Experiment 4

Chisel Testers

Objective

Be familiar with the Chisel testing and learn how to write and run tests to verify proper functioning of the device-under-test (DUT).

4.1 Testing in Chisel

As is the case with almost all hardware description languages, at the end of the day, we need to verify for proper working of the hardware that has been designed. *Chisel* based hardware design is no exception. The strength of testing in *Chisel* lies in the fact that we can use all the features available in Scala to write tests.

4.1.1 Chisel Tester

Most common tool, that is used for testing in Chisel, is the iotesters. The iotesters supports following three different harnesses for testing a DUT.

1. PeekPokeTester
2. SteppedHWIOTester
3. OrderedDecoupledHWIOTester

Among the three different testing harnesses available, we will primarily focus on `PeekPokeTester`. Furthermore, there is another tester named *tester2*, which is available for testing. However, it is in the experimental phase and we will briefly touch it in the upcoming labs.

4.2 PeekPokeTester

PeekPokeTester is a mature testing methodology in *Chisel*. The library for PeekPokeTester can be imported using the following command.

```
import chisel3.iotesters.{ChiselFlatSpec, Driver, PeekPokeTester}
```

When performing an IO testing, we need to drive the inputs and check whether the outputs are according to our expectation. For debugging purpose, we can display the driving inputs along with corresponding outputs produced by the DUT on a terminal window. To test sequential circuits, we also need to drive the clock. The PeekPokeTester has four constructs as discussed next.

- peek: As the name suggests, we can lookup the value of an output using the construct peek. Peek returns a literal value. Peek can also be used to get the values of driving inputs. The syntax for peek is:

```
peek(io)
```

- poke: In order to drive an input of the DUT, we poke that input with a permissible value. The syntax for poke is:

```
poke(io, value)
```

- expect: To compare an output produced by the DUT with a literal value or another input/output, we can use expect. The syntax for expect is:

```
expect(io, equal_to)
```

- step: To drive the clock, we have the step construct. This allows us to drive the implicit clock of the module by an arbitrary number of cycles. The syntax for step is:

```
step(n)    // n is no. of cycles and accepts integer values
```

4.2.1 Project Directory Hierarchy

To manage a project involving multiple source files implementing different modules along with corresponding tests, Chisel uses a predefined project directory hierarchy. This structure of project directory hierarchy is shown in Figure 4.1.

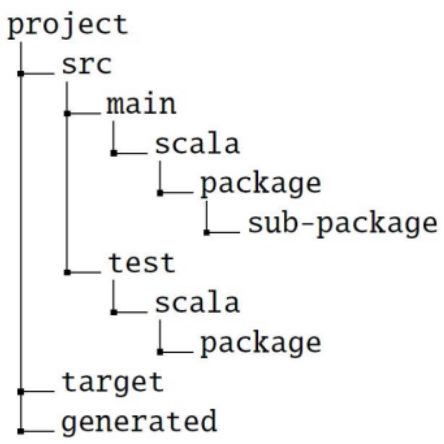


Figure 4.1: Project structure hierarchy.

4.2.2 Writing Test

We will follow learn by example methodology to be able to write tests using PeekPokeTester. For that purpose we have chosen the example of a Mux tree. The two main components of any module testing are the DUT and the Test as discussed below.

- DUT: We select a Mux tree with three inputs and two sections lines as our DUT. The implementation of this Mux is provided in Listing 4.1.

```
package LM_Chisel
import chisel3._

class MuxTreeIO extends Bundle {
   val in_1   = Input(UInt(32.W))
   val in_2   = Input(UInt(32.W))
   val in_3   = Input(UInt(32.W))
   val sel_1  = Input(Bool())
   val sel_2  = Input(Bool())
   val out    = Output(UInt())
}

// 3 to 1 MuxTree implementation
class MuxTree extends Module {
   val io = IO(new MuxTreeIO)

   // update the output
   io.out := Mux(io.sel_2, io.in_3, Mux(io.sel_1, io.in_2, io.in_1))
}
```

Listing 4.1: Mux tree with three inputs.

- Test: Writing a test requires driving of the DUT's inputs and checking the corresponding outputs. In addition, input and output values can also be printed for debugging. Listing 4.2 implements the test.

```
package LM_Chisel
import chisel3._
import chisel3.iotesters.{Driver, PeekPokeTester}

class TestMuxT(c: MuxTree) extends PeekPokeTester(c) {
   val in1  = 0x11111111
   val in2  = 0x22222222
   val in3  = 0x33333333

   poke(c.io.in_1, in1.U)
   poke(c.io.in_2, in2.U)
   poke(c.io.in_3, in3.U)

   poke(c.io.sel_1, false.B)
   poke(c.io.sel_2, false.B)
   expect(c.io.out, in1.U)
   step(1)

   poke(c.io.sel_1, true.B)
   poke(c.io.sel_2, false.B)
```

```
    expect(c.io.out, in2.U)
    step(1)

    poke(c.io.sel_1, true.B)
    poke(c.io.sel_2, true.B)
    expect(c.io.out, in3.U)
    step(3)
}
```

Listing 4.2: Tester for the DUT.

From Listing 4.2 we observe that the class is inherited from `PeekPokeTester`. The parameter passed to the 'TestMuxT' class is of type DUT, i.e. of type 'MuxTree'. It is worth mentioning that the DUT file is placed in "\scr \main \scala", while test file is placed in "\scr \test \scala". Observe that the first line in both the files is package LM_Chisel, which puts both the files in the same package.

4.2.3 Running Test

To run the test, we need to either create a main method in the object or extend the object from the class `App`. Our test initializer is implement by extending an object from the class `App` and is given in Listing 4.3. This test initializer code in appended to the test file.

```
// object for tester class
object MuxT_Main extends App {
   iotesters.Driver.execute(Array("--is-verbose", "--generate-vcd-output",
   "on"), () => new MuxTree) {
      c => new TestMuxT(c)
   }
}
```

Listing 4.3: Initializer for the tester.

In Listing 4.3, observe the use of configuration array to pass user options to configure the tools. For instance, we use `"--is-verbose"` to turn on the verbose mode. We can run this test, using command line or an sbt shell. An sbt shell can be opened by typing sbt in the command terminal and hitting enter. When the sbt shell opens, one can run the test using the following syntax.

```
test:runMain packageName.objectName
```

For our specific example, use the following command.

```
test:runMain LM_Chisel.MuxT_Main
```

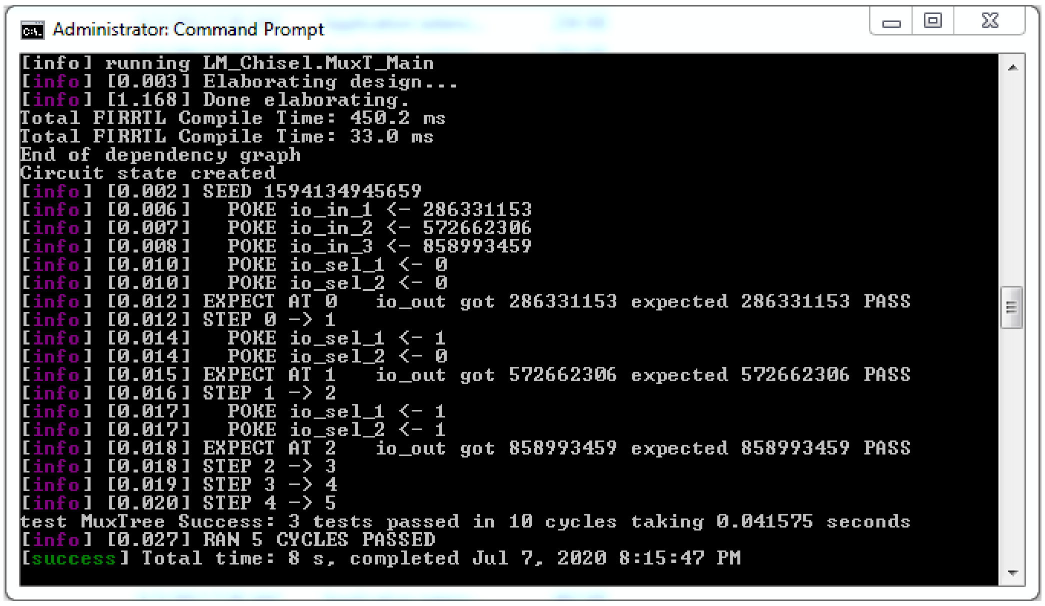
4.2.4 Tools Invoked while Testing

When a test is run by the user command **test:run**, different tools are invoked at different stages as listed below. Further details regarding tools usage can be found in Appendix A.

- At first step the tester (PeekPokeTester in this case) is invoked
- Tester invokes chisel3 to generate the circuit
- The chisel3 invokes firrtl to compile the circuit into low firrtl
- The low firrtl invokes the firrtl-interpreter to execute the test on the DUT

4.2.5 Viewing Output

When the test is run, we can view the driving inputs and corresponding outputs in the terminal window. This detailed information is printed to the terminal (as depicted in Figure 4.2) because the verbose mode was turned on. A .vcd file is also generated due to the configuration options. One can open the .vcd file using a waveform viewer application. Figure 4.3 shows the signals for the 'MuxTree' testing.



```
[info] running LM_Chisel.MuxT_Main
[info] [0.003] Elaborating design...
[info] [1.168] Done elaborating.
Total FIRRTL Compile Time: 450.2 ms
Total FIRRTL Compile Time: 33.0 ms
End of dependency graph
Circuit state created
[info] [0.002] SEED 1594134945659
[info] [0.006]   POKE io_in_1 <- 286331153
[info] [0.007]   POKE io_in_2 <- 572662306
[info] [0.008]   POKE io_in_3 <- 858993459
[info] [0.010]   POKE io_sel_1 <- 0
[info] [0.010]   POKE io_sel_2 <- 0
[info] [0.012] EXPECT AT 0   io_out got 286331153 expected 286331153 PASS
[info] [0.012] STEP 0 -> 1
[info] [0.014]   POKE io_sel_1 <- 1
[info] [0.014]   POKE io_sel_2 <- 0
[info] [0.015] EXPECT AT 1   io_out got 572662306 expected 572662306 PASS
[info] [0.016] STEP 1 -> 2
[info] [0.017]   POKE io_sel_1 <- 1
[info] [0.017]   POKE io_sel_2 <- 1
[info] [0.018] EXPECT AT 2   io_out got 858993459 expected 858993459 PASS
[info] [0.018] STEP 2 -> 3
[info] [0.019] STEP 3 -> 4
[info] [0.020] STEP 4 -> 5
test MuxTree Success: 3 tests passed in 10 cycles taking 0.041575 seconds
[info] [0.027] RAN 5 CYCLES PASSED
[success] Total time: 8 s, completed Jul 7, 2020 8:15:47 PM
```

Figure 4.2: Command line output in verbose mode.

4.2.6 Tester Configurations

Apart from the few configurations that we have used so far, there are many other user configurations available. Some of the useful configurations are listed in Listing 4.4.

```
                        Selected Tester Options
                        -----------------------

-tbn, --backend-name  <firrtl|treadle|verilator|ivl|vcs>
                      backend to use with tester, default is treadle

-tiv, --is-verbose    set verbose flag on PeekPokeTesters, default is false

-twffn, --wave-form-file-name <value>
                      wave form file name

-tts, --test-seed <value>
                      provides a seed for random number generator
```

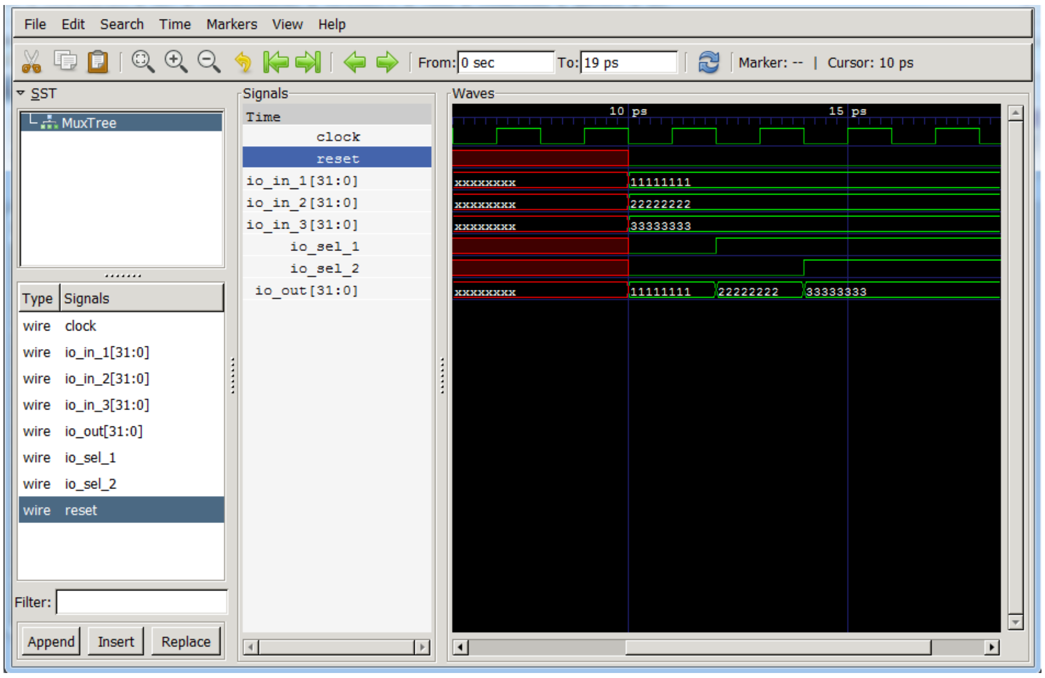


Figure 4.3: Viewing input/output signals using GTKWave.

```
-tgvo, --generate-vcd-output <value>
                          set this flag to "on" or "off", otherwise it defaults to on
                          for verilator, off for scala backends

-td <target-directory>, --target-dir <target-directory>
                          defines a work directory for intermediate files, default is
                          current directory
```

Listing 4.4: Selected user configurations.

Furthermore, to print the Verilog generated in a terminal window, we can simply add the following command to the object extending from the App.

```
println((new chisel3.stage.ChiselStage).emitVerilog(new DUT))
```

To dump the verilog and FIRRTL output, one can use the corresponding options as described below.

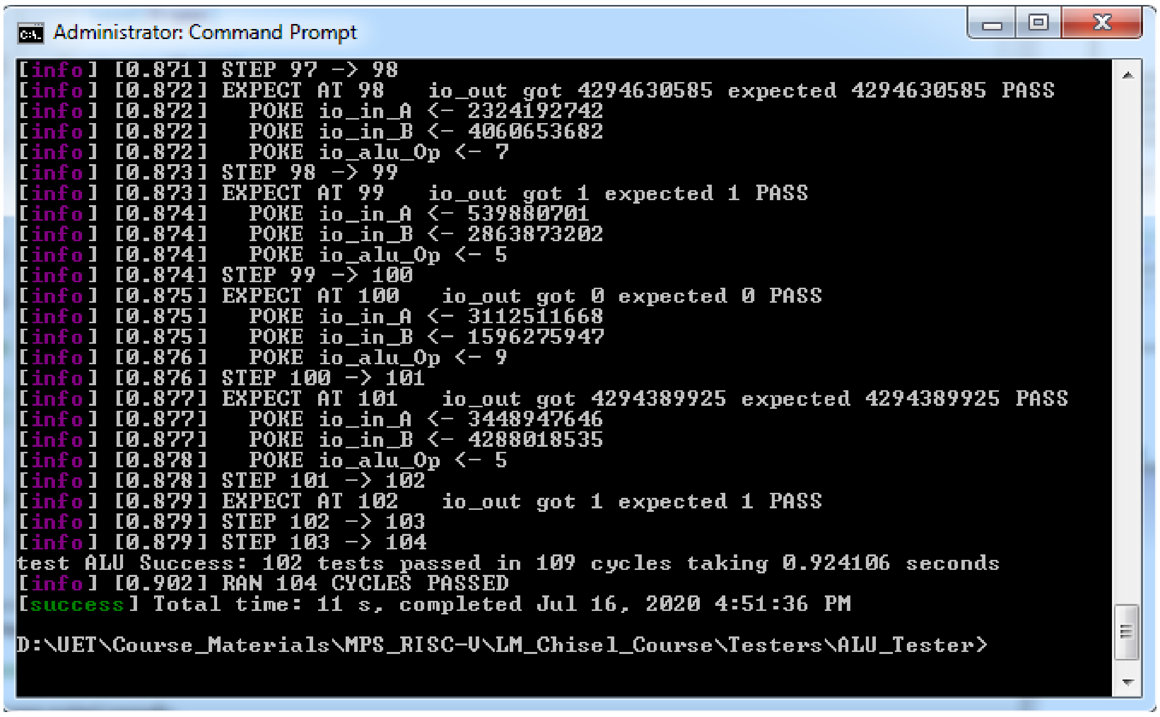
```
chisel3.Driver.execute(Array("--target-dir","RTL_files"), () => new DUT)
```

4.3 ALU Tester

We have already implemented the ALU in Experiment 4. Now we are going to write a test, which will randomly poke different operations implemented by the ALU and will verify the results for correctness. Listing 4.5 implements random testing of the ALU and the test results are depicted in Figure 4.4

```
package LM_Chisel
```

```
import chisel3._
import chisel3.util
import chisel3.iotesters.{
    Driver, PeekPokeTester
}
import scala.util.Random
import ALUOP._

class TestALU(c: ALU) extends PeekPokeTester(c) {
    // ALU operations
    val array_op = Array(ALU_ADD , ALU_SUB, ALU_AND, ALU_OR, ALU_XOR,ALU_SLT, ALU_SLL
    , ALU_SLTU, ALU_SRL, ALU_SRA, ALU_COPY_A, ALU_COPY_B, ALU_XXX)

    for (i <- 0 until 100) {
        val src_a  = Random.nextLong()& 0xFFFFFFFFL
        val src_b = Random.nextLong()& 0xFFFFFFFFL
        val opr = Random.nextInt(12)
        val aluop  = array_op(opr)

        // ALU functional implementation using Scala match
        val result = aluop match {
            case ALU_ADD  => src_a + src_b
            case ALU_SUB  => src_a - src_b
            case ALU_AND  => src_a & src_b
            case ALU_OR   => src_a | src_b
            case ALU_XOR  => src_a ^ src_b
            case ALU_SLT  => (src_a.toInt < src_b.toInt).toInt
            case ALU_SLL  => src_a  << (src_b & 0x1F)
            case ALU_SLTU => (src_a < src_b).toInt
            case ALU_SRL  => src_a >> (src_b & 0x1F)
            case ALU_SRA  => src_a.toInt >> (src_b & 0x1F)
            case ALU_COPY_A => src_a
            case ALU_COPY_B => src_b
            case _        => 0
        }

        val result1: BigInt = if (result < 0)
        (BigInt(0xFFFFFFFFL)+result+1) & 0xFFFFFFFFL
        else result & 0xFFFFFFFFL

        poke(c.io.in_A, src_a.U)
        poke(c.io.in_B, src_b.U)
        poke(c.io.alu_Op, aluop)
        step(1)
        expect(c.io.out, result1.asUInt)
    }
    step(2)
}

// object for tester class
object ALU_Main extends App {
    iotesters.Driver.execute(Array("--is-verbose", "--generate-vcd-output", "on", "--
     backend-name", "firrtl"), () => new ALU) {
```



```
[info] [0.871] STEP 97 -> 98
[info] [0.872] EXPECT AT 98   io_out got 4294630585 expected 4294630585 PASS
[info] [0.872]   POKE io_in_A <- 2324192742
[info] [0.872]   POKE io_in_B <- 4060653682
[info] [0.872]   POKE io_alu_Op <- 7
[info] [0.873] STEP 98 -> 99
[info] [0.873] EXPECT AT 99   io_out got 1 expected 1 PASS
[info] [0.874]   POKE io_in_A <- 539880701
[info] [0.874]   POKE io_in_B <- 2863873202
[info] [0.874]   POKE io_alu_Op <- 5
[info] [0.874] STEP 99 -> 100
[info] [0.875] EXPECT AT 100   io_out got 0 expected 0 PASS
[info] [0.875]   POKE io_in_A <- 3112511668
[info] [0.875]   POKE io_in_B <- 1596275947
[info] [0.876]   POKE io_alu_Op <- 9
[info] [0.876] STEP 100 -> 101
[info] [0.877] EXPECT AT 101   io_out got 4294389925 expected 4294389925 PASS
[info] [0.877]   POKE io_in_A <- 3448947646
[info] [0.877]   POKE io_in_B <- 4288018535
[info] [0.878]   POKE io_alu_Op <- 5
[info] [0.878] STEP 101 -> 102
[info] [0.879] EXPECT AT 102   io_out got 1 expected 1 PASS
[info] [0.879] STEP 102 -> 103
[info] [0.879] STEP 103 -> 104
test ALU Success: 102 tests passed in 109 cycles taking 0.924106 seconds
[info] [0.902] RAN 104 CYCLES PASSED
[success] Total time: 11 s, completed Jul 16, 2020 4:51:36 PM

D:\UET\Course_Materials\MPS_RISC-V\LM_Chisel_Course\Testers\ALU_Tester>
```

Figure 4.4: ALU test results.

```
        c => new TestALU(c)
    }
}
```

Listing 4.5: ALU tester implementation.

4.4 Exercises

Exercise 1: Extend the ALU tester to test operations that are not supported by the ALU.

4.5 Assignments

Task 1: Write the test for Branch module.

Task 2: Write the test for Immediate generation module.

Task 3: You are provided with a code of a bugged ALU, test it and figure out the bug/s.