Testing and Verification

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

March 6, 2025

Overview

- ► Review components
- ► A little bit of Scala (for testing)
- ► Debugging and testing
- ► Digital designers (sometimes) call testing verification
  - ► To distinguish from final chip testing

DTU Chip Day

- ▶ Note the date: Tu 8 April afternoon
- ▶ Start with sandwiches and finish with beer
- ▶ Presentation of chip design and verification work/companies in Denmark
- ▶ Several chip companies will present and are participating
- ▶ Opportunity to network for: theses with companies, internship, student jobs

Direction of a Connection

- ► The flow on a `Wire` has a direction
- ► One end is the output/driver/source and the other end is the input/sink
- ► Or producer and consumer
- ► An expression has also a direction:
  - ► The right hand site produces a value
  - ► The left hand site consumes the value
- ► `sink := source1 + source2`
- ► Like in Java and other programming languages
- ► Draw a figure

Last Chisel Lab (week 3)

- ▶ On components and small sequential circuits
  - ▶ Registers plus combinational circuits
- ▶ Did you finish the exercises?
  - ▶ Do the poll
- ▶ They are not mandatory, but helpful for preparation for the final project
- ▶ Let's look at solutions

Components are Modules

- ▶ Components are building blocks
  - ▶ Like concrete, physical ICs
- ▶ Components have input and output ports (= pins)
  - ▶ Organized as a `Bundle`
  - ▶ Assigned to the field `io`
- ▶ We build circuits as a hierarchy of components
  - ▶ You did a 4:1 multiplexer out of three 2:1 multiplexers
- ▶ In Chisel a component is called `Module`
- ▶ Components/Modules are used to organize the circuit
  - ▶ Similar to using methods in Java
  - ▶ But they are connected with *wires*

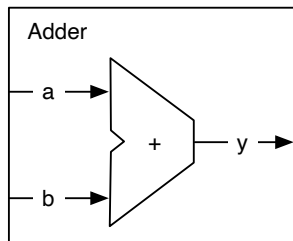# A Binary Watch

► Built out of discrete, digital components
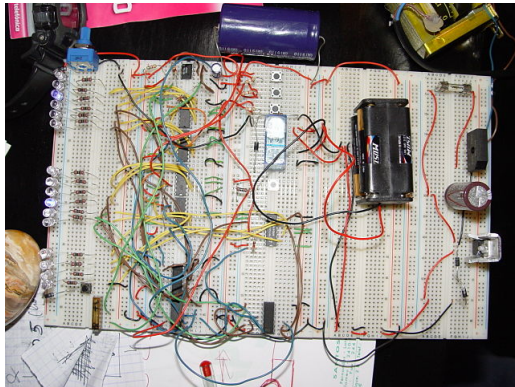
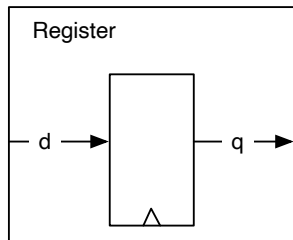Source: Diogo Sousa, public domain

Let Us Build a Counter

- ► Counting from 0 up to 9
- ► Restart from 0
- ► Build it out of components
- ► We need:
  - ► Adder
  - ► Register
  - ► Multiplexer
- ►
- ► But these are very tiny components

An Adder (Component/Module)


```
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val y = Output(UInt(8.W))
  })

  io.y := io.a + io.b
}
```
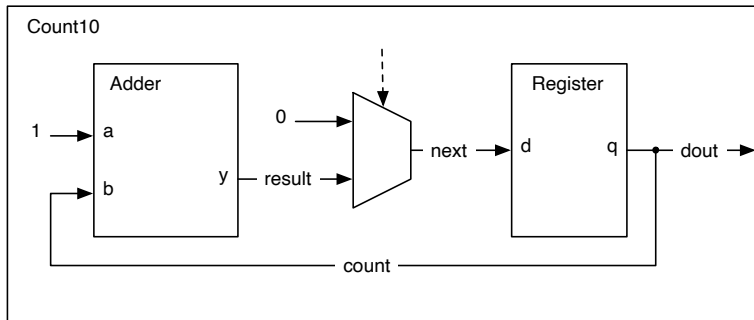
A Register


```scala
class Register extends Module
    {
  val io = IO(new Bundle {
    val d = Input(UInt(8.W))
    val q = Output(UInt(8.W))
  })

  val reg = RegInit(0.U)
  reg := io.d
  io.q := reg
}
```

The Counter Schematics

The Counter in Chisel

```
class Count10 extends Module {
  val io = IO(new Bundle {
    val dout = Output(UInt(8.W))
  })

  val add = Module(new Adder())
  val reg = Module(new Register())

  // the register output
  val count = reg.io.q
  // connect the adder
  add.io.a := 1.U
  add.io.b := count
  val result = add.io.y
  // connect the Mux and the register input
  val next = Mux(count === 9.U, 0.U, result)
  reg.io.d := next
  io.dout := count
}
```

Summarize Components

- ▶ Think like concrete components (ICs)
- ▶ They have named pins (`io.name`)
  - ▶ In hardware language these pins are often called ports
  - ▶ Ports have a direction (input or output)
- ▶ They need to be created:
  - ▶ `val mc = Module(new MyComponent())`
- ▶ and pins need to be connected with `:=`
- ▶ One module is special, as it is the top module

Chisel Main

- ▶ Create one top-level Module
- ▶ Invoke the `emitVerilog()` from the App
- ▶ Pass the top module (e.g., `new Hello()`)
- ▶ Optional: pass some parameters (in an `Array`)
- ▶ Following code generates Verilog code for the *Hello World*

```
object Hello extends App {
  emitVerilog(new Hello())
}
```

Scala

- ▶ Is object oriented
- ▶ Is functional
- ▶ Strongly typed with very good type inference
- ▶ Runs on the Java virtual machine
- ▶ Can call Java libraries
- ▶ Consider it as Java++
    - ▶ Can almost be written like Java
    - ▶ With a more lightweight syntax

Scala Hello World

```scala
//- start hello_scala
object HelloScala extends App {
  println("Hello Chisel World!")
}
//- end
```

- ▶ Compile with scalac and run with scala
- ▶ You can even use Scala as a scripting language
- ▶ Or run with sbt run
- ▶ Show both

Scala Values and Variables

- ► Scala has two type of variables: vals and vars
- ► A val cannot be reassigned, it is a constant
- ► We use a val to name a hardware component in Chisel

```scala
// A value is a constant
val zero = 0
// No new assignment is possible
// The following will not compile
zero = 3
```

- ► Types are usually inferred
- ► But can be explicitly stated as follows

```scala
val number: Int = 42
```

Scala Variables

▶ A var can be reassigned, it is like a classic variable

▶ We use a var to write a hardware generator in Chisel

```scala
// We can change the value of a var variable
var x = 2
x = 3
```

Simple Loops

```
// Loops from 0 to 9
// Automatically creates loop value i
for (i <- 0 until 10) {
  println(i)
}
```

▶ We can use a loop for testing

Scala `for` Loop for Circuit Generation

```scala
val regVec = Reg(Vec(8, UInt(1.W)))

regVec(0) := io.din
for (i <- 1 until 8) {
  regVec(i) := regVec(i-1)
}
```

- ▶ `for` is Scala
- ▶ This loop generates several connections
- ▶ The connections are parallel hardware
- ▶ This is a shift register

Conditions

```
for (i <- 0 until 10) {
  print(i)
  if (i%2 == 0) {
    println(" is even")
  } else {
    println(" is odd")
  }
}
```

▶ Executed at runtime, when the circuit is created
▶ This is *not* a mlutplexer

Testing and Debugging

- ▶ Nobody writes perfect code ;-)
- ▶ We need a method to improve the code
- ▶ In Java we can simply print values:
  - ▶ `println("42");`
- ▶ What can we do in hardware?
  - ▶ Describe the whole circuit and hope it works?
  - ▶ We can switch an LED on and off
  - ▶ Test it with switches and LEDs in an FPGA
- ▶ We need some tools for debugging
- ▶ Writing testers in Chisel
- ▶ We test by running a simulation of the circuit

ScalaTest

- ► Testing framework for Scala and Java
- ► Tests are placed under `src/test/scala`
- ► `sbt` understands ScalaTest
- ► Run all tests with:

  ```
  sbt test
  ```

- ► When all (unit) tests are ok, the test suit passes
- ► A little bit funny syntax
- ► ChiselTest is based on ScalaTest

Testing with Chisel

- A test contains
  - a device under test (DUT) and
  - the testing logic
- Set input values with `poke`
- Advance the simulation with `step`
- Read the output values with `peekInt`
- Compare the values with `expect`
- Import following packages

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec
```

An Example DUT

- A device-under test (DUT)
- Just 2-bit AND logic and equvicalence

```scala
class DeviceUnderTest extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
    val equ = Output(Bool())
  })

  io.out := io.a & io.b
  io.equ := io.a === io.b
}
```

A ChiselTest

- ▶ Extends class `AnyFlatSpec` with `ChiselScalatestTester`
- ▶ Has the device-under test (DUT) as parameter of the `test()` function
- ▶ Test function contains the test code
- ▶ Testing code can use all features of Scala
- ▶ Is placed in `src/test/scala`
- ▶ Is run with `sbt test`

A Simple Tester

▶ Just using `println` for manual inspection

```
class SimpleTest extends AnyFlatSpec with
    ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      println("Result is: " +
          dut.io.out.peekInt())
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      println("Result is: " +
          dut.io.out.peekInt())
    }
  }
}
```

A Real Tester

▶ Poke values and `expect` some output

```scala
class SimpleTestExpect extends AnyFlatSpec
    with ChiselScalatestTester {
  "DUT" should "pass" in {
    test(new DeviceUnderTest) { dut =>
      dut.io.a.poke(0.U)
      dut.io.b.poke(1.U)
      dut.clock.step()
      dut.io.out.expect(0.U)
      dut.io.a.poke(3.U)
      dut.io.b.poke(2.U)
      dut.clock.step()
      dut.io.out.expect(2.U)
    }
  }
}
```

Generating Waveforms

- ▶ Waveforms are timing diagrams
- ▶ Good to see many parallel signals and registers

  ```
  sbt "testOnly SimpleTest -- -DwriteVcd=1"
  ```

- ▶ Or setting an attribute for the test() function

  ```
  test(new DeviceUnderTest)
      .withAnnotations(Seq(WriteVcdAnnotation))
  ```

- ▶ IO signals and registers are dumped
- ▶ Option --debug puts all wires into the dump
- ▶ Generates a .vcd file in

  ```
  test_run_dir/test-name
  ```

- ▶ Viewing with GTKWave or ModelSim

Display Waveform with GTKWave

- ▶ Run the tester: `sbt test`
- ▶ Locate the .vcd file in test_run_dir/...
- ▶ Start GTKWave
- ▶ Open the .vcd file with
  - ▶ File – Open New Tab
- ▶ Select the circuit
- ▶ Drag and drop the interesting signals

Waveform Testing Demo

- ► Counter with a limit from last Chisel lab (`Count6`)
- ► Show Count6 tester: the original and the waveform
- ► Run it and look at waveform
- ► Add the solution
- ► Run again and reload the waveform

A Self-Running Circuit

- `Count6` is a self-running circuit
- Needs no stimuli (`poke`)
- Just run for a few cycles

```
test(new Count6) { dut =>
  dut.clock.step(20)
}
```

The WaveForm

- ▶ The complete test
- ▶ Note the .withAnnotations(Seq(WriteVcdAnnotation)

```
class Count6WaveSpec extends AnyFlatSpec with
    ChiselScalatestTester {
  "CountWave6 " should "pass" in {
    test(new
        Count6).withAnnotations(Seq(WriteVcdAnnotation))
        { dut =>
      dut.clock.step(20)
    }
  }
}
```

Display Waveform with GTKWave

- ▶ Run the tester: `sbt test`
- ▶ Locate the .vcd file in test_run_dir/...
- ▶ Start GTKWave
- ▶ Open the .vcd file with
  - ▶ File – Open New Tab
- ▶ Select the circuit
- ▶ Drag and drop the interesting signals

Vending Machine Testing

- ▶ I provide a minimal tester to generate a waveform
- ▶ Adding some coins and buying
- ▶ You can and shall extend this tester
- ▶ Better having more than one tester
- ▶ Show the waveform of the test

Printf Debugging

- ▶ We can *print* in the hardware during simulation
- ▶ Printing happens on the rising edge of the clock
- ▶ Good to see many parallel signals and registers
- ▶ `printf` anywhere in the module definition

```
class DeviceUnderTestPrintf extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(2.W))
    val b = Input(UInt(2.W))
    val out = Output(UInt(2.W))
  })

  io.out := io.a & io.b
  printf("dut: %d %d %d\n", io.a, io.b, io.out)
}
```

Test Driven Development (TDD)

- ▶ Software development process
  - ▶ Can we learn from SW development for HW design?
- ▶ Writing the test first, then the implementation
- ▶ Started with extreme programming
  - ▶ Frequent releases
  - ▶ Accept change as part of the development
- ▶ A path to *Agile Hardware Development!*
- ▶ Not used in its pour form
  - ▶ Writing all those tests is simply considerer too much work
  - ▶ **But**, write at least one test for each component

Regression Tests

- ▶ Tests are collected over time
- ▶ When a bug is found, a test is written to reproduce this bug
- ▶ Collection of tests increases
- ▶ Runs every night to test for *regression*
  - ▶ Did a code change introduce a bug in the current code base?

Continuous Integration (CI)

- ▶ Next logical step from regression tests
- ▶ Run all tests whenever code is changed
- ▶ Automate this with a repository, e.g., on GitHub
- ▶ Run CI on GitHub
- ▶ Show about this on the Chisel book
  - ▶ Show `sbt test`
  - ▶ Live demo on GitHub
  - ▶ Mail from GitHub when it fails
- ▶ https://github.com/schoeberl/chisel-book/actions
- ▶ Maybe show how to set this up (it is easy ;)
  - ▶ Start with the chisel-empty template
  - ▶ Open it with IntelliJ
  - ▶ Add action in GitHub (but it misses `sbt` setup)

Testing versus Debugging

▶ Debugging is during code development
▶ Waveform and `println` are easy tools for debugging
▶ Debugging does not help for regression tests
▶ Write small test cases for regression tests
▶ Keeps your code base *intact* when doing changes
▶ Better confidence in changes not introducing new bugs

Scala Build Tool (sbt)

- ▶ Downloads Scala compiler if needed
- ▶ Downloads dependent libraries (e.g., Chisel)
- ▶ Compiles Scala programs
- ▶ Executes Scala programs
- ▶ Does a lot of magic, maybe too much
- ▶ Compile and run with:

```
sbt "runMain simple.Example"
sbt run
sbt test
sbt "testOnly MySpec"
sbt compile
```

Build Configuration

- ► File name: build.sbt
- ► Defines needed Scala version
- ► Library dependencies

```
scalaVersion := "2.12.13"

scalacOptions ++= Seq("-feature",
    "-language:reflectiveCalls")

resolvers ++=
    Seq(Resolver.sonatypeRepo("releases"))

addCompilerPlugin("edu.berkeley.cs" %
    "chisel3-plugin" % "3.5.0" cross
    CrossVersion.full)
libraryDependencies += "edu.berkeley.cs" %%
    "chisel3" % "3.5.0"
libraryDependencies += "edu.berkeley.cs" %%
    "chiseltest" % "0.5.0"
```

Today's Lab

- ▶ Testing a faulty multiplexer
- ▶ Do not look into the multiplexer code, find out with testing
- ▶ Use ChiselTest (the description has been updated)
- ▶ You have to start from scratch with the tester
- ▶ Show and discuss your testing code with a TA (or me)
- ▶ Lab 4
- ▶ And an additional challenge brought to you by Tjark

Summary

- ▶ Small sequential circuits are our building blocks
- ▶ We build larger circuits by combining components (modules)
- ▶ There is no *println* in (real) hardware
- ▶ We need to write tests for the development
- ▶ Debugging versus regression tests