CHISELVERIFY: A VERIFICATION FRAMEWORK FOR CHISEL

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

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

OUTLINE

- Hardware Verification: Why do we need it?
- Current Verification Solutions for Chisel designs.
- Our solution: bringing verification to Chisel with Scala.
    - Functional coverage
    - Constraint-random testing
    - Bus functional models

HARDWARE VERIFICATION : WHAT AND WHY?

HARDWARE VERIFICATION: WHAT IS IT ?

- Verification of a design = Guarantying 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 test his work before spending a lot of money printing 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.

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**: Not really for Verification.
    - **ScalaTest**: Not made for Hardware.

    - **SystemVerilog & UVM**:
        - Too verbose, requires ~1000 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
  - Constraint-random testing
  - Bus Functional Models

CHISELVERIFY: FUNCTIONAL COVERAGE

FUNCTIONAL COVERAGE: WHAT IS IT?

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

- <u>Functional coverage</u> = **qualitative** approach to getting the verification progress.
    - How many 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)

Transition: val0 => val1 (port goes from val0 to val1 in a cycle)

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:

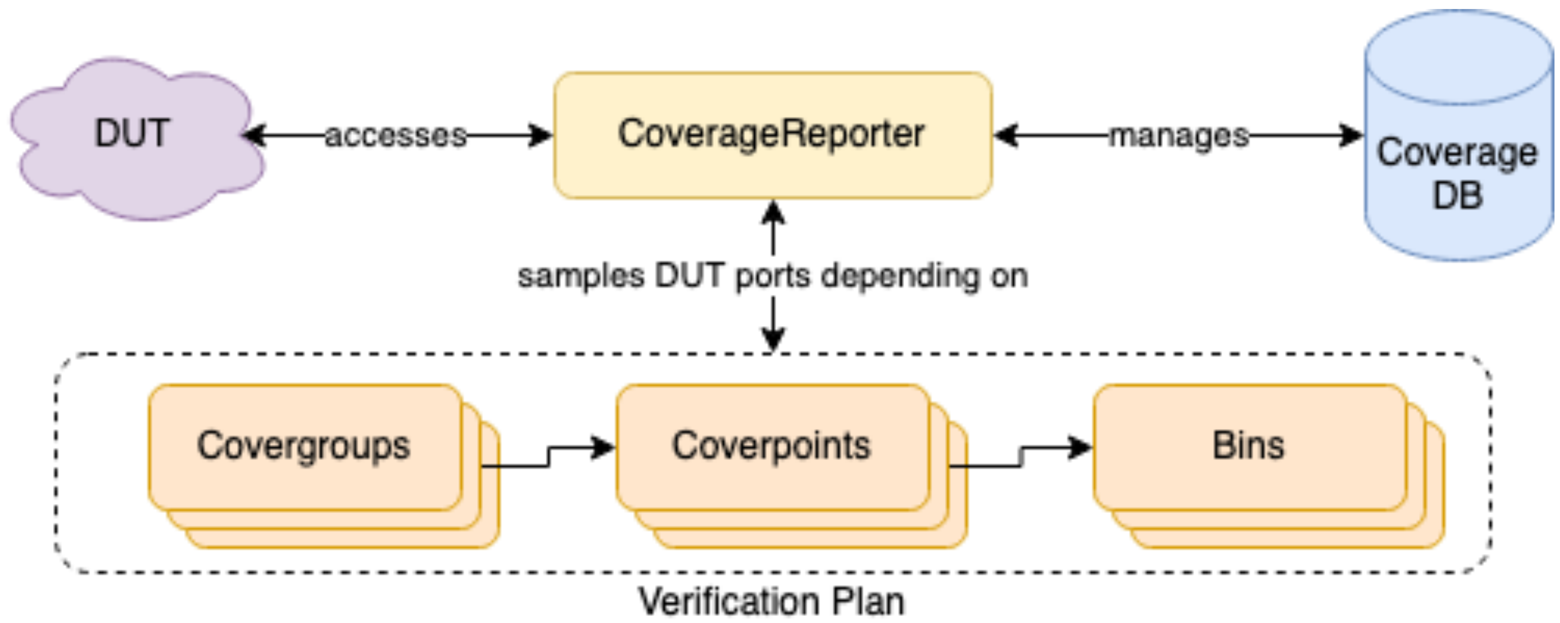
Cross(A, B, 1 to 1, 1 to 1) => Have ports A and B reached the value 1 at the same time?

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: USING IT

- Create the coverage reporter and verification plan.

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

- 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:
CONSTRAINT RANDOM TESTING

CONSTRAINT RANDOM TESTING: 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 CSP Solver to Chisel:
    - Enable the creation of Random “Frames” which in turn enables the creation of randomness constraints.

CONSTRAINT RANDOM TESTING: SYSTEMVERILOG

- Create random objects represented by a **frame** class.
- Store random fields as **rand** (discrete) or **randc** (continuous) 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;
}
```

CONSTRAINT RANDOM TESTING: 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`)
    - **ConstraintGroups** 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 payload: RandArr = new RandArr(0, 7, _model)

  val legal: ConstraintGroup = new ConstraintGroup {
    len #>= 2
    len #<= 5
    payload.size #= len
  }
}
```

CONSTRAINT RANDOM TESTING: 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:
  - **ifThen**: If a condition is met, then use the constraint
  - **ifThenElse**: If a condition is met, then use the constraint, else use an other.

Example:

```
val constraint1: crv.Constraint = ifThenElse(len #= 1)(payload.size #= 3)(payload.size #= 10)
```

CONSTRAINT RANDOM TESTING: USING IT

- Each random class exposes a `randomize()` method, which automatically solves the constraints specified in the class and assign to each random filed a random value.

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

Example:

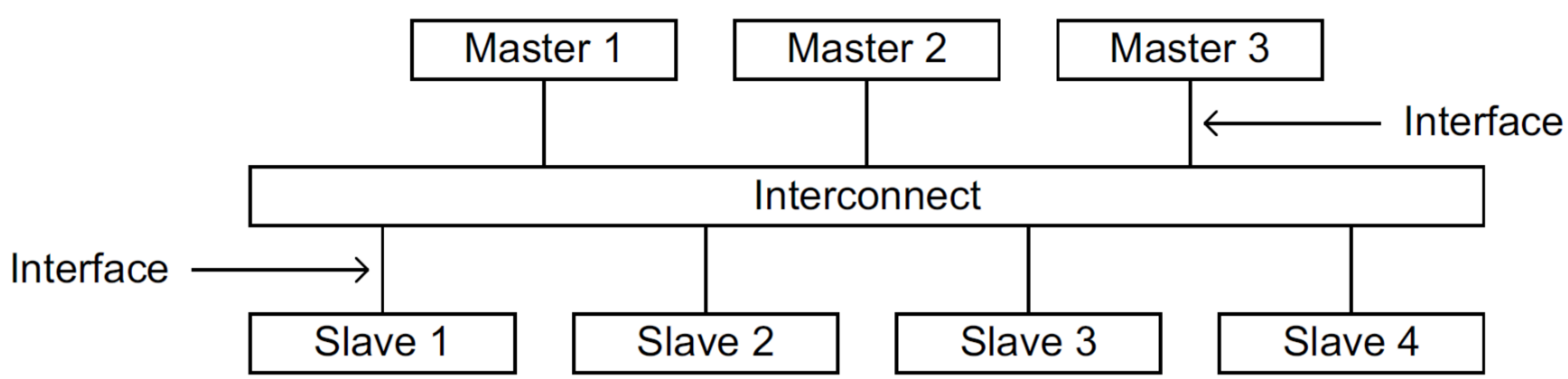
```
val myPacket = new Frame(new Model)
assert(myPacket.randomize)
```

CHISELVERIFY:
BUS FUNCTIONAL MODELS

BUS FUNCTIONAL MODEL: WHAT IS IT?

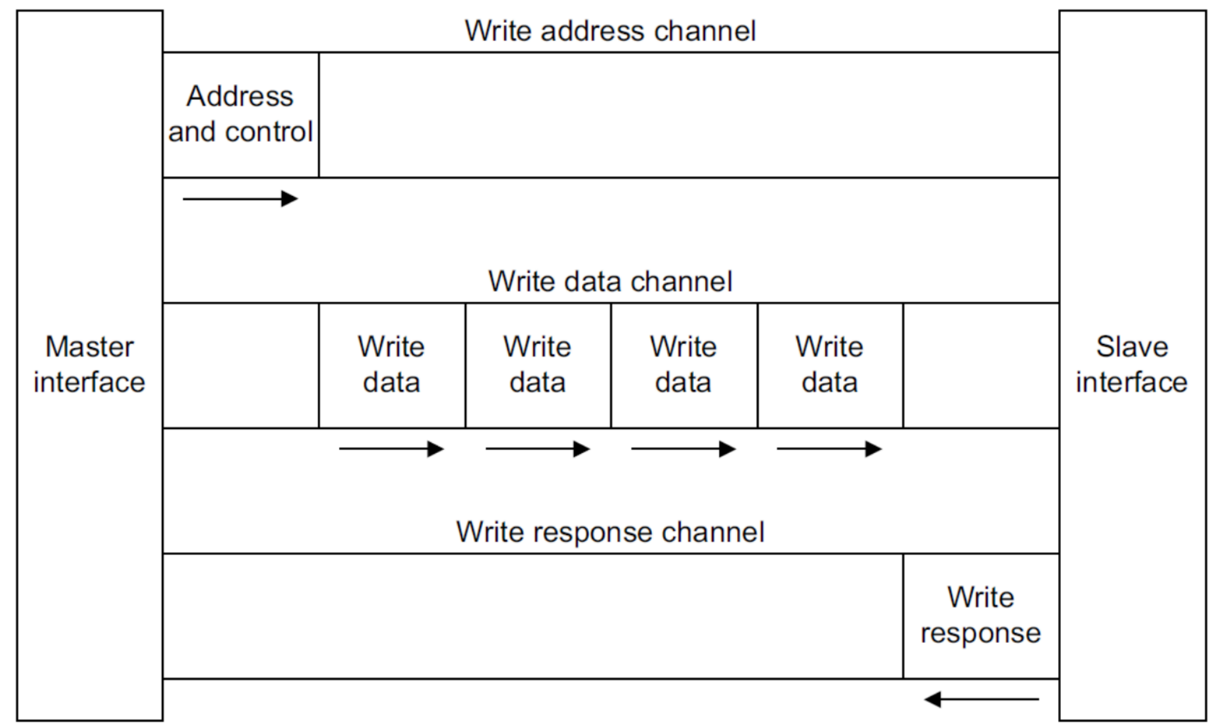
- Abstraction of the inner workings of a standardised interface.
- Allows for the use of data transfer via **Transactions**, rather than having to deal with the inner wiring manually.
- Software abstraction, is useful for faster verification.

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

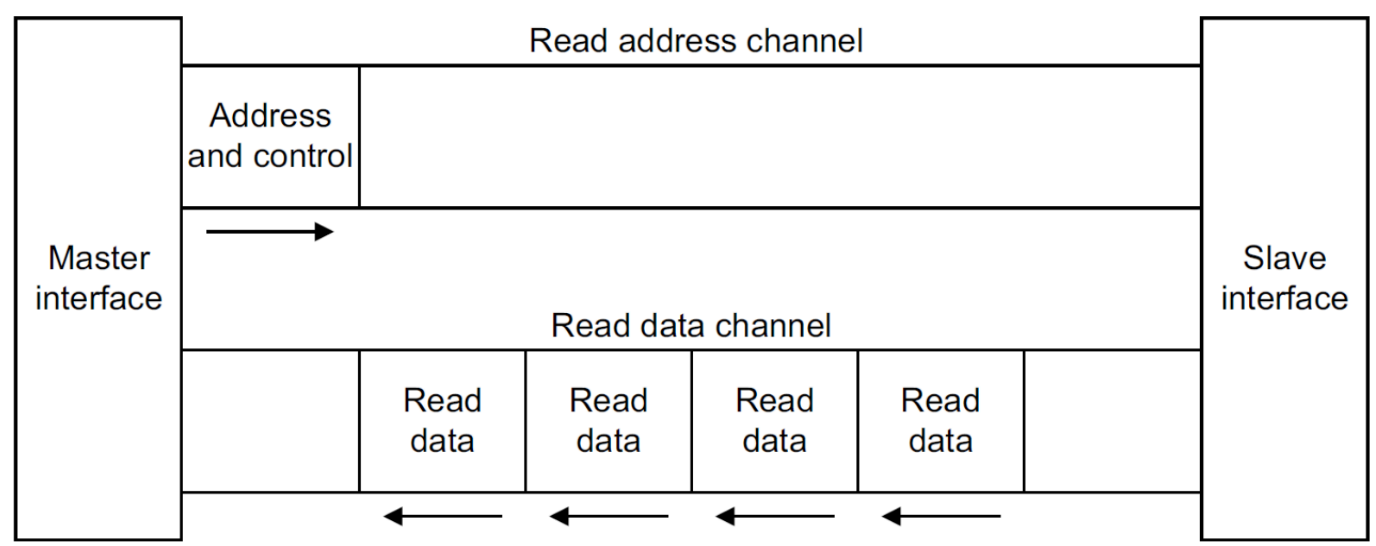


BUS FUNCTIONAL MODEL: TRANSACTIONS

- 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.

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.*
- *Choco Solver, Open-Source Java library for Constraint Programming:* https://github.com/chocoteam/choco-solver
- *Xillinx AXI reference guide: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf*

- **Current Project repository:**

  https://github.com/chiselverify

QUESTIONS?