Comparing functional Embedded Domain-Specific Languages for hardware description

João Paulo Pizani Flor

Department of Information and Computing Sciences, Utrecht University

February 13th, 2014

Universiteit Utrecht

Table of Contents

Introduction
Motivation
Hardware EDSLs
Analyzed EDSLs
Chosen EDSLs
Evaluation criteria
Modeled Circuits
ALU
Memory bank
CPU
Analysis of the EDSLs
Lava
ForSyDe
Coquet
Conclusions

Universiteit Utrecht

Section 1

Introduction

Hardware design

Hardware design is *very complex* **and** *very expensive*:

- Mistakes discovered after sales are much more serious
  - No such thing as an “update” to a chip
- Thus the need for extensive simulation
  - Sometimes even *exhaustive* simulation
  - Using specific and *expensive* systems
- The downfall of *Moore’s Law* doesn’t help either
  - More need for parallelism, fault-tolerance, etc.
  - Design *even more error-prone* and validation *even more* complex

Hardware design

- The level of abstraction has been lifted already...
  - Verilog and VHDL in the 1980s
  - Popular, *de facto* industry standards
- *Functional* hardware design languages, also since the 1980s
  - Expressive type systems, equational reasoning, etc.
  - First, languages designed *from scratch*
  - Then, *embedded* in general-purpose functional languages
    - Prominently, in Haskell
    - Several of them available nowadays
    - Each with its own strengths and weaknesses

Goals of this project

Compare exisiting functional Embedded Domain-Specific Languages (EDSLs) for hardware description.

- A representative sample of EDSLs
- Analyze a well-defined set of *criteria*
- Practical analysis, with a set of circuits as *case studies*

Universiteit Utrecht

Goals of this project

Compare exisiting functional Embedded Domain-Specific Languages (EDSLs) for hardware description.

- A representative sample of EDSLs
- Analyze a well-defined set of *criteria*
- Practical analysis, with a set of circuits as *case studies*

Let's first review out object of study

Domain-Specific Languages

A computer language (turing-complete or *not*) targeting a *specific application domain.*

Example DSLs:

- SQL (database queries)
- CSS (document formatting)
- MATLAB (Matrix programming)
- VHDL (Hardware description)

Universiteit Utrecht

Domain-Specific Languages

A computer language (turing-complete or *not*) targeting a *specific application domain.*

Example DSLs:

- SQL (database queries)
- CSS (document formatting)
- MATLAB (Matrix programming)
- VHDL (Hardware description)

A DSL can also be *embedded* in a general-purpose language.
Example EDSLs:

- Boost.Proto (C++ / parser combinators)
- Diagrams (Haskell / programmatic drawing)
- Parsec (Haskell / parser combinators)

Example of an EDSL: Parsec

A simple parser for a "Game of Life"-like input format:

```
dead, alive :: Parser Bool
dead  = fmap (const False) (char '.')
alive = fmap (const True) (char '*')

line :: Parser [Bool]
line  = many1 (dead <|> alive)

board :: Parser [[Bool]]
board = line `endBy1` newline

parseBoardFromFile :: FilePath -> IO [[Bool]]
parseBoardFromFile filename = do
    result <- parseFromFile board filename
    return $ either (error . show) id result
```

Hardware EDSLs

An EDSL used for hardware design-related tasks. Can encompass:

- Modeling / description
- Simulation (validation)
- Formal verification
- Synthesis to other (lower-level) languages

Example of a hardware EDSL (Lava):

```
halfAdder :: (Signal Bool, Signal Bool)
           -> (Signal Bool, Signal Bool)
halfAdder inputs = (xor2 inputs, and2 inputs)
```


Universiteit Utrecht

Section 2

Analyzed EDSLs

Choice criteria

Chosen EDSLs

The language we chose to evaluate, with the respective host language, were:

- Lava (Haskell - `chalmers-lava` *dialect*)
- ForSyDe (Haskell)
- Coquet (Coq interactive theorem prover)

Universiteit Utrecht

Evaluation criteria

- Simulation
- Verification
- Genericity
- Depth of embedding
- Tool integration
- Extensibility

Section 3

Modeled Circuits

Universiteit Utrecht

Choice criteria

- Not too simple, not too complex
- Familiar to any hardware designer
  - No signal processing, etc.
- Well-defined, pre-specification
  - Results to verify the models against

Chosen circuits

We cherry-picked circuits from the book "Elements of Computing Systems", as they satisfied all of our demands.

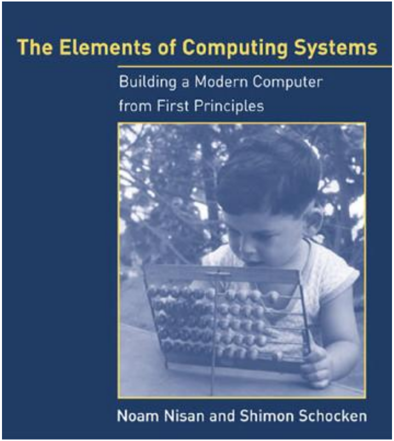


Figure: "Elements of Computing Systems" - Nisan, Schocken, available at `http://www.nand2tetris.org`.

Chosen circuits

Circuit 1 A 2-input, 16-bit-wide, simple ALU

Circuit 2 A 64-word long, 16-bit wide memory block

Circuit 3 An *extremely* reduced instruction set CPU, the *Hack* CPU.

Let's take a quick look at each of these circuit's specification...

Circuit 1: ALU

Some of the circuit's key characteristics:

- 2 *operand inputs* and 1 *operand output*, each 16-bit wide
- 1 *output flag*
- Can execute 18 different *functions*, among which:
  - Addition, subtraction
  - Bitwise AND / OR
  - Constant outputs
  - Addition of constants to an operand
  - Sign inversion

Universiteit Utrecht

Circuit 1: block diagram

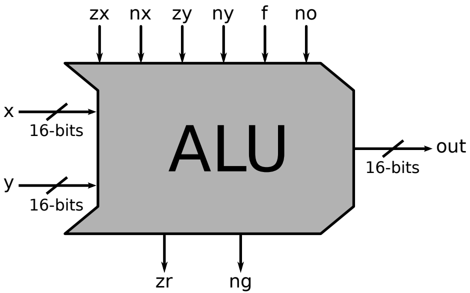


Figure: Input/Output ports of *circuit 1*, the ALU.

Circuit 1: specification

The behaviour of the ALU is specified by the values of the *control bits* and *flags*:

- **zx and zy** *Zeroes* the "x" and "y" inputs, respectively
- **nx and ny** *bitwise negation* on the "x" and "y" inputs
- **f** Selects the function to be applied: "f" $= 1$ for addition, "f" $= 0$ for bitwise AND
- **no** *bitwise negation* on the output ALU output
- **zr and ng** The output *flag* "zr" $= 1$ *iff* the ALU output is zero. "ng" $= 1$ *iff* the output is negative.

Formal definition and test cases in the book.

Circuit 2: RAM64

Some of the circuit's key characteristics:

- *Sequential* circuit, with clock input
- 64 memory words stored, each 16-bit wide
- Address port has width $\log_2 64 = 6$ bit

Circuit 2: block diagram

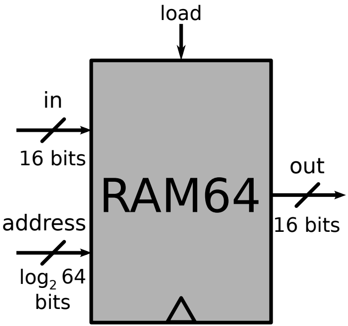


Figure: Input/Output ports of *circuit 2*, the RAM64 block.

Circuit 2: specification

- The output “out” holds the value at the memory line indicated by “`address`”.
- *Iff* “`load`” $= 1$, then the value at input “`in`” will be loaded into memory line “`address`”.
- The loaded value will be emitted on “out” at the *next* clock cycle.

Circuit 3: Hack CPU

- A *very* reduced instruction set CPU
  - Only 2 instructions: "`C`" and "`A`"
- Follows the *Harvard architecture*
  - Separate address spaces for *data* and *instruction* memory.
- Instructions are 16-bits wide
  - As well as the memory input and output
- Two *internal* registers: "`D`" and "`A`"

Circuit 3: block diagram

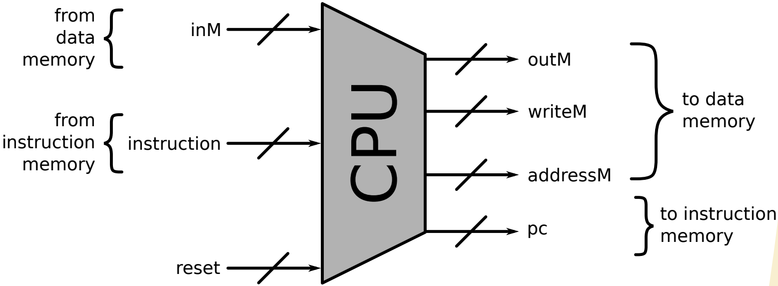


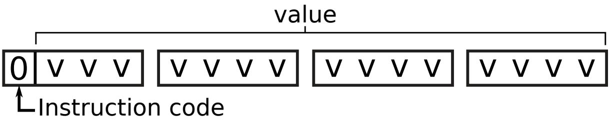
Figure: Input/Output ports of *circuit 3*, the *Hack* CPU.

Universiteit Utrecht

Circuit 3: specification

Circuit 3 runs "A" and "C" instructions, according to the *Hack assembly specification*.

- The "A" instruction: sets the "A" register.



- The value in "A" can be used:
  - As operand for a subsequent computation
  - As address for jumps

Universiteit Utrecht

Circuit 3: specification

Circuit 3 runs "A" and "C" instructions, according to the *Hack assembly specification*.

- The "C" instruction: sets the "C" register, performs *computation* or jumps.

			comp									*dest*		*jump*	
1	x	x	a	c1	c2	c3	c4	c5	c6	d1	d2	d3	j1	j2	j3
Instruction code															

- Some peculiarities:
  - Bits "c1" to "c6" control the ALU
  - *conditional* or *unconditional* jumps
  - *destination* of the computation result: "A", "D", "M"

Circuit 3: specification (parts)

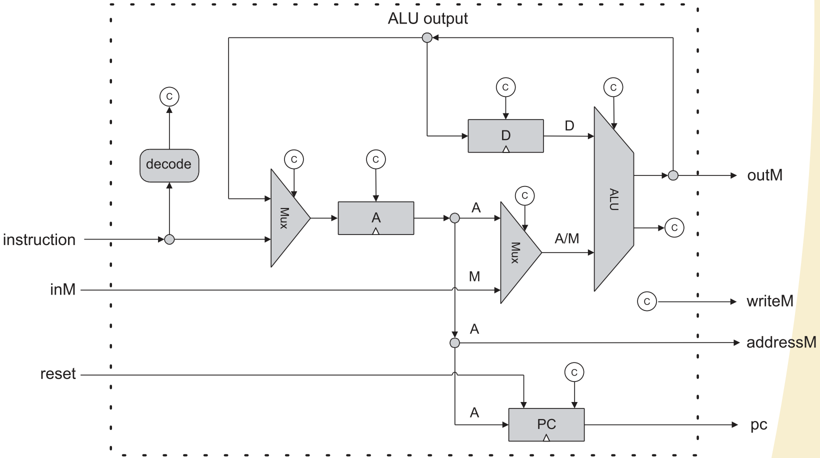


Figure: Parts used to build the *Hack* CPU, and their interconnection.

Section 4

Analysis of the EDSLs

Universiteit Utrecht

Lava

- Developed at Chalmers University of Technology, Sweden
  - Initially by Koen Claessen and Mary Sheeran
  - Later also Per Bjesse and David Sands
- Has several *dialects*
  - `chalmers-lava`, `xilinx-lava`, `kansas-lava`, etc.
  - We focus on the "canonical" `chalmers-lava`

Lava's *key* characteristics

- Deep-embedded
- Observable sharing
  - "Type-safe pointer equality" to detect sharing and recursion
  - Advantages and disadvantages clearer with examples
- Capable of simulation, verification and synthesis
  - Generates *flat* VHDL
  - External tools for verification
- Very "functional" style of hardware description
  - Will become clearer with examples

Universiteit Utrecht

Lava: Adders

```haskell
type SB = Signal Bool

halfAdder :: (SB, SB) -> (SB, SB)
halfAdder inputs = (xor2 inputs, and2 inputs)

fullAdder :: (SB, (SB, SB)) -> (SB, SB)
fullAdder (cin, (a, b)) = (s, cout)
    where
      (ab, c1) = halfAdder (a, b)
      (s, c2)  = halfAdder (ab, cin)
      cout     = or2 (c1, c2)

rippleCarryAdder :: [(SB, SB)] -> [SB]
rippleCarryAdder ab = s
    where (s, _) = row fullAdder (low, ab)
```

- Straightforward Haskell constructs
- "and2", "xor2", etc. are Lava's *atomic* circuits

Lava: Simulation and verification

- A taste of simulation in Lava:

```
type SB = Signal Bool
testHalfAdder :: [(SB, SB)]
testHalfAdder = map (simulate halfAdder) input
    where input = [ (low,low), (low,high)
                  , (high,low), (high,high)]
```

  - Cannot be easily automated: equality of `Signal` is non-trivial

- And verification...

```
prop_FullAdderCommutative :: (SB, (SB, SB)) -> SB
prop_FullAdderCommutative (c, (a, b)) =
    fullAdder (c, (a, b)) <==> fullAdder (c, (b, a))

-- satzoo prop_FullAdderCommutative
```

  - Advantage: Used in conjunction with an external SAT solver (e.g. *Satzoo*)
  - Disadvantage: Only verifies instances of *specific size*

Lava: ALU

```
type ALUControlBits = (SB, SB, SB, SB, SB, SB)

alu :: ([SB], [SB], ALUControlBits) -> ([SB], SB, SB)
alu (x, y, (zx, nx, zy, ny, f, no)) = (out', zr, ng)
    where x'     = mux (zx, (x, replicate (length x) low))
          x''    = mux (nx, (x', map inv x'))
          y'     = mux (zy, (y, replicate (length x) low))
          y''    = mux (ny, (y', map inv y'))
          out    = let xy'' = zip x'' y''
                   in mux (f, (andl xy'', adder xy''))
          out'   = mux (no, (out, map inv out))
          zr     = foldl (curry and2) low out'
          ng     = equalBool high (last out')
          adder = rippleCarryAdder
```

Remarks

- Cannot introduce new, meaningful datatypes
  - Only `Signal Bool` is synthesizable
  - Or tuples/lists thereof
- Input/Output types have to be *uncurried*
- Weak type-safety over the inputs/outputs
  - Working with tuples is tiresome and has limitations
  - Lists don't enforce *size* constraints

Lava: RAM64

```
reg :: (SB, SB) -> SB
reg (input, load) = out
    where dff = mux (load, (out, input))
          out = delay low dff

regN :: Int -> ([SB], SB) -> [SB]
regN n (input, load) = map reg $ zip input (replicate n load)

ram64Rows :: Int -> ([SB], (SB,SB,SB,SB,SB,SB), SB) -> [SB]
ram64Rows n (input, addr, load) = mux64WordN n (addr, regs)
    where memLine sel = regN n (input, sel <&> load)
          regs        = map memLine (decode6To64 addr)
```

Remarks

Positive:

- Uses host language for binding (`let`/`where`) and recursion
- Uses host language for structural combinators

Negative:

- Again, weak type-safety of lists
  - Extra `Int` parameter controls port *sizes*
- *No modularity* in the generated VHDL code.

Lava: *Hack* CPU (some parts)

```
programCounter :: Int -> (SB, SB, [SB]) -> [SB]
programCounter n (reset, set, input) = out where
    incr     = increment out
    out      = delay (replicate n low) increset
    incinput = mux (set, (incr, input))
    increset = mux (reset, (incinput, replicate n low))

type Dest     = (SB, SB, SB)
type JumpCond = (SB, SB, SB)
type CPUCtrl  = (SB, SB, Dest, JumpCond, ALUCtrl)

instructionDecoder :: HackInstruction -> CPUCtrl
instructionDecoder (i0,_,_,i3,i4,i5,i6,i7,i8,i9,...,i15)
    = (aFlag, cAM, cDest, cJump, cALU) where
    aFlag = i0
    cAM   = inv i3
    cDest = (i10, i11, i12)
    cJump = (i13, i14, i15)
    cALU  = (i4, i5, i6, i7, i8, i9)
```

Remarks

Could benefit from:

- Fixed-length vectors
  - ForSyDe-style or with type-level naturals in recent GHC.
- Slicing operators over vectors
- *Synthesizable* user-defined datatypes

Universiteit Utrecht

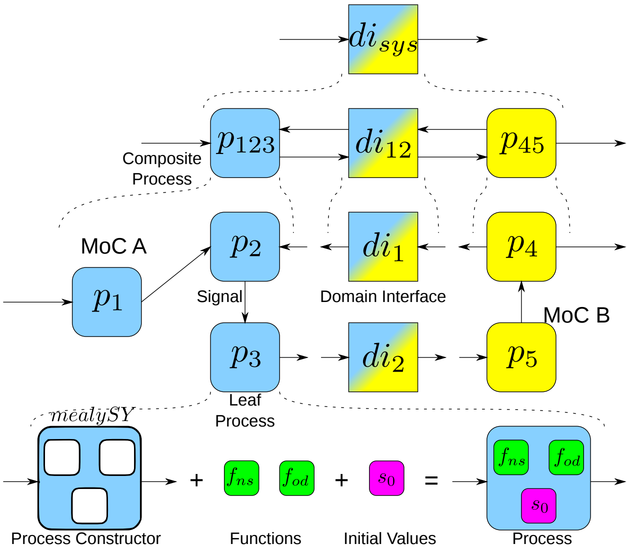
ForSyDe

- Based on the "Formal System Design" approach
  - Royal Institute of Technology - KTH, Sweden
- Available for Haskell and SystemC
- Has BOTH shallow and deep-embedded "versions"
  - Same library, subtle distinction
  - Will become clearer with examples
- *Template Haskell* to express circuits with Haskell syntax

ForSyDe's key concepts

- Models of Computation (MoCs)
  - We focus on the *synchronous* MoC
- Processes
  - A process belongs to a MoC
  - Built with a *process constructor*
- Signals
  - Connections among processes

ForSyDe's key concepts



ForSyDe: ALU (non-synth)

```haskell
type S = Signal
type Word = Int16

data ALUOp = ALUSum | ALUAnd
    deriving (Typeable, Data, Show)

$(deriveLift1 ''ALUOp)

type ALUCtrl = (Bit, Bit, Bit, Bit, ALUOp, Bit)
type ALUFlag = (Bit, Bit)

bo, bb :: Bit -> Bool
bo = bitToBool
bb = boolToBit
```

ForSyDe: ALU (non-synth)

```
aluFunc :: ProcFunc (ALUCtrl -> Word -> Word -> (Word,ALUFlag))
aluFunc = $(newProcFun [d|
  aluFunc' (zx,nx,zy,ny,f,no) x y =
      ( out,  (bb (out == 0), bb (out < 0)) )
    where
      zf z w    = if bo z then 0 else w
      nf n w    = if bo n then complement w else w
      (xn, yn) = (nf nx $ zf zx $ x,  nf ny $ zf zy $ y)
      out       = nf no $ case f of
                          ALUSum -> xn + yn
                          ALUAnd -> xn .&. yn  |] )

aluProc :: S ALUCtrl -> S Word -> S Word -> S (Word,ALUFlag)
aluProc = zipWith3SY "aluProc" aluFunc
```

ForSyDe: synthesis restrictions

Restrictions imposed on a model by ForSyDe so that it can be translated to VHDL:

- ProcFun-related:
  - Limited argument types (instances of `ProcType`)
  - `Int`, `Int8`, ..., `Bool`, `Bit`
  - Enumerated types (deriving `Data` and `Lift`)
  - Tuples and `FSVec`'s
- VHDL engine-related:
  - No point-free notation
  - Single clause / no pattern matching
  - No `where` or `let` bindings

ForSyDe: ALU (synthesizable)

```
zProc :: ProcId -> S Bit -> S Word -> S Word
zProc name = zipWithSY name $(newProcFun [d|
    f :: Bit -> Word -> Word
    f z w = if z == H then 0 else w |])

nProc :: ProcId -> S Bit -> S Word -> S Word
nProc name = zipWithSY name $(newProcFun [d|
    f :: Bit -> Word -> Word
    f n w = if n == H then negate w else w |])

compProc :: S Bit -> S Word -> S Word -> S Word
compProc = zipWith3SY "compProc" $(newProcFun [d|
    f :: Bit -> Word -> Word -> Word
    f o x y = if o == H then x + y else x .&. y |])

tzProc :: S Word -> S Bit ...
tnProc :: S Word -> S Bit ...
```

ForSyDe: ALU (synthesizable)

```
type ALUCtrl = (Bit, Bit, Bit, Bit, Bit, Bit)
type ALUFlag = (Bit, Bit)

aluProc :: S ALUCtrl -> S Word -> S Word -> S (Word, ALUFlag)
aluProc c x y =
    zipSY "aluProc" out (zipSY "flagsProc"
                            (tzProc out) (tnProc out))
  where
    (zx,nx,zy,ny,f,no) = unzip6SY "ctrlProc" c
    out  = nProc "no" no comp
    comp = compProc f (nProc "nx" nx $ zProc "zx" zx $ x)
                      (nProc "ny" ny $ zProc "zy" zy $ y)
```

ForSyDe: Muxes

```
mux2 :: S Bit -> S Word -> S Word -> S Word
mux2 = zipWith3SY "zipWith3SY" $(newProcFun [d|
  f s x y = if s == L then x else y |])

mux2SysDef :: SysDef (S Bit -> S Word -> S Word -> S Word)
mux2SysDef = newSysDef mux2 "mux2" ["s","i1","i2"] ["o"]

mux4 :: S (FSVec D2 Bit) -> S (FSVec D4 Word) -> S Word
mux4 ss is = (mux2' "m1") (sv ! d1) m00 m01 where
  mux2' l = instantiate l mux2SysDef
  sv      = unzipxSY "unzipSel" ss
  iv      = unzipxSY "unzipInp" is
  m00     = (mux2' "m00") (sv ! d0) (iv ! d0) (iv ! d1)
  m01     = (mux2' "m01") (sv ! d0) (iv ! d2) (iv ! d3)

mux4SysDef :: SysDef ( S (FSVec D2 Bit) -> S (FSVec D4 Word)
                     -> S Word)
mux4SysDef = newSysDef mux4 "mux4" ["s","is"] ["o"]
```

Remarks

Positive:

- Generated VHDL is very *modular*
  - One VHDL *entity* per ForSyDe component
  - Good for tool integration

Negative:

- Interface "conflicts" caused by FSVec and process constructors
  - "zip-unzip" pattern

Universiteit Utrecht

ForSyDe: RAM64

```
reg :: S Word -> S Bit -> S Word
reg input load = out where
    out = delaySY "delay" (0 :: WordType) dff
    dff = (instantiate "mux2" mux2SysDef) load out input

ram64 :: S Word -> S (FSVec D6 Bit) -> S Bit -> S Word
ram64 input addr load = mux' addr (zipxSY "zipRows" rs) where
  mux'       = instantiate "mux" mux64SysDef
  decoder'   = instantiate "decoder" decode6To64SysDef
  reg' l     = instantiate l regSysDef
  and' l     = instantiate l andSysDef
  r (s,l) = (reg' l) input ((and' (l ++ ":and")) load s)
  rs'     = unzipxSY "unzipAddr" $ decoder' addr
  rs      = V.map r $ V.zip rs' (V.map (\n -> "r" ++ show n)
                                (V.unsafeVector d64 [0..63]))

ram64SysDef = newSysDef ram64 "ram64" ["i","a","l"] ["o"]
```

Remarks

- Component *instantiation*
  - Introduces *hierarchy* in the design
  - Influences generated VHDL
- *Manual* name management
  - Error-prone
  - Every process must have a *unique* identifier
  - Already was a (lesser) issue with the muxes

ForSyDe: *Hack* CPU (part)

```haskell
type HackInstruction = FSVec D16 Bit
type Dest = (Bit, Bit, Bit)
type Jump = (Bit, Bit, Bit)

instructionDecoder :: S HackInstruction
                   -> S (Bit, Bit, Dest, Jump, ALUCtrl)
instructionDecoder = mapSY "mapSYdecoder" decoderFun where
  decoderFun = $(newProcFun [d|
    f :: HackInstruction -> (Bit, Bit, Dest, Jump, ALUCtrl)
    f i = ( i!d0
          , not (i!d3)
          , (i!d10, i!d11, i!d12)
          , (i!d13, i!d14, i!d15)
          , (i!d4,  i!d5,  i!d6, i!d7, i!d8, i!d9)
          ) |])
```


Universiteit Utrecht

Coquet

- Developed by Thomas Braibant (INRIA, France)
  - Seminal paper published in 2011
- Library embedded in the *Coq* proof assistant
  - Deep-embedded
  - Models the *architecture* of circuits
- Allows for *correctness proofs* of circuits
  - According to a given *specification*
  - Provides *tactics* to help with these proofs
  - More powerful, inductive proofs

Coquet: The Circuit type

```
Context {tech : Techno}
Inductive Circuit : Type -> Type -> Type :=
| Atom : forall {n m : Type} {Hfn : Fin n} {Hfm : Fin m},
             techno n m -> Circuit n m

| Plug : forall {n m : Type} {Hfn : Fin n} {Hfm : Fin m}
            (f : m -> n), Circuit n m

| Ser  : forall {n m p : Type},
             Circuit n m -> Circuit m p -> Circuit n p

| Par  : forall {n m p q : Type},
             Circuit n p -> Circuit m q
             -> Circuit (n + m) (p + q)

| Loop : forall {n m p : Type},
             Circuit (n + p) (n + p) -> Circuit n m
```


Universiteit Utrecht

Features from the `Circuit` type

- Circuit *structure* as constructors of the datatype
  - *Explicit* loops (recursion) as constructor
- Parameterized by one type of *fundamental gate* (`Atom`)
  - For example, `NOR` or `NAND`
- Circuit I/O ports are defined by *finite* types
  - Instances of the "`Fin`" typeclass

Universiteit Utrecht

Coquet: circuit example

```
Definition HADD a b s c: circuit ([:a]+[:b]) ([:s]+[:c]) :=
    Fork2 ([:a] + [:b])  |>  (XOR a b s  &  AND a b c).


Program Definition FADD a b cin sum cout :
    circuit ([:cin] + ([:a] + [:b])) ([:sum] + [:cout]) :=

   (ONE [: cin]  &  HADD a b "s" "co1")
|> Rewire (*  (a, (b,c)) => ((a,b), c)  *)
|> (HADD cin "s" sum "co2" & ONE [: "co1"])
|> Rewire (*  ((a,b), c) => (a, (b,c))  *)
|> (ONE [:sum]  &  OR "co2" "co1" cout).

Next Obligation. revert H; plug_def. Defined.
Next Obligation. plug_auto. Defined.
Next Obligation. revert H; plug_def. Defined.
Next Obligation. plug_auto.Defined.
```


Universiteit Utrecht

Features from the example

- Circuit I/O types (finite types)
  - Parameterized by strings: *tagged units*
  - Default "`Fin`" instances for sums, units
- Serial/Parallel composition
- *Associativity plugs* (reordering) automatically defined
  - With help of proof search

Coquet: How to prove correctness

To understand Coquet proofs, we need 2 concepts:

- Meaning relation
  - Circuit $\rightarrow$ *Prop*
- Behavioural specification
  - What should a circuit *do* with its inputs

Let's take a look at the definition for each of these concepts...

Coquet: Meaning relation

```
Inductive Sem : forall {n} {m},
  C n m -> (n -> Data) -> (m -> Data) -> Prop :=

| KAtom: forall n m {Hfn: Fin n} {Hfm: Fin m}
         (t: techno n m) i o, spec t i o -> Sem (Atom t) i o

| KSer: forall n m p (x: C n m) (y: C m p) i mid o,
        Sem x i mid -> Sem y mid o -> Sem (Ser x y) i o

| KPar: forall n m p q (x: C n p) (y: C m q) i o,
           Sem x (select_left i) (select_left o)
        -> Sem y (select_right i) (select_right o)
        -> Sem (Par x y) i o

| KPlug: forall n m {Hfn: Fin n} {Hfm: Fin m} (f: m -> n) i,
         Sem (Plug f) i (Data.lift f i)

| KLoop: forall n m l (x: C (n + l) (m + l)) i o ret,
            Sem x (Data.app i ret) (Data.app o ret)
         -> Sem (Loop x) i o
```


Universiteit Utrecht

Coquet: Specification

```
Context {n m N M : Type}
        (Rn : Iso (n -> T) N) (Rm : Iso (m -> T) M).

Class Realise (c : Circuit n m) (R : N -> M -> Prop) :=
  realise: forall i o, Semantics c i o -> R (iso i) (iso o)

Class Implement (c : Circuit n m) (f : N -> M) :=
  implement: forall i o, Semantics c i o -> iso o = f (iso i)
```

The semantics of a circuit *entails* (implies):

- A *relation* between inputs and outputs
- The application of a *function* to the inputs
- Up to isomorphisms...

Now for a (small) example of correctness proof...

Coquet: Correctness proofs

```
Instance HADD_Implement {a b s c} :
    Implement (HADD a b s c) _ _
    (fun (x : bool * bool) =>
        match x with (a,b) => (xorb a b, andb a b) end).
Proof.
    unfold HADD; intros ins outs H; tac.
Qed.
```


Universiteit Utrecht

Coquet: How to prove correctness

```
Ltac tac :=
    rinvert;                    (* destruct the circuit *)
    realise_all;                (* use the hint data-base *)
    unreify_all bool;           (* unreify *)
    destruct_all;               (* destruct the booleans *)
    intros_all;
    clear;
    boolean_eq.
```

Section 5

Conclusions

Results

Future work

Universiteit Utrecht

Thank you!

Questions?