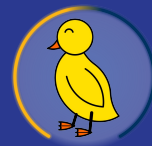


Testers2

towards easy, friendly, fun testing!

CCCC 19

Richard Lin
@ducky64



This talk focuses on
directed testing strategies

Fundamentally, think **non-synthesizable Verilog**

Assign

- Write (assign) the value of a wire

Delay

- Advances time

Assert

- Assert wire value equal to some reference

PeekPokeTesters currently supports these

Assign => **Poke**

- Write (assign) the value of a wire

Delay => **Step**

- Advances time, by a clock cycle

Assert => **Expect**

- Assert wire value equal to some reference

Peek

- Returns the value on a wire as a Scala numeric type

ScalaTest automates test invocation

ScalaTest

- Unit testing framework for Scala
- Automated test discovery and execution with just `sbt run`

```
[info] - should work with FixedPoint
[info] [0.001] Done elaborating.
[info] - should detect combinational-dependent operations when a poke is active
[info] - should detect combinational-dependent operations through internal modules
[info] - should detect combinational paths across operations
[info] ThreadJoinTest:
[info] Testers2 threading fork-joins
[info] - should join a single thread
[info] - should join multiple threads of uneven length, order 1
[info] - should join multiple threads of uneven length, order 2
[info] ExceptionPropagationTest:
[info] Testers2
[info] - should propagate exceptions in the main test thread
Total FIRRTL Compile Time: 2.4 ms
[info] - should propagate exceptions in a forked thread
[info] ClockDividerTest:
[info] Testers2 with a clock divider
[info] - should test 1:2 clock divider counter
Total FIRRTL Compile Time: 2.8 ms
file loaded in 0.003469435 seconds, 3 symbols, 0 statements
Unable to guess top-level testdriver filename from stack trace
[info] ThreadSafetyTest:
[info] Testers2 thread safety checker
[info] - should disallow simultaneous pokes from parallel threads
[info] - should disallow simultaneous peeks and pokes from parallel threads
[info] - should allow overriding pokes from child -> parent thread
[info] - should allow combinational checks from child -> parent thread
[info] - should allow pipelined pokes in child threads
[info] - should allow pipelined pokes in parent and child threads
[info] - should count signal overriding by thread spawn location
[info] - should require overriding pokes be strictly contained
[info] - should contain forks within the calling thread
[info] - should disallow peeks and pokes from parallel threads, when poking at the end of a poke
[info] ScalaTest
[info] Run completed in 1 second, 940 milliseconds.
[info] Total number of tests run: 35
[info] Suites: completed 11, aborted 0
[info] Tests: succeeded 34, failed 1, canceled 0, ignored 0, pending 0
[info] *** 1 TEST FAILED ***
[error] Failed: Total 35, Failed 1, Errors 0, Passed 34
[error] Failed tests:
[error]   chisel3.tests.TimescopeTest
[error] (Test / test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 4 s, completed Nov 8, 2018 9:04:58 PM
[master] chisel-testers2:
```

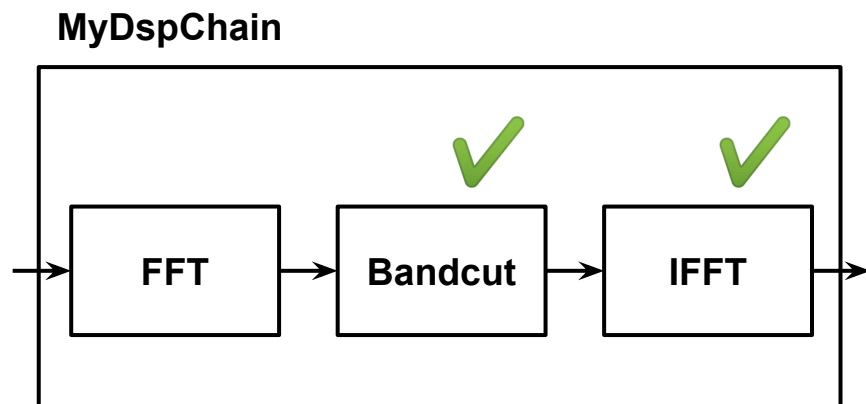
www.exe [64]-48908

+ 180506[64] 1/2 (+) NUM PRI: 119x63 (27,1289) 25V 48256 100%

PeekPokeTesters encourages unit testing

Unit testing is beneficial

- Good localization power
- Regressions testing
- Continuous integration
- Documentation



... but PeekPokeTesters doesn't scale (well)

Need higher-level abstractions for system tests

- a step above peek and poke
- enqueue / dequeue
- stream abstractions

UVM provides for re-use

Need higher-level abstractions for system tests

- a step above peek and poke
- enqueue / dequeue
- stream abstractions

```
class simpleadder_transaction
    extends uvm_sequence_item;
    rand bit[1:0] ina;
    rand bit[1:0] inb;
    bit[2:0] out;

    function new(string name = "");
        super.new(name);
    endfunction: new

    `uvm_object_utils_begin(
        simpleadder_transaction)
    `uvm_field_int(ina, UVM_ALL_ON)
    ...
    `uvm_object_utils_end
endclass: simpleadder_transaction
```

<https://colorlesscube.com/uvm-guide-for-beginners/> (ch4)

UVM provides for re-use (but not well)

Need higher-level abstractions for system tests

- a step above peek and poke
- enqueue / dequeue
- stream abstractions

```
class simpleadder_transaction
    extends uvm_sequence_item;
    rand bit[1:0] ina;
    rand bit[1:0] inb;
    bit[2:0] out;

    function new(string name = "");
        super.new(name);
    endfunction: new

    `uvm_object_utils_begin(
        simpleadder_transaction)
    `uvm_field_int(ina, UVM_ALL_ON)
    ...
    `uvm_object_utils_end
endclass: simpleadder_transaction
```

<https://colorlesscube.com/uvm-guide-for-beginners/> (ch4)

What do we want?

- Automated regressions
- Easy unit testing
- Scalable to system testing

Writing tests is easy

Minimize test invocation boilerplate

- Default `test()` method encapsulates common default configurations
- Test body inline with test invocation

```
"GCD should work" in {  
  test(new Gcd) { c =>  
  
    ... test body here ...  
  
  }  
}
```

Define basic test APIs on fundamental types

- `(data).poke (value)`
- `(data).expect (value)`
- `(clock).step ()`

```
test{new Gcd) { c =>
  c.in.bits.a.poke(15.U)
  c.in.bits.b.poke(6.U)
  c.clock.step(3)
  c.out.valid.expect(true.B)
  c.out.bits.expect(3.U)
}
```

Allow users to define custom abstractions

Custom Bundles can define test helper:

- `(Decoupled).enqueue (data)`
- `(Decoupled).dequeueExpect (data)`
- ... or anything else you want

```
test{new Gcd) { c =>
  c.in.bits.a.poke(15.U)
  c.in.bits.b.poke(6.U)
  c.clock.step(3)
c.out.valid.expect(true.B)
c.out.bits.expect(3.U)
  c.out.dequeueExpect(3.U)
}
```

We need some kind of concurrency

FSM / Callbacks

- Similar to writing hardware (i.e., hard)
- Requires writing scaffolding of FSM, in addition to core test logic

Threading

- Fork-join parallelism: multiple threads run in parallel
 - `fork`: create new thread
 - `join`: wait for target thread to finish
- State implicit from by position in code, maintained by infrastructure inbetween
- Low-overhead, better test-logic-to-boilerplate ratio

Wire ownership addresses threading pitfalls

Threading has pitfalls

- What happens if threads conflict with each other?

Intuitively

- Fundamentally unclear who owns what
- But if that were clear, race conditions could be detected and cause errors

What should happen here?

```
c.in.poke(1.U)
fork {
    c.in.poke(2.U)
}
fork {
    c.in.poke(3.U)
}
```

Explicit durations are cumbersome

Simple method: user-specified durations

- Test writer must specify directions, adds additional cognitive overhead
- Misses that groups of signals may be related

```
class Decoupled[T] extends Bundle {  
  ...  
  def enqueue(data: T) {  
    this.in.valid.poke(true.B, 1)  
    this.in.bits.poke(data, 1)  
    this.in.ready.expect(true.B)  
    clock.step(1)  
  }  
}
```


Scoping for boilerplate-free wire ownership

Structured programming inspiration

- Duration 'implicitly' specified by the enclosing scope (block of code)
- Can group multiple pokes together
- Duration obvious visually

```
this.in.valid.poke(false.B)
clock.step(2)
timescope { // enqueue
    this.in.valid.poke(true.B)
    this.in.bits.poke(data)
    this.in.ready.expect(true.B)
    clock.step(1)
}
```

Scoping for boilerplate-free wire ownership

Structured programming inspiration

- Duration 'implicitly' specified by the enclosing scope (block of code)
- Can group multiple pokes together
- Duration obvious visually

```
this.in.valid.poke(false.B)
clock.step(2)
timescope { // enqueue
    this.in.valid.poke(true.B)
    this.in.bits.poke(data)
    this.in.ready.expect(true.B)
    clock.step(1)
}
```

```
clock
in.valid
in.bits
```

Scoping for boilerplate-free wire ownership

Structured programming inspiration

- Duration 'implicitly' specified by the enclosing scope (block of code)
- Can group multiple pokes together
- Duration obvious visually

```
this.in.valid.poke(false.B)  
clock.step(2)  
timescope { // enqueue  
    this.in.valid.poke(true.B)  
    this.in.bits.poke(data)  
    this.in.ready.expect(true.B)  
    clock.step(1)  
}
```

```
clock      1  2  
in.valid  F  F  
in.bits   x  x
```

Scoping for boilerplate-free wire ownership

Structured programming inspiration

- Duration 'implicitly' specified by the enclosing scope (block of code)
- Can group multiple pokes together
- Duration obvious visually

```
this.in.valid.poke(false.B)  
clock.step(2)  
timescope { // enqueue  
    this.in.valid.poke(true.B)  
    this.in.bits.poke(data)  
    this.in.ready.expect(true.B)  
    clock.step(1)  
}
```

clock	1	2	3
in.valid	F	F	T
in.bits	x	x	D

Scoping for boilerplate-free wire ownership

Structured programming inspiration

- Duration 'implicitly' specified by the enclosing scope (block of code)
- Can group multiple pokes together
- Duration obvious visually

```
this.in.valid.poke(false.B)
clock.step(2)
timescope { // enqueue
  this.in.valid.poke(true.B)
  this.in.bits.poke(data)
  this.in.ready.expect(true.B)
  clock.step(1)
}
```

clock	1	2	3	4
in.valid	F	F	⊥	F
in.bits	x	x	⊔	x

Scoping for boilerplate-free wire ownership

Structured programming inspiration

- Duration 'implicitly' specified by the enclosing scope (block of code)
- Can group multiple pokes together
- Duration obvious visually
- Bonus: being clear about durations is elegant in general, avoids state errors

Rules

- Scopes can take ownership from parents, including parent threads
- Combinational pokes and expects can be checked for reachability

```
this.in.valid.poke(false.B)
clock.step(2)
timescope { // enqueue
  this.in.valid.poke(true.B)
  this.in.bits.poke(data)
  this.in.ready.expect(true.B)
  clock.step(1)
}
```

clock	1	2	3	4
in.valid	F	F	⊥	F
in.bits	x	x	⊔	x

Examples

Example: concurrency with shift register

Shifter test abstractions possible

- Multiple “instances” of `shiftTest` run concurrently with `fork`
- Causality is obvious within each `shiftTest`

```
def shiftTest(c: Shift2, v: UInt) {
  timescope {
    c.in.poke(v)
    c.clock.step(1)
  }
  c.clock.step(1)
  c.out.expect(v)
}

test(new Shift2) {
  fork { shiftTest(c, 0.U) }
  c.clock.step(1)
  fork { shiftTest(c, 1.U) }
  ...
}
```


Example: we can build and use libraries

```
class DecoupledSource
  (x: Decoupled, clk: Clock) {
    x.valid.poke(false.B) // init
    def enqueue(data: T) = timescope{
      x.ready.expect(true.B)
      x.bits.poke(data)
      x.valid.poke(true.B)
      clk.step(1) // hold for 1 clk
    }
  }
```

```
class DecoupledSink ...
```

```
test(new Gcd) { c =>
  val inSource = new
    DecoupledSource(c.in, clock)
  val outSink = new
    DecoupledSink(c.out, clock)

  inSource.enqueue(
    c.in.Lit(15.U, 6.U))
  outSink.waitForReady()
  outSink.dequeueExpect(3.U)
}
```

Future Work: Integrating Constrained Random

Directed testing is easy, but industry has moved onto constrained-random and coverage-driven

Testers2 provides abstractions, can have constrained random built on top

```
test(new Gcd) { c =>
  ...
  val gcd_in = c.in.randomize(
    0 to 1024)
  val exp_out = calcGcd(
    gcd_in.a, gcd_in.b)

  c.in.enqueue(gcd_in)
  c.out.waitForReady()
  c.out.dequeueExpect(exp_out)
}
```

Recap

tl;dr: encourage unit testing by making writing them **easy** and **painless**

Try the open alpha!

<https://github.com/ucb-bar/chisel-testers2>

(also available as a managed dependency)

- **Minimize boilerplate** by providing clean, intuitive, minimal interfaces
- **Encourage re-use** by enabling abstraction of test functions
- **Allow concurrency** to enable re-use of sequences with fork-join concurrency
- **Clarify wire ownership** to detect race conditions and avoid nondeterminism
- **Time scopes** establish ownership while minimizing user effort