实验七：ChiselTest 介绍与使用

1. 实验目的

1.1. 学习了解 ChiselTest 的相关背景知识，对测试有一个基本的概念与认识；

1.2. 掌握常用的 ChiselTest 测试 API，能够使用 ChiselTest 测试组合逻辑电路与时序逻辑电路；

1.3. 能够使用 ChiselTest 生成 VCD 波形。

2. 实验内容

2.1. ChiselTest 相关背景知识介绍；

2.2. 使用 ChiselTest 测试组合逻辑、时序逻辑电路；

2.3. 使用 ChiselTest 生成 VCD 波形。

3. 实验相关 Chisel 语法介绍

在本节中我们将介绍 ChiselTest 的相关常用 API，主要包括了 peek、poke、expect、step 等，这些 API 会在我们编写 ChiselTest 测试平台的时候经常使用到。需要注意的是在我们介绍的这些 API 中，peek、poke、expect 主要用于具体的端口信号中，而 step 则只能用于 clock 信号中，因为 step 的目的是为了推进 clock 的周期。

peek

peek 方法用于读取 DUT（待测单元，Device Under Test）中信号的当前值。通过 dut.io.signal.peek()，我们可以获取该信号的值。为了更方便地处理信号值，Chisel 提供了 litValue 方法，可以将信号转换为其字面值。如果不使用 litValue，那么我们得到的值是一个 Chisel 的硬件信号值，例如 UInt、SInt 等，这些类型在 Chisel 中用于表示硬件信号，但在 Scala 中直接打印这些硬件信号值可能并不直观。因此，为了在测试中将其打印出来或进行进一步的逻辑处理，我们需要将其转换为 Scala 的字面量（如 Int 或 Long），通过使用 litValue 方法，我们可以轻松地将这些硬件信号值转换为相应的 Scala 类型。例如：

```
1. val currentValue = dut.io.value.peek().litValue
2. println(s"当前信号值为: $currentValue")
```

poke

poke 方法用于设置信号的值。通过 dut.io.signal.poke(value)，我们可以将特定的值写入 DUT 的输入信号中，以模拟不同的输入条件。这个方法在测试中非常重要，因为它允许我们控制

DUT 的输入，并观察其对输出的影响。

在使用 poke 时，我们通常会指定一个值，Chisel 中的值通常是无符号整数（使用 U 后缀表示）。例如：

```
1.  dut.io.input.poke(12.U) // 将输入信号设置为 12
```

expect

expect 方法用于验证 DUT 输出信号是否符合预期。通过 dut.io.<some_signal>.expect(<expected_value>)，我们可以将当前信号的值与期望值进行比较。如果实际输出值与预期值不匹配，测试将失败，并报告错误信息。这个类似我们其他编程语言的 assert 的用法。

```
1.  dut.io.out_value.expect(11.U) // 验证输出信号是否为 11
```

在这个示例中，我们期望 DUT 的输出信号 out_value 在特定输入下为 11。如果实际输出不为 11，ChiselTest 将输出错误信息，指示预期值与实际值之间的差异。这种机制有助于快速定位设计中的问题，并确保设计在各种输入条件下的正确性。

step

step 方法用于控制仿真进程的步进。通过 dut.clock.step(n)，我们可以逐步执行仿真，观察信号在每个时钟周期内的变化。step 方法只能用于时钟信号，因为它的目的是推进时钟周期。下面这个例子中用于推进一个时钟周期，这里的 1 也可以省略，不加参数 step 默认是推进一个时钟周期。

```
1.  dut.clock.step(1) // 进行一个时钟周期的仿真
```

我们也可以执行多个周期的仿真：

```
1.  dut.clock.step(2) // 进行两个时钟周期的仿真
2.  dut.clock.step(100) // 进行一百个时钟周期的仿真
```

通过这种方式，我们能够观察到 DUT 在不同时间点的行为变化，并验证其时序逻辑是否符合设计要求。

4. 实验步骤

4.1. ChiselTest 介绍

在编写完 Chisel 代码后，我们需要进行测试来验证我们所编写的 Chisel 代码的逻辑功能正确性，只有经过验证后并且功能符合预期才能说明我们的设计是可用的。硬件的验证和设计一样重要，

甚至可以说验证更加重要，我们可能会消耗更多的时间在验证上面以确保设计的正确性。硬件验证可以有很多种方式，在现在的工业界上，由于大多是采用 Verilog 来进行硬件的编码，因此验证的很大部分都是在 Verilog 上进行的，或者是采用 Verilog 的超集--SystemVerilog，特别是 SystemVerilog，它提供了更加丰富多样的语法来支撑高质量高效率的验证的进行。而我们采用的是 Chisel 来编写我们的硬件逻辑，因此在验证方式上有所不同，但是最根本地，我们还是能采用 Verilog/SystemVerilog 的那一套来进行验证，因为 Chisel 可以被编译为 Verilog/SystemVerilog，不过在这里我们并不会采用这种方式来进行验证测试，我们将采用 ChiselTest，ChiselTest 是一个用于验证用 Chisel 编写的硬件设计的测试库/验证框架。ChiselTest 提供了一组工具和 API，使开发者能够编写单元测试来验证硬件模块的功能和性能，确保设计符合预期。也就是说 ChiselTest 这一个验证框架允许我们采用 Chisel 来进行验证，这就能够维持设计和验证的一致性，而不用先将 Chisel 转化为 Verilog。我们在之前的实验中都是采用 ChiselTest 来展示各种模块的运行与测试效果，这一节实验中我们将对 ChiselTest 进行具体的介绍。

4.2.简单的 Chisel 测试代码

组合逻辑例子

接下来我们将通过下面这个一个 SimpleAdder 例子来介绍 ChiselTest 的基本使用方式。需要强调的是我们下面给出的这个例子是一个使用 ChiselTest 测试组合逻辑的例子，组合逻辑的特点是其输出仅依赖于当前的输入信号，而不依赖于过去的状态。这使得组合逻辑的测试相对简单，因为我们可以直接对输入进行设置，并验证输出是否符合预期。

src/main/scala/exp7/SimpleAdder.scala

```
1.  package exp7
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class SimpleAdder extends Module {
7.    val io = IO(new Bundle {
8.      val a = Input(UInt(4.W))
9.      val b = Input(UInt(4.W))
10.     val sum = Output(UInt(4.W))
11.   })
12.
13.   io.sum := io.a + io.b
14. }
```

我们的测试代码通常是放在 src/test/scala 里面，这个目录里面的每一个子目录对应的是一个 package，例如本次实验的测试代码会放在 src/test/scala/exp7 中，其中的 exp7 就是 package 名，

Scala 的工程管理通常都是按照这种方式来组织的，这在实验二中已经进行了介绍。接下来我们就可以编写一个基于 ChiselTest 的简单测试的代码（通常也叫 testbench）：

src/test/scala/exp7/TestSimpleAdder.scala

```
package exp7

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

class TestSimpleAdder extends AnyFlatSpec with ChiselScalatestTester {
  "SimpleAdder" should "correctly add two numbers" in {
    test(new SimpleAdder) { c =>
      // Set inputs
      c.io.a.poke(3.U)
      c.io.b.poke(1.U)

      // Step the clock
      // c.clock.step(1)

      // Check the output
      c.io.sum.expect(4.U)

      // Peek at the output
      val result = c.io.sum.peek().litValue
      println(s"Result of addition: $result")

      // Another test case
      c.io.a.poke(7.U)
      c.io.b.poke(5.U)

      // c.clock.step(1)

      // This will fail as the result is 12 (0b1100) and we expect 2 (0b0010)
      c.io.sum.expect(2.U) // This will throw an error
    }
  }
}
```

执行下面的命令就可以运行这个测试：

```
mill MyChiselProject.test.testOnly exp7.TestSimpleAdder
```

得到如下的运行结果说明运行测试是成功的：

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp7.TestSimpleAdder
[50/83] MyChiselProject.compile
[info] compiling 2 Scala sources to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp7.TestSimpleAdder
[76/83] MyChiselProject.test.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestSimpleAdder:
SimpleAdder
Result of addition: 4
- should correctly add two numbers *** FAILED ***
  io_sum=12 (0xc) did not equal expected=2 (0x2) (lines in TestSimpleAdder.scala: 31, 9) (TestSimpleAdder.scala:31)
1 targets failed
MyChiselProject.test.testOnly 1 tests failed:
  exp7.TestSimpleAdder SimpleAdder should correctly add two numbers
```

上面代码中，第 3~5 行中，用于导入 Chisel 和 ChiselTest 库以及 ScalaTest 的基础类。

在第 7 行中，定义一个测试类 TestSimpleAdder，继承 AnyFlatSpec 和 ChiselScalatestTester，这两个集成是必备的，其中的 ChiselScalatestTester 用于引入 ChiselTest 中的各种测试 API（peek、poke 等）；

第 8 行是 Scalatest 提供的一个测试的语法，可以用书面的方式描述当前要测试的功能特性等；

第 9 行则是创建了一个我们需要进行测试的模块，并且将其赋予了一个别名 c；

第 11~12 行，使用到了一个 ChiselTest 提供的测试 API：poke，poke 方法用于对信号进行输入赋值，在这里用 poke 将输入的 a 信号赋值为 3，将输入的 b 信号赋值为 1；

第 15 行中，对 clock 使用了 step 方法，step 方法用于前进时钟一个或多个周期，使电路状态更新，需要注意的是 clock 是由于 SimpleAdder 在继承后 Module 后自带的一个信号，对于 clock 信号才能使用 step 方法来推进始终周期，但是请注意我们的设计是一个组合逻辑设计，实际上并不需要推进时钟周期就能使得电路状态更新，因此我们也将这里写了一个注释将其注释掉，这个 step 只会在时序逻辑设计中起作用，第 28 行同理；

第 18 行中，使用了 expect 方法检查 sum 是否等于预期的 4，如果这个值与实际的信号输出不符就会报错，例如在第 31 行中，由于在第 25~26 行对 a 赋值为 7，b 赋值为 5，因此此时 SimpleAdder 的 sum 输出值应该为 12，但是此时我们期望的值是 2，显然两者是不符合的因此会报错，如下图所示：

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp7.TestSimpleAdder
[50/83] MyChiselProject.compile
[info] compiling 2 Scala sources to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp7.TestSimpleAdder
[76/83] MyChiselProject.test.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestSimpleAdder:
SimpleAdder
Result of addition: 4
- should correctly add two numbers *** FAILED ***
  io_sum=12 (0xc) did not equal expected=2 (0x2) (lines in TestSimpleAdder.scala: 31, 9) (TestSimpleAdder.scala:31)
1 targets failed
MyChiselProject.test.testOnly 1 tests failed:
  exp7.TestSimpleAdder SimpleAdder should correctly add two numbers
```

第 21~22 行中，使用到了 peek 方法，peek 方法可以获得一个信号的值，这里则是使用 peek 方法读取 sum 的值，并打印到控制台，可以看到这里在使用了 peek 后还跟着一个 litValue，

这是因为 peek 得到的信号值是一个 Chisel 类型的变量类型，例如 UInt、Bool，但是这不是 Scala 的变量类型，使用 litValue 可以将其转化为一个 Scala 的类型，例如 Int，这样才能在 22 行中使用 println 打印到命令行。

时序逻辑例子

除了组合逻辑设计的例子，我们下面再提供一个时序逻辑的 ChiselTest 例子，下面这个模块是一个简单的带使能端的 D 触发器，在 en 拉高的时候会将 d 的值进行锁存，然后通过 q 端口输出。

src/main/scala/exp7/EnableDFF.scala

```
1.  package exp7
2.
3.  import chisel3._
4.
5.  class EnableDFF extends Module {
6.    val io = IO(new Bundle {
7.      val d  = Input(Bool())
8.      val en = Input(Bool())
9.      val q  = Output(Bool())
10.   })
11.
12.   val qReg = RegInit(false.B)
13.
14.   when(io.en) {
15.     qReg := io.d
16.   }
17.
18.   io.q := qReg
19. }
```

上述代码相对应的测试代码放置在 src/test/scala/exp7/TestEnableDFF.scala

```
1.  package exp7
2.
3.  import chisel3._
4.  import chiseltest._
5.  import org.scalatest.freespec.AnyFreeSpec
6.
7.  class TestEnableDFF extends AnyFreeSpec with ChiselScalatestTester {
8.    "EnableDFF should correctly latch data when enabled" in {
9.      test(new EnableDFF) { dut =>
10.       // Initialize signals
11.       dut.io.d.poke(false.B)
12.       dut.io.en.poke(false.B)
```

```
13.         dut.clock.step() // Apply clock edge
14.
15.         // Print initial state
16.         println(s"Initial state: d = ${dut.io.d.peek().litValue}, en = ${dut.io.en.peek().lit
    Value}, q = ${dut.io.q.peek().litValue}")
17.
18.         // Initial state check
19.         dut.io.q.expect(false.B)
20.         println(s"Initial q: q = ${dut.io.q.peek().litValue}")
21.
22.         // Apply data with enable = false
23.         dut.io.d.poke(true.B)
24.         dut.clock.step() // Apply clock edge
25.         println(s"After applying d = true with en = false: q = ${dut.io.q.peek().litValue}
    ")
26.         dut.io.q.expect(false.B)
27.
28.         // Enable and apply data
29.         dut.io.en.poke(true.B)
30.         dut.clock.step() // Apply clock edge
31.         println(s"After setting en = true: d = ${dut.io.d.peek().litValue}, q = ${dut.io.q.
    peek().litValue}")
32.         dut.io.q.expect(true.B)
33.
34.         // Change data with enable = true
35.         dut.io.d.poke(false.B)
36.         dut.clock.step() // Apply clock edge
37.         println(s"After applying d = false with en = true: q = ${dut.io.q.peek().litValue}
    ")
38.         dut.io.q.expect(false.B)
39.
40.         // Disable and change data
41.         dut.io.en.poke(false.B)
42.         dut.io.d.poke(true.B)
43.         dut.clock.step() // Apply clock edge
44.         println(s"After disabling en and applying d = true: q = ${dut.io.q.peek().litValue}
    ")
45.         dut.io.q.expect(false.B)
46.
47.         // Enable again and check latching
48.         dut.io.en.poke(true.B)
49.         dut.clock.step() // Apply clock edge
50.         println(s"After enabling en again: d = ${dut.io.d.peek().litValue}, q = ${dut.io.q.
    peek().litValue}")
```

```
51.           dut.io.q.expect(true.B)
52.       }
53.     }
54. }
```

第 11~12 行中，对两个输入信号进行赋值，赋值了两个 false，也就是逻辑 0，之后在第 13 行使用 clock.step 更新状态，然后在第 16 行中将当前的值打印出来；

在第 23 行中我们往 d 输入了一个 true，但是还没有给 en 输入 true，然后在第 24 行中进行 clock.step 推进时钟周期更新状态，之后我们得到的 q 应该是 false，因为 en 是 false 的；

到了第 29 行，我们将 en 赋值为 true，同时保持 d 也是 true，然后 30 行的地方推进时钟周期，由于 en 是 true，满足 D 触发器的锁存条件，因此在 31 行得到的将是一个 true，接下来的代码以此类推进行分析。

执行下面的命令就可以运行这个测试：

```
mill MyChiselProject.test.testOnly exp7.TestEnableDFF
```

```
lc3@lc3-virtual-machine:~/chisel_exp$ mill MyChiselProject.test.testOnly exp7.TestEnableDFF
[50/83] MyChiselProject.compile
[info] compiling 1 Scala source to /home/lc3/chisel_exp/out/MyChiselProject/compile.dest/classes ...
[info] done compiling
[76/83] MyChiselProject.test.compile
[info] compiling 2 Scala sources to /home/lc3/chisel_exp/out/MyChiselProject/test/compile.dest/classes ...
[info] done compiling
[83/83] MyChiselProject.test.testOnly
TestEnableDFF:
Initial state: d = 0, en = 0, q = 0
Initial q: q = 0
After applying d = true with en = false: q = 0
After setting en = true: d = 1, q = 1
After applying d = false with en = true: q = 0
After disabling en and applying d = true: q = 0
After enabling en again: d = 1, q = 1
- EnableDFF should correctly latch data when enabled
```

测试结果也符合我们的预期，通过这个例子我们也了解学习了时序逻辑下的 ChiselTest 测试方法。

我们还可以在 ChiselTest 中结合 Scala 的一些循环语法例如 for 循环来使我们的测试代码更加灵活精简，考虑下面这个例子：

src/test/scala/exp7/TestEnableDFF_1.scala

```
1.  package exp7
2.
3.  import chisel3._
4.  import chiseltest._
5.  import org.scalatest.freespec.AnyFreeSpec
6.  import scala.util.Random
7.
8.  class TestEnableDFF_1 extends AnyFreeSpec with ChiselScalatestTester {
9.    "EnableDFF should correctly latch data when enabled" in {
10.     test(new EnableDFF) { dut =>
11.       // Initialize random number generator
```

```
12.         val rand = new Random()
13.
14.         // Loop through a fixed number of test cases
15.         for (i <- 0 until 10) {
16.           // Randomly assign d and toggle en
17.           val dValue = rand.nextBoolean() // Random boolean for d
18.           val enValue = if (i % 2 == 0) true.B else false.B // Toggle en every iteratio
    n
19.
20.           // Set d and en values
21.           dut.io.d.poke(dValue.B)
22.           dut.io.en.poke(enValue)
23.           dut.clock.step() // Apply clock edge
24.
25.           // Print current state
26.           println(s"Test $i: d = $dValue, en = $enValue, q = ${dut.io.q.peek().litValue}")
27.
28.           // Expectation logic based on enable state
29.           if (enValue.litValue == 1) {
30.             // When enabled, q should follow d
31.             dut.io.q.expect(dValue.B)
32.           } else {
33.             println(s"Expecting q to retain previous value (not checked in this simple t
    est)")
34.           }
35.         }
36.       }
37.     }
38. }
```

我们这里使用到了 Scala 的 Random 库，需要使用 import scala.util.Random 来导入，并且在第 12 行中需要使用使用 new Random() 来创建一个随机数生成器，在第 17 行中就能使用随机数生成器的 nextBoolean()生成一个 Scala Boolean 类型的随机数，也就是 true 或者 false，同理也存在一个 nextInt()函数来生成随机的 Scala Int 类型数据。

在第 15 行中我们采用了一个 for 循环来循环 10 个周期进行测试，这里的 0 until 10 也就是对应的 0~9 这 10 个整数，在第 17 行中生成随机的输入后（Boolean 类型），我们在 18 行根据此时的 for 循环 index 值来生成 en 的值，之后在 21、22 行将这些生成的数据使用 poke 方法输入到我们的 DUT 中，从而完成一次信号的输入，在 23 行调用 step 来推进仿真周期。

第 26 行使用 println 方法将端口信号值打印出来，请注意这里同样采用了 litValue 方法将 Chisel 的信号值转化为可以在 Scala 进行打印的字面量。

在 29~34 行则是验证正确性的内容，需要根据我们输入的 enValue 来进行判断。

我们通过这个例子来说明了可以采用一些 Scala 的循环语法来简化我们的测试代码，使得整个测试代码更加简洁，可读性强，结合更多的 Scala 语法还能让我们写出更加灵活多变测试、

上面的这几个例子提供的都是一个 ChiselTest 的基本框架，我们大多数的测试都能按照这个框架来进行编写。

至此我们学习了 peek、poke、clock.step、expect 等常用的 ChiselTest API，结合 Scala 的语法以及这些基本的 ChiselTest API 能够提供灵活高效的验证方式。

4.3.使用 ChiselTest 生成 VCD 波形

我们在实验六中学习了 VCD 波形的相关内容，还学习了 gtkwave 的使用，在其中的一些测试中，我们使用 ChiselTest 生成了 VCD 波形，在这里我们正式介绍 ChiselTest 生成 VCD 波形的方法。

首先我们可以看 exp6 中的一个例子：

src/test/scala/exp6/TestTriangleWave.scala

```
1.  package exp6
2.
3.  import chisel3._
4.  import chiseltest._
5.  import org.scalatest.freespec.AnyFreeSpec
6.
7.  class TestTriangleWave extends AnyFreeSpec with ChiselScalatestTester {
8.    "TestTriangleWave should generate VCD file" in {
9.      val maxCycles = 500
10.
11.     test(new TriangleWave).withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
12.       dut.clock.setTimeout(0) // setting it to 0 means 'no timeout'
13.
14.       for (i <- 0 until maxCycles) {
15.         dut.clock.step()
16.       }
17.     }
18.   }
19. }
```

其中生成 VCD 波形的核心代码在第 11 行，我们添加了 `.withAnnotations(Seq(WriteVcdAnnotation))` 这一部分的内容，在 ChiselTest 中，使用各种 Annotations 来管理各种的附件功能，这里使用到了一个 WriteVcdAnnotation 这一个 Annotation 来让下面的测试能够生成 VCD 波形，执行 `mill MyChiselProject.test.testOnly exp6.TestTriangleWave` 后会在 test_run_dir/TestTriangleWave_should_generate_VCD_file 这个文件夹中生成一个叫 TriangleWave.vcd 这个 VCD 波形文件。

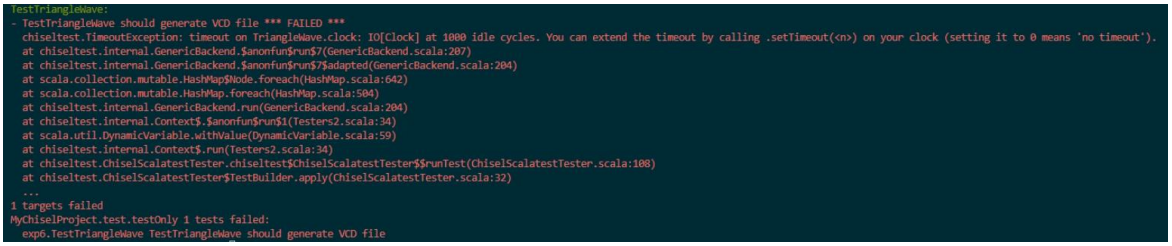
4.4. ChiselTest 的 clock TimeOut 设置

在 ChiselTest 中，设置时钟超时 (clock.setTimeout) 是为了防止测试在意外情况下无限期地运行。这是一个安全措施，确保测试不会因为设计中的某个错误而卡住。例如，可能由于逻辑错误导致某个信号没有在预期时间内改变，导致测试无法完成。通过合理设置超时时间，可以确保测试用例既足够灵活以覆盖设计中的各种情况，又能及时发现和报告异常行为。

在默认情况下，ChiselTest 的 clock TimeOut 阈值为 1000 个 cycle，如果在这 1000 个 cycle 中，没有对待测模块的 IO 口进行任何操作，例如 peek 或者 poke，那么就会超时报错，我们以 exp6 中的 TestTriangleWave 作为例子，对其进行修改得到如下的代码：

```
1.  package exp6
2.
3.  import chisel3._
4.  import chiseltest._
5.  import org.scalatest.freespec.AnyFreeSpec
6.
7.  class TestTriangleWave extends AnyFreeSpec with ChiselScalatestTester {
8.    "TestTriangleWave should generate VCD file" in {
9.      val maxCycles = 5000
10.
11.     test(new TriangleWave).withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
12.       // dut.clock.setTimeout(0) // setting it to 0 means 'no timeout'
13.
14.       for (i <- 0 until maxCycles) {
15.         dut.clock.step()
16.       }
17.     }
18.   }
19. }
```

在上面的代码中，我们将运行的时间设置在了 5000 个 cycle，这已经超过了默认 1000 个 cycle 的超时时间，然后在一个循环中（请注意这里我们可以使用一个 Scala 的 for 循环来执行多个 clock.step，这样能够减少我们的重复代码量，你也可以结合其他的 ChiselTest API，例如在循环中使用 peek 取出信号值并打印出来），只执行 clock.step()而不进行其他操作，那么我们将得到下面的报错：



这里提示我们 clock 超时了，如果想要设置超时时间，可以使用 clock.setTimeout 来实现，并且如果 setTimeout 的值设置为 0 就表示没有 timeout，在我们这个测试场景下，我们只是要生成波形，不需要对信号进行操作，因此就可以使用 clock.setTimeout 将这个 timeout 设置为无限，所以在上面的代码中我们只需要将第 12 行的注释去掉即可。

设置 timeout 的这个操作在后续如果需要使用到 ChiselTest 生成波形的时候可能会使用到，因此这里就单独介绍 timeout 的设置方法。

任务一：

使用 ChiselTest 编写一个测试下面的一个简单 Counter 模块，模块的功能是一个 4bit 的计数器，在 io.enable 有效的时候才会开始计数，测试需要满足以下的要求：

1. 编写的测试需要能够反映这个模块的功能特性，例如 enable 有效的时候才计数；
2. 打印每个 clock 周期的计数值；
3. 需要使用 expect 语法检查正确性；
4. 能够生成 VCD 波形，并用 gtkwave 打开查看这个波形。
5. 你的测试文件可以创建在 src/test/scala/exp7/todo 文件夹中，取名为 TestCounter4.scala。

src/main/scala/exp7/todo/Counter4.scala

```
1.  package exp7.todo
2.
3.  import chisel3._
4.  import chisel3.util._
5.
6.  class Counter4 extends Module {
7.    val io = IO(new Bundle {
8.      val enable = Input(Bool())
9.      val count = Output(UInt(4.W))
10.   })
11.
12.   val counter = RegInit(0.U(4.W))
13.
14.   when(io.enable) {
15.     counter := counter + 1.U
```

```
  }

  io.count := counter
}
```

你的测试应该包括下面的内容：

```
package exp7.todo

import chisel3._
import chiseltest._
import org.scalatest.freespec.AnyFreeSpec

class TestCounter4 extends AnyFreeSpec with ChiselScalatestTester {
  "Counter4 should count only when enabled" in {
    // 下面这里使用 WriteVcdAnnotation 开启了 VCD 波形
    // VCD 波形将会输出到 test_run_dir/Counter4_should_count_only_when_enabled/Counter4.vcd 这个文件中
    test(new Counter4).withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
      // 初始状态
      dut.io.enable.poke(false.B)
      dut.clock.step(1) // 第一个时钟周期
      println(s"Count after 1 clock: ${dut.io.count.peek().litValue}")
      dut.io.count.expect(0.U)

      // 使能计数
      dut.io.enable.poke(true.B)

      /**
       * TODO: 计数几次，使用 println 将 count 的值打印出来。
       *       为了观察到计数超时复位的情况，你可能需要
       *       使用 for 循环 step 并打印 15 次, 例如 for(i <- 0 until 15),
       *       接下来再次 step 应该会观察到数值变为了 0
       */

      // 你可以将 enable 赋值为 false.B 再试试看

    }
  }
}
```

测试编写完成后，测试命令为：

```
mill MyChiselProject.test.testOnly exp7.todo.TestCounter4
```