Testing and Verification

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

February 25, 2021

Overview

- Thomas Aakjer presents digital design at Microchip
- Review components
- Debugging and testing
- Digital designers call testing verification
  - To distinguish from final chip testing

Some Clarification

- Online learning is hard, I understand
- Installing all tools on your laptop can also be hard
- I take this into consideration on final grading
- Don’t be afraid from the *power user* demos
  - Just for “entertainment” and for advanced users
  - You can also simply ignore it
- Learning Objectives
  - How to build medium sized digital circuits
  - Describe them in Chisel and implement them in an FPGA
  - Timing analysis of digital circuits

Last Lab

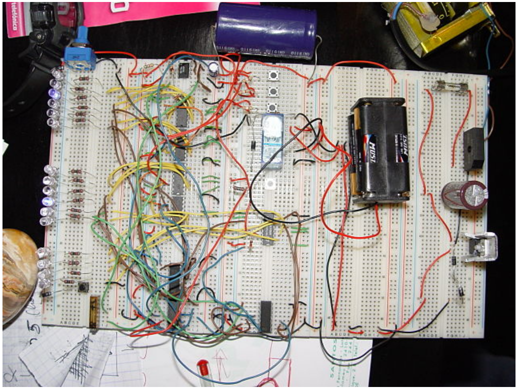
- On components and small sequential circuits
  - Registers plus combinational circuits
- Did you finish the exercises?
  - Do the poll
- They are not mandatory, but helpful for preparation for the final project
- Let’s look at solutions

Components/Modules

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

A Binary Watch

- Built out of discrete, digital components

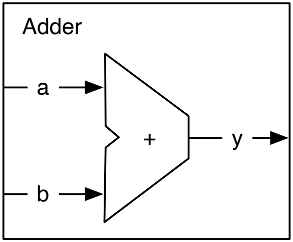


Source: Diogo Sousa, public domain

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

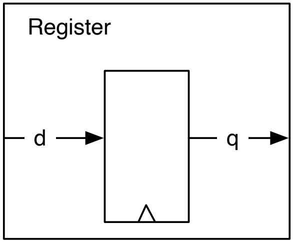
An Adder (Component/Modul)



```
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
}
```

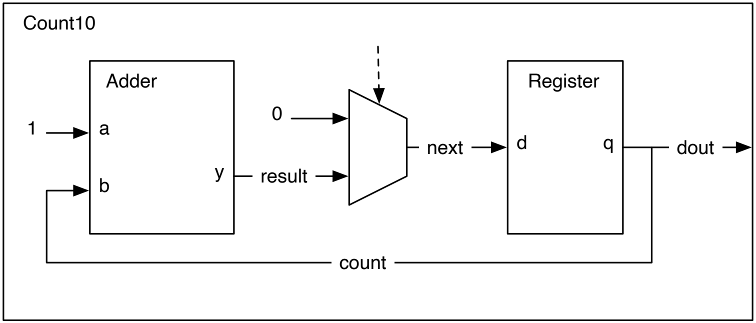
A Register



```
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())

  val count = reg.io.q

  // connect the adder
  add.io.a := 1.U
  add.io.b := count
  val result = add.io.y

  val next = Mux(count === 9.U, 0.U, result)
  reg.io.d := next

  io.dout := count
}
```

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 {
  (new chisel3.stage.ChiselStage).emitVerilog(new
      Hello())
}
```

Testing and Debugging

- Nobody writes perfect code ;-)
- We need a method to improve the code
- In Java we can simply print the result:
  - `println("42");`
- What can we do in hardware?
  - Describe the whole circuit and hope it works?
  - We can switch an LED on or off
- We need some tools for debugging
- Writing testers in Chisel

Testing with Chisel

- Set input values with `poke`
- Advance the simulation with `step`
- Read the output values with `peek`
- Compare the values with `expect`
- Import following packages:

```
import chisel3._
import chisel3.iotesters._
```

Using peek, poke, and expect

```
// Set input values
poke(dut.io.a, 3)
poke(dut.io.b, 4)
// Execute one iteration
step(1)
// Print the result
val res = peek(dut.io.result)
println(res)

// Or compare against expected value
expect(dut.io.result, 7)
```

A Chisel Tester

- Extends class `PeekPokeTester`
- Has the device-under test (DUT) as parameter
- Testing code can use all features of Scala

```
class CounterTester(dut: Counter) extends
    PeekPokeTester(dut) {

  // Here comes the Chisel/Scala code
  // for the testing
}
```

Example DUT

- A device-under test (DUT)

```
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))
  })

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

A Simple Tester

- Just using `println` for manual inspection

```
class TesterSimple(dut: DeviceUnderTest)
    extends PeekPokeTester(dut) {

  poke(dut.io.a, 0.U)
  poke(dut.io.b, 1.U)
  step(1)
  println("Result is: " +
      peek(dut.io.out).toString)
  poke(dut.io.a, 3.U)
  poke(dut.io.b, 2.U)
  step(1)
  println("Result is: " +
      peek(dut.io.out).toString)
}
```

The Main Program for the Test

- Extend an App and invoke the `iotesters` driver
- With the DUT and the tester

```
object TesterSimple extends App {
  chisel3.iotesters.Driver(() => new
      DeviceUnderTest()) { c =>
    new TesterSimple(c)
  }
}
```

A Real Tester

- Poke values and `expect` some output

```
class Tester(dut: DeviceUnderTest) extends
    PeekPokeTester(dut) {

  poke(dut.io.a, 3.U)
  poke(dut.io.b, 1.U)
  step(1)
  expect(dut.io.out, 1)
  poke(dut.io.a, 2.U)
  poke(dut.io.b, 0.U)
  step(1)
  expect(dut.io.out, 0)
}
```

ScalaTest

- Testing framework for Scala
- `sbt` understands ScalaTest
- Run all tests: `sbt test`
- When all `expects` are ok, the test passes
- A little bit funny syntax
- Add library to `build.sbt`

```
libraryDependencies += "org.scalatest" %%
    "scalatest" % "3.0.5" % "test"
```

- Import ScalaTest library

```
import org.scalatest._
```

ScalaTest Version of our Tester

```
class SimpleSpec extends FlatSpec with Matchers {

  "Tester" should "pass" in {
    chisel3.iotesters.Driver(() => new
        DeviceUnderTest()) { c =>
      new Tester(c)
    } should be (true)
  }
}
```

Generating Waveforms

- Waveforms are timing diagrams
- Good to see many parallel signals and registers
- Additional parameters: `"--generate-vcd-output", "on"`
- IO signals and registers are dumped
- Option `--debug` puts all wires into the dump
- Generates a .vcd file
- Viewing with GTKWave or ModelSim

Waveform Testing Demo

- Counter with a limit from last week (`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

```
class Count6Wave(dut: Count6) extends
    PeekPokeTester(dut) {
  step(20)
}
```

Call the Tester

- Using here ScalaTest
- Note `Driver.execute`
- Note `Array("--generate-vcd-output", "on")`

```
class Count6WaveSpec extends
  FlatSpec with Matchers {

  "CountWave6 " should "pass" in {
    chisel3.iotesters.Driver.
    execute(Array("--generate-vcd-output",
        "on"),() => new Count6)
    { c => new Count6Wave(c) }
    should be (true)
  }
}
```

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 working VM

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
- Not used in its pour form
  - Writing all those tests is simply considerer too much work

Regresssion Tests

- But 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 Travis (with GitHub integration)
- Show about this on the Chisel book
  - Show `sbt test`
  - Mails from travis
  - Live demo on travis
- https://travis-ci.com/schoeberl/chisel-book

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

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

```
scalaVersion := "2.12.12"

scalacOptions := Seq("-deprecation",
    "-Xsource:2.11")

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

libraryDependencies += "edu.berkeley.cs" %%
    "chisel-iotesters" % "1.5.1"
// Chisel 3.4.1 is loaded as a dependency on the
    tester
```

Today's Lab

- This is the start of group work
  - Please register your group here
- Binary to 7-segment decoder
- First part of your vending machine
- Just a single digit, only combinational logic
- Use the nice tester provided to develop the circuit
- Then synthesize it for the FPGA
- Test with switches
- Show a TA your working design
- Lab 4

Summary

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