Chiselizing SDR Blocks

Albert Magyar

April 8, 2015

Why we're excited

- RFNoC shares many of our goals
  - High-level system design
  - Hardware-software integration
- Interesting applications in SDR
  - Chisel is great for DSP

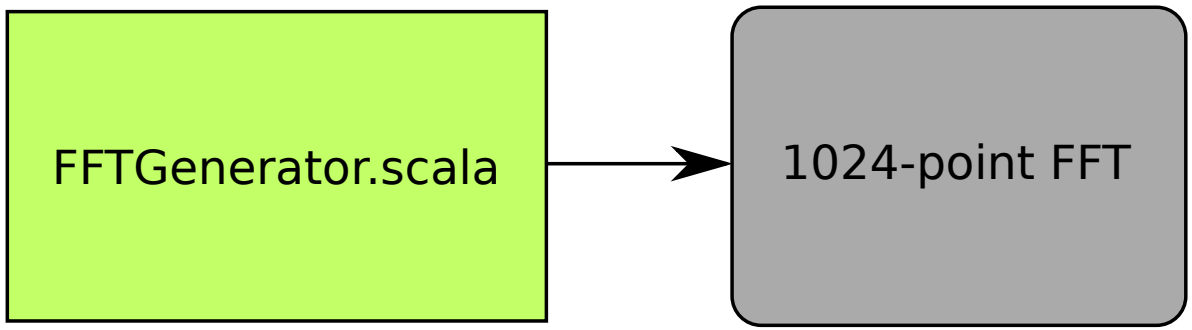
Why we're excited

- RFNoC shares many of our goals
  - High-level system design
  - Hardware-software integration
- Interesting applications in SDR
  - Chisel is great for DSP
  - Build our external user base!

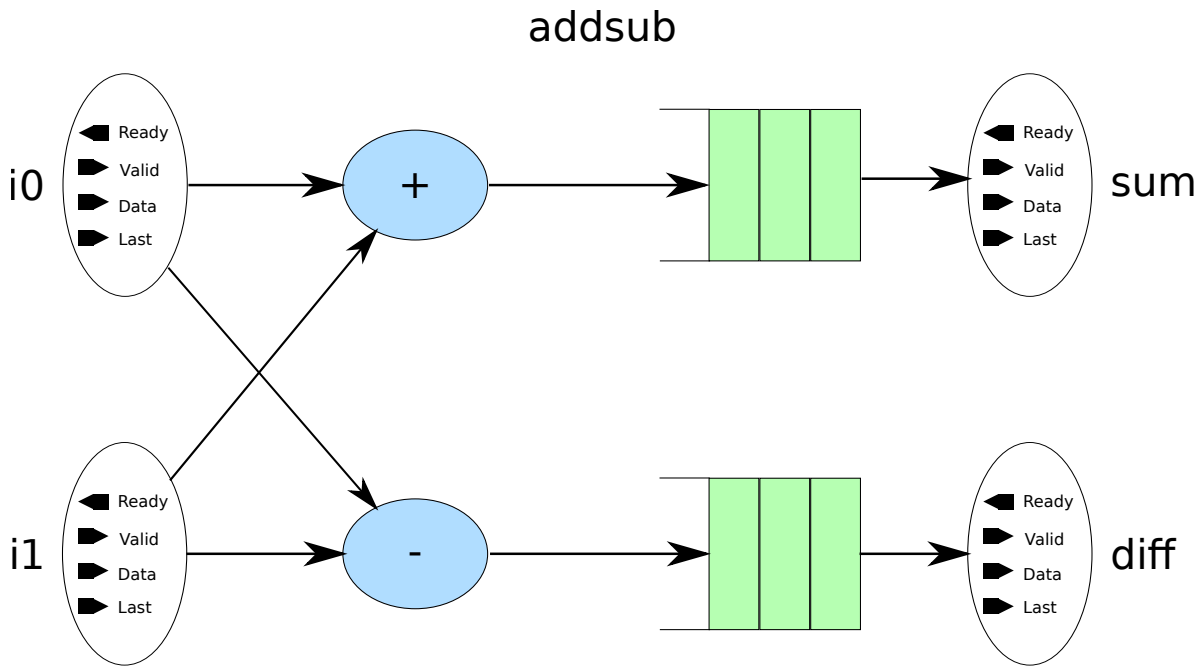
Goals for Chisel on the USRP

Release a Chisel SDK for USRP users

- Allow for a smooth transition from Verilog
- Provide a library of common components
- Support easy integration with Verilog ecosystem
- Increase user productivity!



The target: addsub block



Step 1: Porting Verilog

- Simple ports are easy
- No Scala magic
- Familiar RTL semantics

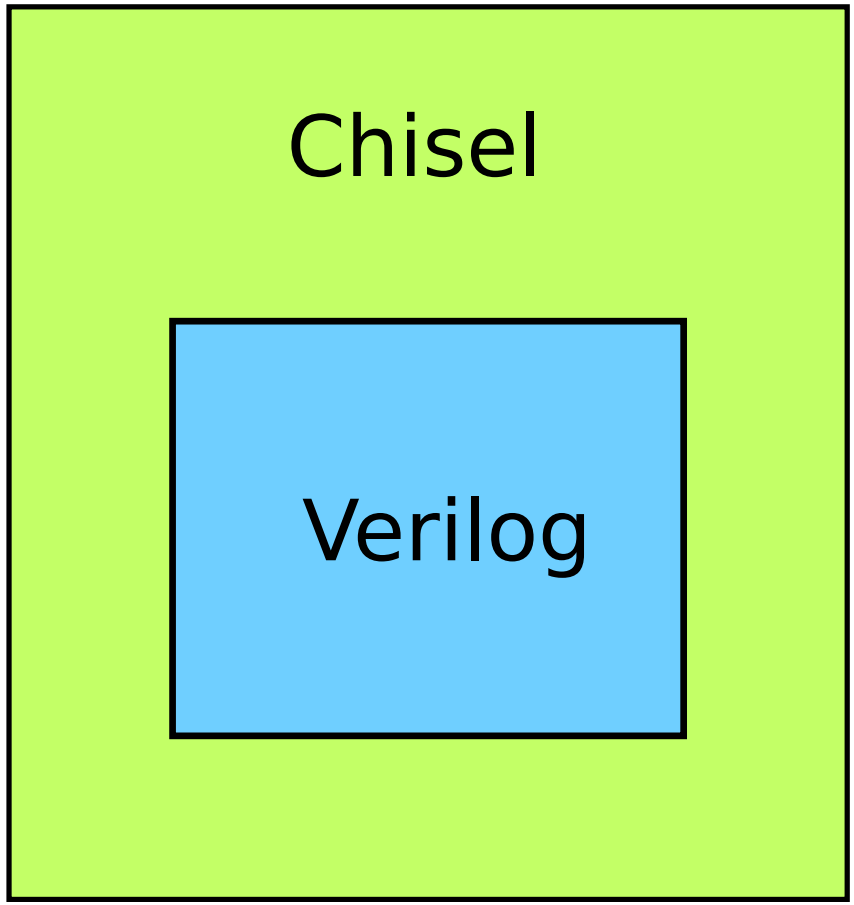
Verilog	Chisel

`if (en && incr)`
`  count <= count + 1;`
`else if (en && decr)`
`  count <= count - 1;`
`...`
	`when (en && incr) {`
	`  count := count + 1`
	`} .elsewhen (en && decr) {`
	`  count := count - 1`
	`...`

Step 1: Porting Verilog

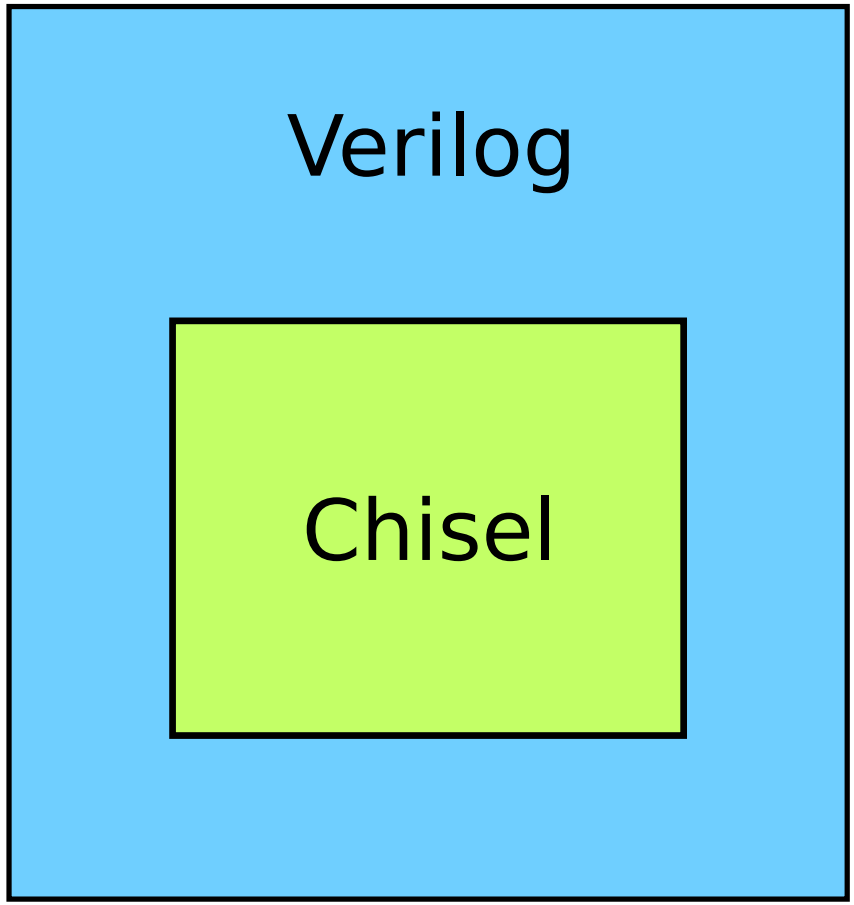
Code Example

Interfacing with Verilog



- Easily accomplished using `Blackbox` class
- Used in research tapeouts

Interfacing with Verilog

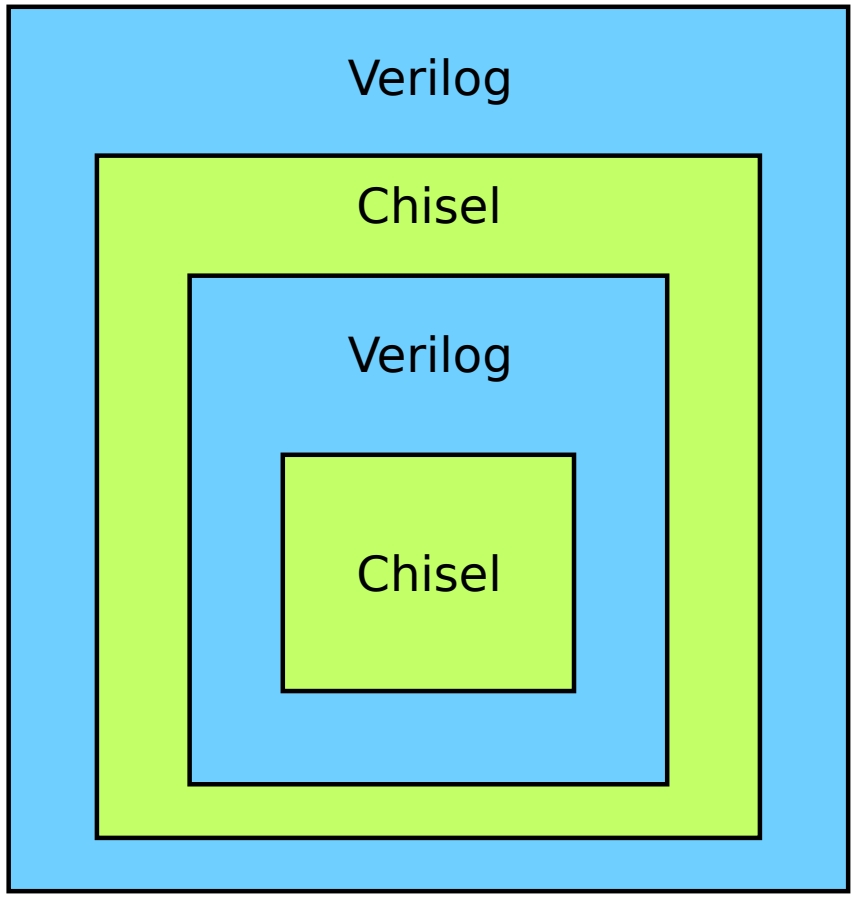


- Chisel backend produces Verilog
- Net names can be coerced with `setName`

Interfacing with Verilog

Verilog Generation Example

Interfacing with Verilog: no limits



- In practice, 3 layers has been used
- Verilog $<$ Chisel $<$ Verilog

Step *N*: Getting the most from Chisel

- Chisel has helped us define systems at every level
  - Reusability & flexibility in low-level RTL
  - Defining large systems on a chip
  - Producing research prototypes
- How can we apply Chisel-isms to `addsub`?
  - Rid code of boilerplate
  - Use parameters for flexible reuse

Step *N*: Getting the most from Chisel

```
class AddSubComplex(dataWidth: Int = 16) extends Module {
  val io = new AddSubComplexIO(dataWidth)
  val sum_q = Module(new Queue(Sample(dataWidth),16))
  val diff_q = Module(new Queue(Sample(dataWidth),16))

  sum_q.io.enq.bits := io.i0.bits + io.i1.bits
  diff_q.io.enq.bits := io.i0.bits - io.i1.bits
  SyncDecoupled(GroupDecoupled(io.i0,io.i1),
    GroupDecoupled(sum_q.io.enq,diff_q.io.enq))

  io.sum.bits := sum_q.io.deq.bits
  io.diff.bits := diff_q.io.deq.bits
  SyncDecoupled(GroupDecoupled(sum_q.io.deq,diff_q.io.deq),
    GroupDecoupled(io.sum,io.diff))
}
```

"On the wire" types

```
class AddSubComplexIO(dataWidth: Int = 16) extends Bundle {
  val i0 = new DecoupledIO(Sample(dataWidth)).flip
  val i1 = new DecoupledIO(Sample(dataWidth)).flip
  val sum = new DecoupledIO(Sample(dataWidth))
  val diff = new DecoupledIO(Sample(dataWidth))
}

class AddSubComplex(dataWidth: Int = 16) extends Module {
  ...
  val sum_q = Module(new Queue(Sample(dataWidth),16))
```

- Can define arbitrary signal types
  - `DecoupledIO`: generic FIFO interface
  - `Sample`: complex number, `last` bit
- Allows for generic modules!

Programmatically constructing circuit graphs

It's hard to package RTL as reusable pieces!

- Modules constrain users to rigid portlists
- Verilog functions are very limited
- Biggest danger: overhead of packaging a module deters designers!

Chisel allows *explicit* access to circuit graph

- Pointers to nets can be passed around
- Functions in Chisel perform any specific graph manipulation
- Modularize with or without "modules"

Programmatically constructing circuit graphs

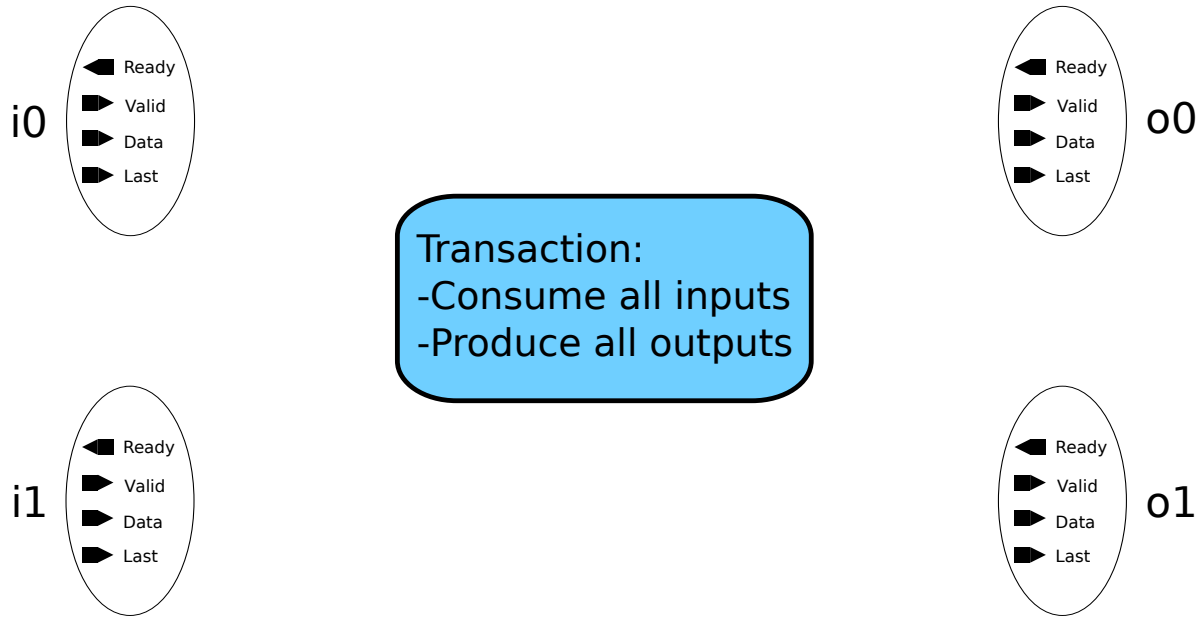
It's hard to package RTL as reusable pieces!

- Modules constrain users to rigid portlists
- Verilog functions are very limited

Chisel allows *explicit* access to circuit graph

- Pointers to nets can be passed around
- Functions in Chisel perform any specific graph manipulation

Programmatically constructing circuit graphs




```
SyncDecoupled(ins = ..., outs = ...)
```

- Adds necessary control logic to graph
- More flexible than module-level reuse

Using Scala

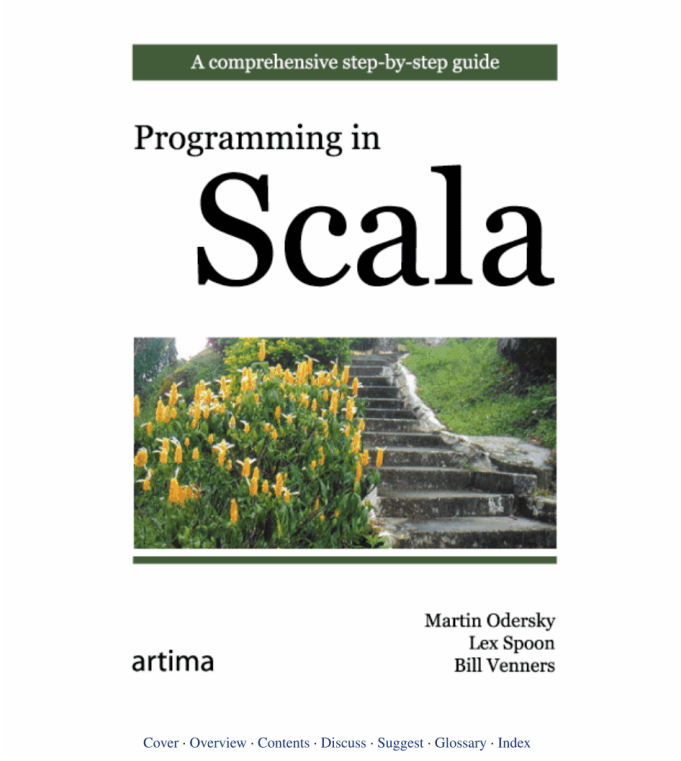
```
sum_q.io.enq.bits := io.i0.bits + io.i1.bits
diff_q.io.enq.bits := io.i0.bits - io.i1.bits
```

- Operator overloading makes this work
- Performs add using built-in `Complex`'s +
- `Complex` itself has overloaded operators

Using Scala

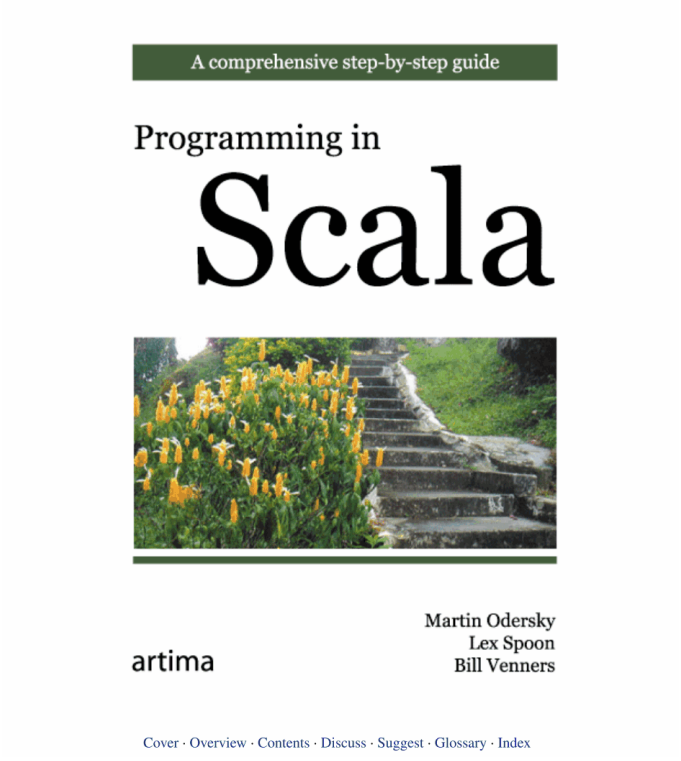
```
SyncDecoupled(ins = ..., outs = ...)
object SyncDecoupled {
  def apply(ins: Seq[DecoupledIO[Data]],
    outs: Seq[DecoupledIO[Data]]): Unit = {
    val allInsValid = ins.map(_.valid).reduce(_ && _)
    val allOutsReady = outs.map(_.ready).reduce(_ && _)
    for (in <- ins) {
      val otherInsValid =
        ins.filter(_ != in).map(_.valid).reduce(_ && _)
      in.ready := otherInsValid && allOutsReady
    }
    for (out <- outs) {
      val otherOutsReady =
        outs.filter(_ != out).map(_.ready).reduce(_ && _)
      out.valid := otherOutsReady && allInsValid
    }
  }
}
```

Does this imply a steep learning curve?



- Java-style OOP alone can unlock extreme capability

Does this imply a steep learning curve?



- Java-style OOP alone can unlock extreme capability
- We want to provide a library of powerful tools!

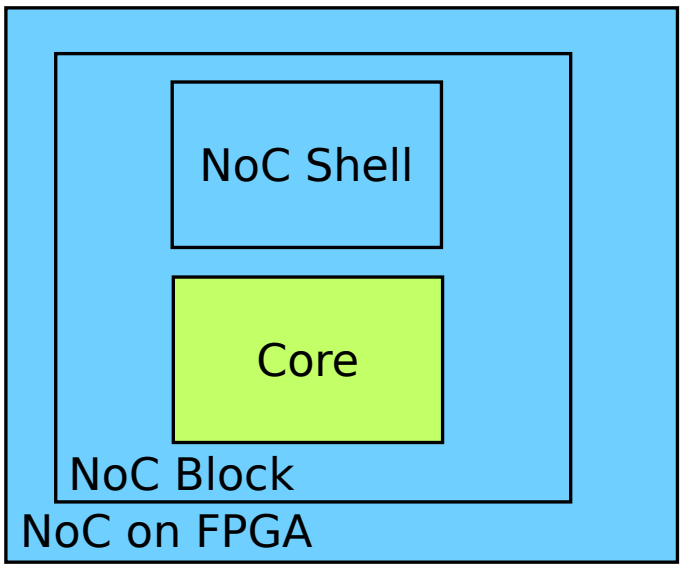
SDK: Powerful tools, simple interfaces

- Library of components and patterns
- Simple use helps newcomers from Verilog
- Generality helps experienced users

SDK: Powerful tools, simple interfaces

- Library of components and patterns
- Simple use helps newcomers from Verilog
- Generality helps experienced users
- Identifying the patterns helps us!
  - We're interested in the structure of DSP hardware
  - Can we identify a basis for quickly building systems?

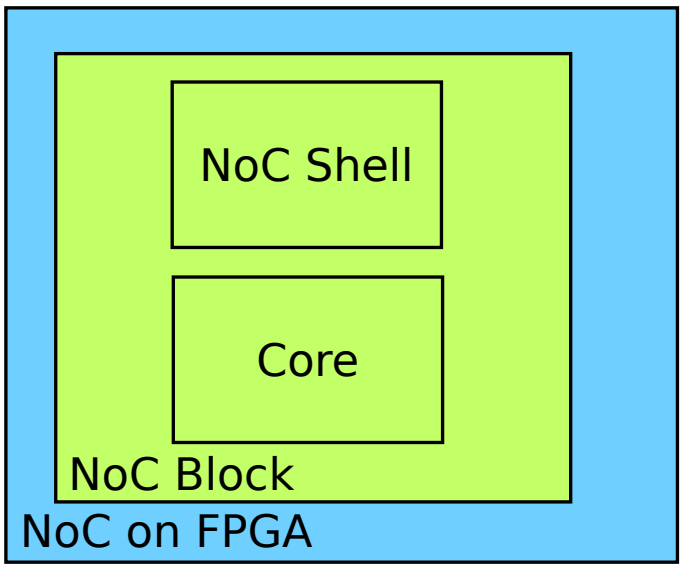
Chisel can fit at any level



Within the "core logic" of a block

- Easy to get started
- Library supports FIFO blocks

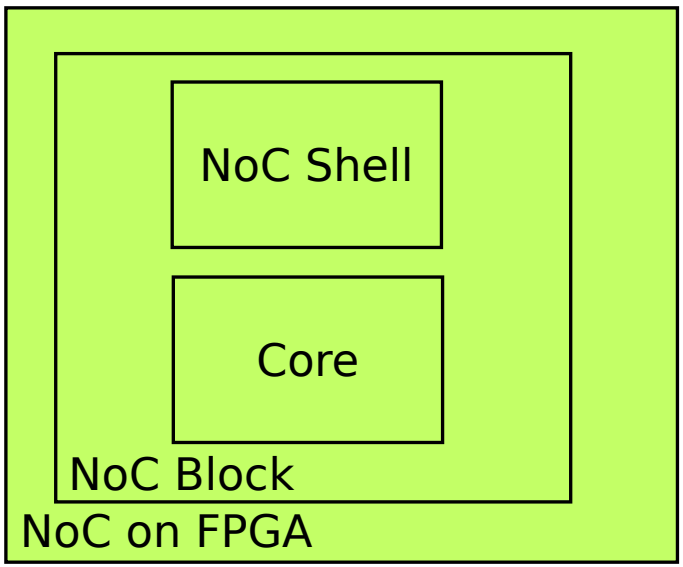
Chisel can fit at any level



Within an endpoint block

- NoC shell is highly parametrized – Chisel helps
- Amortize work of "language boundary crossings"

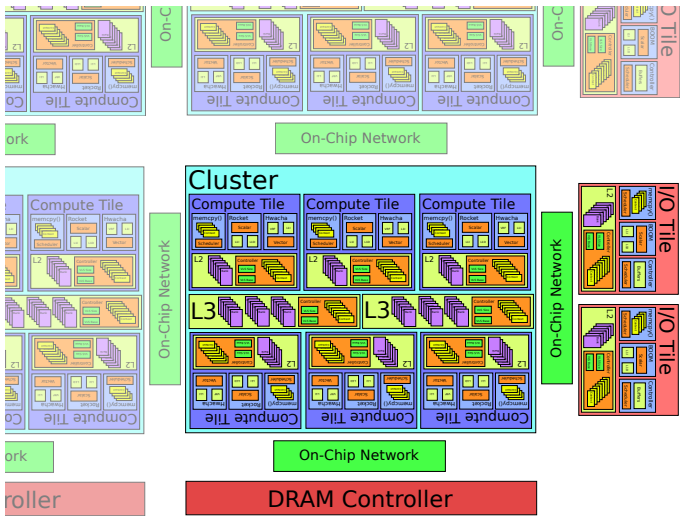
Chisel can fit at any level



Whole system RTL

- Chisel generators: flexible NoC configuration
- Parameters move easily though hierarchy

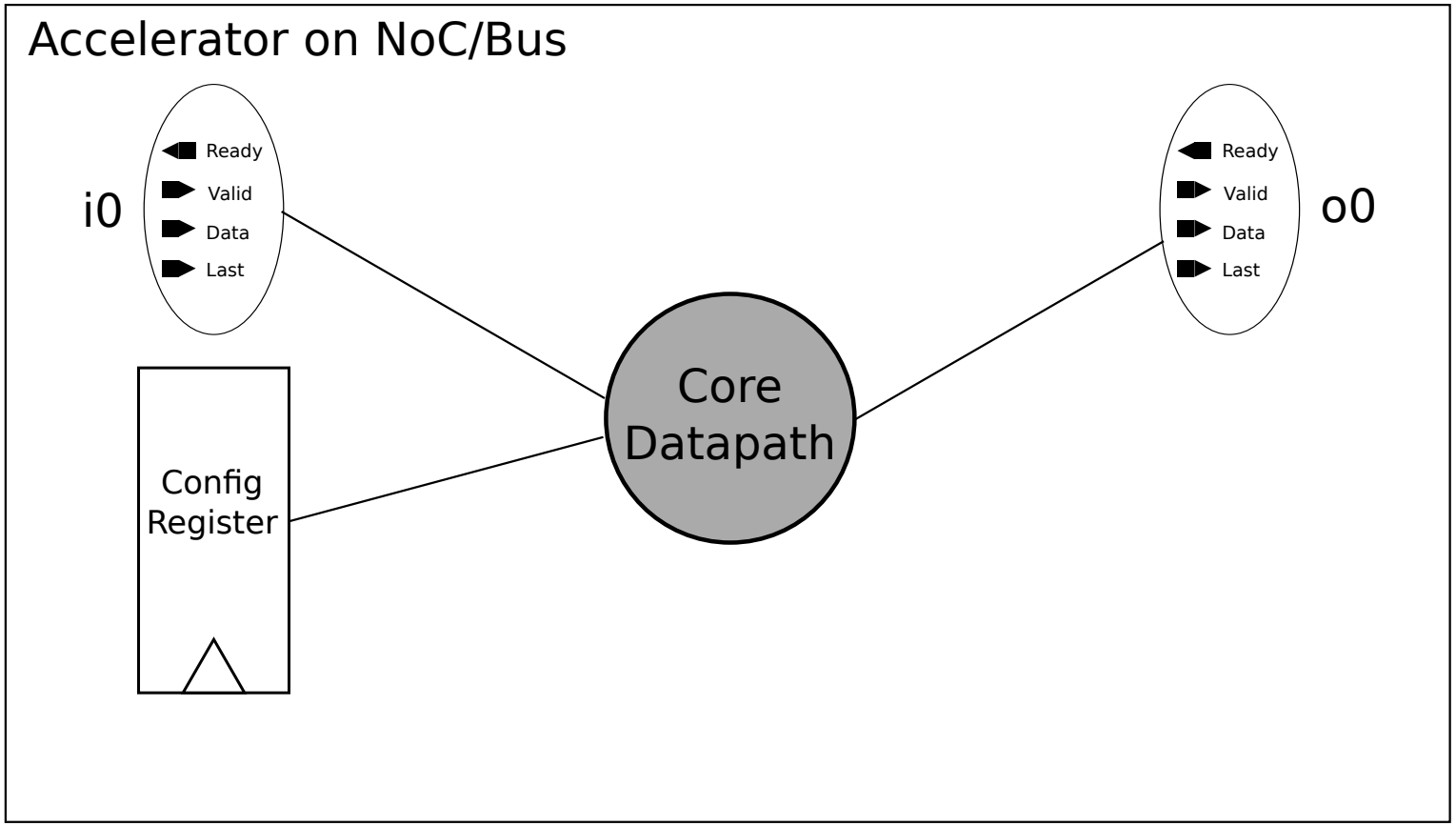
We're building solutions for DSP in software



- Hurricane: a spatial array for DSP applications
- Aims to be better than past embedded manycores
  - Better DLP with vector units
  - Co-existance of implicit and explicit locality
  - Support for cheap producer-consumer synchronization
  - And more!

We still like specialized hardware

We want to make designing accelerators easier



- Remove burden of platform-specific details
- Abstract non-performance-critical features (config, etc.)

Chisel above the RTL

- We work in the hardware-software co-design space
- The USRP model is great
  - GNU radio blocks stream across HW/SW
  - Boundaries are handled abstractly
- Where can Chisel fit in?
- How high-level of a system can it describe?