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 22nd November, 2013

1 Introduction

Hardware design has become nowadays an activity which is, to say the least, extremely complex, if not error-prone. The sheer size of circuits has increased. Also, low-level concerns such as power consumption, error correction, parallelization and layout in general have to be incorporated earlier and earlier in the design process, breaking modularity and making it harder to validate and verify the correctness of circuits.

In this context, researchers have been suggesting for a long time (since the 1980s, in fact) 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 is a multitude of EDSLs for hardware description out there, but they vary wildly and 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 extremely 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 representative 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 that the analyzed EDSLs could benefit from. 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 all of them. To avoid using any of the analyzed EDSLs as "base", 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[2] 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[6], 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[3] 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 strenghts 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:

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

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"[5]. 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 The ALU circuit

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 an integer in two’s-complement encoding). 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

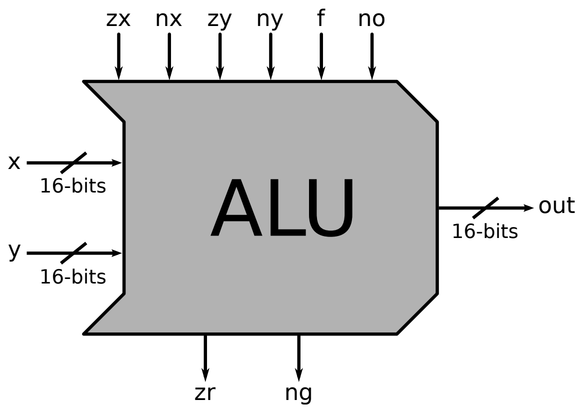

Figure 1: Block diagram of circuit 1, showing its input and output ports.

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* 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 $out < 0$.

With the ALU 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 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 The RAM circuit

Circuit 2 is a block of RAM with 64 lines and 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.

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 1-bit "load" input 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 at 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" pin 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.

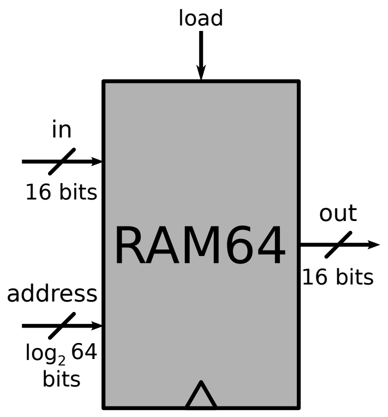

Figure 2: Block diagram of circuit 2, a RAM of 64 lines

3.3 The Hack CPU circuit

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"[5].

The Hack computer is based on the *Harvard architecture*, that means that it has different storage component 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

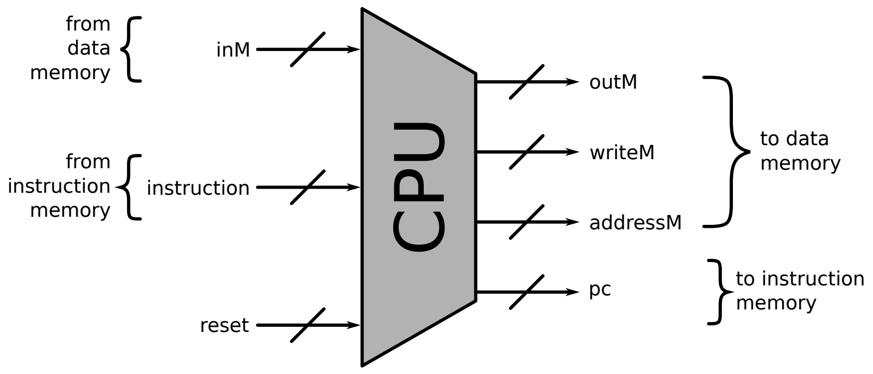

Figure 3: Block diagram of circuit 3, the Hack CPU

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 [5].

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.

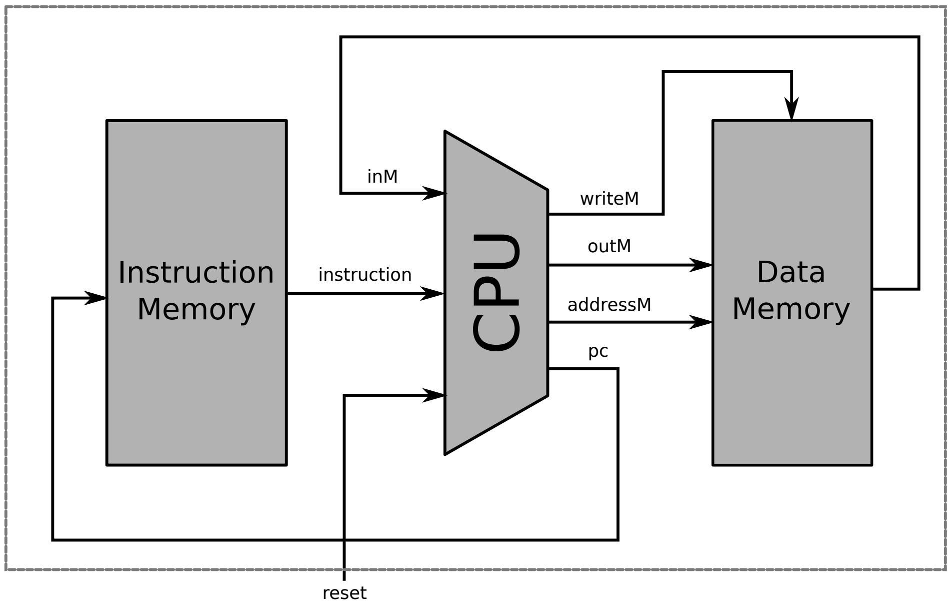

Figure 4: The Hack CPU connected to the data and instruction memory blocks

4 Analysis of the EDSLs

4.1 Lava

Lava[2] 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 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 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, etc.), RAM memory blocks, among others. 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[2] contains the first version, while the current version is the one defined in [4]. 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 figure 5. 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

```haskell
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
```

Figure 5: Lava's `Signal` datatype, used to represent circuits.

provide circuits as outputs. For example, on figure 6 we show some boolean circuit combinators:

```haskell
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
```

Figure 6: Some of Lava's boolean circuit combinators.

With Lava, one can also model circuits operating on `Int`s (and there are, in fact, 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.

4.1.1 Circuits modeled

TODO: expose and discuss the circuits modeled in Lava, in light of the aspects being analyzed.

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 subcircuits. 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 subcomponents, 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:

```
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)
```

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 a 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 a low **extensibility**, not allowing – for example – user-defined types.
- In Lava, (families of) circuits with variable-sized inputs/outputs are modeled by using 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 `C1`, which uses the assumption that its inputs are 32-bit wide. There is no way to enforce, at *Haskell compilation time*, that the inputs with correct size are provided. Possible problems could only be detected at simulation or VHDL generation.

Now, after having defined all the necessary subcomponents, lets take a look at the ALU circuit itself:

```
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
        out' = ifThenElse no (out, map inv out)
        zr   = foldl (curry or2) low out'
        ng   = equalBool high (last out')
        out  = let xy'' = zip x'' y'' in mux (f, (andl xy'', rippleCarryAdder xy''))
        x'   = ifThenElse zx (x, replicate (length x) low)
        x''  = ifThenElse nx (x', map inv x')
        y'   = ifThenElse zy (y, replicate (length x) low)
        y''  = ifThenElse ny (y', map inv y')
```

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 [5] 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:

```
verifyHalfAdder :: [(SB, SB)]
verifyHalfAdder = simulateSeq halfAdder input
    where
        input = [ (low,  low)
                , (low,  high)
                , (high, low)
                , (high, high)
                ]
```

Our `halfAdder` circuit is combinational, therefore we could have used the function `simulate` if we wanted to test its behaviour when given one single combination of inputs. However, in this case we give the circuit under test a *sequence* of inputs and check the corresponding sequence of outputs.

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`):

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

With the setback 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:

```haskell
prop_IncrementIsAlwaysDifferentThanInput :: Int -> Property
prop_IncrementIsAlwaysDifferentThanInput n =
        forAll (list n) (\x -> verifyIncrement x)
    where verifyIncrement 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 sequential "primitive" 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:

```haskell
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:

```haskell
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`; of course we can only simulate and synthesize specific instances of this family of circuits, as the example below:

```haskell
testRegN4 :: [[Signal Bool]]
testRegN4 = simulateSeq (regN 4) ins
    where
        lows  = replicate 4 low
        highs = replicate 4 high
        ins   = [(lows,high), (highs,low), (highs,low), (highs,high), (lows,low)]
```

The `simulateSeq` function, which we already used to test the ALU (a combinational circuit), is actually 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:

```haskell
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 subcomponent of the CPU: the *program counter*:

```
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))
```

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 subcomponents 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:

```
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)
```

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 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 [1] extension [1] (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[6]. The ForSyDe approach per se has several significant

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

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 network of processes interconnected by *signals*.

Signal A signal is, intuitively, a stream of information that flows between processes. More concretely, 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*4.2.1 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 simple combinational ones, is built with a *process constructor*. A process constructor can be seen as a skeleton of 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 is given by 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 those, 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 figure 7.

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.

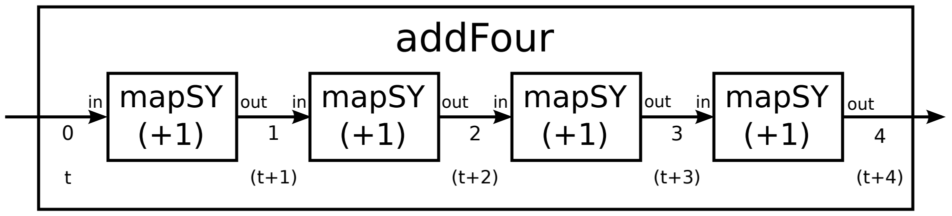

Figure 7: The addFour circuit, example of usage of the Synchronous MoC

Another characteristic of ForSyDe which makes the synchronous MoC even more significatn 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.

This approach has some advantages and disadvantages, which we will look into with more details on section 4.2.3. For now, let’s first take a look at how to build processes in ForSyDe using the *synchronous process constructors*.

4.2.2 Synchronous Process Constructors

TODO: Example of usage of `mapSY`, with a simple `ProcFun`. Mention that limitations to ProcFun will be discussed as the case study circuits are analyzed.

TODO: Example of usage of a state-machine or scanl constructor, mentioning the approach taken by ForSyDe, in which conceptually ALL processes are sequential in nature (have clock input), but some have the property that the current state does NOT depend on previous states.

4.2.3 Circuits modeled

TODO: expose and discuss the circuits we modeled as case study, highlighting the advantages and disadvantages of ForSyDe found while modelling them.

Advantages: more modular VHDL generated (hierarchical). Higher-level descriptions.

Disadvantages: name handling, not actively maintained

For each of the 3 circuits studied, we considered 2 models under ForSyDe:

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:

```haskell
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

aluSysDef :: SysDef (  Signal ALUControl -> Signal WordType -> Signal WordType
                    -> Signal (WordType, ALUFlags) )
aluSysDef = newSysDef aluProc "alu" ["ctrl", "x", "y"] ["outs"]
```

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. TODO: explain the ProcFun type and how the newProcFun TH splice and the “d” quasi-quoters work.

Now let’s compare the high-level model of the ALU with the *synthesizable* one, and discuss along the way how the limitations of ForSyDe’s VHDL generation backend constrained our design:

```
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
tzProc = mapSY "tzProc" $(newProcFun [d| f :: WordType -> Bit
                                         f w = if w == 0 then H else L |])

tnProc :: Signal WordType -> Signal Bit
tnProc = mapSY "tnProc" $(newProcFun [d| f :: WordType -> Bit
                                         f w = if w < 0 then H else L |])

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)

aluSysDef :: SysDef (  Signal ALUControl -> Signal WordType -> Signal WordType
                    -> Signal (WordType, ALUFlags) )
aluSysDef = newSysDef aluProc "alu" ["ctrl", "x", "y"] ["outs"]
```

TODO: Discuss how we can't use user-defined datatypes, and the restrictions on the Haskell syntax that can be used as input for a newProcFun splice (single clauses, no pattern matching, import restrictions, etc).

By analyzing the second studied circuit (the RAM block), we will gain some insight into other significant characteristic of the ForSyDe library – the usage of fixed-length vectors, and the *manual* management of process names:

TODO: discuss how – on the one hand – ForSyDe's FSVec's can help make modelling more type safe but on the other hand the current implementation is very "hackish" and could ALSO make some use of recent developments in GHC.

TODO: discuss how ForSyDe's concepts of processes and (sub)components is responsible for really elegant and modular generated VHDL code but, on the other hand, forces the user to manage naming of the processes manually. Give some possible suggestions for this problem.

4.3 Coquet

The third analyzed EDSL for hardware description, Coquet[3], 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.[2]

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 :=
```

[2]"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

```
  | 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*, a formal logic in which almost all of mathematics can be proven.

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 definition:

```
Context {tech : Techno}
Inductive ℂ : Type -> Type -> Type :=
| Atom : ∀ {n m : Type} {Hfn : Fin n} {Hfm : Fin m}, techno n m -> ℂ n m
| Plug : ∀ {n m : Type} {Hfn : Fin n} {Hfm : Fin m} (f : m -> n), ℂ n m
| Ser : ∀ {n m p : Type}, ℂ n m -> ℂ m p -> ℂ n p
| Par : ∀ {n m p q : Type}, ℂ n p -> ℂ m q -> ℂ (n + m) (p + q)
| Loop : ∀ {n m p : Type}, ℂ (n + p) (n + p) -> ℂ n m
```

Figure 8: The Circuit (ℂ) datatype in Coquet

The ℂ 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 they grouped and how they are named.

There are 2 *atomic* constructors from which an element of ℂ 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.

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 : ℂ [: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. Now let's take a look at a half-adder described in Coquet:

```
Definition HADD a b s c : ℂ ([: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 ℂ 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 ℂ n m has input type n and output type m). Usually, in the Coquet paper[3] 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 :
    ℂ  ([: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. 8), 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

TODO: Explain what the meaning relation for a circuit in Coquet is, give the (inductive) definition of the meaning relation and comment on it.

TODO: Explain how the meaning relation is too "low-level", and how Coquet uses *data abstraction* and *function abstraction* to prove the correctness of circuits. The proofs of correctness are proofs that a circuit either **Realizes a high-level relation** or **Implements a high-level function** over the high-level input/output types.

We give the definitions of Realizes and Implements, as well as walk over a small example of functional correctness proof.

4.3.3 Circuits modeled

TODO: Expose the studied circuits modeled in Coquet, highlighting advantages and disadvantages of the Coquet approach along the way.

TODO: circuit 1 (ALU) – Model, proof of some of its components.

TODO: circuit 2 (RAM) – Model.

4.3.4 Possible improvements and future work

The approach to circuit modelling is *very* generic, and leaves room for extension in several aspects, even though the library is already provided with some reasnoable deafults.

For example, the definition of the *meaning relation* for circuits (by induction on circuit structure) is parameterized by the *type* T *of was is carried in the wires*. Even though the library already provides examples for booleans and streams of booleans, 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 function 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 Results summary

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).

6 Conclusions

References

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

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

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

[4] 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.

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

[6] 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.