Hardware design using Haskell and CλaSH

Rinse Wester Jan Kuper Christiaan Baaij
Dep. of Electrical Engineering, Mathematics and Computer Science
University of Twente
Enschede, The Netherlands
{r.wester, j.kuper, c.p.r.baaij}@utwente.nl

In this paper we present the usage of Haskell and CλaSH in lectures on hardware design for students at the University of Twente. We discuss some examples from the field of digital signal processing, namely a high-pass filter and a low-pass filter. We demonstrate the continuous formulation of a filter, the discretized formulation, some possibilities to schedule it over time, and the mapping towards an FPGA. The advantage of using Haskell/CλaSH is that the underlying mathematical formulations that students learn in other courses, are readily translated into Haskell and mapped on an FPGA.

1 Introduction

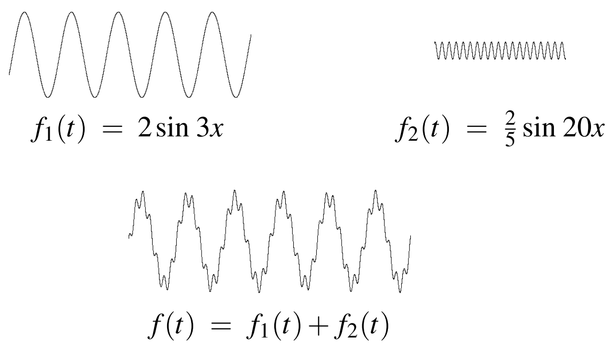
At the University of Twente, students of the master program Embedded Systems can take an optional course on the design of embedded computer architectures. An important part of the course is to design regular architectures for signal processing applications, such as high-pass filters and low-pass filters. Until recently, the filters were introduced during the course as C programs, and from there a hardware architecture was derived in a systematic way, be it manually. Since we introduced Haskell as a specification language, the filters are given in a mathematical form and have to be translated into Haskell. That holds as well for the definition of the continuous signal as of the discretized signal, and of the hardware architecture that results from that. In order to bring the theory into practice, students also have to do a practical assignment and actually configure an FPGA with their design using CλaSH, which automatically translates a slight variant of the Haskell code into synthesizable VHDL. Hence, the introduction of Haskell made it possible to include the continuous mathematical definition as well as the actual mapping onto an FPGA.

The course is given at the University of Twente, but can be chosen by students from the universities of Delft and Eindhoven as well through a life video connection. Typically, some 30-40 students in total follow the course. It should be noted, however, that most of these students do not have any background in Haskell, or in functional programming in general. Nevertheless, the usage of Haskell does not burden the students with great difficulties, as the focus of the usage of Haskell is on the structural intrpretation of highe rorder functions.

In the practical exercise that students have to make, low level and time consuming details like pin mappings and support for peripherals on the FPGA board are hidden by the framework offered during the course. In addition to education on digital hardware design, the practical assignment is also used to communicate research within our group to attract students to do their master's thesis work on digital hardware design with CλaSH.

Related work on using a functional language for hardware design in education are Hydra [3] and Lava [2]. Hydra in particular, is used as a functional language to perform processor design on different levels of abstraction (gate level, architecture level and assembly). All designs are implemented in a single environment too keep students away from complicated low level hardware design. During the approach

Submitted to:
TFPIE 2013



$f_1(t) \ = \ 2 \sin 3x$ $\qquad$ $f_2(t) \ = \ \frac{2}{5} \sin 20x$

$f(t) \ = \ f_1(t) + f_2(t)$

Figure 1: some basic signal functions

presented in this paper, however, students are exposed to real FPGA hardware and related tooling since that gives the most insight in the trade-offs that have to be made during design. It also is a preparation for other courses that use FPGA technology and industry where FPGAs have become a popular platform for signal processing applications, be it, that the configuration of an FPGA in VHDL often is considered a hindrance.

The rest of the paper is structured as follows: Section 2 presents our approach to design hardware starting from the theoretical definition in continuous hardware to the translation into CλaSH, and Section 3 discusses the more concrete details of the practical assignment that we base on this. Section 4 finally gives some educational discussion of our experiences.

2 Signal filtering

The most obvious form of a signal is a combination of trigonometric functions of different frequencies and amplitudes. As a simple example consider Figure 1 where three signals are shown: signal f_1 has a low frequency and a large amplitude, signal f_2 has a high frequency and a small amplitude, and signal f is the sum of the other two. In a realistic situation, for example with radio signals, one receives a mix of signals, and the problem is to select one of them with a reasonable precision. In this case, we will first isolate the low frequency signal f_1, i.e., we will eliminate the high frequency signal f_2 from the combined signal f. In other words, we will design a *low-pass filter*. Using this low-pass filter we will then also define a high-pass filter.

First of all, we will give the formula in continuous mathematics, second, we give a discretized form, and third we derive several hardware architectures from that which can be put on an FPGA. The discussion below will be rather superficial, but enough precise to derive a working system.

2.1 Continuous mathematics formulation

To remove a high frequency signal from a low frequency signal in a combined signal f, one has to determine the convolution of f with a suitable function h, but we will not go into details of that. Instead, we will take a more intuitive approach, and stipulate that this boils down to taking some (weighed) average value for all time moments t over a segment $[t-\frac{w}{2}, t+\frac{w}{2}]$ of the combined signal f. Here, the width w of the segment is determined by the frequencies of the signals that have to be eliminated. Of course, in a realistic situation one can not look ahead in time, so the interval will be $[t-w, t]$ and will give rise to a phase shift, but that doesn't bother us here.

The (weighed) average signal F^L over the interval $[t-\frac{w}{2}, t+\frac{w}{2}]$ of the combined signal f is expressed as an integral:

$$F^L(t) = \int_{t-\frac{w}{2}}^{t+\frac{w}{2}} f(\tau)\cdot h(t-\tau)\,d\tau \tag{1}$$

As mentioned above, for the function h several choices can be made, the most obvious one being a constant function:

$$h_1(t) = \begin{cases} 0 & \text{if } t < -\frac{w}{2} \\ \frac{1}{w} & \text{if } -\frac{w}{2} \leqslant t \leqslant \frac{w}{2} \\ 0 & \text{if } t > \frac{w}{2} \end{cases}$$

As an alternative choice for the function h often a Gaussian function (with standard deviation σ) is chosen:

$$h_2(t) = \frac{e^{-\frac{t^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}$$

Note that for every choice of the function h the following requirement has to be satisfied:

$$\int_{-\infty}^{\infty} h(t)\,dt = 1$$

The filtering technique defined above is suitable for filtering away a high frequency signal. To extract instead the *low* frequency signal, one has to subtract the result of the low-pass filter from the combined result, i.e.:

$$F^H(t) = f(t) - F^L(t)$$

Haskell. At this point the students have to give a Haskell formulation of the above mathematical definitions. Thus, they have to define a function *int* to calculate the integral, according to a standard integration algorithm such as Euler's algorithm used here, so that this function can be used a s areference function for the accuracy of the later to be defined digital filter on an FPGA. Students are allowed to choose an appropriate value for *dt* in a global definition, rather than giving it as a parameter to the function *int*. We avoid strictness issues, so for performance reasons, the value of *dt* should not be chosen too small. A straightforward defintion of *int* then is:

```
int f (a,b) = foldl (+) 0 vs
  where
    vs = [f t * dt | t <- [a,a+dt .. b]]
```

An issue that requires some attention is the curried format of function application, but students quickly get used to that, even to such an extent, that later in the course partial application is quite natural. List comprehension is quite immediate for students to understand, even though most of them has no background in a functional programming language. We stimulate the usage of higher order functions, and after some hands on exercises students are well able to corecty use them in the context of the course, in which we mainly use higher order functions in combination with artithmetical operations. Especially after the structural presentation of higher order functions, which is very adequate for the specification of hardware architecture (see 2.3), higher order functions in fact become fairly intuitive to recognize.

Defining the low-pass filter as a Haskell function now is immediate:

```
lowpass f t = int g (t−w/2,t+w/2)
  where
    g tau = f tau ∗ h (t−tau)
```

And of course:

```
highpass f t = f t − lowpass f t
```

Using these definitions of filterings, students are able to compare the result of the original signals with the filtered combined signal and recognize the influence of various choices for a specific function h and width w. In future we will offer students a graphical environment, so that they can also visualize the graphs of their functions.

2.2 Discretization

In the previous section filtering is defined in a setting of continuous mathematics, as known by students from their courses on signal processing. In order to move to digital filtering, the formulas have to be discretized. The consequence of that is a signal function f is replaced by a (possibly infinite) sequence of sampled values $[f_0, f_1, \ldots]$. The same holds for the function h: that now becomes a sequence of $N+1$ filter co-efficients $[h_0, h_1, \ldots, h_N]$, which can be calculated from the original function h, given a chosen sample frequency, and normalized such that the sum of the co-efficients equals 1.

Thinking in terms of filtering as an average over a segment, a discretized version is not unnatural:

$$F_i^L = \sum_{k=i-N/2}^{i+N/2} f_k \cdot h_{i-k} \tag{2}$$

As is standard in the area of digital filtering, we now switch without going into details to an equivalent and more standard form of this definition:

$$F_i^L = \sum_{k=0}^{N} f_{i-k} \cdot h_k \tag{3}$$

An advantage of this definition in a realistic context is that, given that i intuitively refers to the current sample, f_{i-k} refers to samples from the past, so they are known already. Note that this definition just expresses the dot product of two vectors.

A high pass filter is defined as before, though we have to take into account that the result has to be shifted over a distance of $\frac{N}{2}$.

Haskell. We first define the dot product of two vectors in Haskell:

```
dotpr xs ys = foldl (+) 0 ws
  where
    ws = zipWith (∗) xs ys
```

Again, students are stimulated to use higher order functions, which is close to their hardware background as will become clear in Section 2.3. For the same reason we do not use predefined functions such as *sum*, since then the correspondence with hardware architecture gets lost.

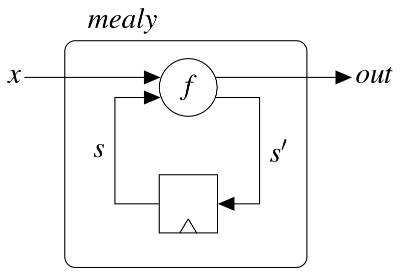


Figure 2: Mealy Machine

Now realize that a low-pass filter passes through the signal over time, meanwhile keeping the list of filter co-efficients *hs* fixed, hence we need the following definition:

```
lowpass hs fs = y : lowpass hs (tail fs)
  where
    n = length hs
    y = dotpr (take n fs) hs
```

This definition is recursive and takes some time to explain. Even though students know what recursion is and used it in an imperative context before, most of them are not really familiar with it. We strongly emphasize an *equational* way to read a recursive definition, rather than an *operational* style, i.e., assume that the subexpression

```
lowpass hs (tail fs)
```

simply *is* the filtered signal of the whole input apart from the first sample, instead of trying to imagine a few steps of the recursive definition.

Clearly, the initial part of the signal *fs* will give wrong results, since not enough samples passed by to match the length of *hs*, but that is standard in the field of signal processing and is well recognized by students. Also the fact that the end of the testinput *fs* raises an empty list exception is readily recognized, as is its solution:

```
lowpass hs [] = []
```

Issues like pattern matching, and the fact that this clause should be put *before* the clause above, do not raise any difficulties for the students.

2.3 Towards hardware

The step towards actual (synchronous) hardware introduces a *clock* and *memory*. In the context of the course on the design of hardware architectures we assume that an architecture is modelled as a *Mealy Machine* (see Figure 2), which specifies what happens during one clock cycle. The functional definition of a Mealy Machine is immediate:

```
mealy s x = (s′, out)
  where
    s′  = ...
    out = ...
```

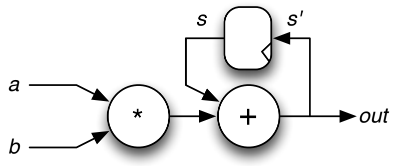


Figure 3: Multiply accumulate architecture

In this definition, s denotes the state of the architecture, x the input, s' the updated state, and out the output, where s' and out are both calculated by the function f which describes the logical circuit of the architecture. We agree that the state parameter is the first argument, and the input the second argument.

As a simple example we show a *multiply-accumulate architecture*, which gets two numbers as inputs in parallel, multiplies them, and adds the result to an accumulation register (see Figure 3):

$$
\begin{aligned}
&mac\ s\ (a,b) \;=\; (s',out)\\
&\quad \textbf{where}\\
&\qquad s' \;=\; s{+}a*b\\
&\qquad out \;=\; s'
\end{aligned}
$$

The simulation of a Mealy Machine over time is readily defined as follows:

$$
\begin{aligned}
&sim f\ s\ (x:xs) \;=\; z\ :\ sim f\ s'\ xs\\
&\quad \textbf{where}\\
&\qquad (s',z) \;=\; f\ s\ x
\end{aligned}
$$

Note that the function *sim* works for any architecture f, e.g., for the multiply-accumulate architecture *mac* (apart from a possible empty list exception):

$$sim\ mac\ 0\ [(1,1),(2,2),(3,3),...] \;=\; [1,5,14,...]$$

To apply this approach to the low pass filter discussed above, we start with the architectural pattern of *foldl* and *zipWith*, the dot product, and a compressed dot product, as shown in Figure 4. In the compressed dot product we represent multiplication and addition into a single node, referring back to equation 3. Since the co-efficients h_k are constant over time for the filter application, we also include these into the same node.

For students in a computer architecture these pictures are immediately clear, and express the corresponding archtitectures themselves.

Now assume a filter of four elements h_k long, and assume a signal f_i of values "streams through" the dot product with these four values h_k. In Figure 5 the processing of four samples is shown, where the diagonal blue lines indicate how the f-values move forward through the filter of h-values at every application of the filter. In order to design a hardware architecture based on this filtering process, we assume that every clock cycle a sample arrives. We remark that this does not necessarily coincide with reality, in the sample frequency may be lower than the hardware clock frequency. There are different possibilities for hardware architectures that execute the same filter. We give two examples of these possibilities and their derivation in Figure 6. The the first step in the derivation of the hardware architecture is to decide which nodes from Figure 4 will be executed by the same component on the hardware. In this

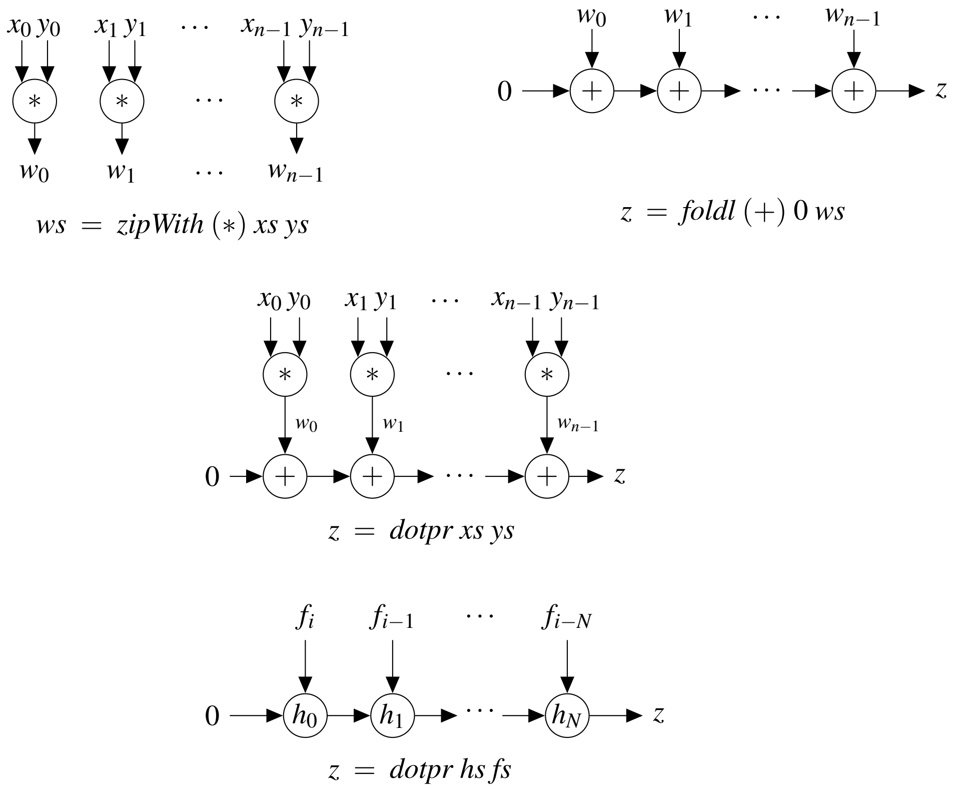


Figure 4: Structure of higher order functions

case, an obvious choice is to have four equal components, each executing those components which are in the same column in Figure 4. Note, that we might also choose to have *all* nodes be executed by the same component, leading to a re-use of the multiply-accumulate architecture (see Figure 3). Here, we won't discuss that possibility further.

The second step is to indicate which nodes have to be executed in the same time frame. We will assume that all operations in the nodes (multiplications, additions) require only one clock cycle. In Figure 6 two ways are shown, the left one performs all nodes in the same row within the same clock cycle, the right one performs those nodes in the same clock cycle that operate on the same input sample. The red lines in in the upper subfigures in Figure 6 indicate how the nodes are grouped to be executed in the same time frame.

The third step now is to realize that at those points where a *data line* crosses a *time line*, the data has to be stored for being processed in the next time frame (i.e., clock cycle). Thus, in the first possibility we need registers to store the values f_i – i.e., the values on the blue lines, whereas in the second possibility we need registers to store the values on the data lines connection two adjacent nodes. This gives rise to the architectures shown in Figure 6: the left architecture has its registers on the input, and all calculations are done in the same clock cycle, whereas in the architecture on the right the registers are between the additions, and the inputs are not stored at all.

There are many other possibilties do derive an architecture for the given algorithm. The question which is the best one is a question for hardware designers to answer, and then criteria such as longest path, maximum cock frequency, energy concumption, etc, may play a role.

Finally, in Figure 6 the Haskell code describing the two alternative architectures is shown. We

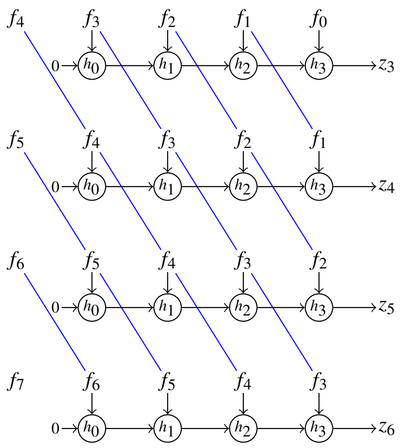
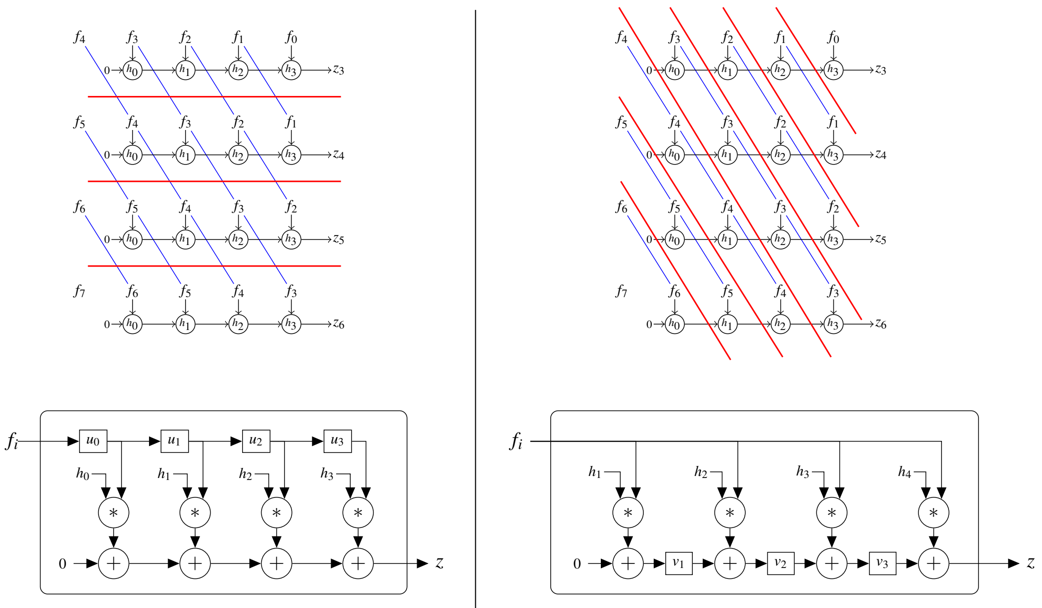


Figure 5: Filtering over a stream of values



$$
\begin{aligned}
&\mathit{fir}_1\ \mathit{hs}\ \mathit{us}\ f_i = (\mathit{us}', z)\\
&\quad \textbf{where}\\
&\qquad \mathit{us}' = f_i : \mathit{init}\ \mathit{us}\\
&\qquad \mathit{ws} = \mathit{zipWith}\ (*)\ \mathit{hs}\ \mathit{us}\\
&\qquad z = \mathit{foldl}\ (+)\ 0\ \mathit{ws}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fir}_2\ \mathit{hs}\ \mathit{vs}\ f_i = (\mathit{vs}', z)\\
&\quad \textbf{where}\\
&\qquad \mathit{ws} = \mathit{map}\ (*f_i)\ \mathit{hs}\\
&\qquad \mathit{vs}'' = \mathit{zipWith}\ (0 : \mathit{vs})\ \mathit{ws}\\
&\qquad \mathit{vs}' = \mathit{init}\ \mathit{vs}''\\
&\qquad z = \mathit{last}\ \mathit{vs}''
\end{aligned}
$$

Figure 6: Two FIR filters

choose the names fir_1 and fir_2 for these architectures to match the standard terminology *finite impulse response filter* (*fir*) for these architectures. In the definition of fir_1 the state parameter is *us*, indicating the registers $u_1,\ldots,u_4$, whereas in fir_2 the state parameter is *vs*, indicating the *v*-registers. Note that in both definitions, the list of filter co-efficients *hs* is the first argument of the architecture function, meaning that *hs* is a *parameter* of the architecture, i.e., the values h_k are fixed in the arcitecture.

The correspondence of these Haskell definitions match the architectures immediately, and students have no problems with understanding them, once they are used to meaning of the higher order functions involved.

We conclude this section with the remark that these definitions can be simulated with the function *sim* as given above. Note that the architectures to simulate are fir_1 *hs* and fir_2 *hs*, for a suitable choice of *hs*. The partial application involved is quickly accepted by the students, since it is fairly close to their hardware intuition.

2.4 Mapping to an FPGA with Cλ aSH

For the mapping of the above Haskell code we use Cλ aSH, a hardware specification mechanism developed at the university of Twente. Cλ aSH is written in Haskell and translates slightly adapted versions of specifications such as the FIR-filters above into synthesizable VHDL. The VHDL output can then be given to standard VHDL tool for the actual synthesis and the mapping on an FPGA.

The specifications that Cλ aSH translates are all Haskell programs, requiring some special hardware libraries. The Cλ aSH input language maitains many features of Haskell, such as polymorphism, pattern matching, type derivation and higher-order functions. These features allow circuit designers to describe parameterizable circuits in a natural way as we saw in the above examples of filters. Especially higher-order functions provide a lot of insight into the structure of the resulting hardware.

As mentioned before, Cλ aSH specifications are expressed in terms of a Mealy machine i.e. every output and new state is a function of the current state and input (see Section 2.3). Before Cλ aSH can actually translate the architectures defined above, these definitions have to be slightly reformulated due to typical hardware requirements. One of these requirements is that lists on hardware are of fixed length. During a computation in Haskell a list may vary in length, but on hardware that is not possible. Hence, a sequence of registers can not simply be defined as a list of values, as was done in the above examples. For that, Cλ aSH uses vector types:

Vector n a

where *n* is a natural number indicating the length of a vector, and *a* is the type of the elements in the vactor. This also means that standard Haskell functions that work for lists are not usable as hardware sepcification functions. Cλ aSH defines its own variants of such functions, indicated by a '*v*' (for "vector") in front, such as *vtake*, *vmap*, *vfoldl*, etc. There are also special notations for :, ++, etc.

Second, with respect to numbers there are several types available in Haskell, but in hardware a designer often has to specify a specific bit width for numbers. Cλ aSH offers special hardware types for that such as *Signed n* for signed numbers of *n* bits, *Unsigned n* for unsigned numbers of *n* bits, etc.

Furthermore, on hardware one often prefers fixed point number representations over floating point number representations, so the designer may have to translate a specification accordingly. A library for fixed point computations in in preparation.

As became above, Cλ aSHuses *dependent types* for vectors and for numbers of specific widths. However, in the course we avoid dependent types in combination with polymorphism, such that students are not confronted with the theorem proving aspects of dependent typing. The workflow that we choose is to first develop a Haskell specification using lists (as in the examples above), which allow for a lot of flexibility. When that is ready and the specification has to be translated into a specification that is accepted by CλaSH, students switch to specific choices for the length of their vectors, or for the bit widths of the numbers. Hence, dependent types are only used with constants at the position of the natural numbers.

There are some more CλaSH specifics, such as the fact that composition of Mealy machines is done with the arrow construction, but a discussion of that falls outside the scope of this paper.

Applyingthe above remarks to, e.g., the specification fir_1, we get:

$$\begin{aligned}
&fir_1\ hs\ us\ f_i\ =\ (us', z)\\
&\quad \mathbf{where}\\
&\qquad us' = f_i +\!\!\gg us\\
&\qquad ws = \mathbf{vzipWith}\ (*)\ hs\ us\\
&\qquad z\ \ = vfoldl\ (+)\ 0\ ws
\end{aligned}$$

Since it occurs so often in hardware designs, the operation $+\!\!\gg$ is defined as follows:

$$x +\!\!\gg xs\ =\ x :> vinit\ xs$$

where :> is the cons operation : for vectors.

3 A practical assignment

During the course a practical assignment is given in which students have to implement, simulate, and synthesize the above given FIR-filters (also called the *standard* and the *transposed* FIR filter, respectively), and in addition to that, they have to define a variant of these FIR filters, viz., a *symmetrical* filter.. The goal of the the practical assignment is to let students experiment with different, mathematically equivalent, FIR filter hardware structures in CλaSH. Students are given a framework in which they can instantiate these FIR filters. This framework targets the DE1 Development and Education board [4]. Central is the FPGA around which peripherals like an audio CODEC, buttons and LEDs are placed. By programming the FPGA with the complete framework, the filters can be physically tested by listening to the filtered audio.

When the filters are implemented using CλaSH and synthesized using the Quartus FPGA tooling [1], students are asked to describe the relation between the higher-order functions in the CλaSH description and in the resulting hardware on FPGA.

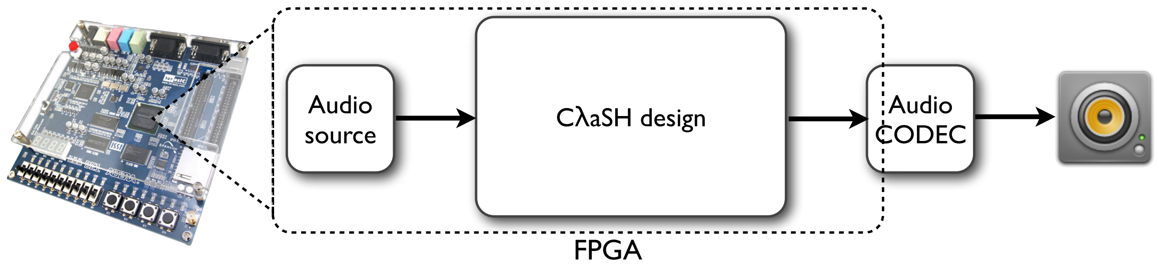


Figure 7: DE1 framework

Figure 7 shows an overview of the framework used for filter design. The framework hides all the details of communication with audio chips and pin-outs of the FPGA such that students can focus on the filter design. On the FPGA, the CλaSH design is fed with an audio sample of a bass guitar (low frequency) and a piano (high frequency). After filtering by the CλaSH design, the audio signal is forwarded to the audio CODEC connected to speakers such that the effect of the filter can be heard.

3.1 The two FIR filters again

In order to make students familiar with the CλaSH tool chain, we give the reformulations of the above FIR-filters that are acceptible for CλaSH and in which in addition all the FPGA specific aspects are mentioned. The argument *audio_in* corresponds to the sample f_i as shown before, whereas the others are connected to keys and switches on the DE1 board. The VHDL code is then integrated in the DE1 framework and synthesized using the Quartus FPGA synthesis tooling. Finally, students have to isolate the original Haskell definition of fir_1 and relate it to the structure of the design in which the aspects of the DE1 board are integrated.

```
arch :: SVect → (Sample, Bit, Bit, Bit, Byte) → (SVect, (Sample, Byte, Byte))
arch us (audio_in, key1, key2, key3, switches) = (us', (audio_out, red_leds, green_leds))
  where
    audio_out      = if key2 ≡ Low
                     then audio_filtered
                     else audio_in
    us'            = audio_in +>> us
    ws             = vzipWith fpmult filtercoefs us
    audio_filtered = vfoldl (+) 0 ws
    red_leds       = switches
    green_leds     = switches
```

Since the coefficients given to the students are fractional numbers, the multiplications are implemented using the *fpmult* function that performs a fixed-point multiplication.

For the second filter architecture, the transposed form, only a schematic is given and students have to specify the circuit in CλaSH. After specifying the design in CλaSH, the filter should be simulated to show that the transposed FIR filter behaves exactly the same as the normal filter. If this is the case, VHDL code can be generated and synthesized such that the resulting hardware can be analyzed.

The implementation in CλaSH of the transposed FIR filter shows the same details as the standard FIR-filter , including again the connection to the keys and switches on the FPGA board. Compared to the standard FIR filter implementation, the transposed form is advantageous since the maximum path length through non-register components is shorter (only a multiplier and adder). This results in a higher clock frequency on the FPGA and therefore better performance.

$$
\begin{aligned}
&\mathit{arch} :: \mathit{SVect} \rightarrow \mathit{Sample} \rightarrow (\mathit{SVect}, \mathit{Sample}) \\
&\mathit{arch}\ \mathit{vs}\ \mathit{audio\_in} = (\mathit{vs}', \mathit{audio\_out}) \\
&\quad \textbf{where} \\
&\qquad \mathit{ws} = \mathit{vmap}\ (\mathit{fpmult}\ \mathit{audio\_in})\ \mathit{filtercoefs} \\
&\qquad \mathit{vs}' = \mathit{vzipWith}\ (+)\ (0 :> \mathit{vs})\ \mathit{ws} \\
&\qquad \mathit{audio\_out} = \mathit{vlast}\ \mathit{xs}
\end{aligned}
$$

3.2 Symmetrical FIR filter

FIR filters are usually symmetrical in their coefficients i.e. the coefficients form a palindrome. Due to this symmetry, the formula of the FIR filter can be rewritten such that the number of multipliers is reduced. Consider the first and last coefficient (of the given sequence of $N+1$ coefficients):

$$
\begin{aligned}
h_0 * x_i + h_N * x_{i-N} &= h_0 * x_i + h_0 * x_{i-N} \\
&= h_0 * (x_i + x_{i-N})
\end{aligned}
$$

Since the whole FIR formula can be rewritten such that first pairs of samples are added before weighted, the number of multiplications is halved. The resulting architecture is shown in Figure 8.

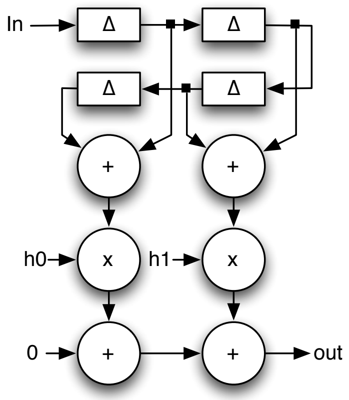


Figure 8: Symmetrical FIR filter

Again, students have to specify the architecture in CλaSH, perform a simulation to verify its correctness and generate hardware that has be analyzed. The listing below shows an example implementation in CλaSH similar to the previous architectures.

4 Discussion

In this paper we described a full Haskell/CλaSH workflow for filters in digital signal processing, from the theoretical definition in continuous mathematics to the final mapping on an FPGA. This approach is presented during a course on architecture design for students in the master program EMbedded Systems. Most of these students do not have any preliminary Haskell knowledge, nevertheless, they are well able

```
arch :: (SVect, SVect) → Sample → ((SVect, SVect), Sample)
arch ((xs1, xs2))          audio_in  =  ((xs1′, xs2′), audio_out)
  where
    xs1′        = audio_in +≫ xs1
    xs2′        = xs2 ≪+ (vlast xs1)
    vs          = vzipWith (+) xs1 xs2
    ws          = vzipWith fpmult vs filtercoefs2
    audio_out   = vfoldl (+) 0 ws
```

to follow the course. More than that, many students appreciate the approach, and it attracts several students (in fact, more than we can handle) to do their master project according to it. Projects that students choose based on this experience, include the design of non-trivial processors, the mapping to an FPGA of complex algorithms involved in, e.g., radar processing such a particle filtering. Also multi-core systems, and high performance applications such as heat diffusion and modelling the cochlea membrane are defined using Haskell during student projects.

This approach using Haskell extended the course substantially. Before Haskell was used, the emphasis was on a manual transformation from C programs to architectures, which was far more time consuming than sticking to the same perspective. The consequence was that actually mapping an application to an FPGA board and experiment with them, was not possible because of lack of time. Besides, the different semantics that a C program has in comparison with the mathematical structure of teh algorithm, and also with a VHDL formulation of the same algorithm, took a lot of effort in unifying the various perspectives.

On the other hand, the structural presentation of higher order functions now is also used in our course on Functional Programming that is given to Computer Science students during their bachelor phase. Most of these students do not choose for the master program Embedded Systems, in which the approach described in this paper is taught. Nevertheless, also these students appreciate the structural presentation of higher order functions. Students feel that this structural presentation switches the perspective on higher order functions from a data oriented interpretation to a structural interpretation.

References

[1] Altera (2013): *Quartus design software*. Available at `http://www.altera.com/products/software/sfw-index.jsp`.

[2] Per Bjesse, Koen Claessen, Mary Sheeran & Satnam Singh (1998): *Lava: hardware design in Haskell*. In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, ICFP '98, ACM, New York, NY, USA, pp. 174–184, doi:10.1145/289423.289440. Available at `http://doi.acm.org/10.1145/289423.289440`.

[3] John T. O'Donnell (2012): *Connecting the Dots: Computer Systems Education using a Functional Hardware Description Language*. In Marco T. Morazán & Peter Achten, editors: *TFPIE*, *EPTCS* 106, pp. 20–39. Available at `http://dx.doi.org/10.4204/EPTCS.106.2`.

[4] Terasic (2013): *Terasic DE1 board*. Available at `http://www.terasic.com.tw/cgi-bin/page/archive.pl?No=83`.