Safe Haskell	None
Language	Haskell2010
Extensions	DataKinds

Testbenches.FuzzifierTestbench

This module defines a simple testbench for the `fuzzifier` we defined in the `Fuzzifier` module.

Documentation

```
inp :: Vec 3 Int
```

Writing a testbench module is very straightforward, as it relies on the existence of some "magic functions". Specifically, those are:

```
topEntity :: Signal a -> Signal b
testInput :: Signal a
expectedOutput :: Signal b -> Signal Bool
```

The easiest way to look at it is as follows: each testbench focuses on the testing of a single entity, which the CλaSH compiler expects to be called `topEntity`. The `topEntity` must be fed an input so as to be tested, that input `Signal` being `testInput`. Finally, for ensuring that the `topEntity` is behaving properly; we must also define `expectedOutput`, which is component that, when fed the output `Signal` of the `topEntity`, returns a `Signal Bool` which remains `True` as long as the `topEntity` is returning the expected output.

CλaSH defines numerous helper functions to aid in the creation of all the components our testbench might need. This also the point where we unrap the `Reader` wrapper of the fuzzifier.

```
topEntity = runReader fuzzifier (def :: Config)
testInput = stimuliGenerator inp
testReference = runReader fuzzifierT (def :: Config)
expectedOutput = outputVerifier $ CLaSH.Prelude.map (testReference) inp
```

Special mention is warranted towards the rigidity of the types of `topEntity`, `testInput` and `expectedOutput`. They are constrained to being monomorphic (at least at the signal level), however, composing multiple components within the body of the topEntity is should come as the trivial and obvious thing to do.

Note that we can directly test against the combinational circuit `fuzzifierT` after we run it through the `Reader` as well. This is great; again, because it allows us to happily write the "pure" combinational circuit and and take it as a reference implementation. In fact, we are actually taken `fuzzifierT` for granted, as it should idealy be tested separately with QuickCheck anyhow.

The best way to "run" a testbench is just sampling the `Signal` coming from `expectedOutput` and looking at the result of each of the tested values. We have chosen to define a simple convenience function for it:

```
runTestBench = sampleN 3 $ expectedOutput (topEntity testInput)
```

One must note, however, that it is still only `topEntity`, `testInput` and `expectedOutput` which are cherished by the CλaSH compiler. In fact, based on these three definitions alone, CλaSH can transpile your high-level, pure hardware designs into VHDL, Verilog or System Verilog using the `:vhdl`, `:verilog` and `:systemverilog` commands in the interpreter.

inp is the input of this testbench.