Testing and Verification

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

March 2, 2021

Overview

- Synopsis and Syosil presentation on verification
- Review components
- Debugging and testing
- Digital designers call testing verification
  - To distinguish from final chip testing

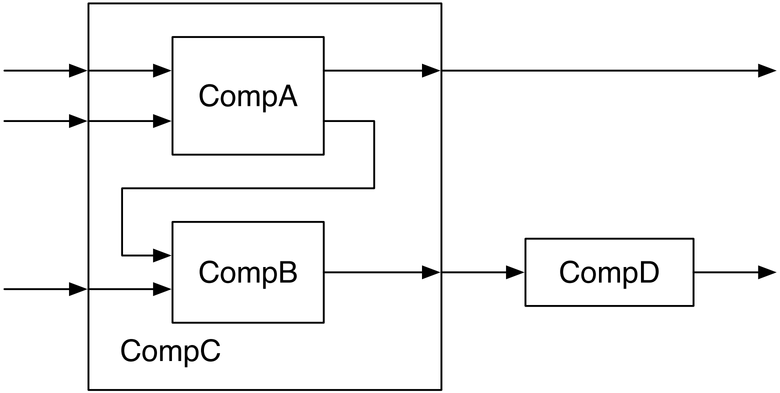
Last Lab

- On components and small sequential circuits
  - Registers plus combinational circuits
- Did you finish the exercises?
- 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 3 2:1 mulitplexers
- In Chisel a component is called `Module`
- Components/Modules are used to organize the circuit
  - Similar as using methods in Java

Hierarchy of Components Example



Ports

- Ports are the connections of modules
- Ports are bundles with a direction
- The direction be reversed with `Flipped`

```
class Channel extends Bundle {
  val data = Input(UInt(32.W))
  val ready = Output(Bool())
  val valid = Input(Bool())
}

class ChannelUsage extends Bundle {
  val input = new Channel()
  val output = Flipped(new Channel())
}
```

An Adder Module

- A `class` that `extends` `Module`
- Interface (port) is a `Bundle`, wrapped into an `IO()`, and stored in the field `io`
- Circuit description in the constructor

```scala
class Adder extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(4.W))
    val b = Input(UInt(4.W))
    val result = Output(UInt(4.W))
  })

  val addVal = io.a + io.b
  io.result := addVal
}
```

Connections

- Simple connections just with assignments, e.g.,

```
adder.io.a := ina
adder.io.b := inb
```

- Automatic bulk connections between components

```
dec.io <> exe.io
mem.io <> exe.io
```

Module Usage

- Create with `new` and wrap into a `Module()`
- Interface port via the `io` field
- Note the assignment operator `:=` on `io` fields

```
val adder = Module(new Adder())
adder.io.a := ina
adder.io.b := inb
val result = adder.io.result
```

Chisel Main

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

```
object Hello extends App {
  chisel3.Driver.execute(Array[String](), () =>
      new Hello())
}
```

Midterm Evaluation

- An anonymous Google form (no login required)
- 20 minutes time, including the break
- We will look into it after the break

- https://forms.gle/wtvDrA4peD4oLvt16

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:

```scala
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 := "2.11.7"

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

libraryDependencies += "edu.berkeley.cs" %%
    "chisel3" % "3.1.2"
libraryDependencies += "edu.berkeley.cs" %%
    "chisel-iotesters" % "1.2.2"
```

Today’s Lab

- 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 5

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