CHISELVERIFY: A VERIFICATION LIBRARY FOR CHISEL

Andrew Dobis*, Tjark Petersen*, Hans Jakob Damsgaard*, Kasper Hesse*, Enrico Tolotto*, Simon Thye Andersen*, Richard Lin† & Martin Schoeberl*.

**Technical University of Denmark (DTU)*
Department of Applied Mathematics & Computer Science

†University of California Berkeley
Department of Electrical Engineering & Computer Sciences

OUTLINE

- Background: Verification & Chisel.

- Current solutions: Current ways to verify chisel designs.

- Motivation: Need for Verification tools for Chisel.

- Our solution: bringing verification to Chisel directly in Scala.

- Evaluation: Comparing verification with ChiselVerify to verification with UVM.

BACKGROUND

BACKGROUND: VERIFICATION

- Verification of digital systems:
  - Testing a design before it has been taped out.
  - Verification tools allow to verify that the tests we are writing “test the right things”.

- Universal Verification Methodology (UVM):
  - Testing methodology implemented in SystemVerilog, a popular solution for verification.
  - Allows for the writing of standardised test-benches, which can be reused on multiple designs.

BACKGROUND: VERIFICATION TOOLS

- Functional Coverage:
    - **Qualitative** and **fine-grained** coverage metric.
    - Gives progress on "which features have been tested?".

- Constrained Random Verification (CRV):
    - Enables the creation of random variables which generate values that satisfy a set of constraints.
    - Allows for the generation of constantly valid inputs, making a small test case capable of covering many of the component's functionalities.

BACKGROUND: CHISEL

- Chisel:
  - **Hardware Construction Language** embedded in Scala.
  - Allows for the use of **Object-Oriented** and **Functional** programming constructs in the scope of hardware description.
  - Compiles through an intermediate representation called **FIRRTL** before generating output **Verilog**.
  - Can be either **synthesised** or **simulated** using:
    - Output **Verilog** (e.g. with **verilator**)
    - Intermediate **FIRRTL** with **Treadle** (a custom simulator and testing engine).

CURRENT SOLUTIONS

OVERVIEW OF THE CURRENT SOLUTIONS

- For Chisel:
    - **ChiselTest**: “traditional” test-benches with peek-poke-expect interfaces and forking, lacks verification features.
    - **ScalaTest**: Software testing framework, not ideal for hardware, doesn’t simulate the hardware, only checks the Chisel code itself.
- For Verilog:
    - **SystemVerilog**: Extension of Verilog that enables object oriented programming and verification features inside of the test-benches.
    - **UVM:** Verification Methodology, enables a standardised testing method that can be reused for many different DUTs.

CURRENT SOLUTIONS: WHAT'S MISSING?

- Other types of solutions:
  - RFuzz: Fuzzer for RTL circuits. Only allows fuzzing, no verification tools.
  - Chisel Formal: Formal Verification in Chisel, static verification, doesn't solve the same problems.

- What's missing in these solutions:
  - **ChiselTest**: For test-benches, not really verification.
  - **ScalaTest**: Not made for Hardware.
  - **SystemVerilog & UVM**:
    - Too verbose, requires ~800 LOC for a test-bench.
    - Requires multiple languages to test a Chisel design.

MOTIVATION

MOTIVATION: CURRENT LACK OF VERIFICATION TOOLS

- So far Chisel doesn't have any "native" tools that allow the easy use of verification functionalities.
- Current solutions all require external tools to function.
- We need tools which are built **for** Chisel.
- We also want tools which are **efficient** to use and **easy to learn**, not the case of current popular solution like SystemVerilog with UVM.

OUR SOLUTION: CHISELVERIFY

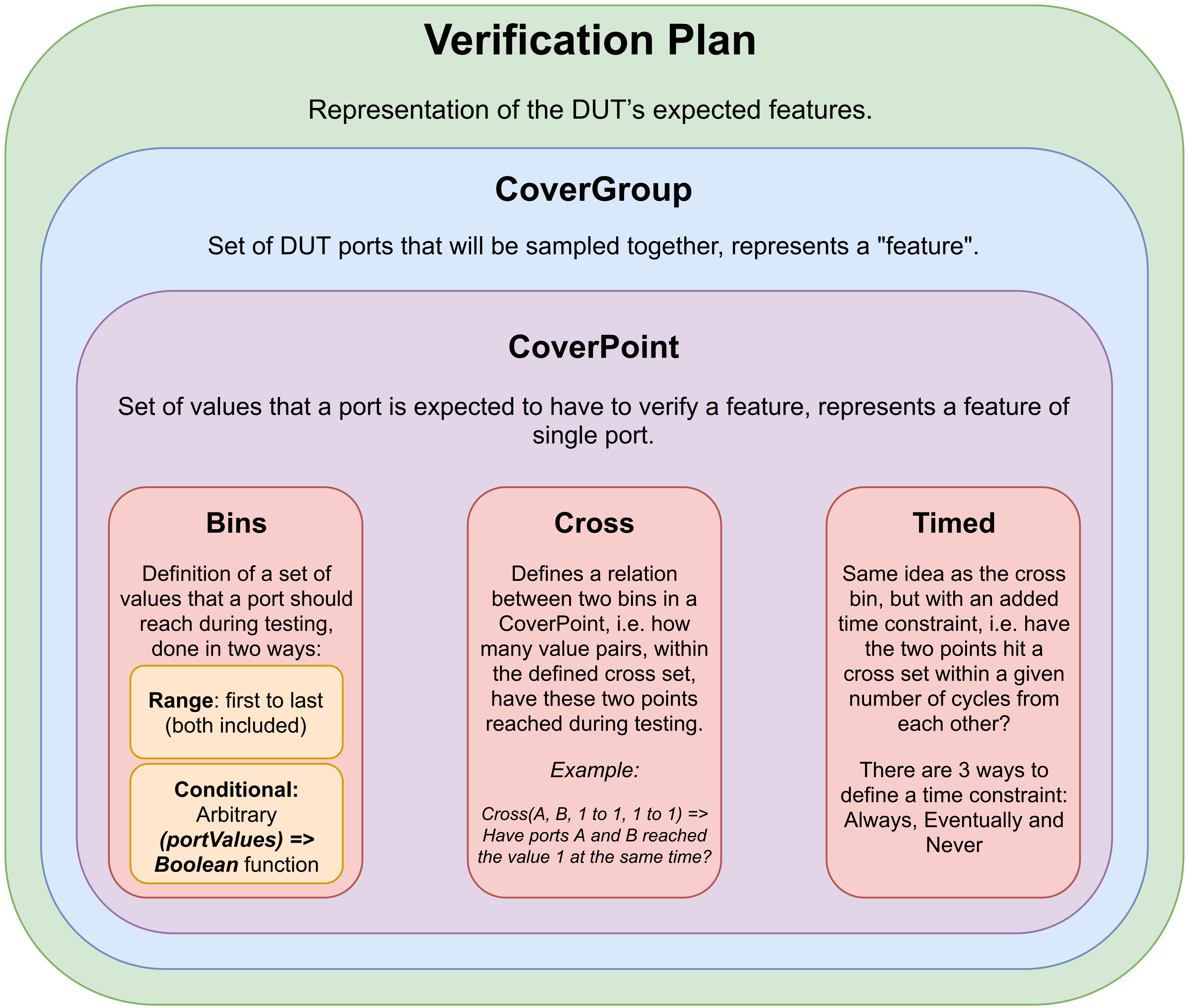
CHISELVERIFY: OVERVIEW

- Hardware Verification library for Chisel, inspired by UVM.
- Powered entirely by Scala and ChiselTest.

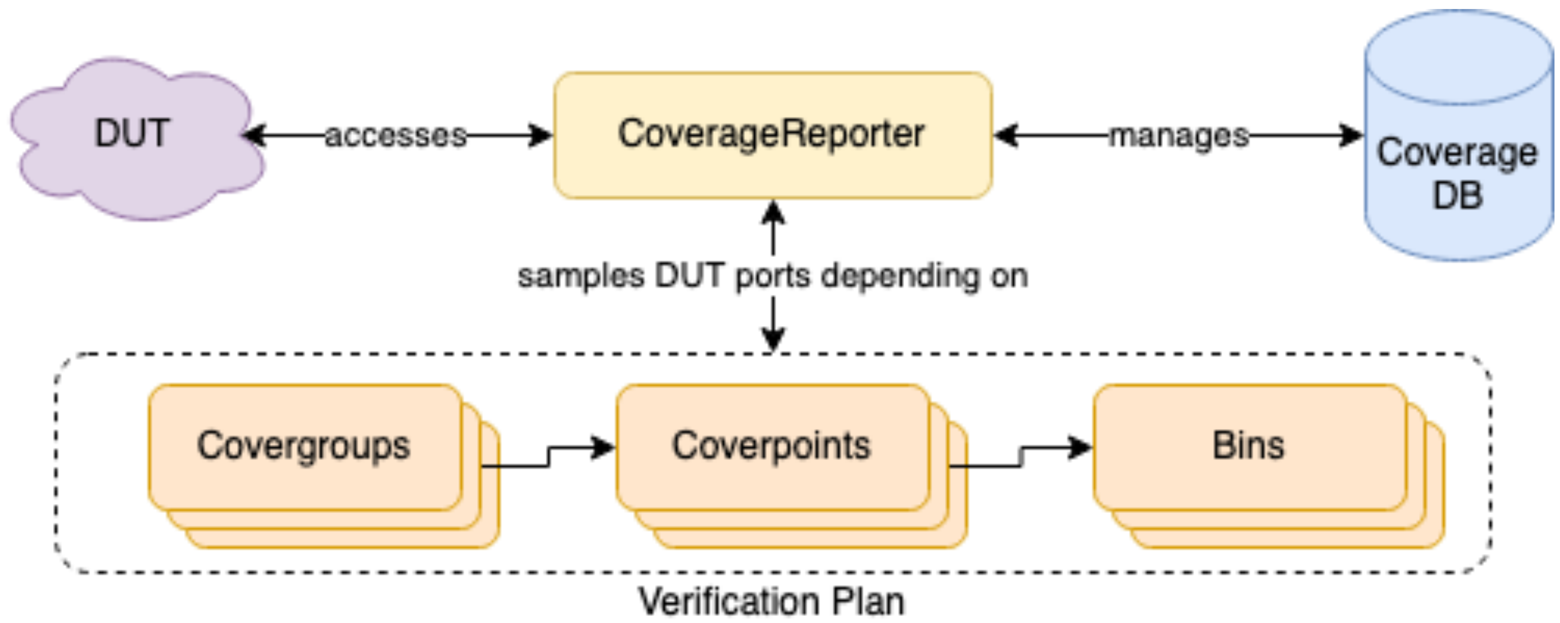
- ChiselVerify brings the following to the Chisel ecosystem:
  - Functional Coverage
  - Constrained Random Verification
  - Bus Functional Models
  - Timed Assertions

CHISELVERIFY: FUNCTIONAL COVERAGE

FUNCTIONAL COVERAGE: WHAT IS IT?



FUNCTIONAL COVERAGE: HOW DID WE DO IT?



- **CoverageDB:** DataBase that maintains the values gathered for all of the bins across multiple tests in a test suite.
- **Coverage Reporter:** Handles the registration of CoverPoints and Bins to the DB, samples the bin values and creates the coverage report.
    - This is used to create the verification plan.

FUNCTIONAL COVERAGE: API

Two main constructs:

- **Cover**: Represents a feature of the DUT. Can be represented by both a single and multiple DUT ports. Its behavior is defined by the **bins** it contains.

```
case class cover(pointName: String, ports: Data*)
 def apply(b: Bin*)
 def apply(delay: DelayType)(b: Bin*)
```

- **Bins**: Definition of a part of a feature. Defined using value ranges or condition, and can contain timing information.

```
def bin(name: String, range: Option[Range] = None, condition: Option[Seq[BigInt] => Boolean] = None, expectedHits: BigInt = 0)
```

FUNCTIONAL COVERAGE: USING IT

- Create the coverage reporter and verification plan.

```
val cr = new CoverageReporter(dut)
cr.register(
    cover("accu", dut.io.outA)( //CoverPoint 1
        bin("lo10", 0 until 10), bin("First100", 0 until 100)),
    cover("test", dut.io.outB)( //CoverPoint 2
         bin("testLo10", 0 until 10)),
    //Declare cross points
    cover("accuAndTest", dut.io.outA, dut.io.outB)(
        cross("both1", Seq(1 to 1, 1 to 1)))
)
```

- Sample the CoverPoints inside of the test.

```
cr.sample()
```

RESULT: FUNCTIONAL COVERAGE REPORT

- Create and print the coverage report

```
//Generate report
val report = cr.report
print(report.serialize)
```

- Example result:

```
============ COVERAGE REPORT ============
============== GROUP ID: 1 ==============
COVER_POINT PORT NAME: accu
BIN lo10 COVERING Range 0 until 10 HAS 10 HIT(S) = 100,00%
BIN First100 COVERING Range 0 until 100 HAS 50 HIT(S) = 50,00%
=========================================
COVER_POINT PORT NAME: test
BIN testLo10 COVERING Range 0 until 10 HAS 4 HIT(S) = 40,00%
=========================================
CROSS_POINT accuAndTest FOR POINTS accu AND test
BIN both1 COVERING Range 1 to 1 CROSS Range 1 to 1 HAS 1 HIT(S) = 100,00%
=========================================
=========================================
```

CHISELVERIFY: CONSTRAINED RANDOM VERIFICATION

CONSTRAINED RANDOM VERIFICATION: WHAT IS IT?

- Model tests using randomness:
  - Give random inputs to the DUT and expect values using a golden model.
- Guide the randomness using constraints:
  - We don’t want the randomness to be uniformly distributed => use constraints to describe the random distribution.
- Idea: Add a Constraint Programming Language to Scala/Chisel
  - Create Random Objects, then define Random Variables and Constraints in it.
  - Random Objects define a **Constraint Satisfaction Problem (CSP).** This problem is solved using a CSP Solver.
  - We use **JaCoP** as a CSP Solver.

CONSTRAINED RANDOM VERIFICATION: HOW DID WE DO IT?

- Random objects created by extending the **RandObj** trait using a given **Model** with a seed.
- Random fields are added as **Rand/Randc** values inside the class.
- Constraints are defined using either:
  - Single constraints: using "#" (e.g. `val lenConstraint = len > 2`)
  - **ConstraintGroups** which are equivalent to SV

Example:

```
class Frame extends RandObj(new Model) {
  val pkType: RandVar = rand(0, 3)
  val len: RandVar = rand(0, 10)
  val noRepeat: RandVar = rand(0, 1, Cyclic)

  val legal: ConstraintGroup = new ConstraintGroup {
    len >= 2
    len <= 5
  }
}
```

CONSTRAINED RANDOM VERIFICATION: WHAT DO WE ADD?

- The list of operator used to construct constraint is the following: `<`, `<=`, `>`, `>=`, `==`, `div`, `*`, `mod`, `+`, `-`, `\=`, `^`, `in`, `inside`.
- We also added conditional constraints using:
    - **IfCon**: If a condition is met, then use the constraint
    - **ElseC**: If a condition is met, then use the constraint, else use an other.

Example:

```
val constraint1: crv.Constraint = IfCon(len == 1) {
        payload.size == 3
    } ElseC {
        payload.size == 10
    }
```

CONSTRAINED RANDOM VERIFICATION: USING IT

- `dist` can also be used to define custom distributions on a random variable by associating ranges to weights using the `:=` operator.

- The `randomize()` method returns `true` only if the CSP Solver found a set of values that satisfy the current constraints.

Example:

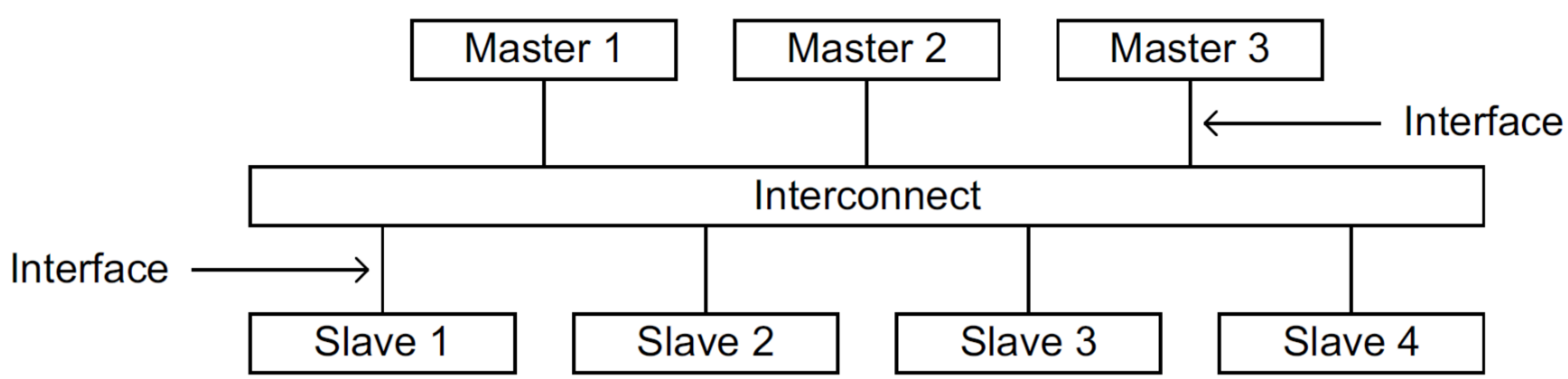
```
val myPacket = new Frame(new Model)
assert(myPacket.randomize)
```

CHISELVERIFY:
BUS FUNCTIONAL MODELS

BUS FUNCTIONAL MODEL: WHAT IS IT?

- Abstraction of the inner workings of a standardised interface.
- Allows for the use of data transfer via **Transactions**, rather than having to deal with the inner wiring manually.
- Software abstraction, is useful for faster verification.

- We chose to create a first BFM for the AXI4 Bus:



BUS FUNCTIONAL MODEL: HOW TO USE IT

- Create a **FunctionalMaster** for your AXI interfaced DUT.

```
val manager = new FunctionalManager(dut)
```

- Create transactions:
  - Write:
  - Read:

```
// Create write transaction
manager.createWriteTrx(0, Seq(42), size = 2)

// Wait for the write to complete (spin on response)
var resp = manager.checkResponse()
while (resp == None) {
  resp = manager.checkResponse()
  dut.clock.step()
}

// Create read transaction
manager.createReadTrx(0, size = 2)

// Wait for read to complete (spin on read data)
var data = manager.checkReadData()
while (data == None) {
  data = manager.checkReadData()
  dut.clock.step()
}
```

CHISELVERIFY: TIMED ASSERTIONS

TIMED ASSERTIONS: QUICK OVERVIEW

- Allows to create predicated assertions that take into account certain timing delays.
- **Types of delays:** Given a delay of x cycles:
  - **Exactly**: Assertion is true exactly in x cycles.
  - **Eventually**: Assertion is true at least once in the next x cycles.
  - **Always:** Assertion is true every cycle for the next x cycles.
  - **Never:** Assertion isn’t true at any cycle for the next x cycles.
- Two types of Timed Assertions:
  - **ExpectTimed:** Uses *ChiselTest*’s expect for the assertion.
  - **AssertTimed:** Uses a software *assert(_)* for the assertion.

TIMED ASSERTIONS: USING IT

```scala
def exact[T <: Module](d: Int,  msg: String = s"EXACTLY ASSERTION FAILED")
                                (cond: => Boolean)(implicit dut: T): Unit =

def never[T <: Module](d: Int = 100,  msg: String = s"NEVER ASSERTION FAILED")
                          (cond: => Boolean)(implicit dut: T): Unit =

def eventually[T <: Module](d: Int = 100, msg: String = s"EVENTUALLY ASSERTION FAILED")
                               (cond: => Boolean)(implicit dut: T) : Unit =

def always[T <: Module](d: Int = 100,  msg: String = s"ALWAYS ASSERTION FAILED")
                     (cond: => Boolean)(implicit dut: T): Unit =
```

Example:

```scala
eventually(2, "aEqb expected timing is wrong") { dut.io.a.peek() === dut.io.b.peek() }
exact(2, "aEqb expected timing is wrong") { dut.io.a.peek() === dut.io.b.peek() }
always(2, "aEqb expected timing is wrong") { dut.io.a.peek() === dut.io.b.peek() }
never(2, "aEqb expected timing is wrong") { dut.io.a.peek() === dut.io.b.peek() }
```

CONCLUSION

CONCLUSION

- ChiselVerify brings verification to the Chisel ecosystem.

- High-Level Functional backend (i.e. Scala) allows for much more efficiency during verification process (in comparison to SystemVerilog with UVM).

- Can be used to verify non-Chisel designs as well thanks to Chisel Blackboxes.

REFERENCES

- *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and VerificationLanguage.IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012), pages 1–1315, 2018.*
- *C. Spear; SystemVerilog for verification: a guide to learning the testbench language features; Springer Science & Business Media, 2008.*
- *K. Kuchcinski and R. Szymanek, "Jacop - java constraint programming solver," 2013, cP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming ; Conference date: 16-09-2013.*
- *Xillinx AXI reference guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf*

GETTING STARTED USING CHISELVERIFY

- **Current Project repository:**

  https://github.com/chiselverify/chiselverify/

- **Project Wiki (Good way to get started):**

  https://github.com/chiselverify/chiselverify/wiki/

- ChiselVerify is published on Maven. To use it, add following line to your `build.sbt`:

```
libraryDependencies += "io.github.chiselverify" % "chiselverify" % "0.1"
```

QUESTIONS?