UCB-BAR Chisel Tutorial

These are the tutorials for Chisel.

Chisel is an open-source hardware construction language developed at UC Berkeley that supports advanced hardware design using highly parameterized generators and layered domain-specific hardware languages.

The source for this tutorial is https://github.com/ucb-bar/chisel-tutorial/wiki

This chapter is an installation guide for *Chisel* (Constructing Hardware In a Scala Embedded Language) and is intended to prepare your system for subsequent tutorials. Chisel is a hardware construction language embedded in the high-level programming language Scala.

Development Tool Installation

The chisel tutorials should be able to be run on a variety of modern systems. For the most update instructions on preparing your machine for Chisel see Chisel Installation Preparation

Setting Up the Tutorial

In subsequent tutorials, you will be using the files distributed in the chisel-tutorial repository. To obtain these tutorial files, `cd` to the parent of the directory where you want to place the Chisel tutorial and type:

```
export DIR=~/chisel-workspace
mkdir -p $DIR
cd $DIR
```

```
git clone https://github.com/ucb-bar/chisel-tutorial.git
cd chisel-tutorial
```

You will now be in your copy of the Chisel Tutorial repository will then be in **$DIR/chisel-tutorial**. For later convenience, define this as a variable in your bash environment named `TUT_DIR`:

```
export TUT_DIR=$DIR/chisel-tutorial
```

This is the Chisel tutorial directory structure you should see, which is explained more in the next tutorial:

```
chisel-tutorial
 - build.sbt # project description
 - doc/
 - src/
   - main/
     - scala/
       - examples/
         - More advanced example circuits are found here
       - problems/
         - A series of problem circuits to be completed by the user
       - solutions/  # solutions to problems
         - Solutions to the problems circuits
   - test/
     - resources/
       - Resources required by some of the tests
     - scala/
       - examples/
         - Test harnesses for the example circuits
       - problems/
         - Test harnesses for the problem circuits
       - solutions/
         - Test harnesses for the solution circuits
       - util/
         - TutorialRunner.scala
```

The Chisel Directory Structure

Once you have acquired the tutorial files you should see the following Chisel tutorial directory structure under `$TUT_DIR`:

```
chisel-tutorial/
  build.sbt # project description
  run-examples.sh   # shell script to execute one or more examples
  run-problem.sh    # shell script to execute one or more problems
  run-solution.sh   # shell script to execute one or more solutions
  src/
    main/
```

```
        scala/
          examples/   # chisel examples
            Accumulator.scala ...
          problems/   # skeletal files for tutorial problems
            Counter.scala ...
          solutions/  # solutions to problems
            Counter.scala ...
      test/
        resources/
          in.im24
          in.wav
       scala/
          examples/   # examples testers
            Adder.scala ...
          problems/   # problems testers
            Accumulator.scala ...
          solutions/  # solutions testers
            Accumulator.scala ...
          util/
            TutorialRunner.scala
```

Chisel source files are distributed between `examples`, `problems`, and `solutions` directories. The tutorial contains the files that you will be modifying under `problems/` while the `solutions/` folder contains the reference implementations for each of the problems. The folder `examples/` contains source to the complete examples given in this tutorial.

Finally, the `build.sbt` file contains the build configuration information used to specify what version of Chisel to make your project with.

Running Your First Chisel Build

In this section, we explain how to run your first build to explore what Chisel has to offer. We will go through a simple example for a GCD module and familiarize ourselves with the source files, simulation, and Verilog generation. More comprehensive details will follow in subsequent sections of the tutorial.

The Chisel Source Code

Now that you are more familiar with what your Chisel directory structure contains, let's start by exploring one of the Chisel files. Change directory into the `src/main/scala/examples/` directory and open up the `GCD.scala` file with your favorite text editor.

You will notice that file is already filled out for you to perform the well known GCD algorithm and should look like:

```
// See LICENSE.txt for license details.
package examples

import chisel3._
```

```
class GCD extends Module {
  val io = IO(new Bundle {
    val a  = Input(UInt(16.W))
    val b  = Input(UInt(16.W))
    val e  = Input(Bool())
    val z  = Output(UInt(16.W))
    val v  = Output(Bool())
  })
  val x  = Reg(UInt())
  val y  = Reg(UInt())
  //in chisel2.2 tutotial use: when   (x > y) { x := x - y }  unless   (x
> y) { y := y - x }
  when   (x > y) { x := x - y }
    .elsewhen (x <= y) { y := y - x }
  when (io.e) { x := io.a; y := io.b }
  io.z := x
  io.v := y === 0.U
}
```

The first thing you will notice is the `import chisel3._` declaration; this imports the Chisel library files that allow us to leverage Scala as a hardware construction language. After the import declarations you will see the Scala class definition for the Chisel component you are implementing. You can think of this as almost the same thing as a module declaration in Verilog.

Next we see the I/O specification for this component in the `val io = IO(new Bundle{...})` definition. You will notice that the bundle takes several arguments as part of its construction, each with a specified type (UInt, Bool, etc.), a bit width, and wrapped into a direction (either Input or Output). If a bit width is not specified, Chisel will infer the appropriate bit width for you (in this case default to 1). The `io` Bundle is essentially a constructor for the component interface.

The next section of code performs the actual GCD computation for the module. The register declarations for `x` and `y` tell Chisel to treat `x` and `y` as registers of type UInt().

```
val x = Reg(UInt()) // declares x as UInt register
val y = Reg(UInt()) // declares y as UInt register
```

The `when` statement tells Chisel to perform the operation when the condition is true. In hardware this is basically a multiplexer selecting the values in the `when` block when the condition is true, otherwise selecting the default assignment or keep a register value. With an register on the left hand side of the expression, the assignment is executed on a positive clock edge if the condition is true. This is similar to how Verilog uses `always @ (posedge clk)` to specify synchronous logic.

Finally we see the output assignments for the computation for `io.z` and `io.v`. One particular thing to notice is that, we do not have to specify the width of `x` and `y` in this example. This is because Chisel does the bit width inference for you and sets these values to their appropriate widths based on the computations they are executing.

Running the Chisel Simulation

Now that we are familiar with the Chisel code for the `GCD.scala` file, let's try to simulate it by using the tester and the firrtl interpreter. First, have a look at the Chisel code in `src/test/scala/examples/GCDTests.scala`. This contains a Chisel implementation of a `PeekPokeTester` and a tester driver which will arrange to simulate the circuit, connect it to the tester, and run the tester on the simulated circuit. The tester uses `poke` to drive the circuit's inputs, `step` to run the circuit for a single cycle, and `expect` to verify its output.

Change back to the root directory of the tutorials (the directory containing the file `build.sbt` and the `run-examples.sh` shell script), and invoke the `run-examples.sh` shell script with the name of the example circuit to be simulated:

```
./run-examples.sh GCD
```

This will generate the firrtl description of the circuit for the component defined in `src/main/scala/examples/GCD.scala` and use the firrtl interpreter to simulate it using the test harness defined in `src/test/scala/examples/GCDTests.scala`. If the simulation succeeds, you should see some debug output followed by:

```
RAN 5 CYCLES PASSED
Tutorials passing: 1
[success] Total time: 2 s, completed Sep 16, 2016 10:15:32 AM
```

In addition to the debug output, the build also creates a `test_run_dir/examples/GCD/GCD.fir` file, containing the firrtl definition of the GCD circuit and its tester. We will talk about this more later.

Generating Verilog

One of the most powerful features of Chisel is its ability to generate FPGA and ASIC Verilog from the Scala sources that you use to construct a circuit. To do this, we use an environment variable to specify a different tester backend:

```
./run-examples.sh GCD --backend-name verilator
```

This will instruct the tester driver to generate Verilog output and use Verilator to generate a C++ simulation from the generated Verilog for the specified Chisel component. When the Verilog generation finishes, you should see a [success] message similar to the one you saw after running the firrtl-interpreter backend. If you look in the `test_run_dir/examples/GCD/` directory, you will see quite a few additional files, notably:

- GCD.v - Verilog representation of the Chisel circuit,
- GCD.vcd - waveform dump of the circuit signals during the simulation.

The Verilog source is roughly divided into three parts:

- Module declaration with input and outputs
- Temporary wire and register declaration used for holding intermediate values

- Register assignments in `always @ (posedge clk)`

You can use `gtkwave` to view the vcd dump.

Combinational Logic

The Scala Node: Declaring Wires

Constructing combinational logic blocks in Chisel is fairly straightforward; when you declare a `val` in Scala, it creates a node that represents the data that it is assigned to. As long as the value is not assigned to be a register type (explained later), this tells the Chisel compiler to treat the value as wire. Thus any number of these values can be connected and manipulated to produce the value that we want.

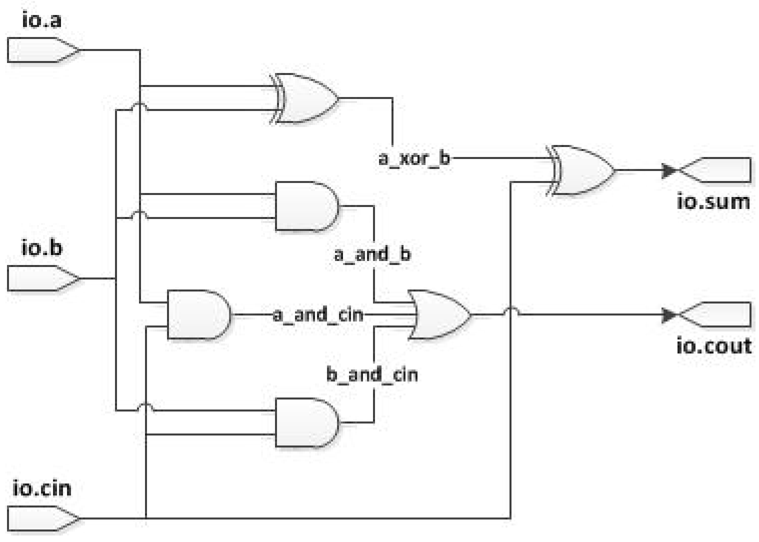
Suppose we want to construct a single full adder. A full adder takes two inputs `a` and `b`, and a carry in `cin` and produces a `sum` and carry out `cout`. The Chisel source code for our full adder will look something like:

```
class FullAdder extends Module {
  val io = IO(new Bundle {
    val a    = Input(UInt(1.W))
    val b    = Input(UInt(1.W))
    val cin  = Input(UInt(1.W))
    val sum  = Output(UInt(1.W))
    val cout = Output(UInt(1.W))
  })
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout := a_and_b | b_and_cin | a_and_cin
}
```

where `cout` is defined as a combinational function of inputs `a`, `b`, and `cin`.

You will notice that in order to access the input values from the `io` bundle, you need to first reference `io` since the input and output values belong to the `io` bundle. The `|`, `&`, and `^` operators correspond to bitwise OR, AND, and XOR operations respectively.

The corresponding wires for each of these values is shown below in Figure *Full Adder Circuit*. You will notice that each `val` corresponds to exactly one of the wires.



Bit Width Inference

If you don't explicitly specify the width of a value in Chisel, the Chisel compiler will infer the bit width for you based on the inputs that define the value. Notice in the `FullAdder` definition, the widths for `a_xor_b, a_and_b, b_and_cin,` and `a_and_cin` are never specified anywhere. However, based on how the input is computed, Chisel will correctly infer each of these values are one bit wide since each of their inputs are the results of bitwise operations applied to one bit operands.

A quick inspection of the generated Verilog shows these values are indeed one bit wide:

```
module FullAdder(
  input   clk,
  input   reset,
  input   io_a,
  input   io_b,
  input   io_cin,
  output  io_sum,
  output  io_cout
);
  wire  a_xor_b;
  wire  T_5;
  wire  a_and_b;
  wire  b_and_cin;
  wire  a_and_cin;
  wire  T_6;
  wire  T_7;
  assign io_sum = T_5;
  assign io_cout = T_7;
  assign a_xor_b = io_a ^ io_b;
  assign T_5 = a_xor_b ^ io_cin;
  assign a_and_b = io_a & io_b;
  assign b_and_cin = io_b & io_cin;
  assign a_and_cin = io_a & io_cin;
  assign T_6 = a_and_b | b_and_cin;
  assign T_7 = T_6 | a_and_cin;
endmodule
```

Suppose we change the widths of the `FullAdder` to be 2 bits wide each instead such that the Chisel source now looks like:

```
class FullAdder extends Module {
  val io = IO(new Bundle {
    val a    = Input(UInt(2.W))
    val b    = Input(UInt(2.W))
    val cin  = Input(UInt(2.W))
    val sum  = Output(UInt(2.W))
    val cout = Output(UInt(2.W))
  })
  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout := a_and_b | b_and_cin | a_and_cin
}
```

As a result, the Chisel compiler should infer each of the intermediate values `a_xor_b, a_and_b, b_and_cin,` and `a_and_cin` are two bits wide. An inspection of the Verilog code correctly shows that Chisel inferred each of the intermediate wires in the calculation to be 2 bits wide.

```
module FullAdder(
  input   clk,
  input   reset,
  input  [1:0] io_a,
  input  [1:0] io_b,
  input  [1:0] io_cin,
  output [1:0] io_sum,
  output [1:0] io_cout
);
  wire [1:0] a_xor_b;
  wire [1:0] T_5;
  wire [1:0] a_and_b;
  wire [1:0] b_and_cin;
  wire [1:0] a_and_cin;
  wire [1:0] T_6;
  wire [1:0] T_7;
  assign io_sum = T_5;
  assign io_cout = T_7;
  assign a_xor_b = io_a ^ io_b;
  assign T_5 = a_xor_b ^ io_cin;
  assign a_and_b = io_a & io_b;
  assign b_and_cin = io_b & io_cin;
  assign a_and_cin = io_a & io_cin;
  assign T_6 = a_and_b | b_and_cin;
  assign T_7 = T_6 | a_and_cin;
endmodule
```

Using Registers

Unlike Verilog, specifying a register in Chisel tells the compiler to actually generate a positive edge triggered register. In this section we explore how to instantiate registers in Chisel by constructing a shift register.

In Chisel, when you instantiate a register there are several ways to specify the connection of the input to a register. As shown in the GCD example, you can "declare" the register and assign what it's input is connected to in a `when...` block or you can simply pass the value that the register is clocking as a parameter to the register.

If you choose to pass a next value to the register on construction using the `next` named parameter, or the specialized register constructor `RegNext`, it will clock the new value every cycle unconditionally:

```
// Clock the new register value on every cycle
val y = io.x
val z = RegNext(y)
```

If we only want to update if certain conditions are met we use a `when` block to indicate that the registers are only updated when the condition is satisfied:

```
// Clock the new register value when the condition a > b
val x = Reg(UInt())
when (a > b) { x := y }
.elsewhen ( b > a) {x := z}
.otherwise { x := w}
```

It is important to note that when using the conditional method, the values getting assigned to the input of the register match the type and bitwidth of the register you declared. In the unconditional register assignment, you do not need to do this as Chisel will infer the type and width from the type and width of the input value.

The following sections show how these can be used to construct a shift register.

Unconditional Register Update

Suppose we want to construct a basic 4 bit shift register that takes a serial input `in` and generates a serial output `out`. For this first example we won't worry about a parallel load signal and will assume the shift register is always enabled. We also will forget about the register reset signal.

If we instantiate and connect each of these 4 registers explicitly, our Chisel code will look something like:

```
class ShiftRegister extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(1.W))
    val out = Output(UInt(1.W))
  })
  val r0 = RegNext(io.in)
  val r1 = RegNext(r0)
  val r2 = RegNext(r1)
  val r3 = RegNext(r2)
  io.out := r3
}
```

If we take a look at the generated Verilog, we will see that Chisel did indeed map our design to a shift register. One thing to notice is that the clock signal and reset signals are implicitly attached to our design.

```
module ShiftRegister(input clk, input reset,
    input  io_in,
    output io_out);

  reg[0:0] r3;
  reg[0:0] r2;
  reg[0:0] r1;
  reg[0:0] r0;

  assign io_out = r3;
  always @(posedge clk) begin
    r3 <= r2;
    r2 <= r1;
    r1 <= r0;
    r0 <= io_in;
  end
endmodule
```

Conditional Register Update

As mentioned earlier, Chisel allows you to conditionally update a register (use an enable signal) using the `when`, `.elsewhen`, `.otherwise` block. Suppose we add an enable signal to our shift register, that allows us to control whether data is shift in and out on a given cycle depending on an `enable` input signal. The new shift register now looks like:

```
class ShiftRegister extends Module {
  val io = IO(new Bundle {
    val in     = Input(UInt(1.W))
    val enable = Input(Bool())
    val out    = Output(UInt(1.W))
  })

  val r0 = Reg(UInt())
  val r1 = Reg(UInt())
  val r2 = Reg(UInt())
  val r3 = Reg(UInt())

  when (io.enable) {
    r0 := io.in
    r1 := r0
    r2 := r1
    r3 := r2
  }
  io.out := r3
}
```

Notice that it is not necessary to specify an `.otherwise` condition as Chisel will correctly infer that the old register value should be preserved otherwise.

Register Reset

Chisel allows you to specify a synchronous reset to a certain value by specifying an additional parameter when you first declare them. In our shift register, let's add a reset capability that resets all the register values to zero synchronously. To do this we need to provide our register declarations a little more information using the `init` parameter, or using the specialized constructor `RegInit`, with what value we want on a synchronous reset:

```
class ShiftRegister extends Module {
  val io = IO(new Bundle {
    val in     = Input(UInt(1.W))
    val enable = Input(Bool())
    val out    = Output(UInt(1.W))
  })
  // Register reset to zero
  val r0 = RegInit(0.U(1.W))
  val r1 = RegInit(0.U(1.W))
  val r2 = RegInit(0.U(1.W))
  val r3 = RegInit(0.U(1.W))
  when (io.enable) {
    r0 := io.in
    r1 := r0
    r2 := r1
    r3 := r2
  }
  io.out := r3
}
```

Notice that reset value can actually be any value, simply replace the zeros and width to appropriate values.

Chisel also has an implict global `reset` signal that you can use in a `when` block. The reset signal is conveniently called `reset` and does not have to be declared, but in order to treat it as a Bool, you need to add the `toBool` cast. The shift register using this implict global reset now looks like:

```
class ShiftRegister extends Module {
  val io = IO(new Bundle {
    val in     = Input(UInt(1.W))
    val enable = Input(Bool())
    val out    = Output(UInt(1.W))
  })
  val r0 = Reg(UInt())
  val r1 = Reg(UInt())
  val r2 = Reg(UInt())
  val r3 = Reg(UInt())
  when(reset.toBool) {  // interpret reset as a Bool
    r0 := 0.U
    r1 := 0.U
    r2 := 0.U
    r3 := 0.U
  } .elsewhen(io.enable) {
    r0 := io.in
    r1 := r0
    r2 := r1
    r3 := r2
  }
  io.out := r3
}
```

This will generate slightly different looking Verilog source code but will still function the same as the previous implementation of the shift register with reset.

Sequential Circuit

The following exercises can be found in your `$TUT_DIR/src/main/scala/problems/` folder. You will find that some parts of the tutorial files have been completed for you and the section that you need to complete is indicated in the file. The solutions to each of these exercises can be found in the `$TUT_DIR/src/main/scala/solutions/` folder.

The first tutorial problem is to write a sequential circuit that sums `in` values. You can find the template in `$TUT_DIR/src/main/scala/problems/Accumulator.scala` including a stubbed out version of the circuit:

```
class Accumulator extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(1.W))
    val out = Output(UInt(8.W))
  })

  // flush this out ...

  io.out := 0.U
}
```

and a complete tester that confirms that you have successfully designed the circuit in `$TUT_DIR/src/test/scala/problems/Accumulator.scala`.

Run:

```
./run-problem.sh Accumulator
```

This should fail until you complete the circuit implementing the accumulator. Edit the `src/main/scala/problems/Accumulator.scala` source file and execute the `run-problem.sh` shell script until your circuit passes the tests.

Prev (Chisel Installation) Next (Basic Types and Operations)

Chisel Assignments and Re-assignments

When you first define a value in Chisel, we use the = operator in order to tell Chisel to allocate the value for the first time. On every subsequent reassignment to the value, we must use a := when reassigning the value.

Since we are constructing a digital circuit, the notion of reassignment does not make much sense since connections between circuit nodes only need to be specified once. However, there are some cases when we will need to perform reassignment to a value in Chisel since it is compiled sequentially unlike Verilog. Thus it may be necessary to perform reassignment when a value or connection is not known until later in the Chisel source.

A simple example of when reassignment is necessary is in the construction of the top level I/O for your module; the values of the output are not immediately known at the time of declaration.

Consider the simple *FullAdder* circuit from previous tutorial that determines the sum *sum* and carry out *cout* given two values *a* and *b*, and a carry in *cin*.

```
class FullAdder extends Module {
  val io = IO(new Bundle {
    val a    = Input(UInt(1.W))
    val b    = Input(UInt(1.W))
    val cin  = Input(UInt(1.W))
    val sum  = Output(UInt(1.W))
    val cout = Output(UInt(1.W))
  })

  // Generate the sum
  val a_xor_b = io.a ^ io.b
  io.sum := a_xor_b ^ io.cin
  // Generate the carry
  val a_and_b = io.a & io.b
  val b_and_cin = io.b & io.cin
  val a_and_cin = io.a & io.cin
  io.cout := a_and_b | b_and_cin | a_and_cin
}
```

In this example we make sure to use the *:=* reassignment for the *io.sum* and *io.cout* output values because we only know what their values are later in the code and not at the time of construction of the *io* Bundle. All other values in this example use the = assignment operator since they need to be created.

In general, the rule of thumb is to use the reassignment operator *:=* if the value already has been assigned by the = operator, otherwise the = operator should be used. Note that if you do not use the = or *:=* operators correctly you will get an error when you try to compile your design.

The Chisel UInt Class

In the previous examples we have been using the UInt type which is an unsigned integer as the type for all of our values. For many of the basic computations in Chisel the UInt class is sufficient. The following example shows some of the commonly used UInt operations in the context of a simple *ALU*

> We ignore overflow and underflow in this example.

```
class BasicALU extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val opcode = Input(UInt(4.W))
    val out = Output(UInt(4.W))
  })
  io.out := 0.U //THIS SEEMS LIKE A HACK/BUG
  when (io.opcode === 0.U) {
    io.out := io.a //pass A
  } .elsewhen (io.opcode === 1.U) {
    io.out := io.b //pass B
  } .elsewhen (io.opcode === 2.U) {
    io.out := io.a + 1.U //increment A by 1
  } .elsewhen (io.opcode === 3.U) {
    io.out := io.a - 1.U //increment B by 1
  } .elsewhen (io.opcode === 4.U) {
    io.out := io.a + 4.U //increment A by 4
  } .elsewhen (io.opcode === 5.U) {
    io.out := io.a - 4.U //decrement A by 4
  } .elsewhen (io.opcode === 6.U) {
    io.out := io.a + io.b //add A and B
  } .elsewhen (io.opcode === 7.U) {
    io.out := io.a - io.b //subtract B from A
  } .elsewhen (io.opcode === 8.U) {
    io.out := io.a < io.b //set on A less than B
  } .otherwise {
    io.out :=  (io.a === io.b).asUInt //set on A equal to B
  }
}
```

You will notice that there are multiple reassignments to *io.out* inside a *when* block which indicates that the value of *io.out* can take many different values depending on the *io.opcode* in this example. Also notice that in order to specify constants to add to our operands, we must also specify them as a UInt type as UInt operations on different type operands is not allowed.

```
// Specify that 1 is a UInt type
io.output := io.a + 1.U
```

A list of commonly used UInt operations is given in the table below:

Operand	Operation	Output Type
+	Add	UInt
-	Subtract	UInt
*	Multiply	UInt
/	UInt Divide	UInt
%	Modulo	UInt
~	Bitwise Negation	UInt
^	Bitwise XOR	UInt
&	Bitwise AND	UInt
|	Bitwise OR	UInt
===	Equal	Bool
=/=	Not Equal	Bool
>	Greater	Bool
<	Less	Bool
>=	Greater or Equal	Bool
<=	Less or Equal	Bool

Bit Extraction

The UInt class allows you to extract bits based on their index of their representation. Given an *n* bit wide value *value* we can extract the bits *x* through *y* (n > x > y >= 0) by simply doing the following:

```
// extracts the x through y bits of value
val x_to_y = value(x, y)
```

Note that the higher index is specified first in the argument list when extraction the bits. Also notice that the bits in the UInt are zero indexed so the highest bit that can be extracted from an *n* bit wide value is *n-1*.

If you just want to extract a single bit from the value, say bit *x* we simply need to specify a single index instead as follows:

```
// extract the x-th bit from value
val x_of_value = value(x)
```

A more concrete example of bit extraction in action is shown below. In this example, based on the value of the offset, we would like to select a byte from a word which is a common operation when loading a byte from word addressed memories:

```
class ByteSelector extends Module {
  val io = IO(new Bundle {
    val in     = Input(UInt(32.W))
    val offset = Input(UInt(2.W))
    val out    = Output(UInt(8.W))
  })
  io.out := 0.U(8.W)
  when (io.offset === 0.U(2.W)) {
    io.out := io.in(7,0)
  } .elsewhen (io.offset === 1.U) {
    io.out := io.in(15,8)
  } .elsewhen (io.offset === 2.U) {
    io.out := io.in(23,16)
  } .otherwise {
    io.out := io.in(31,24)
  }
}
```

Bit Concatenation

Chisel also allows you to easily concatenate bits together using *Cat*. Suppose you have a data bus that you would like to drive with two seperate words *A* and *B*. In order to concatenate these two values together we simply say:

```
val A = UInt(32.W)
val B = UInt(32.W)
val bus = Cat(A, B) // concatenate A and B
```

Again, the first argument to *Cat* will be placed in the high part while the second argument gets the low part of *bus*. Thus for this example bits 0 to 31 of *bus* correspond to *B*, while bits 32 to 63 correspond to *A*.

Note - *Cat* is implemented in the utility package, external to Chisel core. In order to make it available to your source code, you will need to instruct the Scala compiler to import its definition:

```
import chisel3.util.Cat
```

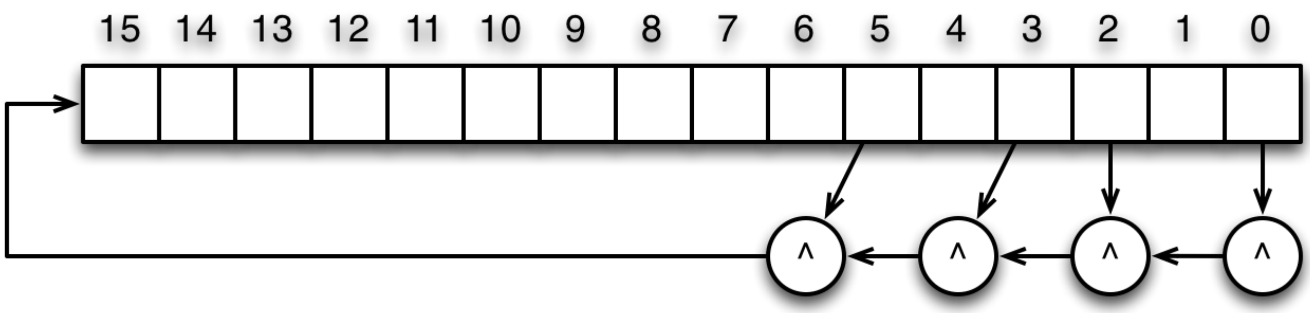
or

```
import chisel3.util._
```

to import all the utility definitions.

LFSR16

In this assignment, write the circuit for a 16-bit Linear Feedback Shift Register (*LFSR16*) as shown below:



by filling in the following module:

```
class LFSR16 extends Module {
  val io = IO(new Bundle {
    val inc = Input(Bool())
    val out = Output(UInt(16.W))
  })
  // ...
  io.out := 0.U
}
```

found in *$TUT_DIR/src/main/scala/problems/LFSR16.scala*. Make sure to define and initialize an internal register to one and update it when *inc* is asserted. Use bit concatentation and bit extraction in conjunction with the xor operator ^. Run:

```
./run-problem.sh LFSR16
```

Fix your program as necessary and re-run until it works.

UInt Operation Bit Inference

Note that for some operations such as addition and multiplication, that number of resulting bits of the computation can be greater than the number of bits for the operands.

Consider the following example where we multiply two 16 bit numbers *A* and *B* together. Note that the product of two 16 bit numbers is at worst 32 bits wide.

```
class HiLoMultiplier() extends Module {
  val io = IO(new Bundle {
    val A  = Input(UInt(16.W))
    val B  = Input(UInt(16.W))
    val Hi = Output(UInt(16.W))
    val Lo = Output(UInt(16.W))
  })
  val mult = io.A * io.B
  io.Lo := mult(15, 0)
  io.Hi := mult(31, 16)
}
```

Notice that we never specify the width of the value *mult* anywhere in the Chisel source. Normally if we performed this in Verilog we would have had to specify the width beforehand. But a look at the generated Verilog for this example shows that Chisel correctly inferred the *mult* value to be 32 bits wide:

```
module HiLoMultiplier(
    input  [15:0] io_A,
    input  [15:0] io_B,
    output[15:0] io_Hi,
    output[15:0] io_Lo);

  wire[15:0] T0;
  wire[31:0] mult; // Chisel infers this to be 32 bits
  wire[15:0] T1;

  assign io_Lo = T0;
  assign T0 = mult[4'hf:1'h0];
  assign mult = io_A * io_B;
  assign io_Hi = T1;
  assign T1 = mult[5'h1f:5'h10];
endmodule
```

As we get to more complicate designs, it will become more clear that bit inference in Chisel is a very powerful feature that makes constructing hardware more efficient. A list of common bit inferences is shown below for commonly used operations:

Operation	**Result Bit Width**
Z = X + Y	max(Width(X), Width(Y))
Z = X - Y	max(Width(X), Width(Y))
Z = X + Y	max(Width(X), Width(Y))

Operation	Result Bit Width
Z = X | Y	max(Width(X), Width(Y))
Z = X ^ Y	max(Width(X), Width(Y))
Z = ~X	Width(X)
Z = Mux(C, X, Y)	max(Width(X), Width (Y))
*Z = X * Y*	Width(X) + Width(Y)
Z = X << n	Width(X) + n
Z = X >> n	Width(X) - n
Z = Cat(X, Y)	Width(X) + Width(Y)
Z = Fill(n, x)	Width(X) + n

The Chisel Bool Class

The Bool class in Chisel is used to represent the result of logical expressions and takes either the values *true* or *false*. These can be used in conditional statements such as *when* blocks.

```
val change = io.a === io.b // change gets Bool type
when (change) {            // exec if change is true
  ...
} .otherwise {
  ...
}
```

You can instantiate a Bool value like this:

```
val true_value  = true.B
val false_value = false.B
```

As shown in the *BasicALU* example, in order to use a Bool value as a UInt type and assign it to an output, a cast to UInt is required.

Casting Between Types

When assigning values, it is required that you assign a value of the same type. For instance, if you try to assign a Bool type to an output value that is expecting a UInt type, you will get an error.

```
...
val io  = IO(new Bundle {
  val in  = Input(UInt(2.W))
  val out = Output(UInt(1.W))
}
// attempted Bool assignment to UInt
io.out := (in === 0.U)
...
```

The correct way to perform the intended operation is to cast the resulting Bool type to a UInt using the *asUInt* cast. The correct Chisel code will look like:

```
...
val io = IO(new Bundle {
  val in  = Input(UInt(2.W))
  val out = Output(UInt(1.W))
})
io.out := (in === 0.U).asUInt // UInt cast
...
```

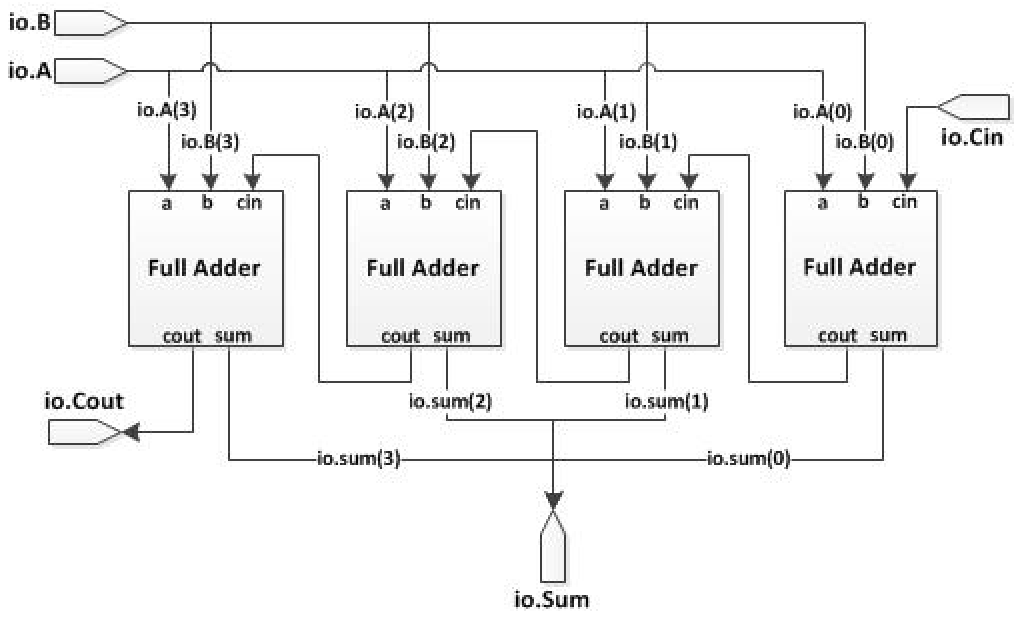
Some of the common casts that you may use are:

- asUInt()
- asSInt()
- asBool()

Module Instantiation

Like other hardware description languages, Chisel allows fairly straightforward module instantiation to enable modularity and hierarchy. In Chisel, instantiating a Module class is the equivalent to instantiating a module in Verilog. To do this, we simply use a call to `Module` with the module created with the Scala `new` keyword in order to indicate that we are instantiation a new module. We want to make sure we assign this to a value so that we can reference its input and outputs, which we also need to connect.

For example, suppose we would like to construct a 4-bit adder using multiple copies of the `FullAdder` module, as shown in the Figure 1.



The Chisel source code is shown below.

```
// A 4-bit adder with carry in and carry out
class Adder4 extends Module {
  val io = IO(new Bundle {
    val A    = Input(UInt(4.W))
    val B    = Input(UInt(4.W))
    val Cin  = Input(UInt(1.W))
    val Sum  = Output(UInt(4.W))
    val Cout = Output(UInt(1.W))
  })
  // Adder for bit 0
  val Adder0 = Module(new FullAdder())
  Adder0.io.a   := io.A(0)
  Adder0.io.b   := io.B(0)
  Adder0.io.cin := io.Cin
  val s0 = Adder0.io.sum
  // Adder for bit 1
  val Adder1 = Module(new FullAdder())
  Adder1.io.a   := io.A(1)
  Adder1.io.b   := io.B(1)
  Adder1.io.cin := Adder0.io.cout
  val s1 = Cat(Adder1.io.sum, s0)
  // Adder for bit 2
  val Adder2 = Module(new FullAdder())
  Adder2.io.a   := io.A(2)
  Adder2.io.b   := io.B(2)
  Adder2.io.cin := Adder1.io.cout
  val s2 = Cat(Adder2.io.sum, s1)
  // Adder for bit 3
  val Adder3 = Module(new FullAdder())
  Adder3.io.a   := io.A(3)
  Adder3.io.b   := io.B(3)
  Adder3.io.cin := Adder2.io.cout
  io.Sum  := Cat(Adder3.io.sum, s2).asUInt
  io.Cout := Adder3.io.cout
}
```

In this example, notice how when referencing each module I/O we must first reference `io` that contains the ports for the I/Os. Again, note how all assignments to the module I/Os use a reassignment operator `:=`. When instantiating modules, it is important to make sure that you connect all the input and output ports. If a port is not connected, the Chisel compiler may optimize away portions of your design that it finds unnecessary due to the unconnected ports and throw errors or warnings.

The Vec Class

The `Vec` class allows you to create an indexable vector in Chisel which can be filled with any expression that returns a chisel data type. The general syntax for a `Vec` declaration is given by:

```
val myVec = Vec(Seq.fill( <number of elements> ) { <data type> })
```

Where `<number of elements>` corresponds to how long the vector is and `<data type>` corresponds to what type of class the vector contains.

For instance, if we wanted to instantiate a 10 entry vector of 5 bit UInt values, we would use:

```
val ufix5_vec10 := Vec(Seq.fill(10) { UInt(5.W) })
```

If we want to define a register of vector...

```
val reg_vec32 = Reg(Vec(Seq.fill(32){ UInt() }))
```

In order to assign to a particular value of the `Vec`, we simply assign the target value to the vector at a specified index. For instance, if we wanted to assign a UInt value of zero to the first register in the above example, the assignment would look like:

```
reg_vec32(0) := 0.U
```

To access a particular element in the vector at some index, we specify the index of the vector. For example, to extract the 5th element of the register vector in the above example and assign it to some value `reg5`, the assignment would look like:

```
val reg5 = reg_vec(5)
```

The syntax for the `Vec` class is slightly different when instantiating a vector of modules.When instantiating a vector of modules the data type that is specified in the {} braces is slightly different than the usualy primitive types. To specify a vector of modules, we use the `io` bundle when specifying the type of the vector. For example, in order to specify a `Vec` with 16 modules , say `FullAdder`s in this case, we would use the following declaration:

```
val FullAdders =
  Vec(Seq.fill(16){ Module(new FullAdder()).io })
```

Notice we use the keyword `new` in the vector definition before the module name `FullAdder`. For how to actually access the `io` on the vector modules, refer to the next section.

Vec Shift Reg

The next assignment is to construct a simple bit shift register. The following is the template from `$TUT_DIR/src/main/scala/problems/VecShiftRegisterSimple.scala`:

```
class VecShiftRegisterSimple extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  val initValues = Seq.fill(4) { 0.U(8.W) }
  val delays = RegInit(VecInit(initValues))
  ...
  io.out := 0.U
}
```

where `out` is a four cycle delayed copy of values on `in`.

Parametrization

In the previous Adder example, we explicitly instantiated four different copies of a `FullAdder` and wired up the ports. But suppose we want to generalize this structure to an n-bit adder. Like Verilog, Chisel allows you to pass parameters to specify certain aspects of your design. In order to do this, we add a parameter in the Module declaration to our Chisel definition. For a carry ripple adder, we would like to parametrize the width to some integer value `n` as shown in the following example:

```
// A n-bit adder with carry in and carry out
class Adder(n: Int) extends Module {
  val io = IO(new Bundle {
    val A    = Input(UInt(n.W))
    val B    = Input(UInt(n.W))
    val Cin  = Input(UInt(1.W))
    val Sum  = Output(UInt(n.W))
    val Cout = Output(UInt(1.W))
  })
  // create a vector of FullAdders
  val FAs = Vec(Seq.fill(n){ Module(new FullAdder()).io })

  // define carry and sum wires
  val carry = Wire(Vec(n+1, UInt(1.W)))
  val sum   = Wire(Vec(n, Bool()))

  // first carry is the top level carry in
  carry(0) := io.Cin

  // wire up the ports of the full adders
  for(i <- 0 until n) {
    FAs(i).a   := io.A(i)
    FAs(i).b   := io.B(i)
    FAs(i).cin := carry(i)
    carry(i+1) := FAs(i).cout
```

```
      sum(i)     := FAs(i).sum.toBool()
    }
    io.Sum  := sum.asUInt
    io.Cout := carry(n)
  }
```

Note that in this example, we keep track of the sum output in a `Vec` of `Bool`s. This is because Chisel does not support bit assignment directly. Thus in order to get the n-bit wide `sum` in the above example, we use an n-bit wide `Vec` of `Bool`s and then cast it to a UInt().

You will notice that modules are instantiated in a Vec class which allows us to iterate through each module when assigning the ports connections to each `FullAdder`. This is similar to the generate statement in Verilog. However, you will see in more advanced tutorials that Chisel can offer more powerful variations.

Instantiating a parametrized module is very similar to instantiating an unparametrized module except that we must provide arguments for the parameter values. For instance, if we wanted to instantiate a 4-bit version of the `Adder` module we defined above, it would look like:

```
val adder4 = Module(new Adder(4))
```

We can also instantiate the `Adder` by explicitly specifying the value of its parameter `n` like the this:

```
val adder4 = Module(new Adder(n = 4))
```

Explicitly specifying the parameter is useful when you have a module with multiple parameters. Suppose you have a parametrized FIFO module with the following module definition:

```
class FIFO(width: Int, depth: Int) extends Module {...}
```

You can explicitly specify the parameter values in any order:

```
val fifo1 = Module(new FIFO(16, 32))
val fifo2 = Module(new FIFO(width = 16, depth = 32))
val fifo3 = Module(new FIFO(depth = 32, width = 16))
```

All of the above definitions pass the same parameters to the FIFO module. Notice that when you explicitly assign the parameter values, they can occur in any order you want such as the definition for fifo3.

Built In Primitives

Like other HDL, Chisel provides some very basic primitives. These are constructs that are built in to the Chisel compiler and come for free. The Reg, UInt, and Bundle classes are such primitives that have already been covered. Unlike Module instantiations, primitive do not require explicit connections of their io ports to use. Other useful primitive types include the Mem and Vec classes which will be discussed in a more advanced tutorial. In this tutorial we explore the use of the `Mux` primitive.

The Mux Class

The `Mux` primitive is a two input multiplexer. In order to use the `Mux` we first need to define the expected syntax of the `Mux` class. As with any two input multiplexer, it takes three inputs and one output. Two of the inputs correspond to the data values `A` and `B` that we would like to select which can be any width and data type as long as they are the same. The first input `select`, which is a Bool type, determines which one to output. A `select` value of `true` will output the value `A`, while a `select` value of `false` will pass `B`.

```
val out = Mux(select, A, B)
```

Thus if `A=10`, `B=14`, and `select` was `true`, the value of `out` would be assigned 10. Notice how using the `Mux` primitive type abstracts away the logic structures required if we had wanted to implement the multiplexer explicitly.

Parameterized Width Adder

The next assignment is to construct an adder with a parameterized width and using the built in addition operator `+`. The following is a the template from `$TUT_DIR/src/main/scala/problems/Adder.scala`:

```
class Adder(val w: Int) extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(1.W))
    val in1 = Input(UInt(1.W))
    val out = Output(UInt(1.W))
  })
  ...
  io.out := 0.U
}
```

where `out` is sum of `w` width unsigned inputs `in0` and `in1`.

Notice how `val` is added to the width parameter value to allow the width to be accessible from the tester as a field of the adder module object.

Edit your copy of `$TUT_DIR/src/main/scala/problems/Adder.scala` and run:

```
./run-problem.sh Adder
```

until your circuit passes the tests.

Chisel's Scala based testbench is the first line of defense against simple bugs in your design. The Scala testbench uses several unique Chisel constructs to perform this. To see how this works, let's first explore a simple example.

Scala Testbench Example

Below is the *ByteSelector.scala* module definition from the previous tutorial and the corresponding Chisel test harness.

```
package examples

import chisel3._

class ByteSelector extends Module {
  val io = IO(new Bundle {
    val in     = Input(UInt(32.W))
    val offset = Input(UInt(2.W))
    val out    = Output(UInt(8.W))
  })
  io.out := 0.U(8.W)
  when (io.offset === 0.U) {
    io.out := io.in(7,0)
  } .elsewhen (io.offset === 1.U) {
    io.out := io.in(15,8)
  } .elsewhen (io.offset === 2.U) {
    io.out := io.in(23,16)
  } .otherwise {
    io.out := io.in(31,24)
  }
}

class ByteSelectorTests(c: ByteSelector)
    extends Tester(c) {
  val test_in = 12345678
  for (t <- 0 until 4) {
    poke(c.io.in,     test_in)
    poke(c.io.offset, t)
    step(1)
    expect(c.io.out, (test_in >> (t * 8)) & 0xFF)
  }
}
```

In the test harness *ByteSelectorTests* we see that the test portion is written in Scala with some Chisel constructs inside a *Tester* class definition. The device under test is passed to us as a parameter *c*.

In the *for* loop, the assignments for each input of the *ByteSelector* is set to the appropriate values using *poke*. For this particular example, we are testing the *ByteSelector* by hardcoding the input to some known value and checking if each of the 4 offsets returns the appropriate byte. To do this, on each iteration we generate appropriate inputs to the module and tell the simulation to assign this value to the input of the device we are testing *c*:

```
val test_in = 12345678
for (t <- 0 until 4) {
  // set in of the DUT to be some known word
  poke(c.io.in,     test_in)
  // set the offset of the DUT
  poke(c.io.offset, t)
  ...
}
```

Next we step the circuit. We next advance the simulation by calling the *step* function. This effectively advances the simulation one clock cycle in the presence of sequential logic.

```
step(1)
```

Finally, we check for expected outputs. In this case, we check the expected output of *ByteSelector* as follows:

```
expect(c.io.out, (test_in >> (t * 8)) & 0xFF)
```

This defines the reference output expected for this particular cycle of the simulation. Since the circuit we are testing is purely combinational, we expected that the output we define appears on any advancement of the simulation. The *expect* function will record either true or false after checking if the output generates the expected reference output. The results of successive *expect*'s are anded into a *Tester* field called *ok* which starts out as *true*. The value of the *ok* field determines the success or failure of the tester execution.

Actually *expect* is defined in terms of *peek* roughly as follows:

```
def expect (data: Bits, expected: BigInt) =
  ok = peek(data) == expected && ok
```

where *peek* gets the value of a signal from the DUT.

Simulation Debug Output

Now suppose we run the testbench for the *ByteSelector* defined previously. To do this, run `./run-examples.sh ByteSelector --is-verbose` from the tutorials directory. We've added the *is-verbose* flag to get the actual sequence of *peeks* and *pokes* used during the test.

When we run the testbench, we will notice that the simulation produces debug output every time the *step* function is called. Each of these calls gives the state of the inputs and outputs to the *ByteSelector* and whether the check between the reference output and expected output matched as shown below:

```
Starting tutorial ByteSelector
[info] [0.006] Elaborating design...
[info] [0.201] Done elaborating.
Total FIRRTL Compile Time: 363.1 ms
Total FIRRTL Compile Time: 56.7 ms
End of dependency graph
Circuit state created
[info] [0.001] SEED 1505836830809
[info] [0.003]   POKE io_in <- 12345678
[info] [0.004]   POKE io_offset <- 0
[info] [0.004] STEP 0 -> 1
[info] [0.006] EXPECT AT 1   io_out got 78 expected 78 PASS
[info] [0.007]   POKE io_in <- 12345678
[info] [0.007]   POKE io_offset <- 1
[info] [0.007] STEP 1 -> 2
[info] [0.009] EXPECT AT 2   io_out got 97 expected 97 PASS
[info] [0.009]   POKE io_in <- 12345678
[info] [0.010]   POKE io_offset <- 2
[info] [0.010] STEP 2 -> 3
[info] [0.012] EXPECT AT 3   io_out got 188 expected 188 PASS
[info] [0.012]   POKE io_in <- 12345678
[info] [0.012]   POKE io_offset <- 3
[info] [0.012] STEP 3 -> 4
[info] [0.014] EXPECT AT 4   io_out got 0 expected 0 PASS
test ByteSelector Success: 4 tests passed in 9 cycles taking 0.028873
seconds
[info] [0.015] RAN 4 CYCLES PASSED
Tutorials passing: 1
```

Also notice that there is a final pass assertion "PASSED" at the end which corresponds to the *allGood* at the very end of the testbench. In this case, we know that the test passed since the allGood assertion resulted in a "PASSED". In the event of a failure, the assertion would result in a "FAILED" output message here.

General Testbench

In general, the scala testbench should have the following rough structure:

- Set inputs using *poke*
- Advance simulation using *step*
- Check expected values using *expect* (and/or *peek*)
- Repeat until all appropriate test cases verified

For sequential modules we may want to delay the output definition to the appropriate time as the *step* function implicitly advances the clock one period in the simulation. Unlike Verilog, you do not need to explicitly specify the timing advances of the simulation; Chisel will take care of these details for you.

Max2 Testbench

In this assignment, write a tester for the *Max2* circuit (found in *$TUT_DIR/src/main/scala/problems/Max2.scala*):

```
class Max2 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(8.W))
    val in1 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  io.out := Mux(io.in0 > io.in0, io.in0, io.in1)
}
```

by filling in the following tester (found in *$TUT_DIR/src/test/scala/problems/Max2Tests.scala*):

```
class Max2Tests(c: Max2) extends PeekPokeTester(c) {
  for (i <- 0 until 10) {

    // Implement below ----------

    poke(c.io.in0, 0)
    poke(c.io.in1, 0)
    step(1)
    expect(c.io.out, 1)

    // Implement above ----------
  }
}
```

using random integers generated as follows:

```
// returns random int in 0..lim-1
val in0 = rnd.nextInt(lim)
```

Run

```
./run-problem.sh Max2
```

until the circuit passes your tests.

Limitations of the Testbench

The Chisel testbench works well for simple tests and small numbers of simulation iterations. However, for larger test cases, the Chisel testbench quickly becomes more complicated and slower simply due to the inefficiency of the infrastructure. For these larger and more complex test cases, we recommend using the C++ emulator or Verilog test harnesses which run faster and can handle more rigorous test cases.

Creating Your Own Projects

In order to create your own projects from scratch, you will need to create a directory, a Chisel source file, an optional tester, and a build.sbt configuration file. You should clone the chisel-template repo, which contains a minimal project suitable as a template. Follow the instructions on that repo's README.

Conditional Register Updates

As shown earlier in the tutorial, conditional register updates are performed with the `when` block which takes a Bool value or some boolean expression to evaluate. In this section we more fully explore how to use this `when` conditional update structure.

If a `when` block is used by itself, Chisel will assume that if the condition for the `when` block doesn't evaluate to true, there is no update to the register value. However, most of the time we don't want to limit ourselves to a single conditional. Thus in Chisel we use `.elsewhen` and `.otherwise` statements to select between multiple possible register updates as shown in the following sections.

The `.elsewhen` Clause

When specifying a conditional update, we may want to check several conditions which we want to check in some order. To do this for register updates, we use a `when` ... `.elsewhen` structure. This is analagous to an if... else if control structure in sequential programming. As with else if clauses, as many `.elsewhen` statements can be chained together in a single `when` block.

The general structure thus looks like:

```
when (<condition 1>) {<register update 1>}
.elsewhen (<condition 2>) {<register update 2>}
...
.elsewhen (<condition N>) {<register update N>}
```

Where <condition 1> through represent the trigger conditions of their respective segments.

An example of this statement in action is shown in the following implementation of a simple stack pointer. Suppose, we need to maintain a pointer that keeps track of the address of the top of a stack. Given a signal pop that decrements the stack pointer address by 1 entry and a signal push that increments the stack pointer address by 1 entry, the implementation of just the pointer would look like the following:

```
class StackPointer(size:Int) extends Module {
  val io = IO(new Bundle {
    val push = Input(Bool())
    val en   = Input(Bool())
    val pop  = Input(Bool())
  })

  val sp = RegInit(0.U(log2Ceil(size).W))

  when (io.en && io.push && (sp != (size-1).U)) {
    sp := sp + 1.U
  } .elsewhen(io.en && io.pop && (sp > 0.U)) {
    sp := sp - 1.U
  }
}
```

Notice that in this implementation, the push signal has higher priority over the pop signal as it appears earlier in the `when` block.

The `.otherwise` Clause

In order to specify a default register update value if all the conditions in the `when` block fail to trigger, we use an `.otherwise` clause. The `.otherwise` clause is analagous to the else case that completes an if ... else block. The `.otherwise` statement must occur last in the `when` block.

The general structure for the complete `when` block now looks like:

```
when (<condition 1>) {<register update 1>}
.elsewhen (<condition 2>) {<register update 2>}
...
.elsewhen (<condition N>) {<register update N>}
.otherwise {<default register update>}
```

In the previous example, we could add a default statement which just assigns sp to the current value of sp. The block would then look like:

```
when(io.en && io.push && (sp != (size-1).U)) {
  sp := sp + 1.U
} .elsewhen(io.en && io.pop && (sp > 0.U)) {
  sp := sp - 1.U
} .otherwise {
  sp := sp
}
```

The explicit assignment to preserve the value of sp is redundant in this case but it captures the point of the `.otherwise` statement.

The unless Clause

To complement the `when` statement, Chisel also supports an unless statement. The unless statement is a conditional assignment that triggers only if the condition is false. The general structure for the unless statement is:

```
unless ( <condition> ) { <assignments> }
```

For example, suppose we want to do a simple search of the contents of memory and determine the address that contains some number. Since we don't know how long the search will take, the module will output a done signal when it is finished and until then, we want to continue to search memory. The Chisel code for the module would look like:

```
class MemorySearch extends Module {
  val io = IO(new Bundle {
    val target  = Input(UInt(4.W))
    val address = Output(UInt(3.W))
    val en      = Input(Bool())
    val done    = Output(Bool())
  })
  val index  = RegInit(0.U(3.W))
  val list   = Vec(0.U, 4.U, 15.U, 14.U, 2.U, 5.U, 13.U)
  val memVal = list(index)

  val done = (memVal === io.target) || (index === 7.U)

  unless (done) {
    index := index + 1.U
  }
  io.done    := done
  io.address := index
}
```

In this example, we limit the size of the memory to 8 entries and use a vector of literals to create a read only memory. Notice that the unless statement is used to terminate the iteration if it see that the done signal is asserted. Otherwise, it will continue to increment the index in memory until it finds the value in target or reaches the last index in the memory (7).

Combinational Conditional Assignment

You can also use the `when .elsewhen .otherwise` block to define combinational values that may take many values. For example, the following Chisel code show how to implement a basic arithmetic unit with 4 operations: add, subtract, and pass. In this example, we check the opcode to determine which operation to perform and conditionally assign the output.

```
class BasicALU extends Module {
  val io = IO(new Bundle {
    val a      = Input(UInt(4.W))
    val b      = Input(UInt(4.W))
    val opcode = Input(UInt(2.W))
    val output = Output(UInt(4.W))
  })
  io.output := 0.U
  when (io.opcode === 0.U) {
    io.output := io.a + io.b   // ADD
  } .elsewhen (io.opcode === 1.U) {
    io.output := io.b - io.b   // SUB
  } .elsewhen (io.opcode === 2.U) {
    io.output := io.a        // PASS A
  } .otherwise {
    io.output := io.b          // PASS B
  }
}
```

Notice that this can easily be easily expanded to check many different conditions for more complicated arithmetic units or combinational blocks.

Read Only Memories

To instantiate read only memories in Chisel, we use a vector of constant literals. For example, in order to instantiate an 4 entry read only memory with the values 0 to 3, the definition would look like the following:

```
val numbers =
  Vec(0.U, 1.U, 2.U, 3.U)
```

The width of the Vec is width of the widest argument. Notice that we need to specify the type of literal in the ... braces following the literals. Accessing the values in the read only memory is the same as accessing an entry in a Vec. For example, to access the 2nd entry of the memory we would use:

```
val entry2 = numbers(2)
```

Read-Write Memories

Chisel contains a primitive for memories called Mem. Using the Mem class it is possible to construct multi-ported memory that can be synchronous or asynchronous read.

Basic Instantiation

The Mem construction takes a memory size and a data type which it is composed of. The general declaration structure looks like:

```
val myMem = Mem(<size>, <type>)
```

Where corresponds to the number of entries of are in the memory.

For instance, if you wanted to create a 128 entry memory of 32 bit UInt types, you would use the following instantiation:

```
val myMem = Mem(128, UInt(32.W))
```

Note that when constructing a memory in Chisel, the initial value of memory contents cannot be specified. Therefore, you should never assume anything about the initial contents of your Mem class.

Synchronous vs. Asynchronous Read

It is possible to specify either synchronous or asynchronous read behavior.

For instance, if we wanted an asynchronous read 128 entry memory of 32 bit UInt types, we would use the following definition:

```
val combMem =
  Mem(128, UInt(32.W))
```

Likewise, if we wanted a synchronous read 128 entry memory of 32 bit UInt types, we use a SeqMem object:

```
val seqMem =
  SyncReadMem(128, UInt(32.W))
```

Adding Write Ports

To add write ports to the Mem, we use a `when` block to allow Chisel to infer a write port. Inside the `when` block, we specify the location and data for the write transaction. In general, adding a write port requires the following definition:

```
when (<write condition> ) {
  <memory name>( <write address> ) := <write data>
}
```

Where refers to the entry number in the memory to write to. Also notice that we use the reassignment operator := when writing to the memory.

For example, suppose we have a 128 entry memory of 32 bit UInt types. If we wanted to write a 32 bit value dataIn to the memory at location writeAddr if the write enable signal wen is true, our Chisel code would look like:

```
...
val myMem = Mem(128, UInt(32.W))
val wen = io.writeEnable
val writeAddr = io.waddr
val dataIn = io.wdata
when (wen) {
  myMem(writeAddr) := dataIn
}
...
```

<what is the behavior of multiple write ports?>

Adding Read Ports

Depending on the type of read behaviour specified, the syntax for adding read ports to Mem in Chisel is slightly different for asynchronous read and synchronous read memories.

Asynchronous Read Ports For asynchronous read memories, adding read ports to the memory simply amounts to placing an assignment inside a `when` block with some trigger condition. If you want Chisel to infer multiple read ports, simply add more assignments in the `when` definition. The general definition for read ports is thus:

```
when (<read condition>) {
  <read data 1> := <memory name>( <read address 1> )
  ...
  <read data N> := <memory name>( <read address N>)
}
```

For instance, if you wanted a 128 entry memory of 32 bit UInt values with two asynchronous read ports, with some read enable `re` and reads from addresses `raddr1` and `raddr2`, we would use the following `when` block definition:

```
...
val myMem = Mem(128, UInt(32.W))
val raddr1 = io.raddr
val raddr2 = io.raddr + 4.U
val re = io.readEnable
val read_port1 = UInt(32.W)
val read_port2 = UInt(32.W)
when (re) {
  read_port1 := myMem(raddr1)
  read_port2 := myMem(raddr2)
}
...
```

Note that the type and width of the `read_port1` and `read_port2` should match the type and width of the entries in the Mem.

Synchronous Read Ports In order to add synchronous read ports to the Chisel Mem class, Chisel requires that the output from the memory be assigned to a Reg type. Like the asynchronous read port, a synchronous read assignment must occur in a `when` block. The general structure for the definition of a synchronous read port is as follows:

```
...
val myMem = SyncReadMem(128, UInt(32.W))
```

```
  val raddr = io.raddr
  val read_port = Reg(UInt(32.W))
  when (re) {
    read_port := myMem(raddr)
  }
  ...
```

Example of Mem in Action

Here we provide a small example of using a memory by implementing a stack.

Suppose we would like to implement a stack that takes two signals push and pop where push tells the stack to push an input dataIn to the top of the stack, and pop tells the stack to pop off the top value from the stack. Furthermore, an enable signal en disables pushing or popping if not asserted. Finally, the stack should always output the top value of the stack.

```
class Stack(size: Int) extends Module {
  val io = IO(new Bundle {
    val dataIn  = Input(UInt(32.W))
    val dataOut = Output(UInt(32.W))
    val push    = Input(Bool())
    val pop     = Input(Bool())
    val en      = Input(Bool())
  })

  // declare the memory for the stack
  val stack_mem = Mem(size, UInt(32.W))
  val sp = RegInit(0.U(log2Ceil(size).W))
  val dataOut = RegInit(0.U(32.W))

  // Push condition - make sure stack isn't full
  when(io.en && io.push && (sp != (size-1).U)) {
    stack_mem(sp + 1.U) := io.dataIn
    sp := sp + 1.U
  }
    // Pop condition - make sure the stack isn't empty
    .elsewhen(io.en && io.pop && (sp > 0.U)) {
    sp := sp - 1.U
  }

  when(io.en) {
    dataOut := stack_mem(sp)
  }

  io.dataOut := dataOut
}
```

Since the module is parametrized to be have size entries, in order to correctly extract the minimum width of the stack pointer sp we take the log2Ceil(size). This takes the base 2 logarithm of size and rounds up.

Load/Search Mem Problem

In this assignment, write a memory module that supports loading elements and searching based on the following template:

```
class DynamicMemorySearch(val n: Int, val w: Int) extends Module {
  val io = IO(new Bundle {
    val isWr   = Input(Bool())
    val wrAddr = Input(UInt(log2Ceil(n).W))
    val data   = Input(UInt(w.W))
    val en     = Input(Bool())
    val target = Output(UInt(log2Ceil(n).W))
    val done   = Output(Bool())
  })
  val index  = RegInit(0.U(log2Ceil(n).W))
  val memVal = 0.U
  /// fill in here
  io.done   := false.B
  io.target := index
}
```

and found in $TUT_DIR/src/main/scala/problems/DynamicMemorySearch.scala. Notice how it support size and width parameters n and w and how the address width is computed from the size. Run

```
./run-problem.sh DynamicMemorySearch
```

until your circuit passes the tests.

Using the For loop

Often times parametrization requires instantiating multiple components which are connected in a very regular structure. A revisit to the parametrized `Adder` component definition shows the `for` loop construct in action:

```
// A n-bit adder with carry in and carry out
class Adder(n: Int) extends Module {
  val io = IO(new Bundle {
    val A    = Input(UInt(n.W))
    val B    = Input(UInt(n.W))
    val Cin  = Input(UInt(1.W))
    val Sum  = Output(UInt(n.W))
    val Cout = Output(UInt(1.W))
  })
  // create a vector of FullAdders
  val FAs = Vec(Seq.fill(n) { Module(new FullAdder()).io })
  val carry = Vec(Seq.fill(n + 1) { UInt(1.W) })
  val sum = Vec(Seq.fill(n) { Bool() })
  // first carry is the top level carry in
  carry(0) := io.Cin

  // wire up the ports of the full adders
  for (i <- 0 until n) {
    FAs(i).a := io.A(i)
    FAs(i).b := io.B(i)
    FAs(i).cin := carry(i)
    carry(i + 1) := FAs(i).cout
    sum(i) := FAs(i).sum.toBool()
  }
  io.Sum := sum.asUInt
  io.Cout := carry(n)
}
```

Notice that a Scala integer `i` value is used in the `for` loop definition as the index variable. This indexing variable is specified to take values from 0 `until` n, which means it takes values 0, 1, 2..., n-1. If we wanted it to take values from 0 to n inclusive, we would use `for (i <- 0 to n)`.

It is also important to note, that the indexing variable `i` does not actually manifest itself in the generated hardware. It exclusively belongs to Scala and is only used in declaring how the connections are specified in the Chisel component definition.

The for loop construct is also very useful for assigning to arbitrarily long `Vec`s

Using If, Else If, Else

As previously mentioned, the `if, elseif,` and `else` keywords are reserved for Scala control structures. What this means for Chisel is that these constructs allow you to selectively generate different structures depending on parameters that are supplied. This is particularly useful when you want to turn certain features of your implementation "on" or "off", or if you want to use a different variant of some component.

For instance, suppose we have several simple counters that we would like to package up into a general purpose counter module: UpCounter, DownCounter, and OneHotCounter. From the definitions below, we notice that for these simple counters, the I/O interfaces and parameters are identical:

```
// Simple up counter that increments from 0 and wraps around
class UpCounter(CounterWidth: Int) extends Module {
  val io = IO(new Bundle {
    val output = Output(UInt(CounterWidth.W))
    val ce     = Input(Bool())
  })...
}

// Simple down counter that decrements from
// 2^CounterWidth-1 then wraps around
class DownCounter(CounterWidth: Int) extends Module {
  val io = IO(new Bundle {
    val output = Output(UInt(CounterWidth.W))
    val ce     = Input(Bool())
  })...
}

// Simple one hot counter that increments from one hot 0
// to CounterWidth-1 then wraps around
class OneHotCounter(CounterWidth:Int) extends Module {
  val io = IO(new Bundle {
    val output = Output(UInt(CounterWidth.W))
    val ce     = Input(Bool())
  })...
}
```

We could just instantiate all three of these counters and multiplex between them but if we needed one at any given time this would be a waste of hardware. In order to choose between which of these three counters we want to instantiate, we can use Scala's `if, else if, else` statements to tell Chisel how to pick which component to instantiate based on a `CounterType` parameter:

```
class Counter(CounterWidth: Int, CounterType: String)
    extends Module {
  val io = IO(new Bundle {
    val output = Output(UInt(CounterWidth.W))
    val ce     = Input(Bool())
  })
  if (CounterType == "UpCounter") {
     val upcounter = new UpCounter(CounterWidth)
     upcounter.io.ce := io.ce
     io.output := upcounter.io.output
  } else if (CounterType == "DownCounter") {
    val downcounter = new DownCounter(CounterWidth)
    downcounter.io.ce := io.ce
    io.output := downcounter.io.output
  } else if (CounterType == "OneHotCounter") {
    val onehotcounter = new OneHotCounter(CounterWidth)
    onehotcounter.io.ce := io.ce
    io.output := onehotcounter.io.output
  } else {
    // default output 1
    io.output := 1.U
  }
}
```

By consolidating these three counter components into a single `Counter` module, we can instantiate a different counter by simply changing the parameter `CounterType`. For instance:

```
// instantiate a down counter of width 16
val downcounter =
  Module(new Counter(16, "DownCounter"))
```

```
// instantiate an up counter of width 16
val upcounter =
  Module(new Counter(16, "UpCounter"))

// instantiate a one hot counter of width 16
val onehotcounter =
  Module(new Counter(16, "OneHotCounter"))
```

This allows seamless alternation between them.

Using def

Chisel also allows the usage of the Scala `def` statement to define Chisel code that may be used frequently. These `def` statements can be packaged into a Scala Object and then called inside a Module. The following Chisel code shows an alternate implementation of an counter using `def` that increments by `amt` if the `inc` signal is asserted.

```
object Counter {
  def wrapAround(n: UInt, max: UInt) =
    Mux(n > max, 0.U, n)

  def counter(max: UInt, en: Bool, amt: UInt) = {
    val x = RegInit(0.U(max.getWidth.W))
    x := wrapAround(x + amt, max)
    x
  }
}

class Counter extends Module {
  val io = IO(new Bundle {
    val inc = Input(Bool())
    val amt = Input(UInt(4.W))
    val tot = Output(UInt(8.W))
  })
  io.tot := counter(255.U, io.inc, io.amt)
}
```

In this example, we use calls to subroutines defined in the `Counter` object in order to perform the appropriate logic.

Parameterized Vec Shift Reg

The next assignment is to construct a bit shift register with delay parameter. The following is the template from `$TUT_DIR/src/main/scala/problems/VecShiftRegisterParam.scala`:

```
class VecShiftRegisterParam(val n: Int, val w: Int) extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(w.W))
```

```
      val out = Output(UInt(w.W))
    })
    ...
    io.out := 0.U
  }
```

where out is a n cycle delayed copy of values on in.

Also notice how val is added to each parameter value to allow those values to be accessible from the tester.

Edit $TUT_DIR/src/main/scala/problems/VecShiftRegisterParam.scala and run:

```
./run-problem.sh VecShiftRegisterParam
```

until your circuit passes the tests.

Mul Lookup Table

The next assignment is to write a 16x16 multiplication table using Vec. The following is the template from $TUT_DIR/src/main/scala/problems/Mul.scala:

```
class Mul extends Module {
  val io = IO(new Bundle {
    val x    = Input(UInt(4.W))
    val y    = Input(UInt(4.W))
    val z    = Output(UInt(8.W))
  })
  val muls = new ArrayBuffer[UInt]()

  // flush this out ...

  io.z := 0.U
}
```

As a hint build the lookup table using a rom constructed from the tab lookup table represented as a Scala ArrayBuffer with incrementally added elements (using +=):

```
val tab = Vec(muls)
```

and lookup the result using an address formed from the x and y inputs as follows:

```
io.z := tab(Cat(io.x, io.y))
```

Edit `$TUT_DIR/src/main/scala/problems/Mul.scala` and run:

```
./run-problem.sh Mul
```

until your circuit passes the tests.

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 relationship 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

```
class WriteCmd extends ReadCmd {
  val data = UInt(Router.addressWidth.W)
}

class Packet extends Bundle {
  val header = UInt(Router.headerWidth.W)
  val body   = UInt(Router.dataWidth.W)
}
```

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

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

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 outputs. 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.

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.

How can I see a VCD output from the tutorial circuits

Here is the quick way. We'll take advantage of the fact that the verilator backend creates a vcd output file by default, so we'll make it run instead of the default interpreter backend. Let's use the example of running the examples.Adder tutorial. Run it as follows from the command line prompt.

```
./run-examples.sh Adder --backend-name verilator
```

The vcd file will appear here

```
ls -l test_run_dir/examples.Launcher.*/*.vcd
```

It is also possible to add your own Driver call which would allow you access to a whole slew of options that are hidden by the tutorials Launchers. Once again with examples.Adder, modify the test harness file `src/test/scala/examples/AdderTests.scala` Add a new tester class to the end the file so it looks like this

```
class MyAdderTester extends ChiselFlatSpec {
  behavior of "Adder"
  it should "run while creating a vcd" in {
    Driver.execute(Array("-fiwv"), () => new Adder(10)) { c =>
      new AdderTests(c)
    }
  }
}
```

You can now run the test from the command line as follows

```
sbt 'test-only examples.MyAdderTester'
```

You'll get a vcd output from the interpreter due to the `-fiwv` flag being passed to the backend. The output is in a different directory when run this way. It should be here

```
ls -ltr test_run_dir/examples.MyAdderTester*/*.vcd
```

For fun, try changing the `-fiwv` flag to `--help` and see a comprehensive list of options