# Exercises in architecture specification using C$\lambda$aSH

Jan Kuper, Christiaan Baaij, Matthijs Kooijman, Marco Gerards
*University of Twente*
*Department of Computer Science*
*Enschede, The Netherlands*
*j.kuper@ewi.utwente.nl*

***Abstract*—This paper introduces the hardware specification system C$\lambda$aSH by elaborating on a few non-trivial examples. C$\lambda$aSH is a compiling system that translates a subset of Haskell into synthesizable VHDL by a rewriting technique. This subset of Haskell includes higher order functions, polymorphism, lambda abstraction, pattern matching, and choice constructs.**

## I. Introduction

A combinational digital circuit transforms input signals into output signals. Each time such a circuit gets the same input signals, it produces the same output signals, i.e., it behaves as a mathematical function. Things become a bit more complicated when a circuit contains memory elements, i.e., when the circuit has state, since a (mathematical) function does not have state. Still, intuitively a circuit strongly refers to the concept of function and several attempts have been made to develop hardware description languages based on a functional language, see [1]– [10].

Two of the most well-known of these are *Lava* (see [8]) and *ForSyDe* (see [9]). These languages are domain specific embedded languages, and are both defined in Haskell. In both languages a digital circuit is specified as a function which operates on (possibly infinite) *streams* of values, where at the same time a clock is represented in the stream: at each clock cycle one stream element is processed. Furthermore, both Lava and ForSyDe model state by a *delay* function which intuitively holds each stream element during one clock cycle.

In C$\lambda$aSH we take a different perspective. Instead of defining a domain specific embedded language, C$\lambda$aSH compiles specifications written in (a subset of) plain Haskell itself. Furthermore, these specifications do not work explicitly on streams of signals, but rather express a *structural* description of a circuit. In order to model state, C$\lambda$aSH considers a circuit as a Mealy Machine, i.e., the function representing the behaviour of a circuit with state has *two* sorts of argument: the (current) state and the input signal(s). The result of the function also consists of two things: the (new) state and the output signal(s). Thus, C$\lambda$aSH assumes that the type of a function $arch$ describing a hardware architecture, i.e., the type of a circuit specification, is as follows:

$$arch :: State \rightarrow Input \rightarrow (State, Output)$$

for appropriate types $State$, $Input$, $Output$.

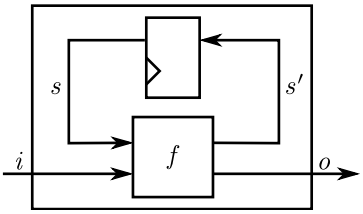


Figure 1. Mealy machine

For C$\lambda$aSH the clock is not explicitly expressed, instead it is assumed that a specification describes the functionality performed during one clock cycle.

Since both Lava and ForSyDe are based on embedded domain specific languages, they define special functions to simulate a given specification expressed in the embedded language. Since C$\lambda$aSH, on the other hand, starts with specifications written in Haskell itself, simulation comes more or less for free and can be done directly by a Haskell interpreter or compiler. For that we only need a function *run*, which is the same for every architecture specification of the type of *arch* above. It is recursively defined as follows:

$$\begin{aligned} &run\ f\ s\ (i : is) = o : run\ f\ s'\ is \\ &\quad \textbf{where} \\ &\qquad (s', o) = f\ s\ i \end{aligned}$$

In this definition, the argument $f$ is assumed to be the function that specifies a circuit, $s$ is the state, and $i : is$ is the stream of input signals, with $i$ the first input signal, and $is$ the remaining stream of the input signals. In the **where** clause the function $f$ is applied to the state $s$ and the first input signal $i$, which results in the output signal $o$ and the new state $s'$.

Then the output stream consists of the output $o$, followed by the result of the $run$ function applied to the same hardware specification $f$, the new state $s'$ and the remaining stream $is$ of input signals. As mentioned before, this approach expresses a *Mealy machine* (see Figure 1). The result of our approach is that architecture specifications are plain Haskell functions which make them both syntactically and semantically very straightforward and simpler than the corresponding specifications written in any other functional HDL known to the authors of the present paper.

Further features of our approach are that several abstraction mechanisms are available, such as choice mechanisms, higher order functions, polymorphism, lambda abstraction, and derivability of types.

There are also features of Haskell that do not have a direct counterpart in hardware. We mention dynamic data structures such as lists and trees, and unlimited recursion. However, when at compile time the maximum size of data structures, or the maximum number of recursion steps is known, hardware might in principle be generated. At the moment, the CλaSH prototype is not yet able to do that.

In Section III we introduce CλaSH by discussing several examples, each illustrating some specific language constructs. The examples are preceded by a description of a few special types and operations that are needed for hardware descriptions in Section II.

## II. Preliminary remarks

In this section we will discuss some pre-defined constructs that are typically needed for hardware specifications. In CλaSH the translations of these constructions into VHDL are predefined.

*Hardware types:* First there is, clearly, the type *Bit* which contains two values: *Low* and *High*. Then there is the type *Bool* which contains the boolean values *True* and *False*. The latter type can be used in *if-then-else* expressions.

For integers the constructor *Signed* is available, as in: *Signed* 16, *Signed* 32, etc, where 16, 32 indicate the bitwidth. There also is the constructor *Index*: the type *Index* 12 means that the integer values of this type fall in the range $0 \cdots 12$ (inclusive).

CλaSH does not support dynamic data structures such as lists and trees. Instead, CλaSH recognizes vector types: *Vector n a*, where $n$ is an integer and $a$ an already given type[1]. The interpretation is straightforward: this type denotes a vector of $n$ elements (with indexes $0 \cdots n-1$) of type $a$. The notation $V\ [1,2,3,4]$ is an example of a value of type *Vector* 4 (*Signed* 16), where we assume that $1, 2, 3, 4$ are of type *Signed* 16.

*User defined types:* The designer can also define his own types, though in the present prototype of CλaSH that possibility is limited to enumeration types and record types. We will see an example of enumeration types later on.

*Operations and functions:* In CλaSH several standard Haskell functions for lists have been redefined for vectors. For example, functions such as $head$, $tail$, $init$ (returning the full list, except the last element) and $last$ (returning the last element of a list) are in CλaSH redefined for vectors. Since $cons$ is a data constructor in Haskell, it can not be directly redefined for vectors. Hence, we defined the operations $\triangleright$ for adding an element in front of a vector, and $\triangleleft$ for adding an element to the end of a vector.

Standard higher order functions such as $map$, $zipWith$, etc, are redefined for vectors as well and thus recognized by CλaSH. Also various other features that are standard in Haskell, such as user defined higher order functions, partial application, and polymorphism, are recognized by CλaSH. We will see examples of their use in section III.

[1]Actually, on type level "*Vector n a*" is a slightly simplified notation, but that does not influence the rest of this paper.

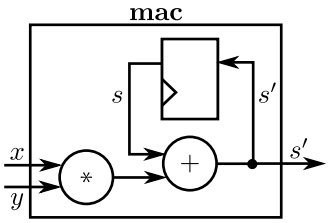


Figure 2. Multiply-accumulate

*Compilation pipeline:* The focus of this paper is on showing the usage of CλaSH in a series of examples. However, without going into details, we will say a few words on the compilation pipeline that CλaSH uses.

The first step is performed by GHC (Glasgow Haskell Compiler) which translates the Haskell specification into an intermediate language, called *Core*. This result is then transformed by applying a set of *rewrite rules* into a *normal form*, which is still written in *Core*, but close to VHDL. The final step, translation of this normal form into VHDL, is now relatively simple.

In fact, the rewriting process results in a *Core* expression that is very close to a netlist format. The reason to choose for a translation into VHDL is the availability of a well-developed toolchain.

## III. Examples

In section III-A we discuss a simple multiply-accumulate architecture, in section III-B some variants of a fir filter are shown, in section III-C a simple cpu, in section III-D a floating point reduction circuit.

### A. Multiply-accumulate

The first example is a simple multiply-accumulate function *mac* (see Figure 2). The input consists of a sequence of pairs of integer numbers $(x, y)$ that have to be pairwise multiplied and accumulated in the state $s$, which in this case consists of a single integer number:

$$
\begin{aligned}
&mac\ s\ (x,y) = (s', s') \\
&\quad \textbf{where} \\
&\qquad s' = s + x * y
\end{aligned}
$$

Let $xs$ and $ys$ be two sequences of numbers, and let $zip\ \ xs\ \ ys$ be the sequence of pairs of corresponding consecutive numbers in $xs$ and $ys$ ($zip$ is a standard Haskell function). Then a simulation yields[2]

$$run\ mac\ 0\ (zip\ \langle 1,2,3\rangle\ \langle 4,5,6\rangle) = \langle 4, 14, 32\rangle$$

Note that in the specification of $mac$ above there is some polymorphism present: it works for any type of value for which $+$ and $*$ exist. So, before CλaSH can translate this definition into synthesizable VHDL, we have to fix the type of $mac$. For example, we might define the type for $mac$ as follows:

$$
\begin{aligned}
mac :: &\ Signed\ 16 \\
&\rightarrow (Signed\ 16, Signed\ 16) \\
&\rightarrow (Signed\ 16, Signed\ 16)
\end{aligned}
$$

[2]Actually, to let the simulation in Haskell end properly, the definition of $run$ above has to be extended with a clause for the empty input sequence.

i.e., the first argument (the state) is of type $Signed\ 16$ and the second argument (the input) is a pair of type $(Signed\ 16, Signed\ 16)$. The result again is a pair of type $(Signed\ 16, Signed\ 16)$, of which the first is the new state, and the second one is the output. That is to say, all values are integers of 16 bits long.

*Remarks:* This first example requires no special definitions or functions and the correspondence between the specification and Figure 2 is immediate.

### B. Variants of a fir-filter

A finite impulse response (fir) filter calculates the dot product of two vectors, i.e., it pairwise multiplies a vector of fixed constants ($h_i$) with an equally long substream of the input ($x_t$), and then adds the results. Thus, the result $y_t$ of a fir-filter at time $t$ is defined as follows:

$$y_t = \sum_{i=0}^{n-1} x_{t-i} * h_i$$

There are many implementations of a fir filter, we show three of them to illustrate that their differences can be concisely expressed in the definitions. In the context of this paper we assume that every clock cycle a new input value arrives.

*Variant 1:* An equivalent Haskell definition is as follows ($hs$ is the vector of constants, $xs$ is the substream of inputs):

$$xs \bullet hs = foldl1\ (+)\ (zipWith\ (*)\ xs\ hs)$$

The function $foldl1$ is a standard Haskell function which applies a binary operation (here: addition) to the first two elements of a vector, and then accumulates the result with the next elements from the vector. Also $zipWith$ is a standard Haskell function which pairwise applies a binary operation (here: multiplication) to the corresponding elements of two vectors. Both $foldl1$ and $zipWith$ are higher order functions since they take a binary operation as their first argument.

The direct implementation $fir_1$ is now specified in Haskell as follows (see Figure 3):

$$\begin{aligned} fir_1\ (hs, us)\ x = &((hs, tail\ us \triangleleft x)\\ &, (us \triangleleft x) \bullet hs\\ &)\end{aligned}$$

Thus, the state of the function $fir_1$ is a pair of two vectors: the fixed values $hs$, and the sequence $us$ of the input values that have to be kept in a sequence of registers. Note that the numbering of the indexes in Figure 3 is the other way around as in the original definition of $y_t$ above, but that is not crucial for the essence of the definition. The same holds for Figures 4, 5.

The result of $fir_1$ consists of two things. First, it contains the new state with $tail\ \ us \triangleleft x$ instead of $us$, i.e., the "oldest" value in $us$ is discarded, and $x$ (the new value) is added at the end. The $hs$-part of the state remains unchanged.

The second part of the result is the output value, i.e., the dot product of the full sequence $us \triangleleft x$ and $hs$.

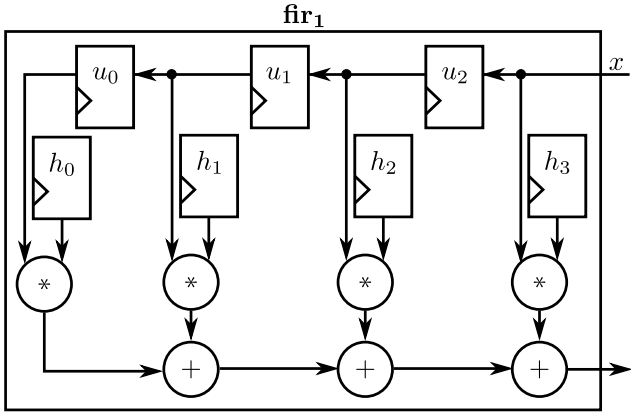


Figure 3. fir-filter, variant 1

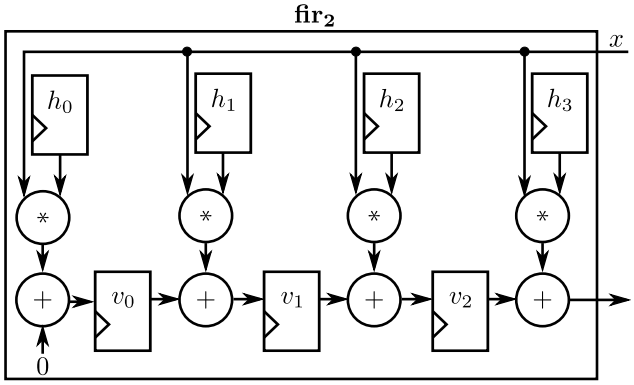


Figure 4. fir-filter, variant 2

*Variant 2:* An alternative definition $fir_2$ of a fir-filter is shown in Figure 4 and defined as follows:

$$\begin{aligned} fir_2\ (hs, vs)\ x = &((hs, init\ vs')\\ &, last\ vs'\\ &)\\ \textbf{where}&\\ &ws = map\ (\lambda h \rightarrow h * x)\ hs\\ &vs' = zipWith\ (+)\ (0 \triangleright vs)\ ws\end{aligned}$$

The standard Haskell function $map$ applies a function to all elements of a vector. In this case that function is denoted by a lambda term which expresses that the argument $h$ is multiplied with $x$. Thus, by using $map$, all elements in $hs$ are multiplied with $x$. Next, the results of this are pairwise added to the values in $0 \triangleright vs$, i.e., a zero prefixed to $vs$.

*Variant 3:* Finally, a third definition $fir_3$ goes as follows (see Figure 5):

$$\begin{aligned} fir_3\ (hs, us, vs)\ x = &((hs, tail\ us \triangleleft x, init\ vs')\\ &, last\ vs'\\ &)\\ \textbf{where}&\\ &ws = zipWith\ (*)\ hs\ (us \triangleleft x)\\ &vs' = zipWith\ (+)\ (0 \triangleright vs)\ ws\end{aligned}$$

It should be clear by now how the $zipWith$ functions take care of the pairwise multiplication and addition. Note that with this last definition the input value $x$ should arrive every other clock cycle, and only every other clock cycle a valid result is delivered.

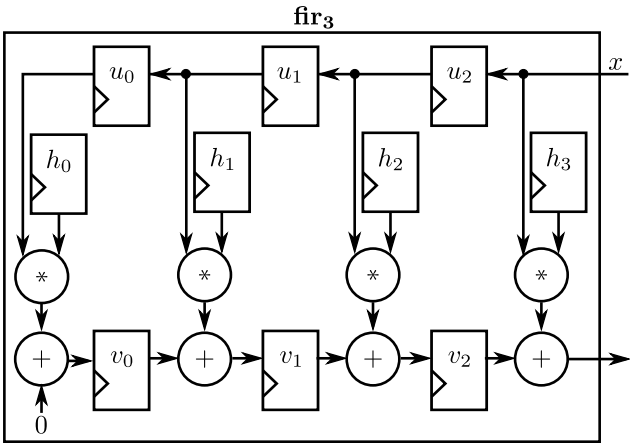


Figure 5. fir-filter, variant 3

*Remarks:* The variants of the fir-filters above exploit several standard higher order functions (*map*, *zipWith*, *foldl1*) which are translated by CλaSH to synthesizable VHDL. Also λ-abstraction is recognized by CλaSH, as can be seen in variant 2.

These features give a high abstraction level to the designs of the fir-filters which makes the essential differences between these variants immediately visible and analyzable, as a comparison of the above definitions shows.

Clearly, as with the multiply-accumulate example, the polymorphic character of these functions leave the concrete type of the fir-filters undecided, so in order to specify concrete hardware, one still has to decide on the types of the fir-filters. The types of $fir_1$, $fir_2$, $fir_3$ differ slightly, for example, the state of $fir_3$ is a tuple of three vectors, whereas for $fir_1$, $fir_2$ the state is a tuple of two vectors. However, the pattern of the type definitions is the same for all three variants, and coincides with the pattern of the general type of the function *arch* as shown in Section I.

Finally, note that the above definitions hold for any number of taps in the fir-filters. This number is fully determined by the *Vector* type for the state parameters chosen by the designer.

### C. Higher order cpu

Next, we describe a higher order cpu, containing three function units $fu_0$, $fu_1$, $fu_2$ (see Figure 6) each of which can perform a binary operation. Every function unit has six data inputs (of type $Signed\ 16$), and two address inputs (of type $Index\ 5$) that indicate which of the six data inputs are to be used as operands for the binary operation that the function unit performs. These six data inputs consist of one external input $x$, two fixed initialization values (0 and 1), and the previous output of each of the three function units. The output of the cpu as a whole is the previous output of $fu_2$. Function units $fu_1$ and $fu_2$ can perform a fixed binary operation, whereas $fu_0$ has an additional input for an opcode to choose a binary operation out of a few possibilities. Each function unit outputs its result into a register, i.e., the state of the cpu is a vector of three $Signed\ 16$ values:

$$\mathbf{type}\ CpuState = Vector\ 3\ (Signed\ 16)$$

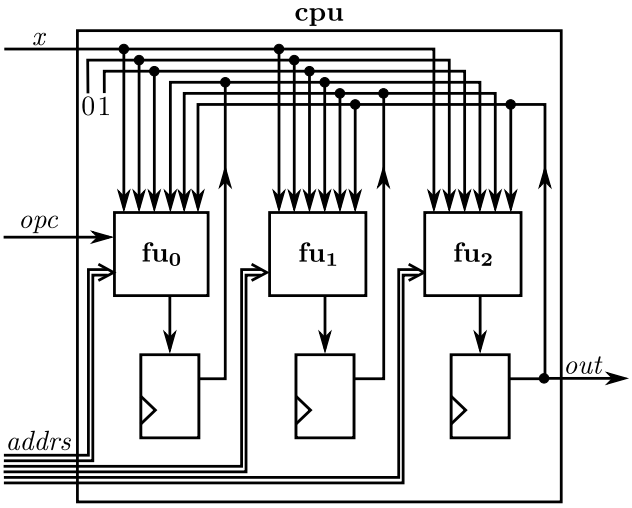


Figure 6. Higher order cpu

The type of the cpu as a whole can now be defined as (*Opcode* will be defined later):

$$\begin{aligned} cpu :: &\ CpuState \\ &\rightarrow (Signed\ 16, Opcode, \\ &\qquad Vector\ 3\ (Index\ 5, Index\ 5)) \\ &\rightarrow (CpuState, Signed\ 16) \end{aligned}$$

Every function unit can be defined by the following higher order function, *fu*, which takes three arguments: the operation *op* that the function unit should perform, the six inputs, and the address pair $(a_0, a_1)$. It selects two inputs, based on these addresses, and applies the given operation to them, returning the result ("!" is the operation for vector-indexing):

$$fu\ op\ inputs\ (a_0, a_1) = op\ (inputs\,!\,a_0)\ (inputs\,!\,a_1)$$

Exploiting partial application we now define (assuming that the binary functions *add* and *mul* already exist):

$$\begin{aligned} fu_1 &= fu\ add \\ fu_2 &= fu\ mul \end{aligned}$$

Note that the types of these functions can be derived from the type of the cpu function and their usage below, thus determining what component instantiations are needed. For example, the function *add* should take two $Signed\ 16$ values and also deliver a $Signed\ 16$ value.

In order to define $fu_0$, the type *Opcode* and the function *multiop* that chooses a specific operation given the opcode, are defined first. It is assumed that the binary functions *shift* (where *shift a b* shifts *a* by the number of bits indicated by *b*) and *xor* (for the bitwise *xor*) exist.

$$\begin{aligned} &\mathbf{data}\ Opcode = Shift \mid Xor \mid Equal \\ &multiop\ Shift\ = shift \\ &multiop\ Xor\ = xor \\ &multiop\ Equal = \lambda a\ b \rightarrow \mathbf{if}\ a \equiv b\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \end{aligned}$$

Note that the result of *multiop* is a binary function from two $Signed\ 16$ values into one $Signed\ 16$ value (hence, the **if-then-else** is needed since $a \equiv b$ is a boolean). The type of *multiop* can be derived by the Haskell type system from the context.

The definition of $fu_0$, which takes an opcode as additional argument, is:

$$fu_0\ c = fu\ (multiop\ c)$$

The complete definition of the function $cpu$ now is (note that $addrs$ contains three address *pairs*):

$$\begin{aligned}
&cpu\ s\ (x, opc, addrs) = (s', out)\\
&\quad \textbf{where}\\
&\qquad inputs = x \triangleright (0 \triangleright (1 \triangleright s))\\
&\qquad s' \quad = V\ [\ fu_0\ opc\ inputs\ (addrs\,!\,0)\\
&\qquad\qquad\quad , fu_1 \quad inputs\ (addrs\,!\,1)\\
&\qquad\qquad\quad , fu_2 \quad inputs\ (addrs\,!\,2)\\
&\qquad\qquad\quad ]\\
&\qquad out \quad = last\ s
\end{aligned}$$

Due to space restrictions, Figure 6 does not show the internals of each function unit. We remark that C$\lambda$aSH generates e.g. $multiop$ as a subcomponent of $fu_0$.

*Remarks:* In this example it is shown that also *user defined* higher order functions can be compiled by C$\lambda$aSH, in this case the function $fu$. Note that in using this function, one may also exploit *partial application*, as in the definitions of $fu_0, fu_1, fu_2$.

In this example it is also shown that the designer may define his own enumeration types. As a final feature of C$\lambda$aSH shown in this example we mention pattern matching: the function $multiop$ is defined by pattern matching on the values of the type $Opcode$.

### D. Floating point reduction circuit

The final example is a reduction circuit in which sequences of floating point numbers are added. Numbers come in one per clock cycle, sequence after sequence. When a sequence is finished, no further numbers belonging to that sequence will arrive.

We assume a pipelined floating point adder which we will exploit as optimally as possible, numbers belonging to different sequences may be in the pipeline at the same time. Only numbers belonging to the same sequence should be added together, so in order to keep numbers belonging to different sequences separated, they are labelled. This algorithm is introduced in [11] where it is also proven that numbers indeed may come in one per clock cycle without causing buffers to overflow.

The example shows that C$\lambda$aSH can deal with architectures which consist of several components, where each component has its own state and is defined as a separate function.

The input $(x, i)$ (see Figure 7) consists of a number and its row index. Since there will only be a limited number of rows "active" in the system, a limited number of labels is needed to distinguish different rows from each other. The discriminator component $discr$ transforms the row index $i$ into such a reduced label $d$ after which the pair $(x, d)$ enters the input component $inp$ (which has a fifo $\iota$ as internal state). The boolean signal $new_d$ says whether a new row starts (hence, the discriminator needs internal memory $\delta$), and is used by the partial result buffer $res$ to decide whether position $d$ may be re-used for intermediate

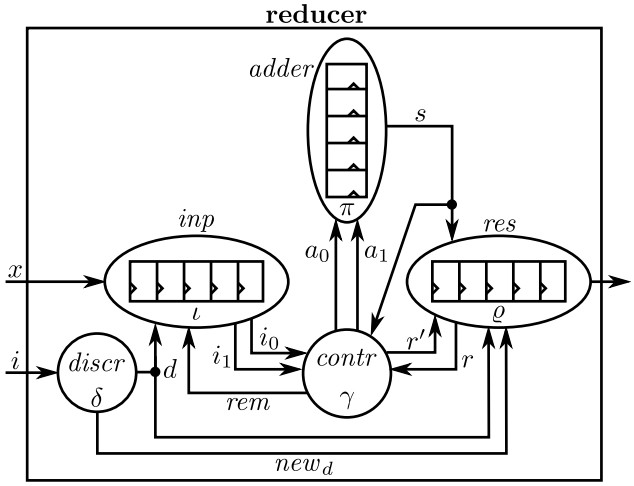


Figure 7. Reduction circuit

results of this new row. Both the memory $\varrho$ in $res$ and the number of labels used are big enough to be sure that the row which had label $d$ before is ready at the moment $d$ is re-used. Finally, the pipelined floating point adder $adder$ (with internal state $\pi$) takes two numbers $a_0, a_1$ and outputs their sum several clock cycles later. Note that the pipeline $\pi$ need not be completely full, so a value $s$ delivered by $adder$ may be undefined.

The central controller $contr$ gathers the output $s$ from $adder$, the corresponding partial result $r$ from $res$ (or an undefined value in case there is no corresponding previous result for the same row), and the first two elements $i_0, i_1$ from $inp$ (without going into detail we remark that $i_0$ is always valid, whereas $i_1$ may be undefined). Based on these inputs, $contr$ decides which values $a_0, a_1$ will be input into $adder$, which value $r'$ will be given back to $res$, and the number of values $rem$ that will be used from $inp$ (and thus have to be removed from $\iota$). This is done according to the following rules (in order of priority):

1) when $s$ and the corresponding result $r$ are both defined, then $s$ and $r$ together enter *adder*,
2) when $s$ and the first element $i_0$ from $inp$ have the same label, then $s$ and $i_0$ enter *adder*,
3) when $i_0, i_1$ are both defined and their labels are the same, then $i_0$ and $i_1$ enter *adder*,
4) when $i_0, i_1$ are both defined but their labels are different, then $i_0$ and 0 enter *adder*,
5) when none of the above applies, no number enters *adder*.

In addition, when a number $s$ with label $d$ comes out of $adder$ but $s$ will not re-enter $adder$, $s$ will be given to $res$ for later use. Remember that every clock cycle a new value $x$ enters *inp*.

In the context of this paper we will only show the definitions of the controller $contr$ and of the full reduction circuit $reducer$. As seen above, there are $valid$ values, consisting of a number and a label, and there are $invalid$ values. We define the type $RValue$ for these values, consisting of a valid flag, the value of the number, and its label.

$$\textbf{type}\ RValue = (Bool, (Float, Index\ 127))$$

Three functions are needed to deal with such values (*fst*, *snd* give the first, second element of a 2-tuple):

$$\begin{aligned} value\ a &= fst\ (snd\ a) \\ lbl\ \ \ a &= snd\ (snd\ a) \\ valid\ a &= fst\ a \end{aligned}$$

In addition, we define the constants $nv$ (for "not valid") as $(False, (0, 0))$ and $zero$ as $(True, (0, 0))$. The definition of the controller can now be formulated as follows (note that the state parameter $\gamma$ does not change, i.e., $\gamma$ is empty. It is only there to match the required global structure of the definition):

$$\begin{aligned} &contr\ \gamma\ (i_0, i_1, s, r) = (\gamma, (a_0, a_1, rem, r')) \\ &\quad \textbf{where} \\ &\quad\quad (a_0, a_1, rem, r') \\ &\quad\quad\quad \mid valid\ s\ \wedge valid\ r\ \ \ \ \ \ \ \ \ \ \ \ = (s\ \ , r\ \ \ \ , 0, nv) \\ &\quad\quad\quad \mid valid\ s\ \wedge lbl\ s\ \equiv lbl\ i_0 = (s\ \ , i_0\ \ \ , 1, nv) \\ &\quad\quad\quad \mid valid\ i_1 \wedge lbl\ i_0 \equiv lbl\ i_1 = (i_0\ , i_1\ \ \ , 2, s\ \ ) \\ &\quad\quad\quad \mid valid\ i_1\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = (i_0\ , zero, 1, s\ \ ) \\ &\quad\quad\quad \mid otherwise\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ = (nv, nv\ \ , 0, s\ \ ) \end{aligned}$$

The guards (indicated by "|", meaning "under the condition that") in this definition express the rules given above. Note that pattern matching is exploited in the way values are given to the four elements $(a_0, a_1, rem, r')$.

The definition of the full reduction circuit now looks as follows:

$$\begin{aligned} &reducer\ (\delta, \iota, \pi, \varrho, \gamma)\ (x, i) = ((\delta', \iota', \pi', \varrho', \gamma'), out) \\ &\quad \textbf{where} \\ &\quad\quad (\delta', (new_d, d)) \ \ \ \ \ \ \ \ = discr\ \ \delta\ \ i \\ &\quad\quad (\iota'\ , (i_0, i_1)) \ \ \ \ \ \ \ \ = inp\ \ \ \ \iota\ \ (d, x, rem) \\ &\quad\quad (\pi', s) \ \ \ \ \ \ \ \ \ \ \ \ \ \ = adder\ \pi\ \ (a_0, a_1) \\ &\quad\quad (\varrho', (r, out)) \ \ \ \ \ \ \ \ = res\ \ \ \ \varrho\ \ (d, new_d, s, r') \\ &\quad\quad (\gamma', (a_0, a_1, rem, r')) = contr\ \ \gamma\ \ (i_0, i_1, s, r) \end{aligned}$$

Note that loops shown in the picture correspond to loops in the code, for example, $a_0$ is a result of $contr$ and an argument for $adder$. At the same time, $s$ is a result of $adder$ and an argument for $contr$. In hardware there is no problem since these values come from memory elements. Also in the Haskell simulation there is no problem because of lazy evaluation.

*Remarks:* This example shows that *guards* can be dealt with by C$\lambda$aSH. It also shows how to combine several components of an architecture together. However, to make the simulation run and to let GHC do its job properly, for now we have to mention the states of nested components in the signature of the combining component.

This reduction circuit was also written and hand-optimized in VHDL by the authors of [11]. Both the VHDL and the functional specification made the same global design decisions and local optimizations. Though it is difficult to compare the exact details of both specifications, the results of synthesizing both were very close: clock speed (around 170 MHz) and area (around 4500 CLB slices & LUTs) were within 10% of each other.

## IV. Future research

Several topics are still under development in C$\lambda$aSH, an already mentioned topic being (limited) recursion. One topic which is being improved at the moment is to suppress the need to show the inner states of nested components on a higher level.

Other topics in future research are how to express multi-clock domains and how to deal with asynchronous hardware.

## References


[1] L. Cardelli and G. Plotkin, An Algebraic Approach to VLSI Design, in: *Proceedings of the VLSI 81 International Conference*, 1981, pp. 173–182.

[2] M. Sheeran, $\mu$FP, a language for VLSI design, in: *LFP 84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, New York, NY, USA, 1984, pp. 104–112.

[3] S. D. Johnson, Applicative programming and digital design, in: *POPL84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, 1984, pp. 218–227.

[4] R. Sharp and O. Rasmussen, Using a language of functions and relations for VLSI specification, in: *FPCA 95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, New York, NY, USA, 1995, pp. 45–54.

[5] Y. Li and M. Leeser, HML, a novel hardware description language and its translation to VHDL, in: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2000, vol. 8, no. 1, pp. 1-8.

[6] J. Matthews, B. Cook, and J. Launchbury, Microprocessor specification in Hawk, in: *Proceedings of 1998 International Conference on Computer Languages*, 1998, pp. 90–101.

[7] E. Axelsson, K. Claessen, and M. Sheeran, Wired: Wire-Aware Circuit Design, in: *Proceedings of Conference on Correct Hardware Design and Verification Methods (CHARME)*, LNCS 3725. Springer, 2005, pp. 5–19.

[8] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, Lava: hardware design in Haskell, in: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, New York, USA, 1998, pp. 174–184.

[9] I. Sander, A. Jantsch, System Modeling and Transformational Design Refinement in ForSyDe, in: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2004, vol. 23, no. 1, pp. 17–32.

[10] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, Introducing kansas lava, November 2009, submitted to *The International Symposia on Implementation and Application of Functional Languages (IFL)'09*, Available: http://ittc.ku.edu/ andygill/papers/kansas-lava-ifl09.pdf

[11] M.E.T. Gerards, J. Kuper, A.B.J. Kokkeler, E. Molenkamp, Streaming Reduction Circuit, in: *Proceedings of the 12th EUROMICRO Conference on Digital System Design, Architectures, Methods and Tools*, Patras, Greece, 2009, pp. 287–292