Comparing functional Embedded Domain-Specific Languages for hardware description

João Paulo Pizani Flor
Department of Information and Computing Sciences,
Utrecht University - The Netherlands
e-mail: j.p.pizaniflor@students.uu.nl

Friday 29th November, 2013

1 Introduction

Hardware design has become a very complex activity. The size of circuits has increased, while low-level concerns (power consumption, error correction, parallelization, layout, etc.) have to be incorporated earlier and earlier in the design process. This breaks modularity and makes it harder to validate and verify the correctness of circuits.

In this context, researchers have been suggesting (since the 1980s) the usage of functional programming languages to model circuits. One particular line of research is to create Embedded Domain-Specific Languages for hardware description based on existing functional programming languages, such as Haskell.

There are a multitude of EDSLs for hardware description, but they vary wildly on a number of aspects: host language, level of abstraction, capabilities of simulation, formal verification, synthesis (generation of netlists) and integration with other tools, to name a few. All this variety can make the task of choosing a hardware EDSL for the task at hand daunting and time-consuming.

The main goal of this experimentation project is to establish some order in this landscape, and to perform a practical analysis of some popular functional hardware EDSLs. By reading the materials produced in this project (circuit models, test cases, generated netlists, report), a hardware designer wishing to use a functional hardware EDSL for his next design should gain some insight about the strengths and weaknesses of each language and have an easier time choosing one.

As an additional result of this research, we intend to identify recent, cutting-edge developments in the Haskell language and its implementations from which the analyzed EDSLs could benefit. Also, we intend to discuss to which extent some shortcomings of the EDSLs could be overcome by having them hosted in a dependently-typed language.

2 Methodology

In this project, we compared a number of functional hardware EDSLs that we considered representative (more details on the choice of EDSLs further ahead). The comparison was performed on a number of *aspects* for each EDSL, and the analysis was done by considering a *sample set of circuits* used as case studies.

We tried to model all circuits in all EDSLs considered, and as similarly as possible in each EDSL. To avoid using any of the analyzed EDSLs as "base" for analysis, we provide a neutral, behavioural description of the circuits.

2.1 The languages

The embedded Hardware Description Languages we decided to analyze are:

Lava The Lava[3] language, developed initially at Chalmers University in Sweden. Lava is deeply embedded in Haskell, and provides features such as netlist generation and circuit verification using SAT-solvers. There are several "dialects" of Lava available, and the one used for this project is considered the "canonical" one, originally developed at Chalmers.

ForSyDe The Haskell ForSyDe library is an EDSL based on the "Formal System Design" approach[10], developed at the swedish Royal Institute of Technology (KTH). It offers both shallow and deep embeddings, and provides a significantly different approach to circuit modelling, using *Template Haskell* to allow the designer to describe combinational functions with Haskell's own constructs.

Coquet The Coquet[4] EDSL differs from the other 2 mainly because it's embedded in a dependently-typed programming language (the Coq theorem prover). Coquet aims to allow the hardware designer to describe his circuits and then *interactively* prove theorems about the behaviour of whole *families* of circuits (using proofs by induction).

2.2 The aspects evaluated

For each of the hardware description EDSLs we experimented with, a number of *aspects* were evaluated. The evaluated aspects do not necessarily make sense for *all* EDSLs, therefore our presentation follows a language-centric approach, in which we expose the strengths and weaknesses of each EDSL concerning the applicable aspects.

Without further ado, the following aspects are considered in the analysis:

Simulation The capability of simulating circuits modeled in the EDSL (and the ease with which it can be performed). Simulation is understood in this context as *functional* simulation, i.e, obtaining the outputs calculated by the circuit for certain input combinations.

Verification The capability of verifying *formal properties* concerning the behaviour of circuits (and the ease with which verification can be performed). The properties we are interested in are those which are *universally quantified* over the circuit's inputs. As an example of such a property, we might have:

$$\forall a \forall b \forall sel\, (\mathrm{MUX}(a, b, sel) = a) \vee (\mathrm{MUX}(a, b, sel) = b)$$

Genericity Whether (and how well) the EDSL allows the modelling of *generic* or *parameterized* circuits. An example of a generic circuit is a multiplexer with 2 n-bit inputs and 1 n-bit output, or a multiplexer with n 1-bit inputs and 1 1-bit output. Besides parametrization in the *size* if inputs and outputs, we will also analyze whether the EDSL provides chances for parametrization on other functional and/or non-functional attributes.

Depth of embedding Whether the EDSL models circuits with a *shallow* embedding (using predicates or functions of the host language), a *deep embedding* (in which circuits are members of a dedicated data type), or anything in between. The depth of embedding of an EDSL might have consequences for other aspects being analyzed.

Integration with other tools How well does the EDSL allow for interaction with (getting input from / generating output for) other tools in the hardware design process. For example, synthesis tools for FPGAs or ASICs, timing analysis tools, model checkers, etc.

Extensibility The extent to which the user can *add* new interpretations, data types, and combinator forms to the language. For example, the user might want to model circuits that consume and produce custom datatypes, or might be interested in extracting *metrics* from a circuit such as power consumption, number of elementary gates, etc.

3 Modeled circuits

When thinking of which circuits to model using the analyzed EDSLs, some principles guided us. First of all, they shouldn't be too simple but also not too complex. Some very simple circuits (adders, counters, etc.) are very often shown as examples in the papers that define the EDSLs themselves, as well as in tutorials. On the other hand, we also did not want to model too complex circuits; that would require too much effort on the hardware design itself, and diverge from the focus of this project, which is to evaluate and analyze the EDSLs.

Another principle that guided our choice is that the circuits should be immediately familiar to anyone with some minimal experience in hardware design. We avoided, therefore, considering application-specific circuits such as those for Digital Signal Processing (DSP), implementing communication protocols, etc. Having ruled out this class of circuits, we were left to choose from circuits that form a general-purpose computing machine, such as arithmetic units, memory blocks, control units and so forth.

Finally, we wanted to choose among circuits that already had a well-defined, *behavioural* description, to avoid using any of the analyzed EDSLs as "basis" of comparison.

Taking these considerations into account, we chose to implement, in each of the EDSLs analyzed, three circuits originating from the book "The Elements of Computing Systems"[7]. This book aims to give the reader a deep understanding of how computer systems work by taking a hands-on approach, in which the reader is given the most basic logic gates and builds, step-by-step, all the hardware and software components necessary to implement a complete computer system.

From the hardware design part of the book, we took our three circuits to be modeled:

- A simple Arithmetic Logic Unit (ALU), from here onwards referred to as "circuit 1".
- A RAM memory block with 64 words, from here onwards referred to as "circuit 2".
- A CPU with an extremely reduced instruction set (capable of executing the *Hack* assembly language defined in the book) from here onwards referred to as "circuit 3".

3.1 Circuit 1: ALU

The Arithmetic Logic Unit built by us is a 2-input ALU, in which each of the inputs (as well as the output) is a 16-bit long word (interpreted as two's-complement signed integer). It is capable of computing several functions, and the choice of which function to compute is made by setting the ALU's 6 *control bits*. To become more familiar with this circuit, let's first take a look at its block diagram, shown in figure 1

Each of the 6 control bits to the ALU has, in isolation, a well-defined effect on the inputs or outputs to the ALU core. The bits `(zx, nx, zy, ny)` control "pre-processing" steps for the inputs `x` and `y`, with the following behaviour:

zx and zy *Zeroes* the x input (respectively y). The ALU core will receive `0` as input.

nx and ny Performs *bitwise negation* of input x (respectively y).

Therefore, the ALU "core" itself (adder, and gate) has, as inputs, the results of performing these pre-processing steps controlled by `(zx, nx, zy, ny)`. Furthermore, the *output*

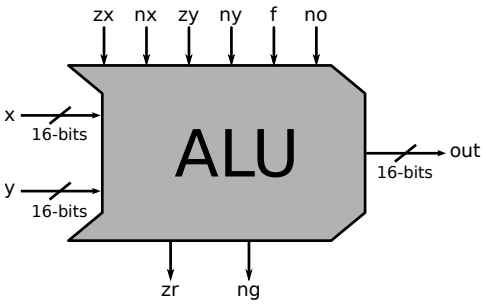

Figure 1: Block diagram of circuit 1, showing its input and output ports.

of the ALU core can also be *bitwise negated* as a "post-processing" step, controlled by bit no.

Finally, the control bit f can be used to select which operation is to be performed by the ALU core: if we wish to add the two inputs, we need to set $f = 1$, and if we want bitwise conjunction, then we need to set $f = 0$.

Besides the main (16-bit wide) output of the ALU, there are two other output *flags*, that indicate predicates over the main output:

zr Is high whenever $out = 0$.

ng Is high whenever $out < 0$.

When the ALU is used in the context of a microprocessor these flags can be used, for example, to facilitate conditional jumps.

Even though there are $2^6 = 64$ possible combinations for the values of the control bits, only 18 of these combinations result in interesting functions – that is because several combinations of control bits can be used to calculate the same function. We show these 18 functions that the ALU can calculate on table 3.1.

zx	**nx**	**zy**	**ny**	**f**	**no**	**out=**
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	1	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	$\neg x$
1	1	0	0	0	1	$\neg y$
0	0	1	1	1	1	$-x$
1	1	0	0	1	1	$-y$
0	1	1	1	1	1	$x + 1$
1	1	0	1	1	1	$y + 1$
0	0	1	1	1	0	$x - 1$
1	1	0	0	1	0	$y - 1$
0	0	0	0	1	0	$x + y$
0	1	0	0	1	1	$x - y$
0	0	0	1	1	1	$y - x$
0	0	0	0	0	0	$x \wedge y$
0	1	0	1	0	1	$x \vee y$

Table 1: Functions that the ALU can calculate, given different settings of the control bits

3.2 Circuit 2: RAM64

Circuit 2 is a block of RAM with 64 lines, in which each line is a 16-bit word. Actually, using the term "RAM" to refer to this component is an abuse of terminology, as this circuit is nothing more than a register bank.

All the input and output ports of the circuit are pictured in its block diagram, shown in figure 2.

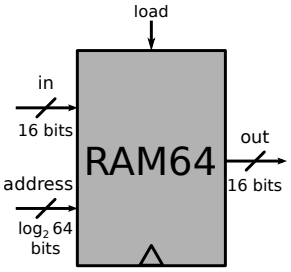

Figure 2: Block diagram of circuit 2, a RAM of 64 lines

The circuit has one 16-bit output, named `out`, and three inputs (`in`, `address` and `load`). The `in` port is 16-bit wide and holds a value to be written into the RAM. The `address` port has a width of $\log_2 64 = 6$ bits and holds the address in which reading or writing is to be performed. Finally, the `load` bit controls whether the value currently at `in` should be written to the selected address. There is also one *implicit* input for a clock signal in this component. Implicit, in this case, means that the clock signal is not present in any of the models that we developed for this circuit, but must be present in any physical implementation.

The temporal behaviour of this memory block is as follows: At any point in time, the output `out` holds the value stored at the memory location specified by `address`. If the `load` bit is high, then the value at `in` is loaded into the memory word specificied by `address`. The loaded value will then be emitted on the output at the **next** clock cycle.

3.3 Circuit 3: The Hack CPU

Circuit 3, the largest and most complex circuit among the ones we have chosen to implement, is the Central Processing Unit for the *Hack* computer, the machine described in the book "The Elements of Computing Systems"[7].

The Hack computer is based on the *Harvard architecture*, that means that it has different storage components and signal pathways for instructions and data. Therefore, the Hack CPU expects to be connected to *two* memory blocks, the instruction memory and the data memory. Having this in mind facilitates the understanding of the CPU's block diagram, shown in figure 3

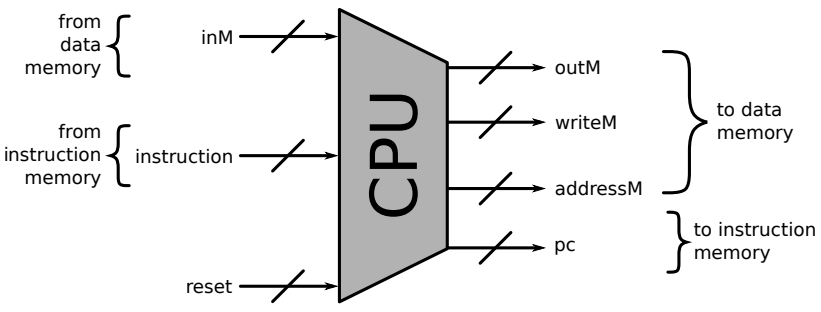

Figure 3: Block diagram of circuit 3, the Hack CPU

The Hack architecture has an *extremely* reduced instruction set, and consists in fact of only two instructions (each 16-bit wide): A (meaning “address”) and C (meaning ”compute”). The A instruction can be used as a means to load numerical literals into the data memory, as well as setting a special “cache” register inside the CPU. The C instruction is the one responsible for effectively performing computations using the ALU, testing outputs and jumping. More details about programming in the Hack assembly language can be found in [8].

The meaning of each of the CPU’s input and output ports becomes much clearer when we look at the context in which the CPU is inserted, namely, the memory modules to which it is connected. So, let’s analyze the CPU’s ports by taking a look at figure 4.

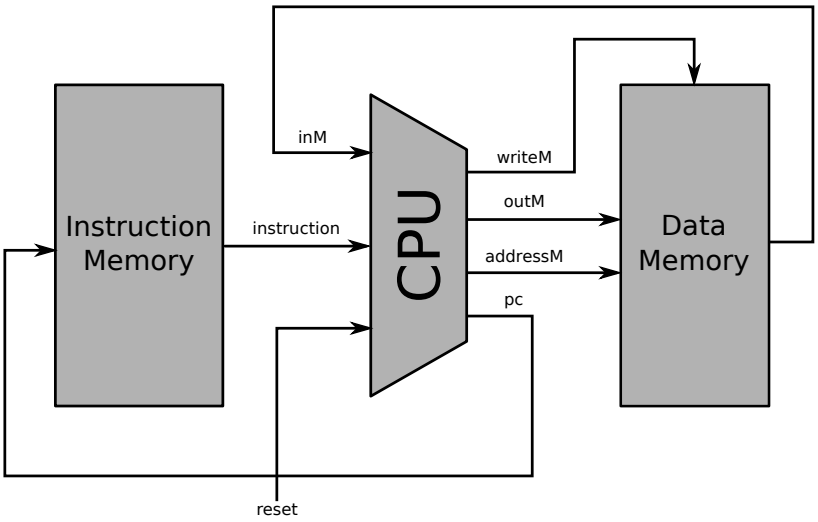

Figure 4: The Hack CPU connected to the data and instruction memory blocks

Finally, the CPU is a circuit which is built mostly from the parts we already defined in circuits 1 and 2. We use the ALU, some registers, multiplexers, an instruction decoder and a counter (the *program counter*). Figure 5 shows the CPU organization.

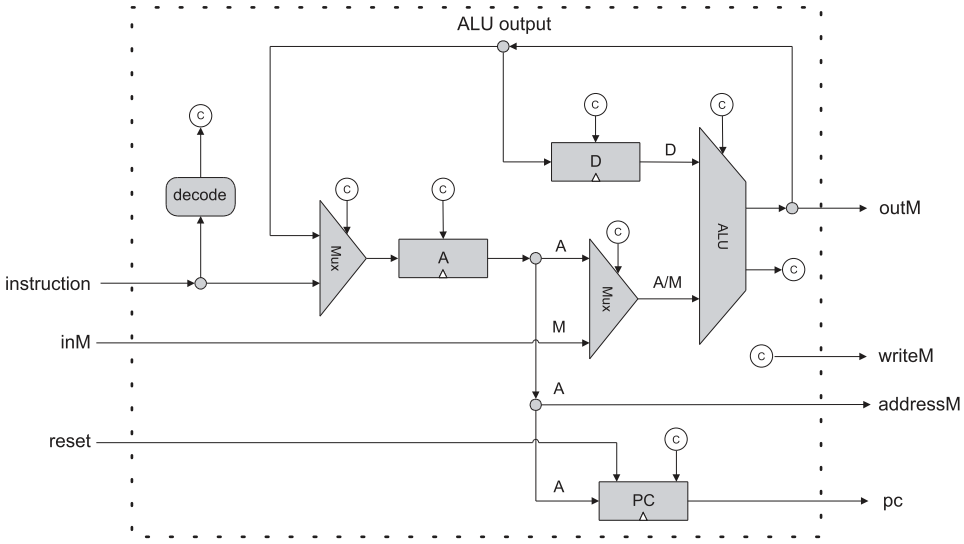

Figure 5: Parts used in building the CPU circuit and how they are connected.

4 Analysis of the EDSLs

4.1 Lava

Lava[3] is an EDSL for hardware description developed originally around 1998 at Chalmers University of Technology, in Sweden. It uses Haskell as the host language, and circuits described in Lava are *deeply embedded*.

The Lava EDSL has several "dialects", among which are Xilinx-Lava, York-Lava, Kansas-Lava and Chalmers-Lava. Xilinx-Lava[11] was developed by Satnam Singh and puts a greater emphasis on the *layout* of the described circuits, focusing on their implementation in Xilinx's FPGAs. York-Lava was developed as part of the Reduceron[6] project, and is a variation of Chalmers-Lava, omitting some features and adding some others, like a "Prelude" of commonly used circuits ((de)multiplexers, (de)coders, RAM memory blocks, etc.). Chalmers-Lava is considered the "standard" dialect, also being the one which was first developed, therefore it was chosen as the one to be studied in this project.

Before diving into the inner workings of the Chalmers Lava library, we first need to make clear that there are two very distinct versions of this library. The original paper that defines the Lava language[3] contains the first version, while the current version is the one defined in a later paper[5] by Koen Claessen and David Sands. This current version of Chalmers Lava is the one in which our case study is implemented.

As already said, Lava uses a *deep embedding*, and the datatype used to represent a circuit is `Signal`, defined in listing 1.

```
newtype Signal a = Signal Symbol
newtype Symbol = Symbol (Ref (S Symbol))

data S s
    = Bool      Bool
    | Inv       s
    | And       [s]
    | Or        [s]
    | Xor       [s]
    | VarBool   String
    | DelayBool s s  -- other constructors...
```

Listing 1: Lava's `Signal` datatype, used to represent circuits.

As can be noticed from the definition, the *actual* circuit type (`S`) is "wrapped" around the `Ref` type constructor. This has to do with the approach that Lava takes to solving the problem of observable sharing, which is to rely on comparing *references to object* given by the Haskell implementation, to detect cycles in syntax graphs representing circuits. This approach is the cause of some of Lava's advantages as well as disadvantages, which will be discussed further ahead.

Having defined a circuit operating on values of a type `a` to have type `Signal a`, then there are several circuit combinators provided by Lava, which take circuits as inputs and provide circuits as outputs. For example, on figure 2 we show some boolean circuit combinators:

With Lava, one can also model circuits operating on `Int`s (and there are several interesting integer circuit combinators already included in the Lava library). However, our goal in this project was to model *boolean* circuits and, besides that, integer circuits offer a reduced set of features.

```
bool :: Bool -> Signal Bool
bool b = lift0 (Bool b)

low, high :: Signal Bool
low  = bool False
high = bool True

inv :: Signal Bool -> Signal Bool
inv = lift1 Inv

andl, orl, xorl :: [Signal Bool] -> Signal Bool
andl = liftl And
orl  = liftl Or
xorl = liftl Xor

and2 (x, y) = andl [x, y]
or2  (x, y) = orl  [x, y]
xor2 (x, y) = xorl [x, y]

nand2 = inv . and2
nor2  = inv . or2
xnor2 = inv . xor2
```

Listing 2: Some of Lava's boolean circuit combinators.

4.1.1 Circuits modeled

In order to be able to describe circuit 1, the ALU (described in more detail on section 3.1), we first needed to model the necessary parts. The "core" of the ALU is composed of a 16-bit ripple-carry adder and a 16-bit AND gate. To model the ripple-carry adder we used full adders as parts, which in turn used half adders. To get used to the wat in which circuits are described in Lava, let us first take a look at the definition of the hierarchy of adders in listing 3:

```
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)
```

Listing 3: Hierarchy of adders used in circuit 1.

Based on this small model we can already make some observations concerning the aspects that we are analyzing. These observations are:

- All circuits in Lava must be modeled as *uncurried* functions, that is, if multiple inputs are needed, they need to be packed into one tuple, the same "packing" happens also in the case of multiple outputs.
- The *basic* type of input/output for all circuits modeled is `Signal Bool`. This is not coincidental: Lava's VHDL generation backend can only work with circuits whose input/output types are `Signal Bool` or any nested combination of tuples and lists thereof. This limitation makes Lava have low **extensibility**, not allowing – for example – user-defined types.
- In Lava, (families of) circuits with variable-sized inputs/outputs are modeled as lists (as can be seen in the definition of `rippleCarryAdder`). This approach has a good **genericity**, but is **not type-safe enough**. For example, we could have a circuit assuming that its inputs are 32-bit wide. There is no way to enforce, at *Haskell compilation time*, that inputs with correct size are provided. Possible problems could only be detected at simulation or VHDL generation.

Now, after having defined all the necessary parts, lets take a look at the ALU circuit itself in listing 4:

```
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
```

Listing 4: Top-level model for circuit 1, the ALU.

In the definition of the ALU itself, we would like to have a user-defined datatype to represent the kinds of functions that can be computed by the ALU (functions listed on table 3.1). However, due to the limitations of the VHDL backend already discussed, we have to define `ALUControlBits` as simply a *type synonym* for a 6-tuple of bits.

Besides modelling the three circuits in Lava, we also simulated them. The definition of the ALU circuit in the book "The Elements of Computing Systems"[7] has a pretty extensive truth table to test the circuit model, which was used to simulate the ALU. However, let's take a look at a simpler simulation case, that of a half-adder, in figure 5:

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

Listing 5: Simulation of a half adder in Lava.

Simulation of combinational circuits is performed by the Lava function `simulate`; it takes as arguments the circuit to simulate and an input combination. In the example of simulation for the `halfAdder`, we *map* the simulation over a list of input combinations, covering all possible cases.

The attentive reader might be asking why is this simulation not an automated test, i.e, why are we **not** comparing the results of the simulation with an *expected* output sequence. This has to do with the way in which Lava handles the problem of observable sharing: values of type `Signal a` encapsulate effectively a *runtime reference* to an object of type `a`. Therefore, even though *actual* and *expected* outputs might appear to be equal, they are considered different by Lava. Here is the offending `Eq` instance from the Lava library (module `Lava.Signal`):

```
ance Eq (Signal a) where
Signal (Symbol r1) == Signal (Symbol r2) = r1 == r2
```

With the drawback of not having *automated* testing, we can say that Lava does provide good **simulation** capabilities, with an interface that is easy to understand for functional programmers.

Now, before moving on to the next circuit studied, let's take a look at how Lava handles *formal verification* with two examples: checking that a full adder is commutative and that the output of an incrementer circuit is always different from its input:

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

A property over a circuit in Lava is modeled as a circuit containing *one boolean output*, which – for the property to be true – needs to be *true* for any combination of inputs (these properties are called *safety properties*). Lava performs the verification by converting the circuit model to a CNF logical formula and executing an external SAT solver on the negation of the formula: the property being verified is *valid* if and only if the negated formula is unsatisfiable. The verification for the incrementer introduces another detail of this kind of verification:

```
prop_IncrementIsAlwaysDifferentThanInput :: Int -> Property
prop_IncrementIsAlwaysDifferentThanInput n =
        forAll (list n) (\x -> prop x)
    where prop x = inv (x <==> increment x)
```

We can see by the type of the verification function that it is a *property generator*, i.e, for each integer `n`, it gives a property. An incrementer is a circuit with generic input/output size, but the SAT-solving approach to verification can only prove properties for *circuits of fixed size*. Therefore, we can only verify a finite number of particular instances of the circuit.

Moving on to circuit 2 (the RAM block), we will take a look at how Lava handles sequential circuits. The "fundamental"[1] sequential circuit in Lava is `delay`. It takes two boolean signals as input and outputs a single boolean signal. Its semantics is that the output signal will correspond tho the input signal *delayed* by one clock cycle, with the other parameter being the first value of the output. Using this fundamental circuit, we modeled the first building block of our hierarchy of memory elements: a 1-bit register with input and load signals:

[1] here, *fundamental* means that all sequential circuits have – directly or indirectly – `delay` as one of its building blocks

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

In this model, we use a `mux` to control whether the next state of the output will be simply the previous state, or the input value will be "loaded" into the register. Now, a 1-bit register can easily be "lifted" into a generic n-bit circuit:

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

The `regN` definition is *generic*, and parameterized by the size of the input and output (`n`). This means that *for each value of n, there is a circuit* `regN n`. In Lava, however, we can only simulate and generate VHDL for specific instances of this family of circuits. The restriction with regards to VHDL generation is not a theoretical limitation, that because VHDL has good support for *generic components*, and one could imagine Lava generating generic VHDL from generic circuit models. But, leaving that discussion aside, let's take a look at the simulation case for `regN`:

```
testRegN4 :: [[Signal Bool]]
testRegN4 = simulateSeq (regN 4) ins
    where los = replicate 4 low
          his = replicate 4 high
          ins = [(los,high), (his,low), (his,low), (his,high), (los,low)]
```

The `simulateSeq` function is intended for the simulation of sequential circuits: the list of inputs its given is the sequence of values present at the input ports of the circuit under test – one element of the list for each clock cycle. The list of outputs given by `simulateSeq` has a similar interpretation.

Having the core sequential component for our memory bank (`regN`), we modeled some other helper components (such as an *address decoder* and a 64-to-1 *multiplexer*). With all the components at our disposal, we then modeled the RAM block itself:

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

All the registers in the memory bank are connected to the "global" input word for the bank, but the load signal for any particular register is active *iff* the global `load` signal is active *and* (`<&>`) that particular memory line is selected. Finally, to be precise, `ram64Rows` actually defines a family of circuits, one for each value of `n`. The one we are interested in is `ram64Rows 16`, for a RAM block with 64 lines, in which each line is 16 bits wide.

Finally, the last circuit we studied under Lava (circuit 3) is the Hack CPU (described in more detail on section 3.3). The CPU circuit is mostly combinational, as it contains *no form of pipelining* and exactly one instruction is executed per clock cycle. However, there is one sequential component of the CPU: the *program counter*, shown in figure 6:

The program counter counts cyclically between 0 and 2^{n-1}, and can have its value reset or set to a particular value at any moment. Having defined the program counter, there are still some helper parts to define before writing the model for the CPU itself: most importantly, we need an *instruction decoder*, responsible for interpreting each `Hack` instruction and outputting several *control bits* that are used to direct the data flow inside the CPU during each instruction execution cycle:

Here we notice again some limitations of Lava with regards to datatypes: We are limited to lists and tuples of `Signal Bool` (to keep the circuit synthesizable). Here we decided to

```
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))
```

Listing 6: Lava model for the program counter inside the Hack CPU.

```
type DestBits       = (SB, SB, SB)
type JumpCondBits   = (SB, SB, SB)
type CPUControlBits = (SB, SB, DestBits, JumpCondBits, ALUControlBits)

instructionDecoder :: HackInstruction -> CPUControlBits
instructionDecoder (i0,_,_,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,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)
```

Listing 7: The instruction decoder of the Hack CPU.

model the *Hack* instruction itself and the control flags as tuples, to prevent size-related runtime errors. However, using tuples made the model more "cluttered", as tuples are *not* data structures prone to slicing and regrouping.

Using *fixed-length vectors*, perhaps based on the recent "Type-level Naturals" GHC [2] extension [2] (introduced in GHC 7.6 and being improved for GHC 7.8) would make modelling in Lava *safer* and more comfortable.

4.2 ForSyDe

The Haskell ForSyDe library is an implementation of the "Formal System Design" approach to hardware modelling[10]. The ForSyDe approach per se has several significant differences when compared to Lava, and even when the two EDSLs agree on *what to do*, sometimes they differ on how to achieve those goals.

To better understand what characterizes the ForSyDe methodology, we first have to establish some vocabulary:

System In ForSyDe, a system or circuit is a set of processes interconnected by *signals*.

Signal A signal is, intuitively, a stream of information that flows between processes. It carries events of some type, and each event has an associated *tag*. The meaning of the tag is defined by the *model of computation* used.

Process A process is nothing more than a pure function on signals. A process *is able* to hold internal state. But, given the same input (possibly infinite) signals, it produces the same output signals.

Process constructor Every ciruit in ForSyDe (even simplest combinational ones) is built using a *process constructor*. A process constructor can be seen as a skeleton of

[2]The Glorious Glasgow Haskell Compilation System: `http://www.haskell.org/ghc`

behaviour, and it clearly separates computation from synchronization aspects. A process constructor takes a combinational function (called *process function*) as parameter – expressing the computation aspect of the process, and possibly some extra values. There are combinational and sequential process constructors, and some representative examples from each class will be described in more detail in a specific subsection (4.2.2).

4.2.1 Models of Computation

The definition of signal given above is purposefully "vague" mainly because the precise definition of signals depends on the *model of computation* being used. ForSyDe has (currently) process constructors for the following models of computation:

Synchronous All processes in this MoC have a global, implicit `clock` input, and the tags in the signals are increasing natural numbers. Therefore, a signal can be viewed as a *stream* of values, one for each clock cycle. At each clock cycle, *all* processes consume exactly one value from each of its inputs and produce one value at each of its outputs.

Untimed In the untimed MoC, the processes fire individually and there is no notion of global clock. A process only evaluates when its inputs have a minimum number of values *ready* to be read. The number of needed values can vary per input, but is constant throughout execution.

Continuous The Continuous MoC interprets signals as continuous, one-variable piecewise functions of time. It can be used to model some forms of analog circuits, for example.

Among all MoCs, perhaps the most "notable" one is the Synchronous MoC, because it reflects the usual interpretation of signals as wires and the vast majority of digital designs nowadays having a global clock. Also, all of our studied circuits were modelled in ForSyDe using the Synchronous MoC. Therefore, it is interesting to take a deeper look at it.

First, lets take a look at the behaviour of a system which has an one integer input port and one integer output, and in which the value of the output is equal to the input plus 4. The interface and internal architecture of this system (*addFour*) is depicted in listing 6.

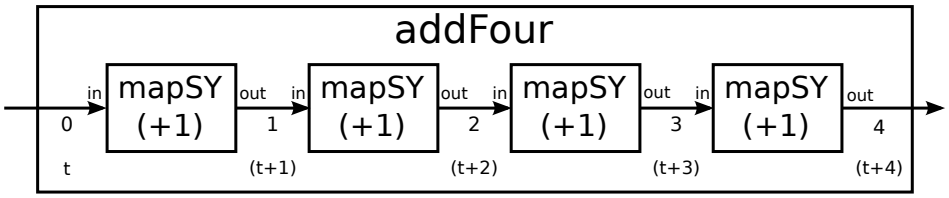

Figure 6: The addFour circuit, example of usage of the Synchronous MoC

In this system, each of the constituent processes is built using the `mapSY` process constructor, a constructor of the synchronous model of computation (its name ends in "SY"). It takes a combinational function (in this case "+1") and evaluates it for each event in the input signal, generating a corresponding event in the output signal.

Another characteristic of ForSyDe which makes the synchronous MoC even more significant is that only systems that are modeled **exclusively** with process constructors of the synchronous model can be translated into VHDL by ForSyDe.

ForSyDe is a *deeply embedded* EDSL, but it takes a significantly different approach than the one taken by Lava: instead of having some set of "atomic" circuits (they correspond to the constructors of the `S` type in Lava), ForSyDe uses Template Haskell to *reify*

Haskell source code into a syntax tree, and use this syntax tree in order to simulate and/or translate the circuit model into VHDL.

On section 4.2.2 We take a closer look at synchronous process constructors, as well as the mechanism by which ForSyDe translates (small fragments of) Haskell source code into the "building blocks" of synchronous systems in VHDL.

On section 4.2.3 we proceed to expose the circuits we modeled in ForSyDe, using the models to make a comparative analysis of ForSyDe with the other EDSLs.

4.2.2 Synchronous Process Constructors

In the ForSyDe Haskell package, the module `ForSyDe.Process.SynchProc` provides the list of process constructors that a can be used to build systems in ForSyDe's Synchronous Model of Computation (4.2.1). The functions provided in that module can be divided in 2 big groups:

Combinational Combinations process constructors build processes in which the value on a certain output at time t depends *only* on the input values at time t. A simple example of a combinational process constructor is:

```
Y :: (ProcType a, ProcType b) => ProcId
  -> ProcFun (a -> b) -> Signal a -> Signal b
```

Sequential On the other hand, sequential process constructors build processes which can *maintain state*, i.e, the value on a certain output at a moment in time can depend on the value of previous inputs and outputs of the circuit. A simple example of a sequential process constructor is:

```
ceSY :: (ProcType a, ProcType b) => ProcId
     -> ProcFun (a -> a) -> a -> Signal a
```

The `mapSY` constructor can be seen as the equivalent of the usual `map` function, but in the context of `Signals`: For each element of the input signal (at each clock cycle), it applies the function to it and then produces as its output the result of the function application.

Instead of being passed a "normal" Haskell function (with type $a \rightarrow b$), `mapSY` is passed a `ProcFun` (Process Function). ForSyDe has instances of `ProcFun` which allow for it to be processed with different *interpretations*, such as simulation, generation of VHDL or generation of graph diagrams. Let's take a look at how one could use the `mapSY` constructor to model a simple incrementer process:

```
incrFunc :: ProcFun (Int16 -> Int16)
incrFunc = \$(newProcFun [d| f :: Int16 -> Int16
                             f x = x + 1 |])

incrementer :: Signal Int16 -> Signal Int16
incrementer = mapSY "incrementerProc" incrFunc
```

As already mentioned, ForSyDe makes heavy use of Template Haskell, and this example already clarifies how. First of all, the "innermost" expression (`f x = x + 1`) is *reified* by the "`[d|`" quasi-quoter into a list of declarations. This list of declarations is then transformed by `newProcFun` into an object of type `ExpQ` (Template Haskell's reified *expression*). Finally, this reified expression is then *spliced into place* and results in an object of type `ProcFun`.

A `ProcFun` represents, intuitively, the syntax tree of the function, and by traversing this `ProcFun` ForSyDe can perform simulation and VHDL generation. There is, however, one big restriction on `ProcFuns`: As already seen in the type signature of `mapSY` above, the input and output types for the `ProcFun` have to be members of the `ProcType` class. Instances of `ProcType` are provided only for:

Primitive types `Int`, `Int8`, `Int16`, `Int32`, `Bool`, `ForSyDe.Bit`.

Enumerated types User-defined enumerations, with derived instances for `Data` and `Lift`.

Containers Tuples and fixed-length vectors (`Data.Param.FSVec`), holding a type of the above two categories and unrestrictedly nested.

For VHDL to be generated from the system definition, ForSyDe imposes a series of extra restrictions on the form that all `ProcFuns` can take. Upon calling the `writeVHDL` function, the `ProcFun` objects are traversed, and a *runtime* error occurs if any of them does not comply with the restrictions. These restrictions are:

"Point-full" notation Declarations with points-free notation are not accepted as synthesizable

Single-clause To be synthesizable, a `ProcFun` cannot have multiple clauses, and it cannot have `let` or `where` blocks. This essentially forbids recursion inside `ProcFuns`. Pattern matching is still possible by using the `case` construct.

Further details on these restrictions and how they constrain circuit design are shown further ahead, when analyzing the studied circuits.

Now that the concept of a *process function* is clear, let's take a look at how to use a sequential process constructor. In this example, we are using the `sourceSY` constructor to build a counter that counts in ascending order starting from 0:

```
ter :: Signal Int16
ter = sourceSY "counterProc" incrFunc 0
```

Notice that we *reuse* the `incrFunc` process function, as it does exactly what we need. The `sourceSY` constructor takes as parameters a process function f and an initial value x, and has as output signal the sequence $[x, f(x), f(f(x)), f(f(f(x))), \ldots]$. This behaviour is equivalent to what the function `iterate` from the Haskell Prelude does.

4.2.3 Circuits modeled

Our comparative analysis of ForSyDe's strengths and weaknesses was done, as usual, by modelling the 3 circuits used as case-studies. ForSyDe has a peculiar "dual" nature, as it supports both shallow *and* deep embedded models, and models written with netlist generation in mind can look *very* different than models which do not comply with the restrictions that allow synthesis.

Because of this dual nature of ForSyDe, when modelling the case-study circuits we considered 2 kinds of models:

High-level A model that uses Haskell constructs inside the process functions (`ProcFun`) as close as possible to what a functional programmer would normally use. These models do not comply with ForSyDe's constraints on the syntax tree of process functions for synthetization, and therefore **can not be translated to VHDL**.

Synthesizable These models are more fine-grained, and use **exclusively** constructs that allow them to be synthesized by ForSyDe's VHDL backend. They "look" much less like functional programs and more like traditional pen-and-paper diagrams of circuits.

Let's start our analysis by looking at the *high-level* model for circuit 1, the ALU, on listing 8:

The first thing to notice is that the system is working over *16-bit integers*, as by the definition of `WordType`. This is not exclusive of the high-level model, however, as ForSyDe can also produce VHDL models working with integers.

```
type WordType = Int16

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

$(deriveLift1 ''ALUOp)

type ALUControl = (Bit, Bit, Bit, Bit, ALUOp, Bit)
type ALUFlags = (Bit, Bit)

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

aluFunc :: ProcFun (ALUControl -> WordType -> WordType -> (WordType, ALUFlags))
aluFunc =
  $(newProcFun
    [d|
        aluFunc' :: ALUControl -> WordType -> WordType -> (WordType, ALUFlags)
        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 :: Signal ALUControl -> Signal WordType -> Signal WordType
        -> Signal (WordType, ALUFlags)
aluProc = zipWith3SY "aluProc" aluFunc
```

Listing 8: High-level ForSyDe model for the ALU.

We defined an enumeration type (`ALUOp`) that encodes possible ALU operations, and derived instances of the `Data` and `Lift` classes for it, as is required for `ALUOp` to be `ProcType`. In the body of the `aluFunc` process function, we perform pattern matching on the f value to discover which operation to perform. Also, the body of `aluFunc` has a `where` block where all the "parts" that constitute the logic of the ALU are defined. Contrast this definition with the *synthesizable* model of the ALU at listing 9:

This model suffers from two consequences of the restrictions imposed by ForSyDe to enable synthesis:

- It is too **fine grained**. As synthesizable `ProcFuns` cannot have local definitions, *every single* step in the datapath inside the ALU has to be a process of its own.
- The parallel and serial combination of processes require several steps of "zipping/unzipping" which have nothing to do with the actual computation. They only adapt the interfaces of the processes to "fit" together, and transform between **tuples of signals and signals of tuples**. We will see the same problem when handling vectors of signals.

An extra weakness of ForSyDe that becomes more problematic in fine-grained models is the need for manual *name management*. Each process in ForSyDe must have a user-provided unique identifier: on the one hand it results in readable and modular VHDL

```
type WordType = Int16
type ALUOp = Bit
type ALUControl = (Bit, Bit, Bit, Bit, ALUOp, Bit)
type ALUFlags = (Bit, Bit)

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

nProc :: ProcId -> Signal Bit -> Signal WordType -> Signal WordType
nProc name = zipWithSY name $(newProcFun [d| f :: Bit -> WordType -> WordType
                                             f n w = if n == H then 42 else w |])

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

tzProc :: Signal WordType -> Signal Bit
tnProc :: Signal WordType -> Signal Bit

aluProc :: Signal ALUControl -> Signal WordType -> Signal WordType
        -> Signal (WordType, ALUFlags)
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)
```

Listing 9: Synthesizable ForSyDe model of the ALU.

netlists, but on the other hand it forces the hardware designer to work at a lower level than desired, making the design more error-prone.

The issue of name management, along with the usage of fixed-length vectors, will become clearer as we analyze our second circuit: a RAM block of 64 lines.

First of all, we model a n-bit register, which is not so different from the n-bit register we modeled in Lava:

```
type WordType = Int16

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

The next needed part for the RAM is a 64-to-1 multiplexer, to choose which of the RAM lines to select as output, depending on the address. We modeled a whole hierarchy of multiplexers upto the one we needed (64-to-1): starting with a 2-to-1 multiplexer, than building a 4-to-1 using 2-to-1 as components, 16-to-1 using 4-to-1 and finally 64-to-1 using 16-to-1 and 4-to-1. We only show the first two degress of the hierarchy in listing 10.

The hierarchy of multiplexers is a perfect example to illustrate the aforementioned issue of zipping/unzipping: as an input to `mux4` we get a signal of binary vectors (each with length 2). But we want to use `mux2` as a component, therefore we need to **unzip** the signal

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

mux2SysDef :: SysDef ( Signal Bit -> Signal WordType -> Signal WordType
                     -> Signal WordType)
mux2SysDef = newSysDef mux2 "mux2Sys" ["sel", "in1", "in2"] ["out"]

mux4 :: Signal (FSVec D2 Bit) -> Signal (FSVec D4 WordType) -> Signal WordType
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 ( Signal (FSVec D2 Bit) -> Signal (FSVec D4 WordType)
                     -> Signal WordType)
mux4SysDef = newSysDef mux4 "mux4Sys" ["sel", "inputs"] ["out"]
```

Listing 10: Excerpt from the hierarchy of multiplexers modeled in ForSyDe.

of vectors into a vector of signals, and then **index** the vector to get each individual signal. In the case of `mux16` (not shown here), this situation becomes even worse as, besides unzipping, we need to **re-zip** the constituent signals into "groups" of the right size to be used with the subcomponents.

As we already mentioned, the requirement of user-given unique names for processes in ForSyDe results in a much more readable VHDL output for the models. Another factor that helps in this direction is the ForSyDe concept of *component instantiation*.

When finished modelling a circuit in ForSyDe, we "wrap it up" in a *system definition* (a value of type `SysDef`) by calling the function `newSysDef`. When we call `writeVHDL` on this system definition, a VHDL top-level `entity` is generated. If we then want to use this "finished" model as a subcomponent in another circuit, we can use the *instantiate* function to create a named process out of the component's `SysDef`. When the VHDL code for the bigger circuit is generated, the ForSyDe instantiation is mapped to a VHDL `component` declaration (with accompanying `port map` statements), which makes for pretty modular VHDL code.

Let's now finally look at the top-level ForSyDe model for circuit 2, the RAM block. The code is presented at listing 11.

This model is also similar to the one wrote in Lava, and that is the reason for why we don't have separate *high-level* and *synthesizable* models for circuit 2. The "natural" model, i.e, the one that came to mind immediately reading the description of the circuit, happens to also be synthesizable. In this model, we use the parts already defined before (register, 64-to-1 multiplexer), as well as some simple gates (`and`, `or`) and an *address decoder* (`decoder'`). We omit here the code for the address decoder, as it consists simply of an enumeration of all *minterms* (all possible boolean products involving the 6 input bits and their negation).

Lastly, let's analyze and discuss the ForSyDe model for circuit 3, the Hack CPU. We built the CPU using mostly already defined circuits (ALU, registers, multiplexers) as building blocks, which made the model also look very similar to the one written in Lava. This modular approach is not coincidental: in the book "The Elements of Computing Systems"[7], great care is taken to make each circuit in the hierarchy add only a small

```
ram64 :: Signal WordType -> Signal (FSVec D6 Bit) -> Signal Bit
      -> Signal WordType
ram64 input addr load = mux' addr (zipxSY "zipRows" rs)
    where
        -- parts declarations
        mux'     = instantiate "mux" mux64SysDef
        decoder' = instantiate "decoder" decode6To64SysDef
        reg' l   = instantiate l regSysDef
        and' l   = instantiate l andSysDef
        -- using the parts
        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 :: SysDef ( Signal WordType -> Signal (FSVec D6 Bit) -> Signal Bit
                     -> Signal WordType)
ram64SysDef = newSysDef ram64 "ram64" ["input","addr","load"] ["outWord"]
```

Listing 11: Top-level ForSyDe model of circuit 2, the RAM block.

step in complexity when compared to its already defined subcomponents.

In the case of the CPU, we needed to model three main additional components: a program counter, an instruction decoder, and a component that decides when to perform a jump. Let's first start by looking at the program counter:

```
type AddrType = Int16

pc :: Signal Bit -> Signal Bit -> Signal AddrType -> Signal AddrType
pc = scanld3SY "programCounter" nextStateFun 0
  where
    nextStateFun =
        $(newProcFun [d| f :: AddrType -> Bit -> Bit -> AddrType -> AddrType
                         f cur reset set new = if reset == H then 0
                                               else if set == H then new
                                               else cur + 1 |])
```

The address type `AddrType` is defined as `Int16` because the specification of circuit 3 requires so. The program counter is a simple counter with `reset` and `set` inputs. Presenting a `high` value at the `reset` input will cause the program counter to output 0 at the next clock cycle, which will make the CPU fetch the instruction from memory address 0, effectively rebooting the computer. Presenting a `high` value at the `set` input will cause the program counter to have as its next output the value currently present at input `addr`. This is the way in which *jumps* are performed in the Hack architecture.

Having defined the model for the program counter, we proceeded to test its behaviour, according to the table of test cases present in chapter 5[9] of the book defining the circuits:

```
testPC3 :: Bool
testPC3 = (simulate pcSysDef) resets sets vals == expected
    where
        (r, s)   = (H, H)  -- nicknames for reset and set
        x        = 0       -- nickname for "don't care"
        expected = [0, 1, 2, 3, 1, 2, 3, 0, 1, 2, 3, 4]
        (resets, sets, vals) = unzip3 inputs
```

```
        inputs = [ (L,L,x), (L,L,x), (L,L,x), (L,s,1), (L,L,x), (L,L,x)
                 , (r,L,x), (L,L,x), (L,L,x), (L,L,x), (L,L,x), (L,L,x) ]
```

An important aspect of simulation with ForSyDe is that we can actually *compare* the outputs of simulation for equality with an *expected* sequence of inputs, which could *not* be done in Lava.

With the program counter defined and tested, we proceeded to model the instruction decoder, whose code is presented on listing 12.

```
type HackInstruction = FSVec D16 Bit
type DestType = (Bit, Bit, Bit)
type JumpType = (Bit, Bit, Bit)

instructionDecoder :: Signal HackInstruction
                   -> Signal (Bit, Bit, DestType, JumpType, ALUControl)
instructionDecoder = mapSY "mapSYdecoder" decoderFun
  where
    decoderFun =
        $(newProcFun [d| f :: HackInstruction
                           -> (Bit, Bit, DestType, JumpType, ALUControl)
                         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)
                               ) |])
```

Listing 12: ForSyDe model for the Hack CPU instruction decoder.

The job of the instruction decoder is very simple: it takes an instruction as input and outputs several signals to *control* different parts of the CPU. It performs *no computation* and merely *rearranges* the wires. But even though it is such a simple circuit, the ForSyDe model is still "ugly" (full of indexing operators and tuple constructors). Because of ForSyDe's single-clause restriction on synthesizable `ProcFuns`, we cannot introduce a `where` block and give meaningful names to the several "slices" of the instruction that we are selecting.

Now we go over the last needed subcomponent of the CPU we needed to model: a logical block which decides when to set the program counter and cause a jump to occur. The output of this circuit is connected to the `set` input of the program counter, as can be seen in figure 5. The code for the ForSyDe version of the `decideJump` block is shown below:

```
decideJump :: Signal JumpType -> Signal ALUFlags -> Signal Bit
decideJump = zipWithSY "zipWithDecide" decideFun
  where
    decideFun =
        $(newProcFun [d| f :: (Bit, Bit, Bit) -> ALUFlags -> Bit
                         f (jl,je,jg) (stZ,stN) = if stN == H then jl
                                                  else if stZ == H then je
                                                  else jg |])
```

The decision on whether or not to perform a jump is taken based on two parameters: the first is a set of `jump selection bits` (named in the model as `JumpType`, and comes from the instruction. These bits indicate in *which conditions* a jump is to be performed. If they are all low, then no jump is performed, and if they are all high, an *unconditional*

jump will happen. The second input for the `decideJump` circuit is the set of flags coming from the ALU. When some conditional jump is described in the `JumpType` bits, it will only actually happen if the correspondent ALU flags are active. The `decideJump` model was also tested for the input combinations described in the book[7], but we omit the test code here for brevity.

Having all the necessary parts we could then model the Hack CPU itself, whose code is presented on listing 13.

```
hackCPU :: Signal WordType  -- ^ inM: M value input (M = contents of RAM[A]
        -> HackInstruction  -- ^ instruction of Hack assembly
        -> Signal Bit       -- ^ reset
        -> Signal ( WordType  -- ^ outM: M value output
                  , Bit       -- ^ writeM: whether to write to M
                  , AddrType  -- ^ addressM: address of M in data memory
                  , AddrType  -- ^ pc: address of the next instruction
                  )
hackCPU inM instr reset = zip4SY "zipOuts" aluOut writeM aReg nextInst
    where
        -- parts declaration
        mux2' l     = instantiate (l ++ ":mux") mux2SysDef
        aReg'       = instantiate "aReg" regSysDef
        dReg'       = instantiate "dReg" regSysDef
        alu'        = instantiate "alu" aluSysDef
        decideJump' = instantiate "decideJump" decideJumpSysDef
        pc'         = instantiate "pc" pcSysDef
        orSetA'     = instantiate "setA:or" orSysDef
        invSetA'    = instantiate "setA:inv" invSysDef
        and' l      = instantiate (l ++ ":and") andSysDef
        decoder'    = instantiate "decoder" decoderSysDef
        -- using the parts
        aReg = aReg' aMux setA
        dReg = dReg' aluOut setD
        aMux = (mux2' "aMux") aFlag instr aluOut
        am   = (mux2' "am") cAM inM aReg
        nextInst = pc' reset setPC aReg

        (aFlag, cAM, cDest, cJump, cALU) = unzip5SY "unzipDec" (decoder' instr)
        (writeA, writeD, writeM) = unzip3SY "unzipDest" cDest

        (aluOut, aluFlags) = unzipSY "unzipALU" (alu' cALU dReg am)
        setPC = decideJump' cJump aluFlags
        setD = (and' "setD") aFlag writeD
        setA = orSetA' (invSetA' aFlag) ((and' "setA") aFlag writeA)
```

Listing 13: Top-level ForSyDe model for circuit 3, the Hack CPU.

This model also looks similar to the CPU model written in Lava, but the generated VHDL is very different, and that is a big advantage of ForSyDe. While Lava *flattens* all the definitions and generates **one big** VHDL entity for the whole model, ForSyDe is able to use the *component instantiations* to produce a *hierarchical* VHDL design, where program counter, register, ALU, decoder, etc., all have their *own entity declarations in separate files*.

As a closing remark on ForSyDe we can emphasize a general weakness of the library, which is not seen *particularly* in any circuit model, but contributes to some of the problems discussed: ForSyDe is relatively old and not actively maintained. The last version available

on Hackage[1] dates from 2010, and on the library's Hackage page there are still promises of a "next version".

This is a problem specially because ForSyDe uses some technologies which are heavily dependent on GHC, and some aspects of the library could benefit from recent GHC developments. The `parameterized-data` package (containing the module `Data.Param.FSVec` of fixed-length vectors) could benefit from additions to the GHC type system (in particular the `TypeNats`[2] extension) which facilitate the kind of dependent types emulated in that package.

4.3 Coquet

The third analyzed EDSL for hardware description, Coquet[4], is strikingly different from both others. Most of these differences can be explained in one way or another by its choice of host "language" – Coq[3].

Coq is an interactive theorem prover based on *intuitionistic type theory*. In the context of Coq, the concepts of "term" and "type" are far more intertwined than, say, in Haskell. Types in Coq can contain references to terms and vice-versa. A very typical example of these so-called *dependent types* is the type(-family) of vectors with a certain length:

```
Inductive vec A : nat -> Type :=
    | nil  : vec A 0
    | cons : forall (h : A) (n : nat),  vec A n -> vec A (S n).
```

By having the length of the vector being part of the type, we can *enforce* several useful properties of functions operating on vectors. In fact, the type-system of Coq is so expressive that it can encode any proposition of *intuitionistic propositional logic*.

Given such expressive power, one can imagine that it might be useful to express circuits in Coq, and use it to prove interesting properties about these circuits. This is exactly the goal of Coquet. How this goal is achieved and the modelling of our studied circuits in Coquet is discussed in the following subsections.

4.3.1 Modelling circuits

Coquet is a deep-embedded DSL, thus it represents circuits as a datatype. By using dependent types, a designer modelling a circuit in Coquet is able to prevent certain kinds of "errors" much earlier in the design process, because the *well-formedness* is guaranteed by construction, i.e, every circuit built using the constructors provided by Coquet are well-formed by definition. Let's take a look at the *Circuit* data type declaration:

The `Circuit` type is parameterized by two types. These types are the *input* and *output* types of the circuit, respectively. They **do not** represent what is "carried" on the wires, but the *structure* of the circuit's input and output ports: How many of them there are, how are they grouped and how are they named.

There are 2 *atomic* constructors from which an element of `Circuit` can be built and 3 *combinators*, which build a circuit based on other circuit(s). By observing the serial and parallel composition combinators (`Ser` and `Par`, respectively), we notice that the input and output types match exactly as expected.

The 2 atomic constructors constrain the types `n` and `m` by requiring them to have instances of the "Fin" type class, i.e, they have to be *finite* types (types from which a finite list of unique elements can be obtained). This constraint is important given the interpretation that these types (`n` and `m`) have: each element of `n` (respectively `m`) stands for an input (respectively output) "wire" in the circuit interface.

[3] "Coq" is not the name of a language, but a theorem-proving system that uses different languages for defining terms, interactive commands, and user-defined tactics

```
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
```

Listing 14: The Circuit datatype in Coquet

The case of the "Atom" constructor is particularly revealing of how Coquet works: this constructor is parameterized by an *instance* of the type class `Techno` for the types `n` and `m`. What this instance provides (in the code fragment that reads "`techno n m`") is the *type of the fundamental gate* in the technology being used. We could choose our modeled circuits to have, for example, NAND, NOR, or other (more exotic) gates as fundamental.

As an "usage example" of Coquet, we show two simple circuits (`NOT` and `HALFADD`), along with proofs that they implement the expected functions over booleans. Let's start with `NOT`:

```
Definition NOT x nx : Circuit [:x] [:nx]  :=  Fork2 _ |> (NOR x x nx).

Instance NOT_Implement {x nx} : Implement (NOT x nx) _ _ negb.
Proof.
    intros ins outs H.
    unfold NOT in H.
    tac.
Qed.
```

The input type of `NOT` is a *tagged unit* with tag `x`, similarly, the output type has tag `nx`. There is some *notation* introduced by Coquet to make the creation of tagged units more convenient. The `NOT` circuit is a serial composition (denoted as `|>` of a `Fork2` circuit (which simply splits the input into two identical copies) and a `NOR` circuit, which is the underlying fundamental gate in this case.

Below the definition of the circuit itself we state and prove the fact that our circuit implements the desired function (boolean negation, `negb`). More details on exactly what is meant by "implements", as well as the workings of plugs and tags, are exposed further ahead. A walkthrough of proofs in Coquet, where we explain the "`tac`" tactic, is given in section 4.3.3. Now let's take a look at a half-adder described in Coquet:

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

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.
```

First of, the sum types that are given as parameters to `Circuit` indicate that we have two input ports and two output ports. By using `Fork2` on a binary sum (`[:a] + [:b]`), we create as output a sum type in which each of the components is in itself a sum: that matches exactly the interface of the component after the `|>` operator. On the right side of the serial composition, we have a parallel composition of `XOR` and `AND`, giving two outputs: respectively the sum (`[:s]`) and carry-out (`[:c]`)

Together with the definition, we prove that the `HADD` circuit implements the boolean function we would expect, and the proof is similar to the case of `NOT`. It makes use of a Coquet custom tactic (`tac`), but a more throughout example of proof of functional correctness will be given futher ahead. Also, in the case of the adder used in our case study (ripple-carry adder), we prove that the circuit implements the *actual* addition function on binary integers, and not some boolean equivalent.

As a last detail on the "user interface" of Coquet, there is the definition of what exactly are the input and output types (a circuit of type `Circuit n m` has input type `n` and output type `m`). Usually, in the Coquet paper[4] and in the examples provided with the library, input and output types are *sum types* in which the terms of the sum are *tagged units*. Using tags serves a form of "documentation", giving someone reading the circuit model an idea of what role does each input/output port play. The *type family* of tagged unit types is defined as follows:

```
Inductive tag (t : string) : Type := _tag : tag t
Notation "[: x ]" := (tag x).
Notation "[! x ]" := (_tag x).
Notation "[!!]"   := (_tag _).
```

For each string `t`, there is a type `tag t`, and this type has exactly one inhabitant. There are also, as part of Coquet, some definitions to make working with sum types less tiresome. For example, there is the function `sumn` which, given a type `t` and a natural `n`, returns a sum type with `n` elements and in which each element has type `t`. This might be useful if we are defining a n-bit adder:

```
Definition RIPPLE cin a b cout s n :
    Circuit  ([:cin] + sumn [:a] n + sumn [:b] n)  (sumn [:s] n + [:cout]) := ...
```

While the provided examples use sums of tagged units as the input/output types, they can be more general: as seen in the definition of the circuit type (Fig. 14), the only requirement is that they belong to the `Fin` type class, which is defined as follows:

```
Class Fin A := {
    eq_fin : eqT A;
    enum   : list A;
    axiom  : forall (x : A), count (equal x) enum = 1
}.
```

4.3.2 Circuit semantics in Coquet

In Coquet, the *structure* and *semantics* of a circuit are strictly separated. The structure of a circuit is modeled by a value of type `Circuit`, and it describes solely which are the parts that the circuit is made of and how they are interconnected[4].

[4]The `Circuit` datatype is also parameterized by the type of fundamental gate used in the design.

On the other hand, circuit *semantics* (what operation does the circuit *perform*) is described in Coquet by a *meaning relation*. The meaning relation for a circuit relates its inputs to outputs, and is defined by induction on circuit structure.

For a circuit type `Circuit n m` and considering $\mathbb{T}$ as the type of what is carried in the wires, we can define the type *ins* (stands for "inputs") as $n \to \mathbb{T}$ and *outs* as $m \to \mathbb{T}$. These are functions that, for each input/output port, provide the a value present at that port – they are in this way *isomorphic* to cartesian products, and this isomorphism is indeed used to facilitate proofs of correctness in Coquet, as will be seen later. The definition of the meaning relation in Coquet is presented on figure 15.

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

| KAtom : forall n m {Hfn : Fin n} {Hfm : Fin m}
             (t : techno n m) ins outs,
                 spec t ins outs -> Semantics (Atom t) ins outs

| KSer : forall n m p (x : circuit n m) (y : circuit m p) ins middles outs,
                   Semantics x ins middles
                 -> Semantics y middles outs
                 -> Semantics (Ser x y) ins outs

| KPar : forall n m p q (x : circuit n p) (y : circuit m q) ins outs,
               Semantics x (select_left ins) (select_left outs)
             -> Semantics y (select_right ins) (select_right outs)
             -> Semantics (Par x y) ins outs

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

| KLoop : forall n m l (x : circuit (n + l) (m + l)) ins outs retro,
                Semantics x (Data.app ins retro) (Data.app outs retro)
             -> Semantics (Loop x) ins outs
```

Listing 15: Coquet definition of circuit semantics.

We can notice that the definition of `Semantics` has constructors that correspond to the constructors of `Circuit`. So, for example, given the semantics of two circuits `x` and `y`, we can obtain the semantics of their *serial composition* (`Ser x y`) by using the `KSer` constructor. This inductive definition of semantics can also be used in proofs. If we need to prove a statement of the form:

```
Semantics (Ser x y) ins outs
```

Then we can `apply` the `KSer` constructor to split the goal into the following subgoals:

```
Semantics x ins middles
Semantics y middles outs
```

While the meaning relation defines exactly the behaviour of a circuit, it is too low-level. We want to be able to express our *specification* for circuit behaviour in a *higher level* of abstraction – after all, the whole point of proving correctness is making sure that the circuit we are modelling is *equivalent* (in a sense) to a specification that we *assume as correct*.

Coquet offers some tools to help the user in this direction. First of all, it offers the designer two kinds of abstraction to facilitate writing high-level specifications:

Data abstraction The meaning relation (`Semantics`) for a circuit is a relation between two *functions* (*ins* and *outs*), which is cumbersome to reason about. Therefore, Coquet allows the user to express the specification for a circuit in terms of higher-level types, provided that isomorphisms between these higher-level types and the function types are provided. Several isomorphisms for common cases of input/output types are already provided in the Coquet library, such as the isomorphism between (sumn $\mathbf{1}\ n \to \mathbb{B}$) and ($\mathbb{B}^n$), where $\mathbf{1}$ stands for the unit type and $\mathbb{B}$ for boolean).

Behavioural abstraction A circuit can be said to satisfy a weak specification `R` if we can prove the logical entailment of `R` by the meaning relation. The specification `R` already benefits from *data abstraction*, and ranges over the high-level types.

There are two ways to express compliance with a specification: we can say either that a circuit `Realises` a certain relation (up to isomorphisms) or that it `Implements` a certain function (up to isomorphisms). The difference between the *relational* and the *functional* models is that the functional model can only account for *deterministic* specifications (an input combination maps to only one output), while with the relational model we can also write non-deterministic specifications. The definitions for the Coquet classes `Realise` and `Implement` are shown in figure 16.

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

Listing 16: Definition of the `Realise` and `Implement` type classes.

Is we want to prove that a certain circuit `c` *implements* a certain function f (the high-level specification), then our goal in Coq will be:

```
Implement c f
```

To prove this kind of statement, one can "break down" the meaning relation hypothesis, resulting in one `Semantics` hypothesis per circuit subcomponent. Then the already-proven specifications of the subcomponents could be used to rewrite the `Semantics` hypotheses into equations. This makes for very modular proofs, and this will become clearer with the example proofs in section 4.3.3.

4.3.3 Example circuits and proofs

Until now we explained several aspects of the "inner workings" of Coquet: How circuits are modeled, how the `Circuit` dependent type guarantees well-formed models *by construction*, what Coquet considers as the *semantics* of a circuit, and what does it mean to say in Coquet that a circuit satisfies a specification. What was not covered, however, is how to actually write proofs of correctness in Coquet.

In this section we will walk over some examples of circuit models in Coquet, in increasing order of complexity, and also comment on their proofs of correctness. This review should give the reader an idea of the general structure of circuits and proofs in Coquet. It should also serve as base for our comparative analysis of Coquet with the other EDSLs.

Let's start by analyzing a hierarchy of adders. The most basic of these adders is a half adder:

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

This is pretty straightforward circuit model: we just combine `XOR` and `AND` in parallel, and each of them provides one of the outputs of the circuit (sum and carry-out). The `Fork2` *plug* just "copies" its input into two identical outputs. While there is very little to comment on the circuit model, the proof of correctness will give us a bit more insight into Coquet:

```
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.
```

The proof is considerably short, but all the "work" is being done behind the scenes by the custom tactic `tac`, introduced by Coquet. This tactic is geared towards proving simple circuits concisely, and its definition reads as follows:

```
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.
```

The definition of `tac` is itself just a (sequential) combination of other custom tactics also defined by Coquet. We don't need to go deeper, however, as we can explain the general mechanism of each line in `tac`.

First of all, `rinvert` performs *inversion* using the constructors of `Semantics`, and will transform the meaning relation hypothesis into a series of hypotheses, one for each component of the circuit. Then `realise_all` is called, which uses the *correctness proofs of the components* (stored in a hint database) in order to rewrite *each* `Semantics` *hypothesis into an equality involving high-level types*. Finally, the tactic `unreify_all` uses the isomorphisms to transform the equality in the goal into one involving only booleans. From them on we just `destruct` all booleans, which results in a proof by case analysis.

Going one step up in the hierarchy of adders, we have the Coquet model of a full adder:

```
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.
```

In the definition of a full adder, we use a half adder as component, along with an `OR` gate. The interesting point of this definition, though, is the usage of the `Rewire` components.

Earlier, when presenting the `Circuit` datatype in Coquet, we mentioned that one of the circuit constructors (`Plug`) is meant to be used to "adapt" the interface of two circuits which we need to combine.

In example of `FADD`, all the necessary plugs involved only regropuing of ports (they are all *associativity plugs*) and, in these cases, the plug functions are fully defined by their type, which allows us to avoid writing the functions ourselves and let Coq find the terms using *proof search*. That's why we use Coq's `Program` command to define the circuit: in the definition, there are some *holes* where the plug functions should be (we omit the full `Rewire` lines for brevity), and we need to "fill" each of these holes after defining `FADD` – that is being done in each of the `Next Obligation` blocks.

The correctness proof for `FADD` uses the exact same tactics as the one for `HADD`, we only include it here for completeness:

```
Instance FADD_Implement {a b cin sum cout} :
    Implement (FADD a b cin sum cout) _ _
    (fun x =>
        match x with
        | (c, (a,b)) =>
              (xorb a (xorb b c), (a && b) || c && (xorb a b))%bool
        end).
Proof.
    unfold FADD; intros ins outs H; tac.
Qed.
```

This proof can be used as an example to understand how Coquet provides for highly *modular verification*. The proof `FADD_Implement` also uses the `tac` tactic that we already explained. When the `realise_all` step of `tac` is performed, we already have *the correctness proof of* `HADD` *in the hint database*, and therefore the hypothesis involving the `Semantics` of `HADD` is readily rewritten as an equality involving its high-level specification.

In the last step of the adder hierarchy we are presenting, there is a classic ripple-carry binary adder. With this example, we demonstrate how a *parametric circuit* can be defined using recursion in Coquet:

```
Program Fixpoint RIPPLE cin a b cout s n :
    circuit ([:cin] + sumn [:a] n + sumn [:b] n) (sumn [:s] n + [:cout]) :=
match n with
    | O   => RewireE (*  (c, x, y) => (x, c)  *)
    | S p =>
          RewireE (*  (a, b, c) => (a, (b, c))  *)
        |> (ONE [:cin]  &  high_lows a b 1 p)

        |> RewireE (*  (c, ((a1,b1), (ap,bp))) => (c, a1, b1, (ap,bp))  *)
        |> (FADD a b cin s "mid"  &  ONE (sumn [:a] p + sumn [:b] p))

        |> RewireE (*  (s, c, (a,b)) => (s, (c,a,b))  *)
        |> (ONE (sumn [:s] 1)  &  RIPPLE ("mid")%string a b cout s p)

        |> RewireE (*  (s1, (s2,c)) => (s1, s2, c)  *)
        |> combine' s 1 p  &  ONE [:cout]
end.
Next Obligation.
    revert H; intros [[]| H]; repeat left; constructor.
Defined.
Next Obligation. abstract plug_auto. Defined.
Next Obligation. abstract plug_auto. Defined.
```

```
Next Obligation. revert X; plug_def. Defined.
Next Obligation. abstract plug_auto. Defined.
Next Obligation. revert X; plug_def. Defined.
Next Obligation. abstract plug_auto. Defined.
Next Obligation. abstract plug_auto. Defined.
```

The definition is recursive on the size n of inputs and outputs. The full adder previously defined (`FADD`) is used to calculate the least significant bit of the output, and we use `RIPPLE` recursively to calculate the remaining bits. The component `high_lows` is used to "split" the inputs and then they are combined into the desired output shape by `combine'`. Notable in the structure of `RIPPLE` is the heavy usage of `RewireE` plugs to adapt the several parts of the circuit connected serially. Similarly to when defining `FADD`, we also used proof search to "fill the gaps" left by the associativity plugs.

The proof of correctness for `RIPPLE` is not as straightforward anymore as the ones we have seen until now. In fact, we are only going to show some excerpts of the proof. This difference in proof complexity is due to 2 main reasons:

- It is a true *proof by induction*, whereas in the previous proofs only *case analysis* was performed.
- We prove the compliance of `RIPPLE` to a *high-level specification*: instead of proving that the circuit implements some boolean fucntion (as previously), we prove that `RIPPLE` implements integer addition on n-bit integers.

The function used as specification for `RIPPLE` can be seen in the following excerpt:

```
Lemma Implement_adder n cin a b cout s :
    Implement (RIPPLE cin a b cout s n)
    ([b:_] & Iso_Phi _ n & Iso_Phi _ n)%reif
    (Iso_Phi _ n & [b:_ ])%reif
    (fun x => match x with (c,a,b) => add n a b c end).
```

Besides the higher level of the specification, we can also notice that here we are explicitly providing the isomorphisms between high (specification) and low (implementation) types. To get an idea of how the proof for `RIPPLE` would procees, we show here the base case ($n = 0$):

```
Proof.
    revert cin cout.
    induction n.
    intros cin cout ins outs H. unfold RIPPLE in H.
    realise_all.
    rewrite H. clear. unreify_all bool.
    destruct_all.
    apply eqT_true.
    rewrite (Word.eq_zero w (Word.repr 0 1)).
    rewrite (Word.eq_zero w0 (Word.repr 0 0)).
    destruct b0; reflexivity.
```

There are some similarities with the previous proofs – we also use `realise_all` and `unreify_all` – but tactics related to integer arithmetics are now also needed, because of the way in which the specification is expressed.

Now, to finish our walkthrough of Coquet's circuit models and proofs, we will look at a *sequential* circuit: A 1-bit register. The specification of behaviour for sequential circuits in Coquet is significantly more involved and, in fact, the original Coquet paper[4] leaves a "more thorough investigation of state-holding devices" for future work. But before delving into the specification, let's first look at the circuit model itself:

```
Context a load out : string
Program Definition REGISTER : Circuit ([:load] + [:a]) [:out] :=
    @Loop _ ([:load] + [:a]) [:out] [:out]
    (
           RewireE (*  (load, a, out) => (a, out, load)  *)
        |> MUX2 a out load "in_dff"%string
        |> DFF "in_dff" out
        |> Fork2 [:out]
    )
Next Obligation. revert H. plug_def. Defined.
Next Obligation. plug_auto. Qed.
```

The basic building block for the register is the `DFF` flip-flop. Also, we duplicate (`Fork2`) the output of the flip-flop and use the `Loop` constructor to direct one of these wires back into the `MUX`. The specification for the behaviour of a register is based on *streams*: in the meaning relation, the type $\mathbb{T}$ of what is carried in the wires is not simply $\mathbb{B}$, but $\mathbb{N} \to \mathbb{B}$ (where $\mathbb{B}$ stands for boolean). Also, we don't use the *functional* (`Implement`) model anymore, but the relational one.

```
Instance Register_Spec : Realise
    (Rn : Iso ([:load] + [:a] -> stream bool) (stream (bool * bool)))
    (* ... isomorphism on outputs ... *)
    (fun (ins : stream (bool * bool)) (outs : stream bool) =>
      outs = pre false
               (fun t => if fst (ins t) then snd (ins t) else outs t))
```

The specification dictates that the output stream must have a one-element *prefix* (`pre`) equal to `false` and, from then on, be either equal to the previous value at port `a` (whenever `load` is high) or to the previous value at port `out` (whenever `load` is low). From this example it is already clear that sequential specification does not fit nicely into the `Realise`/`Implement` model, and a more comfortable way to specify and prove the behaviour of sequential circuits would be a welcome addition to Coquet.

4.3.4 Closing remarks and possible improvements

TODO: Closing remarks. Simulation with drawback of not simulating sequential. Great verification, actually surpassing the very definition of verification, great genericity and great extensibility. Integration with other tools is hard.

Coquet employs very well several features of dependently-typed programming in general, and of the Coq system in particular, in order to facilitate circuit modelling and verification. The following advantages of Coquet make it particularly distinct from the other studied EDSLs:

- The use if **dependent types** makes certain classes of design mistakes (such as short-circuits or "floating" wires) *impossible by design*.
- By using **type classes** as a way to structure its definitions, Coquet facilitates the automatization of proofs. For example, when trying to prove the correctness of a circuit, we can use the correctness proofs of all its subcomponents, and they are automatically located by Coq's *instance resolution* mechanism.
- Coquet avoids the problem of **observable sharing** by not using bound variables, and building circuits only with combinators. The usage of combinators is facilitated by some particular Coq features, like *notations* and *proof search*.

- Coquet is able to prove properties over **parametric circuits** for *all values of the parameters* (by induction), while Lava, for example, can only verify those properties for specific instances.
- The **parametrization** of the `Circuit` type by the *type of the fundamental gate* and the parametrization of the `Semantics` relation by the *semantics of the fundamental gate* make Coquet's approach extremely generic. All of Coquet's defined tactics, classes and instances could be used in radically different contexts such as three-valued logics, analog domains or probabilistic domains.

Although Coquet is superior to the other studied EDSLs in the aforementioned aspects, some other aspects of circuit modelling and verification with Coquet could benefit from concepts present in ForSyDe or Lava

First of all, *simulation* of combinational circuits in Coquet is easy, but currently it is *impossible* to simulate any form of sequential circuit. More precisely, any circuit containing the `Loop` constructor cannot be simulated – this restriction also bans the simulation of combinational loops, but simulating combinational loops does not make much sense anyways.

Coquet's definition of the *meaning relation* for circuits depends on the *type* `T` *of what is carried in the wires*. The Coquet library already provides instances for booleans and streams of booleans ($\mathbb{N} \to \mathbb{T}$), but it could also be interesting to add cases for dealing with some three-valued logics, or a case for IEEE1164's std_logic type, used often in VHDL.

Another interesting point is that Coquet defines a `stream` type family, and then proceeds to define several interesting functions over `stream`, as well as an instance for the meaning relation. The type family is defined as follows:

```
Definition stream A := nat -> A.
```

If instead of using `nat` in the above definition, we generalized it a bit more (abstracting a new type parameter), we arrive at the concept of an "event stream":

```
Definition events tag A : tag -> A.
```

This is exactly how ForSyDe defines its "signals", so we could model circuits in models of computation other than the synchronous model (which is the one allowed by the current `stream` definition).

5 Conclusions

The models and test cases that we developed, along with the verification we performed, gave us a better understanding of how the hardware EDSLs Lava, ForSyDe and Coquet compare to each other *from the point of view of a hardware designer*. This practical experience, combined with knowledge of the "inner workings" of each EDSL and their host language, allowed for an informed discussion of each language's strong points and weaknesses. The most significant findings of this practical evaluation, categorized by evaluated aspect, are summarized here.

Depth of embedding None of the three evaluated EDSLs lie at the *extremes* of embedding depth. Lava can be said to be *deeply embedded*, however, its `Signal` datatype collaborates with the host language runtime so that *cyclic structures* in circuits can be modeled as *recursion in the host language*. ForSyDe has *both* deep and shallow modelling capabilities, even though we only studied the deep model. In fact, ForSyDe's hackage page[1] promises a future version in which deep and shallow modelling constructs will be in different packages. Coquet has the "deepest" modelling of all studied EDSLs, and *avoids* the issue of *observable sharing* by not allowing variable binding constructs, and having circuits connect to each other only through combinators.

Simulation Simulation can be performed in all studied EDSLs. In Lava, *automated* test cases (in which the simulation output is compared with an *expected* combination) are not possible due to the way in which the observable sharing issue is handled. Coquet has simulation built into the library as one of several *example interpretations* for circuits, and it works just as well as in the other EDSLs, with the only shortcoming that the simulation of *sequential* circuits is currently *not possible*.

Verification ForSyDe offers no capabilities for formal verification whatsoever, while Lava and Coquet each do, but in different ways. Lava can perform the verification of so-called *safety properties* for circuits of a fixed size – it does this by transforming the circuit model into a CNF (*conjunctive normal form*) logical formula which is fed into a satisfiability solver. Coquet takes a different approach and offers some tools to help the user perform *interactive theorem proving* for circuit correctness. One can say that Coquet does *more than verification*, as it's capable of proving *general* properties of parametric circuits, for all possible values of the parameters.

Genericity

Tool integration Lava can generate VHDL netlists of circuit models that satisfy some requirements and can also generate CNF formulas for a SAT solver. ForSyDe can output its circuits in VHDL and also generate *graph files*, which can be formatted and used for circuit visualization. Coquet is disadvantaged when it comes to tool integration: it currently has no support for exporting circuits in some industry-standard format, even though it *could in theory* be done by traversing the models.

TODO: Reiterate the analysis of the EDSLs spread throughout the analysis of each of the studied circuits, but in a more summarized manner, actually **comparing** the EDSLs among themselves, **by aspect** (the ones we defined in section 2.2).

References

[1] Hackage page for the forsyde library. `http://hackage.haskell.org/package/ForSyDe`, November 2013.

[2] Type-level naturals in ghc. `https://ghc.haskell.org/trac/ghc/wiki/TypeNats`, November 2013.

[3] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in haskell. *SIGPLAN Not.*, 34(1):174–184, September 1998.

[4] Thomas Braibant. Coquet: a coq library for verifying hardware. In *Certified Programs and Proofs*, pages 330–345. Springer, 2011.

[5] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, ASIAN '99, pages 62–73, London, UK, UK, 1999. Springer-Verlag.

[6] MATTHEW NAYLOR and COLIN RUNCIMAN. The reduceron reconfigured and re-evaluated. *Journal of Functional Programming*, 22:574–613, 9 2012.

[7] Shimon Shocken Noam Nisan. *The Elements of Computing Systems: building a modern computer from first principles*. MIT Press, 3rd edition, 2012.

[8] Shimon Shocken Noam Nisan. *The Elements of Computing Systems: building a modern computer from first principles*, chapter Machine Language. In [7], 3rd edition, 2012.

[9] Shimon Shocken Noam Nisan. *The Elements of Computing Systems: building a modern computer from first principles*, chapter Computer Architecture. In [7], 3rd edition, 2012.

[10] Ingo Sander and Axel Jantsch. Formal system design based on the synchrony hypothesis, functional models, and skeletons. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 318–323. IEEE, 1999.

[11] S. Singh. Designing reconfigurable systems in lava. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 299–306, 2004.