Contents

1 Writing chisel

1.1 Prerequisites

- **You should have some idea of how digital logic circuits work.**

  You should have a basic overview on digital circuits. It is assumed that you know what a multiplexer is and how it works, and how a register works.

- **You must be able to run scala programs.**

  If you can run java then you can run scala. If not grab the jvm. Remember to curse Larry Page if you pick it up from the oracle webpage.

- **Some flavor of GNU/Linux, or at least something UNIX-like.**

  If you use anything other than Ubuntu 16.04 or 18.04 I won't be able to offer help if something goes wrong. Running chisel on OSX is probably possible, but you'll have to figure it out yourself.

- **An editor suited for scala.**

  My personal recommendation is GNU Emacs with emacs-lsp for IDE features along with the metals language server (which works for any editor with lsp (language server protocol), such as vim, vscode and atom). If you prefer an IDE I hear good things about intelliJ, however I haven't tested it personally, so if odd stuff happens I can't help you. **DON'T NEGLECT THIS!** This course is hard enough as it is, no point making it harder by neglecting basic IDE functionality. There are plenty of good options, no matter if you're a vim user or prefer GUI based full fledged IDEs.

- **Optional: sbt**

  You can install the scala build tool on your system, but for convenience I've included a bootstrap script in sbt.sh. If you want to install sbt on your own machine, sbt will select the correct version on a per-project basis for you, so you don't have to worry about getting the wrong version.

1.2 Terms

Before delving into code it's necessary to define some terms.

- **Wire**

  A wire is a bundle of 1 to N condictive wires (yes, that is a recursive definition, but I think you get what I mean). These wires are connected either to ground or a voltage source, corresponding to 0 or 1, which is useful for representing numbers

  We can define a wire consisting of 4 physical wires in chisel like this

  ```
  val myWire = Wire(UInt(4.W))
  ```

- **Driving**

  A wire in on itself is rather pointless since it doesn't do anything. In order for something to happen we need to connect them.

  ```
  val wireA = Wire(UInt(4.W))
  val wireB = Wire(UInt(4.W))
  wireA := 2.U
  wireB := wireA
  ```

Here wireA is driven by the signal 2.U, and wireB is driven by wireA.

For well behaved circuits it does not make sense to let a wire be driven by multiple sources which would make the resulting signal undefined.

Similarly a circular dependency is not allowed a la

```
val wireA = Wire(UInt(4.W))
val wireB = Wire(UInt(4.W))
wireA := wireB
wireB := wireA
```

Physically it **is** possible to have multiple drivers, but it's not a good idea as attempting to drive a wire with 0 and 1 simultaneously causes a short circuit which is definitely not a good thing.

- **Module**

  In order to make development easier we separate functionality into modules, defined by its inputs and outputs.

- **Combinatory circuit**

  A combinatory circuit is a circuit whose output is based only on its inputs.

- **Stateful circuit**

  A circuit that will give different results based on its internal state. In common parlance, a circuit without registers (or memory) is combinatory while a circuit with registers is stateful.

- **Clock**

  The clock is omnipresent in chisel, which paradoxically leads to it being nearly invisible. Every register in chisel is automatically wired up to the clock signal, and it may only update its contents when the clock ticks. Typically the term **cycle** is used to describe a single clock tick.

- **Chisel Graph**

  A chisel program is a program whose result is a graph which can be synthesized to a transistor level schematic of a logic circuit. When connecting wires wireA and wireB we were actually manipulating a graph (actually, two subgraphs that were eventually combined into one). The chisel graph is directed, but it does allow cycles so long as they are not combinatorial.

1.3 Your first component

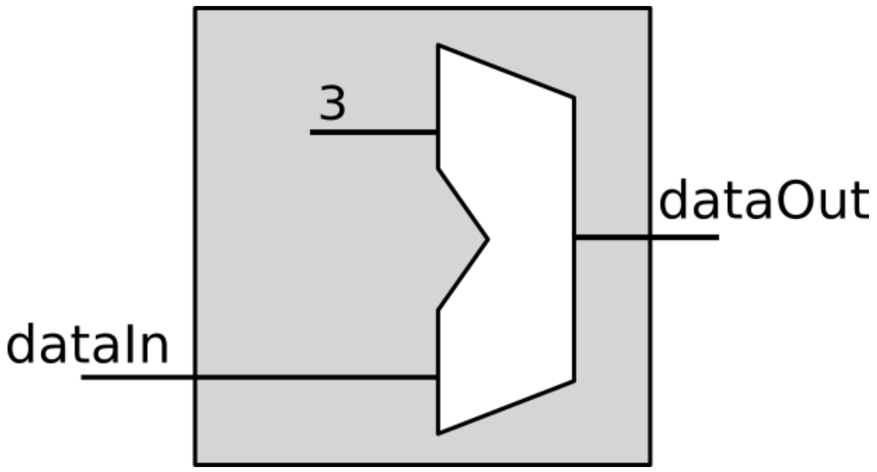
The code for the snippets in this subchapter can be found in the directory src/test/scala/Examples/. You can run them using sbt by running ./sbt in your project root which will open your sbt console.

This will start a large download, so be patient even if it looks like it's stuck.

The first component we will consider is a simple combinatorial incrementor:

```
// Using `val` in a class argument list makes that value
↪  public.
class MyIncrement(val incrementBy: Int) extends Module {
  val io = IO(
    new Bundle {
      val dataIn  = Input(UInt(32.W))
      val dataOut = Output(UInt(32.W))
    }
  )

  io.dataOut := io.dataIn + incrementBy.U
}
```

with incrementBy = 3 we get the following circuit:



The code for this circuit as well as the tests described next subsection can be found in src/test/scala/Examples/basic.scala

1.4 Testing your chisel component

After creating a module you might wonder how it can be run. It is not a program, it's a description of a digital circuit, so in order to "run" a chisel model we have to simulate it.

This is done by creating a test program where the test runner drives inputs and reads outputs from the module using what is called a peek poke tester.

1.4.1 Creating a peek poke tester

```
class TheTestRunner(c: MyIncrement) extends PeekPokeTester(c)
 ↪  {
  for(ii <- 0 until 5){
    poke(c.io.dataIn, ii)
    val o = peek(c.io.dataOut)
    println(s"At cycle $ii the output of myIncrement was $o")
    expect(c.io.dataOut, ii+c.incrementBy)
  }
}
```

There are three features of the peek poke tester on display here:

1. a peek poke tester has the ability to peek at a value, returning its state. This however is restricted to the modules **io only**
2. it has the ability to poke (set) the value of an input signal. Again, this can be done to **input io only**
3. It can expect a signal to have a certain value and throw an exception if not met. Expect is defined in terms of peek.

A peek poke tester can also *step* the circuit, this will be covered when stateful circuits have been introduced.

1.4.2 Running a peek poke tester

The test runner class we have defined requires a MyIncrement module that can be simulated. However, by writing `val inc3 = Module(new MyIncrement(3))` the return value is a *chisel graph*, which if you recall from the hdl introduction is not enough to actually test the circuit. In order to interact with a circuit the schematic must be interpreted, resulting in a **circuit simulator** which the peek poke tester can interact with.

If this isn't clear don't worry, in terms of code all we need to do is to invoke a chisel method for building the circuit:

```
chisel3.iotesters.Driver(() => new MyIncrement(3)) { c =>
  new TheTestRunner(c)
}
```

Unfortunately this code might be a little hard to parse if you're new to scala. Understanding it is not necessary, it is sufficient to simply swap `MyIncrement(3)` in `(() => MyIncrement(3))` with the module you want to test, and `TheTestRunner(c)` with the test runner you want to run.

```
chisel3.iotesters.Driver(() => new MyIncrement(3)) { c =>
  new TheTestRunner(c)
}
```

1.4.3 Putting it together into a runnable test

We want to be able to run our test from sbt. To do this we use the scalatest framework. A test looks like this:

```
class MyTest extends FlatSpec with Matchers {
  behavior of "the test that I have written"

  it should "sum two numbers" in {
    2 + 2
  } should be 4
}
```

The tester class introduces a lot of special syntax, but like above it is not necessary to understand how, simply using the template above is sufficient.

By applying our circuit description and test class to the tester template we end up with:

```
class MyIncrementTest extends FlatSpec with Matchers {
  behavior of "my increment"

  it should "increment its input by 3" in {
    chisel3.iotesters.Driver(() => new MyIncrement(3)) { c =>
    //                                ^^^^^^^^^^^^^^^ The
    ↪  component we want to test
      new TheTestRunner(c)
      //  ^^^^^^^^^^^^^^^^ The tester we want to run
    } should be(true)
  }
}
```

By creating this test it is now possible to run it from sbt. There are two ways to test. By simply writing "test" in the sbt console as so: `sbt:chisel-module-template> test` every single test will be run. Since this creates a lot of noise it's more useful to run "testOnly": `sbt:chisel-module-template> testOnly Examples.MyIncrementTest` where "Examples" is the name of the package and MyIncrementTest is the name of the test. The tests for the exercise itself is located in Ex0, so for instance `sbt:chisel-module-template> testOnly Ex0.MatrixSpec` will run the matrix test.

Note: by running "test" once you can use tab completion in the sbt shell to find tests with testOnly. using testOnly <TAB>

Running the test should look something like this.

```
sbt:chisel-module-template> testOnly Examples.MyIncrementTest
Run starting. Expected test count is: 0
...
Circuit state created
[info] [0.001] SEED 1556890076413
[info] [0.002] At cycle 0 the output of myIncrement was 3
[info] [0.003] At cycle 1 the output of myIncrement was 4
[info] [0.003] At cycle 2 the output of myIncrement was 5
[info] [0.003] At cycle 3 the output of myIncrement was 6
[info] [0.003] At cycle 4 the output of myIncrement was 7
test MyIncrementTestMyIncrement Success: 5 tests passed in 5
 ↪  cycles taking 0.011709 seconds
[info] [0.004] RAN 0 CYCLES PASSED
- should increment its input by 3
Run completed in 771 milliseconds.
...
```

In the Example.scala file you can find the entire test. The only difference is that everything is put in the same class.

1.5 Using modules

Let's see how we can use our module by instantiating it as a submodule:

```
class MyIncrementTwice(incrementBy: Int) extends Module {
  val io = IO(
    new Bundle {
      val dataIn  = Input(UInt(32.W))
      val dataOut = Output(UInt(32.W))
    }
  )

  val first  = Module(new MyIncrement(incrementBy))
  val second = Module(new MyIncrement(incrementBy))

  first.io.dataIn  := io.dataIn
  second.io.dataIn := first.io.dataOut

  io.dataOut := second.io.dataOut
}
```

The RTL diagram now looks like this:

Note how the modules `first` and `second` are now drawn as black boxes. When drawing RTL diagrams we're not interested in the internals of submodules.

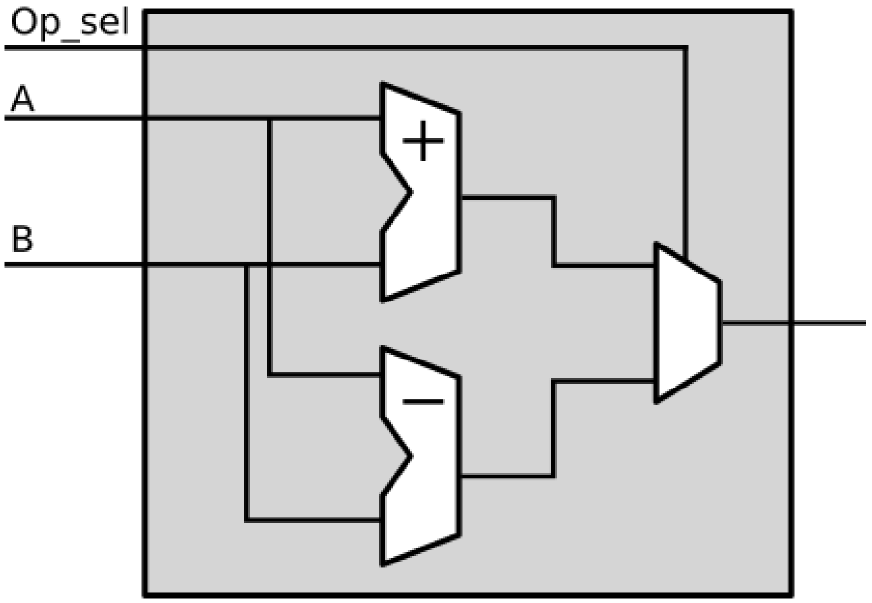
However, what if we want to instantiate an arbitrary amount of incrementors and chain them? To see how this can be done it is necessary to take a detour:

1.6 Leveraging Chisel with Scala

Recall from the hdl chapter how a chisel program is using scala to build chisel. To give an idea of what that means let's consider conditional statements in chisel:

```
class ChiselConditional() extends Module {
  val io = IO(
    new Bundle {
      val a = Input(UInt(32.W))
      val b = Input(UInt(32.W))
      val opSel = Input(Bool())

      val out = Output(UInt(32.W))
    }
  )

  when(io.opSel){
    io.out := io.a + io.b
  }.otherwise{
    io.out := io.a - io.b
  }
}
```

This code describes the following circuit:



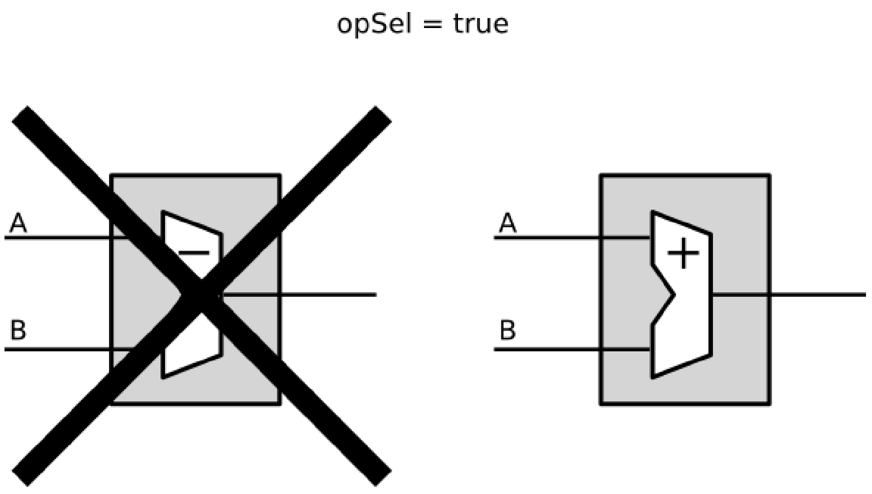
If the RTL is unfamiliar, the two leftmost components are ALUs which do arithmetic (addition and subtraction in this case) The rightmost component is a multiplexer which selects an input signal based on a selector signal, kind of like a railroad switch.

These conditional statements are implemented at a hardware level, but what is their relation to scalas if else statements?

Lets consider an example using if and else:

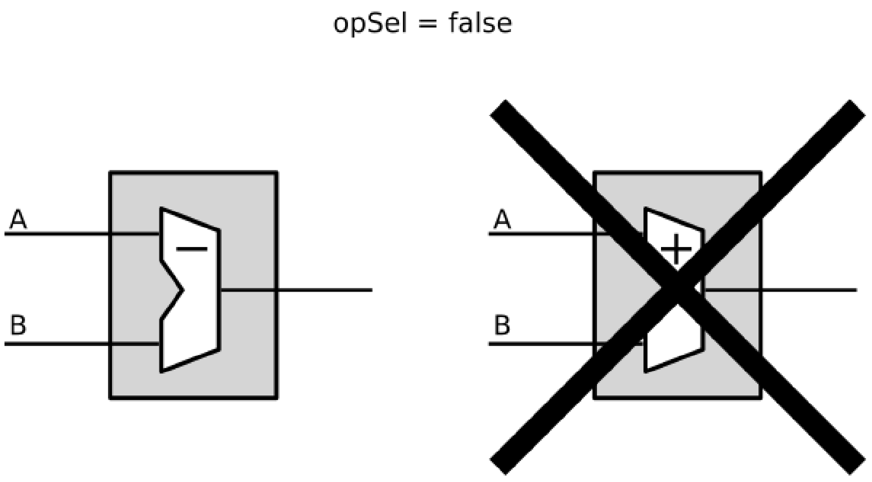
```
class ScalaConditional(opSel: Boolean) extends Module {
  val io = IO(
    new Bundle {
      val a = Input(UInt(32.W))
      val b = Input(UInt(32.W))
      val out = Output(UInt(32.W))
    }
  )

  if(opSel){
    io.out := io.a + io.b
  } else {
    io.out := io.a - io.b
  }
}
```

Which can yield two different circuits depending on the opSel argument:
True:



.
.
.
.
.
.

False:



Let's look at how we can use another scala construct, the for loop, to create several modules and chain them together:

```
class MyIncrementN(val incrementBy: Int, val numIncrementors:
↪  Int) extends Module {
  val io = IO(
    new Bundle {
      val dataIn  = Input(UInt(32.W))
      val dataOut = Output(UInt(32.W))
    }
  )

  val incrementors = Array.fill(numIncrementors){ Module(new
  ↪  MyIncrement(incrementBy)) }

  for(ii <- 1 until numIncrementors){
    incrementors(ii).io.dataIn := incrementors(ii -
    ↪  1).io.dataOut
  }

  incrementors(0).io.dataIn := io.dataIn
  io.dataOut := incrementors.last.io.dataOut
}
```

Keep in mind that the for-loop only exists at design time, just like a for loop generating a table in HTML will not be part of the finished HTML.

1.6.1 Troubleshooting scala and chisel mixups

With the when/otherwise and if/else example the meanings were different, as showed in the resulting circuitry. It is typical to accidentally mix up chisel and scala however, and typically this will not yield a valid program, instead you get compiler errors, which if you recall the toolchain figure in the HDL chapter corresponds to the compiler stage between scala code and a chisel graph builder.

To show some typical errors consider the following code which can be found in src/test/scala/Examples/myVector.scala (The non-compiling examples are commented out)

```
class MyVector() extends Module {
  val io = IO(
    new Bundle {
      val idx = Input(UInt(32.W))
      val out = Output(UInt(32.W))
    }
  )

  val values = List(1, 2, 3, 4)

  io.out := values(io.idx)
}
```

If you uncomment and try to compile this you will get an error: (only running compile works, as it will only compile the code in src/main*)

```
sbt:chisel-module-template> test:compile
...
[error]  found   : chisel3.core.UInt
[error]  required: Int
[error]   io.out := values(io.idx)
[error]                       ^
```

This error tells you that io.idx was of the wrong type, namely a `chisel3.core.UInt`. The List is a scala construct, it only exists when your design is synthesized, thus attempting to index it with a chisel type does not make sense.

Let's try again using a chisel `Vec` which can be indexed by chisel values:

```
class MyVector() extends Module {
  val io = IO(
    new Bundle {
      val idx = Input(UInt(32.W))
      val out = Output(UInt(32.W))
    }
  )

  // val values: List[Int] = List(1, 2, 3, 4)
  val values = Vec(1, 2, 3, 4)

  io.out := values(io.idx)
}
```

Now you will get the following error instead:

```
sbt:chisel-module-template> test:compile
...
[error] /home/peteraa/datateknikk/TDT4255_EX0/src/main/scala/T⌟
↪  ile.scala:30:16: inferred type arguments [Int] do not
↪  conform to macro method apply's type parameter bounds [T
↪  <: chisel3.Data]
[error]   val values = Vec(1, 2, 3, 4)
[error]                ^
[error] /home/peteraa/datateknikk/TDT4255_EX0/src/main/scala/T⌟
↪  ile.scala:30:20: type
↪  mismatch;
[error]  found   : Int(1)
[error]  required: T
[error]   val values = Vec(1, 2, 3, 4)
...
```

The error states that the type `Int` cannot be constrained to a `type T <: chisel3.Data` which needs a little unpacking:

The `<:` symbol means subtype, meaning that the compiler expected the Vec to contain a chisel data type such as chisel3.Data.UInt or chisel3.Data.Boolean, and Int is not one of them!

A scala int represent 32 bits in memory, whereas a chisel UInt represents a bundle of wires that we interpret as an unsigned integer, thus they are not interchangeable. The difference between `scala.Integer` and `chisel3.Data.UInt` is analogous to that of if/else vs when/otherwise seen in the previous section.

To fix this, chisel UInts must be used

```
class MyVector() extends Module {
  val io = IO(
    new Bundle {
      val idx = Input(UInt(32.W))
      val out = Output(UInt(32.W))
    }
  )

  val values = Vec(1.U, 2.U, 3.U, 4.U)

  // Alternatively
  // val values = Vec(List(1, 2, 3, 4).map(scalaInt =>
  ↪  UInt(scalaInt)))

  io.out := values(io.idx)
}
```

Which compiles.

You might be suprised to see that it is possible to index a Vec with an integer as such:

```
class MyVector() extends Module {
  val io = IO(
    new Bundle {
      val idx = Input(UInt(32.W))
      val out = Output(UInt(32.W))
    }
  )

  val values = Vec(1.U, 2.U, 3.U, 4.U)

  io.out := values(3)
}
```

In this case `3` gets automatically changed to `3.U`. It's not a great idea to abuse implicit conversions, so you should refrain from doing this too much. The version above can be run with: `sbt:chisel-module-template> testOnly Examples.MyVecSpec`

In order to get some insight into how a chisel Vec works, let's see how we can implement myVector without Vec:

```scala
class MyVectorAlt() extends Module {
  val io = IO(
    new Bundle {
      val idx = Input(UInt(32.W))
      val out = Output(UInt(32.W))
    }
  )

  val values = Array(0.U, 1.U, 2.U, 3.U)

  io.out := values(0)
  for(ii <- 0 until 4){
    when(io.idx(1, 0) === ii.U){
      io.out := values(ii)
    }
  }
}
```

The for-loop creates 4 conditional blocks boiling down to when 0: output the value in values(0) when 1: output the value in values(1) when 2: output the value in values(2) when 3: output the value in values(3) otherwise: output 0.U

The otherwise clause will never occur, chisel is unable to inferr this (however the synthesizer will likely be able to)

In the conditional block the following syntax is used: `io.idx(1, 0) === ii.U)` which indicates that only the two low bits of idx will be used to index, which is how chisel Vec does it.

From this you can gather that a chisel Vec doesn't really exist on the resulting circuit. Then again, an array is nothing more than an address, so this is in some respects analogous to how a computer works.

1.6.2 Troubleshooting build time errors

In the HTML example, assume that the the last </ul> tag was ommited. This would not be valid HTML, however the code will happily compile. Likewise, you can easily create a valid scala program producing an invalid chisel graph, such as this module found in src/test/scala/Examples/invalidDesigns.scala

```scala
class Invalid() extends Module {
  val io = IO(new Bundle{})
  val myVec = Module(new MyVector)
}
```

This code will happily compile, however when you attempt to create a simulator from the chisel graph the driver will throw an exception. To see this you can run the following test (already implemented in invalidDesigns.scala):

```
class InvalidSpec extends FlatSpec with Matchers {
  behavior of "Invalid"

  it should "fail" in {
    chisel3.iotesters.Driver(() => new Invalid) { c =>

      // chisel tester expects a test here, but we can use ???
      // which is shorthand for throw new
      ↪  NotImplementedException.
      //
      // This is OK, because it will fail during building.
      ???
    } should be(true)
  }
}
```

```
sbt:chisel-module-template> compile:test
...
```

As promised, this code compiles, but when you run the test which actually builds a simulator you get the following:

```
[success] Total time: 3 s, completed Apr 25, 2019 3:15:15 PM
...
sbt:chisel-module-template> testOnly Examples.InvalidSpec
...
firrtl.passes.CheckInitialization$RefNotInitializedException:
↪  @[Example.scala 25:21:@20.4] : [module Invalid]  Reference
↪  myVec is not fully initialized.
 : myVec.io.idx <= VOID
at firrtl.passes.CheckInitialization$.$anonfun$run$6(CheckInit⌋
↪  ialization.scala:83)
at firrtl.passes.CheckInitialization$.$anonfun$run$6$adapted(C⌋
↪  heckInitialization.scala:78)
```

```
at scala.collection.TraversableLike$WithFilter.$anonfun$foreac⌟
↪  h$1(TraversableLike.scala:789)
at scala.collection.mutable.HashMap.$anonfun$foreach$1(HashMap⌟
↪  .scala:138)
at scala.collection.mutable.HashTable.foreachEntry(HashTable.s⌟
↪  cala:236)
at scala.collection.mutable.HashTable.foreachEntry$(HashTable.⌟
↪  scala:229)
at scala.collection.mutable.HashMap.foreachEntry(HashMap.scala⌟
↪  :40)
at scala.collection.mutable.HashMap.foreach(HashMap.scala:138)
at scala.collection.TraversableLike$WithFilter.foreach(Travers⌟
↪  ableLike.scala:788)
at firrtl.passes.CheckInitialization$.checkInitM$1(CheckInitia⌟
↪  lization.scala:78)
```

While scary, the actual error is only this line:

```
firrtl.passes.CheckInitialization$RefNotInitializedException:
↪  @[Example.scala 25:21:@20.4] : [module Invalid]  Reference
↪  myVec is not fully initialized.
 : myVec.io.idx <= VOID
```

Which tells you that myVec.io.idx is unconnected, i.e it needs a driver.

```
// Now actually valid...
class Invalid() extends Module {
  val io = IO(new Bundle{})

  val myVec = Module(new MyVector)
  myVec.io.idx := 0.U
}
```

After fixing the invalid circuit and running the test you will insted get a large error stack trace where you will see that: `- should fail *** FAILED ***` Which I suppose indicates success.

1.7 Stateful circuits

Until now every circuit we have consider has been a combinatory circuit. Consider the following circuit:

```
class SimpleDelay() extends Module {
  val io = IO(
    new Bundle {
      val dataIn  = Input(UInt(32.W))
      val dataOut = Output(UInt(32.W))
    }
  )
  val delayReg = RegInit(UInt(32.W), 0.U)

  delayReg    := io.dataIn
  io.dataOut := delayReg
}
```

This circuit stores its input in delayReg and drives its output with delayRegs output. Registers are driven by a clock signal in addition to the input value, and it is only capable of updating its value at a clock pulse.

In some HDL languages it is necessary to include the clock signal in the modules IO, but for chisel this happens implicitly.

When testing we use the `step(n)` feature of peek poke tester which runs the clock signal n times.

Test this by running `testOnly Examples.DelaySpec` The code for this is already implemented in src/test/scala/Examples/stateful.scala

```
class DelaySpec extends FlatSpec with Matchers {
  behavior of "SimpleDelay"

  it should "Delay input by one timestep" in {
    chisel3.iotesters.Driver(() => new SimpleDelay, verbose =
     ↪  true) { c =>
    //
     ↪  ^^^^^^^^^^^^^^ Optional parameter verbose set to true
      new DelayTester(c)
    } should be(true)
  }
}

class DelayTester(c: SimpleDelay) extends PeekPokeTester(c)  {
  for(ii <- 0 until 10){
    val input = scala.util.Random.nextInt(10)
    poke(c.io.dataIn, input)
    step(1)
    expect(c.io.dataOut, input)
  }
}
```

In order to make it extra clear the Driver has the optional "verbose" parameter set to true. This yields the following:

```
DelaySpec:
SimpleDelay
...
End of dependency graph
Circuit state created
[info] [0.001] SEED 1556898121698
[info] [0.002]   POKE io_dataIn <- 7
[info] [0.002] STEP 0 -> 1
[info] [0.002] EXPECT AT 1   io_dataOut got 7 expected 7 PASS
[info] [0.002]   POKE io_dataIn <- 8
[info] [0.002] STEP 1 -> 2
[info] [0.003] EXPECT AT 2   io_dataOut got 8 expected 8 PASS
[info] [0.003]   POKE io_dataIn <- 2
...
[info] [0.005] STEP 9 -> 10
[info] [0.005] EXPECT AT 10   io_dataOut got 7 expected 7 PASS
test SimpleDelay Success: 10 tests passed in 15 cycles taking
↪  0.010393 seconds
[info] [0.005] RAN 10 CYCLES PASSED
```

Following the output you can see how at step 0 the input is 7, then at step 1 the expected (and observed) output is 7.

1.8 Debugging

A rather difficult aspect in HDLs, including chisel is debugging. When debugging it is necessary to inspect how the state of the circuit evolves, which leaves us with two options, peekPokeTester and printf, however both have flaws.

Code for this section can be found at src/test/scala/Examples/printing.scala

1.8.1 PeekPoke

The peek poke tester should always give a correct result, if not it's a bug, not a quirk. Sadly, peek poke testing is rather limited in that it cannot be used to access **internal state**. Consider the following nested modules:

```
class Inner() extends Module {
  val io = IO(
    new Bundle {
      val dataIn  = Input(UInt(32.W))
      val dataOut = Output(UInt(32.W))
    }
  )
  val innerState = RegInit(0.U)
  when(io.dataIn % 2.U === 0.U){
    innerState := io.dataIn
  }

  io.dataOut := innerState
}


class Outer() extends Module {
  val io = IO(
    new Bundle {
      val dataIn  = Input(UInt(32.W))
      val dataOut = Output(UInt(32.W))
    }
  )

  val outerState = RegInit(0.U)
  val inner = Module(new Inner)

  outerState      := io.dataIn
  inner.io.dataIn := outerState
  io.dataOut      := inner.io.dataOut
}
```

It would be nice if we could use the peekPokeTester to inspect what goes on inside Inner, however this information is no longer available once Outer is rendered into a circuit simulator.

To see this, run `testOnly Example.PeekInternalSpec` Which throws an exception is thrown when either of the two peek statements underneath are run:

```scala
class OuterTester(c: Outer) extends PeekPokeTester(c)  {
  val inner = peek(c.inner.innerState)
  val outer = peek(c.outerState)
}
```

The only way to deal with this hurdle is to expose the state we are interested in as signals. An example of this can be seen in in the bottom of printing.scala

This approach leads to a lot of annoying clutter in your modules IO, so to separate business-logic from debug signals it is useful to use a MultiIOModule where debug signals can be put in a separate io bundle.

1.8.2 printf

`printf` and `println` must not be mixed! println behaves as expected in most languages, when executed it simply prints the argument. In the tests so far it has only printed the value returned by peek.

a printf statement on the other hand does not immediately print anything to the console. Instead it creates a special chisel element which only exists during simulation and prints to your console each clock cycle, thus helping us peer into the internal state of a circuit!

Additionally, a printf statement in a conditional block will only execute if the condiditon is met, allowing us to reduce noise.

```scala
class PrintfExample() extends Module {
  val io = IO(new Bundle{})

  val counter = RegInit(0.U(8.W))
  counter := counter + 1.U

  printf("Counter is %d\n", counter)
  when(counter % 2.U === 0.U){
    printf("Counter is even\n")
  }
}

class PrintfTest(c: PrintfExample) extends PeekPokeTester(c)  {
  for(ii <- 0 until 5){
    println(s"At cycle $ii:")
    step(1)
  }
}
```

When you run this test with `testOnly Examples.PrintfExampleSpec`, did you get what you expected?

As it turns out printf can be rather misleading when using stateful circuits. To see this in action, try running `testOnly Examples.EvilPrintfSpec` which yields the following

```
In cycle 0 the output of counter is: 0
according to printf output is: 0
[info] [0.003]
In cycle 1 the output of counter is: 0
according to printf output is: 0
[info] [0.003]


In cycle 2 the output of counter is: 0
according to printf output is: 1
                               ^^^^^^^^


[info] [0.004]
In cycle 3 the output of counter is: 1
according to printf output is: 1
[info] [0.004]
In cycle 4 the output of counter is: 1
according to printf output is: 1
```

When looking at the circuits design it is pretty obvious that the peek poke tester is giving the correct result, whereas the printf statement is printing the updated state of the register which should not be visible before next cycle.

In conclusion, do not use printf to debug timing issues, and if you do be extremely methodical.

(It is possible to use a different simulator, treadle, which from what I have seen gives correct printf results, it can be used by supplying an extra argument in the peek poke constructor like so: `chisel3.iotesters.Driver(() => new Outer, "treadle") { c =>` Just don't bank your money on the correctness, it might fail in rare circumstances making debugging a nightmare)

1.9 Visualizing generated circuits

While limited, it is possible to visualize your generated circuit using diagrammer. The necessary code to generate .fir file is in the main.scala file, just comment it out to generate these. I encourage you to give it a fair shake to see if you find it useful or not.

1.10 Visualizing circuit state (Optional)

In order to make debugging easier it is helpful to render the state of the circuit to see where errors happen. A prototype for this is included in this project, read more about it here Here

1.11 Resources

Chisel cheat sheet `https://chisel.eecs.berkeley.edu/doc/chisel-cheatsheet3.pdf`