UCB Chisel Tutorial

Table of contents

How to Read a Chisel Program: The Router Circuit

Introduction -- A Chisel Router

This page will take you through a very detailed, nearly line by line walk through of the implementation of a Router and of the code that tests that implementation. Along the way there will be links to further discussions on specific topics, in particular, the relation ship between what is Chisel and what is Scala.

> The goal of this page is primarily to provide a reading knowledge of chisel, how to look at the code and break down what is what. Other docs will describe the development of Chisel code in more detail.

Our Router will support three operations:

- Rout: An input packet will be routed to a specific output based on the packet header. The output is configurable.
- Write: Sets the routing table, i.e. maps the packet header to a specific output
- Read: Allows the routing table to be read. This router has flow control, so operations will only be performed when input is valid and receiving outputs are ready. Let's get started.

The source files.

If you have gotten this far you probably already have understood the organization of a Chisel project. This write-up is about two files:

1. src/main/scala/examples/Router.scala
2. src/test/scala/examples/RouterTests.scala

The containing package

```
 1  // See LICENSE for license details.
 2
 3  package examples
 4
```

Line 1 `//` is a comment prefix, this comment is just advising users to read the accompanying license file.
Line 3 specifies the package name the package name. Package names should reflect the directory hierarchy of the containing files. See Scala: Things you should know

Imports

```
 5  import chisel3._
 6  import chisel3.util.{DeqIO, EnqIO, log2Ceil}
```

Imports tell the scala compiler to look for things that are not locally defined. Line 5 imports the bulk of chisel3 functionality, the _ is, in this context, a wild card directing the compiler to have access to all public code in the chisel3 package. It is a very good idea to always start by putting this line in. IDEs such as IntelliJ will try to automatically import things and in this process will sometimes import the alternative `import Chisel.<...>`. Capital Chisel in an import enables a backward compatibility mode that will allow use of some deprecated constructs. **Important:** always use chisel3 instead of Chisel unless you have very specific reasons to use compatibility mode. Line 6 imports some additional specific classes from the chisel3.util package. More on these later.

A Companion Object

```
 8  object Router {
 9    val addressWidth    = 32
10    val dataWidth       = 64
11    val headerWidth     =  8
12    val routeTableSize  = 15
13    val numberOfOutputs =  4
14  }
```

Here we are using a companion object `Router` as a place to define some useful constants for our project. A companion object is a *singleton* that is automatically instantiated. It's a good place to put constants. In this example, a number of constants are declared, `val addressWidth = 32`. `val` says that addressWidth cannot be changed, i.e. is a constant. This constant can be referenced elsewhere as `Router.addressWidth`. The name Router can still be used as a class name, in addition to it's use as an object name, as it is on line 36.

> We will use the common term *variable* for symbols defined by both *val* and *var* even though those defined by val cannot change.

Finally, Some Chisel, a Bundle

Before we get started here it might be a good idea to take a quick peek at A simple class example

```
16 class ReadCmd extends Bundle {
17   val addr = UInt(Router.addressWidth.W)
18 }
```

At last we have some actual Chisel (as opposed to Scala). Here, we define a Scala class **ReadCmd** Let's break it down.

- `class ReadCmd` begins the definition of **ReadCmd**.
- `extends Bundle` says this class is a subclass of **Bundle** or that **ReadCmd** *inherits* the properties of **Bundle**.
  - **Bundle** is defined in the Chisel library and is used to create a collection of hardware elements.
  - One of the primary uses of bundles is to define **IO** ports.
- The braces following Bundle contain the software components of the class. In this case only one line
  - `val addr = UInt(Router.addressWidth.W)` a member variable addr is created
  - addr is a reference to a UInt, this where some magic begins.
    - A **UInt** is a hardware type representing an unsigned integer.
    - **UInt** takes a width parameter. For historical and practical reasons this parameter is not an integer but must be a **Width**.
    - The notation `Router.addressWidth.W` takes an integer value **Router.addressWidth**, and uses the **.W** as a shorthand conversion of the Int type to a Chisel **Width** type.

Some more data bundles

```
20 class WriteCmd extends ReadCmd {
21   val data = UInt(Router.addressWidth.W)
22 }
23
24 class Packet extends Bundle {
25   val header = UInt(Router.headerWidth.W)
26   val body   = UInt(Router.dataWidth.W)
27 }
```

These follow straightforwardly from the **ReadCmd** above. Note: The **WriteCmd** Bundle extends the **ReadCmd Bundle** which means it inherits the properties of **Bundle** and **ReadCmd**. The **WriteCmd** ends up with two data fields, data and addr. **Packet** has two Chisel elements.

RouterIO

```
30   * The router circuit IO
31   * It routes a packet placed on it's input to one of n output ports
32   *
33   * @param n is number of fanned outputs for the routed packet
34   */
35 class RouterIO(n: Int) extends Bundle {
36   val read_routing_table_request   = DeqIO(new ReadCmd())
37   val read_routing_table_response  = EnqIO(UInt(Router.addressWidth.W))
38   val load_routing_table_request   = DeqIO(new WriteCmd())
39   val in                           = DeqIO(new Packet())
40   val outs                         = Vec(n, EnqIO(new Packet()))
41 }
```

Ok, things are getting a bit more interesting. First of all we saw a comment, it's about time, but we are trying to keep things succinct here. The **RouterIO** is a definition of the **IO** ports of our **Router** Module. Here we go

- `class RouterIO(n: Int) extends Bundle {` This class has a parameter **n**, that according to the comments is the number of fanned outpus. Perhaps in a more perfect world, one with perfect automatic variable name completion, **n** would have been named **numberOfFannedOutputs**. **n** will be used in the Bundle to create the desired outputs

- `val read_routing_table_request = DeqIO(new ReadCmd())` there are a number of things going on here.
    - We are now using the **ReadCmd** discussed earlier, we are creating an instance of it using the Scala **new** keyword
    - More interestingly we have wrapped the `new ReadCmd()` in a **DeqIO**.
        - The **DeqIO** adds **Ready/Valid** flow control or decoupled behavior
        - The flow control will be used to by the **Router** module to dequeue read requests from the outside world.
        - **DeqIO** adds to read_routing_table_request:
            - **valid** as **Bool** input port, ready as **Bool** output port
                - **Bool** is a basic Chisel type that can only take on literal values **true.B** or **false.B** values
                - **true.B** being shorthand for **Bool(true)** and **false.B** shorthand for **Bool(false)**
            - a deq() method that will return the incoming **ReadCmd**
            - a nodeq() method that will assert false on ready
- `val read_routing_table_response = EnqIO(UInt(Router.addressWidth.W))` here we see **EnqIO** that like **DeqIO**
    - Adds **ready** and **valid** to a **UInt** port, but with the directionality reversed.
    - an **enq()** method that places a UInt on the port
    - a **noenq()** method that false on valid
- `val load_routing_table_request = DeqIO(new WriteCmd())` provides the ports for decoupled **load_routing_table** ports
- `val in = DeqIO(new Packet())` defines a decoupled port for reading packets to be routed.
- `val outs = Vec(n, EnqIO(new Packet()))` defines the outputs that incoming packets will be routed to.
    - **Vec** is a Chisel aggregate that allows for a collection of identically typed elements.
        - we use the **n** parameter of **RouterIO** to create the proper number of output ports.
        - each element of is a enqueue decoupled port with a packet, ready and valid ports.

How to Read the Router Example Continued

How to Read a Chisel Program: The Router Circuit, Continued

Here comes the DUT

```
/**
  * routes packets by using their header as an index into an externally loaded and readable table,
  * The number of addresses recognized does not need to match the number of outputs
  */
```

```
class Router extends Module {
```

As before we are creating a Scala class that in this case inherits from the Chisel **Module** class. **Modules** are generators of hardware descriptions, analogous to the **Modules** of Verilog etc. **Modules** should define IO ports, reset and clock behavior, and the circuit implementation.

```
  val depth = Router.routeTableSize
  val n     = Router.numberOfOutputs
  val io    = IO(new RouterIO(n))
  val tbl   = Mem(depth, UInt(BigInt(n).bitLength.W))
```

These first few lines define some variables that are really little more than alias for the more verbose parameters that are contained in the **Router** object. This is one way of passing in parameters, we have seen the alternative in **RouterIO** in which a parameter is passed as an argument to the class creation. We shall see this below. The way parameters are passed into module is somewhat a matter of taste. In general larger and more complex systems like **Rocket** have powerful and complex parameterization schemes, with implicit defaults and notation for local overrides.

```
  when(reset) {
    io.read_routing_table_request.nodeq()
    io.load_routing_table_request.nodeq()
    io.read_routing_table_response.noenq()
    io.in.nodeq()
    io.outs.foreach { out => out.noenq() }
  }
```

Starting with `when(reset)`, **when** is the hardware equivalent of an if statement. The contents of **when**'s code block (the stuff in the braces) will be generated with any connections to hardware components outside the **when** block gated with the necessary muxes based on, in this case, **reset**, in this case it is the **Bool** signal reset. **reset** is an implicit IO **Bool** input port, that is provided by the **Module** (There are other types of **Module** that can be used that do not provide these ports automatically)

```
  .elsewhen(io.load_routing_table_request.valid) {
    val cmd = io.load_routing_table_request.deq()
    tbl(cmd.addr) := cmd.data
    printf("setting tbl(%d) to %d\n", cmd.addr, cmd.data)
  }
```

The elsewhen here is pretty much like an `else if`, the code blocks connections will be gated by muxes based on the condition. In this case that condition is when the load_routing_table.valid signal is asserted. `val cmd = io.load_routing_table_request.deq()` creates a local reference to the return value of `deq()`. Dequeue is a simple convenience method on a decoupled interface that sets the ready, and returns a reference to the data portion of the decouple interface. `tbl(cmd.addr) := cmd.data` sets the memory at the specified address cmd.addr to the specified output cmd.data. Because this is in an elsewhen block that connection will only happen when the condition is true. There is another printf here, probably used by the developer during the debugging process.

> A note on the dot in front of **elsewhen**. The **when** operator is implemented a via companion object **when** that has an **apply** method that requires Bool parameter and a code block as its argument. That **apply** method returns an an instance of a class **WhenContext**. The **WhenContext** instance has a method **elsewhen** which, like when, requires a Bool and a code block. At **FIRRTL** generation time these additional blocks are emitted in a way that they are dependent upon the negation of all preceding connected **when** and **elsewhen** blocks. Dots are not required in front of methods when they follow on the same line, but for style reasons this developer chose to put the elsewhen on it's own line. Thus the dot in front of the **elsewhen** was necessary for the scala compiler to recognize it as a method.

```
  .elsewhen(io.in.valid) {
    val pkt = io.in.bits
    val idx = tbl(pkt.header(log2Ceil(Router.routeTableSize), 0))
    when(io.outs(idx).ready) {
      io.in.deq()
      io.outs(idx).enq(pkt)
      printf("got packet to route header %d, data %d, being routed to out(%d)\n", pkt.header, pkt.body, tbl(pkt.header))
    }
  }
}
```

> The final **.elsewhen** is gated by **io.in.valid**. The `val pkt = io.in.bits` gets a reference to the data in the incoming packet. It does not used the `deq()` method here, because we don't want to assert the ready on the input until we the output associated with the packet header is ready to be enqueued. `val idx = tbl(pkt.header(log2Ceil(Router.routeTableSize), 0))` figures out the index (idx) that is associated with the packet header according to the current routing table. The `tbl(pkt.header(log2Ceil(Router.routeTableSize), 0))` is doing a couple of notable things. **tbl** (the routing table) is being indexed by the packet header, the parenthesized argument to header is a specification of bits of the header to use as the index, the argument to the implicit apply method on **pkt.header** is (high start bit, low end bit) inclusive. The **log2Ceil** computes the high from the number of bits necessary to accommodate the parameterized size of the routing table.

Once the index is computed, a `when(io.outs(idx).ready) {` checks the ready bit on that output port and if so, `io.in.deq()` assert the ready on the input port indicating the value has packet had been processed and `io.outs(idx).enq(pkt)` puts the packet on the selected input and asserts valid on that port.