Lava: Hardware Design in Haskell

Michael Christensen
Alvin Glova

CS595E
01/13/2021

Outline

- Considerations
- Lava HDL Introduction
- A Little about Haskell
- Lava Language
- Circuit Examples and Demos
  - Simulation
  - VHDL Code Generation
  - Verification
- Lava Families

Questions to Consider

Consider the following when picking an HDL:

- What level of abstraction is being described?
- How are circuits modelled? Shallow or deep embedding?
- What kind of data can travel through your circuits/functions?
- How will you represent sequential logic?
- What components/basic components are allowed? How to represent them?
- How will you model check it?
- How will you simulate it?
- How does lowering occur? What ops are supported?
- Will you expose the clock? Multiple clocks? Allow asynchronity?

Lava

- Family of Haskell-hosted EDSLs for expressing gate-level hardware descriptions
- These circuit descriptions that can be interpreted in several ways
  - Description
  - Simulation
  - Verification
  - Code generation
- Using Haskell
  - Take advantage of polymorphism, higher-order functions, typeclasses, purity
  - Extensible
- Control low-level details with nice functional abstractions
- Influential ancestors include µFP and Ruby[1], and directly influenced derivatives like Kansas, Xilinx Lava, and functional HDLs

[1] Probably not the Ruby you're thinking of

A Little About Haskell

- Statically-typed, HM-based, type-inferred, focus on purity
- Functions, recursion **everywhere**
- Polymorphism (`map :: (a -> b) -> [a] -> [b]`)
- `data` for creating ADTs (data structures)
- `type` for type synonyms (typedefs)
- `class` for type classes, subclassing (i.e. interfaces)
- Monads (abstract datatype of actions)
    - Often used for allowing computations to implicitly carry additional data and sequence computations in special ways (i.e. state, IO, nondeterminism, failure, etc.)

Lava Language (according to the paper)

- `Circuit` type class
- Type class hierarchy for several interpretations, enable certain operations
  - Circuit
  - Arithmetic
  - Symbolic
  - Layout
- Composition (>−>)

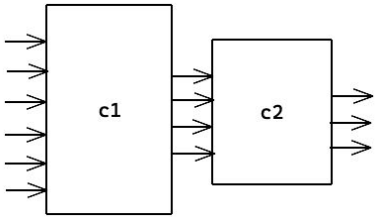
```
class Monad m ⇒ Circuit m where
        and2, or2 :: (Bit, Bit) → m Bit

class Circuit m ⇒ Arithmetic m where
        plus, times :: (NumSig, NumSig) → m NumSig

class Circuit m ⇒ Sequential m where
        delay :: Bit → Bit → m Bit
        loop :: (Bit → m Bit) → m Bit

class Circuit m ⇒ Symbolic m where
        newBitVar :: m Bit
        newNumVar :: m NumSig

class Circuit m ⇒ Layout m where
        row :: ...
        column :: ...
```

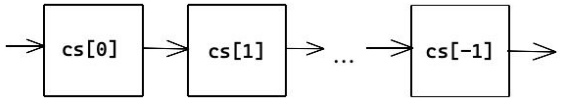
Chalmers Lava

- `Signal` type
  - Bit ≡ `Signal Bool`
  - Binary number ≡ `[Signal Bool]` (element 0 is LSB)
  - `low :: Signal Bool`
  - `high :: Signal Bool`
  - `99 :: Signal Int`
- Circuits are just functions from input (tuple of wires) to output (tuple of wires)
  - `and2 :: (Lava.Signal Bool, Lava.Signal Bool) -> Lava.Signal Bool`
  - For simulation, each input and output must be an instance of the `Generic` typeclass

Connection Patterns

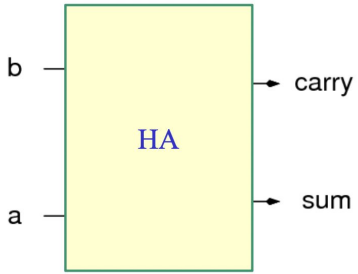
- Higher order functions that capture common ways of plugging circuits together
  - Serial (->-)
    - `serial c1 c2 = c1 . c2`
  - Compose
    - `compose cs = foldl (->-) id cs`
  - Par (-|-)
    - `par c1 c2 (a, b) = (c1 a, c2 b)`
  - Parl
    - `parl c1 c2 = halveList ->- (c1 -|- c2) ->- append`



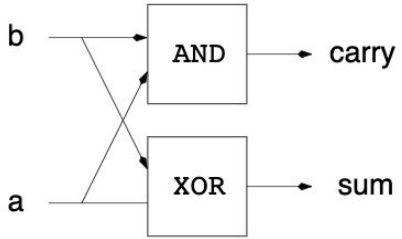


Circuit Examples and Simulation

Comb. Circuit Example: Half Adder



a	b	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Code:

```
halfAdd :: (Signal Bool,Signal Bool) ->
  (Signal Bool,Signal Bool)
halfAdd (a, b) = (sum, carry)
  where
    sum = Lava.xor2 (a,b)
    carry = Lava.and2 (a,b)
```

type signature (inferred automatically)

order does not matter inside "where"

Notes:

A->B: a function with input of type A and output of type B

(A1, A2): a pair allows several signals to be grouped together

https://github.com/pllab/Lava-Testing

Simulation

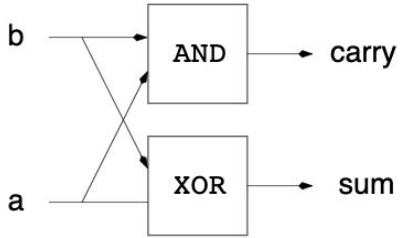
Combinational:

```
simulate :: Generic b => (a -> b) -> a -> b
```

Sequential:

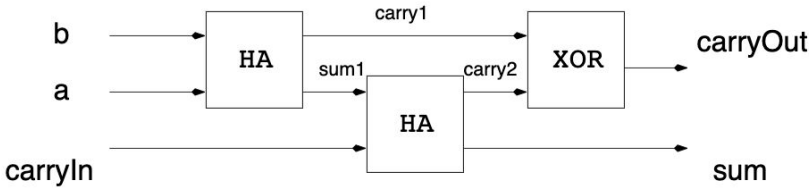
```
simulateSeq :: (Generic a, Generic b) => (a -> b) -> [a] -> [b]
```

Comb. Circuit Example: Half Adder (Simulation)




```
main> Lava.simulate halfAdd (Lava.high, Lava.low)
(high, low)

main> Lava.simulate halfAdd (Lava.high, Lava.high)
(low, high)

main> Lava.simulateSeq halfAdd Lava.domain
[(low,high), (low,low), (high,low), (high,low),
(low,high)]
```

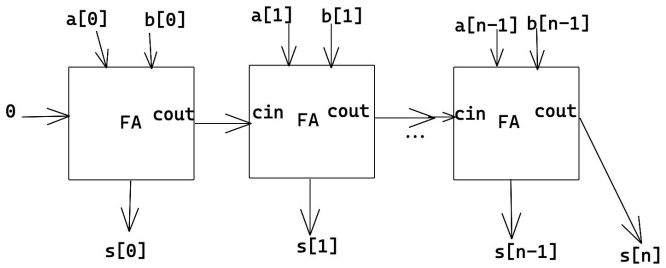
Full Adder



Code

```
fullAdd (cin, (a, b)) = (sum, cout)
  where
    (sum1, carry1) = halfAdd (a, b)
    (sum, carry2)  = halfAdd (cin, sum1)
    cout           = Lava.xor2 (carry2, carry1)
```

Generating N-bit Adder from Full Adder



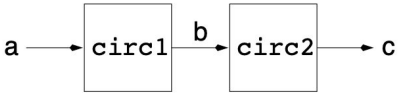
Recursion of Lists

```
adder :: (Bit, ([Bit], [Bit])) -> ([Bit], Bit)
adder (cin, ([], [])) = ([], cin)
adder (cin, (a:as, b:bs)) = (sum:sums, cout)
  where
    (sum, carry) = fullAdd (cin, (a, b))
    (sums, cout) = adder (carry, (as, bs))
```

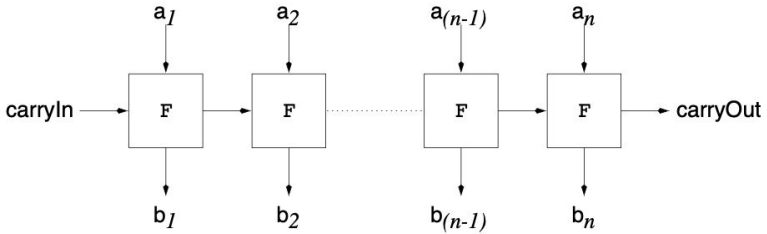
Notes:
[A]: lists of values of type A. Examples of lists:
[] (empty list)
[low,high,low,low] :: [Bit]
[(low,high),(low,low)] :: [(Bit,Bit)]
List are used both for sequences in time and for parallel signals (busses).
Length can be arbitrary and is not indicated in the type

Generating N-bit Adder from Full Adder

Using Connection Pattern




```
serial circ1 circ2 a = c
  where
    b = circ1 a
    c = circ2 b
```




```
row circ (carryIn, []) = ([], carryIn)

row circ(carryIn, a:as) = (b:bs, carryOut)
  where
    (b, carry)  =  circ(carryIn, a)
    (bs, carryOut) = row circ (carry, as)
```

Simpler Way: N-bit Adder using `row` Connection Pattern

```
adder' :: (Bit, [(Bit, Bit)]) -> ([Bit], Bit)
adder' = Lava.row fullAdd
```

Sequential Circuits

Delay Component

`delay init s`

- delays the signal **s** by one time unit
- The output during the first time unit is **init**

```
edge inp = change
  where
    inp' = delay low inp
    change = Lava.xor2(inp, inp')

main> Lava.simulateSeq edge [high, low, low, high]
[high, high, low, high]
```

```
toggle change = out
  where
    out' = delay low out
    out = Lava.xor2(change, out')

main> Lava.simulateSeq toggle [high, low, low, high]
[high, high, high, low]
```

Sequential Circuits

n-bit Counter

Output n-bit binary number at every clock cycle (from 0, +1 every clock cycle)

```
counter n () = number'
  where
    number'             = Lava.delay (Lava.zeroList n) number
    (number, carryOut) = adder (Lava.high, (Lava.zeroList n, number'))

main> Lava.simulateSeq (counter 3) (replicate 10 ()) |> print
[[low,low,low],[high,low,low],[low,high,low],[high,high,low],[low,low,
high],[high,low,high],[low,high,high],[high,high,high],[low,low,low],[
high,low,low]]
```

LSB first

Sequential Circuits

Sequential Adder

```
adderSeq (a,b) = sum
  where
    carryIn           = delay low carryOut
    (sum, carryOut)   = fullAdd (carryIn, (a,b))

main> Lava.simulateSeq adderSeq [(high, low),
(high, high), (low, high)]
[high, low, low]
```

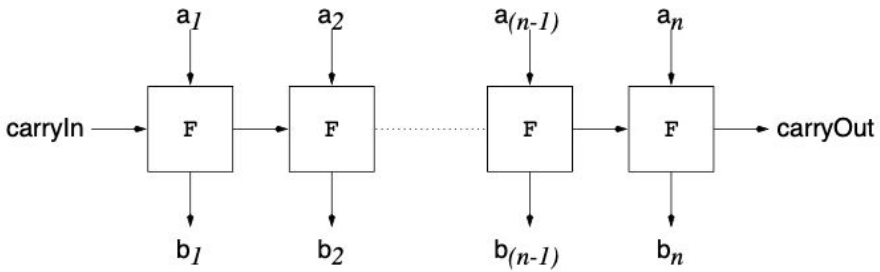
Sequential Adder via `rowSeq`

```
adderSeq' = rowSeq fullAdd
```

Connection Patterns

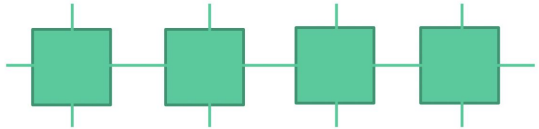
- Row
  - `row :: ((c, a) -> (b, c)) -> (c, [a]) -> ([b], c)`
  - Connect an instance of a circuit's output to another instance's input, in sequence
  - Internally, it's like a fold in purely functional terms

- RowSeq
  - Builds a row of circuits, interpreting the row over time

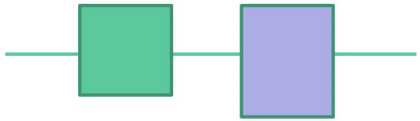


Other Connection Patterns

```
row (combinational)
rowSeq (sequential)
```

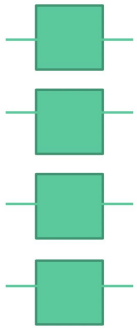


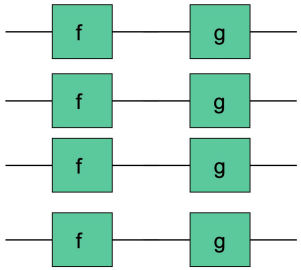
```
->- (serial composition)
```



```
Combining patterns:
map f ->- map g = map (f ->- g)
```

map





Many other available:

- column
- grid
- compose
- par
- red
- lin
- tri
- etc.

Code Generation and Verification

VHDL Generation

`Constructive` typeclass

```
class Generic a => Constructive a
where
 zero   :: a
 var    :: String -> a
 random :: Rnd -> a
```

`Generic` typeclass

```
class Generic a where
  struct    :: a -> Struct Symbol
  construct :: Struct Symbol -> a
```

VHDL netlist generation only supports bit-level operations

VHDL Generation

Code

```
Main> Lava.writeVhdlInputOutput "4BitAdder" adder
   (Lava.var "cin", (Lava.varList 4 "a", Lava.varList 4 "b"))
   (Lava.varList 4 "sum", Lava.var "cout")
```

VHDL

```
use work.all;
entity
  4BitAdder
is
port
  -- clock
  ( clk : in bit

  -- inputs
  ; cin : in bit
  ; a_0 : in bit
  ; a_1 : in bit
  …
  -- outputs
  ; sum_0 : out bit
  ; sum_1 : out bit

end entity 4BitAdder;
…
```

```
architecture
  structural
of
  4BitAdder
is
  signal w1 : bit;
  signal w29 : bit;
begin
  c_w2  : entity id    port map (clk, cin, w2);
  c_w4  : entity id    port map (clk, a_0, w4);
  c_w5  : entity id    port map (clk, b_0, w5);
  c_w3  : entity xor2  port map (clk, w4, w5, w3);
  c_w1  : entity xor2  port map (clk, w2, w3, w1);
  c_w8  : entity and2  port map (clk, w2, w3, w8);
…
 c_sum_2 : entity id    port map (clk, w13, sum_2);
 c_sum_3 : entity id    port map (clk, w20, sum_3);
 c_cout  : entity id    port map (clk, w27, cout);
end structural;
```

Verification

`Checkable` typeclass, `Property` type synonym, `ProofResult` data type

```
class Checkable a where
 property :: a -> Property

newtype Property
  = P (Gen ([Signal Bool], Model -> [[String]]))

data ProofResult
 = Valid
 | Falsifiable
 | Indeterminate
deriving (Eq, Show)
```

Verification

SAT-Solver (`minisat`) / Model Checking (smv via `nusmv`)

```
minisat :: Checkable a => a -> IO ProofResult
minisat a =
  do checkVerifyDir
     noBuffering
     (props,_) <- properties a
     proveFile defsFile (writeDefinitions defsFile props)
```

Verification tool wrappers

- Captain
- Eprover
- HeerHugo
- ISC
- Limmat
- **Minisat** (`minisat`) SAT solver
- Modoc
- Satnik
- Satzoo
- **SMV** (`nusmv`) model checker
- Vis
- Zchaff

Verification (using `minisat`)

```
prop_HalfAddOutputNeverBothTrue (a, b) = ok
 where
   (sum, carry) = halfAdd (a, b)
   ok           = Lava.nand2 (sum, carry)

prop_FullAddCommutative (c, (a, b)) = ok
 where
   out1 = fullAdd (c, (a, b))
   out2 = fullAdd (c, (b, a))
   ok = out1 Lava.<==> out2
```

```
main> Lava.minisat
  prop_HalfAddOutputNeverBothTrue
  >>= print

Minisat: ...
real    0m0.011s
user    0m0.001s
sys     0m0.002s
(t=) Valid.
Valid
```

Verification (using `nusmv`)

```
prop_SameAdderSeq inp = ok
 where
   out1 = adderSeq inp
   out2 = adderSeq' inp
   ok   = out1 Lava.<==> out2

main> Lava.smv prop_SameAdderSeq >>= print

Smv: ...
real    0m0.059s
user    0m0.005s
sys     0m0.009s
(t=) Valid.
Valid
```

Compare to simulation over all inputs:

```
main > Lava.simulateSeq
prop_SameAdderSeq Lava.domain

[high,high,high,high]
```

Questions to (Re)Consider

Consider the following when picking an HDL:

- What level of abstraction is being described?
- How are circuits modelled? Shallow or deep embedding?
- What kind of data can travel through your circuits/functions?
- How will you represent sequential logic?
- What components/basic components are allowed? How to represent them?
- How will you model check it?
- How will you simulate it?
- What can be generated? How does lowering occur? What ops are supported?
- Will you expose the clock? Multiple clocks? Allow asynchronity?

Personal Experience

Good things

- Access to all of Haskell's combinators and libraries
- Focus on verifiability with wrappers for external solvers/model checkers

Limitations

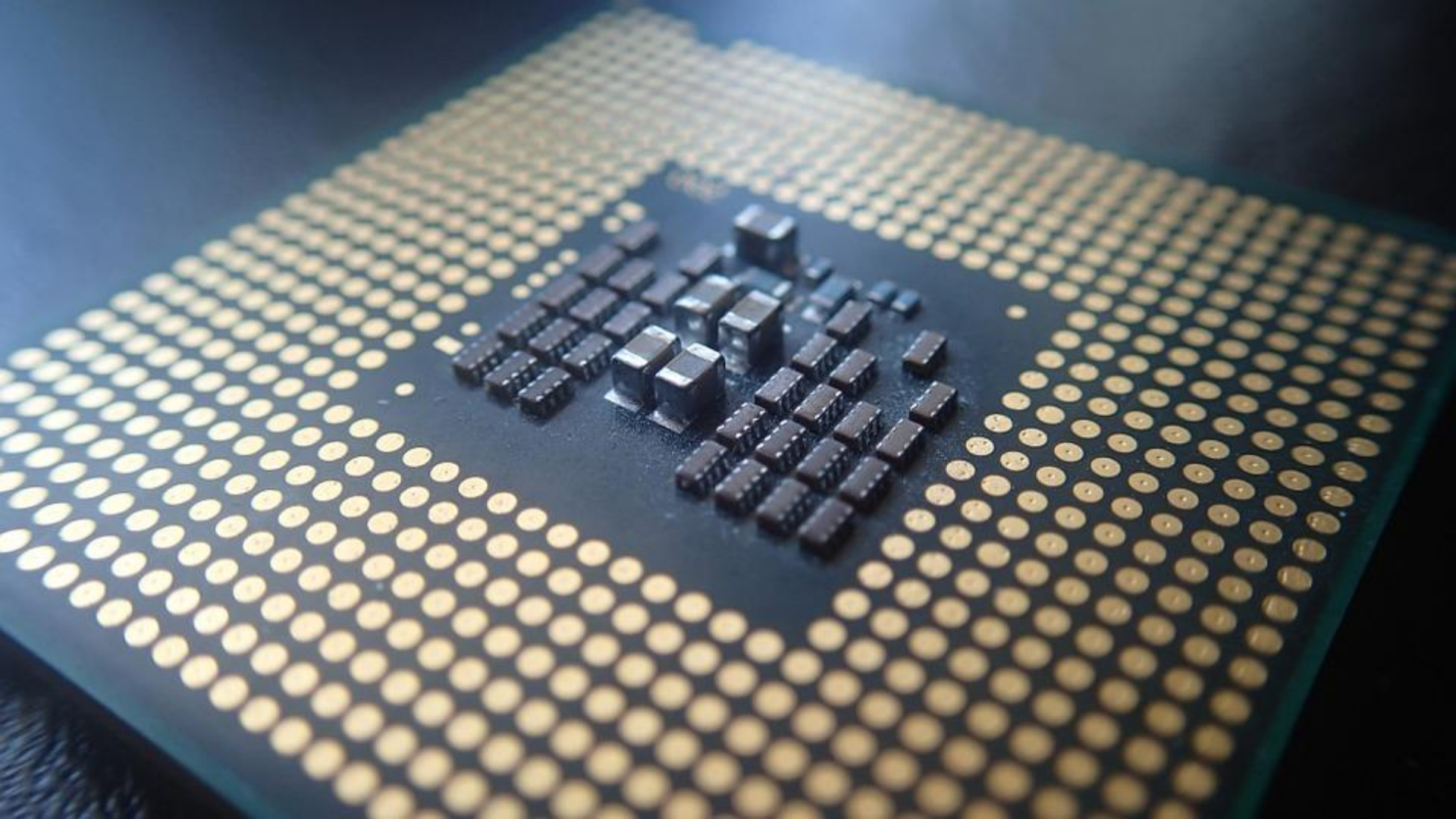
- Having to distinguish between `Signal Bool` and `Signal Int`
- Hard to synthesize custom data types (e.g. enum for mux control)
  - **What is simulatable is not always synthesizable**
- Uncurrying inputs/outputs (circuits are `(a -> b)`)
- List don't enforce size constraints

Lava Families

- Chalmers Lava (focus of this presentation)...
- Kansas Lava
  - `Signal` type (an applicative functor) for all types of signals (now in Chalmer's Lava)
  - Direct support for including existing VHDL libraries as primitives
  - Type functions to implement sized types
- York Lava
  - Recipes, RAMs, no model checking
- Xilinx Lava
  - Extra modules used for FPGA layout and hardware verification
  - Direct way to specify the mapping into CLB, slices and LUT
  - Tutorial and examples
- Cava
  - Coq-based Lava (theorem prover)

The End

https://github.com/pllab/Lava-Testing



References

- Satnam Singh's LAVA Resources:
  - https://fpcastle.com/lava/
- Chalmers University TDA956 Course site:
  - http://www.cse.chalmers.se/edu/year/2012/course/TDA956/
- Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. **Lava: hardware design in Haskell**. In Proceedings of the third ACM SIGPLAN international conference on Functional programming (ICFP '98). Association for Computing Machinery, New York, NY, USA, 174–184.
- Gill A., Bull T., Kimmell G., Perrins E., Komp E., Werling B. (2010) **Introducing Kansas Lava**. In: Morazán M.T., Scholz SB. (eds) Implementation and Application of Functional Languages. IFL 2009. Lecture Notes in Computer Science, vol 6041. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-16478-1_2
- Jones G., Sheeran M. (1993) **Designing arithmetic circuits by refinement in Ruby**. In: Bird R.S., Morgan C.C., Woodcock J.C.P. (eds) Mathematics of Program Construction. MPC 1992. Lecture Notes in Computer Science, vol 669. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-56625-2_15
- http://www.perbjesse.com/phd.pdf