Dynamic Verification Library for Chisel

Yuan-Cheng Tsai

Electrical Engineering and Computer Sciences
University of California, Berkeley

May 15, 2021

Acknowledgement

Dynamic Verification Library for Chisel

by Yuan-Cheng (Anson) Tsai

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Borivoje Nikolić
Research Advisor

5/14/2021

(Date)

* * * * * * *

Professor Krste Asanović
Second Reader

5/14/2021

(Date)

Dynamic Verification Library for Chisel

by

Yuan-Cheng (Anson) Tsai

A thesis submitted in partial satisfaction of the

requirements for the degree of

Masters of Science

in

Electrical Engineering Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolić, Chair
Professor Krste Asanović

Spring 2021

Abstract

Dynamic Verification Library for Chisel

by

Yuan-Cheng (Anson) Tsai

Masters of Science in Electrical Engineering Computer Science

University of California, Berkeley

Professor Borivoje Nikolić, Chair

As Chipyard (Berkeley's open-source SoC development framework) and Chisel (Berkeley's open-source hardware description language) are rapidly growing in popularity within both academia and industry, the need of a compatible verification library is stronger than ever. The industry standard UVM is not suitable with Chisel circuits, as although the generated Verilog can be verified, readability and debuggability is challenging for large designs. Moreover, as Chisel is built upon Scala, a general multi-purpose programming language, special high-level constructs such as type classes and generics that Chisel circuits are parameterized by are very difficult to represent in SystemVerilog. The open-source Chisel Verification Library remedies this disconnect between Chisel and UVM by providing high level verification constructs for Chisel testbenches. It is modeled off of UVM to contain the benefits of the test component abstractions such as the driver and monitor, and it also supports common interfaces such as the Decoupled and TileLink interface. Moreover, the library includes a SVA-like specification language eDSL that enable users to write property assertions on output traces. The library has been used in several applications, such as the verification of a RAM module, L2 cache, as well as an AES cryptography accelerator.

Contents

List of Listings

List of Figures

List of Tables

Acknowledgments

I would first like to thank my research advisor, Professor Borivoje Nikolić, for giving me the opportunity to join ADEPT and BWRC during my undergraduate and graduate years. He has supported me throughout these years and has offered great guidance in my research projects. I have learned many things since, and the experience has given me more inspiration to pursue a career in digital hardware.

I would also like to thank my mentor, Vighnesh Iyer, for his mentorship throughout the research project. Whenever I had questions on the project, debugging, or even other work, he was always very helpful.

Last but not least, I would like to thank my family, friends, and peers for their continuous support throughout these remote times.

Chapter 1

Background and Project Introduction

1.1 Background

1.1.1 Chipyard

Developed at Berkeley, Chipyard[2] is a framework for building Chisel-based SoCs (systems-on-chip). It is a collection of open-source Chisel generators such as the Rocket Chip SoC generator and other Berkeley projects that together enable the generation of RISC-V SoCs. Figure 1.1 shows the design flows supported by Chipyard.

Chipyard lacks unit or subsystem-level verification infrastructure, and as such individual components are rarely tested at unit-level. Instead, the SoC is tested as a whole via software tests running on its CPU cores. This can result in situations where a working design can fail if parameterized differently, as previously hidden bugs are now exposed under a different configuration. As Chipyard has grown substantially in usage across both academia and industry, there is a stronger need for infrastructure to support unit-level testing of Chisel RTL to ensure that individual components and subsystems are bug-free. The next sections describe how Chisel-generated RTL is different from Verilog from the perspective of verification, and why the current industry verification practices are inadequate for Chisel RTL.
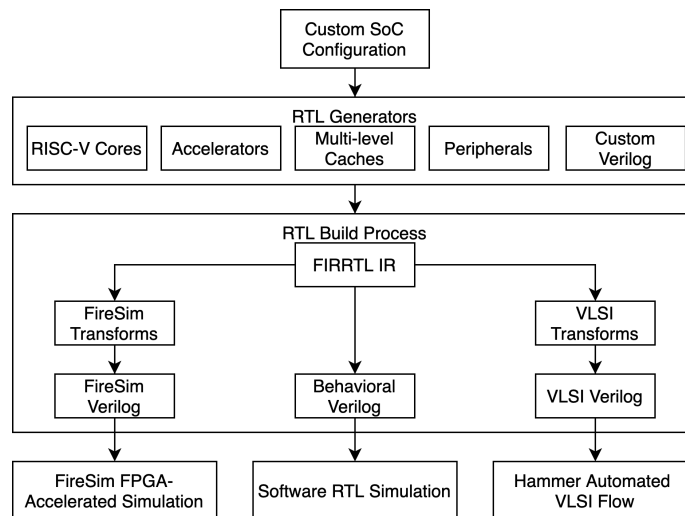
Figure 1.1: Chipyard flows. Adapted from Chipyard paper[3].
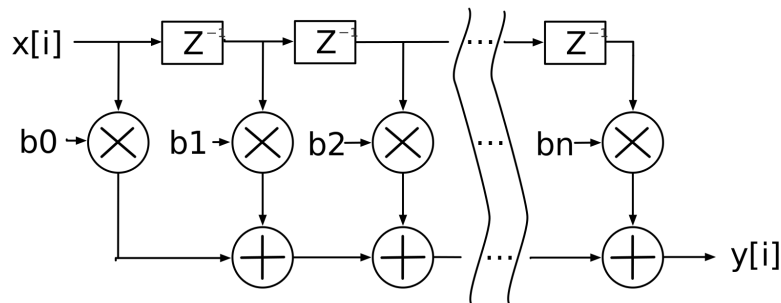
Figure 1.2: N-FIR Filter.

1.1.2 Chisel

Chisel is a Berkeley-made domain specific language (DSL) that adds hardware construction primitives to the Scala programming language. Embedded in a general-purpose programming language, Chisel supports metaprogramming facilities such as parameterization that allows designers to create circuit generators that can be re-used in many designs. In the backend, Chisel emits FIRRTL, or Flexible Internal Representation for RTL, that can be further compiled into the standard Verilog or SystemVerilog (SV) that can then be used with the standard toolchains for synthesis, place and route, etc. To see Chisel in action, consider the following FIR filter example that is adapted from the Chisel website[1]. Figure 1.2 displays the general block diagram of a N-FIR filter that performs convolution operations.

Using only the Chisel primitives similar to that of synthesizable Verilog, Listing 1.1 displays the implementation for a 3-point moving average implemented in a FIR filter style.

```scala
1  case class FIRIO(bitWidth: Int) extends Bundle {
2    val in = Input(UInt(bitWidth.W))
3    val out = Output(UInt(bitWidth.W))
4  }
5
6  class MovingAverage3(bitWidth: Int) extends Module {
7    val io = IO(FIRIO(bitWidth))
8
9    val z1 = RegNext(io.in)
10   val z2 = RegNext(z1)
11
12   io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)
13 }
```

Listing 1.1: Verilog-like Chisel for 3-point moving average (FIR Filter)[1].

Starting at the top, the IO of the FIR filter is defined as a Chisel `Bundle`, a struct-like object that groups together fields of varying types. Within the `Bundle`, two fields, `in` and

out, are defined as `UInt` types (unsigned integer hardware type) that have a parameterizable width of `bitWidth`. The directionality of the fields are indicated by `Input` and `Output`.

The circuit is defined on line 6 via a class that inherits from `Module` to declare that this object is a hardware module. Line 7 instantiates the `FIRIO` and wraps it within a `IO` function to declare it as the IO. On lines 9 and 10, two registers defined with the constructors `RegNext` combine together to create a shift register chain that stores the values from the previous two cycles. Note that the `clock` and `reset` ports are implicitly defined and connected to the registers when the module is elaborated. Lastly, the Chisel `:=` operator on line 10 connects the sum of the multiplication operations with unsigned integers (denoted by `.U`) to `io.out`, implementing the 3-point moving average circuit.

However, if we utilize Chisel's strengths in that it is embedded within Scala, we can write a generator that can create a generalized N-FIR filter defined by a list of input coefficients:

```
1   // Generalized FIR filter parameterized by the convolution coefficients
2   class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
3     // FIRIO specified in previous example
4     val io = IO(FIRIO(bitWidth))
5
6     // Create the serial-in, parallel-out shift register
7     val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
8     zs(0) := io.in
9     for (i <- 1 until coeffs.length) {
10      zs(i) := zs(i-1)
11    }
12
13    // Perform the multiplications
14    val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))
15
16    // Sum up the products
17    io.out := products.reduce(_ + _)
18  }
```

Listing 1.2: N-FIR Filter generator[1].

The above implementation utilizes Scala constructs to create a generalizable FIR Filter circuit. Specifically, it uses a `for` loop to create the shift register chain, `tabulate` on line 14 to create a vector of multiplications, and `reduce` on line 17 for the final sum. Now, additional variants can now be defined without having to implement separate circuits:

```
1   // same 3-point moving average filter as before
2   val movingAverage3Filter = Module(new FirFilter(8, Seq(1.U, 1.U, 1.U)))
3   // 1-cycle delay as a FIR filter
4   val delayFilter = Module(new FirFilter(8, Seq(0.U, 1.U)))
5   // 5-point FIR filter with a triangle impulse response
6   val triangleFilter = Module(new FirFilter(8, Seq(1.U, 2.U, 3.U, 2.U, 1.U)))
```

Listing 1.3: Various types of FIR Filters[1].

Moreover, there is no limit on which Scala constructs can be used. For example, the FIR filter could have parameterized the sum function to take on different data types. The function could be more expressive with the use of higher-order functions, typeclasses and type generics, as shown below with an example implementation of an accumulate function for Ring values:

```scala
def accumulate[T <: Data : Ring](v: Vec[T]): T = {
  val ring = implicitly[Ring[T]]
  v.foldLeft(ring.zero)((a, b) => ring.add(a, b))
}
```

Listing 1.4: Example of accumulate function.

Even with these software constructs, the circuit can still be generated into functional Verilog. As mentioned earlier, Chisel can emit FIRRTL which is an RTL intermediate representation that can then be compiled to Verilog/SystemVerilog. For simplicity, let's consider the FIRRTL and generated Verilog for the simple FIR circuit implemented in Listing 1.1:

```
circuit FirFilter :
  module FirFilter :
    input clock : Clock
    input reset : UInt<1>
    output io : {flip in : UInt<8>, out : UInt<8>}

    reg zs : UInt<8>[3], clock
    zs[0] <= io.in
    zs[1] <= zs[0]
    zs[2] <= zs[1]
    node _T = mul(zs[0], UInt<1>("h01"))
    node _T_1 = mul(zs[1], UInt<1>("h01"))
    node _T_2 = mul(zs[2], UInt<1>("h01"))
    wire products : UInt<9>[3]
    products[0] <= _T
    products[1] <= _T_1
    products[2] <= _T_2
    node _T_3 = add(products[0], products[1])
    node _T_4 = tail(_T_3, 1)
    node _T_5 = add(_T_4, products[2])
    node _T_6 = tail(_T_5, 1)
    io.out <= _T_6
```

Listing 1.5: Generated FIRRTL.

Above is the generated FIRRTL of the 3-point moving average FIR filter. From brief inspection, the structure can be seen to be very similar to that of listing Listing 1.1. Next, Listing 1.6 displays the Verilog generated from the FIRRTL code.

```verilog
1  module FirFilter(
2    input         clock,
3    input         reset,
4    input  [7:0] io_in,
5    output [7:0] io_out
6  );
7    reg [7:0] zs_0;
8    reg [7:0] zs_1;
9    reg [7:0] zs_2;
10   wire [8:0] products_0 = zs_0 * 1'h1;
11   wire [8:0] products_1 = zs_1 * 1'h1;
12   wire [8:0] products_2 = zs_2 * 1'h1;
13   wire [8:0] _T_4 = products_0 + products_1;
14   wire [8:0] _T_6 = _T_4 + products_2;
15   assign io_out = _T_6[7:0];
16   always @(posedge clock) begin
17     zs_0 <= io_in;
18     zs_1 <= zs_0;
19     zs_2 <= zs_1;
20   end
21 endmodule
```

Listing 1.6: Generated Verilog.

Although Chisel circuits can be converted into Verilog, verifying a Chisel circuit is different than verifying a Verilog one. Verifying Chisel generated Verilog is more complex than handwritten Verilog due to the metaprogramming features from Scala. Chisel elaboration can consume many parameters, and those parameters can affect the test execution, stimulus generation, and correctness checking. Consider the FIR filter example: the circuit is parameterized by the `bitWidth` and `coefficients` variables, which when changed will affect stimulus generation and golden model checking. Even in more complex designs like a level 2 (L2) cache, there are many microarchitecture (uArch) parameters and interface parameters that affect circuit behavior. Moreover, Chisel circuits come with metadata about the interfaces around a DUT that should be utilized by the test environment to provide compatible test components (e.g. if the interface operates on ready-valid signals). With highly parameterizable Chisel circuits, a notion of coverage of the RTL generation space is also needed. To allow easy enumeration of the design space and easy testing of a DUT, it should be done in the same language that Chisel circuits are written in, Scala.

Using Scala as the test driver host language has many benefits over the standard Verilog + SV testbench. SystemVerilog is a limited language with restricted object oriented programming features; it lacks higher-kinded types, lacks typeclass support, lacks of co/contravariance for type generics, and has poor support for higher order functions (e.g. ad hoc lambdas). For example, the earlier `accumulate` example shown in Listing 1.4 would be very difficult, if not impossible, to represent in SV. As such, industry standard SV libraries like UVM have complex abstractions that could be represented in a more friendly way with a
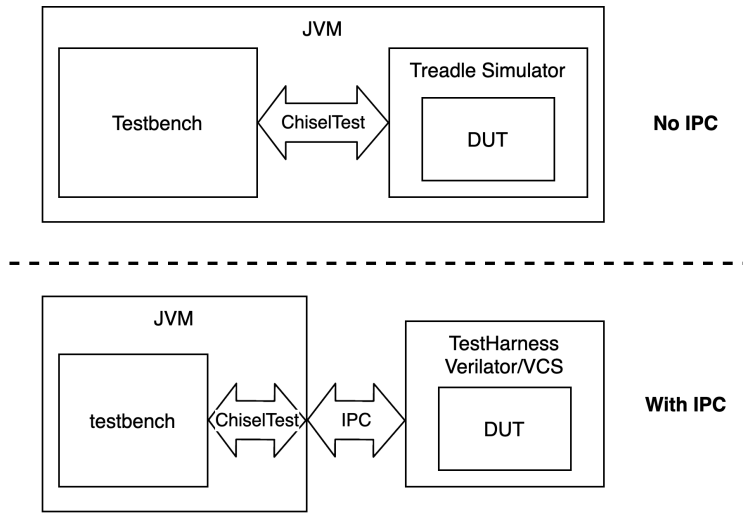
Figure 1.3: ChiselTest API overview.

general purpose programming language (see Section 1.1.4 for examples).

Scala, on the other hand, is a general-purpose programming language that offers more unified interaction interfaces and more complete functionalities. For example, Scala has easy interaction with models written in C/C++ via JNI. Moreover, Scala language constructs enable hybrid object oriented programming and functional programming paradigms through classes, traits, and typeclasses. It also has the ability to utilize the Java ecosystem to build golden models and Java's IPC mechanisms to talk to other processes. Some of these benefits can be seen in the ChiselTest, explained in the following section.

1.1.3 ChiselTest

ChiselTest is a test harness designed for Chisel RTL. Written in Scala, it operates as the test interaction API between Scala and an arbitrary RTL simulator, enabling easy swapping of different simulators. ChiselTest can also integrate with ScalaTest, the default testing tool in Scala, using the `expect()` API. For a visual diagram of where ChiselTest fits in the testbench, Figure 1.3 displays the interaction between user code, ChiselTest, and various RTL simulators such as Treadle, Verilator, and VCS.

For simulators that run as different processes (e.g. Verilator and VCS), ChiselTest utilizes Java's IPC functionality to connect to them. This adds performance overhead as every simulation cycle within the test has to communicate across processes. Treadle, on the other hand, avoids the IPC overhead as it is a Scala-native simulator and shares the same Java virtual machine (JVM) as the testbench.

In terms of functionality, ChiselTest contains a library that enables basic interaction with the DUT interface such as `peek`, `poke`, `step` (to step the clock), and `fork/join`. In terms of `fork` semantics, ChiselTest's `fork` is similar to SV's `fork` as they both execute the specified

code block in another thread. Moreover, the `join` construct in ChiselTest is similar to SV's in that it causes the main thread to wait until all spawned threads have completed. If `join` is not given, the main thread will continue to execute alongside the spawned threads. Something to note is that ChiselTest does not natively support SV's `join_any` construct (where the main thread will wait until a thread has completed), although it can be implemented using shared state between all spawned threads that forces all threads to prematurely end when one completely finishes. For a brief example of using ChiselTest, consider the following code that is testing a standard 2-input adder.

```
1   test(new Adder(8)){ c =>
2     fork {
3       for (i <- 1 to 3) {
4          c.io.a.poke(i.U)
5          c.clock.step()
6       }
7     } .fork {
8       for (i <- 1 to 3) {
9          c.io.b.poke(i.U)
10         c.clock.step()
11      }
12    } .join
13    c.clock.step(2)
14    c.io.a.poke(4.U)
15    c.io.b.poke(4.U)
16    c.clock.step()
17  }
```
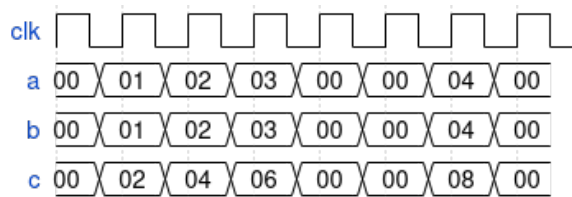

Figure 1.4: Adder test wave.

Listing 1.7: Adder test code.

Line 1 defines a ChiselTest `test` environment with `new Adder(8)`, or an 8-bit adder, as the DUT. Within this `test` block, `c` refers to the DUT in simulation. Within the test, two threads are instantiated, with each poking an input of the adder. Note that in the waveform, the inputs increment together, as the call to `c.clock.step()` acts as a thread synchronizer and blocks until all relevant threads have reached that point. When both threads finish, the main thread then continues and waits 2 cycles before poking 4 on both inputs. Note that if `.join` was not included, all three threads (2 forked + main) will be running concurrently and cause an assertion for conflicting pokes on both inputs.

In terms of ChiselTest's underlying implementation of `fork/join`, each `fork` block creates a parked operating system (OS) thread. During test execution, all available threads are multiplexed such that only one thread can run at a time. When the running thread encounters a call to `step()`, the thread is parked and the runtime scheduler finds the next thread to run. While the thread execution order is defined and fully deterministic, ChiselTest will still register and check that the actions of each thread do not conflict with another's or potentially cause non-intuitive behavior. For instance, ChiselTest will throw an assertion when multiple threads attempt to `poke` the same wire in the same cycle, or if a thread attempts to `poke` a wire that has a combinational path with a wire that another thread is trying to `peek`
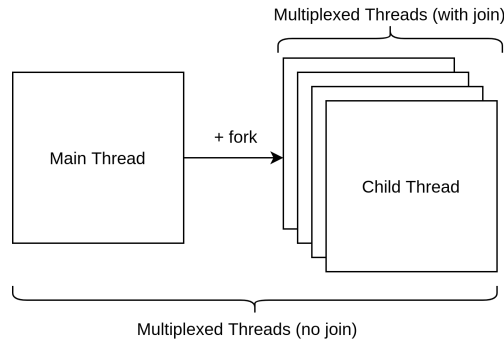
Figure 1.5: ChiselTest fork/join diagram.

at the same cycle. For the latter situation, ChiselTest offers a `fork.region` construct that allows ordering of threads, as demonstrated in the following code snippets. There are two pre-defined regions `TestdriverMain` and `Monitor`, with `TestdriverMain` regions executed first. Lastly, if `join` is used on the forked threads, the main thread will remain parked until all spawned threads complete. Figure 1.5 displays a high-level diagram of ChiselTest's implementation of `fork`/`join`.

On top of the base `peek` and `poke` functionality, ChiselTest also contains wrapper `enqueue` and `dequeue` functions that simplify poking and peeking decoupled, or ready-valid, interfaces. Below is a short code snippet of the `enqueue` and `expectDequeue` functions. Note that the `expectDequeue` function contains a helper method `waitForValid()` that checks if the interface is valid, as it should not be checked within the `Monitor` region.

```
def enqueue(data: T): Unit = {
  x.bits.poke(data)
  x.valid.poke(true.B)
  fork.withRegion(Monitor) {
    while (!x.ready.peek().litToBoolean) {
      getSourceClock.step(1)
    }
  } .joinAndStep(getSourceClock)
}
```

Listing 1.8: ChiselTest enqueue.

```
def expectDequeue(data: T): Unit = {
  x.ready.poke(true.B)
  fork.withRegion(Monitor) {
    waitForValid()
    x.valid.expect(true.B)
    x.bits.expect(data)
  } .joinAndStep(getSinkClock)
}
```

Listing 1.9: ChiselTest expectDequeue.

However, these base constructs do not allow fuzzing capability other than direct stimulus driving. Take the below naive ChiselTest testbench for a queue for example:

```
1  test(new QueueModule(UInt(8.W), 8)) { c =>
2    fork {
3      for (i <- 0 until 10)
4        c.in.enqueue(i.U)
5    } .fork {
6      for (i <- 0 until 10)
7        c.out.expectDequeue(i.U)
8    } .join
9  }
```

Listing 1.10: Naive ChiselTest QueueModule testbench.

This test enqueues and dequeues from the queue on every cycle. If we wanted more complex DUT interactions such as rate throttling on the enqueuing interface, we would have to implement it ourselves as there is currently no systematic way to do this using the given ChiselTest constructs. This can be accomplished by utilizing a counter that increments on each cycle, and once the counter reaches a threshold, we can enqueue the next value and reset the counter. Given that we are unable to modify the enqueue function, we would have to poke and peek the control signals ourselves, as shown in Listing 1.11. If we also wanted backpressure on the dequeuing interface, a similar tactic can be done with the ready signal.

```
1  test(new QueueModule(UInt(8.W), 8)) { c =>
2    fork {
3      // Rate throttling counter
4      var counter = 0
5      for (i <- 0 until 10) {
6        c.in.valid.poke(false.B)
7        // Example threshold of at least 5 cycles
8        while (!c.in.ready.peek() || counter < 5) {
9          // Get source clock because the Queue can be asynchronous
10          getSourceClock.step(1)
11          counter += 1
12        }
13        c.in.valid.poke(true.B)
14        c.in.enqueue(i.U)
15        // Reset counter
16        counter = 0
17        getSourceClock.step(1)
18      }
19      c.valid.poke(false.B)
20    } .fork {
21      for (i <- 0 until 10)
22        c.out.expectDequeue(i.U)
23    } .join
24  }
```

Listing 1.11: Naive ChiselTest QueueModule testbench with rate throttling.

First, we can see how convoluted the code has gotten with the additional logic for rate throttling. Second, we have now intertwined the protocol (how to drive one transaction) with the metaprotocol (rate throttling) and with the data itself. This methodology is not systematic and scalable, as the logic is hardcoded and baked into the test itself. If, for example, the `QueueModule` was parameterizable by cycle delay, then portions of the test that check the output may have to be re-written to account for the additional delay. Thus the protocol, metaprotocol, and data should be separated such that these test library collateral can be applied to other tests and designs as well. Note that there is already an industry standard Verilog verification library, UVM, that offers test component abstractions that separate the responsibilities across multiple components.

1.1.4 Prior Work and State of the Art - UVM

The Universal Verification Methodology (UVM)[4] is a industry standardized methodology used for verifying Verilog RTL designs. It was based on the Open Verification Methodology (OVM)[5], which is another dynamic verification methodology library. Both are SystemVerilog based verification methodologies, and their usefulness comes from their abstractions that divide the testbench into test components that each have their distinct roles.

The UVM figures below display the clean separation of stimulus generation, DUT interactions, and data consistency/model checking constructs. The first figure is the structure of a UVM testbench:
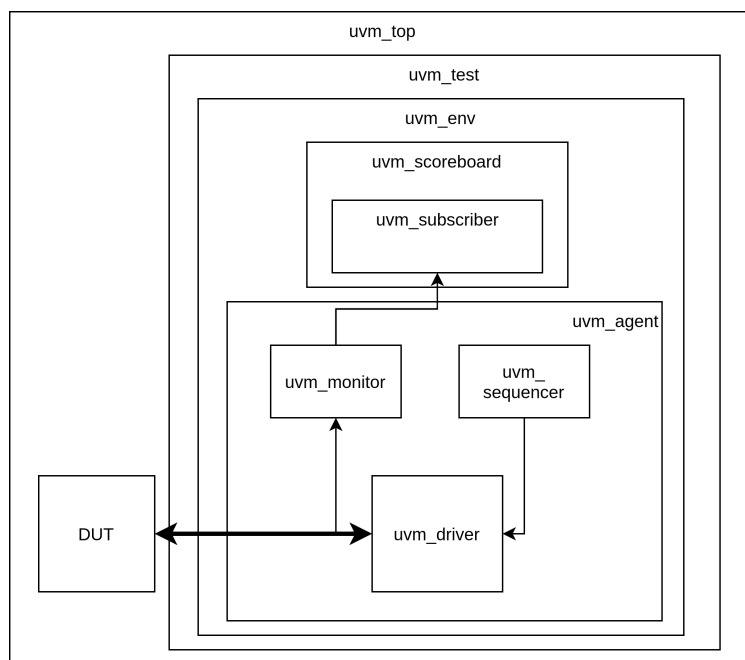
Figure 1.6: UVM testbench layout.

First, all of the components are housed within `uvm_top`. It contains the DUT and `uvm_test`, which contains all the test components that interact with the DUT's interface. Within the `uvm_test`, the `uvm_env` holds the `uvm_agent`, which is responsible for interacting with the DUT, and the `uvm_scoreboard`, which is responsible for checking the output of the DUT. Looking inside the `uvm_agent`, the core test components are connected to the DUT's interface. The functionality of the `uvm_driver` is to both drive the transactions supplied from the `uvm_sequencer` onto the DUT's interface and receive any responses the DUT may have. The `uvm_monitor`, on the other hand, only observes the DUT interface and records all exchanged transactions. The transactions recorded by the `uvm_monitor` can also be sent to other components within the environment, such as the `uvm_scoreboard` for output checking. For the observed transactions to be forwarded, the `uvm_scoreboard` must contain a `uvm_subscriber` for that particular monitor.

Note that all of the test components within the environment can appear in any number; there can be multiple `uvm_agents` communicating on different interfaces of the DUT, multiple `uvm_scoreboards` for multiple `uvm_monitors`, etc. The flexibility and modular usage is possible as UVM has clean separation between its components.
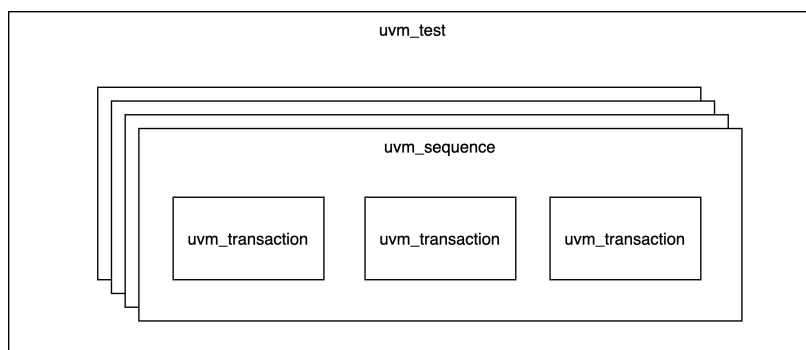
Figure 1.7: UVM transactions.

Figure 1.7 describes the structure of test stimulus. Here, the `uvm_transaction` is the unit abstraction that represents a single piece of data sent to the DUT. A group of `uvm_transactions` forms a `uvm_seqeuence`, and a set of `uvm_seqeuences` forms a `uvm_test`. Connecting this with the above diagram, the `uvm_seqeuencer` takes the `uvm_transactions` from the `uvm_test` and sends them to the driver to push to the DUT. However, a distinction to be made here is that all UVM components, including the transactions, are written in SystemVerilog. For it to be pushed onto the DUT interface, the driver must convert the SV datatype struct into the DUT HW datatype struct. This also applies to the transactions observed by the monitor as well. As such, there will be two objects for each transaction.

Looking at the overall UVM structure, there are some abstractions that can be represented in a more user friendly way with a general purpose programming language. For example, the `uvm_test` abstraction is essentially a wrapper for a list of sequences, which in Scala could be recreated with `Seq[Seq[Transaction]]`. This Scala interpretation also

avoids the `uvm_sequence` abstraction all together. Moreover, other abstractions such as `uvm_agent` and `uvm_subscriber` are additional complexities that come from how interfaces are connected in different phases in SV. In Scala, all of these complex abstractions can be replaced with generic structures.

1.1.5 Motivation

As described above, there is a strong need for Scala-based verification infrastructure. It would greatly simplify the verification of Chisel circuits by consolidating the design language and the test language into the same one. For instance, it would enable easy enumeration of the design space for highly parameterizable Chisel circuits. Moreover, as Scala is a general-purpose programming language, the test infrastructure can benefit from many of the flexible language constructs and existing software packages in the Java ecosystem: JNI for easy interaction with C/C++ models, Java's IPC mechanisms to talk to other processes, etc. We've seen the flexibility of the IPC mechanisms through ChiselTest's ability to easily swap RTL simulators. And although it only provides the base functionality of interacting with a DUT, ChiselTest provides a suitable foundation for more complex verification infrastructure to be built on top. As such, the goal of the this verification library is to provide test library collateral and infrastructure that can implement the UVM methodologies in a Chisel compatible environment.

1.2 Dynamic Verification Library for Chisel

The Dynamic Verification Library for Chisel[6][1] aims to adapt UVM methodologies in a Chisel compatible environment. As such, the library has a UVM-like hierarchical structure that contains verification IPs (VIPs) such as drivers, monitors, and scoreboard-like checking components that operate on the transaction level. These VIPs can be instantiated in a ChiselTest simulation environment and connected to a DUT. Moreover, the library can be standalone or intergrated within Chipyard, allowing for the library to be used to verify complex designs such as level 2 caches and Rocket accelerators.

1.2.1 Library Structure

The library currently consists of three sublibraries: `verifCore`, `verifTL`, and `verifCosimGemmini`. The `verifCore` sublibrary contains the base test components for standard interfaces such as the generic, valid-only, and decoupled interface. It also contains an embedded SVA-like specification language for transaction-level trace verification. The `verifTL` sublibrary extends `verifCore` by providing test components for the TileLink interface, a chip-scale interconnect used in Chipyard-based SoCs. The `verifCosimGemmini`

---

[1]As the verification library is still under development, the contents of this paper may be outdated at the time of reading.

sublibrary contains the test components required for Gemmini cosimulation, or the ability to test Gemmini with simulations of their environment (e.g. a RISCV-ISA simulator). Note that this report will cover only the `verifCore` and `verifTL` sublibraries.

1.2.2 Test Component Library Overview

The verification library contains test components that mimic UVM abstractions. Figure 1.8 displays what a testbench looks like using the library.
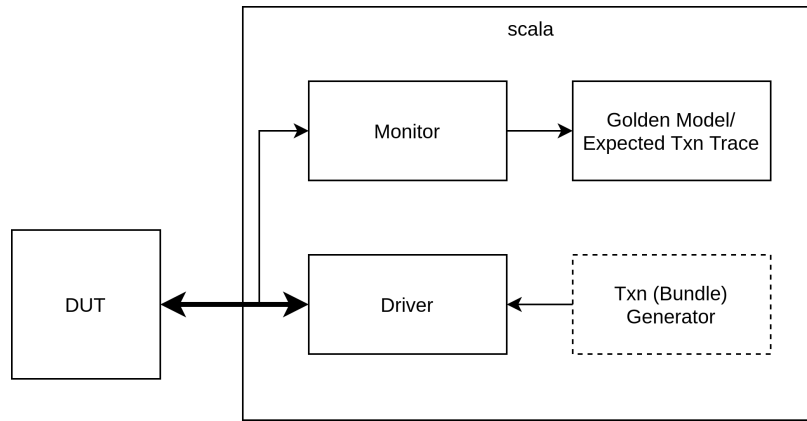
Figure 1.8: Generic library testbench diagram.

Notice the similar but simplified structure compared to the UVM testbench shown in Figure 1.6. Starting from the bottom right, the transaction generator will generate transactions to send to the driver. Note that the generator is optional, as the user can also supply transactions directly to the driver. Transactions in this library are defined as Chisel bundles that are, or contain, the same Chisel bundle which is used by DUT's interface. For example, referring back to the very Chisel example of a 3-point moving average FIR filter (Listing 1.1), the transactions for that design could be the `FIRIO` bundle. By using the DUT's interface bundle datatype as transactions, there is no need to have separate software and hardware versions of transactions as seen with UVM + SV. Another thing to notice is that the library does not have any transaction abstractions such as `uvm_sequence` found in UVM. All collections of transactions can be easily subsumed by Scala collections such as `Seq`.

Moving on to the driver, it is responsible for driving the given transactions on the DUT interface. As transactions are Chisel bundles, they can contain metadata that can configure how the driver handles that transaction. Next, the monitor listens on the DUT interface and records all observed transactions. The observed transactions can then be checked against a software golden model. Note that the library does not have a explicit scoreboard class, as the monitor and checking functions now cover that responsibility.

For more details on the implementation of the test components, see Chapter 2.

1.2.3 Specification Language Overview

The Dynamic Verification Library for Chisel also contains an embedded SVA-like specification language. It supports the basic unit components such as `Atomic Propositions` for unit boolean expressions, `Time Operators` to enable the checking of cycle latency, and `Implications` to indicate a condition to be checked. These unit components can be combined into `Sequences` to model correct sequential behavior of a transaction trace or subtrace. `Properties` can then be used to check if a transaction trace conforms to the given `Sequence`. Moreover, the specification language also has a modeling layer that supports per-property local variables as well as global memory models for transaction checking. More details will be covered in Chapter 3.

1.2.4 Library Application Overview

Chapter 4 will go over a few applications of the verification library. It includes testbench designs and results for the `TLRAM`, SiFive's `InclusiveCache` (L2 cache), and an AES RoCC accelerator. The `TLRAM` is the default RAM block used in Chipyard, and SiFive's `InclusiveCache` is the L2 cache used with the Rocket Core, an open-source RISC-V core within Chipyard. Moreover, the AES RoCC accelerator is an AES block cipher accelerator/co-processor for the Rocket Core. In addition to the test component applications, the chapter will also discuss how the specification language was used to verify `TLRAM` output transaction traces as well as detect malicious mutations within the trace.

Chapter 2

Test Component Library

The test component library consists of VIPs that interact with the DUT. This includes UVM-like abstractions of test components such as transactions, fuzzers, drivers, monitors, and even correctness checking functions. The library can be divided into two groups: the core test components and the TileLink test components. The core test components refers to a subset of VIPs that operate on the fundamental interfaces: generic, decoupled (ready-valid), and valid-only interfaces. The TileLink test components are composed of VIPs that operate on the TileLink interface, which is built upon the decoupled interface. Each of the supported interfaces and their VIPs are described in this chapter.

2.1 Core Interfaces

2.1.1 Generic Interface

The generic interface is one that describes any simple input-output interface without ready-valid control signals (e.g. Listing 1.1, line 1). Without such signals, the VIPs for the generic interface will attempt to drive and monitor transactions on every cycle, as designs that use the generic interface are usually combinational or stateful circuits with same-cycle results. Note that the VIPs for this interface have been removed from the library, as they can be subsumed by the bare ChiselTest constructs. However, they are still discussed here as they provide the base implementations for the other VIPs.

2.1.2 Decoupled + Valid Only Interfaces

The decoupled and valid-only interfaces extend the generic interface by introducing ready-valid control signals. The receiving interface is responsible for controlling the ready signal, where a high signal signifies that the consumer is ready to accept new data on the interface. The sending interface is responsible for controlling the valid signal, where a high signal signifies that the producer has valid data on the interface. Only when both ready and valid are high in the same cycle is the data interpreted to be successfully transmitted.

The decoupled interface operates with both the ready and valid signal, allowing both the producer and consumer to control the flow of data. Complex interactions such as rate throttling and backpressure, or when the producer decreases the rate of sending data and the consumer decreases the rate of accepting data, are now possible. To enable these behaviors during testing, the VIPs on this interface can be configured to add artificial delays.

The valid-only interface differs from the decoupled interface in that it only has a valid control signal. Without a ready control signal, the consumer must always be available to accept data. As the producer can still modify the data flow, the VIPs for this interface allow support rate throttling configuration.

2.2 Core Transactions

All test components operate on interface-specific transactions that encapsulate all DUT interactions into discrete data objects. As mentioned earlier, transactions are defined as Chisel datatypes that contain the DUT's IO and potentially other metadata. Moreover, as the transactions are written in Chisel/Scala, they can be checked with software, avoiding the redundancy issue in UVM where two sets of transactions are needed: a Verilog version for the DUT itself, and a SystemVerilog version for the testbench. Each transaction type is discussed below.

2.2.1 Generic Transactions

As the generic interface describes a simple interface with no control signals or other metaprotocols, transactions are defined as either the DUT's input/output datatypes or the IO itself. Let's take the IO from the FIR filter from earlier as example (shown again for convenience):

```
1  case class FIRIO(bitWidth: Int) {
2    val in = Input(UInt(bitWidth.W))
3    val out = Output(UInt(bitWidth.W))
4  }
```

Listing 2.1: FIR Filter IO, Taken from Listing 1.1.

If we wanted two separate transaction types such that there is one for the input and one for the output, our input and output transactions would simply be Chisel `UInt`s of the correct width. For example, an input of 100 with a width of 8 would be represented via `val inputTxn = 100.U(8.W)`, and the output transaction would be another 8-bit `UInt`.

However, if we wanted to consolidate the input and output into one transaction, the transactions would then be the DUT's IO `Bundle`. For example, an input of 100 with a width of 8 would be represented via `val inputTxn = FIRIO(8).Lit(_.in -> 100.U, _.out -> 0.U)`. Note that `out` could be set to anything as it is defined as an `Output` and would not be driven to the DUT. For the output transaction, it would be created the same way except with `out` defined with the real output.

2.2.2 Decoupled + Valid-Only Transactions

As briefly mentioned in Section 2.1.2, the rate at which the VIPs push transaction on the DUT can be configured. This is done via two custom transaction bundles, `DecoupledTX` and `ValidTX`, that hold metadata fields. Below is the code for `DecoupledTX`:

```
1  class DecoupledTX[T <: Data](gen: T) extends Bundle {
2    val data: T = gen.cloneType
3    val waitCycles: UInt = UInt(32.W)
4    val postSendCycles: UInt = UInt(32.W)
5    val cycleStamp: UInt = UInt(32.W)
6    ...
7    // Helper fn to create DecoupledTX with given params
8    def tx(data: T, waitCycles: Int, postSend: Int): DecoupledTX[T]
9    ...
10 }
```

Listing 2.2: DecoupledTX implementation.

The metadata fields are `waitCycles` (cycles to wait before sending data), `postSendCycles` (cycles to wait after sending data), and `cycleStamp` (when data was received). The first two will be used by the driver to determine when to push a transaction to the DUT, while the last field will be written by the monitor to mark when the data was observed. As for the datatype of the `data` field, it will correspond to the DUT's IO. For an example, let's look at the IO of the `QueueModule` used earlier in the example in Listing 1.10:

```
1  class QueueModule[T <: Data](ioType: T, entries: Int) extends Module {
2    val in = IO(Flipped(Decoupled(ioType)))
3    val out = IO(Decoupled(ioType))
4    ...
5  }
```

Listing 2.3: QueueModule implementation.

In the code above, `Decoupled` is a Chisel construct that wraps the enclosed datatype with ready-valid signals. Moreover, the `Flipped` operator reverses the direction of the signals. Recall that the queue earlier was instantiated via `QueueModule(UInt(8.W), 8)`, meaning that both `in` and `out` are decoupled ports with the datatype of 8-bit unsigned integers. Thus, the input and output transactions would be of type `DecoupledTX(UInt(8.W))`. The following code shows an example of creating an input transaction:

```
1  val inTxnProto = new DecoupledTX(UInt(8.W))
2  val inputTxn = inTxProto.tx(100.U, 4, 5)
```

Listing 2.4: DecoupledTX example.

Note that line 1 is creating a transaction prototype, and line 2 is utilizing the helper function to create an input transaction with a data payload as `100.U`, `waitCycles` of 4, and `postSendCycles` of 5.

Moving on to `ValidTX`, the implementation is exactly the same, as it is a functional parity of `DecoupledTX` but for the valid-only interface. Thus, all the above also applies for `ValidTX`.

Note that both `DecoupledTX` and `ValidTX`'s datatype `T` can be any Chisel datatype, including `Bundle` types. Moreover, as bundles are concretized to their types, bundles with differing generic types are considered different objects (e.g. `DecoupledTX[UInt]` is different from `DecoupledTX[SInt]`). This type requirement is helpful as Scala will statically catch errors if a wrong transaction type is used. However, due to Chisel's width inference, mismatched widths will not be caught during compile time but instead will be caught during runtime.

2.2.3   Randomization - SMTSampler

In terms of transaction randomization, the library utilizes the Maltese-SMT project[7] to enable constrained randomization of transaction fields. The API is straightforward: call `.rand()` on a Chisel `Bundle` type with a boolean function representing the field constraints of the bundle. Using the `FIRIO` as an example, `FIRIO(8).rand{ b => b.in > 10.U && b.in < 20.U }` will return a `FIRIO` bundle with an 8-bit input that is between 10 and 20.

2.3   Core Sequencers

For a transaction to reach the DUT, it must be sent to a driver. In UVM, `sequencers` are responsible for sending `sequences`, or groups of transactions, to the driver to drive on the DUT interface. In this library, however, the use of sequencers is optional, as all drivers have public functions that allow the user to directly pass in transactions. In cases where there are message handshakes or special message ordering such as in TileLink connections, a sequencer-like object can be used to feed transactions into the driver to ensure that message handshake and ordering rules are followed. For examples of TileLink sequencers, please refer to Section 2.8.6.

2.4   Core Drivers

As an *active* test component, the driver is responsible for driving transactions onto the DUT interface. The sections below will go into detail on how the drivers are implemented for each core interface.

2.4.1   Generic Driver

While there are different variants of drivers for each interface, they all have a common implementation: a forked thread that has a continuous while loop that pokes the DUT IO when a transaction is available, as shown in Listing 2.5. If there are no transactions available, the while loop will continue to step the clock forward. The transactions are fed via an internal mutable queue that the user can access and fill using a method `push(tx: Seq[T]): Unit` (not shown). The usage of a non-constrained infinite loop can pose some problems with test determinism, however, as the loading of transactions into the internal state can take a non-deterministic amount of simulation clock cycles. If all of the transaction data were to be preloaded before the start of the simulation, then cycle nondeterminism would be avoided. Nondeterminism is mainly an issue when it comes to the initial latency of starting up the simulation (e.g. dead cycles at the beginning of a simulation) or when stimulus is designed to be reactive to the DUT behavior during testing.

```
1   // Queue that can be continuously loaded with transactions
2   val inputTransactions = new mutable.Queue()
3   fork {
4     while (true) {
5       if (inputTransactions.nonEmpty) {
6         dutInterface.pokePartial(inputTransactions.pop)
7       }
8       clock.step()
9     }
10  }
```

Listing 2.5: Generic driver implementation.

2.4.2   Decoupled + Valid-Only Drivers

As mentioned previously in Section 2.2.2, the rate at which the drivers push transactions is configurable via the metadata fields contained in both `DecoupledTX` and `ValidTX`.

Starting with the decoupled interface, there are two types of drivers: the master variant to control the valid signal while the slave[1] variant to control the ready signal. The mastering driver builds upon the generic driver by adding logic before and after poking the DUT. Before each transaction is driven, the driver sits idle (steps the clock forward) for `waitCycles` cycles. Similarly, the driver does the same and waits for `postSendCycles` cycles after driving the transaction. The code implementation is shown in the Appendix, Listing A.1.

The implementation for the slaving driver is much simpler as it only has to stall `waitCycles` amount of cycles after receiving a transaction. As such, the code is shown below in full. Note that in this case `waitCycles` is not set by the transaction, but by a parameter that is passed into the class constructor (not shown). Another thing to note is that the driver does not

---

[1]The outdated terms *master* and *slave* are used in this paper to avoid confusion and remain consistent with the TileLink spec, which currently uses the outdated terminology. See Section 2.8 for more information.

store or process the received data — it simply drives the ready signal. Instead, the monitor is responsible for storing the transactions. The reason for using a separate slaving driver instead of a monitor to control the ready signal is that the monitor is a *passive* component and thus should not drive anything on the interface.

```
1   while (true) {
2       interface.ready.poke(false.B)
3       while (idleCycles > 0) {
4           idleCycles -= 1
5           clock.step()
6       }
7       interface.ready.poke(true.B)
8       if (interface.valid.peek().litToBoolean) {
9           idleCycles = waitCycles
10      }
11      clock.step()
12  }
```

Listing 2.6: Decoupled driver slave implementation.

As for the valid-only interface, there is only the mastering driver as there is no ready signal. Moreover, as the `ValidTX` is a functional parity of `DecoupledTX`, the driver for the valid-only interface is almost exactly the same with the exception of the `ready` signal.

2.5 Core Monitors

Unlike the driver, the monitor is a *passive* element that is responsible for recording all exchanged transactions on the interface. As such, the monitor implementations are fairly uniform across all interfaces.

2.5.1 Generic Monitor

```
1   val storedTransactions = new mutable.Queue()
2   fork.withRegion(Monitor) {
3       while (true) {
4           storedTransactions += dutInterface.peek()
5           clock.step()
6       }
7   }
8
9   def clearMonitoredTransactions() = monitoredTransactions.clear()
```

Listing 2.7: Generic monitor implementation.

Similar to the driver, the monitor for the generic interface contains an infinite non-blocking while loop that `peeks` the DUT interface on every cycle. The transactions are stored within an internal list that can be publicly accessed by the user. The internal list can also be cleared with the method `clearMonitoredTransactions`.

Note that the storing of transactions within the monitor deviates from UVM, where monitors only peek the interface and send observed transactions elsewhere, like to a scoreboard. As such, the monitor implementation in this library can be thought as a hybrid of UVM's `monitor` and `scoreboard`, with the exception of transaction checking. Instead, the transactions are processed and checked outside the monitor, as explained in the Section 2.6.

2.5.2   Decoupled + Valid-Only Monitors

Both the decoupled and valid-only monitors are similar to that of the generic interface:

```scala
1   while (true) {
2     if (interface.valid.peek() && interface.ready.peek()) {
3       val t = new DecoupledTX(interface.bits)
4       val tLit = t.Lit(_.data -> interface.bits.peek(), _.cycleStamp -> cycleCount.U)
5       monitoredTransactions += tLit
6     }
7     cycleCount += 1
8     clock.step()
9   }
```

Listing 2.8: Decoupled monitor implementation.

The difference here is that the monitors only peek the interface when it observes that the control signals are high: both ready and valid for decoupled, and valid for valid-only. Moreover, there is an additional `cycleCount` counter to keep track of when each transaction was observed. Everything else remains the same; the monitor contains an internal list that stores all valid transactions, and it can be accessed in the testbench. Note that only the `DecoupledMonitor` is shown, as the only difference in `ValidMonitor` is on line 2-3.

2.6   Core Scoreboards

Currently, the library does not contain a special abstractions that directly mimic a UVM scoreboard. Instead, they are implemented as forked imperative checkers in the main simulation function that instantiates the test components; they retrieve monitored transactions and compare them against a golden model.

2.6.1   Golden Models

Golden models in this library can be written in Scala and other languages such as C and C++ (integrated via JNI). Moreover, as golden models are software based, they can

be re-used anywhere in the testbench. As an example, later sections describe how the TLMemoryModel, a golden model for basic TileLink memory devices, is used as a slaving function for the TileLink driver as well as a golden model for the TLRAM. For more information on the TLMemoryModel, see Section 2.8.8. For information on the TLDriverSlave and its usage in the TLRAM testbench, see Section 4.1.1.

2.7   Example Testbench

Revisiting the example QueueModule from Chapter 1, recall that the issue earlier was that complex interface interactions such as rate throttling and backpressure were difficult to introduce without intertwining the protocol, metaprotocol, and data. Now, with test components that are cleanly separated, the test can now be reconstructed as shown below.

```
1   test(new QueueModule(UInt(8.W), 8)) { c =>
2     // Defining Push Driver, Pull Driver, Output Monitor
3     val enqDriver = new DecoupledDriverMaster(c.clock, c.in)
4     val deqDriver = new DecoupledDriverSlave(c.clock, c.out, waitCycles = 5)
5     val deqMonitor = new DecoupledMonitor(c.clock, c.out)
6
7     // Defining input transactions + sending to Driver
8     val txProto = new DecoupledTX(UInt(8.W))
9     val inputTxns = Seq(txProto.tx(165.U, 0, 1), txProto.tx(15.U, 3, 2),
10                        txProto.tx(251.U, 1, 2), txProto.tx(29.U, 4, 3))
11    enqDriver.push(inputTxns)
12    c.clock.step(100)
13
14    // Collecting output, generating golden trace, and check matches (Scoreboard)
15    val output = deqMonitor.monitoredTransactions
16
17    val model = new SWQueue(8, UInt(8.W))
18    val swoutput = model.process(inputTxn, cycles = 100, waitCycles = 5)
19
20    output.zip(swoutput).foreach {
21      case (dut_out, sw_out) =>
22        assert(dut_out.data.litValue() == sw_out.data.litValue())
23        assert(dut_out.cycleStamp.litValue() == sw_out.cycleStamp.litValue())
24    }
25  }
```

Listing 2.9: Queue testbench with rate throttling and backpressure.

Lines 3-5 define both drivers on the master and slave interface as well as the output monitor on the slave interface. Note that the driver on the slave interface (line 4) is configured by a parameter waitCycles, which defines the number of cycles to wait in between accepting transactions (backpressure). Next, on lines 8-12, input DecoupledTX transactions are formed with hardcoded values and parameters and then given to the driver to push to the DUT.

Once the clock has been advanced (constant 100 cycles for this example), the DUT output is then extracted from the monitor. Lines 17-18 then demonstrate how a golden model is used to generate a golden trace. Note that `SWQueue` is a software-implementation of a cycle-accurate queue that consumes and produces `DecoupledTX` transactions. On lines 20-24, the DUT output data payload and cycle stamp is then compared with the golden output.

Note that one caveat of this test structure is that the whole DUT output trace is generated before checking against a golden model, causing the testbench to have linear memory usage. One method of resolving this for constant memory usage is to have the checks done per cycle, but it would require changes to the golden model to support cycle-by-cycle processing.

For another testbench, an AES RoCC Accelerator testbench is described in Section 4.1.3.

2.8 TileLink Test Components

Now that the core test components have been covered, we can take a look at the test components within the `verifTL` subproject, as it builds off the Decoupled VIPs. For a brief overview, TileLink is a SoC-level interconnect standard that connects multiple masters to multiple slave[2] devices, similar to AXI. It contains protocols for cache-coherent memory access and thus is a common interconnect standard used in Chisel SoCs. The basics of TileLink will be covered in the below sections, starting with the interface.

2.8.1 TileLink Interface

To introduce the interface, a TileLink connection consists of 5 logically independent channels that transfer messages sent between master and slave components. The diagram shown in Figure 2.1 displays the directionality of each channel.
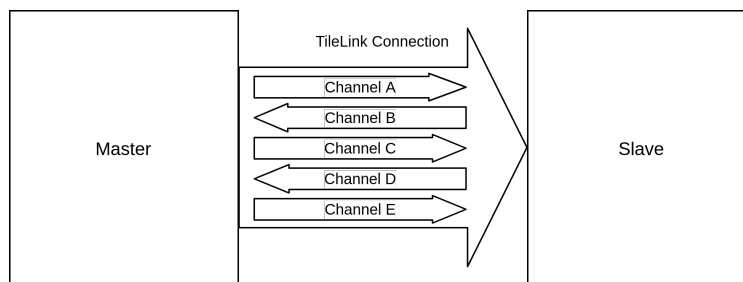
Figure 2.1: TileLink connection diagram.

Channels A and D are the core channels required to perform any type of memory operation. Channel A is used to transmit memory requests of a master, whether it be to

---

[2]Repeat of footnote 1: The outdated terms *master* and *slave* are used in this paper to avoid confusion and remain consistent with the TileLink spec, which currently uses the outdated terminology.

access/write data or to cache the data. Channel D is responsible for transmitting a response back to the master in the form of a data response or acknowledgement.

Channels B, C, and E are additional channels that enable the permission management of cached blocks of data. Channels B and C are similar to that of Channels A and D in the way that both Channel A and B carry requests while Channel C and D carry responses. The difference is the directionality, where requests are sent to the master. Channel E is responsible for carrying a final acknowledgement of a cache block transfer from the original requester (used for serialization).

Each channel, operating independently from each other, transmits a channel-specific `TLChannel` datatype over a decoupled interface. For example, Channel A transmits `TLBundleA` datatypes. A simplified code snippet of the TileLink connection is shown below:

```
1  class TLBundle(val params: TLBundleParameters) extends Record
2  {
3    def a: DecoupledIO[TLBundleA] = Decoupled(new TLBundleA(params))
4    def b: DecoupledIO[TLBundleB] = Decoupled(new TLBundleB(params).flip)
5    def c: DecoupledIO[TLBundleC] = Decoupled(new TLBundleC(params))
6    def d: DecoupledIO[TLBundleD] = Decoupled(new TLBundleD(params).flip)
7    def e: DecoupledIO[TLBundleE] = Decoupled(new TLBundleE(params))
8    ...
```

Listing 2.10: TLBundle definition - simplified.

Each `TLChannel` datatype contains fields that together describe a request or response. Below are two tables that briefly describe the fields of `TLBundleA` and `TLBundleD`. The semantics of each field will be discussed in the following sections.

TLBundleA	
Field	**Description**
opcode	Message type
param	Sub-opcode identifier
size	Log (base 2) of operation size
source	Master source identifier
address	Target byte address
mask	Byte lane select
data	Data payload
corrupt	Corrupt data indicator

Table 2.1: TLBundleA fields.

TLBundleD	
Field	**Description**
opcode	Message type
param	Sub-opcode identifier
size	Log (base 2) of operation size
source	Master source identifier
sink	Slave sink identifier
denied	Denied request indicator
data	Data payload
corrupt	Corrupt data indicator

Table 2.2: TLBundleD fields.

2.8.2 TileLink Protocol

When it comes to protocol compliance, there are two areas in which the TileLink messages must meet the spec. First, the fields within a message must be legal (e.g. the `opcode` must be a legal value for the channel, the `param` must be valid for the message type, etc.). Second, the ordering of messages must follow handshake protocols. For example, if a master requests data from a memory device, the device *must* respond with either the data or a response indicating that the request was denied.

There are three compliance levels in TileLink. From basic to complex, the first level of compliance is TL-UL, or TileLink-Uncached Lightweight, followed by TL-UH, or TileLink-Uncached Heavyweight, followed by the highest level of TL-C, or TileLink-Cached. Moreover, the levels of compliance build off the previous (i.e. TL-UH supports TL-UL, TL-C supports TL-UH and TL-UL). The following sections will briefly describe each compliance level[3].

2.8.2.1 TileLink-Uncached Lightweight

In TL-UL, there are two types of memory operations: `Get` (read) and `Put` (write). Moreover, there are two subtypes of `Put`: `PutFullData` and `PutPartialData`. `PutFullData` is used to write a contiguous block of data, while `PutPartialData` used to write arbitrary aligned data at a byte granularity. There are two response message types: `AccessAck` (no data payload), and `AccessAckData` (with data payload). The table below describes the handshake protocols:

Message (A)	Response (D)
Get	AccessAckData
PutFullData	AccessAck
PutPartialData	AccessAck

Table 2.3: TL-UL handshake protocols.

As these operations are uncached, they only travel on Channels A and D. Moreover, the data payload in messages for TL-UL can only span one `beat`, or a single `TLBundleA` or `TLBundleD` transaction. The maximum data payload size is determined by `beatBytes`, which is also related to the physical width of the TileLink connection. This connection parameter, along with others, are determined by both the parameters of the master and slave interfaces. More on this in Section 2.8.2.4.

2.8.2.2 TileLink-Uncached Heavyweight

TL-UH builds off of TL-UL by adding two more types of messages: `Atomic` operations and `Hint` operations. `Atomic` operations modify an existing value in memory via a arithmetic

---

[3]The following sections are meant to be quick introductions to the main components of the TileLink protocol — some details have been intentionally left out. For complete details, the entire TileLink spec can be found here[8].

or logical operation and return the old data value to the requestor. `Hint` messages notify the memory of an upcoming operation for potential performance optimization.

Moreover, TL-UH allows burst messages, or messages that span multiple `beats`. This enables single operations on data larger than the width of the physical bus, as it can now be split among multiple `TLBundleA` and `TLBundleD` messages. Note that all messages that involve data payloads can be burst messages. The table below describes the added handshake protocols in TL-UH.

Message (A)	Response (D)
ArithmeticData	AccessAckData
LogicalData	AccessAckData
Hint	HintAck

Table 2.4: Added TL-UH handshake protocols.

2.8.2.3 TileLink-Cached

TL-C is the highest conformance level, and it enables master agents to cache blocks of data. It adds a new set of `transfer` operations that create or remove cached copies of data blocks via the transfer of read or write permissions. There are three types of `transfer` operations: `Acquire`, which gives the requesting master read or write permissions on a cached block, `Release`, which the requesting master agent voluntarily gives up read or write permissions on a cached block, and `Probe`, which the requesting slave agent forcibly removes a master agent's read or write permissions on a cached block. Note that the specific message types will not be discussed here, as the details are not needed for the contents of this paper.

TL-C also introduces the 3 other channels: B, C, and E. As mentioned earlier, Channels B and C are mirrored versions of Channels A and D as they transmit requests from a slave agent to a master agent. These channels are used for `Probe` and `Release` messages, as well as any memory operation that needs to be forwarded to a master agent with a cached version of a block. As a result, all earlier messages discussed in TL-UL and TL-UH that travel on Channel A and D can now be sent via Channel B and C respectively. Channel E is solely used for `GrantAck`, or grant acknowledgement, messages. The table below describes the added handshake protocols in TL-C in simplified form (see spec[8] for more details).

Message (Channel)	Response (Channel)
Acquire (A)	Grant, GrantData (D)
Probe (B)	ProbeAck, ProbeAckData (C)
Release (C)	ReleaseAck (D)
Grant (D)	GrantAck(E)

Table 2.5: Added TL-C handshake protocols - simplified.

2.8.2.4 TileLink Parameters

The protocol conformance level of a TileLink connection is automatically determined by the parameters of the master and slave interface. Represented by a `TLMasterPortParameters` and a `TLSlavePortParameters` object, the master and slave interface parameters are used to create a `TLBundleParameters` object that defines the parameters of the entire TileLink connection (e.g. the wire widths of each field). Note that the bundle parameters are automatically created when a TileLink connection is formed via an automatic process called Diplomacy. These parameters are important, as they are used to create channel messages that are compatible with the specific TileLink connection. For example, the previous code snippet of the TileLink interface (Listing 2.10) shows `params` being passed into each `TLChannel` object.

2.8.3 TileLink Protocol Checker

As the protocol is critical in determining proper TileLink operation, a TileLink Protocol Checker VIP was implemented to provide light sanity checks on the transactions observed on the connection[4]. Note that it only checks the protocol of the messages (e.g. legal opcode, size, handshakes etc), but does not check data payload or logical correctness (e.g. expected data, repeated/conflicting `Release` requests, etc.). Moreover, the TileLink Protocol Checker supports all levels of conformance (TL-UL, TL-UH, TL-C).

The protocol checker has two main functionalities. First, it checks if the fields of all transactions are compliant with the TileLink parameters. Referencing the fields of a `TLBundleA` (Table 2.1) for example, the `opcode` needs to be a valid message type, the `size` must be within the maximum allowed message size, etc. Second, the protocol checker needs to ensure that all message handshakes are complete and that there are no unexpected transactions. For instance, after a `Get` request, there must be the correct number of `AccessAckData` responses with the same `source` ID.

To accomplish these two tasks, the protocol checker iterates through each transaction in order. First, depending on the message type, the transaction's fields are checked for compliance. Second, it checks and updates an internal hashmap that maps source ID to handshake progress to ensure that the current transaction is expected. For instance, if it sees a response transaction with a given source ID of 1, the protocol checker will ensure that the internal state is expecting a response transaction with a source ID of 1. Note that because the protocol checker has internal state, transaction status and progress are preserved across function calls. Moreover, the protocol checker can also be integrated with the `TLMonitor` for light sanity checks, as shown in Section 2.8.9.

---

[4]The library also contains another TileLink protocol checker that is created using the specification language API. While that variant currently only supports up to TL-UH compliance, it supports data correctness checking. More details on the SLTileLink protocol checker can be found in Section 4.2.1 — it is recommended to first read Chapter 3 for information on the specification language API.

2.8.4 TileLink Transactions

To create TileLink transactions, we will again utilize the `DecoupledTX` construct as each channel operates on the decoupled interface. The following code snippet is an example of creating a transaction for Channel A:

```
1  // Creating a TLBundleA object
2  val inputTLA = new TLBundleA(param).Lit(_.opcode -> 4.U,...)
3
4  // Transaction Prototype for TLBundleA
5  val txnProto = new DecoupledTX(new TLBundleA(param))
6  val inputTxn = txnProto.tx(inputTLA, 4, 5)
```

Listing 2.11: TLBundleA transaction example.

The setup is very similar to the earlier `DecoupledTX` example, except with an `TLBundleA` object is the data payload. On line 2, a `TLBundleA` object is defined and populated with specific fields. Next, on line 5, the transaction prototype to take in that specific `TLBundleA` datatype is defined. Finally, on line 6, the `tx` function is used to create a `DecoupledTX` with a data payload of `inputTLA`, `waitCycles` of 4, and `postSendCycles` of 5. Note that the `param` object above is a `TLBundleParameters` object, as explained in Section 2.8.2.4.

2.8.5 TileLink Transaction Generator

To generate TileLink transactions, the library contains a `TLTransactionGenerator` that can be configured to generate transactions of all protocol compliance levels. The user can select specific message types to be generated via the class signature shown in Listing 2.12.

```
1  class TLTransactionGenerator(
2    params: TLSlavePortParameters, bundleParams: TLBundleParameters,
3    overrideAddr: Option[AddressSet] = None,
4    get: Boolean = true, putFull: Boolean = true, putPartial: Boolean = true,
5    burst: Boolean = false, arith: Boolean = false,
6    logic: Boolean = false, hints: Boolean = false,
7    tlc: Boolean = false, cacheBlockSize: Int = -1, acquire: Boolean = false)
```

Listing 2.12: TLTransactionGenerator configuration options.

To generate transactions, the function `def generateTransactions(numbTxn : Int, permState: RWPermState)` is used to generate a sequence of `numbTxn` TileLink transactions. The `RWPermState` object is a state object that keeps track of read and write permissions of cached blocks and will only be used if TL-C is enabled. This is needed for TL-C as blocks can be cached by the master agent, which in this case is mocked the transaction generator. The `RWPermState` is used to ensure that generated TL-C transactions follow protocol (e.g. a generated `Release` message is releasing an existing cache block). Note that the `RWPermState` need not to be created or modified by the user, as it is managed by the `TLCFuzzer`.

2.8.6 TileLink-U and TileLink-C Fuzzer

Similar to the `Sequencer` object in UVM, the `TLUFuzzer` and `TLCFuzzer` VIPs are responsible for dispatching transactions to TileLink drivers to send to the DUT. Transactions can either be manually defined (e.g. a directed test) and/or be randomly generated (e.g. a random test) via a `TLTransactionGenerator`. Moreover, these fuzzers are compliant with the TileLink protocol. For example, if the manually defined stimulus only contained requests with the same `source` ID, the fuzzers will only dispatch one request at a time and will wait for the response before dispatching the next. To accomplish this, the fuzzers contain internal state that keep track of which `source` IDs are in flight. Additionally, the `TLCFuzzer` also keeps track of read and write permissions of cached blocks via a `RWPermState` object that is also passed into the transaction generator, as described in the previous section. To keep these internal states updated, the fuzzer requires access to the response transactions. As a result, most implementations will have a fuzzer paired with a `TLMonitor` such that the monitor that will forward all response transactions to the fuzzer.

In terms of usage, both fuzzers contain a method `def next(resp: Seq[TLChannel]): Seq[TLChannel]` that consumes transactions seen by the monitor and returns a new TileLink request to be driven to the DUT. Note that the returned transactions represent a complete request and may be multiple transactions in the case of burst messages. As mentioned above, the transactions returned by the fuzzer are either manually defined or randomly generated by the transaction generator. Transactions are only randomly generated when all of the directed stimulus is sent (i.e. directed stimulus has priority). Note that directed stimulus can be supplied to a fuzzer via the constructor or via `def addTxns(add: Seq[TLChannel]): Unit`.

2.8.7 TileLink Drivers

As the TileLink interface consists of 5 channels operating via the decoupled interface, the Tilelink drivers consist of 5 decoupled drivers. Moreover, as there is a mastering and slaving interface, there are two types of TileLink drivers as well. Starting with the TileLink driver on mastering interface, the simplified code implementation is shown below:

```
1  private val aDriver = new DecoupledDriverMaster(interface.a)
2  // Note: dDriver and bDriver just consume transactions blindly
3  private val dDriver = new DecoupledDriverSlave(interface.d, 0)
4  private val bDriver = if (interface.hasBCE) new DecoupledDriverSlave(interface.b, 0)
5  private val cDriver = if (interface.hasBCE) new DecoupledDriverMaster(interface.c)
6  private val eDriver = if (interface.hasBCE) new DecoupledDriverMaster(interface.e)
7
8  def push(tx: Seq[TLChannel]): Unit = {
9    // **Logic that filters transactions to the correct driver
10 }
```

Listing 2.13: TLDriverMaster implementation - simplified.

Separate decoupled drivers are instantiated with the directions corresponding to the master interface as seen in Figure 2.1. The full code can be found at Listing A.2.

As for the TileLink driver for the slaving interface, it is required to respond to memory requests sent by a master. To enable this, the driver utilizes a state monad, TLSlaveFunction, that has the following trait (abstract) definition:

```scala
trait TLSlaveFunction[S] {
  def response(tx: TLChannel, state: S): (Seq[TLChannel], S)
  def respondFromState(txns: Seq[TLChannel], state: S): (Seq[TLChannel], S) = {
    // ** Logic that batch process transactions using response()
  }
}
```

Listing 2.14: TLSlaveFunction trait - simplified.

Above, the trait is defined with a configurable datatype S that is used as the memory state to respond to TileLink requests, as shown in the functions response and respondFromState. The function response takes in a TileLink request tx and state and returns a Seq of response messages along with an updated state. The respondFromState is a wrapper for the response function to allow for batch processing. Note that the library has a TLMemoryModel VIP (Section 2.8.8) that implements the trait to respond to TileLink requests.

Below is a simplified code snippet describing how the TLSlaveFunction is used to respond to requests within the TLDriverSlave.

```scala
private val aDriver = new DecoupledDriverSlave...
private val dDriver = new DecoupledDriverMaster...
// ** B,C,E driver code not shown here
private val monitor = new TLMonitor...
var state = initialState

fork {
  while (true) {
    val txFromMaster = monitor.getMonitoredTransactions()
    // Generating response transactions and updating state
    val (responseTxns, newState) = slaveFn.respondFromState(txFromMaster, state)
    state = newState
    dDriver.push(responseTxns)
    clock.step()
  }
}
```

Listing 2.15: TLDriverSlave implementation - simplified.

On lines 1-3, separate decoupled drivers are instantiated with the directions corresponding to the slaving interface. There is also an additional TLMonitor defined on line 4 to forward requests to the slaving function. Then, within the fork defined on lines 7-15, the

`TLSlaveFunction` is used to respond to requests. Specifically, line 11 demonstrates how the `respondFromState` function is called, and lines 12-13 display how the state is updated and the response transactions are driven back to the master. The full code implementation can be found in Listing A.3.

2.8.8 TileLink Memory Model

The `TLMemoryModel` is a software-based golden memory model for uncached TileLink connections. As mentioned previously, it utilizes a state-monad and extends `TLSlaveFunction` with the function definitions listed in Listing 2.14. Under the hood, the memory model utilizes a state object with the definition: `case class State(mem: Map[WordAddr, Array[Byte]], burstStatus: BurstStatus)`. The `mem` object holding the memory state is a `Map` of word addresses to byte arrays of size `beatBytes` (a word). Instead of using byte addresses, the memory map uses word addresses because a TileLink transaction's data payload is naturally aligned to `beatBytes`, guaranteeing that no single beat will contain data that spans across multiple words in the memory map. Next, the `BurstStatus` object keeps track of any on-going burst messages, as the memory model processes one beat at a time.

As for read and write operations, the memory model updates the `State` object accordingly. Below is the implementation for a `Get` operation:

```
1  val byteAddr = txA.address.litValue()
2  val wordAddr = (byteAddr / bytesPerWord).toLong
3  val wordsToProcess = ceil(pow(2, txA.size.litValue().toInt) / bytesPerWord).toInt
4  ...
5  case TLOpcodes.Get =>
6    val responseTxs = (0 until wordsToProcess).map {
7      wordIdx => TLMemoryModel.read(state.mem, wordAddr + wordIdx, bytesPerWord,
         ↪   txA.mask.litValue().toInt, maskReads)
8    }.map {
9      word => AccessAckData(word, txA.size.litValue().toInt, txA.source.litValue().toInt,
         ↪   denied=false)
10   }
11   (responseTxs, state)
```

Listing 2.16: TLMemoryModel "Get" implementation.

On lines 1-3, the transaction's byte address is converted into a word address and the number of words to read is calculated. Then, on lines 6-10, the response transactions are generated via nested map statements, with the first map (line 7) reading in the data word via `TLMemoryModel.read` and the second map (line 9) packaging it into a `AccessAckData` message. The response transactions are returned along with the unchanged state, as this is a read operation. The implementation for write operations, however, is more complex due to beat requests. Although it is not covered, the code is shown in the appendix in Listing A.4.

2.8.9 TileLink Monitor

Like all monitors, the TileLink monitor's function is to observe all transactions on a TileLink interface. As there are five channels, there will be five `DecoupledMonitor` objects:

```
1  private val aMonitor = new DecoupledMonitor...
2  private val bMonitor = new DecoupledMonitor...
3  ... // ** Monitors for C, D, and E are instantiated the same way
4
5  def getMonitoredTransactions(): Seq[DecoupledTX[TLChannel]] = {
6    val tx = //* Logic that collects all transactions from monitors and sorts by cycleStamp
7    if (protocolChecker.isDefined) {protocolChecker.get.check(tx.map {_.data})}
8    tx
9  }
```

Listing 2.17: TLMonitor implementation - simplified.

The main difference for the TileLink monitor is that it can be instantiated with an optional `TLProtocolChecker` that will check the observed transactions for protocol compliance. Note that this check is only performed when `getMonitoredTransactions` is called. See Section 2.8.3 for more information on the TileLink Protocol Checker. The complete code implementation can be found in Listing A.5.

2.8.10 TileLink Standalone Blocks

In order to test TileLink modules standalone (e.g. in a unit test), additional wrapper classes that connect to the dangling TileLink endpoints must be used, as TileLink connections require both sides of the interface to be defined. In this library, they are defined as standalone blocks. The listing below displays a simplified version of `TLRAMStandalone`:

```
1  class TLRAMStandalone (val mPortParams: TLMasterPortParameters, ...
2    ) (implicit p: Parameters) extends LazyModule {
3    val ram = LazyModule(new TLRAM(...))
4
5    // Connect TLRAM's endpoint to Bridge object that can be poked/peeked
6    val bridge = BundleBridgeToTL(mPortParams)
7    val ioInNode = BundleBridgeSource(() => TLBundle(TLBundleParameters(mPortParams,
     ↪  ram.node.edges.out.head.slave)))
8    ram.node := bridge := ioInNode
9
10   // Exposing IO
11   val in = InModuleBody { ioInNode.makeIO() }
12   lazy val module = new LazyModuleImp(this) {}
13 }
```

Listing 2.18: TLRAM standalone block implementation - simplified.

The important code to focus on are on lines 6-8. In line 6, a `BundleBridgeToTL` object is instantiated with the given `mPortParams`; its function to mock the master interface that will connect to TLRAM's slave interface. Then, on line 7, we define an endpoint node, `BundleBridgeSource`, such that the `BundleBridgeTL` can be poked and peeked by TileLink VIP. Note that the endpoint node is configured by a `TLBundleParameters` object that is automatically generated from the master and slave port parameters. On line 8, the components are then connected together. Line 11 then exposes the `BundleBridgeSource` as an IO port. In the perspective of the `TLRAM`, it sees that it is fully connected to a TileLink master device but in reality it would be connected to a TileLink test component. Figure 2.2 displays a visual diagram of the components in the `TLRAMStandalone` block. Note that the master and slave interface parameters are defined by the DUT/user, while the bundle parameters are automatically generated.
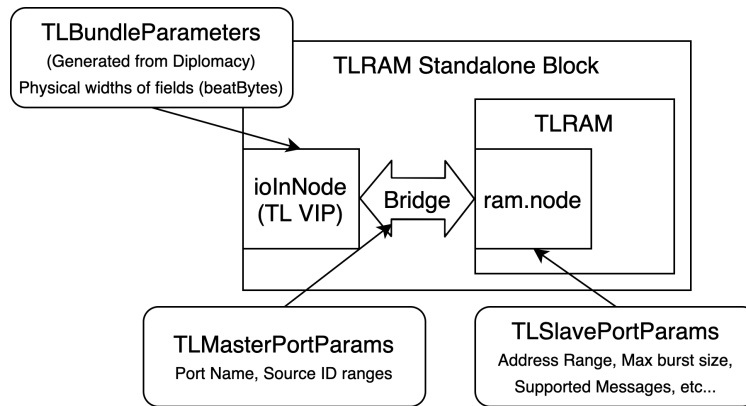
Figure 2.2: TLRAM standalone block diagram.

2.8.11   Application Examples

Applications using the TileLink test components can be found in Section 4.1, including a full testbench for the `TLRAM` module, the `InclusiveCache` (`L2`), as well as an AES RoCC accelerator. The examples will describe in more detail on how the TileLink test components interact with each other to create a fully functional testbench.

Chapter 3

SVA-like Specification Language as an eDSL

3.1 Introduction

3.1.1 Motivation

The motivation of creating a SVA-like specification language API came about when we were creating a testbench to verify an L2 cache. With the standard UVM-like testbench setup described in the previous chapter, we would need a golden SW model of the L2 cache for correctness checking. However, such a model would have to be designed to mimic the specific internals of the L2 cache (e.g. the uArch). The model would only be useful for that specific cache, and it would go against the original direction of implementing infrastructure to verify *any* L2 cache. One alternative would be to have a generic cache golden model that generates a relative/partial ordering of transactions that could be reordered to match specific uArch designs, but it would increase the testing complexity substantially.

Instead of defining a cycle accurate SW model of what should happen on the interface, we would prefer a method of systematically describing valid behaviors to a checker function that could check if an output trace violates the valid behaviors. Moreover, we would want to describe a language for specifying these valid behaviors, called properties. This language should be portable across cycle-level properties (e.g. ensuring that the data is not changed when valid is high until ready is asserted) and transaction-level properties (e.g. after a `Get` request, there is an `AccessAckData` response with the same `source` ID).

In industry, there is already a language that does this: SystemVerilog Assertions, or SVA. In brief, it allows users to describe the behavior of a design through assertions on specified properties. SVA is used mostly everywhere in industry, and it is very familiar among verification engineers. However, similar to the reasons of creating test components in Chisel for a unified test and DUT environment, we would prefer an SVA-like specification language that is embedded in Scala. The following sections will first briefly introduce SVA in more detail and then introduce SpecLang, the SVA-like specification language API for Scala.

3.1.2 SystemVerilog Assertions (SVA)

SystemVerilog Assertions is a language that allows designers to verify their design using assertions that represent properties of the design. In terms of assertions, there are two types: immediate and concurrent. Immediate assertions are similar to an `if` statement, with the difference that if the statement is false, the assertion fails and the simulator reports the error. For example, an immediate assertion `assert (A == B)` states that A must be equal to B. Note that the immediate assertion is evaluated like a procedural statement and follows code ordering for execution.

Concurrent assertions, on the other hand, enable users to write complex expressions that support temporal restrictions based on a clock edge. The complex expression, indicated by a `property`, is built using a `sequence` that describes the checked sequential logic. Take the following SVA assertion statement as an example:

```
1  assert property (@(posedge Clock) Req |-> ##[1:2] Ack);
```

Listing 3.1: SVA property example.

Here, the `property` is checking for the sequential behavior described by the `sequence`: `Req |-> ##[1:2] Ack`. Breaking it down, both `Req` and `Ack` are boolean expressions (e.g. they return true if a request or acknowledgement is seen), `|->` is the implication operator, and `##[1:2]` is a temporal operator used to denote temporal restrictions. Note that the `@(posedge Clock)` qualifier means that the values are sampled at every clock edge. Putting everything together, the `sequence` is describing that after every request, there *must* be a acknowledgement after 1 to 2 cycles. For example purposes, let's simplify `Req` and `Ack` to boolean functions that check if its corresponding signals are high. Figure 3.1 displays a few waveforms of traces that pass and fail the above assertion.

(a) Traces that pass property.   (b) Traces that fail property.
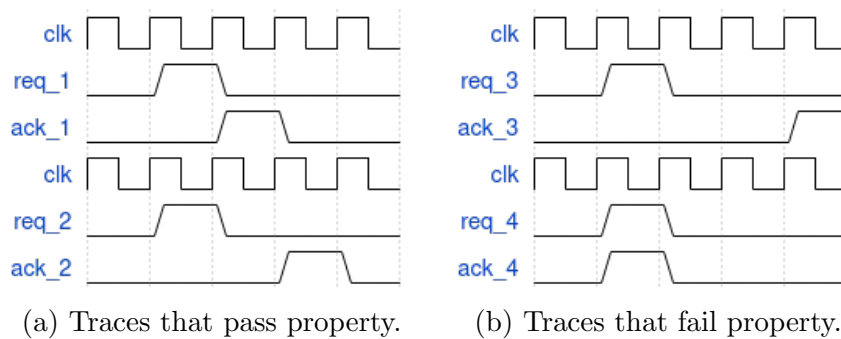
Figure 3.1: Example waveforms for Req-Ack property.

The two traces on the left pass, as the ack signals are high within the expected boundaries (1 to 2 cycles). The traces on the right fail as the `ack` signals are too late and too early,

respectively. Although this example is very simplified, the SVA constructs described are generalizable and allow designers to describe arbitrary automata to be asserted.

In terms of assertion statements, there are four types: `assert`, `cover`, `assume`, and `restrict`. The `assert` type, as shown above, is used to check if the given property is true. The `cover` type is similar to `assert` in that it can take in a property, but instead of checking the correctness of the property, it monitors property evaluation for coverage. The `assume` and `restrict` types are mainly used for formal tools and not simulators, so it is not covered in this introduction.

Overall, SVA is a mature design specification language and is used everywhere in industry. Although not mentioned here, SVA contains many features not only specific to test simulators but also to formal tools as well. To read more about SVA and all of its features, the IEEE standard for SystemVerilog can be found here[9], with SVA covered in Chapter 16.

3.1.3 Specification Language API (SpecLang)

The Specification Language API, or SpecLang for short, is a SVA-like specification language embedded in Scala. Note that at the time of writing, this project is in its early stages, and that only a subset of SVA functionalities are implemented. That being said, SpecLang has a workable set of functions that enable users to define any property to be asserted on a given output trace. Moreover, the language contains a modeling layer that enables per-property local variable support as well as global memory model support. In terms of coverage, each property collects basic statistics such as how many times the property passed and failed. One thing to note is that the SpecLang constructs are currently not synthesizable, but it is something that to be implemented in the future.

This chapter will describe the syntax of SpecLang as well as the different components in detail. For applications of SpecLang, see Section 4.2 for how it is used to re-define the TileLink protocol checker and check for malicious mutations of a `TLRAM` output trace. Note that there was not have enough time to use SpecLang to test the L2, but the application with the `TLRAM` demonstrates that it can be extended to test the L2.

3.2 Abstract Syntax Tree

Below is the abstract syntax tree (AST) for SpecLang. Each asterisk (*) describes a SpecLang object, each dash (-) describes methods of the object, and each plus (+) describes ways of combining other SpecLang objects to create the specified object.

```
Note: T -> Transcation DataType, M -> Memory State DataType

* AtmProp/AP: Unit Boolean Function
    - check(T, HashMap, Option[SLMemState]) => Boolean
    + AP & AP => AP
    + AP | AP => AP
* TimeOp/TO: Time Constraint Function
    - check(Cycles) => Boolean
    - invalid(Cycles) => Boolean
* Implication/Imp: Enables Assertion Checking within Sequence
* PropSet: Condensed Representation of AP+TO or Imp within Sequence
    - check(T, HashMap, Option[SLMemState]) => Boolean
    - invalid(CycleStamp, LastPassedCycle) => Boolean
    - isImplication => Boolean
    - isIncomplete => Boolean
* Sequence: Collection of PropSet objects
    + AP + AP/TO/Imp => Sequence
    + Sequence + AP/TO/Imp => Sequence
    + Sequence + Sequence => Sequence
    + Sequence * # => Sequence
* Property: Enables checking of Sequence on transaction trace
    - check(Seq[T], Seq[SLMemState]) => Boolean
* SLMemoryState/SLMemState: Memory State from Global Model
    - get(Int) => M
* SLMemoryModel/SLMemModel: Global Memory Model
    - model(Seq[T]) => Seq[Option[SLMemState]]
```

Listing 3.2: SpecLang AST.

3.3 Sequence Elements

A sequence element is a SpecLang object that creates `Sequence` objects when combined with other sequence elements. So far, they are `AtmProp`, `TimeOp`, and `Implication`. Note that these objects are not directly used in `Sequence` objects, as a condensed form is used instead. See Section 3.4 for more information on the implementation of `Sequence` objects.

3.3.1 Atomic Propositions

Atomic Propositions (`AtmProp`) are one of the base building blocks for creating sequences, as they allow the user to specify correct behavior via a boolean function. The user-defined boolean function takes in a transaction, a hashmap for local variables, and an optional `SLMemoryState` object for global memory state. The hashmap is a map from `String` -> `H`, where `H` can be any datatype, and its purpose is to allow data sharing between atomic propositions within the same sequence. On the other hand, the `SLMemoryState` object is used to define a static global memory state for each transaction and is visible to all atomic propositions operating on that transaction.

The signature for an atomic proposition is `AtmProp[T,H,M](proposition: (T, HashMap[String,H], Option[SLMemoryState[M]]) => Boolean, desc: String)`. The parameter `proposition` holds the boolean function that the atomic proposition checks for, or matches on, and the `desc` is a string description of what the proposition is checking. Note that each atomic proposition is concretized to `[T,H,M]`, or the transaction, hashmap, and memory state datatypes. Moreover, there is a shorthand definition `qAP[T,H,M](proposition: (...), desc: String = "Default")` that utilizes the `sourcecode` package[10] to automatically set the description of the proposition to the variable name as default. For example, the below code demonstrates how to instantiate an atomic proposition that matches if a given `TLChannel` object is a `Get` request and stores the transcation's `source` field into the hashmap. This value stored in the map can then be read by a successive atomic proposition in the same sequence, as shown in later examples.

```
1  def IsGetOpFn(t: TLChannel, h: HashMap[String, Int], m: Option[SLMemoryState[UInt]]) =>
↪  Boolean {
2    t match {
3      case t: TLBundleA =>
4        h("source") = t.source.litValue()
5        t.opcode.litValue() == TLOpcodes.Get
6      case _ => false
7  }
8  val IsGetOpAP = qAP(IsGetOpFn) // Description of AP will be "IsGetOpAP"
```

Listing 3.3: Atomic proposition for TileLink "Get" request.

With regards to checking if the atomic proposition is matched, there is a method `check(t: TLChannel, h: HashMap[String, Int], m: Option[SLMemoryState[UInt]])` that calls and returns the boolean of the proposition function.

As atomic propositions are essentially boolean functions, boolean operators `&` and `|` can be used to combine multiple atomic propositions into one. For instance, continuing off the earlier example, if there were another atomic proposition that checked if a TileLink message's `param` field was 0, the `&` operator can be used to combine them and create a new atomic proposition that is true if the message is a `Get` request with a `param` field of 0.

3.3.2 Time Operators

Time operators (`TimeOp`) introduce timing constraints for atomic propositions within a sequence. The object signature is as follows: `TimeOp(lowerCycles: Int, upperCycles: Int)`, where `lowerCycles` is used to define the lower limit of cycles and `upperCycles` is used to define the upper limit of cycles. For open-ended constraints (i.e. at least, at most), -1 can be used as a wildcard for the lower or upper limit. In terms of shorthand definitions, there are two options: `###(lower, upper)` and `###(exact)`. Note that in SpecLang, it is assumed that each transaction in the trace corresponds to one cycle.

To see `TimeOp` objects in action, let's continue the TileLink Get Request code example shown above in Listing 3.3. The below code example creates a sequence that specifies a complete TileLink Get handshake:

```
1  def IsAckDataOpFn(t: TLChannel, h: HashMap[String, Int], m: Option[SLMemoryState[UInt]])
↪    => Boolean {
2    t match {
3      case t: TLBundleD =>
4          t.opcode.litValue() == TLOpcodes.AccessAckData &&
5            t.source.litValue() == h("source")
6      case _ => false
7    }
8  }
9  val IsAckDataAP = qAP(IsAckDataOpFn)
10
11 // IsGetOpAP is from earlier example
12 val GetHandshakeBasicSeq = IsGetOpAP + ###(1,-1) + IsAckDataAP
```

Listing 3.4: Time operator example: TileLink "Get" handshake sequence.

In the above code, lines 1-9 defines another atomic proposition that checks if the current transaction is a `AccessAckData` message that has the same `source` field as the preceding `Get` request. Line 12 then defines a sequence that is matched when after a `Get` request, an `AccessAckData` message with the same `source` field is observed. If the time operator was not included, then both atomic propositions would try to match in the same cycle. Moreover, note that this sequence does not assert anything, as it does not contain an `Implication`. See the following section for more information.

To check if a time operator is matched, the method `check(elapsedCycles: Int)` returns true if the elapsed cycles are within the given bounds. However, as a time operator that mismatches on a given cycle can still match on a future cycle (e.g. an "at least" operator), there is an additional method `invalid(elapsedCycles: Int)` that returns true if the time operator can never meet the time requirement (i.e. `elapsedCycles > upperCycles`).

3.3.3 Implications

Implications (`Implication`) signify that the subsequent sequence elements must checked and asserted if and only if the preceding sequence elements were successfully matched. When all preceding conditions are met, the implication has been *activated*.

In terms of object signature, there are no parameters. The instantiation of an `Implication` object is `new Implication`. Note that there is also a shorthand `Implies` that does the same thing. The example below extends the previous example by enabling assertions checking in the Sequence via an `Implies`:

```
1   // See previous code examples for function implementations
2   def IsGetOpFn(...) => Boolean {...}
3   def IsAckDataOpFn(...) => Boolean {...}
4
5   val IsGetOpAP = qAP(IsGetOpFn)
6   val IsAckDataAP = qAP(IsAckDataOpFn)
7
8   val GetHandshakeBasicSeq = IsGetOpAP + Implies + ###(1,-1) + IsAckDataAP
```

Listing 3.5: Implication example: TileLink "Get" handshake sequence pt. 2.

In the above code, the added `Implication` denotes that after a `Get` message, there *must* be a `AccessAckData` message with the same `source` field that follows in at least 1 cycle. If the entire sequence is not matched, an assertion will be thrown.

3.4 Sequences

The `Sequence` object is used to describe and match sequential behaviors, as seen in the above example with the TileLink `Get` handshake sequence. It is created by combining any number of atomic propositions, whether via the addition operator (+) or via the sequence constructor: `new Sequence[T,H,M](input: SequenceElement*)`. Note that there is a shorthand to creating sequences: `qSeq[T,H,M](input: SequenceElement*)`. Moreover, sequences can be created via other sequences via either the addition operator (+) or the static repetition operator (∗), which is a wrapper of the + operation:

```
1   // + Operator with Sequence Elements
2   val seq1 = AP1 + Implies + ###(1, -1) + AP2
3   // Sequence Constructor
4   val seq2 = qSeq(AP3, ###(1, -1), AP4)
5
6   // + Operator with Sequences
7   val combSeq1 = seq1 + seq2 // AP1 + Imp + ###(1, -1) + AP2 + AP3 + ###(1, -1) + AP4
8
9   // * Operator with Sequences
10  val combSeq2 = seq1 * 2 // AP1 + ###(1, -1) + Imp + AP2 + AP1 + Imp + ###(1, -1) + AP2
11
12  // Nested operators supported
13  val combSeq3 = (AP1 + Implies + ###(1, -1) + AP2) + (AP3 + ###(1, -1) + AP4) * 2
```

Listing 3.6: Sequence creation and operators.

While sequences objects are created via sequence elements, the underlying implementation utilizes a list of `PropSet` (proposition set) objects, where each `PropSet` is a condensed representation of the sequence elements. Each `PropSet` object groups together atomic propositions that share a common time operator to simplify sequence matching as well as to reduce

the number of elements to check. More information on `PropSet` objects is found in the following section.

3.4.1   Proposition Sets (PropSet)

`PropSet` objects are used as condensed representations of sequence elements within a sequence. Users should not need to directly instantiate them during normal use, but they can be constructed via the constructor: `new PropSet[T,H,M](ap: AtmProp[T,H,M], to:` `TimeOp, implication: Boolean = false, incomplete: Boolean = false)`. A `PropSet` object is defined by an atomic proposition and a time operator that are checked together during matching. However, there are additional flags and indicators for various types of `PropSet` objects: the `implication` field in the constructor is used to denote empty `PropSet` objects that hold the original implication ordering relative to the other sequence elements, and the `incomplete` flag is to indicate that a `PropSet` object does not have an atomic proposition defined yet (created when appending a `TimeOp` to an existing sequence).

The intended use of `PropSet` objects is to group atomic propositions that share the same time operator for easier sequence matching. Take the abstract code in Listing 3.7 as an example. Looking at the sequence element definition on line 2, there needs to be at least 9 match operations in order for the entire sequence to be matched (one for each sequence element). Moreover, in the case where multiple atomic propositions are dependent on a time operator (e.g. AP2, AP3, AP4 all share the TO1 time constraint), multiple checks will be repeated on failure. For instance, if TO1, AP2, and AP3 match but AP4 mismatches on a given cycle, in the next cycle TO1, AP2, and AP3 would have to be checked again along with AP4 to ensure that they are still matched. To simplify this matching process, `PropSet` objects are used to group atomic propositions that share the same time operator such that they can be all checked at the same time. Line 5 displays how the sequence elements are grouped into `PropSet` objects. In this case, the minimum number of match operations decreases to 4, and in the case where a match fails on a cycle, only that `PropSet` would have to be checked again on the next cycle.

```
1  // User definition (9 total sequence elements)
2  val seq = AP1 + Implies + TO1 + AP2 + AP3 + AP4 + TO2 + AP5 + AP6
3
4  // PropSet grouping (4 total PropSet objects)
5  val seq = (AP1) + (Implies) + (TO1 + AP2 + AP3 + AP4) + (TO2 + AP5 + AP6)
```

Listing 3.7: PropSet grouping visualization.

In terms of checking whether a `PropSet` is matched, there are two methods `check()` and `invalid()` that returns true if the `PropSet`'s atomic propositions match on the current cycle or if the `PropSet`'s time operator is invalid, respectively. Note that for `check`, the time operator is matched first such that the function can short circuit and prevent the check for the atomic proposition to occur. This is because atomic propositions could have state changes

via local variables (e.g. counters) that should only be updated when the time operator is matched. However, this does not prevent state updates if the atomic proposition itself fails — it is the user's responsibility to ensure that the state change is gated by the proper conditional statement. Note that for implication `PropSet` objects, `check` always returns true and `invalid` always returns false.

3.5   Properties

A `Property` object is used to check whether a given transaction trace adheres the sequential behavior encoded within a `Sequence` object. It is instantiated with either the following constructor `new Property[T,H,M](seq: Sequence[T,H,M], name: String)` or the shorthand `qProp[T,H,M](input: Sequence[T,H,M], desc: String = "Default")` that utilizes the `sourcecode` package[10] to automatically set the description to the variable name as default, similar to `qAP` for atomic propositions. With a `Property` object, the `check(input: Seq[T], mems: Seq[Option[SLMemoryState[M]]] = Seq())` method can be called in order to check whether the given transaction trace `input` matches the sequential behavior specified by the `Property`'s `Sequence`. Note that the `check` method also takes in a sequence of optional `SLMemoryState` objects that define global memory states at each transaction cycle and are used in atomic proposition matching (see Section 3.6 and Section 3.6.1). Moreover, `Property` objects automatically collect primitive coverage data that provide an overview on sequence match successes and failures. See Section 3.5.1 for more information on property coverage collection. The below abstract code briefly demonstrates the creation and usage of a `Property` object.

```
1  // Creating Sequence
2  val seq = AP + Implies + TimeOp + AP + ...
3
4  // Creating property with above sequence (via shorthand)
5  val prop = qProp[T,H,M](seq)
6
7  // Check transaction trace
8  // Note: Memory states are generated via memory model (see later section)
9  prop.check(transactionTrace, Some(memoryStates))
10
11 // Print coverage
12 prop.getCoverage()
```

Listing 3.8: SpecLang property example.

To correctly match all occurrences of the sequence within a transaction trace, each `Property` object contains internal state to keep track of all concurrent property instances that occur from overlapping sequences. A property instance, or a pending/ongoing sequence match, is initiated when a transaction matches the first `PropSet` of the sequence. When this

occurs, a new property instance, along with other data such as its local variable hashmap, is created and kept track in the internal state.

In terms of the checking process, the `check` method iterates through the transaction trace and checks if each transaction creates a new property instance or matches an existing incomplete property instance. As it is possible for a transaction to initiate a new property instance and match an existing one at the same time, the transaction will be checked against existing instances first. If a matched transaction completes a property instance (i.e. all `PropSet` objects were matched), the completed property instance and its corresponding hashmap are removed from the local state. If the transaction fails to match on an existing property instance, the function will also check to see if the `PropSet` is invalid (missed time constraint). If it is invalid, the instance and corresponding hashmap are removed from the internal state. A failure is noted if the removed instance had an activated implication.

When the function is finished iterating through each transaction, each incomplete property instance is checked for activated implications. Activated implications within incomplete instances are noted as failures as the implication was never met.

3.5.1   Coverage Collection

As mentioned earlier, each `Property` object collects primitive coverage on sequence matches and failures. During transaction trace checking, the `Property` object keeps track of how many times each `PropSet` object within a sequence was correctly matched or completely failed (invalid). From this information, we can extract additional statistics such as how many times the property was activated, how many of the activated properties passed or failed, as well as the per-`PropSet` object pass/fail rate. Moreover, additional transaction coverage data is collected by marking which transactions successfully matched a `PropSet` object. Below is a simplified example of using coverage:

```
1   // Creating property that checks if all Get messages have a zero param field
2   val getParamSeq = IsGetOP + Implies + ParamZero
3   val getParamProp = qProp[...](getParamSeq)
4
5   // Input transactions pseudocode
6   val input = Seq(Get, Get, Put, Put, BadGet, Get)
7   getParamProp.check(input) // Will fail check
8   getParamProp.getCoverage()
```

Listing 3.9: Coverage example code - simplified.

In the above code, line 3 defines a sequence that describes that all TileLink `Get` messages must have a `param` field of zero. Line 7 then defines the input trace with a malformed `Get` message with a non-zero `param` field, named `BadGet`. Once the trace has been checked via line 8, the coverage is printed on line 9 with the actual output shown in Listing 3.10. The output displays that there were 4 activated properties (4 `Get` messages) but one of them failed

the check. It also lists which AP Group (`PropSet`) it failed on, along with the transaction coverage and bitmap.

```
PROPERTY COVERAGE DATA (Property getParamProp):

# of Activated Properties: 4
# of Completed Properties: 3
# of Failed Properties: 1

Coverage stats of each AP Group: (PASS #, FAIL #)
AtmProp: If is Get request : (4, 0)
AtmProp: Param field is zero : (3, 1)
Transaction coverage: (4/6)
Transaction coverage bitmap:
 [1, 1, 0, 0, 1, 1]

END OF COVERAGE REPORT.
```

Listing 3.10: Coverage output.

3.6   Memory Model

The `SLMemoryModel` is responsible for generating `SLMemoryState` objects that correspond to global memory states for each transaction in the trace. As the memory model implementation is dependent on the given design, the library only gives a skeleton as to what the memory model should look like; the implementation will be entirely up to the user. The object `trait` definition is as follows:

```
trait SLMemoryModel[T,M] {
  def model(input: Seq[T]): Seq[Option[SLMemoryState[M]]]
}
```

Listing 3.11: SLMemoryModel trait.

The memory model is only required to have a method `model` that takes in the input transaction trace and outputs the corresponding `SLMemoryState` objects at each transaction. Note that `T` is the transaction datatype while `M` is the memory state datatype. For a real application of a `SLMemoryModel` object, see Section 4.2.2 for an example of a TileLink memory model that is used to test the `TLRAM` module.

3.6.1   Memory State

Similar to the memory model, the `SLMemoryState` objects are largely dependent on the design as well as the intended use case. As such, the skeleton is defined as:

```scala
trait SLMemoryState[M] {
  def get(addr: Int): M
}
```

Listing 3.12: SLMemoryState trait.

The only requirement is that the state must have a `get` function that takes in an integer address and returns data of type `M`. For a quick example, the following code is an implementation of a generic `UInt` memory state:

```scala
class SLUIntMemoryState(init: HashMap[Int, UInt]) extends SLMemoryState[UInt] {
  // Non-destructive
  val int_state = init.clone()

  def get(addr: Int): UInt = {
    int_state(addr)
  }
}
```

Listing 3.13: Generic UInt SLMemoryState example.

The underlying data structure used to store state is a `HashMap[Int, UInt]`. For an application of a `SLMemoryState` object, a TileLink example can be found in Section 4.2.2.

3.7   Application Examples

Although the current state of SpecLang offers a limited subset of SVA features, it consists of a workable set of functions that can be used to define arbitrary properties that can be asserted on any given output trace. Section 4.2 describes how SpecLang was used to redefine the TileLink protocol checker, and how it was used to detect incorrect `TLRAM` output transaction traces.

Chapter 4

Library Applications

4.1 Library Test Components

Throughout its development, the library's test components have been used to verify components ranging from TileLink memory modules to custom accelerators and components.

4.1.1 TLRAM Verification

The `TLRAM` module is a Chipyard default RAM design for the TileLink interface. As it is a TileLink component, it is configured by the interface parameters: address space (also used to determine size), set of legal operations, set of legal parameters, etc. Outside of TileLink interface parameters, it also contains robustness features such as ECC functionality. In terms of operations, it is able to support all operations defined in the TL-UH protocol (see Section 2.8.2.2). To verify the `TLRAM` module, we only focused on the protocol and data correctness of the output — other features such as ECC functionality were not verified.

4.1.1.1 Test Setup

In terms of test components, Figure 4.1 displays the testbench setup. Starting from the top left, the `TLU-Fuzzer` is responsible for sending stimuli (directed or randomly generated) to the `TLDriver`, which in turn drives the transactions to the DUT. Another copy of the stimuli is also sent to the `TLMemoryModel`, which will produce the expected output used for output correctness checking later on. At the same time, two `TLMonitor` objects observe the transactions on the TileLink connection between the driver and the DUT. The left monitor forwards the transactions to the fuzzer while the right one records transactions for output checking. Note that only one of the monitors contains a protocol checker (PC). Once all of the stimuli has been driven and all response transactions are observed, the entire DUT transaction trace is checked for protocol compliance and compared with the output of the golden model for correctness.

Input Stimulus

In terms of input stimulus, both directed and randomly generated stimuli were used. The directed tests were used to sanity check basic functionality such as handling burst messages and special operations, and the random tests were to verify overall correctness by introducing a large number of stimuli.
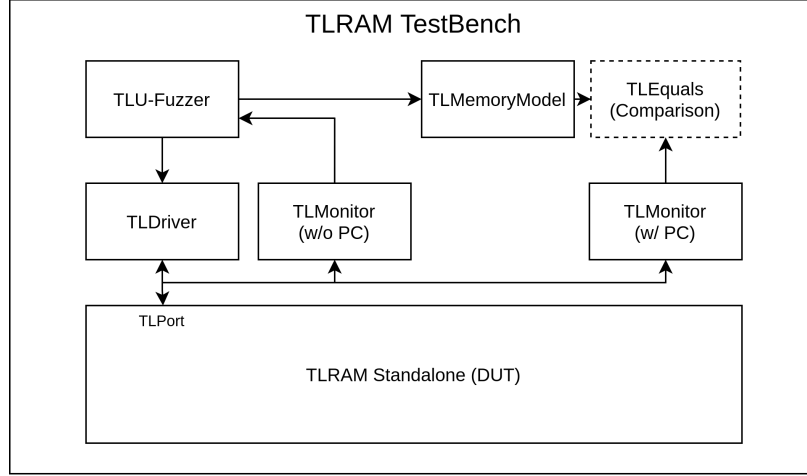
Figure 4.1: TLRAM test setup.

Test Coverage

To ensure that the DUT was properly tested, RTL coverage, a measurement of the design RTL exercised by the test, was collected across the various directed and random tests to check for completeness. Merged coverage across all directed and random tests resulted in 93.7% Line, 95.88% Condition, 66.93% Toggle, and 96.23% Branch coverage, resulting in an overall score of 88.18% coverage. More details on the coverage collection for the `TLRAM` tests are described in Section 4.1.4.

4.1.2   SiFive's Inclusive Cache (L2) Verification

SiFive's `InclusiveCache` is an open-source L2 cache for the TileLink interface, and it is commonly used in RocketChip designs. It has the highest TileLink protocol compliance as it supports all operations defined in the TL-C protocol (see Section 2.8.2.3). In terms of configurability, the `InclusiveCache` is parameterized via TileLink interface parameters as well as a custom `CacheParameters` object that contains cache related details such as number of ways and sets as well as the size of each cache block. For verification, we only focused on the TileLink protocol correctness of the output — cache uArch protocol and data correctness were not checked. See note at the end of the section for more information.

4.1.2.1   Test Setup

In terms of test setup, it is similar to that of the `TLRAM` testbench with the exception that the `InclusiveCache` has two TileLink interfaces: one on the TL-C protocol for the L1 cache and another on the TL-UH protocol for the backing DRAM memory as shown in Figure 4.2. Starting with the L1 port, a `TLDriver` drives transactions from the `TLC-Fuzzer` to the L2. At the same time, two `TLMonitors` record all transactions on the L1-L2 interface — one
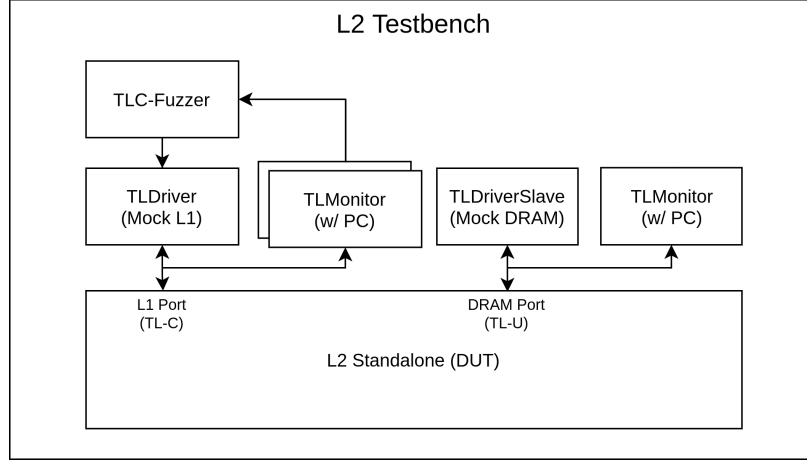
Figure 4.2: SiFive's InclusiveCache (L2) test setup.

monitor (without protocol checking) is used to forward L2 requests to the `TLC-Fuzzer` to determine responses (e.g. if L2 sends a probe request, L1 needs to respond) and the other monitor (with protocol checking) is used to check for protocol compliance.

The configuration on the DRAM port consists of a `TLDriverSlave`[1] to mock the backing DRAM memory and a `TLMonitor` to record and check for protocol compliance at the end of the test. Moreover, the memory state within the `TLDriverSlave` can be initialized with custom values by the user before the test is run.

Input Stimulus

Similar to the `TLRAM` tests, both directed and randomly generated stimuli were used. The directed tests were used as sanity checks to ensure that the cache is able to correctly process requests such as loading cache blocks from DRAM, writing back dirty blocks, revoking cache block permissions, etc. Once the directed tests were verified, randomized tests were used to cover a bigger breadth of operation in terms of addresses and message ordering. Note that data correctness and L2 uArch specific protocol compliance were not checked, and only the TileLink protocols (TL-C for L1 and TL-U for DRAM) were checked.

Test Coverage

RTL coverage was also collected across the various directed and random tests to check for completeness. Merged coverage across all directed and random tests resulted in 77.54% Line, 82.24% Condition, 39.27% Toggle, and 79.45% Branch coverage, resulting in an overall score of 69.92% coverage. The total coverage is much lower compared to that of the `TLRAM` tests, and we suspect it is due to the much bigger design and complexity of the `InclusiveCache` in

---

[1]Repeat of footnote 1 from Chapter 2: The outdated terms *master* and *slave* are used in this paper to avoid confusion and remain consistent with the TileLink spec, which currently uses the outdated terminology.

that both the directed and randomized tests together are unable to exercise a considerable portion of the RTL. Due to limited time and shift of focus towards the development of the specification language API, more comprehensive tests for the `InclusiveCache` were left as future work.

L2 uArch Protocol and Data Correctness

The reason for not verifying the uArch protocol and data correctness is that it would have required a golden model that essentially re-implemented the cache in software. A model with this detail would be specifically designed for this cache and would not be scalable to other cache designs with a different uArch. Instead, we directed our effort into developing a general specification language API in Scala (SpecLang) that would be easier to represent uArch details (see Chapter 3). Although we did not have the time to implement the L2 testing using SpecLang, we were able to re-verify the `TLRAM` module as described in Section 4.2.2.

4.1.3 AES RoCC Accelerator Verification

The AES RoCC (Rocket Custom Coprocessor) Accelerator was designed in the Spring 2021 offering of EE290C to provide AES block cipher hardware acceleration for the SoC that the whole course taped out. In terms of operation, the accelerator receives instructions from the Rocket core via the `RoCCIO` interface and accesses data (i.e. key and plain/ciphertext data) via a TileLink interface to the backing memory of the SoC. To verify the accelerator for tapeout, the decoupled and TileLink test components as well as Java's standard cryptography library were utilized to build fully randomized accelerator-level tests that checked for functional correctness.

4.1.3.1 Test Setup

For the test setup shown in Figure 4.3, two sets of test components are used. On the left, a `DecoupledDriver` acting as a master agent drives instructions from the stimulus generator on the `RoCCIO`; at the same time, a `DecoupledMonitor` is observing all instructions and responses on the interface. On the right, a `TLDriverSlave`, in the place of the backing DRAM memory, interfaces with the accelerator via TileLink and supplies the key and text data randomly generated via the stimulus generator. Note that there is also a `TLMonitor` attached to the interface that checks for protocol compliance. At the top, a software model of the AES block cipher (implemented via Java's standard cryptography library) computes the expected results to be compared with the accelerator's actual results.

Input Stimulus

Similar to the above tests, both directed and randomly generated stimuli were used. The directed tests consisted of the example test vectors given in the NIST publication of the
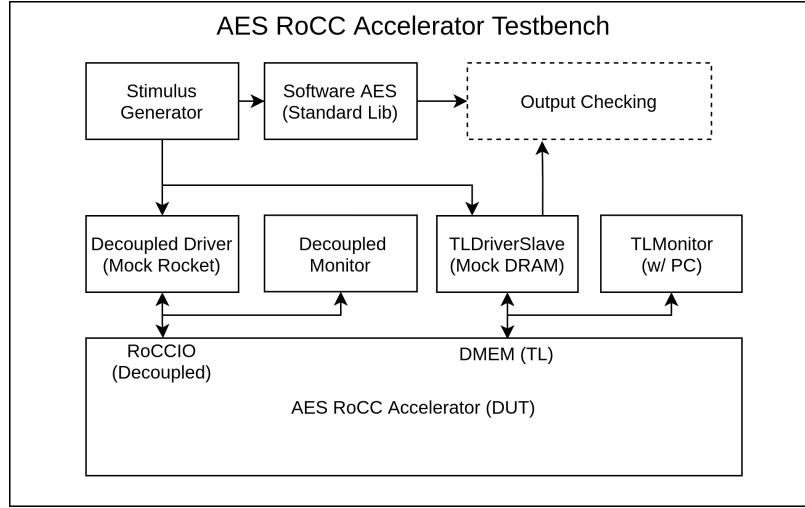
Figure 4.3: AES RoCC Accelerator test setup.

block ciphers[11] and were used as sanity tests. The randomized tests were then used to verify that the functionality is still correct for random input key and data values.

In terms of input stimulus, the stimulus generator is responsible for generating the type of operation (encryption or decryption), the key size (128 or 256 bit), the key data, the text data (blocks to encrypt or decrypt), as well as the addresses in which the key and text data are stored in memory. The type of operation, key size, and addresses for the key and text data are sent to the `DecoupledDriver` in the form of instructions. The key and text data, along with their addresses, are sent to the `TLDriverSlave` such that the accelerator can access the correct data. Moreover, the data is also sent to the software model to generate the correct result for output checking.

Test Coverage

RTL coverage was also collected across the various directed and random tests to check for completeness. Merged coverage across all directed and random tests resulted in 97.32% Line, 95.98% Condition, 63.45% Toggle, and 95.52% Branch coverage, resulting in an overall score of 88.07% coverage. For more details on the process of coverage collection, Section 4.1.4 describes how it was done in the `TLRAM` tests as an example.

4.1.4 Other: Coverage Collection

As ChiselTest interfaces with RTL simulators such as Verilator and VCS (as mentioned in Section 1.1.3), the test infrastructure is able to utilize the built-in coverage collection tools within the simulators. For example, VCS supports various types of automatic coverage collection: Line, Conditional, Toggle, Branch, and Structural (all of the above). To enable

coverage collection within the tests, flags such as `StructuralCoverageAnnotation` can be added to the test to enable the corresponding coverage collection.

For example purposes, we will go over the coverage of the `TLRAM` tests. Below, Table 4.1 displays the individual coverage of each directed test (D), random test (R), as well as total merged coverage of the tests.

	Score	Line (%)	Cond (%)	Toggle (%)	Branch (%)
1. Basic Burst (D)	73.94	91.34	79.41	37.95	87.06
2. Atomic (D)	75.91	90.94	84.71	39.31	88.68
3. Non-Aligned (D)	75.87	90.94	84.12	39.99	88.41
4. Random (R)	88.09	93.7	95.88	66.56	96.23
Merged 1,2	76.86	91.34	84.71	41.91	89.49
Merged 1,2,3	79.87	91.34	89.41	47.08	91.64
Merged 1,2,3,4	88.18	93.7	95.88	66.93	96.23

Table 4.1: TLRAM test coverage.

Starting with directed tests (labeled with a '(D)'), the coverage scores, or average % coverage across the different metrics, are around mid-70's. The reason for the lower coverage is due to directed tests only testing a specific type of operation (e.g. the `Atomic` directed test only had atomic operations as input stimulus). With the randomized testing, the coverage score rose to high-80's, as this test did not have any restrictions on input stimulus and was able to test more of the design. Note that the Line, Condition, and Branch coverage metrics for the randomized test are all in the mid-90's, while the Toggle coverage metric is only at 66%, thus bringing down the average coverage score.

To consider the total coverage of the entire testbench, the coverage of each test were merged as shown in the bottom of the table. As coverage is only collected per test, VCS's URG (Unified Report Generator) was used to merge coverage statistics across different tests on the same design. Moreover, to demonstrate the incremental coverage of each test, the merged statistic of each additional test is shown. Each additional test increased the total coverage score, with the total coverage of all tests very similar to the randomized test.

4.1.5 Other: Testbench Performance

To briefly mention testbench performance, let's take a look at previous `TLRAM` test again as an example. For a randomized test of 1000 transactions, it took 67,408 simulation cycles and approximately 14.2 seconds to execute (averaged across 5 runs). This comes out to be a frequency of 4747 Hz (cycle/second) and a throughput of $2.107 * 10^{-4}$ second/cycle. This is actually quite slow for a testbench simulation, and even more so as `TLRAM` is a relatively small design. With further investigation with YourKit[12], a Java Profiler, we found that there is non-optimal execution of threads where most threads are sitting idle waiting to be run. Moreover, the threads are executed for a small amount of time before switching to another,

resulting in a very high thread switching overhead. Figure 4.4 displays the resulting thread execution during a test. Throughout the test execution, threads spend most of their time parked (yellow) and a small amount of time running (green).
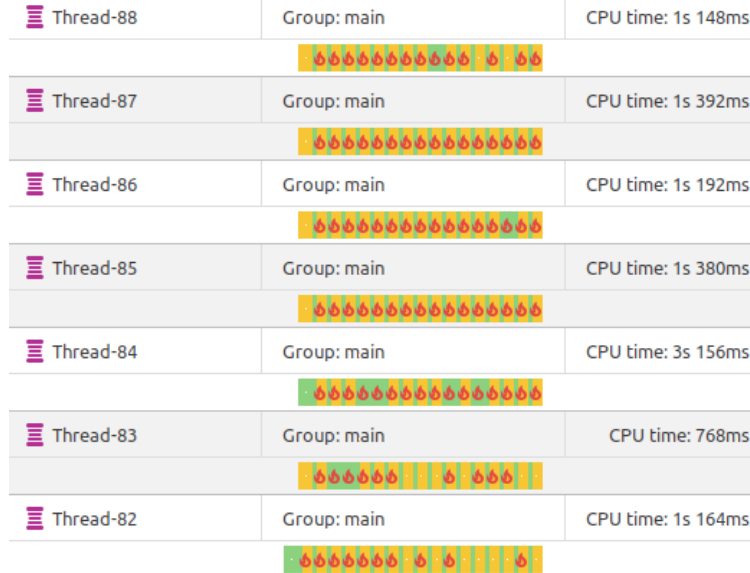

Figure 4.4: Thread execution of TLRAM test - YourKit.

This non-optimal thread execution behavior is most likely caused by how ChiselTest handles Fork/Join constructs. As described in Section 1.1.3, ChiselTest multiplexes each forked thread such that only one thread can run a time. This is to ensure that signal interaction behaviors are deterministic during simulation, but as a result, test performance degrades due to high thread switching overhead in addition to a single-threaded-like execution style. That being said, ChiselTest is still under heavy development and performance optimizations are very possible in the future.

4.2 Specification Language

To demonstrate that the specification language API (Chapter 3) can be used to define arbitrary sequential rules, the TileLink protocol checker was recreated using the API. Moreover, as the specification language has a modeling layer that supports local variables and global memory states, the new protocol checker is able to check for data correctness if given a corresponding memory model. To verify that the new TileLink protocol checker is correct, both correct and incorrect `TLRAM` output traces were passed in to ensure that it was able to correctly detect the incorrect traces.

4.2.1 TLProtocolChecker + TLMemoryModel Revisisted

As the purpose of the `TLProtocolChecker` is to check that a given transaction trace follows the sequential rules of TileLink, it was a perfect application to demonstrate the usability of the specification language API. Note that this variant of the TileLink protocol checker currently only supports up to TL-UH protocol compliance, but it can be extended to support TL-C compliance.

4.2.1.1 TLSLProtocolChecker

`TLSLProtocolChecker`, which is the SpecLang version of the `TLProtocolChecker`, follows similar checking methods as described in Section 2.8.3 in that it checks two areas: message field compliance and message handshake compliance. To accomplish this, the protocol checker contains two sets of properties, with one set to check fields and the other set to check for handshakes.

Message Field Compliance Checking

As message field compliance checking only involves checking fields within a single transaction, the properties in this set will not contain a time operator and will take on the form of `MSG_TYPE` + `Implies` + `FIELD_APs`. Moreover, because there is no time operator, all field APs can be combined via the & operator to create a composite AP that contains all checks for a given message type's fields. As field correctness is often shared between messages (e.g. the address field in ALL messages must be aligned), the protocol checker defines unit APs for each field and creates combinations of them for specific message types. Listing 4.1 displays a code snippet of how the fields within `Get` and `Arithmetic` messages are checked. At the top, unit APs are defined for each field. Those unit APs are then combined to create composite APs for specific message types, as shown on lines 11-12. The composite APs are then used to define properties for each message type, as shown on lines 13-14. Again, note that no time operators are used as these properties are meant to only check within a single transaction. During operation, the protocol checker ensures that all properties are checked on the transaction trace.

```
1   // Unit APs for each field
2   val IsGetOp = // AP that checks if PARAM field is GET
3   val ZeroParam = // AP that checks if PARAM field is zero
4   val ArithParam = // AP that checks if PARAM field is legal for arith. messages
5   def SizeWithinMaxTx(...) = // AP that checks if SIZE is within legal sizes
6   val AlignedAddr = // AP that checks if ADDRESS is aligned
7   def ContiguousMask(...) = // AP that checks if MASK field is contiguous
8   ...
9
10  // Composite APs for each message type
11  def GetAP(beatBytes: Int, maxTxSize: Int) = IsGetOp & ZeroParam &
    ↪  SizeWithinMaxTx(maxTxSize) & AlignedAddr & ContiguousMask(beatBytes) & ZeroCorrupt
12  def ArithAP(beatBytes: Int, maxTxSize: Int) = IsArithOp & ArithParam &
    ↪  SizeWithinMaxTx(maxTxSize) & AlignedAddr & ContiguousMask(beatBytes)
13
14  // Functions that return a property for the specified maxTxSize
15  def GetProperty(maxTxSize: Int) = qProp(IsGetOp + Implies + GetAP(beatBytes, maxTxSize))
16  def ArithProperty(maxTxSize: Int) = qProp(IsArithOp + Implies + ArithAP(beatBytes,
    ↪  maxTxSize))
17  ...
```

Listing 4.1: TLSLProtocolChecker field properties code snippet - simplified.

Message Handshake Compliance Checking

As handshakes compliance checking occurs across multiple transactions, the properties will take on the general form of `REQ_TYPE` + `Implies` + `###(CYCLE_DELAY)` + `RESP_TYPE`. Moreover, as request and response messages are matched by the `source` field, there are special APs that utilize the modeling layer to store the source value in a local variable that is checked when matching the response message. The code shown in Listing 4.2 displays how handshake properties are created for `Get` messages.

On lines 10-11 of Listing 4.2, helper sequences are created from the APs defined on lines 2-7 to represent parts of the handshake. Focusing on line 10, this sequence checks if the current message is a `Get` message and stores the `source` and `size` fields in local variables. Moreover, as the message could be a multi-beat request, the sequence is dynamic in that it utilizes the `SizeCheck` AP to match a specific beat count (e.g. `GetInitSequence`(4) will return a sequence that matches a `Get` request of 4 beats). Line 11 then describes the sequence that matches the response message: it must have the same `source` and `size` fields as the request, and it also needs to have the expected data specified by the memory model (not shown). Line 14 gives an example of a single beat `Get` handshake property. However, in the case of burst messages, there is either repeated requests (writing) or responses (reading). This is remedied by utilizing the static repetition operator, as shown on line 17. By passing in a specific beat count to the `GetHandshakeProp` function, it will return a property with the corresponding `GetInitSequence` sequence to match the request along with the corresponding repetitions of `AccessAckDataCheckSequence` sequences. Note that the protocol checker enumerates all

possible burst sizes and creates a property for each case. And similar to the message field properties, all message handshake properties are checked across the transaction trace.

Along with the message handshake properties, the protocol checker also contains a non-message-specific property that checks for any extra requests or responses that may appear in the transaction trace. It does this by first traversing the entire transaction trace to count the number of requests and responses and then checking if they match. This property is needed since the message handshake properties above do not check for extra responses (e.g. property finishes checking when it sees the expected number of responses but nothing more) or missing requests (e.g. property never gets activated).

```scala
1   // Handshake APs
2   val SaveSource = // AP that saves SOURCE into a local variable (REQ)
3   val SaveSize = // AP that saves SIZE into local variable (REQ)
4   def SizeCheck(...) = // Configurable AP that matches SIZE to # of beats (REQ)
5   val CheckSource = // AP that checks if SOURCE matches the local variable  (RESP)
6   val CheckSize = // AP that checks if SIZE matches the local variable (RESP)
7   val CheckData = // AP that checks if DATA matches the memory state (RESP)
8
9   // Helper sequences to match part of GET handshake
10  def GetInitSequence(beatCount: Int) = qSeq(IsGetOp & SaveSource & SaveSize &
    ↪   SizeCheck(beatCount, beatBytes))
11  val AccessAckDataCheckSequence = qSeq(###(1, -1), IsAccessAckDataOp & CheckSource &
    ↪   CheckSize & CheckData)
12
13  // Example one-beat GET handshake property
14  val OneBeatGetProp = qProp(GetInitSequence(beatCount) + Implies +
    ↪   AccessAckDataCheckSequence)
15
16  // Dynamic multi-beat GET handshake property using static repetition
17  def GetHandshakeProp(beatCount: Int) = qProp(GetInitSequence(beatCount) + Implies +
    ↪   (AccessAckDataCheckSequence * beatCount))
```

Listing 4.2: TLSLProtocolChecker handshake properties code snippet - simplified.

4.2.1.2   TLSLMemoryModel

As the TLSLProtocolChecker supports data correctness checking, the TLSLMemoryModel is implemented to generate the required memory states for each transaction. It is implemented as a wrapper for the TLMemoryModel. In terms of generating the memory states, the memory model iterates through the given transaction trace and produces either a memory state or a filler object for each transaction. When a write request is encountered, the TLSLMemoryModel will update its internal state; when a response message with data is reached, it will generate a TLSLMemoryState with the expected data according to its internal state. Note that the memory state object is optimized for Tilelink — as each transaction can hold at most one data value, each memory state only carries one data value instead of

the entire memory state. In contrast to Listing 3.13, the optimized memory state shown in Listing 4.3 only carries one data value rather than an entire hashmap.

```scala
// Custom optimized TL memory state (only one data value per transaction)
class TLSLMemoryState(init: UInt = 0.U) extends SLMemoryState[UInt] {
  var int_state = init

  def get(addr: Int): UInt = {
    int_state // Addr unused
  }
}
```

Listing 4.3: Optimized TLSLMemoryState for TileLink.

4.2.2 TLRAM Output Trace Mutation Tests

To test the functional correctness of the `TLSLProtocolChecker`, trace mutation tests were used to verify that the protocol checker is able to correctly detect transaction traces that fail to meet protocol compliance. The test setup is shown in Figure 4.5: take an existing verified transaction trace and maliciously mutate some transactions within it. If the protocol checker correctly detects the mutation in the modified trace, then the mutation is caught and the protocol checker passes the test. The use of mutations in the output trace is to mimic the output of a faulty `TLRAM` design. Although it would be more accurate to mutate the design itself, trace mutations accomplish the same in a simpler fashion. In terms of mutations, several modifications were used: adding extra transactions, removing transactions, modifying transaction fields, and combining both good and bad traces together. All mutations were designed to introduce protocol-breaking modifications, with the exception where two good traces are combined. For the cases where individual transactions were modified, the `TLSLProtocolChecker` was able to detect the mutations and correctly flag all malformed traces. For the cases where traces were combined, the protocol checker correctly flagged all cases when a malformed trace was created.


Figure 4.5: TLRAM trace mutation test setup.

Chapter 5

Conclusion and Future Work

The verification library presented in this work can accomplish many types of verification tasks. It includes a library of VIPs to interact with common DUT interfaces such as the decoupled and TileLink interface, as shown by the testbench designs in Chapter 4 that verify the `TLRAM`, `InclusiveCache` (L2), and an AES RoCC accelerator. The library also contains an embedded specification language to define and check transaction-level properties, as shown in the `TLRAM` trace mutation tests.

While the library in its current state is usable, it has performance limitations and issues with VIP determinism arising from the use of infinite loops in the VIP body and the use of unsynchronized, but safely accessed, mutable data structures.

5.1 VIP Determinism

One area of future work is to define a VIP API such that a VIP is deterministic and able to handle combinational paths through a DUT by construction. Consider the case of a combinational path through a DUT and VIPs on each interface driving stimulus as shown in Figure 5.1.


Figure 5.1: VIP race condition with combinational logic.

In this example, there is a combinational path from the slave (right) port's ready signal to the master[1] (left) port's ready signal. If we were using Verilog to drive stimulus into the DUT using 2 threads as shown in the figure, there would be a race condition. Verilog simulators do not have a specified order of execution so the simulation process can potentially vary across simulators and across runs from the same simulator. If we tried expressing the

---

[1]Repeat of footnote 1 from Chapter 2: The outdated terms *master* and *slave* are used in this paper to avoid confusion and remain consistent with the TileLink spec, which currently uses the outdated terminology.

same stimulus driving algorithm using chiseltest, we would get a runtime error that catches this source of non-determinism, but doesn't give us a way to fix it.

To resolve this issue, we have been prototyping an API where a VIP is defined as a set of functions, shown below.

```
1  trait VIP[I, S] {
2    def drive(io: I, state: S): I
3    def update(io: I, state: S): S
4  }
```

Listing 5.1: Deterministic API prototype.

The `drive` function specifies how a VIP should poke an interface `I` given a current state `S`. The `update()` function is called right before the clock on the interface is actually stepped with the latest values of the interface so the VIP can update its state. This fundamentally amounts to describing a VIP as an explicit FSM.

Now, the simulation runtime can perform fixpoint iteration to resolve combinational paths through a DUT:

1. Peek the interfaces of a DUT

2. Pass the peeked value to each VIP's `drive()` function

3. Poke the values returned by each `drive()` into the DUT and force combinational propagation through the DUT

4. Repeat from step 1 until the values driven into the DUT no longer change

This procedure guarantees deterministic VIPs and proper handling of combinational paths. It also has the benefit of removing new thread creation from the testbench components to improve performance.

5.2 VIP Synthesizability

If the VIP's state type `S` were restricted to a subtype of `chisel3.Data`, a VIP implementation becomes a description of a Chisel circuit. This would make the VIP and the fixpoint iteration runtime synthesizable and able to be packed into the simulation binary with the DUT RTL.

Future work involves generalizing this approach for the specification language eDSL too, and using synthesizable VIPs to reduce the IPC overhead of communication between Scala and the RTL simulator.

Bibliography

[1] Chisel/FIRRTL Developers. *Chisel/FIRRTL Hardware Compiler Framework*. URL: https://www.chisel-lang.org.

[2] Berkeley Architecture Research. *Chipyard Documentation*. URL: https://chipyard.readthedocs.io/en/latest/.

[3] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. DOI: 10.1109/MM.2020.2996616.

[4] "IEEE Standard for Universal Verification Methodology Language Reference Manual". In: *IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)* (2020), pp. 1–458. DOI: 10.1109/IEEESTD.2020.9195920.

[5] *Open Verification Methodology*. URL: https://verificationacademy.com/verification-methodology-reference/ovmworld/docs_2.1.2/html/index.html.

[6] Anson Tsai, Vighnesh Iyer, and Ryan Lund. *Open-source Verification Library for Chisel RTL*. URL: https://github.com/TsaiAnson/verif.

[7] Kevin Laeufer. *Maltese-SMT GitHub Project*. URL: https://github.com/ucb-bar/maltese-smt.

[8] SiFive Inc. *SiFive TileLink Specification*. URL: https://sifive.cdn.prismic.io/sifive%5C%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_tilelink-spec-1.8.0.pdf.

[9] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), pp. 1–1315. DOI: 10.1109/IEEESTD.2018.8299595.

[10] Li Haoyi. *Sourcecode GitHub Project*. URL: https://github.com/com-lihaoyi/sourcecode.

[11] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation Methods and Techniques*. en. Dec. 2001. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=51031.

[12] *YourKit Java Profiler*. URL: https://www.yourkit.com/java/profiler/features/.

Appendix A

Test Component Library Code

```
1   while(true) {
2     if (inputTransactions.nonEmpty && idleCycles == 0) {
3       val t = inputTransactions.dequeue()
4
5       // Setting waitCycles
6       if (t.waitCycles.litValue().toInt > 0) {
7         idleCycles = t.waitCycles.litValue().toInt
8         while (idleCycles > 0) {
9           idleCycles -= 1
10          clock.step()
11        }
12      }
13      while (!interface.ready.peek().litToBoolean) {
14        clock.step()
15      }
16
17      interface.valid.poke(true.B)
18      interface.pokePartial(t.data)
19      clock.step()
20
21      interface.valid.poke(false.B)
22
23      // Setting postSendCycles
24      idleCycles = t.postSendCycles.litValue().toInt
25    } else {
26      if (idleCycles > 0) idleCycles -= 1
27      clock.step()
28    }
29  }
```

Listing A.1: DecoupledDriverMaster implementation.

```scala
1   private val aDriver = new DecoupledDriverMaster(clock, interface.a)
2   // dDriver and bDriver just consume transactions blindly
3   private val dDriver = new DecoupledDriverSlave(clock, interface.d, 0)
4   // Conditional instantiation as Channels B,C,E are optional (only defined if interface
    ↪   allows)
5   private val bDriver = if (interface.params.hasBCE) Option(new DecoupledDriverSlave(clock,
    ↪   interface.b, 0)) else None
6   private val cDriver = if (interface.params.hasBCE) Option(new
    ↪   DecoupledDriverMaster(clock, interface.c)) else None
7   private val eDriver = if (interface.params.hasBCE) Option(new
    ↪   DecoupledDriverMaster(clock, interface.e)) else None
8
9   // Push method filters transactions into their respective channels
10  // e.g. all TLBundleA transactions  go to the driver for Channel A
11  // Note: Catches error where user tries to send something on Channel B or D
12  def push(tx: Seq[TLChannel]): Unit = {
13    tx.foreach { channel: TLChannel =>
14      assert(channel.isLit())
15      channel match {
16        case a: TLBundleA =>
17          val txProto = new DecoupledTX(new TLBundleA(params))
18          val tx = txProto.tx(a)
19          aDriver.push(tx)
20        case c: TLBundleC =>
21          val txProto = new DecoupledTX(new TLBundleC(params))
22          val tx = txProto.tx(c)
23          cDriver.get.push(tx)
24        case e: TLBundleE =>
25          val txProto = new DecoupledTX(new TLBundleE(params))
26          val tx = txProto.tx(e)
27          eDriver.get.push(tx)
28        case _ =>
29          throw new RuntimeException("TLDriverMaster got a TLBundleB or TLBundleD which
            ↪   can't be driven")
30      }
31    }
32  }
```

Listing A.2: TLDriverMaster implementation.

```scala
1  private val aDriver = new DecoupledDriverSlave[TLBundleA](clock, interface.a, 0)
2  private val dDriver = new DecoupledDriverMaster[TLBundleD](clock, interface.d)
3  // Conditional instantiation as Channels B,C,E are optional (only defined if interface
   ↪   allows)
4  private val bDriver = if (interface.params.hasBCE) Option(new
   ↪   DecoupledDriverMaster(clock, interface.b)) else None
5  private val cDriver = if (interface.params.hasBCE) Option(new DecoupledDriverSlave(clock,
   ↪   interface.c, 0)) else None
6  private val eDriver = if (interface.params.hasBCE) Option(new DecoupledDriverSlave(clock,
   ↪   interface.e, 0)) else None
7  private val monitor = new TLMonitor(clock, interface)
8
9  var state = initialState
10
11 fork {
12   while (true) {
13     // extract TLBundle A,C,E (request / GrantAck channels)
14     val txFromMaster = monitor.getMonitoredTransactions().map(_.data).flatMap{
15       case _:TLBundleD | _:TLBundleB => None
16       case other => Some(other)
17     }
18     // Generating response transaction and new state from slaving function
19     val (responseTxns, newState) = slaveFn.respondFromState(txFromMaster, state)
20     state = newState
21     dDriver.push(responseTxns.collect{ case t: TLBundleD => t }.map {
22       t: TLBundleD =>
23         new DecoupledTX(new TLBundleD(params)).tx(t)
24     })
25     if (params.hasBCE) {
26       bDriver.get.push(responseTxns.collect{ case t: TLBundleB => t }.map {
27         t: TLBundleB =>
28           new DecoupledTX(new TLBundleB(params)).tx(t)
29       })
30     }
31   clock.step()
32   }
33 }
```

Listing A.3: TLDriverSlave implementation.

```scala
1   // Memory in this model is word addressed
2   val byteAddr = txA.address.litValue()
3   val wordAddr = (byteAddr / bytesPerWord).toLong
4   val wordsToProcess = ceil(pow(2, txA.size.litValue().toInt) / bytesPerWord).toInt
5   ...
6   case TLOpcodes.PutPartialData | TLOpcodes.PutFullData =>
7     val writeData = txA.data.litValue()
8     val writeMask = txA.mask.litValue().toInt
9     if (state.burstStatus.isDefined) {
10      // Currently in a write burst
11      val burstStatus = state.burstStatus.get
12      val newMem = TLMemoryModel.write(state.mem, burstStatus.baseAddr +
        ↪  burstStatus.currentBeat, writeData, writeMask, bytesPerWord)
13      val newBurstStatus = if ((burstStatus.currentBeat + 1) == burstStatus.totalBeats) {
14        None
15      } else {
16        Some(burstStatus.copy(currentBeat = burstStatus.currentBeat + 1))
17      }
18      (Seq.empty[TLChannel], state.copy(mem = newMem, burstStatus = newBurstStatus))
19    } else {
20      val newMem = TLMemoryModel.write(state.mem, wordAddr, writeData, writeMask,
        ↪  bytesPerWord)
21      if (wordsToProcess == 1) {
22        // Single beat write
23        (Seq(AccessAck(txA.size.litValue().toInt, txA.source.litValue().toInt)),
        ↪  state.copy(mem = newMem))
24      } else {
25        // Starting a burst
26        val burstStatus = TLMemoryModel.BurstStatus(wordAddr, 1, wordsToProcess)
27        (Seq(AccessAck(txA.size.litValue().toInt, txA.source.litValue().toInt)),
        ↪  state.copy(mem = newMem, burstStatus = Some(burstStatus)))
28      }
29    }
```

Listing A.4: TLMemoryModel "PutPartial/Full" implementation.

```scala
1  private val aMonitor = new DecoupledMonitor[TLChannel](clock, interface.a)
2  private val dMonitor = new DecoupledMonitor[TLChannel](clock, interface.d)
3  private val bMonitor = if (interface.params.hasBCE) Option(new
   ↪  DecoupledMonitor[TLChannel](clock, interface.b)) else None
4  private val cMonitor = if (interface.params.hasBCE) Option(new
   ↪  DecoupledMonitor[TLChannel](clock, interface.c)) else None
5  private val eMonitor = if (interface.params.hasBCE) Option(new
   ↪  DecoupledMonitor[TLChannel](clock, interface.e)) else None
6
7  def getMonitoredTransactions(): Seq[DecoupledTX[TLChannel]] = {
8    // Collecting all transactions from monitors
9    val tx = aMonitor.monitoredTransactions ++
10     dMonitor.monitoredTransactions ++
11     bMonitor.map(_.monitoredTransactions).getOrElse(Seq()) ++
12     cMonitor.map(_.monitoredTransactions).getOrElse(Seq()) ++
13     eMonitor.map(_.monitoredTransactions).getOrElse(Seq())
14   // Clearing transactions to avoid repetition
15   aMonitor.monitoredTransactions.clear()
16   dMonitor.monitoredTransactions.clear()
17   if (interface.params.hasBCE) {
18     bMonitor.get.monitoredTransactions.clear()
19     cMonitor.get.monitoredTransactions.clear()
20     eMonitor.get.monitoredTransactions.clear()
21   }
22   // Sorting by cycle stamp and running protocol checker if available
23   val res = tx.sortBy(_.cycleStamp.litValue())
24   if (protocolChecker.isDefined) {protocolChecker.get.check(res.map {_.data})}
25   res
26 }
```

Listing A.5: TLMonitor implementation.