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 Testers 2

ChiselTest is a test harness for Chisel-based RTL designs, currently supporting directed testing. ChiselTest emphasizes tests that are lightweight (minimizes boilerplate code), easy to read and write (understandability) and compose (for better test code reuse).

The core primitives are similar to non synthesizable Verilog: input pin assignment (poke), pin value assertion (expect), and time advance (step). Threading concurrency is also supported with the use of fork and join, and concurrent accesses to wires are checked to prevent race conditions.

4.2 Writing a Test

ChiselTest integrates with the ScalaTest framework, which provides a framework for detection and execution of unit tests.

Assuming a typical Chisel project with MyModule defined in src/main/scala/MyModule.scala:

```
class MyModule extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(16.W))
    val out = Output(UInt(16.W))
  })

  io.out := RegNext(io.in)
}
```

Create a new file in src/test/scala/, for example, BasicTest.scala.
In this file:

1. Add the necessary imports:

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

2. Create a test class:

```
class BasicTest extends FlatSpec with ChiselScalatestTester with Matchers {
  behavior of "MyModule"
  // test class body here
}
```

   - FlatSpec is the default and recommended ScalaTest style for unit testing.
   - ChiselScalatestTester provides test-driver functionality and integration (like signal value assertions) within the context of a ScalaTest environment.
   - Matchers provide additional syntax options for writing ScalaTest tests. Potentially optional, since it's mainly for Scala-land assertions and does not inter-operate with circuit operations.

3. In the test class, define a test case:

```
it should "do something" in {
  // test case body here
}
```

   - There can be multiple test cases per test class, and we recommend one test class per Module being tested, and one test case per individual test.

4. In the test case, define the module being tested:

```
test(new MyModule) { c =>
  // test body here
}
```

   - test automatically runs the default simulator (which is treadle), and runs the test stimulus in the block. The argument to the test stimulus block (c in this case) is a handle to the module under test.

5. In the test body, use poke, step, and expect operations to write the test:

```
c.io.in.poke(0.U)
c.clock.step()
c.io.out.expect(0.U)
c.io.in.poke(42.U)
c.clock.step()
c.io.out.expect(42.U)
println("Last output value :" + c.io.out.peek().litValue)
```

```
```

6. With your test case complete, you can run all the test cases in your project by invoking ScalaTest. If you're using sbt, you can either run sbt test from the command line or test from the sbt console. testing can also be used to run specific tests.

ˆ 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:

```
c . io . out . peek ( )
```

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

```
c . io . in . poke ( value )
```

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

```
c . io . out . expect ( equal_to )
```

ˆ 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:

```
c . clock . ,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 corre sponding tests, Chisel uses a predefined project directory hierarchy. This structure of project directory hierarchy is shown in Figure 4.1.

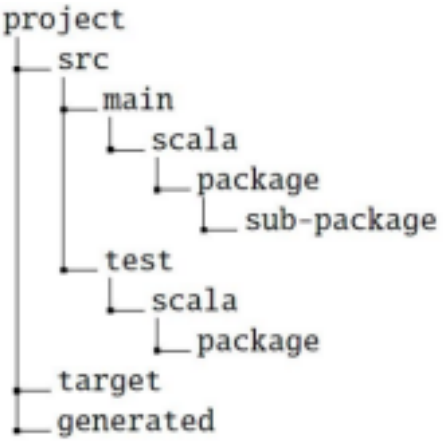

Figure 4.1: Project structure hierarchy.

4.2.3 Running Test

We can run the 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.

```
testOnly packageName . objectName
```

4.2.5 Viewing Output

We can view the output in the .vcd file that will be generated by using the command.

```
testOnly packageName . objectName  --  - DwriteVcd = 1
```

In order to use Verilator as the simulator add this line in your test class.

```
test(new MyModule).withAnnotations(Seq(VerilatorBackendAnnotation)) { c =>
  // test body here
}
```

Then run the command for generating VCD.

```
testOnly packageName . objectName  --  - DwriteVcd = 1
```

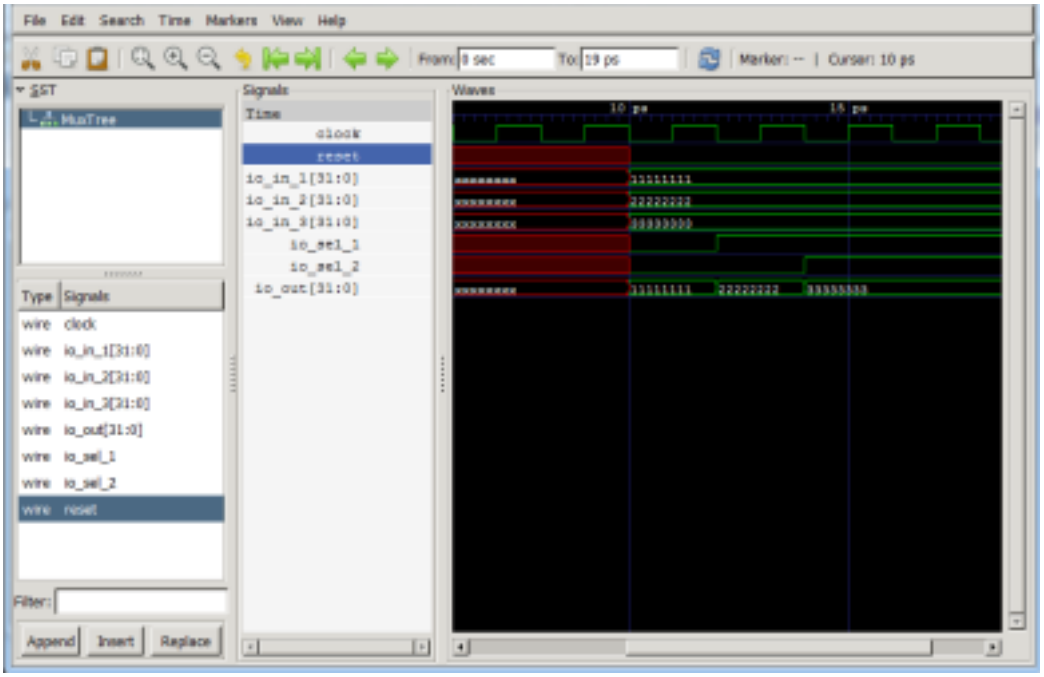

Figure 4.3: Viewing input/output signals using GTKWave.

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 org . scalatest . _
import chiseltest . _
import chiseltest . experimental . TestOptionBuilder . _
import chiseltest . internal . VerilatorBackendAnnotation


import scala . util . Random
import ALUOP . _

class TestALU ( c : ALU ) extends FreeSpec with ChiselScalaTester {
      “ALU Test” in {
            test ( new ALU ) . withAnnotations( Seq ( VerilatorBackendAnnotation ) ) { 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 () & 0 xFFFFFFFFL
                     val src_b = Random . nextLong () & 0 xFFFFFFFFL
                     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 & 0 x1F )
                            case ALU_SLTU = > ( src_a < src_b ) . toInt
                            case ALU_SRL = > src_a > > ( src_b & 0 x1F )
                            case ALU_SRA = > src_a . toInt > > ( src_b & 0 x1F )
                            case ALU_COPY_A = > src_a
                            case ALU_COPY_B = > src_b
                            case _ = > 0
                    }

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

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


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.