CHISELVERIFY: A VERIFICATION LIBRARY FOR CHISEL

Andrew Dobis, Tjark Petersen, Kasper Hesse,
Enrico Tolotto, Hans Jakob Damsgaard &
Martin Schoeberl.

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

Speaker: Andrew Dobis (andrew.dobis@alumni.epfl.ch)

OUTLINE

- Hardware Verification: Why do we need it?
- Current Verification Solutions for Chisel designs.
- Our solution: bringing verification to Chisel with Scala.
    - Functional Coverage
    - Constrained Random Verification
    - Bus Functional Models
    - Timed Assertions

HARDWARE VERIFICATION : WHAT AND WHY?

HARDWARE VERIFICATION: WHAT IS IT ?

- Verification of a design = Guaranteeing that the expected features, as described in the specification, have been correctly implemented and have the correct behaviour.

- Verification = testing before tape-out.
- Validation = testing after tape-out.

HARDWARE VERIFICATION: WHY DO WE NEED IT?

- Enables the Computer Engineer to guarantee the working state of a design before spending a lot of money taping it out.
- Saves time by allowing for automated and randomised checking, rather than writing test-benches for each possible value reached by a given port.
- Shouldn't take up too many resources. Too much time is currently spent on making sure that a design is correct.

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?

- For Chisel:
  - **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.

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?

- Statement coverage = **quantitative** approach to getting the verification progress.
    - How much code have we tested?

- Functional coverage = **qualitative** approach to getting the verification progress.
    - Which features have we tested?

FUNCTIONAL COVERAGE: WHAT IS IT?

Verification Plan

Representation of the DUT’s expected features.

CoverGroup

Set of DUT ports that will be sampled together, represents a "feature".

CoverPoint

Set of values that a port is expected to have to verify a feature, represents a feature of single port.

Bins

Definition of a set of values that a port should reach during testing, done in two ways:

Range: first to last (both included)

Conditional: Arbitrary ***(portValues) => Boolean*** function

Cross

Defines a relation between two bins in a CoverPoint, i.e. how many value pairs, within the defined cross set, have these two points reached during testing.

Example:

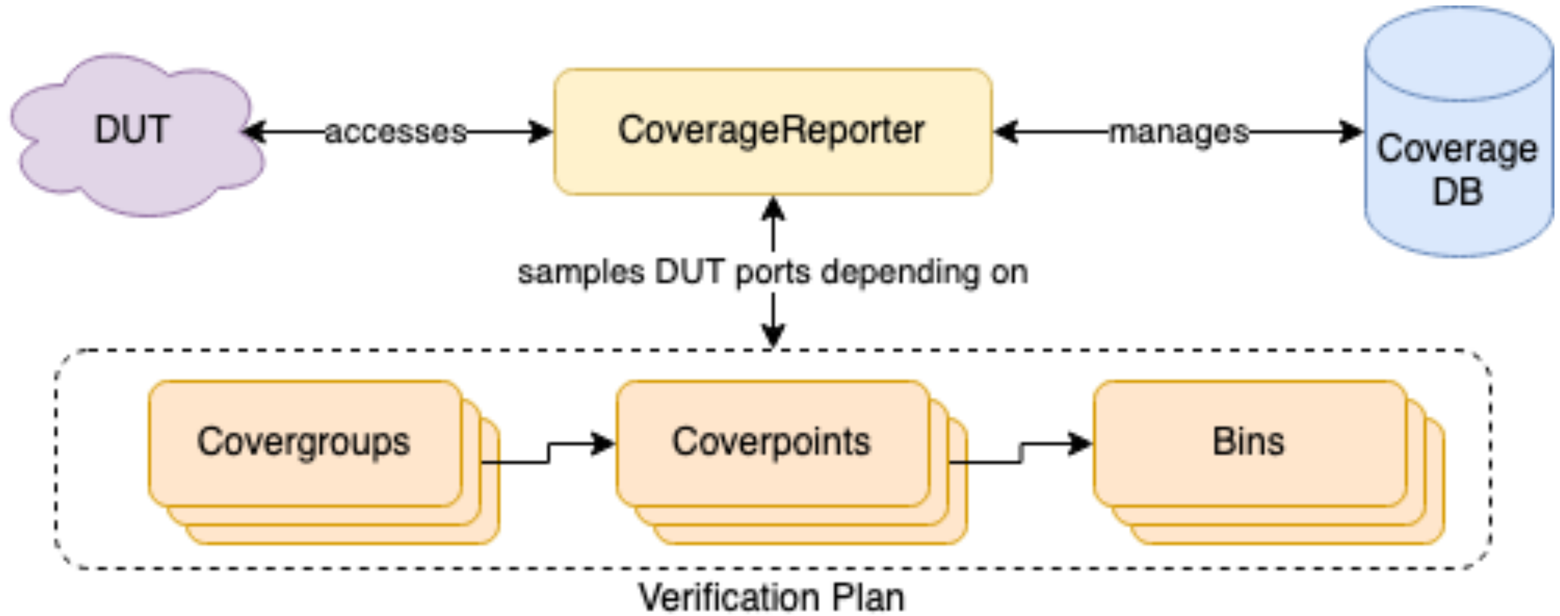
Cross(A, B, 1 to 1, 1 to 1) => Have ports A and B reached the value 1 at the same time?

Timed

Same idea as the cross bin, but with an added time constraint, i.e. have the two points hit a cross set within a given number of cycles from each other?

There are 3 ways to define a time constraint: Always, Eventually and Never

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

Rule of thumb: PointType(name, ports)(bins/conditions)

Actual API:

```
case class CoverPoint(pointName: String, port: Data)(bins: Bins*)

        Bins(val name: String, val range: Range)

        Bins(val name: String, val range: Range, val condition: Condition)

        case class Condition(name: String, cond: Seq[BigInt] => Boolean)

case class CrossPoint(n: String, p: Data*)(b: CrossBin*)

        case class CrossBin(name: String, ranges: Range*)

case class CoverCondition(pointName: String, ports: Data*)(conditions: Condition*)

case class TimedCross(name: String, port1: Data, port2: Data)(delay: DelayType)(bins: CrossBin*)

         case class Never(delay: Int)
         case class Eventually(delay: Int)
         case class Always(delay: Int)
         case class Exactly(delay: Int)
```

FUNCTIONAL COVERAGE: USING IT

- Create the coverage reporter and verification plan.

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

```
//Print coverage report
cr.printReport()
```

- 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: SYSTEMVERILOG

- Create random objects represented by a **RandObj** class.
- Store random fields as **rand** or **randc** (cyclic) variables.
- Constraints are defined for the whole object in a **constraint** block.

Example:

```
class frame_t;
rand pkt_type ptype;
rand integer len;
randc bit [1:0] no_repeat;
rand bit  [7:0] payload [];
// Constraint the members
constraint legal {
  len >= 2;
  len <= 5;
  payload.size() == len;
}
```

CONSTRAINED RANDOM VERIFICATION: HOW DID WE DO IT?

- Same ideas as in SV:
  - 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`)
    - **ConstraintGroup**s which are equivalent to SV

Example:

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

  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)
```

CONSTRAINED RANDOM VERIFICATION: USING IT

Create Random object:

```
class Packet extends RandObj(new Model(3)) {
    val idx = new Randc("idx", 0, 10)
    val size = new Rand("size", 1, 100)
    val len = new Rand("len", 1, 100)
    val payload: Array[Rand] = Array.tabulate(11)(
        new Rand(s"byte[$_]", 1, 100)
    )

    //Example Constraint with operations
    val single = payload(0) #= (len #- size)

    //Example conditional constraint
    val conditional = IfCon(len #= 1) {
        payload.size #= 3
    } ElseC {
        payload.size #= 10
    }
    val idxConst = idx #< payload.size
}
```

Instantiate and use it:

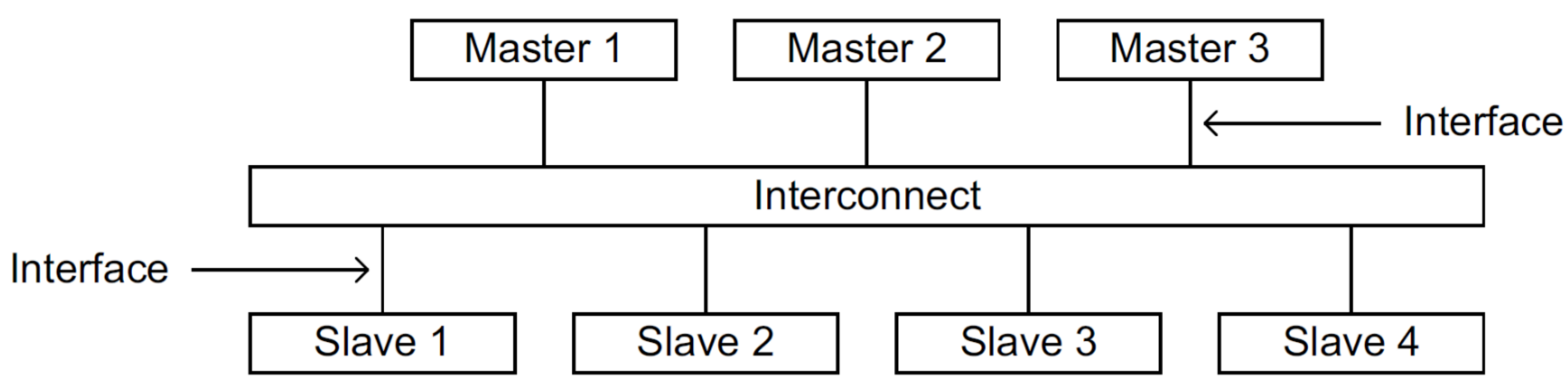
```
//Instantiate RandObj
val pckt = new Packet

//Check that the constraints were solvable
assert(pckt.randomize)

// [...] ChiselTest boilerplate [...]

//Example use of random variables in a DUT
while(pckt.randomize && /*Coverage isn't 80%*/) {
    dut.portA.poke(
        pckt.payload(pckt.idx.value()).value()
    )

    // [...] Sample the coverage and update constraints
}
```

CHISELVERIFY:
BUS FUNCTIONAL MODELS

BUS FUNCTIONAL MODEL: WHAT IS IT?

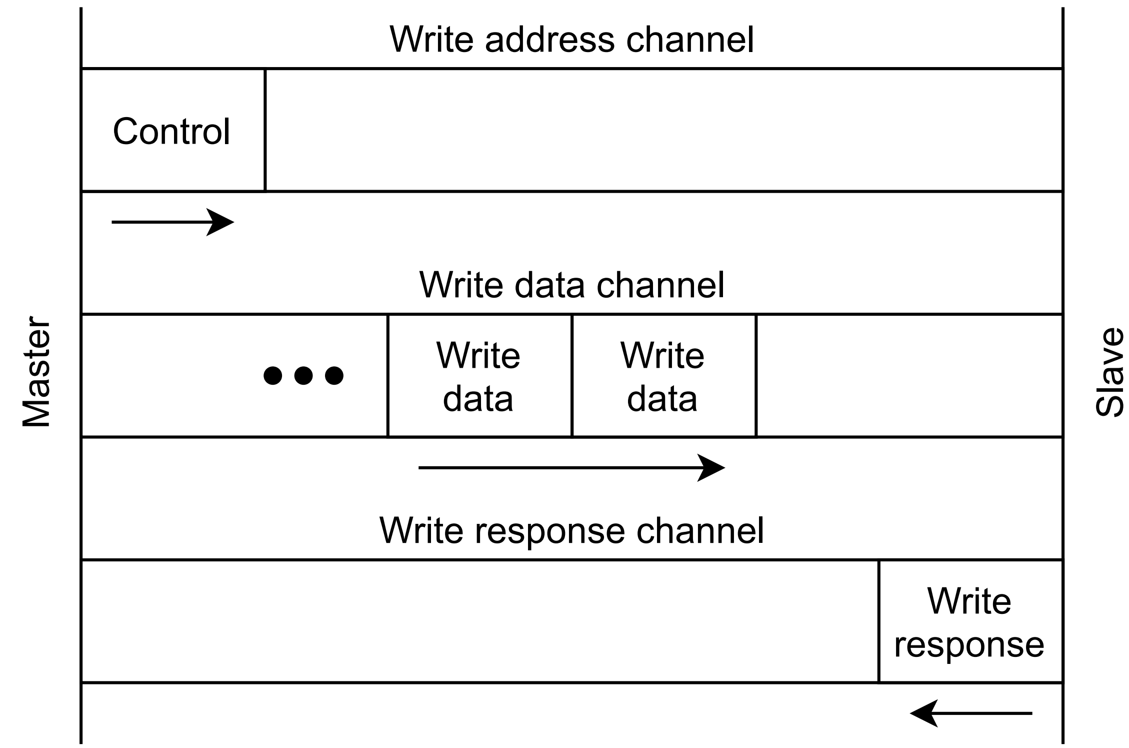
- Abstraction of the inner workings of a standardised interface.
- Allows for the use of data transfer via **Transactions**, rather than having to deal with the inner wiring manually.
- Software abstraction, is useful for faster verification.

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

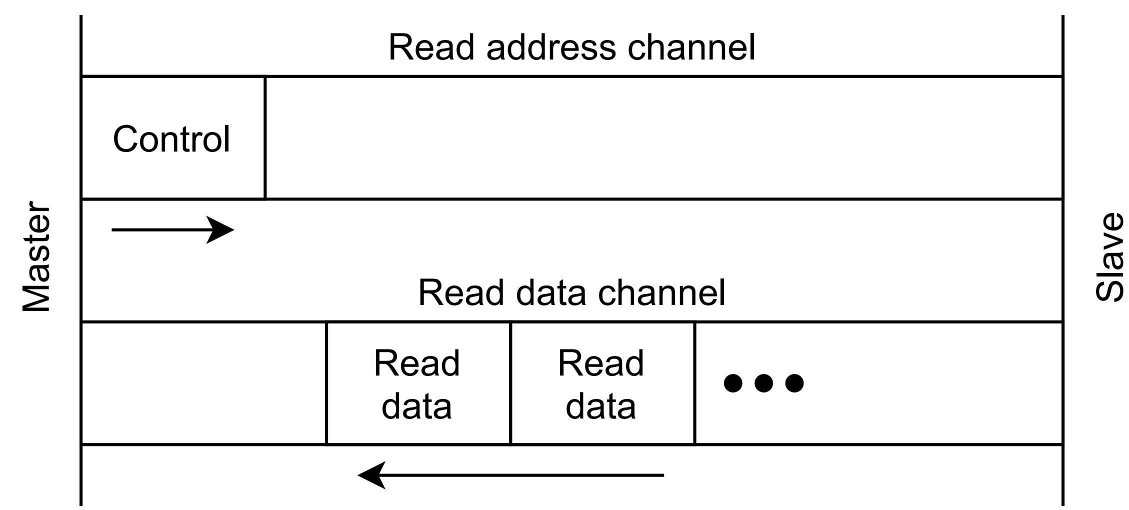


BUS FUNCTIONAL MODEL: TRANSACTIONS

- Rather than setting every wire manually, use either:
    - **WriteTransaction**



    - **ReadTransaction**



BUS FUNCTIONAL MODEL: HOW TO USE IT

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

```
val master = new FunctionalMaster(dut)
```

- Create transactions:
  - Write:

```
master.createWriteTrx(0, Seq(42), size = 2)

var resp = master.checkResponse()
while (resp == None) {
    resp = master.checkResponse()
    dut.clock.step()
}
```

  - Read:

```
master.createReadTrx(0, size = 2)

var data = master.checkReadData()
while (data == None) {
    data = master.checkReadData()
    dut.clock.step()
}
```

BUS FUNCTIONAL MODEL: API

- WriteTransaction API:

```
createWriteTrx(
    addr: BigInt, data: Seq[BigInt], id: BigInt, len: Int,
    size: Int, burst: UInt, lock: Bool, cache: UInt,
    prot: UInt, qos: UInt, region: UInt, user: UInt
)
```

- ReadTransaction API:

```
createReadTrx(
    addr: BigInt, id: BigInt, len: Int, size: Int,
    burst: UInt, lock: Bool, cache: UInt, prot: UInt,
    qos: UInt, region: UInt, user: UInt
)
```

  - **addr**: Start write address, must fit in the slave's address width.
  - **data**: List of data to write, defaults to random data, entries in **data** must fit within the slave DUT's write data width, and the list can have at most **len** entries.
  - **id**: Transaction id, defaults to 0, **id** must fit within DUT's ID width, likewise **size** cannot be greater than the DUT's write data width.
  - **len**: Burst length, defaults to 0 (i.e. 1 beat).
  - **size**: Beat size, defaults to 1B.
  - **burst**: Burst type, defaults to FIXED.

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

```
ExpectTimed[T <: Module](dut: T, port: Data, expectedVal: UInt, message: String)(delayType: DelayType)
```

```
AssertTimed[T <: Module](dut: T, cond: () => Boolean, errorMsg: String)(delayType: DelayType)
```

Example:

```
AssertTimed(dut, () => dut.io.aEvEqC.peek().litValue() == 1, "a eventually isn't c")(Eventually(11)).join()

//And the same thing but with expect
ExpectTimed(dut, dut.io.aEvEqC, 1.U, "aEqb expected timing is wrong")(Exactly(6)).join()
```

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?