Formal Verification

Martin Schoeberl

Technical University of Denmark
Embedded Systems Engineering

October 28, 2025

Outline

- What is formal verification
- How can it be used to verify digital designs
- Chisel Formal

The New New Product Development Game

- by Hirotaka Takeuchi and Ikujiro Nonaka
- Paper that started the Scrum movement (mid 80s)
  - Move like a rugby team together to reach the aim
  - Instead of a relay race, where runners wait for arrival
- In production, not in software development
- Observing mostly Japanese, some USA companies
- Examined multinational companies
  - Fuji-Xerox, Canon, Honda, NEC, Epson, Brother, 3M, Xerox, and Hewlett-Packard
- Available at Harward Business Review
- Easy reading, give it a try

New product development processes

1. Built-in instability
2. Self-organizing project teams
3. Overlapping development phases
4. “Multilearning”
5. Subtle control
6. Organizational transfer of learning

When is Simulation and Testing Enough?

- How much do you need to test?
- How confident are you?
  - Did you test the corner cases?
  - An issue when the same person did design and tests
- We cannot cover all input possibilities
- Except in trivial cases
  - Parameters might help
  - Cover all cases for a 4-bit ALU
  - Assume the design also works for 32 bits
- Any other option?

Assertions in Chisel

- An Assertion statement states assumptions about a program
- You have seen `assert` in Scala in the first lab
- Can also be used in Chisel
- Assertion is checked during simulation time
- Syntax:

```
io.sum := io.a + io.b
assert(io.sum === io.a + io.b, "error
    message")
```

Assert Example

```
class Assert extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val sum = Output(UInt(8.W))
  })
  io.sum := io.a + io.b

  /* the following will not be true when
  the addition overflows
  assert(io.sum >= io.a)
  assert(io.sum >= io.b)
   */
  assert(io.sum === io.a + io.b)
}
```

`require` vs `assert`

- Chisel `assert` checks value in simulation
  - Emits non-synthesizable Verilog
  - Using a Chisel expression that results in a `Bool`
  - Can also have a failure message
- Scala `assert` checks value at circuit construction
- Using a Scala expression that results in a `Boolean`
- Better use Scala's `require`
- `require` is used for input sanity checking

```
abstract class Fifo[T <: Data](gen: T, val
    depth: Int) extends Module {
  val io = IO(new FifoIO(gen))

  require(depth > 0, "Number of buffer
      elements needs to be larger than 0")
}
```

Formal Verification

- Simulation explores (only) some test cases
- Formal verification explores *all* possible behaviors
- Uses SAT/SMT solvers to prove properties
  - As bounded model checking (BMC)
- Finds corner cases that simulation might miss
- Sounds a bit too good to be true, right?
  - Might take a long time, days or more

Properties in Formal Verification

- Properties describe what must always hold
- Types of properties:
  - Safety: something bad never happens
  - Liveness: something good eventually happens
- Examples:
  - FIFO never overflows (safety)
  - Every request is eventually granted (liveness)

Chisel Formal

- Extension of `ChiselTest` by Kevin Laeufer
  - Part of his PhD at UC Berkeley
- Chisel `assert` is *formally* checked
  - Up to a bound
- If the proof fails, Chisel formal provides a counter example (as VCD waveform)
- Install the open Z3 theorem prover
  - With your favorite packet manager, e.g., brew, apt,...

Chisel Formal

- Import the library

```
import chiseltest.formal._
```

- Formal is part of ChiselTest (add treat `Formal`)

```
class FormalTest extends AnyFlatSpec with
    ChiselScalatestTester with Formal {
```

- Use `verify` instead of `test`

```
  "AssertTest" should "pass" in {
    verify(new Assert(), Seq(BoundedCheck(5),
        WriteVcdAnnotation))
  }
```

- Have the asserts in your module

Access the History

- `past()` allows us to make assamptions over time
- If `in` was 10 last cycle, then `out` must be 11 now

```
when (past(in) === 10.U) {
   assert(out === 11.U)
 }
```

- A signal must stay stable, be the same as in the last cycle

```
assert(x === past(x))
```

- With `past(x)` call we get the value of `x` from the last clock cycle

Past Example

```
class PastTest() extends Module {
  val io = IO(new Bundle() {
    val in = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })

  val reg = RegNext(RegNext(io.in))
  assert(reg === past((past(io.in))))

  io.out := reg
}
```

Assumptions

- We can limit the search space
- Assuming that some (input) signals have restricted values
- Assume that a signal will never be zero:

```
assume(x > 0.U)
```

Assume Example

```
class AssumeTest extends Module {
  val in = IO(Input(UInt(8.W)))
  val out = IO(Output(UInt(8.W)))
  out := in + 1.U
  assume(in > 3.U && in < 255.U)
  assert(out > 4.U)
  assert(out <= 255.U)
}
```

More Functions

- Synthactic sugar for `past`
- changed
- fell
- rose
- stable
- See Javadoc

Limits to Bounded Model Checking

- BMC will check your design only for the first N cycles
- Can fail in clock cycle N+1
- Only checks your assertions
- They need to be correct

Other Options

- Synopsys and Cadence have for sure formal verifiers
- Sby is a free, open-source verification tool
- Part of YosysHQ
- See Matt Venn explaining it
- SymbiYosys (sby) Documentation

Next Week

- Guest lecture by Frans Skarman
- On Spade, a new HDL
- Frans developed also Surfer, the new waveform viewer
  - Check out the VSC plugin (show it)
- Please setup Spade

Summary

- Formal verification is the next step in testing
- It is another tool in your toolbox
- Easy to include in your continuous integration on GitHub
- I have prepared some lab exercises for you