ Bits(width = 32) }
```

where

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

32-entry Register File


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

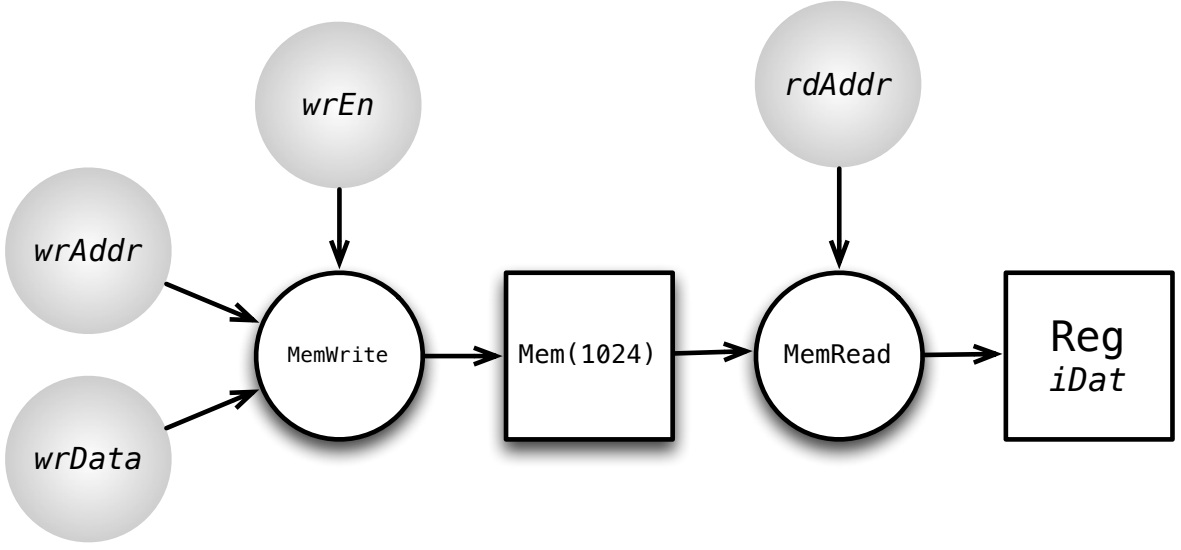


Sequential Read Ports

Sequential read ports are inferred when:

- optional parameter `seqRead` is set and
- a reg is assigned to the output of a MemRead

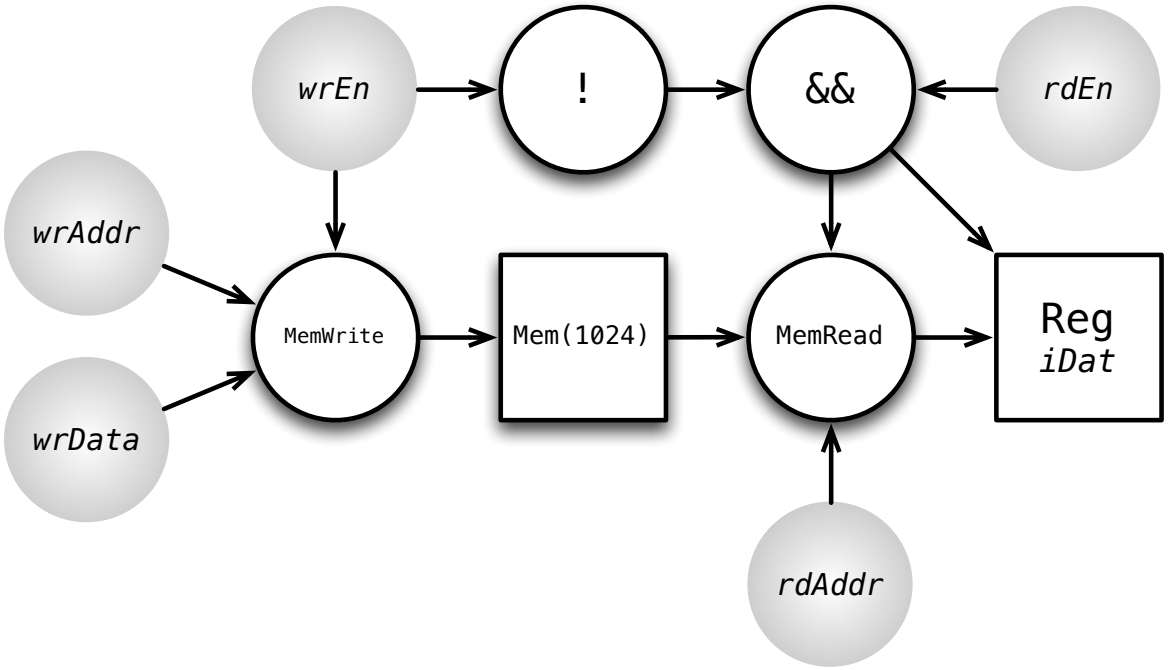
```
val ram1r1w = Mem(1024, seqRead = true) { Bits(width = 32) }
val dOut    = Reg() { Bits() }
when (wrEn) { ram1r1w(wrAddr) := wrData }
when (rdEn) { dOut := ram1r1w(rdAddr) }
```



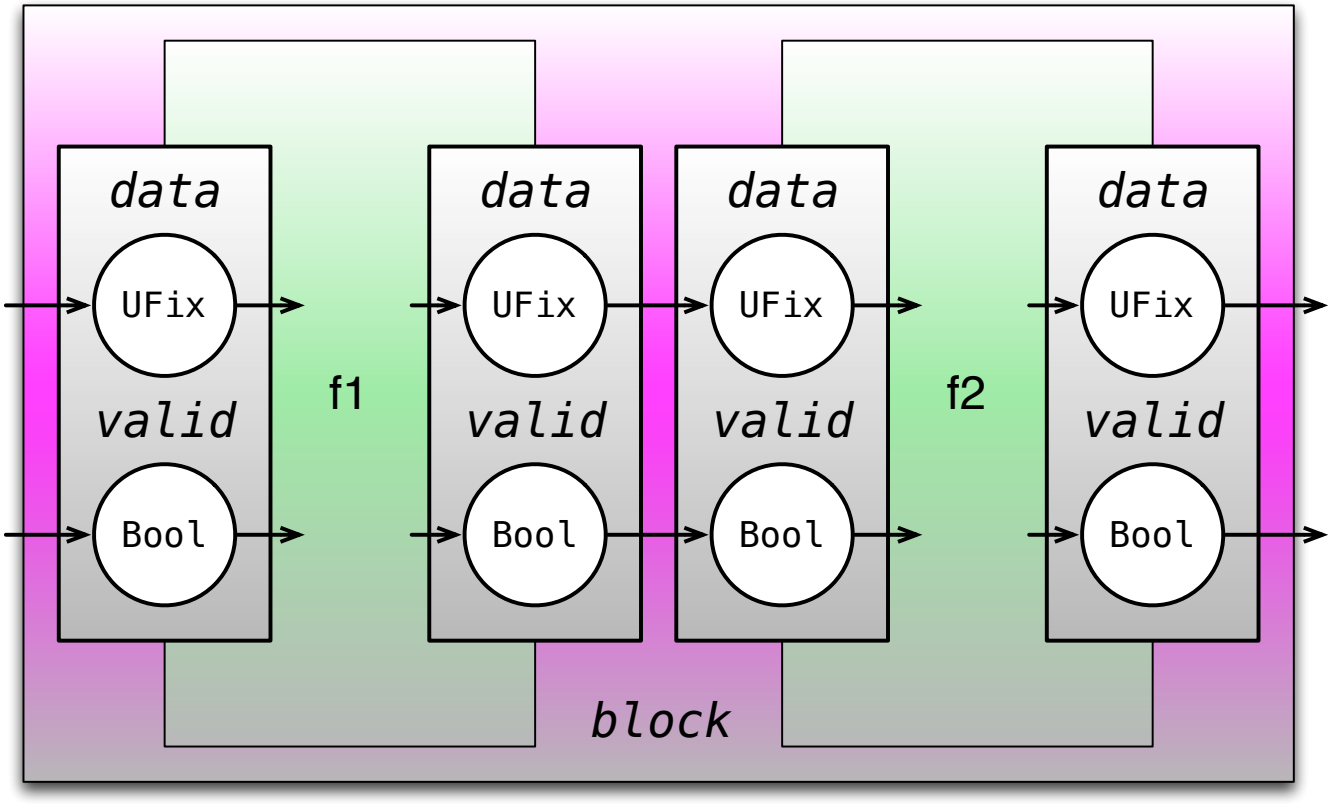
Single-ported SRAM

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 (wrEn) { ram1p(wrAddr) := wrData }
.elsewhen (rdEn) { dOut := ram1p(rdAddr) }
```



Suppose we want to break computation into a series of filters (ala Unix):

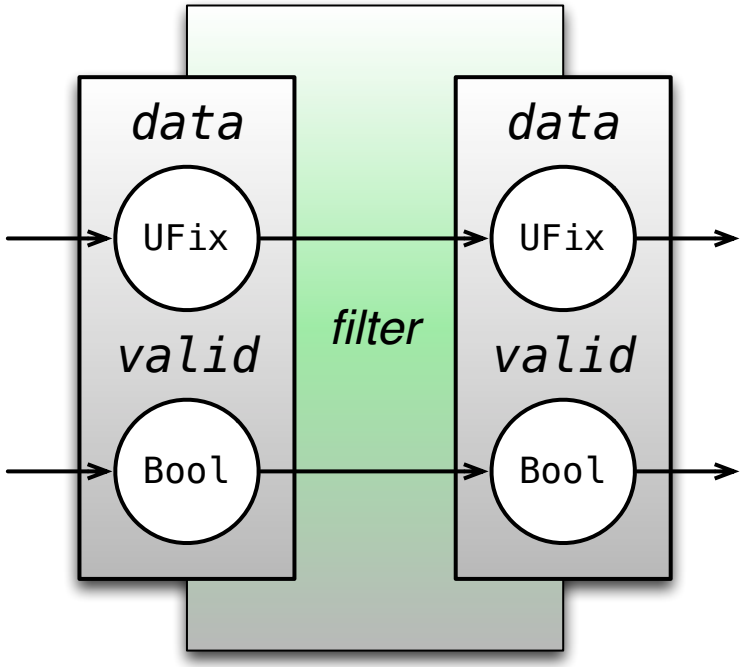


where data is fed though with an additional `valid` signal to say whether data has **not** been filtered.

Pass Through Filter

We can define a pass through filter component by defining a filter class extending component:

```
class Filter extends Component {
  val io  = new FilterIO()
  io.out.data  := io.in.data
  io.out.valid := io.in.valid
}
```



where the `io` field contains `FilterIO`.

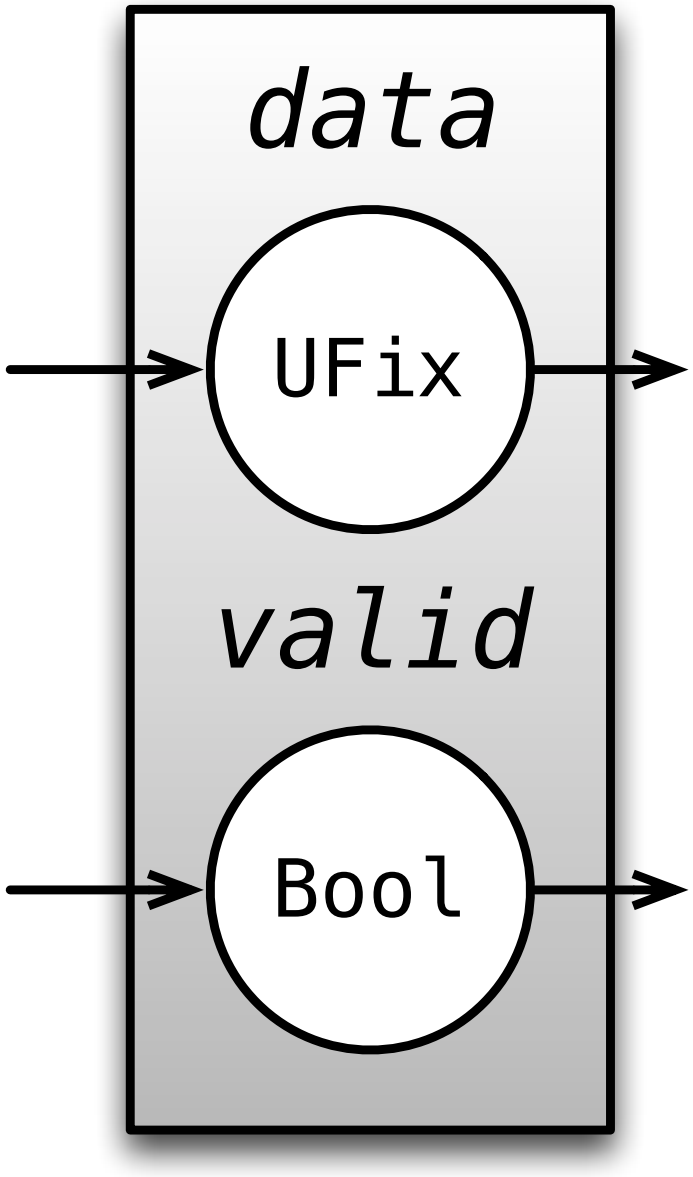
Small and Odd Filters

Suppose we want to write a small and odd filter. We could write these out by hand:

```
class SmallFilter extends Component {
  val io  = new FilterIO()
  io.out.data  := io.in.data
  io.out.valid := io.in.valid && (io.in.data < 10)
}

class OddFilter extends Component {
  val io  = new FilterIO()
  io.out.data  := io.in.data
  io.out.valid := io.in.valid && (io.in.data & 1)
}
```

Basic Filter Interface


```
class PipeIO extends Bundle {
  val data  = UFix(OUTPUT, 16)
  val valid = Bool(OUTPUT)
}
```

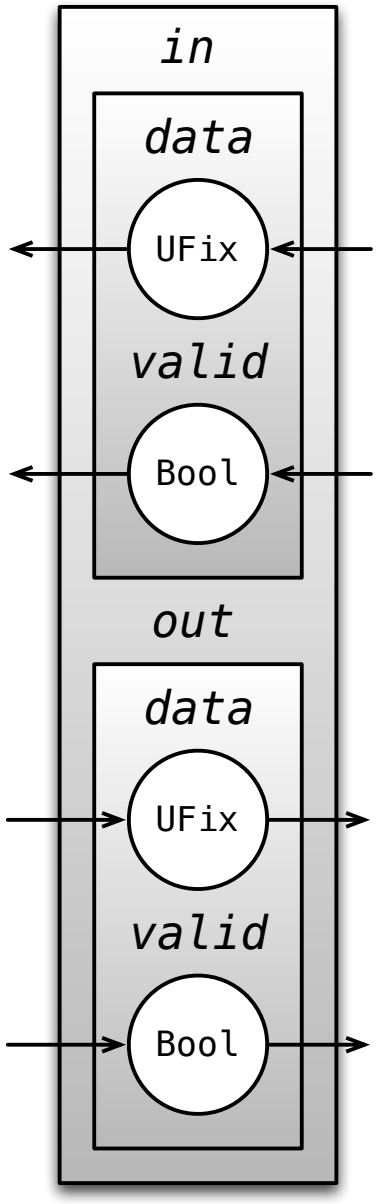


Complete Filter Interface

From there we can define a filter interface by nesting two `PipeIO`s into a new `FilterIO` bundle:

```
class FilterIO extends Bundle {
  val in  = new PipeIO().flip
  val out = new PipeIO()
}
```

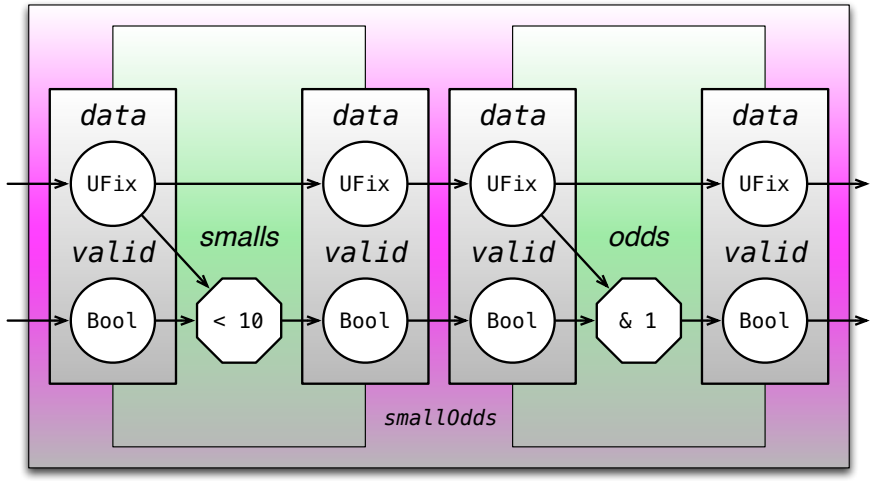
where `flip` recursively changes the "gender" of a bundle, changing input to output and output to input.



Bulk Connections

We can now compose two filters into a filter block as follows:

```
class SmallOdds extends Component {
  val io     = new FilterIO()
  val smalls = new SmallFilter()
  val odds   = new OddFilter()

  smalls.io.in  <> io.in
  smalls.io.out <> odds.io.in
  odds.io.out   <> io.out
}
```



where <> bulk connects interfaces. Note that:

- bulk connections recursively pattern match names between left and right hand sides finally connecting leaf ports to each other, and
- after all connections are made and the circuit is being elaborated, Chisel warns users if ports have other than exactly one connection to them.

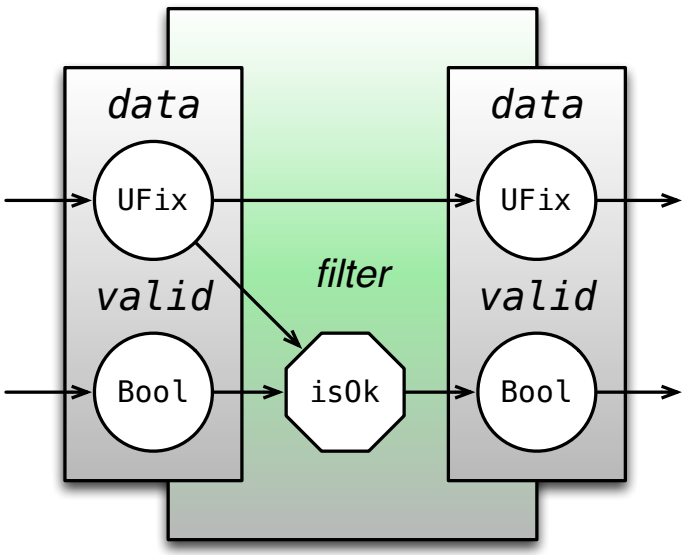
Congratulations, you have all that you need at this point to write Chisel programs! You can write RTL, define components (even with recursive data types), and wire them together.

- In order to attain true hardware description power though, you need to be able to write reusable RTL, components and interfaces.
- This will allow you to both use and write generic component 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!

Instead of writing a `SmallFilter` and `OddFilter`, a better Filter solution would be to create a single reusable Filter class that allows the user to specify the filter function. We can do this by

- specifying a filter function as a Filter constructor argument:

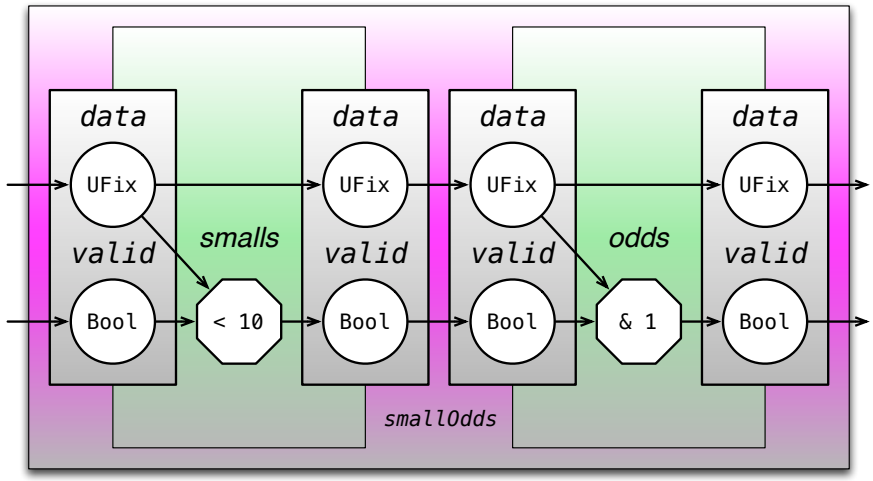
```
class Filter (isOk: (UFix) => Bool)
    extends Component {
  val io  = new FilterIO()
  io.out.data  := io.in.data
  io.out.valid :=
    io.in.valid && isOk(io.in.data)
}

val odds   = new Filter((x) => x & UFix(1))
val smalls = new Filter((x) => x < UFix(10))
```



SmallOdds built out of Parameterized Filters

We can now compose two parameterized filters into a filter block as follows:

```
class SmallOdds extends Component {
  val io     = new FilterIO()
  val smalls = new Filter(_ < 10)
  val odds   = new Filter(_ & 1)

  smalls.io.in  <> io.in
  smalls.io.out <> odds.io.in
  odds.io.out   <> io.out
}
```



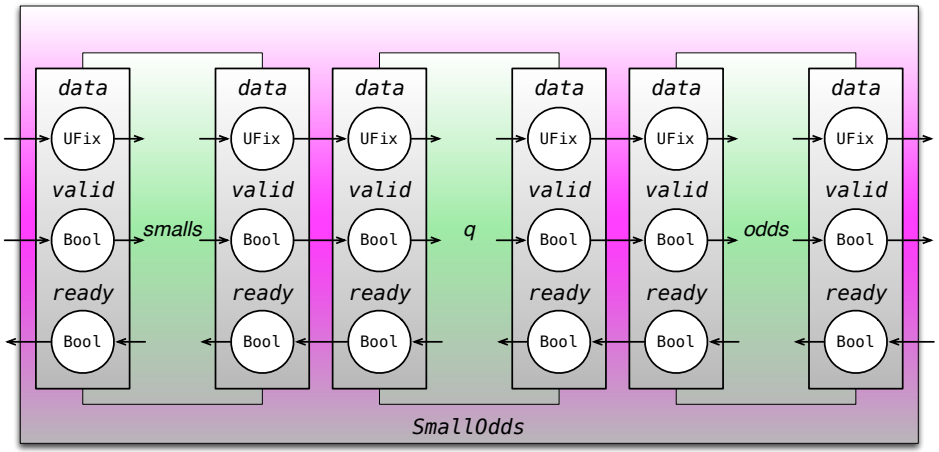
where `_ & 1` is a shorthand for `(x) => x & 1`.

Decoupled Filters

Suppose we want to make a block of two filters, where the filters take differing amounts of time to compute and need to be decoupled from each other. We can do this:

- by using decoupled interfaces with an additional ready signal and
- by connecting the two filters up using a queue

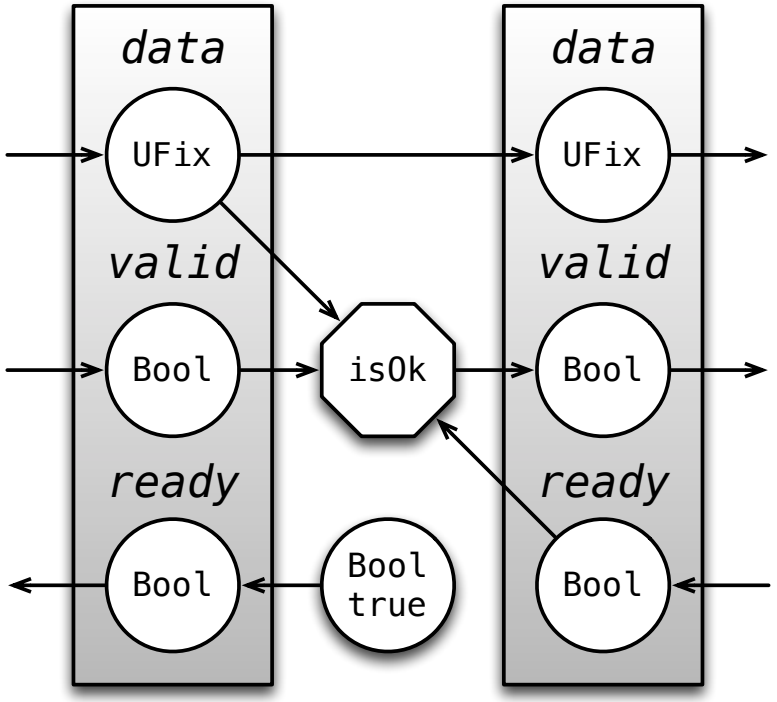
```
class SmallOdds extends Component {
  val io     = new FilterIO()
  val smalls = new Filter(_ < 10)
  val q      = new Queue()
  val odds   = new Filter(_ & 1)

  smalls.io.in  <> io.in
  smalls.io.out <> q.io.in
  q.io.out      <> odds.io.in
  odds.io.out   <> io.out
}
```



Decoupled Filter

Now filtering must consider back pressure:

```
class Filter (isOk: (UFix) => Bool)
    extends Component {
  val io  = new FilterIO()
  io.out.data  := io.in.data
  io.out.ready := Bool(true)
  io.out.valid :=
    io.out.ready &&
    io.in.valid &&
    isOk(io.in.data)
}
```



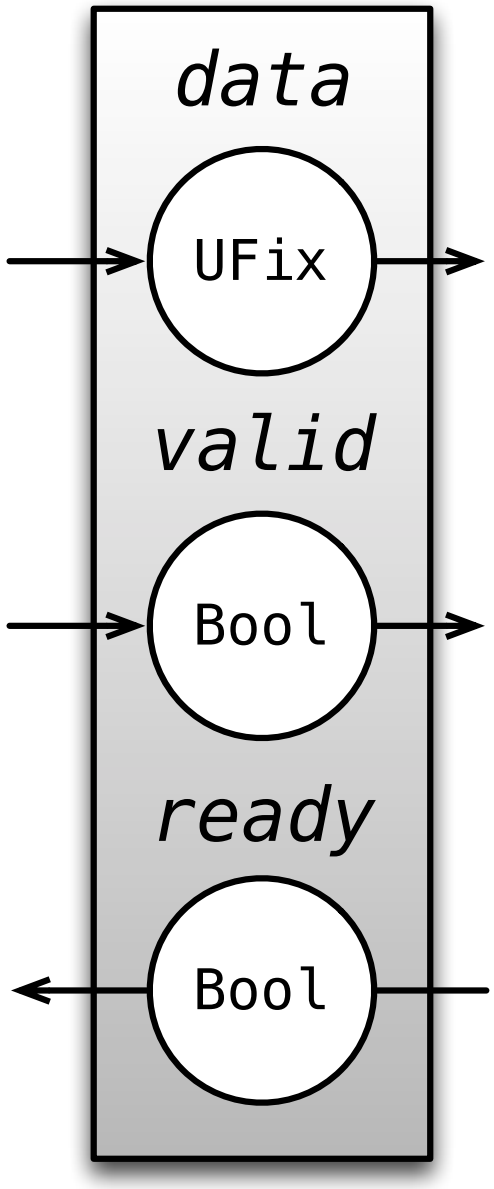
where

- the filter “fires” only when there is input data present and upstream fifos and filters are ready for output data, and
- in this version we are assuming that we are always ready for data.

We can define a decoupled interface by extending `PipeIO` with a ready signal:

```
class FIFOIO extends PipeIO {
  val ready = Bool(INPUT)
}
```

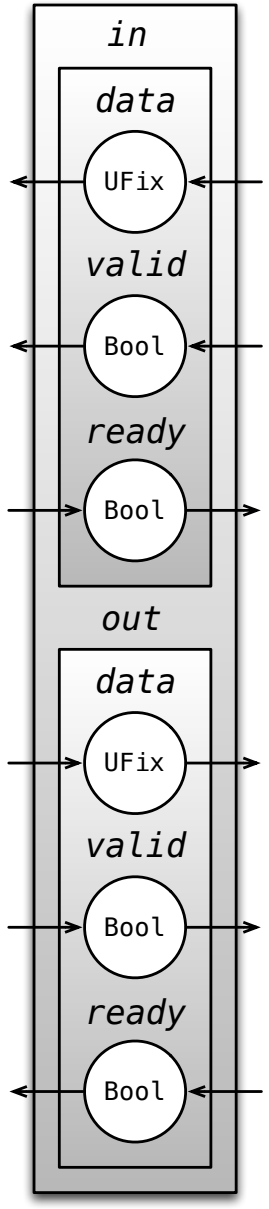
In general, users can organize their interfaces into hierarchies using inheritance in order to promote reuse.



Complete Decoupled Filter Interface

From there we can define a filter interface by nesting two `FIFOIO`s into a new `FilterIO` bundle:

```
class FilterIO extends Bundle {
  val in  = new FIFOIO().flip
  val out = new FIFOIO()
}
```



Unfortunately, as defined, decoupled interfaces are defined only for 16 bit `UFix`s. Obviously, we want to generalize this to allow for arbitrary Chisel data types. We can do this by using:

- Scala parameterized types and
- a curried class constructor argument

We want to be able to write

```
val ufix32s = new FIFOIO(){ UFix(width = 32) }

class Packet extends Bundle {
  val header = UFix(width = 8)
  val body   = Bits(width = 64)
}
val pkts    = new FIFOIO(){ new Packet() }
```

but how do we define this parameterized `FIFOIO`?

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:

```
def Mux[T <: Bits](c: Bool, con: T, alt: T): T { ... }
```

where

- `T` is required to be a subclass of `Bits` 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.

Curried Arguments

In Chisel we use special syntax for passing in a type constructor for parameterized types (such as Reg, Mem, and ROM). For example, for we can construct a reg using the following syntax:

```
val r = Reg(){ Bits(width = 32) }
```

You can write your own functions to allow this syntax and behavior as follows:

```
def myReg[T <: Data]()(type: => T) { ... }

myReg(){ Bits(width = 16) }
```

where the second parameter list has a single zero argument function parameter (aka thunk) that when called with no arguments produces a chisel type.

Now we can define `FIFOIO` and `FilterIO` using parameterized types and a curried argument as follows:

```
class FIFOIO[T <: Data]()(type: => T) extends Bundle {
  val data  = type.asOutput
  val valid = Bool(OUTPUT)
  val ready = Bool(INPUT)
}

class FilterIO[T <: Data]()(type: => T) extends Bundle {
  val in  = new FIFOIO(){ type }.flip
  val out = new FIFOIO(){ type }
}
```

We can now define `FIFOIO` on arbitrary data types:

```
val ufix32s = new FIFOIO(){ UFix(width = 32) }
val pkts    = new FIFOIO(){ new Packet() }
```

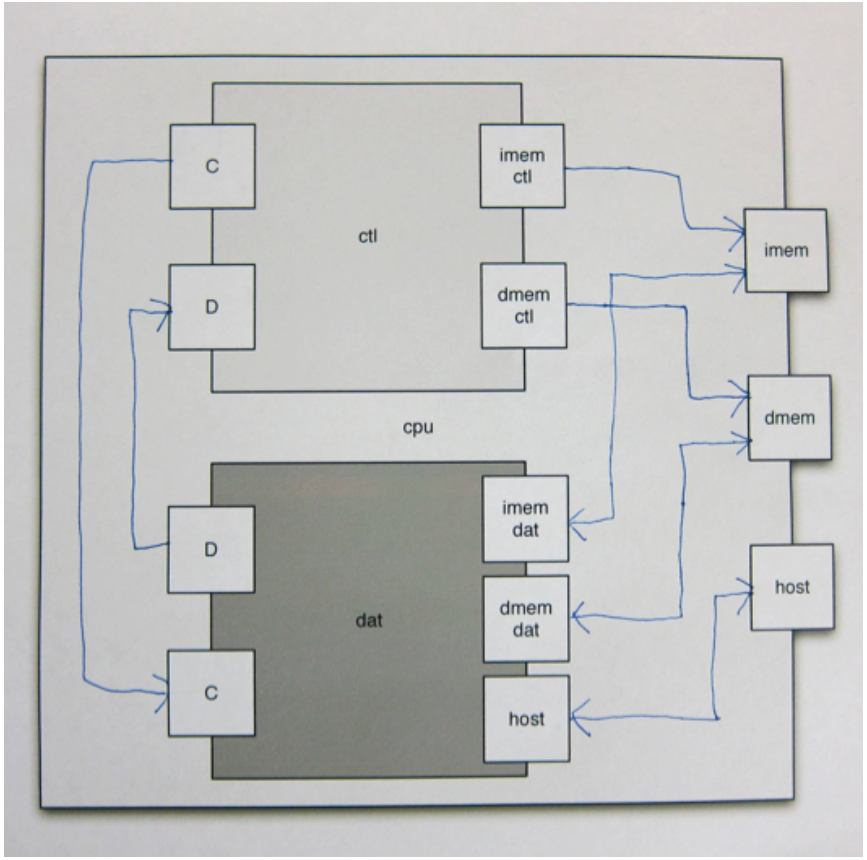
Fully Parameterized Filters

Now we can redo our definition of the `Filter` and `SmallOdds` components:

```
class Filter[T <: Data] (isOk: (T) => Bool) (type: => T) extends Component {
  val io = new FilterIO(){ type }

  io.out.data  := io.in.data
  io.out.ready := Bool(true)
  io.out.valid := io.out.ready && io.in.valid && isOk(io.in.data)
}

class SmallOdds[T <: Data]()(type: => T) extends Component {
  val io     = new FilterIO()
  val smalls = new Filter(_ < UFix(10)){ type }
  val q      = new Queue(){ type }
  val odds   = new Filter(_ & UFix(1)){ type }

  smalls.io.in  <> io.in
  smalls.io.out <> q.io.in
  q.io.out      <> odds.io.in
  odds.io.out   <> io.out
}

val block = new SmallOdds(){ UFix(width = 32) }
```

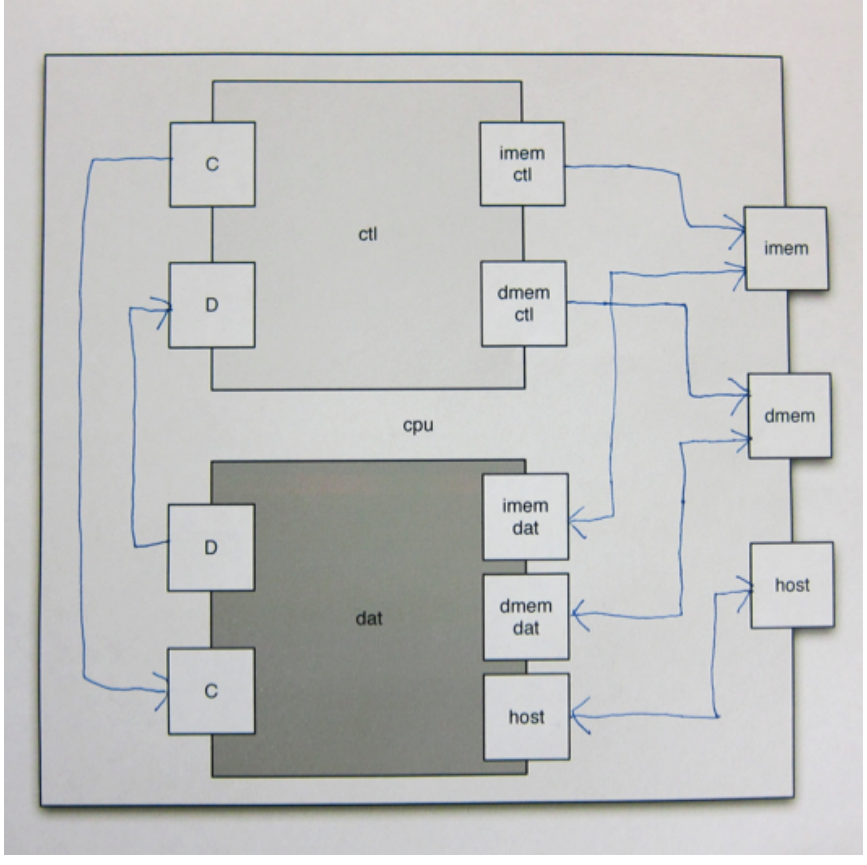
Interface Views


```
class Cpu extends Component {
  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
}
```



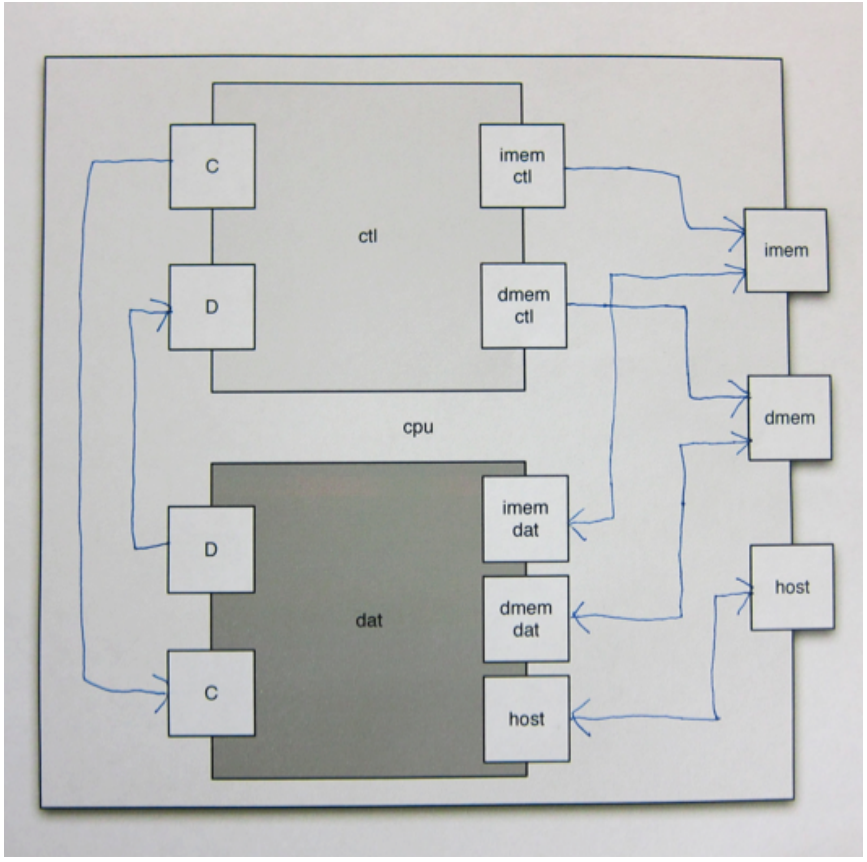
CPU Interfaces


```
class RomIo extends Bundle {
  val isVal = Bool(INPUT)
  val raddr = UFix(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()
  ... }
```



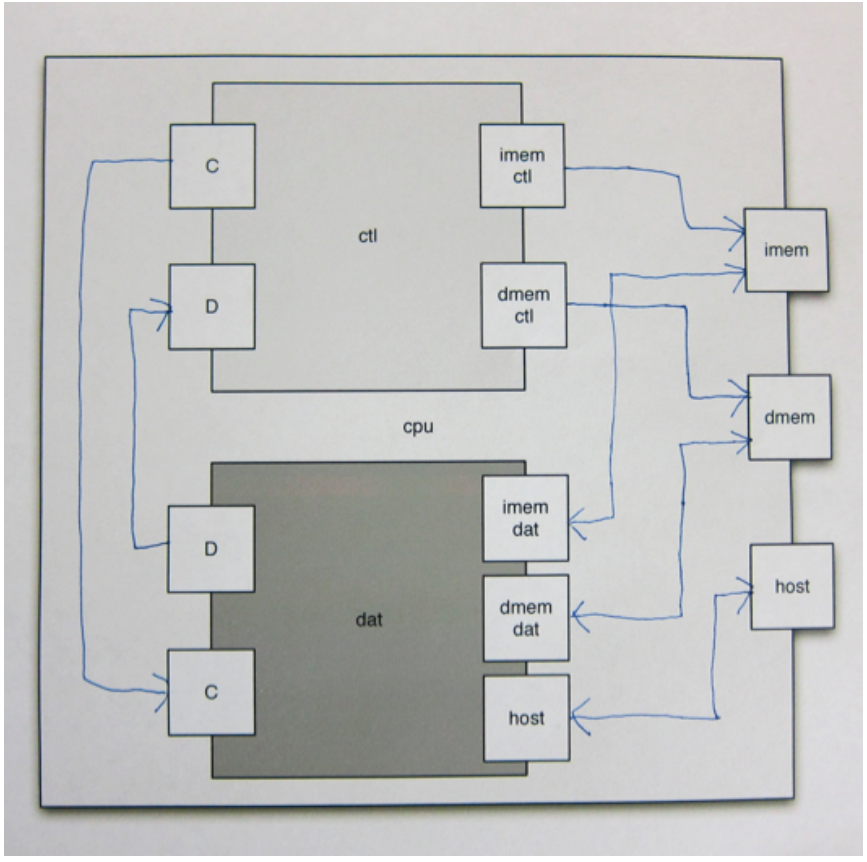
Partial Interface Fulfillment


```
class Cpath extends Component {
  val io = new CpathIo();
  ...
  io.imem.isVal := ...;
  io.dmem.isVal := ...;
  io.dmem.isWr  := ...;
  ...
}

class Dpath extends Component {
  val io = new DpathIo();
  ...
  io.imem.raddr := ...;
  io.dmem.raddr := ...;
  io.dmem.wdata := ...;
  ...
}
```



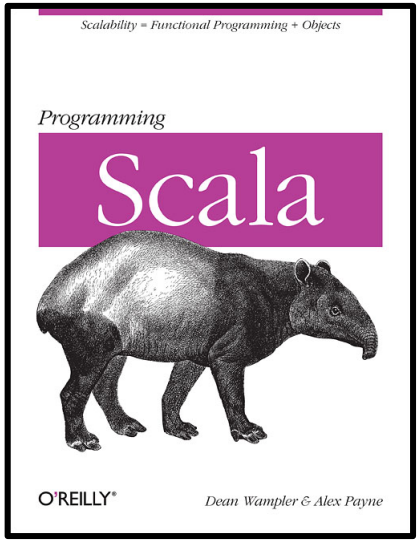
Multiple Partial Bulk Connections

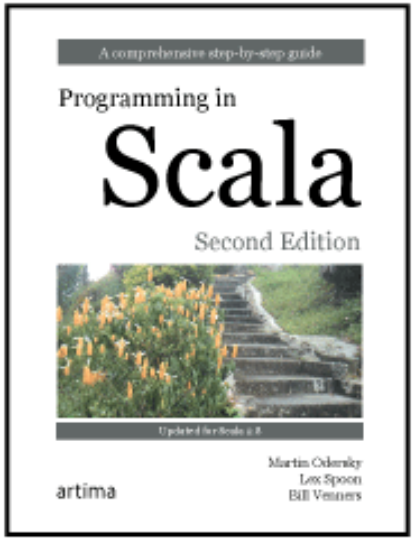

```
class Cpu extends Component {
  val io = new CpuIo()
  val c  = new CtlPath()
  val d  = new DatPath()
  c.io.ctl  <> d.io.ctl
  c.io.dat  <> d.io.dat
  c.io.imem <> io.imem
  d.io.imem <> io.imem
  c.io.dmem <> io.dmem
  d.io.dmem <> io.dmem
  d.io.host <> io.host
}
```



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

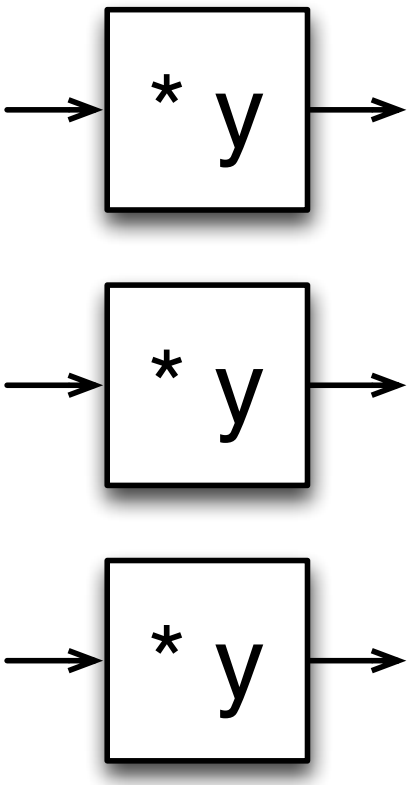




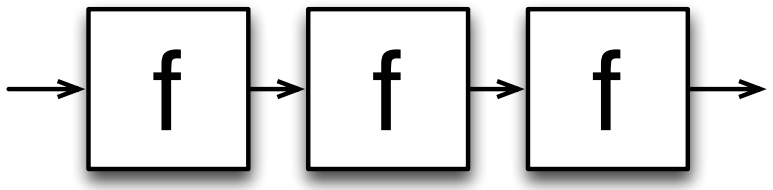
- Functional Composition
- Object Oriented Interfaces
- Layering Domain Specific Languages on Top

Functional Composition

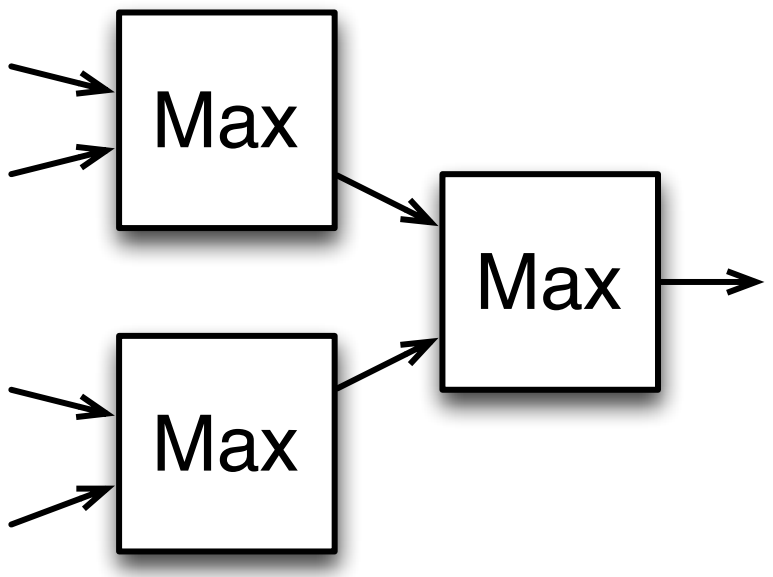

```
Map(ins, x => x * y)
```




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

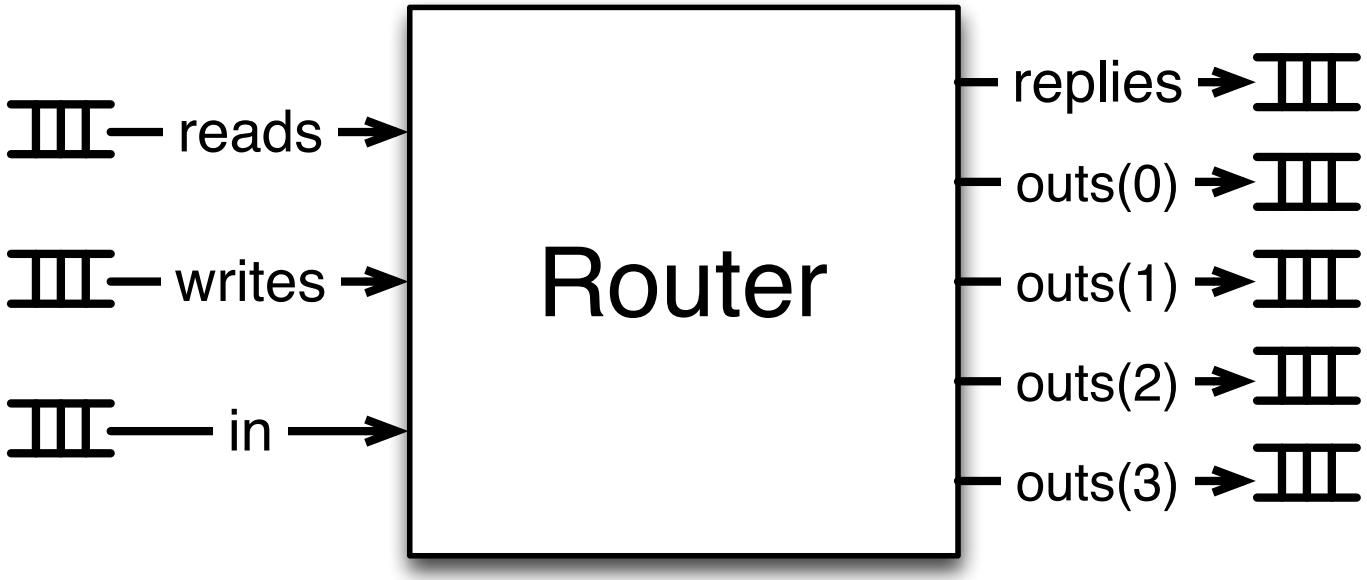



```
Reduce(data, Max)
```



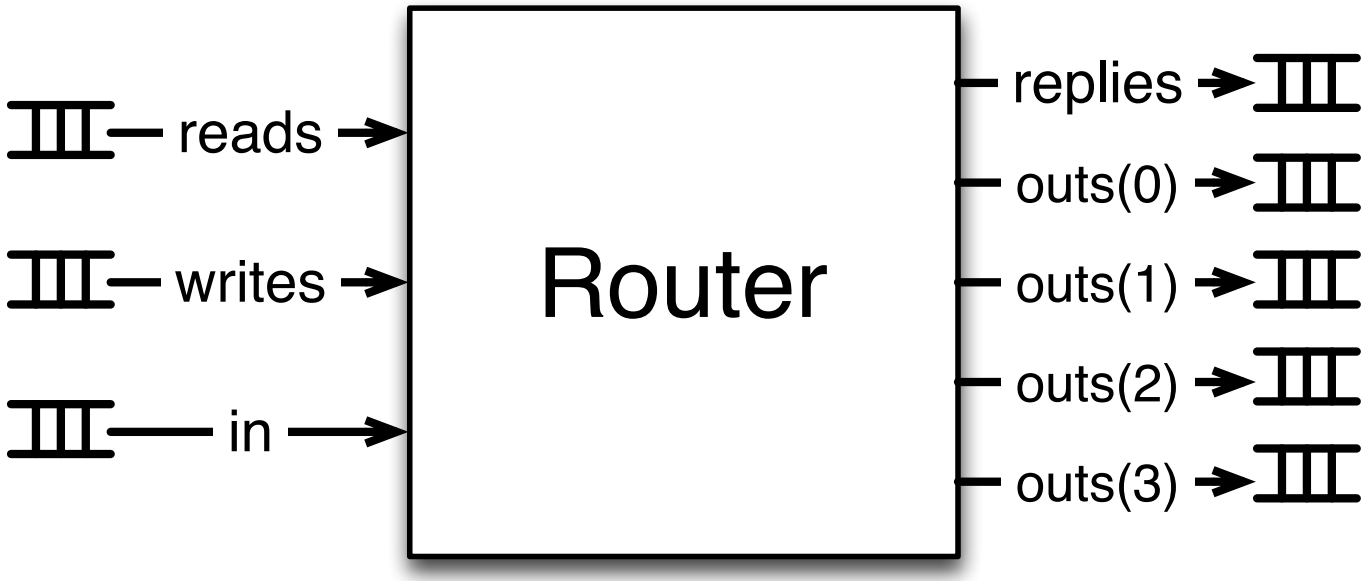
Object-Oriented Parameterized Interfaces


```
class Router extends Component {
  val depth = 32
  val n     = 4
  val io    = new RouterIO(n)
  val tbl   = Mem(depth){ UFix(width = sizeof(n)) }

  ...
}
```



Router Interface


```
class RouterIO(n: Int) extends Bundle {
  override def clone = new RouterIO(n).asInstanceOf[this.type]
  val reads   = new DeqIO(){ new ReadCmd() }
  val replies = new EnqIO(){ UFix(width = 8) }
  val writes  = new DeqIO(){ new WriteCmd() }
  val in      = new DeqIO(){ new Packet() }
  val outs    = Vec(n){ new EnqIO(){ new Packet() } }
}
```



Complete Router Interface


```
class ReadCmd extends Bundle {
  val addr = UFix(width = 32)
}

class WriteCmd extends ReadCmd {
  val data = UFix(width = 32)
}

class Packet extends Bundle {
  val header = UFix(width = 8)
  val body   = Bits(width = 64)
}

class RouterIO(n: Int) extends Bundle {
  override def clone = new RouterIO(n).asInstanceOf[this.type]
  val reads   = new DeqIO(){ new ReadCmd() }
  val replies = new EnqIO(){ UFix(width = 8) }
  val writes  = new DeqIO(){ new WriteCmd() }
  val in      = new DeqIO(){ new Packet() }
  val outs    = Vec(n){ new EnqIO(){ new Packet() } }
}
```

Router Guts


```
class RouterIO(n: Int) extends Bundle {
  override def clone = new RouterIO(n).asInstanceOf[this.type]
  val reads   = new DeqIO(){ new ReadCmd() }
  val replies = new EnqIO(){ UFix(width = 8) }
  val writes  = new DeqIO(){ new WriteCmd() }
  val in      = new DeqIO(){ new Packet() }
  val outs    = Vec(n){ new EnqIO(){ new Packet() } }
}

class Router extends Component {
  val depth = 32
  val n     = 4
  val io    = new RouterIO(n)
  val tbl   = Mem(depth){ UFix(width = sizeof(n)) }

  when(io.reads.valid && io.replies.ready) {
    val cmd = io.reads.deq();  io.replies.enq(tbl(cmd.addr))
  } .elsewhen(io.writes.valid) {
    val cmd = io.writes.deq(); tbl(cmd.addr) := cmd.data
  } .elsewhen(io.in.valid) {
    val pkt = io.in.deq(); io.outs(tbl(pkt.header(0))).enq(pkt)
  }
}
```

```
class EnqIO[T <: Data]()(data: => T) extends FIFOIO()(data) {
  def enq(dat: T): T = { valid := Bool(true); data := dat; dat }
  valid := Bool(false)
  for (io <- data.flatten.map(x => x._2))
    io := UFix(0, io.getWidth())
}

class DeqIO[T <: Data]()(data: => T) extends FIFOIO()(data) {
  flip()
  ready := Bool(false)
  def deq(b: Boolean = false): T = { ready := Bool(true); data }
}

class Filter[T <: Data]()(data: => T) extends Component {
  val io = new Bundle {
    val in  = new DeqIO(){ data }
    val out = new EnqIO(){ data }
  }
  when (io.in.valid && io.out.ready) {
    io.out.enq(io.in.deq())
  }
}
```