Hardware Specification with CλaSH

Jan Kuper(✉)

University of Twente, Enschede, The Netherlands
j.kuper@utwente.nl

Abstract. CλaSH is a recently developed system to specify and synthesize hardware architectures, strongly based on the functional programming language Haskell. Different from other existing approaches to describe hardware in a functional style, CλaSH is not defined as an embedded language inside Haskell, but instead, CλaSH uses Haskell *itself* to specify hardware architectures. In fact, every CλaSH specification is an executable Haskell program. Hence, the simulation of a hardware architecture is immediate, and all abstraction mechanisms that are available in Haskell are maintained in CλaSH, insofar they are directly applicable to the specification of hardware.

This paper describes several examples of specifications of hardware architectures in CλaSH to illustrate the various abstraction mechanisms that CλaSH offers. The emphasis is more on the CλaSH-*style* of specification, than on the concrete technical details of CλaSH. Often, the specifications are given in plain Haskell, to avoid some of the specific CλaSH details that will be indicated in a separate section.

The given examples include regular architectures such as a ripple carry adder, a multiplier, vector and matrix multiplications, finite impulse response filters, and also irregular architectures such as a simple Von Neumann style processor and a reduction circuit. Finally, some specific technicalities of CλaSH will be discussed, among others the processing pipeline of CλaSH and the hardware oriented type constructions of CλaSH.

1 Introduction

In this paper we describe the hardware specification environment CλaSH, which is based on the functional programming language Haskell. The perspective from which a CλaSH specification views a hardware architecture is that of a *Mealy Machine*, that is, as a function of two arguments – one representing the state of a component and the other the input – which yields two results – the new state and the output. We will show several examples in CλaSH, ranging from matrix product and FIR-filters, to a simple processor and a reduction circuit.

Since hardware architectures have specific properties, some extensions have to be added to Haskell, and furthermore, not every Haskell program can be translated into hardware. For example, the data type of *lists*, which is often

Jan Kuper—This work is partly supported by EU project POLCA, FP7-ICT-2013-10, grant agreement no. 610686.

V. Zsók et al. (Eds.): CEFP 2013, LNCS 8606, pp. 336–380, 2015.
DOI: 10.1007/978-3-319-15940-9_8

used in a functional setting, is not suitable to describe architectures since a list may vary in length during a computation, whereas hardware is fixed in size. Besides, data dependent recursion is not possible in CλaSH, since that requires transformations that are not (yet) included in the CλaSH compiler.

In the rest of this paper we first shortly discuss some related work (Sect. 2), after which we outline the pattern along we will specify architectures (Sect. 3). Then, in Sect. 4 we first describe some regular architectures and in Sect. 5 we describe some irregular architectures. in both sections we give examples of state *less* architectures and state*ful* architectures. Finally, in Sect. 6 we give an informal description of some aspects of CλaSH itself.

2 Related Work

The most well-known specification languages for digital hardware are VHDL and Verilog. Also in industry, the design of digital architectures is mostly expressed in VHDL and Verilog. However, abstraction mechanisms available in these languages are not very strong and it is cumbersome to generalize a given specification for different input/output types, or to parameterize for the functionality in a subcomponent. Over the years several attempts are made to improve the abstraction mechanisms in these languages, leading to concepts such as *generics* and *generate statements*. With *generics* a design can be formulated exploiting – a limited form of – polymorphism such that one may use the same design for different types (see [9]).

However, full abstraction is reached only to a limited extent by these extensions, such that using them is still quite verbose and error-prone. Besides, these extensions are not fully supported by synthesis tools. This is widely recognized by the hardware design community, and there are many attempts to base hardware design on standard programming habits, notably on imperative languages such as C/C++ or Java, leading to so-called *high level synthesis*. A well known example of this approach is *System-C*, for an overview we refer to [6].

The perspective from which both VHDL and Verilog, as well as high level synthesis languages view a hardware architecture is — at least partially — imperative in nature. On the other hand, we argue that the concept of digital hardware is closer to a *function*, than to an imperative *statement*: a digital circuit may be viewed as a structure that transforms an input signal into an output signal, exactly what a function in mathematics does, though in the case of a function one speaks of arguments and results rather than input signals and output signals.

This observation makes it likely that a functional language might be better suitable to specify hardware architectures than languages which are partly based on an imperative perspective. Besides, abstraction mechanisms available in functional programming languages are high, and include features such as higher order functions, polymorphism, lambda abstraction, and pattern matching. This observation was made several times before, dating back to the early eighties of the 20th century, and is expressed in papers such as [4,10,15]. Since then several languages are proposed which approach the specification of hardware architectures from a functional perspective, some of the most important

ones being Lava [3,8], Bluespec [13], ForSyDe [14]. For an overview of several of these languages see [5].

Most of these functional hardware description languages, however, are *embedded languages* inside a functional programming language, which has certain limitations concerning the abstraction mechanisms that are available in a functional language. For example, choice constructs that most functional language offer, such as guards, pattern matching and case constructs, are not easily exploitable in embedded languages, and give rise to more verbose formulations.

On the other hand, the method described in this paper, called CλaSH, uses the functional programming language Haskell *itself*. Hence, all above mentioned abstraction mechanisms that are available in Haskell are automatically also available in CλaSH. It falls outside the scope of this introduction into CλaSH to go into further details concerning a comparison with other functional hardware description languages.

3 Basic Program Structure for Hardware Descriptions

In this section we will give a first introduction to the general principles according to which a CλaSH specification is built up.

3.1 Mealy Machine and Simulation

Below we assume that a hardware architecture consists of memory elements together with a combinatorial circuit, and that it is connected to input and output ports. The values in the memory elements form the *state* of the architecture, whereas the combinatorial circuit generates its *functionality*. At every clock cycle, the input signals and the values from the memory elements are going into the combinatorial circuit, defined by some function f, which results in output signals and in new values in the memory elements. Thus, the general format of the function f is that f has *two* arguments (the state and the input) and the result of f consists of two values as well (the updated state and the output):

$$f\ s\ x\ =\ (s', y) \tag{1}$$

where s denotes the current content of the state, x is the input, s' is the updated value of the state, and y is the output of the circuit described by f[1]. Clearly, both the new state s' and the output y must be defined separately, but we will come to that later. Here only the top-level structure of the definition of a hardware specification function f is relevant.

This function f describes the *structure* of the architecture, in addition we need a function to *simulate* the described architecture. The simulation works by executing the hardware function f repeatedly, on every clock cycle. The following function *simulate* realizes this simulation process:

$$\begin{aligned} simulate\ f\ s\ (x{:}xs) &= y\ :\ simulate\ f\ s'\ xs \\ &\quad \textbf{where} \\ &\quad (s', y)\ =\ f\ s\ x \\ simulate\ f\ s\ [\,] &= [\,] \end{aligned}$$

[1] Note that we follow the convention used in Haskell, and write $f\ s\ x$ instead of $f(s, x)$.

This definition consists of *two* clauses, where the difference is in the third argument (`x:xs` vs `[]`, see below). The function *simulate* can be used for simulation purposes by applying it to a given initial value of the state and a given list of concrete arguments and executing that in Haskell.

The function *simulate* has *three* arguments, which can be described as follows:

- the first argument is a *function* f, which determines the functionality of some hardware architecture as described above. We emphasize that f is just a formal parameter of the function *simulate*, i.e., with every usage of *simulate* this parameter f will be instantiated to the functionality of a concrete hardware architecture.

  Since *simulate* has a function as argument, *simulate* is called a "higher order function".
- the second argument is the *state* s, which contains all the values in all memory elements in the architecture. Note that s need not be a simple parameter, consisting of just one integer (say). Instead, s may be a structured parameter which consists of several parts representing various memory elements.
- the third argument of the function *simulate* is the *list of inputs*, denoted by the "patterns" $x{:}xs$ and $[\,]$, respectively. The second pattern denotes the empty list, so the second clause only will be chosen when the input is empty, i.e., when all input values are processed by the first clause (in case an input list is finite).

  The first pattern $x{:}xs$ denotes a non-empty list of input values, so the first clause is chosen as long as the input still contains values. The colon ":" breaks the input in its first element x and the rest xs (suggesting the plural of one x, and pronounced as x-es). The value x will be dealt with during the present clock cycle, and xs will be dealt with in future.

  Here too, x may be a compound value, consisting of several parts which all come in parallel during the same clock cycle.

In the result of *simulate* the values y and s' are used, which are calculated by the function f. The result of f then consists of a *pair* (s', y) of two things: the output y (which again may consist of several parallel values), and the new state s'. This corresponds to the idea of a *Mealy machine*, as depicted in Fig. 1.

The global result of the *simulate*-function now is the output y followed by (indicated by ":") a recursive call of the function *simulate*, but with the new state s' and the rest of the input xs. That means that the function *simulate* repeats itself, each time with the state resulting from the previous execution of *simulate*, and with the rest of the input sequence. Thus, the total result of the function *simulate* is a list of outputs generated by a repeated evaluation of the architecture f, meanwhile updating the state at every step.

Note that the function *simulate* simulates a clock cycle at every recursive call. Note also that we assume that at every clock cycle a new input value x is available, though that can be weakened by choosing a *Maybe* type for the input values, indicating that x can be a meaningful value, or *Nothing*.

3.2 A Simple Architecture Example

In this section we will give an example of definition of a concrete architecture by means of a function f according to the pattern as shown in Eq. (1). We will also show the simulation of this architecture, using of the function *simulate* as defined above.

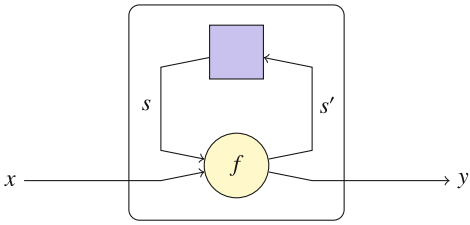


Fig. 1. Mealy machine

Suppose we have to calculate the dot product of two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of integers, i.e., we have to calculate the following expression:

$$\sum_{i=1}^{n} x_i \cdot y_i.$$

Suppose further that we have only one multiplier and one adder available. Then we clearly need a memory element *acc* to accumulate intermediate results of the addition, and which should initially contain the value 0. In Fig. 2 the architecture is shown that does the job: at each clock cycle the inputs x_i and y_i are first multiplied, and then added to the value in the accumulator *acc*. The result of this is put both back into the accumulator, and on the output.

Description in a Functional Language. The function `macc` (for "multiply-accumulate", see Fig. 2), which expresses the above behavior, may be defined as follows:

$$\begin{aligned} &macc \;::\; Int \;\rightarrow\; (Int, Int) \;\rightarrow\; (Int, Int) \\ &macc\ acc\ (x, y) = \;\; (z, z) \\ &\qquad\qquad\qquad\quad \textbf{where} \\ &\qquad\qquad\qquad\quad z \;=\; acc \;+\; x * y \end{aligned}$$

On the first line in this definition the *type* of the function *macc* is mentioned, which expresses that *macc* is a *function* with an *Int* as its first argument, a pair *(int,int)* as its second argument, and a pair *(int,int)* as its result.

We remark that the structure of the function *macc* matches the structure of the function f in Eq. (1) and of the function f in the definition of *simulate*. That is to say, where in the definition of *simulate* the pair (s', y) is calculated by *using* the function f, now the function *macc* is *defined* such that it can be used in the role of f.

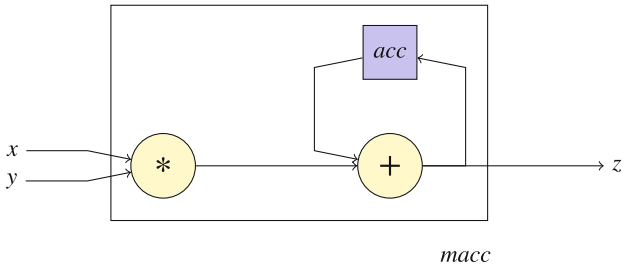


Fig. 2. Multiply-accumulate

To explain the correspondence between *macc* and f in greater detail, we observe that

- the first argument *acc* is the state of the architecture, and corresponds to s in the expression "$f\ s\ x$" in the function *simulate*. In the case of *macc*, the state consists of a single number only,
- the second argument (x, y) is the input that arrives at each clock cycle, and corresponds to the parameter x in the expression "$f\ s\ x$" in the function *simulate*. In this case the input consists of two numbers,
- the result (z, z) matches the pair (s', y) in the definition of *simulate*, so in this example both the output and the new content of the state are the same value z. For reasons of readability we use a where-clause to define z, though we might have written directly

$$macc\ acc\ (x, y)\ =\ (acc + x * y, acc + x * y).$$

Suppose we want to simulate and test this architecture with the vectors:

$$\boldsymbol{x} = \langle 1, 2, 3, 4 \rangle$$
$$\boldsymbol{y} = \langle 5, 6, 7, 8 \rangle$$

Then the input for the architecture is a sequence of parallel x and y values, as follows (in Haskell notation as a list of 2-tuples):

$$input\ =\ [(1, 5),\ (2, 6),\ (3, 7),\ (4, 8)]$$

The initial value of the accumulator is 0, so in Haskell we can now simulate this by evaluating:

$$simulate\ macc\ 0\ input$$

The output of the simulation then is:

$$[5, 17, 38, 70]$$

The last value is the dot product of the two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$.

Description in VHDL. In order to illustrate some differences between a functional language and a standard hardware specification language, we describe the same multiply-accumulator in VHDL. One possible specification is as follows (leaving out the standard initial `LIBRARY` and `USE` statements):

```
ENTITY macc IS
  PORT (x, y : IN  integer;
        z    : OUT integer;
        rst,
        clk  : IN  std_logic);
END macc;

ARCHITECTURE behaviour OF macc IS
  SIGNAL acc : integer;
  SIGNAL zi  : integer;
BEGIN
  zi  <= acc + x * y;

  acc <= 0  when rst='0'           else
         zi when rising_edge(clk);

  z   <= zi;
END behaviour;
```

Assuming that the reader is not familiar with VHDL, we make some remarks about this specification. First of all we remark that, in order to keep the VHDL code as short as possible, we omitted the size of the type `integer` from the above code.

Second, we remark that in VHDL it is not allowed to read from an `OUT` signal, hence inside the architecture a local signal `zi` (for "`z`-internal") is defined which is used for both the `OUT` signal `z` and for the accumulator `acc`.

We further remark that the `when` statement is shorthand notation for a concurrent `process`.

Concerning a comparison between CλaSH and VHDL we restrict ourselves to some obvious differences. A more detailed comparison falls outside the scope of this text.

A first difference of course is the huge difference in syntactical notation: what is a "type" in Haskell corresponds to a certain extent to an "entity" in VHDL, and what is a "function definition" in Haskell, corresponds more or less to an "architecture" in VHDL. We remark that in a functional language the concept of "type" is wider than in VHDL, for example, in a functional language for every type a and b, the type $a \to b$ is the type of all functions from a to b.

As a second difference we mention that in the VHDL-specification time and space are mixed in the sense that references to the clock (`clk`) are present on the same level in the code as the description of the functionality of the architecture. In the functional specification, on the other hand, time and space are strictly

separated: the clock is represented by the recursion in the *simulate* function, whereas the circuit itself is described in the "architecture function" (such as *macc*).

Note that in VHDL also a reset (`rst`) is present, whereas in the functional specification a reset is not expressed. Without going into details we mention that adding a reset to the functional specification is done on the level of the *simulate* function as well, by distinguishing it as a special type of input value. That means that for a reset too it holds that in VHDL it is part of the code describing the architecture, whereas in a functional description it is dealt with on a separate level.

A third difference has to do with the way how we understand the code: in a functional specification we are strictly talking about *values* of variables, such that a functional description is very close to a mathematical, structural description. In VHDL one is more tempted to understand the code as a description of *behavior*, i.e. what actions take place under certain conditions. In the *macc*-example one might say that the outcome of the expression *macc acc* (x, y) *is* the value (z, z), whereas in VHDL one has to perform an action of putting the value of an expression on a signal (channel).

4 Regular Architectures

In this section we will discuss several examples of regular architectures, and illustrate the power of higher order functions to specify such architectures. In particular we will define a ripple carry adder, a multiplier, and several variants of an FIR-filter. Besides, we will show that the fact that functions are *first class citizens* in Haskell can be used to parameterize architecture specifications beyond the level of numerical constants, i.e., we show that we can parameterize an architecture with respect to the functionality of its subcomponents.

4.1 Introduction

To introduce the topic of this approach we start with the dot product as already discussed in Sect. 3.2. First we repeat the definition of the dot product:

$$\boldsymbol{x} \bullet \boldsymbol{y} = \sum_{i=0}^{n-1} x_i \cdot y_i \tag{2}$$

and mention that in Sect. 3.2 the dot product was calculated by an architecture which performed one multiplication and one addition per clock cycle. Consequently, there were as many clock cycles needed as there were elements in the vectors to calculate the full dot product. It is however also possible to execute the calculation of the dot product in a single clock cycle, by using more multipliers and adders. In Fig. 3 the architecture is shown that calculates a dot product in one clock cycle — assuming of course that all adders and multipliers also take a single clock cycle. Clearly, there are hardware limitations to the number

of components that can be reasonably executed in a single clock cycle. Besides, the more components are combined in a so-called *combinatorial path*, the more energy it takes. However, we will ignore such aspects here and concentrate on the structure of the architecture and its specification.

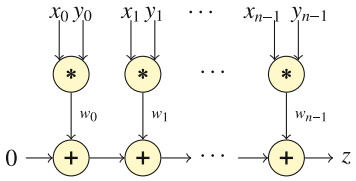
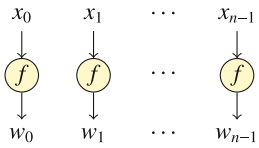


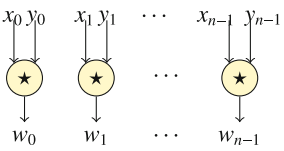
Fig. 3. Dot product

As can be seen from Fig. 3, this is a rather regular structure, in which the same combination of operations is repeated several times. In words the dot product can be described as follows: multiply the corresponding values pairwise, and add the results. In Haskell there exist the functions *zipWith* and *foldl* which perform exactly these operations. Before we come to the definition of the dot-product and the convolution example in Haskell, we first give the meaning in hardware of some standard higher order functions.

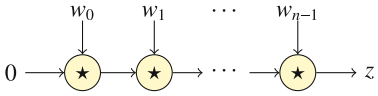
Some standard higher order functions. We show the architectures indicated by the standard higher order functions: *map*, *zipWith*, and *foldl*. Note that the architectures hold for any function f and for any operation $\star$.



(a) *map*



(b) *zipWith*



(c) *foldl*

Fig. 4. Some standard higher order functions

(a) *map.* The function *map* applies a function to all elements in a given list of elements, for example:

$$\begin{aligned} map\ (+1)\ [3,5,8,6] &= [3{+}1, 5{+}1, 8{+}1, 6{+}1] \\ &= [4,6,9,7] \end{aligned}$$

Here, the function $(+1)$ is applied to all the numbers is the list $[3,5,8,6]$. The meaning of *map* as an architecture specification is shown in Fig. 4(a).

(b) *zipWith.* The function *zipWith* combines two sequences of elements by applying a given binary operation or function to the elements of the lists pairwise. For example:

$$\begin{aligned} zipWith\ (+)\ [3,5,8,6]\ [4,6,9,2] &= [3{+}4, 5{+}6, 8{+}9, 6{+}2] \\ &= [7,11,17,8] \end{aligned}$$

The architectural meaning of *zipWith* is shown in Fig. 4(b). Note that both *map* and *zipWith* are strongly parallel.

(c) *Variants of fold.* There are several variants of *fold*: *foldl*, *foldr*, *foldl*1, *foldr*1. Here we only show *foldl* (for *fold-left*), which intuitively works as follows (see Fig. 4(c)):

$$\begin{aligned} foldl\ (+)\ 0\ [7,11,17,8] &= (((0+7)+11)+17)+8 \\ &= 43 \end{aligned}$$

The "left" nature of these operations is indicated by the brackets, saying that the operation (addition in this case) proceeds from left to right through the list.

We remark that for associative operations it is more efficient to give the architecture of a *fold* function the form of a tree, but in the context of this text we ignore such issues of efficiency.

Below we first describe some regular architectures which do not have state and after that we describe some regular architectures which do have state. In particular, in Sect. 4.2 we describe matrix operations and elementary arithmetical architectures, and in Sect. 4.3 we describe some variants of FIR-filters.

4.2 Regular Stateless Architectures

In this section we describe again the dot product of two vectors, followed by matrix-vector multiplication and matrix-matrix multiplication.

Dot Product. Combining the architectures of the functions *foldl* and *zipWith*, we can describe the dot product from Fig. 3 as follows:

$$\boldsymbol{x}\ .*.\ \boldsymbol{y}\ =\ foldl\ (+)\ 0\ \ (zipWith\ (*)\ \boldsymbol{x}\ \boldsymbol{y})$$

Equivalently, in a somewhat more elaborate notation we may define ($\boldsymbol{w}$ and z refer to Fig. 3):

$$\begin{aligned} \boldsymbol{x}\ .*.\ \boldsymbol{y} = \ & z \\ & \textbf{where} \\ & \boldsymbol{w}\ = zipWith\ (*)\ \boldsymbol{x}\ \boldsymbol{y} \\ & z\ = foldl\ (+)\ 0\ \boldsymbol{w} \end{aligned}$$

We choose for ".∗." as notation for the dot product, since the notation "•" cannot be typed directly on a keyboard. We remark that both definitions are valid Haskell definitions, and thus executable in a simulation.

CλaSH translates specifications given in terms of higher order functions like *zipWith* and *foldl*, and in the case of the definition of the dot product, it indeed yields the architecture as shown in Fig. 3.

Matrix-Vector Product. We continue the usage of higher order functions by discussing a matrix vector product, an example being given in Fig. 5.

$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \\ 41 & 42 & 43 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 74 \\ 134 \\ 194 \\ 254 \end{pmatrix}$$

Fig. 5. Matrix-vector product

A fairly standard way to deal with matrices is to consider them as a sequence of rows, thus the matrix in Fig. 5 actually is represented in Haskell as

$$[[11, 12, 13],\ [21, 22, 23],\ [31, 32, 33],\ [41, 42, 43]]$$

That is to say, a row in the matrix is in fact an *element of* the matrix.

Now note that the i-th element of the result of the matrix-vector multiplication is obtained by taking the dot product of i-th row with the vector. For example,

$$[21, 22, 23]\ .*.\ [1, 2, 3]\ =\ 134.$$

Hence, the result vector is computed by applying the dot product with the vector $[1, 2, 3]$ to every row in the matrix. And thus, since the matrix is a list of rows, this can be done by the *map* function. In other words, if xss is a matrix (seen as a list of lists, hence the notation "xss"), and ys is a vector, then the matrix-vector multiplication mxv can be defined as

$$mxv\ xss\ ys\ =\ map\ (.*.\ ys)\ xss$$

Matrix-Matrix Product. This can even further be extended to matrix-matrix-multiplications, see Fig. 6.

To define matrix-matrix multiplication in Haskell, let xss and yss be two matrices, then matrix-matrix multiplication is obtained by multiplying the matrix xss with every *column* of matrix yss, which gives the *columns* of the result matrix. Thus, when we first *transpose* the matrix yss, and transpose the result back, then the above reasoning applies to the rows of yss. Hence, the multiplication of matrix xss with matrix yss (denoted as the function mxm) may be defined as

$$mxm\ xss\ yss\ =\ transpose\ (map\ (mxv\ xss)\ (transpose\ yss))$$

We invite the reader to test these definitions in Haskell.

Generating Architectures by CλaSH. In order to offer the above definitions on vector and matrix operations to CλaSH, we first have to add an empty state arguments, e.g., as in

$$\begin{aligned} mxv'\ s\ (xss, ys) = &\ (s, zs) \\ &\textbf{where} \\ &zs\ =\ mxv\ xss\ ys \end{aligned}$$

Note that the state s remains unchanged, so it can be anything we like, the most obvious choice being $s = ()$. Note further, that the input of the architecture formally is one item (xss, ys) again, though it consists of many elements. Extended in this way, CλaSH translates these definitions into hardware architectures, performing the described operations directly in hardware. For matrix-vector multiplication of the size as in Fig. 5 the resulting architecture looks as follows, exactly as intended:

We leave it to the reader to draw the architecture for matrix-matrix multiplication (Fig. 7).

A Ripple-Carry Adder. The ripple-carry adder is a standard way to add integer numbers, and is an immediate translation to binary number representations of the usual way in which we add numbers by hand. For example:

$$\begin{array}{r} 1\,1\,0\,1\,0 \\ 1\,0\,1\,0\,1\,1 \\ \hline 1\,0\,0\,0\,1\,0\,1 \end{array}$$

Clearly, we need the elementary logical gates for *and*, *or*, and *xor*:

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & 0 \otimes 0 = 0 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & 0 \otimes 1 = 1 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & 1 \otimes 0 = 1 \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & 1 \otimes 1 = 0 \end{array}$$

We remark that Haskell recognizes Unicode, so the above definitions are valid Haskell definitions.

The most common way to define a ripple-carry adder is by means of a half adder and a full adder, where a half adder takes two input bits, and a full adder additionally also the carry-bit from the right neighbor. In both cases the result is the *pair* of the sum-bit of the input bits and the carry-bit. In Fig. 8 we give the

$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \\ 41 & 42 & 43 \end{pmatrix} \star \begin{pmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{pmatrix} = \begin{pmatrix} 74 & 182 & 290 & 398 \\ 134 & 332 & 530 & 728 \\ 194 & 482 & 770 & 1058 \\ 254 & 632 & 1010 & 1388 \end{pmatrix}$$

Fig. 6. Matrix-matrix product

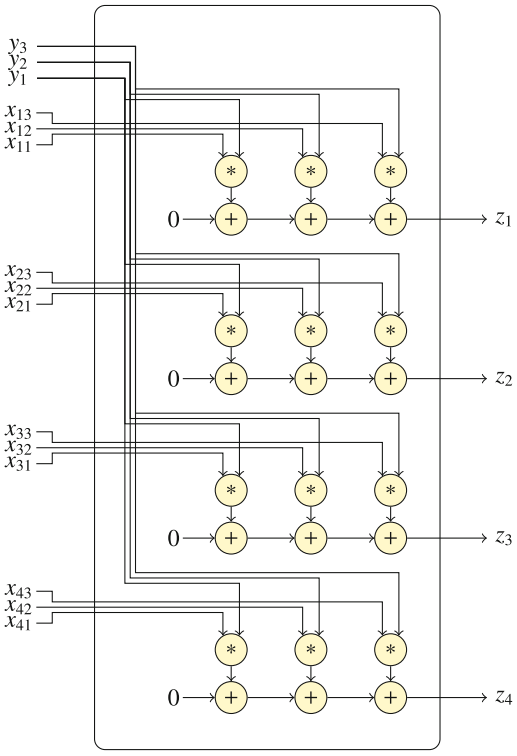


Fig. 7. Architecture for matrix-vector product

x	y	(c, s)
0	0	$(0, 0)$
0	1	$(0, 1)$
1	0	$(0, 1)$
1	1	$(1, 0)$

$$
\begin{aligned}
ha\ (x, y) &= (c, s) \\
&\textbf{where} \\
&\quad c = x \wedge y \\
&\quad s = x \otimes y
\end{aligned}
$$

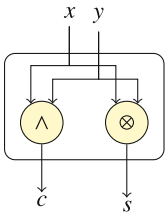
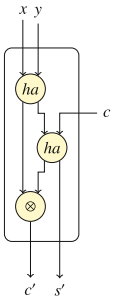


Fig. 8. Half adder

truth table of the half-adder, the Haskell definition which calculates this truth table, and the architecture which is specified by the Haskell definition. In Fig. 9 we do the same for the full adder.

We will say that the Haskell definition of the full adder has *two* arguments (the pair of input bits (x, y), and the carry bit c), and *two* results (the pair of the carry bit c' and the sum bit s). Although this is a somewhat inconsistent formulation since we consider a pair on the input side as *one* value and on the output side as *two* values, we nevertheless choose for that formulation, since it gives us the possibility to connect the full adders using a general *mapAccumL* function:

x	y	c	(c', s)
0	0	0	(0, 0)
0	0	1	(0, 1)
0	1	0	(0, 1)
0	1	1	(1, 0)
1	0	0	(0, 1)
1	0	1	(1, 0)
1	1	0	(1, 0)
1	1	1	(1, 1)



$$
\begin{aligned}
&fa\ \ c\ \ (x, y) = \ (c', s)\\
&\qquad\qquad \textbf{where}\\
&\qquad\qquad c' = (x \wedge y) \vee (x \wedge c) \vee (y \wedge c)\\
&\qquad\qquad s\ \ = x \otimes y \otimes c
\end{aligned}
$$

Fig. 9. Full adder

the function *mapAccumL* can combine a sequence of functions of this structure into a combined function that gets a *list* and a starting value a as arguments. Figure 10 shows the Haskell definition and the corresponding architecture of the function *mapAccumL*. Note that the function *mapAccumL* is a combination of the function *map* and an *accumulation* (from the *left*, hence its name).

```
mapAccumL f a xs = (a', zs)
                   where
                   (as, zs) = unzip (zipWith f (a:as) xs)
                   a'       = last as
```

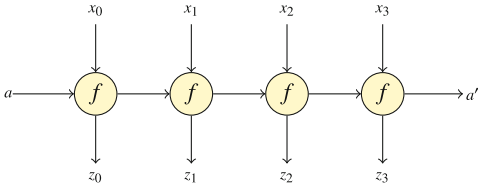


Fig. 10. *mapAccumL*

As a short explanation of the recursive structure in the where clause of the definition of *mapAccumL*, we remark that the list *as* is developed element by element: the first element a_0 is calculated by applying f to a and the first element x_0 of *xs*. Then the second element a_1 is calculated by applying f to a_0 and x_1, and so on. The function *unzip* is needed, since *zipWith* results in a list of pairs, and we need the lists of all first elements *as* and all second elements *zs* of these pairs.

Already now we remark that *mapAccumL f* has the same structure as needed for *f*: it gets *two* arguments (a starting value *a* and a list of inputs *xs*), and it results in *two* values as well (a final result a' and the list of intermediate results *zs*). We will use this fact later, in Sect. 4.2.

In order to define the ripple-carry adder in Haskell, assume that *xs* and *ys* are the bit representations of two integer numbers *x* and *y*, where the first elements of *xs* and *ys* are the *least* significant bits, and the last elements are the *most* significant bits. Assumed is further that *xs* and *ys* are extended with leading zeroes to a given fixed length (say 16 or 32).

Now the ripple carry adder *rca* can be defined by combining full adders *fa* by the function *mapAccumL* with starting value 0 (the initial carry bit) and the list corresponding pairs of bits from *xs* and *ys* as inputs. The result of *mapAccumL* is the pair of the list of intermediate sum bits *ss* and the last carry bit *c*. Clearly, to get the result of the ripple carry adder *rca*, the sum bits *ss* and the last carry bit *c* have to be concatenated.

```
rca xs ys =  ss ++ [c]
        where
        xys     = zip xs ys
        (c, ss) = mapAccumL fa 0 xys
```

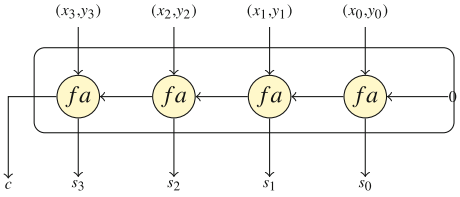


Fig. 11. Ripple-carry adder

As the matrix operations, we remark that in order to let CλaSH generate hardware from the definition of the ripple carry adder, we have to extend the definition of *rca* with an empty state argument (Fig. 11).

An Elementary Multiplier. We conclude the stateless architectures with the definition of an elementary multiplier. Again, we start from the way we would multiply two binary numbers by hand, as in:

```
      1 1 0 1 0
        1 0 1 1 ×
      ----------
      1 1 0 1 0
    1 1 0 1 0
  0 0 0 0 0
1 1 0 1 0       +
-----------------
1 0 0 0 1 1 1 1 0
```

Assume that two 4-bit numbers x and y are given, whose bit sequences are xs and ys where x_0 and y_0 are the least significant bits:

$$\begin{array}{r} x_3\ x_2\ x_1\ x_0 \\ \underline{y_3\ y_2\ y_1\ y_0\ \times} \end{array}$$

An elementary multiplier can be constructed by using the ripple-carry adder as defined in Sect. 4.2 and is as shown in Fig. 12.

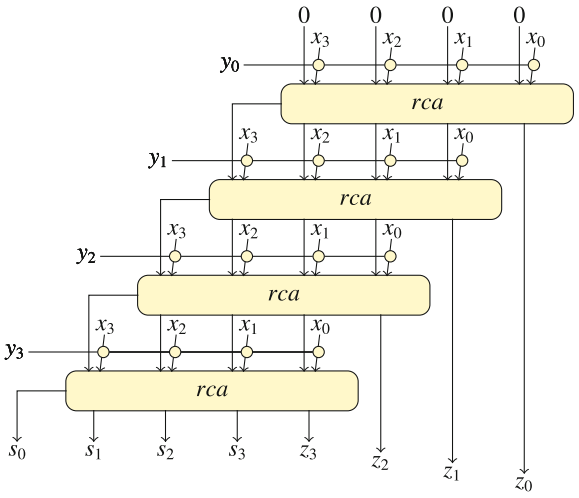


Fig. 12. Elementary multiplier

First note that every horizontal line either consists of the bits from xs or it consists of zeroes only, depending on the question whether the corresponding y-bit is 1 or 0. To calculate this we have to calculate

$$map\ (\wedge y_i)\ xs$$

on every line before the results are given to a ripple-carry adder, which then adds it to the first four bits of the previous line, taking all zeroes at the first line. Note that a ripple-carry adder yields one bit more than the length of the inputs, and the last bit z_i of that result is given to the total result immediately – just as in the case of the calculation by hand. To get the total result, these last bits resulting from all ripple-carry adders have to be concatenated with the first four bits from the last ripple-carry adder.

Before we give the Haskell code for this elementary multiplier, we observe that every line itself again is a function of the form as requested by the *mapAccumL* function:

– every line has *two* inputs: the list ss of the first four sum bits from the previous line (four zeroes on the first line), and the bit y_i indicating whether the bit sequence xs should be added or whether there should be zeroes instead,

– and it has *two* outputs: the first four sum bits going to the next adder, and the last bit z going straight to the final result.

Note that at every line the bit y_i makes the choice whether or not to use the bit sequence xs. That means that the sequence xs can be considered *the same* at every line, i.e., it is a *global* input which is the same at every line. This global pattern is shown in Fig. 13.

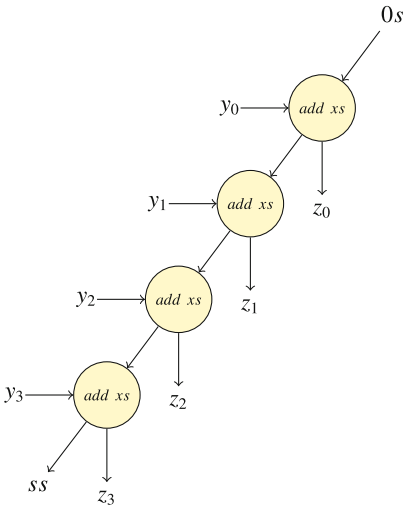


Fig. 13. Multiplier pattern

The Haskell definitions can now be given as follows:

$$
\begin{aligned}
&add\ xs\ ss\ y = \ (ss', z)\\
&\qquad\qquad \textbf{where}\\
&\qquad\qquad z{:}ss' \ = \ rca\ ss\ (map\ (y\wedge)\ xs)
\end{aligned}
$$

The function *add* takes xs as its first argument, meaning that *add xs* is a *function* which takes sum bits ss and a single bit y, and applies the ripple-carry adder to ss and $map\ (\wedge y)\ xs$. Note that the first bit of the result of the function *rca* is the least significant bit, so that is the bit that has to be separated from the rest. This is done by the pattern matching $z{:}ss'$ in the where-clause.

Note also that *add xs* is the actual addition function that is performed at every line, and furthermore, *add xs* answers the pattern as described above. hence, *add xs* can be given to the *mapAccumL* function:

$$
\begin{aligned}
&mul\ xs\ ys = \ zs\ +\!\!+\ ss\\
&\qquad\qquad \textbf{where}\\
&\qquad\qquad zeroes \ = \ replicate\ (length\ xs)\ 0\\
&\qquad\qquad (ss, zs) \ = \ mapAccumL\ (add\ xs)\ zeroes\ ys
\end{aligned}
$$

Here, the function *replicate* produces the initial sequence of zeroes, to start the additions. Clearly, the sum bits coming from the *mapAccumL* function, and the individual z values that were given to the final output now have to be concatenated in order to turn the result into a single number. We mention again that the first bit of this number is the least significant bit.

The reader is invited to test the above definitions in Haskell, and even more so, to experiment with CλaSH to see that these definitions actually can be translated into hardware and, e.g., put on an FPGA.

We conclude with the remark that this elementary multiplier is not very efficient. More efficient, for example, is the Baugh-Whooley multiplier, but we leave it as an exercise to define this multiplier.

4.3 Regular Architectures with State

In this section we return to the dotproduct, but now we assume that there is an ongoing stream of input values and we repeatedly need the dotproduct of an initial part of the input stream with a fixed vector of co-efficients. So, this is a "sliding window" over the input stream, and the computational technique we will discuss is called *convolution*. With the right choice of co-efficients, this technique can be used to filter high or low tones from a music stream, it can be used for video processing, in astronomy, etcetera. In such situations one often speaks of *FIR-filters* (for "Finite Impulse Response" filters). In this section we will discuss the derivation of some variants of FIR-filters, and show their architectures and their specifications in Haskell.

We start with the formula that expresses the convolution function. Let $\boldsymbol{h}$ be a vector of n co-efficients, and let $x_0, x_1, x_2, \ldots, x_t, \ldots$ be a stream of input values, with the index t indicating the moment in time that the value arrives. The FIR-filter determined by the vector $\boldsymbol{h}$ is called an n-tap FIR filter, and its output y_t at time t is defined as

$$y_t \;=\; \sum_{i=0}^{n-1} h_i * x_{t-i} \tag{3}$$

So the FIR-filter calculates at every moment t the dotproduct of the co-efficients $\boldsymbol{h}$ and the last n input values $x_t, \ldots, x_{t-n+1}$. For $n = 4$, Fig. 14 shows three time steps, where the dashed lines follow the values x_i from one time moment to the next (for reasons of space we join multiplication of h_i and x_{t-i}, and addition into one computational component). Note that y_3 is the first correct result of the convolution.

Above, time was introduced with respect to the moments that the input values x_i arrive, but it does not say anything on the *scheduling of the computation* of the results y_i. Even though Fig. 14 suggests that the computation is done on an architecture that consists of (in this case) four computing units doing a multiplication and an addition, it in fact only expresses the dependencies between the computations. Concerning the actual scheduling of the computations, there are many different possibilities, which each give rise to a different architecture.

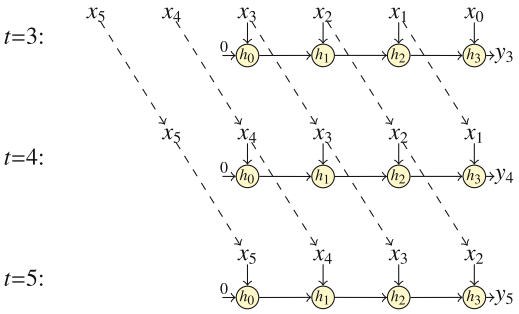


Fig. 14. Convolution on a stream

The remaining parts of this section discuss some of these possible architectures and the way they can be derived from the dependencies expressed in Fig. 14.

FIR-Filter: Variant 1. A straightforward way to schedule the data dependencies from Fig. 14 is to schedule *horizontally*, as indicated by the thick black lines in Fig. 15. All operations between two thick black lines are executed within the same time frame. In the context of this text we will assume that a time frame takes one clock cycle. Hence, data that moves from one time frame to the next has to be remembered, i.e., at every position where a data line crosses a time line, a memory element will be introduced. In Fig. 15 the data lines that cross a time line, are the dashed lines indicating the traversal of the input values x_i. For example, at the end of the first time frame, input x_3 has to be put in a memory element before it will be multiplied by h_1. That is realized by memory element u_1 in the right hand side of Fig. 15. In the same way memory elements u_2 and u_3 can be explained. For memory element u_0 the same reasoning holds, but as will be noted, it would not have been necessary to extend the time line as far to the left as we did. In that case, an input value x_i would be multiplied with h_0 in the same clock cycle as x_i arrives.

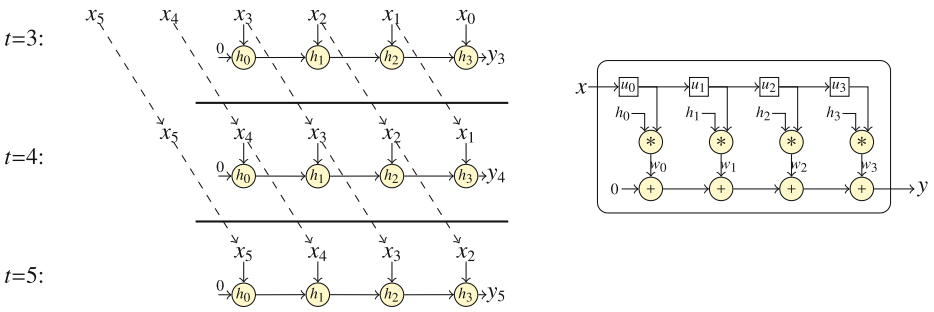


Fig. 15. FIR-filter, variant 1

As can now be seen from Fig. 15, the dot product of the convolution has to be applied to the co-efficients hs and to all values us in the memory elements, i.e., the architecture has to calculate the expression

$$hs \mathbin{.*.} us$$

Besides, the values in the memory elements us all have to be shifted one position to the right, and the next value x_i has to be put in u_0. For this we define the operation $+\!\!>\!\!>$, saying that a value has to be "shifted into" a sequence of memory elements:

$$x +\!\!>\!\!> us = x : init\ us$$

For example:

$$5 +\!\!>\!\!> [1,2,3,4] = [5,1,2,3]$$

Since the co-efficients hs are constant during the operation of the FIR-filter on an input stream, we take those as a parameter to the FIR-filter. Hence, the *first* argument of the FIR-filter consists of the co-efficients hs, the *second* argument is the state us, and the *third* argument is the next input value x. As before, the result consists of the updated state us' and the output value y. That leads to the following definition of the first variant of the FIR-filter:

$$\begin{aligned} fir1\ hs\ us\ x &= (us',\ y) \\ &\textbf{where} \\ us' &= x +\!\!>\!\!> us \\ y &= hs \mathbin{.*.} us \end{aligned}$$

Note that $fir1\ hs$ matches the pattern of an architecture description as required by the function *simulate*, thus $fir1\ hs$ indeed defines an architecture. That coincides with the intuition, that the co-efficients hs are part of the architecture of the FIR-filter.

FIR-Filter: Variant 2. For the second variant of the FIR-filter we choose the time frames as indicated by the thick lines in Fig. 16. Note that now an input value x_i is multiplied with all co-efficients hs within the same time frame, expressed in the right hand side of Fig. 16 by the fact that an input value x is not delayed by a memory element before all multiplications with the co-efficients hs.

The data lines that cross the time lines are now the connections that are between the computational units. Thus, the result of each computational unit has to be put in a memory element before it is given to the next computational units. That is realized by the memory elements vs between the computational units in the right hand side of Fig. 16.

In this variant, the dotproduct operation by itself is not performed within a single time slice, so we cannot use the standard dotproduct function. Instead, we observe that the results ws are pairwise added to the values from the memory elements vs (plus 0 in front). That is to say, the additions correspond to a *zipWith* operation. However, the *zipWith* with + results in a sequence of *four* values, the

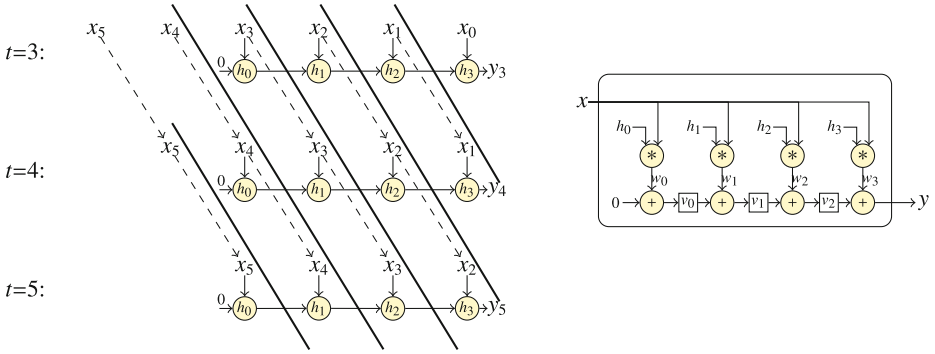


Fig. 16. FIR-filter, variant 2

last of which is the output y, and the initial three are the new content of the memory elements vs. Finally, we remark that the fact that all co-efficients hs are multiplied with the same x-value is expressed by the *map* function.

This gives rise to the following Haskell definition:

$$
\begin{aligned}
&\mathit{fir2}\ \mathit{hs}\ \mathit{vs}\ x\ =\ (\mathit{vs}',\ y)\\
&\qquad\textbf{where}\\
&\qquad\begin{aligned}
\mathit{ws}\ &= \mathit{map}\ (*x)\ \mathit{hs}\\
\mathit{vs}''\ &= \mathit{zipWith}\ (+)\ (0{:}\mathit{vs})\ \mathit{ws}\\
\mathit{vs}'\ &= \mathit{init}\ \mathit{vs}''\\
y\ &= \mathit{last}\ \mathit{vs}''
\end{aligned}
\end{aligned}
$$

We leave it to the reader to check that y indeed is the dot product of four consecutive inputs.

FIR-Filter: Variant 3. In the third variant we choose a different slope of the time lines, and again, we check where the data lines and the time lines cross. Now note that there are *two* crossings in the lines for x_i before it reaches the next computational unit, expressed by two memory elements u_{2i-1} and u_{2i} in the right hand side of Fig. 17. As with variant 2, there again is one memory element v_i between the computational units.

To define this architecture in Haskell, we define an operation to select a sequence of elements (indicated by a list of indexes is) from a list:

$$\mathit{xs}\ !!!\ \mathit{is}\ =\ \mathit{map}\ (\mathit{xs}!!)\ \mathit{is}$$

For example:

$$[2,1,6,4,3]\ !!!\ [0,2,4]\ =\ [2,6,3]$$

Finally, note that the state now consists of two lists us and vs of memory elements. This leads to the following definition:

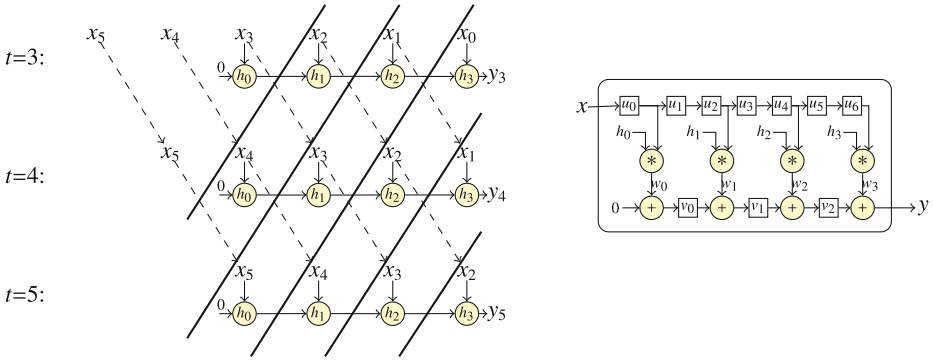


Fig. 17. FIR-filter, variant 3

$$
\begin{aligned}
&\mathit{fir3}\ hs\ (us, vs)\ x \ = \ ((us', vs'),\ y)\\
&\quad \textbf{where}\\
&\quad ws \ = \mathit{zipWith}\ (*)\ hs\ (us!!![0, 2..])\\
&\quad vs'' \ = \mathit{zipWith}\ (+)\ (0{:}vs)\ ws\\
&\quad (us', vs') \ = (x\ +\!\!>\!\!>\ us,\ \mathit{init}\ vs'')\\
&\quad y \ = \mathit{last}\ vs''
\end{aligned}
$$

Again we leave it to the reader to check that indeed this architecture produces the dotproduct of the co-efficients hs and four consecutive values from the input stream. Apart from checking that by hand, one may also run the function *simulate* on the architecture $\mathit{fir3}\ hs$ for a given list hs of co-efficients, and some input stream xs.

FIR-Filter: Variant 4. As a last variant we discuss variant 4, in which the input stream goes from right to left. Furthermore, observe that there are only two computational units in the same time frame. We remark that these units are not consecutive, i.e., either the first and the third, or the second and the fourth computational unit are in the same time frame. The consequence is that not all elements of the input stream will be meaningfully processed, thus the input stream has to be interleaved with arbitrary values. We leave it as an exercise to the reader to check the crossings of the data lines and the time lines, and to connect these to the memory elements in the right hand side of Fig. 18.

We mention that the notation $us\ {<\!\!<\!\!+}x$ means that x is "shifted into" the list us from the right. It is defined as follows:

$$us\ {<\!\!<\!\!+}x \ = \ \mathit{tail}\ us\ +\!\!+\ [x].$$

For example:

$$[1, 2, 3, 4]\ {<\!\!<\!\!+}\ 5 \ = \ [2, 3, 4, 5]$$

Now the Haskell definition should be straightforward:

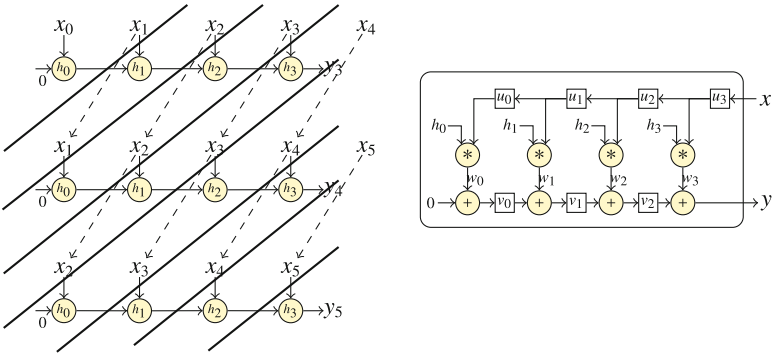


Fig. 18. FIR-filter, variant 4

$$
\begin{aligned}
&\mathit{fir4}\ \mathit{hs}\ (\mathit{us}, \mathit{vs})\ x \quad = \quad ((\mathit{us}', \mathit{vs}'),\ y) \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad \mathit{ws} \qquad\quad = \mathit{zipWith}\ (*)\ \mathit{hs}\ \mathit{us} \\
&\qquad\qquad \mathit{vs}'' \qquad\quad = \mathit{zipWith}\ (+)\ (0{:}\mathit{vs})\ \mathit{ws} \\
&\qquad\qquad (\mathit{us}', \mathit{vs}') = (\mathit{us} \lll\!+\! x,\ \mathit{init}\ \mathit{vs}'') \\
&\qquad\qquad y \qquad\quad = \mathit{last}\ \mathit{vs}''
\end{aligned}
$$

Concluding Remarks. The above architectures are derived by a systematic method, starting from the data dependencies generated by the mathematical formula of the dotproduct of a list of co-efficients and an equally long initial part of the input stream. By varying on the division in time frames, the concrete architectures can be developed by introducing memory elements on the crossings of data lines and time lines.

The major difference between these architectures consists of the number and positioning of memory elements, and may cause some difference in delay of the output and in maximum clock frequency. For example, in variant 1 there is a long combinatorial path, going from the input through the first multiplication, followed by four additions. Clearly, the output is available in the same clock cycle as in which the last input arrived (or very quickly after that), but the consequence of such a long combinatorial path may be that the clock frequency will be low.

In variant 2 the maximal length of the combinatorial paths is much shorter, but there still is the need to deliver the input value to many operations in parallel, taking a lot of energy and possibly a low clock frequency. In variant 3, on the other hand, all combinatorial paths are rather short, so the clock frequency can be high, but there is a longer delay between the last input and the moment that the output becomes available.

Such issues are examples of the considerations which may be relevant which architecture suits a given situation best. This question falls outside the scope of

this text which is mainly aiming at the correspondence between an architecture and its Haskell specification.

We conclude with a possible generalization that is made possible by the high level abstraction mechanisms the Haskell offers: *parameterization*. It is possible to generalize each of the above architectures with the functionality of subcomponents. We will illustrate this for variant 1 of the FIR-filter above. If we abstract away from the concrete functionalities of the subcomponents, and instead turn them into arguments of the architecture, we get a higher level architecture, shown in the following Haskell code:

$$
\begin{aligned}
&\mathit{genfir1}\ (f, g, a, hs)\ us\ x\ =\ (us',\ y)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \textbf{where}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ us' = x +\!\!\gg us\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ ws\ = \mathit{zipWith}\ f\ hs\ us\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ y\ \ \ = \mathit{foldl}\ g\ a\ ws
\end{aligned}
$$

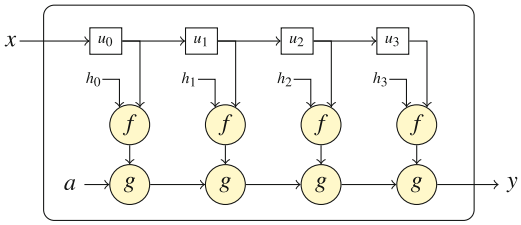


Fig. 19. Parameterized filter

In this code not only the co-efficients *hs* are taken as parameters, but also the functionalities f and g, and the initial value a in the application of *foldl*. The corresponding architecture is shown in Fig. 19.

Note that we can now define

$$\mathit{fir1}\ hs\ us\ x\ =\ \mathit{genfir}\ ((*), (+), 0, hs)\ us\ x$$

It is equally well possible to define a *pattern matcher*, which selects subsequences from an input stream that match a given pattern *hs*:

$$\mathit{pattm}\ hs\ us\ x\ =\ \mathit{genfir}\ ((==), (\&\&), \mathit{True}, hs)\ us\ x$$

This definition leads to the architecture in Fig. 20.

5 Irregular Architectures

In this section we turn to an example of an irregular architecture, the *Sprockell*: a *S*imple *proc*essor in Has*kell* (see Fig. 21). It is an instruction set architecture

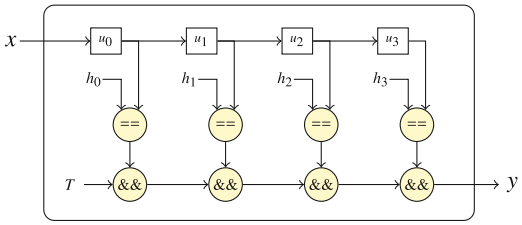


Fig. 20. Pattern matcher

which has many simplifications in comparison with a real processor, for example, we will assume that the execution of an instruction as well as fetching data from memory takes only one clock cycle, there is no pipelining, there are no cache memories, there is no I/O. We assume a program memory that is separated from data memory, and only one program can be executed at the same time. Nevertheless, the architecture together with its instruction set are Turing complete, so it is a non-trivial processor.

The aim of showing it here is to demonstrate the natural character of its specification by means of mathematical functions, which are all executable in Haskell. The irregular character shows itself by the fact that no usage of higher order functions is made, i.e., there is no repeating pattern in the architecture. On the other hand, the way the definitions are given does show a regular pattern, most definitions are just straightforward case-expressions.

5.1 The Sprockell

In Fig. 21 it can be seen that the program memory (pmem) contains a list of instructions (see below for the complete instruction set). The decode function $\mathcal{D}$ decodes these instructions one by one and sends signals onto all its outgoing wires. The formulation "sends signals onto all its outgoing wires" is represented in the definition of the function $\mathcal{D}$ by the fact that the result of $\mathcal{D}$ for every instruction is a record consisting of 13 fields, where every field corresponds to one of the outgoing wires of the decoder.

The Sprockell is a *load-store* architecture, where the load function $\mathcal{L}$ is able to load data from various sources into some register in the register bank $\mathcal{R}$. The sources of these data can be a constant value delivered by the decoder, it can be the output of the alu, or it can be a value from some address in data memory. Which value the load function has to choose, is determined by a special code sent to the load function by the decoder. Clearly, also the address of the register in which the load function has to put the value, is coming from the decoder.

The store function $\mathcal{S}$ saves a value in data memory. As with the load function, this value may come from different sources: it may be a constant sent by the decoder, or it may be a value from some address in the register bank. Here too, the decoder delivers the information which value to choose, which register to read, and which address in data memory to save to.

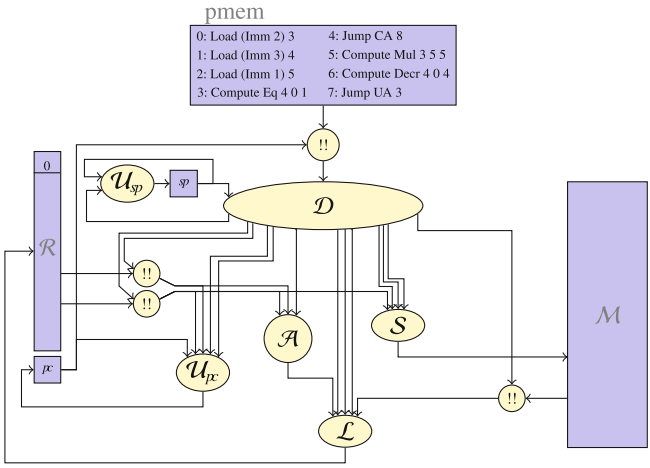


Fig. 21. Sprockell

The alu $\mathcal{A}$ performs an operation, indicated by an opcode, on two values from the register bank, and sends its result to the load function $\mathcal{L}$.

The last elements we mention in this introductory description are the program counter and the stack pointer. As always, the program counter tells which instruction from program memory should be fetched for the decoder (shown in Fig. 21 by the indexing operation !! from Haskell). The program counter is stored in a register which is updated by the function $\mathcal{U}_{pc}$, the program counter update function, based on information from again the decoder. For the stack pointer the same holds: it is stored in a register that is updated by the stack pointer update function $\mathcal{U}_{sp}$.

So, all in all the state of the architecture consists of the register bank $\mathcal{R}$, data memory $\mathcal{M}$, and two registers for the program counter *pc*, and for the stack pointer *sp*. The Sprockell itself is defined as a function which transforms its state every clock cycle, based on the instruction that has to be executed. In the sections below we will formalize the above intuitive descriptions of the various subcomponents and combine them in the definition of the Sprockell as a whole.

We remark that in order to save space and to have some visual recognition based on the names of the components, we choose for a more mathematical formulation. However, this formulation may be readily translated into Haskell in a word for word fashion, by choosing names fro the symbols, such as *alu* for $\mathcal{A}$, *load* for $\mathcal{L}$, *dataMemory* for $\mathcal{M}$, etcetera. Since Haskell recognizes Unicode, one might also choose to leave some of the symbols unchanged, and the result will nevertheless be an executable Haskell program, and simulation can be done with the same function *simulate* as before.

The specification given below is complete in the sense that it can also be mapped onto real hardware, e.g., onto an FPGA. However, in order to give the

code to CλaSH to be translated into synthesizable code, still some mainly minor transformations have to be executed on the Haskell code. We will come back to that issue in Sect. 6.

Memory Structure. As mentioned above, the *state* of the Sprockell consists of the register bank $\mathcal{R}$, the data memory $\mathcal{M}$, and the two registers *pc* and *sp* for the program counter and the stack pointer, respectively. For reasons of simplicity we choose to let all values be integers, and $\mathcal{M}$ and $\mathcal{R}$ be lists of integers:

Register bank:	$\mathcal{R}$	$:: [Int]$
Data memory:	$\mathcal{M}$	$:: [Int]$
Program counter:	pc	$:: Int$
Stack Pointer:	sp	$:: Int$

Note that for real hardware it is not sufficient to choose for integers, nor for lists of integers: for integers one has to choose the number of bits with which the integers will be represented, and also for lists one has to make a choice for the length of the list. We will come back to this in Sect. 6.

To update the register bank or the data memory we define an update operation $<\sim$ to put a value v on position i in a list:

$$\begin{aligned} xs \ <\sim \ (i, v) = \ & ys +\!\!+ [v] +\!\!+ zs \\ & \textbf{where} \\ & (ys, \_ : zs) \ = \ splitAt \ i \ xs \end{aligned}$$

Applying this operation to the register bank or to the data memory has the following limitations:

- register 0 of the register bank always contains the value 0, so putting a value in this register means that the value will be lost,
- before putting a value in the data memory, it has to be enabled for writing.

The Alu $\mathcal{A}$. Concerning the functional components in the *Sprockell*, we start with the alu function $\mathcal{A}$. As can be seen in Fig. 21, the alu has three input signals. Thus, the function $\mathcal{A}$ that specifies the alu has three arguments. The first of these arguments is the opcode *opc* which decides which operation the alu should perform, the other two arguments x and y are the values on which this operation should be performed. The opcodes are defined as an *embedded language*, i.e., as an algebraic data type in Haskell, which can be extended as desired:

$$\textbf{data} \ \ OpCode \ = \ NoOp \mid Id \mid Incr \mid Decr \mid Neg \mid Add \mid Sub \mid Mul \mid Eq \mid Gt \mid \cdots$$

The meaning of these opcodes become clear in the definition the alu function $\mathcal{A}$, which is a simple case-expression, defined by *pattern matching* on the opcode:

$$
\begin{aligned}
\mathcal{A}\ \ opc\ x\ y = & \textbf{case}\ opc\ \textbf{of} \\
& \quad NoOp\ \ -> 0 \\
& \quad Id\ \ \ \ -> x \\
& \quad Incr\ \ -> x + 1 \\
& \quad Decr\ \ -> x - 1 \\
& \quad Neg\ \ \ -> -x \\
& \quad Add\ \ \ -> x + y \\
& \quad Sub\ \ \ -> x - y \\
& \quad Mul\ \ \ -> x * y \\
& \quad Eq\ \ \ \ -> tobit\ (x == y) \\
& \quad Gt\ \ \ \ -> tobit\ (x > y) \\
& \qquad \vdots \\
& \textbf{where} \\
& tobit\ True = 1 \\
& tobit\ False = 0
\end{aligned}
$$

Note that in Haskell the relation ">" results in a boolean, so the function *tobit* is needed to transform this into an integer.

The Load Function $\mathcal{L}$. The *load* function $\mathcal{L}$ has several input values:

- we choose to let the result of the load function $\mathcal{L}$ be the updated register bank as a whole, so also the register bank $\mathcal{R}$ itself is an argument to the load function,
- three values from which the function $\mathcal{L}$ has to choose to put into the register bank: an immediate value c coming from the decoder, a value from data memory d, or the output z from the alu,
- a code *ldc* to tell the load function which value to put in the register bank, or not to load anything at all,
- of course, the register r in which to put the value.

The codes which value to load is defined in an embedded language *LoadCode*:

$$\textbf{data}\ \ LoadCode\ =\ NoLoad \mid LdImm \mid LdAddr \mid LdAlu$$

Now the definition of the load function $\mathcal{L}$ again is a straightforward case-expression, though the case where no value has to be loaded into the register bank is defined in a separate clause:

$$
\begin{aligned}
\mathcal{L}\ \ NoLoad\ \mathcal{R}\ r\ (c,d,z)\ &=\ \mathcal{R} \\
\mathcal{L}\ \ ldc\ \mathcal{R}\ r\ (c,d,z)\ &=\ \mathcal{R} <\sim (r,v) \\
& \quad \textbf{where} \\
& \quad v = \textbf{case}\ ldc\ \textbf{of} \\
& \qquad LdImm -> c \\
& \qquad LdAddr -> d \\
& \qquad LdAlu -> z
\end{aligned}
$$

The Store Function $\mathcal{S}$. The *store* function $\mathcal{S}$ has the following input arguments:

- as with the load function $\mathcal{L}$, we choose to let the result of the store function $\mathcal{S}$ be the updated data memory as a whole, so also the data memory $\mathcal{M}$ itself is an argument to the function $\mathcal{S}$,
- two values from which the function $\mathcal{S}$ has to choose to put into the register bank: an immediate value c coming from the decoder, or a value x from data memory,
- a code *stc* to tell the store function which value to put in the data memory, or not to store anything at all,
- of course, the address a at which to store the value.

The codes which value to store are again defined in an embedded language *StoreCode*:

$$\textbf{data}\ \ \mathit{StoreCode}\ =\ \mathit{NoStore}\ |\ \mathit{StImm}\ |\ \mathit{StReg}$$

Again, the definition of the store function $\mathcal{S}$ is a straightforward case-expression, taking the *NoStore* case as a separate clause leaving the data memory $\mathcal{M}$ unchanged:

$$
\begin{array}{l}
\mathcal{S}\ \ \mathit{NoStore}\ \mathcal{M}\ \ a\ (c,x)\ =\ \mathcal{M} \\
\mathcal{S}\ \ \mathit{stc}\ \mathcal{M}\ \ a\ (c,x)\ =\ \mathcal{M} <\sim (a,v) \\
\qquad\qquad\qquad\qquad\quad \textbf{where} \\
\qquad\qquad\qquad\qquad\qquad v = \textbf{case}\ \mathit{stc}\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{StImm} -> c \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{StReg}\ \ -> x
\end{array}
$$

The Program Counter Update Function $\mathcal{U}_{pc}$. The program counter is updated by the function $\mathcal{U}_{pc}$, based on a jump code to be provided by the decoder. The jump codes are defined in an embedded language *JumpCode*:

$$\textbf{data}\ \mathit{JumpCode}\ =\ \mathit{NoJump}\ |\ \mathit{UA}\ |\ \mathit{UR}\ |\ \mathit{CA}\ |\ \mathit{CR}\ |\ \mathit{Back}$$

The meaning of the jump codes is as follows:

- *NoJump*: just go to the next instruction,
- in *UA*, *UR*, *CA*, *CR* the U/C stand for *Unconditional* and *Conditional*, respectively, i.e., jump in any case, or based on the value x (0 or 1) of a condition. A/R stand for *Absolute* and *Relative*, respectively, i.e., jump to instruction with number n, or jump a n instructions forward (backward in case n is negative) from the current instruction,
- *Back* says that the program counter can jump back to a previously remembered instruction, to be used in case of, e.g., return from a subroutine.

The program counter update function now again is straightforwardly defined by a case-expression (i*pc* is the program counter, *jmpc* the program counter code, y the previously stored program counter):

$$
\begin{aligned}
\mathcal{U}_{pc}\ (jmpc, x)\ (n, y)\ pc\ = & \textbf{case}\ jmpc\ \textbf{of} \\
& \quad NoJump \quad -> pc{+}1 \\
& \quad UA \quad -> n \\
& \quad UR \quad -> pc{+}n \\
& \quad CA \quad \mid x{==}1 \quad -> n \\
& \qquad\quad \mid \textbf{otherwise} -> pc{+}1 \\
& \quad CR \quad \mid x{==}1 \quad -> pc{+}n \\
& \qquad\quad \mid \textbf{otherwise} -> pc{+}1 \\
& \quad Back \quad -> y
\end{aligned}
$$

The Stack Pointer Update Function $\mathcal{U}_{sp}$. The stack is a dedicated sequence of memory locations in the data memory, starting at a freely to determine memory address. The idea of defining the stack pointer update function should be clear by now, and we give the definitions straight away. The stack pointer update code:

$$\textbf{data}\ \ SPCode\ =\ Up \mid Down \mid None$$

The stack pointer update function, where *sp* is the stack pointer, and *spc* the stack pointer code:

$$
\begin{aligned}
\mathcal{U}_{sp}\ spc\ sp\ = & \textbf{case}\ spc\ \textbf{of} \\
& \quad Up \quad -> sp{+}1 \\
& \quad Down \ -> sp{-}1 \\
& \quad None \ -> sp
\end{aligned}
$$

The Instruction Set. Also the *instruction set* is defined as an embedded language, called *Assembly*:

$$
\begin{aligned}
\textbf{data}\ \ Assembly = & \textbf{Compute}\ \ OpCode\ Int\ \ Int\ \ Int \\
\mid & \textbf{Jump}\ \ JumpCode\ \ Int \\
\mid & \textbf{Load}\ \ Value\ \ Int \\
\mid & \textbf{Store}\ \ Value\ \ Int \\
\mid & \textbf{Push}\ \ Int \\
\mid & \textbf{Pop}\ \ Int
\end{aligned}
$$

The type *Value* consists of two sorts of values: immediate values (constants) and values indicated by their address in data memory. It is defined as follows:

$$
\begin{aligned}
\textbf{data}\ \ Value = & Addr\ Int \\
\mid & Imm\ Int
\end{aligned}
$$

The following table describes the meaning of the instructions:

Compute *opc* i_0 i_1 i_2*:* the alu will perform the operation *opc* on the values from registers i_0 and i_1, and the result will be put in register i_2,

Jump *jmpc n:* the program counter will be changed by the number n, based on the jump code *jmpc*,

Load (*Imm n*) *j:* the value n will be loaded into register j,

Load (*Addr i*) *j:* the value from address i in data memory will be loaded into register j,
Store (*Imm n*) *j:* the constant n will be stored in data memory at address j,
Store (*Addr i*) *j:* the value from register i will be stored in data memory at address j,
Push *i:* the value from register i will be pushed onto the stack,
Pop *i:* the top value of the stack ill be loaded into register i.

The program memory is a list of assembly instructions, i.e., the program memory has type [*Assembly*].

The Decode Function $\mathcal{D}$. The *decode* function $\mathcal{D}$ translates an instruction into signals for all other functions in the Sprockell. That is to say, the function $\mathcal{D}$ gets two arguments: the stack pointer *sp* and an assembly instruction α, and produces a record consisting of 13 fields, as shown in Fig. 21 This record type represents the "machine code" and is defined as:

$$
\begin{aligned}
\textbf{data}\ \mathit{MachCode} = \mathit{MachCode}\ \{\ & \mathit{ldCode} :: \mathit{LoadCode},\\
& \mathit{stCode} :: \mathit{StoreCode},\\
& \mathit{opCode} :: \mathit{OpCode},\\
& \mathit{jmpCode} :: \mathit{JumpCode},\\
& \mathit{spCode} :: \mathit{SPCode},\\
& \mathit{jmpN} :: \mathit{Int},\\
& \mathit{immvalR} :: \mathit{Int},\\
& \mathit{immvalS} :: \mathit{Int},\\
& \mathit{reg0} :: \mathit{Int},\\
& \mathit{reg1} :: \mathit{Int},\\
& \mathit{addr} :: \mathit{Int},\\
& \mathit{toreg} :: \mathit{Int},\\
& \mathit{toaddr} :: \mathit{Int}\ \}
\end{aligned}
$$

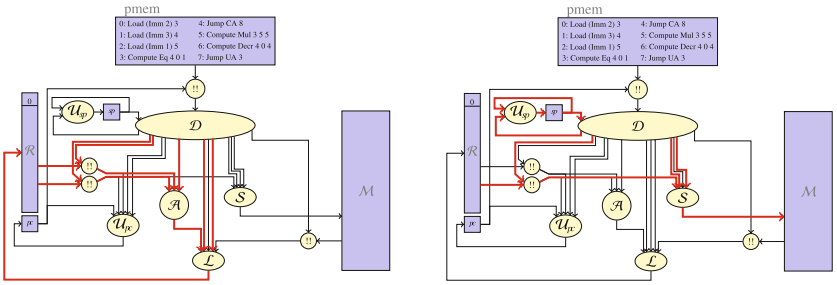
We define an empty record for the machine code $\mathbf{C_0}$:

$$
\begin{aligned}
\mathbf{C_0} = \mathit{MachCode}\ \{\ & \mathit{ldCode}{=}\mathit{NoLoad},\ \mathit{stCode}{=}\mathit{NoStore},\ \mathit{opCode}{=}\mathit{NoOp},\\
& \mathit{jmpCode}{=}\mathit{NoJump},\ \mathit{spCode}{=}\mathit{None},\ \mathit{jmpN}{=}0,\\
& \mathit{immvalR}{=}0,\ \mathit{immvalS}{=}0,\\
& \mathit{reg0}{=}0,\ \mathit{reg1}{=}0,\ \mathit{addr}{=}0,\ \mathit{toreg}{=}0,\ \mathit{toaddr}{=}0\ \}
\end{aligned}
$$

The function $\mathcal{D}$ now is defined by updating the empty machine code $\mathbf{C_0}$ for every instruction separately, by using a case-expression. Note that the fact that the instruction set is defined as an embedded language, offers the possibility of pattern matching on each instruction:

$\mathcal{D}\ sp\ \alpha\ =\ \textbf{case}\ \alpha\ \textbf{of}$

Compute *opc* $i_0\ i_1\ i_2$	$\to \mathbf{C_0}\ \{\ \mathit{ldCode}{=}\mathit{LdAlu},\ \mathit{opCode}{=}\mathit{opc},\ \mathit{reg0}{=}i_0,\ \mathit{reg1}{=}i_1,\ \mathit{toreg}{=}i_2\ \}$
Jump *jc n*	$\to \mathbf{C_0}\ \{\ \mathit{jmpCode}{=}\mathit{jc},\ \mathit{jmpN}{=}n,\ \mathit{reg0}{=}1,\ \mathit{reg1}{=}6\ \}$
Load (*Imm n*) *j*	$\to \mathbf{C_0}\ \{\ \mathit{ldCode}{=}\mathit{LdImm},\ \mathit{immvalR}{=}n,\ \mathit{toreg}{=}j\ \}$
Load (*Addr i*) *j*	$\to \mathbf{C_0}\ \{\ \mathit{ldCode}{=}\mathit{LdAddr},\ \mathit{addr}{=}i,\ \mathit{toreg}{=}j\ \}$
Store (*Imm n*) *j*	$\to \mathbf{C_0}\ \{\ \mathit{stCode}{=}\mathit{StImm},\ \mathit{immvalS}{=}n,\ \mathit{toaddr}{=}j\ \}$
Store (*Addr i*) *j*	$\to \mathbf{C_0}\ \{\ \mathit{stCode}{=}\mathit{StReg},\ \mathit{reg0}{=}i,\ \mathit{toaddr}{=}j\ \}$
Push *i*	$\to \mathbf{C_0}\ \{\ \mathit{stCode}{=}\mathit{StReg},\ \mathit{spCode}{=}\mathit{Up},\ \mathit{reg0}{=}i,\ \mathit{toaddr}{=}sp{+}1\ \}$
Pop *i*	$\to \mathbf{C_0}\ \{\ \mathit{ldCode}{=}\mathit{LdAddr},\ \mathit{spCode}{=}\mathit{Down},\ \mathit{addr}{=}sp,\ \mathit{toreg}{=}i\ \}$



(a) *Compute instruction* (b) *Push instruction*

Fig. 22. Examples of the effect of instructions (color figure online)

In order to illustrate the definition of the decoder, we give two examples. In Fig. 22(a) it is shown which extra signals (marked with red) in comparison to the empty machine code are activated by the decode function $\mathcal{D}$ to execute the *compute* instruction. From the corresponding clause in the definition of $\mathcal{D}$ we derive that these extra signals are:

- two register addresses by which the values for the alu $\mathcal{A}$ are selected,
- the opcode signal directly to the alu $\mathcal{A}$,
- two signals to the load function $\mathcal{L}$, saying that the outcome z of $\mathcal{A}$ has to be put in the register bank, and to which register that value has to be put.

Likewise, Fig. 22(b)can be compared to the clause in the decode function $\mathcal{D}$ to see that the following signals are added to the empty machine code for the *push* instruction:

- the value from register i has to be selected,
- the store function $\mathcal{S}$ should know that the value x from the register bank has to be put in data memory $\mathcal{M}$, and that it has to be stored on top of the stack, i.e., at address $sp{+}1$,
- since an element is put on top of the stack, the stack pointer has to be increased by one, such that the stack pointer again points to the top element of the stack.

We leave it to the reader to check the decoding of the other instructions.

The Sprockell Function. Finally we come to the function *sprockell*, in which all the above defined functions are composed together. We first remark that the function *sprockell* is of the pattern as described by a Mealy Machine (see Sect. 3):

- it is parameterized with a sequence αs of instructions in the program memory,
- its state $(\mathcal{R}, \mathcal{M}, pc, sp)$ consists of the register bank, the data memory, and the program counter and stack pointer,
- the input is irrelevant, since for these lecture notes we chose to leave the processor without I/O. The input may be interpreted as a clock tick,
- the result consists of the updated state and some output, which can be freely defined, e.g., as a specific memory element to follow the changes.

$$
\begin{array}{lll}
sprockell\ \ \alpha s\ \ (\mathcal{R},\mathcal{M},pc,sp)\ \ \_ = & ((\mathcal{R}',\mathcal{M}',pc',sp'),\ \ out) & \\
& \textbf{where} & \\
& MachCode\{..\} = & decode\ sp\ (\alpha s!!pc) \\
& \mathcal{R}^{+} & = \mathcal{R} +\!\!+ [pc] \\
& (x,y) & = (\mathcal{R}^{+}!!reg0,\ \mathcal{R}^{+}!!reg1) \\
& z & = \mathcal{A}\ \ opCode\ x\ y \\
& d & = \mathcal{M}!!addr \\
& \mathcal{R}' & = \mathcal{L}\ \ ldCode\ \mathcal{R}\ toreg\ (immvalR,d,z) \\
& \mathcal{M}' & = \mathcal{S}\ \ stCode\ \mathcal{M}\ toaddr\ (immvalS,x) \\
& pc' & = \mathcal{U}_{pc}\ (jmpCode,x)\ (jmpN,y)\ pc \\
& sp' & = \mathcal{U}_{sp}\ \ spCode\ sp \\
& & \\
& out & = \cdots
\end{array}
$$

Note that the first line of the where-clause says that we may use the field names of the machine code record as if they were normal variables. The next line defines an "extended register bank" such that we can also choose the value of the program counter by indexing this extended register. That is practical in case a value of the program counter is saved on the stack in case of subroutine calls.

The variables x and y are defined as the values from the register bank at addresses $reg0$ and $reg1$, which come from the machine code vector, i.e., they are chosen by the decoder. The variable z results from applying the alu $\mathcal{A}$ to these values x and y, and applying the operation indicated by $opCode$, again afield from the machine code record. Likewise, d is the value from the data memory $\mathcal{M}$.

In the last four lines the various parts of the state are updated by applying the corresponding update functions to their arguments.

Simulation. The Sprockell can now be simulated by choosing an appropriate sequence $\alpha\, s$of instructions, and appropriate values for the initial register bank and data memory. Clearly, the expected values to fill register bank and data memory are zeroes. The program counter should start at 0, and the stack pointer at that value that indicates the address in data memory where the stack starts. Now the processor may be simulated by the following expression:

$$simulate\ \ (sprockell\ \alpha s)\ \ (\mathcal{R}_0,\mathcal{M}_0,pc_0,sp_0)\ \ [0..]$$

The list of instructions in the program memory in Fig. 21 calculates the value of 2^3. It puts 2 in register 3, 3 in register 4, and puts the result in register 5. If we define *out* above as

$$(pc,\mathcal{R}!!1,\mathcal{R}!!3,\mathcal{R}!!4,\mathcal{R}!!5)$$

then the simulation gives the following sequence of 5-tuples:

$$
\begin{array}{l}
[(0,0,0,0,0),\ (1,0,2,0,0),\ (2,0,2,3,0),\ (3,0,2,3,1), \\
\ (4,0,2,3,1),\ (5,0,2,3,1),\ (6,0,2,3,2),\ (7,0,2,2,2),\ (3,0,2,2,2), \\
\ (4,0,2,2,2),\ (5,0,2,2,2),\ (6,0,2,2,4),\ (7,0,2,1,4),\ (3,0,2,1,4), \\
\ (4,0,2,1,4),\ (5,0,2,1,4),\ (6,0,2,1,8),\ (7,0,2,0,8),\ (3,0,2,0,8), \\
\ (4,1,2,0,8),\ (8,1,2,0,8),\ (*** \text{Exception}:\ \text{Prelude.}(!!):\ \text{index too large}
\end{array}
$$

The first line contains the initialization of the values 2, 3, 1 in the registers 3, 4, 5 (respectively), and the other lines all start with the result of instruction 3 which computes whether register 4 equals zero. Note that the values in the registers are the values *before* the instruction indicated by the program counter (on the first position each 5-tuple) is executed.

Note also that instruction 3 puts the result in register 1, since that is the register where the conditional jump looks to decide whether it should jump or not (as determined by the choice *reg*0=1 in the definition of the decode function for the jump instruction).

Finally, note that the simulation ends by an "index too large" error, since instruction 4 will cause that the program counter gets the value 8, whereas the largest index of the sequence is 7. Clearly, that is not the most elegant solution, but in the framework of these lecture notes, we don't elaborate this point any further.

Concluding Remarks. Above we described a non-trivial processor in order to show the naturality by which the components and the total processor can be specified and simulated using Haskell. A further step would be to define a programming language for the Sprockell, which can also be done by embedded languages, a simplified example being:

$$
\begin{aligned}
&\textbf{type } \textit{Variable} &&= \textit{String}\\
&\textbf{data } \textit{Expression} &&= \cdots\\
&\textbf{data } \textit{Program} &&= \textbf{Program } [\textit{Statement}]\\
&\textbf{data } \textit{Statement} &&= \textbf{Assign } \textit{Variable Expression}\\
&&&\mid \textbf{If } \textit{Expression } [\textit{Statement}]\ [\textit{Statement}]\\
&&&\mid \textbf{While } \textit{Expression } [\textit{Statement}]
\end{aligned}
$$

We leave it to the reader to work out the details, including the definition of a *compiler*, which now can be defined as a function from these types to a list of instructions, i.e., to the type $[Assembly]$. Clearly, the compiler also needs a lookup table in which it is registered on which memory location the value of a variable is put.

5.2 Composition of Stateful Components

In the previous section we described the Sprockell processor as an example of an irregular architecture. All subcomponents of the Sprockell are stateless, which makes the composition of these subcomponents straightforward, as can be seen in de definition of the function *sprockell*. In this section we will discuss an example of an irregular architecture which is a composition of stateful subcomponents. The example we choose for that is a reduction circuit as described in [7].

The issue with the composition of subcomponents with state is that the fact that the state is an explicit argument and an explicit result of an architecture definition causes that also the component that contains these subcomponents must have the state of these subcomponents as an argument. The reason is

that each clock cycle the resulting state of a component has to be fed back to the same component as an argument. The consequence is that all states of all subcomponents — and of subcomponents of subcomponents, etcetera — are arguments and results of the top-level architecture. Because of the negative effect of this on the readability of an architecture specification, we wish to hide the state of subcomponents and suppress the visibility of state on a higher level then the subcomponent to which the state belongs.

The Haskell feature that we use for this is called *arrows*. We will only show the usage of arrows in the example below, for a deeper understanding of the concept we refer to the Haskell website (www.haskell.org) where several introductions to the concept can be found.

The Reduction Circuit. The intention of the reduction circuit presented here is to add — on an FPGA — sequences of numbers which enter in order, for example:

$$a_1, \ldots, a_k,\ b_1, \ldots, b_m,\ c_1, \ldots, c_n,\ \ldots$$

Thus, all numbers a_i have to be added, all numbers b_i have to be added, etcetera.

There are a few aspects that have to be taken into account:

- every number is marked with the row to which it belongs, but all numbers belonging to the same row arrive consecutively,
- every number is a floating point number, meaning that addition is a pipeline and takes several clock cycles,
- every clock cycle a number arrives and has to be processed immediately.

Clearly, the combination of the last two points make this a tricky problem, and many architectures are published to do the reduction efficiently. The architecture we will present uses the possibilities of the pipelined adder to process several additions in parallel such that all additions can be executed streamingly. The global idea of the architecture is shown in Fig. 23:

- there is a pipelined floating point adder receiving two numbers at a time, which then travel through the adder upwards until at the top they are completely added. Meanwhile the adder may receive new numbers, possibly belonging to a different row. In the figure the adder is working on four additions, two belonging to row a, and two belonging to row b.
- when the adder finished adding two numbers, the result is put in the partial result memory on a location reserved to the row to which this result belongs. In the figure this is row a, whereas an intermediate result of row b is stored on another location. One clock cycle later, the adder will produce a next intermediate result of row b, and together with the partial b-result from memory, that will be sent to the adder.
- there is an input buffer (a FIFO buffer) where the numbers are received in-order, one-by-one. From this input buffer, the numbers are sent to the pipelined floating point adder, either two at the same time (as shown in the Fig. 23), or one together with a result from the adder belonging to the same row.

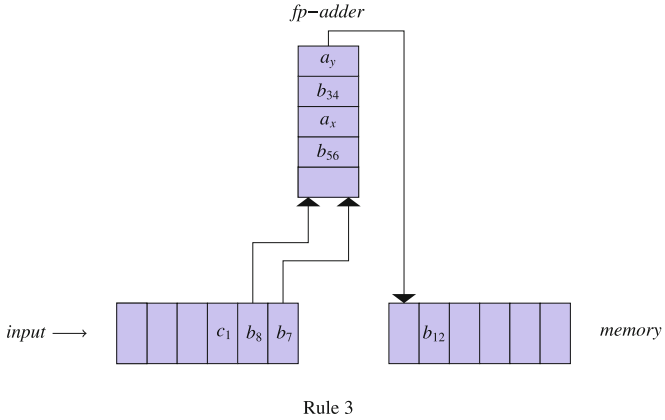


Fig. 23. Reduction circuit, schematic

There are five rules concerning the priority of the number combinations to be sent to the adder:

1. a number from the adder together with a previous result of the same row in memory,
2. a number from the adder together with the first number from the input buffer if it belongs to the same row,
3. the first two numbers from the input buffer if they belong to the same row,
4. the first number from the input buffer if it is the last of a row, together with 0,
5. no number at all if none of the above rules apply.

We refer to [7] for a more extensive description of the algorithm and for a proof that no pipeline stalls and no buffer overflows occur.

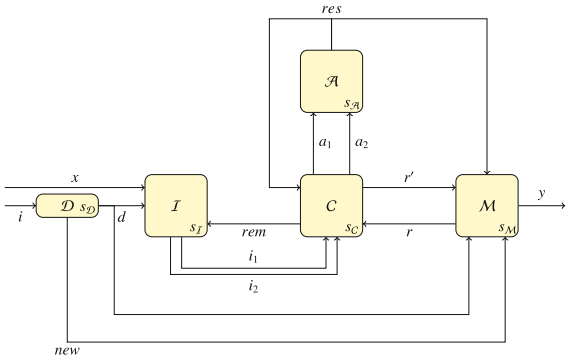


Fig. 24. Reduction circuit, architecture

In Fig. 24 the components $\mathcal{I}$, $\mathcal{A}$, $\mathcal{M}$ correspond to the input buffer, the pipelined adder, and the intermediate result memory, respectively. In addition to these components there are two more components:

- a *discriminator* $\mathcal{D}$ which adds a marker to each number for the row it belongs to. Note that this information is also sent to the partial result memory to make a reservation for a location for the intermediate results of a row. When the end result of a row is completely calculated, the corresponding marker can be re-used for the numbers of a later row.
- a *controller,* $\mathcal{C}$ which checks the above rules and decides which combination of numbers to send to the adder, and which informs the other components how this choice influences the content of these components.

Each component has its own internal state, called $\mathcal{S}_{\mathcal{D}}$, $\mathcal{S}_{\mathcal{I}}$, etcetera. As an example, we mention that the input buffer $\mathcal{I}$ receives every clock cycle a number x together with its marker d. It sends its first two numbers i_1 and i_2 (or an *undefined* value in case there is only zero or one cell of the input buffer filled) to the controller and receives in return (in the same clock cycle) the number *rem* telling whether there were 0, 1, or 2 of the values i_1 and i_2 used and which have thus to be removed from the state of the input buffer.

Without going into the internal details of the other components, we remark that they all are defined according to the pattern of a Mealy Machine, i.e., they have the form

$$f\ state\ input\ =\ (state',\ output)$$

Now the state of the reduction circuit $\mathcal{RC}$ as a whole is the combination of the states of all its subcomponents, i.e.,

$$s_{\mathcal{RC}}\ =\ (\ s_{\mathcal{D}},\ s_{\mathcal{I}},\ s_{\mathcal{A}},\ s_{\mathcal{C}},\ s_{\mathcal{M}}\ )$$

The reducer as a whole is a composition of the nested states and can be defined as follows:

$$\begin{array}{l} reducer\ s_{\mathcal{RC}}\ (x,i)\ =\ (s'_{\mathcal{RC}},\ y) \\ \textbf{where} \\ (\ s_{\mathcal{D}},\ s_{\mathcal{I}},\ s_{\mathcal{A}},\ s_{\mathcal{C}},\ s_{\mathcal{M}}\ )\ =\ s_{\mathcal{RC}} \\ \begin{array}{lcl} (s'_{\mathcal{D}},(new,d)) & = & \mathcal{D}\ s_{\mathcal{D}}\ i \\ (s'_{\mathcal{I}},(i_1,i_2)) & = & \mathcal{I}\ s_{\mathcal{I}}\ (x,d,rem) \\ (s'_{\mathcal{P}},res) & = & \mathcal{P}\ s_{\mathcal{P}}\ (a_1,a_2) \\ (s'_{\mathcal{R}},(r,y)) & = & \mathcal{R}\ s_{\mathcal{R}}\ (new,d,res,r') \\ (s'_{\mathcal{C}},(a_1,a_2,rem,r')) & = & \mathcal{C}\ s_{\mathcal{C}}\ (i_1,i_2,res,r) \end{array} \\ s'_{\mathcal{RC}}\ =\ (\ s'_{\mathcal{D}},\ s'_{\mathcal{I}},\ s'_{\mathcal{A}},\ s'_{\mathcal{C}},\ s'_{\mathcal{M}}\ ) \end{array}$$

Note that the total state of the reduction circuit first has to be unpacked in the 5-tuple of the states of its subcomponents, after which every individual subcomponent is applied to its own state and the corresponding inputs (we leave it to the reader to check these inputs with Fig. 24). The outcome of the application of each subcomponent is a tuple of its updated state, and its outputs,

after which the updated state of the reduction circuit as a whole again is the 5-tuple of the internal states of the various subcomponents.

Though straightforward, this is a cumbersome notation, the technique that CλaSH uses to avoid it is by means of Haskell's *arrow* abstraction mechanism, written as follows (the input of the circuit consists of (x, i), and the output of y, the total of a row):

$$
\begin{array}{l}
reducer \;=\; \textbf{proc}\ (x, i)\ \texttt{-}\texttt{>}\ \textbf{do rec} \\
\quad (new, d) \qquad\quad \texttt{<}\texttt{-}\ (\, D \hat{}\hat{}\hat{}\, s_D^0\,)\ \texttt{-}\texttt{<}\ i \\
\quad (i_1, i_2) \qquad\quad \texttt{<}\texttt{-}\ (\, I \hat{}\hat{}\hat{}\, s_I^0\,)\ \texttt{-}\texttt{<}\ (x, d, rem) \\
\quad res \qquad\quad \texttt{<}\texttt{-}\ (\, P \hat{}\hat{}\hat{}\, s_P^0\,)\ \texttt{-}\texttt{<}\ (a_1, a_2) \\
\quad (r, y) \qquad\quad \texttt{<}\texttt{-}\ (\, R \hat{}\hat{}\hat{}\, s_R^0\,)\ \texttt{-}\texttt{<}\ (new, d, res, r') \\
\quad (a_1, a_2, rem, r') \texttt{<}\texttt{-}\ (\, C \hat{}\hat{}\hat{}\, s_C^0\,)\ \texttt{-}\texttt{<}\ (i_1, i_2, res, r) \\
\textbf{returnA}\ \texttt{-}\texttt{<}\ y
\end{array}
$$

The internal state of each component is now maintained by the arrow mechanism, where the notation $\hat{}\hat{}\hat{}$ instantiates a component with an adequately defined initial component s_x^0. The comparison of this specification with Fig. 24 shows an immediate correspondence between specification and figure.

The totally worked out code of the reduction circuit can be found on the CλaSH website, clash.ewi.utwente.nl.

6 CλaSH

In the previous sections we gave several examples of architectures using Haskell as a specification language, illustrating several aspects of such specifications. We showed that Haskell has many powerful features which are very suitable for the description of hardware architectures. First of all, the mathematical perspective of the language suits the concept of transforming a signal by means of a digital circuit, since that concept is close to th e concept of a function. But also several more concrete features of Haskell are very powerful, for example, polymorphism turned out to be a very pleasant feature when it comes to a first structural design, as well as the possibility of higher order functions in case of regular architectures. Furthermore, the flexibility in choice constructs, the possibility of exploiting embedded languages, and the derivation of types are practical. Finally, we mention the immediate possibility of simulating a design as a very practical feature.

However, in order to produce real hardware from these specifications, for example on an FPGA, we still have to modify the Haskell code in order to make it suitable for processing by CλaSH. Since every CλaSH specification also is an executable Haskell program, these modifications boil down to some rather standard adaptations. In Sect. 6.1 we will describe some of these steps. In Sect. 6.2 we will sketch the processing pipeline of CλaSH, and give an informal idea of the rewrite mechanism that CλaSH performs in order to produce synthesizable

code. Finally, in Sect. 6.3 we will mention some further issues where CλaSH still has to be improved.

6.1 Transforming Haskell Code into CλaSH Code

The modifications of Haskell code into code that can be processed by CλaSH, can be distinguished in three different issues, discussed below:

- Bringing the Haskell code into a specific form that can be dealt with during CλaSH simulation and translation,
- Issues concerning types, in particular number types and list types,
- Issues that have to do with typical hardware deliberations, such as fixed point arithmetic versus floating point arithmetic.

CλaSH Syntactical Form

Types. Several types that are natural in Haskell have to be modified in order to be usable for hardware. The limitation stems from the fact that on hardware one has to choose explicitly how many wires to use, for example, the designer has to decide on the bit width of the involved number types. Besides, in order to use the available area on an FPGA optimally, a designer will often choose for a non-standard bit width of integers, such as 18 bit integers.

Number Types. CλaSH offers several typing constructs to express these choices, the most important ones being *Signed* and *Unsigned* for integer numbers. In addition the bit width of these numbers has to be indicated, for example *Signed*16 for 16 bit signed numbers.

List Types. The same holds for lists: in Haskell a list may vary in length during the evaluation of a program. On hardware, however, that is not possible, so the designer has to make a choice for the length of a "list". For this, CλaSH offers *vector* types, of the following pattern: *Vector* ⟨*width*⟩ ⟨*type*⟩, where the width should be, e.g., 16, 24, etcetera, and the type may be any type that is acceptable on hardware.

Polymorphism. One further point concerning this issue is polymorphism: often a specification in Haskell holds for many different types, for example, the specification of FIR-filters in Sect. 4.3 hold for any number type, they hold for *Int*, *Integer*, *Float*, etcetera, alike. As mentioned above, for hardware a choice has to be made, but it often is sufficient to make this choice at the top level of the specification. For many types of subexpressions of the specification, the types will be derived by the compiler.

Algebraic Types. As we saw in Sect. 5, a very powerful usage of algebraic types is to define *embedded languages*. CλaSH is able to translate these types into bit patterns, which can be mapped onto hardware. These bit patterns are efficient in the sense that parts of the bit pattern can be re-used for other constructors from the same algebraic type.

Further Transformations. The above transformations stem from the need to adapt Haskell code to specific hardware requirements, such as decisions on bit widths of number types. Some other transformations that may have to be necessary have to do with design choices concerning the performance and the precision of the designed hardware. A typical example of such a choice concerns the choice for floating point or fixed point arithmetic, which has to do with a trade off between the usage of area and time on the actual hardware. In Haskell everything is done in floating point, so if the design has to use fixed point arithmetic, the Haskell code has to be adapted correspondingly.

A comparable issue arises with some arithmetical operators which may be complex in hardware, such as division. In Haskell itself, which is evaluated as software, the designer does not need to think about such issues. But in order to avoid the complexity in hardware, and possibly slow execution of such operators, a designer may choose for an approximation of such an operator.

6.2 The Processing Pipeline of CλaSH

The process performed by CλaSH is a pipeline and consists of several stages (see Fig. 25):

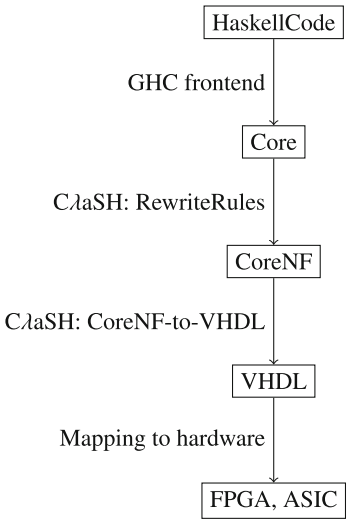


Fig. 25. CλaSH pipeline

GHC Frontend. The first step is done by the standard Haskell compiler *GHC*, or by *GHCi*, the interactive variant of *GHC*. *GHC* takes care of aspects such as syntax analysis, desugaring, parsing, and type checking. Also type derivation is taken care of by *GHC*.

Besides, *GHC* translates the CλaSH specification into the *Core* language, a *GHC* internal language which is a fully fledged functional language, but has

only a limited number of syntactical constructions. That is to say, the result of *GHCI* is a *Core* expression which is equivalent to the original specification, but which is easier to deal with because of its greater syntactical simplicity.

CλaSH Rewrite Rules. This phase in the CλaSH pipeline is the first step of the CλaSH kernel itself: to rewrite a specification into a *normal form* (CoreNF) which makes the hardware architecture explicit in detail. In fact, this CλaSH normal form is close to a so-called *netlist* formalism, which is used in techniques to produce actual hardware from a specification. Informally put, a netlist formalism describes a graph in which every wire is mentioned. The CλaSH normal form has the following structure:

$$
\begin{array}{l}
\lambda \boldsymbol{x}.\ \textbf{let} \\
\quad \begin{array}{rcl} y_0 & = & e_0 \\ y_1 & = & e_1 \\ y_2 & = & e_2 \\ & \vdots & \end{array} \\
\textbf{in} \\
\quad z
\end{array}
$$

Thus, the CλaSH normal form is a lambda expression with zero or more formal parameters which correspond to the inputs of the specified component. The body is a **let** expression with a sequence of local definitions, in which every defining expression e_i is a simple expression, i.e., an expression with only variables as subexpressions. Every variable defined in this let-expression corresponds to a wire in the actual hardware. Finally, the **in** part of the let-expression also is a single variable, which corresponds to the output of the component. In fact, every wire in the CλaSH normal form has a name, and thus the CλaSH normal form is close to a netlist format.

Below we will give an informal example of this part of the pipeline to show that the normal form indeed is close to the hardware architecture.

CoreNF to VHDL. The second step of the CλaSH kernel is the translation the CλaSH normal form into VHDL. The reason to choose for VHDL as a target language is that VHDL is a standardized hardware specification language, and many tools exist that map VHDL specifications to actual hardware, such as an FPGA. In fact, the expression in CλaSH normal form already is in a structural sense already very close to VHDL.

Mapping to Hardware. This is the last phase in the pipeline and consists of the usage of the tools that are available for VHDL to take care of the actual mapping of the specification onto hardware. For example, the synthesis of an FPGA is realized by these VHDL tools.

Example of the Rewrite Step. Assume we specify a simple alu, using pattern matching, as follows:

$$\begin{aligned} alu\ ADD &= (+) \\ alu\ MUL &= (*) \\ alu\ SUB &= (-) \end{aligned}$$

So the alu is only able to add, to multiply, and to subtract numbers, i.e., the embedded language for the opcodes is as follows:

$$\textbf{data}\ \ OpCode\ =\ ADD \mid MUL \mid SUB$$

Note, however, that no matter how simple the specified alu is, the specification is polymorphic and higher order. It is polymorphic in the sense that it works for any number type, and higher order because it is defined in terms of the *functions* $(+)$, $(*)$, $(-)$ only, without using individual variables for numbers.

Intuitively, the hardware architecture specified by this definition is clear, and shown in Fig. 26.

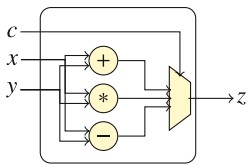


Fig. 26. The specified alu

The first step in the rewrite process is the GHC frontend which removes the syntactic sugar of pattern matching and turns de definition into a lambda-abstraction:

$$\begin{aligned} alu = \lambda c.\, &\textbf{case}\ c\ \textbf{of} \\ &ADD \rightarrow (+) \\ &MUL \rightarrow (*) \\ &SUB \rightarrow (-) \end{aligned}$$

The first rewrite step chosen by CλaSH will be *η-expansion*, i.e., to add lambda abstractions and corresponding arguments:

$$alu\ = \lambda c.\, \lambda x.\, \lambda y.\, \left(\begin{array}{l} \textbf{case}\ c\ \textbf{of} \\ \quad ADD \rightarrow (+) \\ \quad MUL \rightarrow (*) \\ \quad SUB \rightarrow (-) \end{array} \right)\ x\ y$$

The result of η-expansion is that all inputs of the alu (c, x, y) now correspond to formal parameters of the specification.

Since the case-expression is of function type, the next step is *application propagation*, i.e., to move the arguments x, y into the case-expression:

$$\begin{aligned} alu \;=\; \lambda c\, x\, y \mathbf{case}\ c\ \mathbf{of} \\ & ADD \rightarrow (+)\ x\ y \\ & MUL \rightarrow (*)\ x\ y \\ & SUB \rightarrow (-)\ x\ y \end{aligned}$$

The next step might be called *letification*, i.e., the body of the lambda term is turned into a let-expression, by introducing a name z for the expression as a whole and having that name as the only term in the body of the let-expression. The result is that the output of the architecture (see Fig. 26) corresponds to this variable z. For reasons of readability we write the arithmetical operations in an infix way:

$$\begin{aligned} alu \;=\; \lambda c\, x\, y \mathbf{let} \\ & z = \mathbf{case}\ c\ \mathbf{of} \\ & \quad ADD \rightarrow x + y \\ & \quad MUL \rightarrow x * y \\ & \quad SUB \rightarrow x - y \\ & \mathbf{in} \\ & z \end{aligned}$$

Finally, all subexpressions that are not single variables will be *extracted* and defined separately, resulting in a name for every single wire in the architecture:

$$\begin{aligned} alu \;=\; \lambda c\, x\, y \mathbf{let} \\ & p = x + y \\ & q = x * y \\ & r = x - y \\ & z = \mathbf{case}\ c\ \mathbf{of} \\ & \quad ADD \rightarrow p \\ & \quad MUL \rightarrow q \\ & \quad SUB \rightarrow r \\ & \mathbf{in} \\ & z \end{aligned}$$

The resulting expression now is in CλaSH normal form, and corresponds to Fig. 26 to the extent that all wires got names with, e.g., p being the wire that results from the addition component.

6.3 Final Remarks

As described and illustrated in these lecture notes, CλaSH is a system to specify hardware. It is based on Haskell, and translates Haskell definitions of a specific form into synthesizable VHDL, which can be mapped to, e.g., an FPGA. However, CλaSH is still under development, thus these lecture notes are not the final text on CλaSH, for an in-depth presentation of CλaSH we refer to [2]. Further

examples of architectures specified of in CλaSH can be found in, e.g., [1,12,17,18], whereas some first introductions may be found in [1,11,16].

As an example of a topic on which CλaSH still has to be extended is the usage of recursive definitions. At the moment there is no systematic treatment of recursive specifications in CλaSH yet, it is future work to fill in that gap.

References

1. Baaij, C.P.R., Kooijman, M., Kuper, J., Boeijink, W.A., Gerards, M.E.T.: Cλash: Structural descriptions of synchronous hardware using haskell. In: Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures. Methods and Tools, Lille, France, pp. 714–721. IEEE Computer Society, USA, September 2010
2. Baaij, C.P.: Digital Circuits in Cλash – Functional Specification and Type-Driven Synthesis. Ph.D. thesis, University of Twente, The Netherlands (2014, forthcoming)
3. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, pp. 174–184. ACM (1998)
4. Cardelli, L., Plotkin, G.D.: An algebraic approach to vlsi design. In: Gray, J.P. (ed.) Proceedings of the first International Conference on Very Large Scale Integration, pp. 173–182. Academic Press (1981)
5. Chen, G.: A short historical survey of functional hardware languages. ISRN Electronics (2012)
6. Coussy, P., Morawiec, A. (eds.): High-level Synthesis. From Algorithm to Digital Circuit. Springer Publishers, New York (2008)
7. Gerards, M.E.T., Baaij, C.P.R., Kuper, J., Kooijman, M.: Higher-order abstraction in hardware descriptions with Cλash. In: Kitsos, P. (ed.) Proceedings of the 14th EUROMICRO Conference on Digital System Design, DSD 2011, Oulu, Finland, pp. 495–502. IEEE Computer Society, USA (2011)
8. Gill, A., Bull, T., Kimmell, G., Perrins, E., Komp, E., Werling, B.: Introducing kansas lava. In: Morazán, M.T., Scholz, S.-B. (eds.) IFL 2009. LNCS, vol. 6041, pp. 18–35. Springer, Heidelberg (2010)
9. IEEE Standard: VHDL Language Reference Manual. IEEE (2008)
10. Johnson, S.: Applicative programming and digital design. In: Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 218–227. ACM (1984)
11. Kuper, J., Baaij, C.P.R., Kooijman, M., Gerards, M.E.T.: Architecture specifications in CλaSH. In: Kaźmierski, T.J., Morawiec, A. (eds.) System Specification and Design Languages. LNEE, vol. 106, pp. 191–206. Springer, New York (2011)
12. Niedermeier, A., Wester, R., Rovers, K.C., Baaij, C.P.R., Kuper, J., Smit, G.J.M.: Designing a dataflow processor using cλash. In: 28th Norchip Conference, NORCHIP 2010, Tampere, Finland, p. 69. IEEE Circuits and Systems Society, November 2010
13. Nikhil, R.: Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions. In: Coussy, P., Morawiec, A. (eds.) High-Level Synthesis - From Algorithm to Digital Circuit, pp. 129–146. Springer, Netherlands (2008)

14. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **23**, 17–32 (2004)
15. Sheeran, M.: μFP, a language for VLSI design. In: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, pp. 104–112. ACM (1984)
16. Smit, G.J.M., Kuper, J., Baaij, C.P.R.: A mathematical approach towards hardware design. In: Athanas, P.M., Becker, J., Teich, J., Verbauwhede, I. (eds.) Dagstuhl Seminar on Dynamically Reconfigurable Architectures, Dagstuhl, Germany. Dagstuhl Seminar Proceedings, vol. 10281, p. 11. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Dagstuhl, Germany, December 2010
17. Wester, R., Baaij, C.P.R., Kuper, J.: A two step hardware design method using cλash. In: 22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway, pp. 181–188. IEEE Computer Society, USA, August 2012
18. Wester, R., Sarakiotis, D., Kooistra, E., Kuper, J.: Specification of apertif polyphase filter bank in cλash. In: Communicating Process Architectures 2012, Scotland, pp. 53–64. Open Channel Publishing, United Kingdom, August 2012