Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Specification of APERTIF Polyphase Filter Bank in CλaSH

Rinse WESTER [a], Dimitrios SARAKIOTIS [a], Eric KOOISTRA [b] and Jan KUPER [a]
[a] *Department of Computer Science, University of Twente, The Netherlands*
[b] *ASTRON, Dwingelo, The Netherlands*

Abstract. CλaSH, a functional hardware description language based on Haskell, has several abstraction mechanisms that allow a hardware designer to describe architectures in a short and concise way. In this paper we evaluate CλaSH on a complex DSP application, a Polyphase Filter Bank as it is used in the ASTRON APERTIF project. The Polyphase Filter Bank is implemented in two steps: first in Haskell as being close to a standard mathematical specification, then in CλaSH which is derived from the Haskell formulation by applying only minor changes. We show that the CλaSH formulation can be directly mapped to hardware, thus exploiting the parallelism and concurrency that is present in the original mathematical specification.

Keywords. CλaSH, parallel specification, FPGA, filter bank.

Introduction

As the complexity of embedded systems increases, there is a need for more abstraction in the specification of such systems. CλaSH[1] is a relatively new functional Hardware Description Language (HDL) and compiler that addresses these abstractions. In addition, CλaSH is an appropriate language for describing parallel hardware. We use CλaSH because it supports higher order functions and type inference which allows for short and concise hardware descriptions. Currently, CλaSH has only been applied to a few test cases of small to medium complexity (FIR filter [2], a reducer circuit [1] and a dataflow processor [3]). In this paper we investigate a more complex case, the design of a Polyphase Filter Bank which is used in the Westerbork Synthesis Radio Telescope as part of the APERTIF project [4].

This paper describes how the APERTIF Polyphase Filter Bank is specified and simulated using CλaSH. First, all components of the APERTIF Polyphase Filter Bank are specified using the functional language Haskell[5]. Since the CλaSH compiler accepts a subset of Haskell as input language, only minor modifications have to be made to the Haskell description before it can be compiled by CλaSH. Secondly, this CλaSH description is simulated to verify the correctness of the design. The last step is generating the hardware for FPGA such that performance numbers can be extracted.

Using Haskell for hardware design is not new, several embedded languages exist supporting simulation verification and generation of hardware [6] [7]. What makes CλaSH different from other embedded languages is that CλaSH accepts a subset of Haskell itself instead of an embedded language in Haskell.

This paper is organized as follows: Section 1 shows the background on the Polyphase Filter Bank and an introduction to the CλaSH language. The specification of the APERTIF Polyphase Filter Bank is presented in Section 2. Simulation results and performance numbers of the resulting hardware are presented in Section 3. Finally, in Section 4 we draw the final conclusions and discuss future work.

1. Background

The Netherlands Institute for Radio Astronomy (ASTRON) is currently developing technology to increase the field of view (area of the sky that can be observed at the same time) of the Westerbork Synthesis Radio Telescope in the APERTIF project [4]. An important part in the signal processing chain that combines the signals from the telescope dishes is the Polyphase Filter Bank. First, the structure of a Polyphase Filter Bank is introduced, followed by an introduction to the CλaSH language.

1.1. APERTIF Filter Bank

The field of view of the Westerbork Synthesis Radio Telescope is increased by replacing the single antenna in the dishes with a small array of antennas. The signals of this array are combined by a beam former which consist of two parts: a Polyphase Filter Bank for each antenna and a part that combines all these signals. This paper only focuses on the specification of the Polyphase Filter Bank.

A Polyphase Filter Bank consist of two parts, a polyphase filter and an FFT [8]. The polyphase filter is used for decimation the input signal before sending it to the FFT. The FFT on the other hand splits the signal into its frequency components such that all antenna signals can be easily combined in the beamformer. The structure of the APERTIF Polyphase Filter Bank is shown in Figure 1.

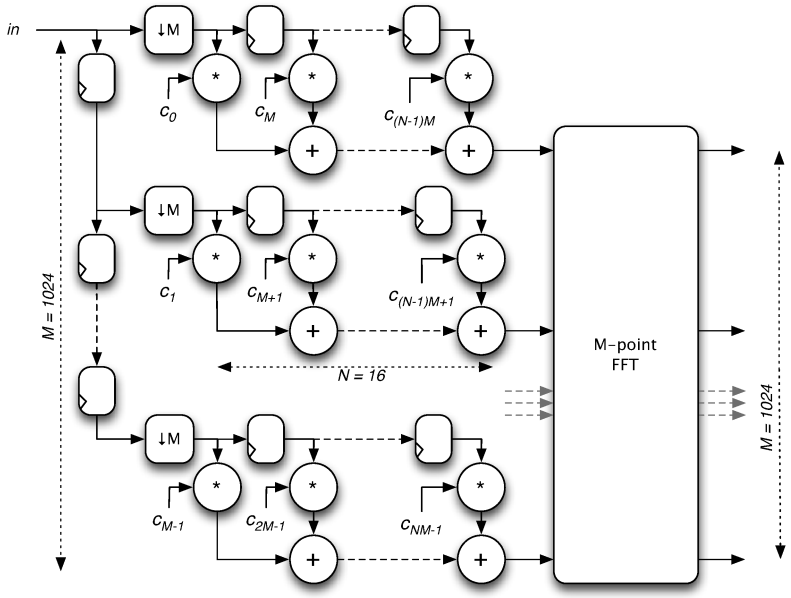

Figure 1. APERTIF Polyphase Filter Bank

The polyphase filter consist of 1024 (M) FIR filters each having 16 (N) taps and is derived from a single filter with $M \times N = 16384$ coefficients [9]. All coefficients are column-wise distributed in the polyphase filter i.e. the first M filter coefficients form the first column $(C_0, C_1, \ldots, C_{M-1})$. In front of the filters, a decimation step $\downarrow M$ is used to reduce the sample rate of the data by a factor M. The combination of delays and decimation has the same effect as a commutator, a switch sending sequentially a single sample to all FIR filters. Since only one filter is active for each sample, the amount of required hardware can be reduced dramatically compared to a fully parallel implementation. All filters can therefore be merged into a single filter alternating between the different sets of coefficients and registers. This also means that for each sample consumed by the filters also one sample will be sent to the FFT.

The FFT splits the signal into M distinct frequency components which are combined for all antennas in the beamformer. The architecture of the FFT is a pipeline as described in [10]. From the size and radix of the FFT, it follows that $\log_4(1024) = 5$ stages are required.

The whole Polyphase Filter Bank design should fit on a single Altera Stratix IV FPGA (EP4SGX230KF40C2). Data from the antenna arrives in the FPGA using high speed serial interconnect producing data at 800 MS/s. The desired clock frequency for the filterbank is 200 MHz. Therefore, the structure has to be parallelized by a factor of $p = 800/200 = 4$ in order to meet the throughput.

1.2. CλaSH

CλaSH [1] is a new functional hardware description language based on Haskell. CλaSH is both a simulation environment and a compiler . The language accepted by the CλaSH compiler (a subset of Haskell that can be translated to hardware) supports advanced features such as poly-morphism, higher-order functions, pattern matching and type derivation. Polymorphism and higher-order functions (functions that have functions as argument or result) allow circuit designers to describe parameterizable circuits in a natural way. Especially Higher Order functions are a powerful abstraction since they allow for reasoning about structure and parallelism of the hardware.

CλaSH is a purely synchronous and cycle accurate hardware description language where everything is, on the lowest level, expressed as a Mealy machine. Therefore, every output and new state is a function of the input combined with the current state. Since every CλaSH description is also a valid Haskell program, simulation comes for free. This combination results in a fast and cycle accurate hardware simulator.

Besides simulating hardware, CλaSH is also able to translate the description to VHDL. For simulation, CλaSH accepts plain Haskell but for translation to VHDL this is limited to descriptions without general recursion and lists (the length may change during runtime).

Listing 1 shows a simple example, a multiply accumulate, written in CλaSH. Every function in CλaSH is formatted as shown in Listing 1. First, the name of the function to be defined is given (mac) followed by the current state (s) and the inputs (a, b). These are arguments of the the function mac and separated by spaces instead of commas. The result consists of the new state s' using the $State$ keyword and the output out. Finally, all calculations are performed, combinatorially, in the *where* clause.

Listing 1 Multiply Accumulate example in CλaSH.

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

The Hardware corresponding with Listing 1 is shown in Figure 2.

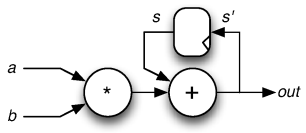

Figure 2. Multiply Accumulate structure

As mentioned before, CλaSH supports an abstraction mechanism called *higher order functions*, which are very useful to describe structure and parallelism. Higher order functions

are functions that can accept functions as argument or return a function as a result which is particularly useful for describing structure. Listing 2 shows a description of a FIR filter utilizing the higher order functions $vzipWith$ and $vfoldl$ (the prefix 'v' refers to vector i.e. a list of fixed length).

The *fir* example of Listing 2 accepts an additional argument cs which contains a vector of filter coefficients. The registers of the filter, the input and output are called us, in and out respectively. The new state of the registers us' is the original state us with the input shifted in one position using the $+\!\!\gg$ operator. $vzipWith$ is used in the second line to pairwise multiply the coefficients cs with the contents of the registers us. Finally, the last line shows the use of $vfoldl$ which accepts $+$ as functional argument and therefore adds all ws together. Note that cs (the filter coefficients) are parameters for the *fir* function.

Listing 2 Higher order functions in a FIR filter described using CλaSH.

```
fir cs (State us) inp = (State us', out)
  where
    us'  = inp +>> us
    ws   = vzipWith (*) us cs
    out  = vfoldl (+) 0 ws
```

Important to note is that the description in the *where* clause only expresses data dependencies and no sequential ordering. Therefore, the description is implicitly parallel and there is no need for special notation. Also, $vzipWith$ is implicitly parallel, it applies a function pairwise to the elements of the two lists us and cs. The schematic corresponding with Listing 2 is shown in Figure 3.

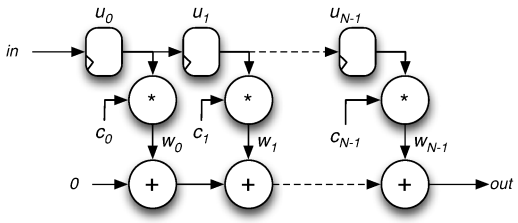

Figure 3. FIR filter structure

2. Specification of APERTIF Polyphase Filter Bank

As explained in Section 1.1, the Polyphase Filter Bank consists of two parts: the Polyphase Filter and the FFT pipeline. In the following two sections, we present the specification of the Polyphase Filter and FFT pipeline. First, the specification is given in Haskell which is then slightly changed such that it is accepted by the CλaSH compiler. These changes are necessary since the CλaSH compiler does not support floating point numbers and a proper fixed point representation for numbers is needed.

2.1. Polyphase Filter

2.1.1. Specification in Haskell

Since the basic building block of the Polyphase Filter is a FIR filter, we start by specifying the CλaSH description of Listing 2 in Haskell. The code of the filter is shown in Listing 3.

Listing 3 FIR filter in Haskell.

```
fir cs us inp = (us', out)
  where
    us'  = inp +>> us
    ws   = zipWith (*) us cs
    out  = foldl (+) 0 ws
```

The *fir* function accepts three input arguments: a list of filter coefficients cs, a list of registers containing the current state us and an input inp. During a single clock cycle, the new state s' and output out are calculated in the *where* clause. As a matter of notation, lists and lists of lists are denoted with one s or two ss respectively. Note that parameters cs remain constant during simulation.

Now that we have defined the FIR filter, we can use the *fir* function to create the Polyphase Filter. Since only one filter of the Polyphase Filter is active at each clock cycle, the hardware performing the *fir* computation can be reused. However, new filter coefficients and register contents have to be supplied to the filter every time the next filter is executed. The architecture that performs this operation is shown in Figure 4.

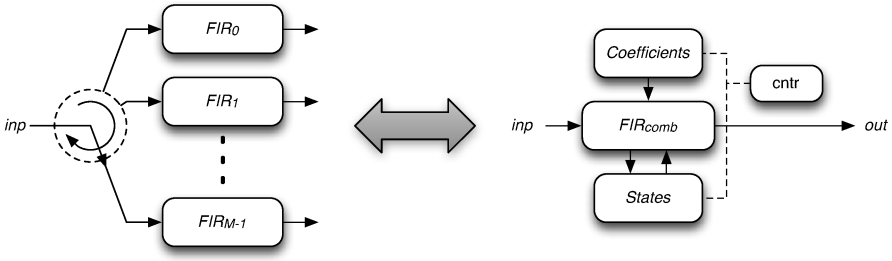

Figure 4. Sequential FIR filter execution

To translate this into a Haskell description, all states of the filters have to be stored in the global state of the Polyphase Filter and a proper selection of the filter coefficients has to be made at every cycle. The easiest way to control this is by a counter that determines which set of coefficients and state registers will be used. Listing 4 shows the Haskell code of the complete Polyphase Filter with proper state and coefficient selection.

Listing 4 Polyphase Filter in Haskell.

```
pfs css (uss, cntr) inp = ((uss', cntr'), out)
  where
    cntr'       = (cntr + 1) `mod` (length css)
    us          = uss !! cntr
    cs          = css !! cntr
    (us', out)  = fir cs us inp
    uss'        = replace cntr us' uss
```

As can be seen in Listing 4, the pfs function accepts three arguments: a parameter list containing lists of filter coefficients css, the internal state of the PF $(uss, cntr)$ (consisting of the memory and a counter) and the actual input inp. Again, only data dependencies are described, everything else is fully parallel. During a single cycle, a new filter state is stored in the memory uss', the internal counter is incremented while the output is sent to out. The

actual set of registers us and the set of coefficients cs are selected based on the counter from uss and css respectively. This is performed using the index operator "!" i.e. us and cs are the appropriate registers and coefficients for the current filter. The actual filtering part is performed in the fourth line in the *where* clause using the selected register states us and set of filter coefficients cs. This calculation results in a new filter state us' and a new output out. Finally, the changed filter state is stored in the global state of the Polyphase Filter uss'.

As is mentioned before, the Polyphase Filter has to be parallelized by a factor of $P = 4$ to meet the throughput of 800 MSps at a clock frequency of 200 MHz. Since there are no data dependencies between the FIR filters, the hardware structure represented by the pfs function can simply be replicated 4 times. However, the filter coefficients and registers have to be distributed among these four Polyphase Filters. The parallelization is depicted in Figure 5.

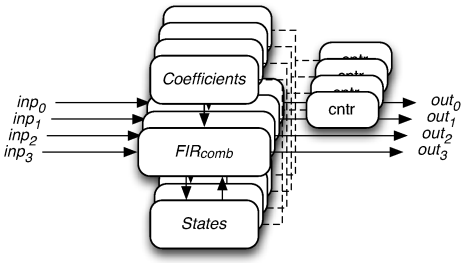

Figure 5. Parallel FIR filters

As can be seen in Figure 5, the Polyphase Filter is parallelized with a factor $P = 4$. The coefficient css and the registers uss are distributed linearly over the four Polyphase Filters i.e. cs_n and us_n are located at pfs_m (where $m = n \bmod p$) respectively. Since the FIR_{comb} function is replicated four times, the new architecture will also have four inputs and four outputs. The Haskell code describing this architecture is shown in Listing 5.

Listing 5 Parallel Polyphase Filter in Haskell.

```
parpfs csss states inps = (states', outs)
  where
    res             = zipWith3 pfs csss states inps
    (states', outs) = unzip res
```

As can be seen in Listing 5, the parallel Polyphase Filter accepts three arguments: a list (of lists of lists) of coefficients $csss$, a list of states $states$ (for the filter registers and counters) and a list of inputs (4 since $P = 4$). Since a single pfs accepts three arguments, the higher order function $zipWith3$ is used to create 4 instantiations of this block. $zipWith3$ accepts a function pfs and three lists ($csss$, $states$, $inps$) after which pfs is applied to each element in the lists. This results in a list of tuples res which are split into a list of new states $states'$ and the outputs $outs$.

2.1.2. Translation to CλaSH

The last step is to apply a few changes to the Haskell description such that the design is accepted by the CλaSH compiler. Since hardware is finite and fixed, standard Haskell lists are not supported because they can be infinite and their length can change during computation. Therefore, lists are replaced by *vectors* which have a finite and fixed length and are easy to translate to hardware. Also floating point operations are not supported since this would require a lot of hardware. Therefore, all floating point operations have to be rewritten in an equivalent *fixed point* representation.

Modifying the Haskell code for CλaSH of the FIR filter of Listing 3 is trivial since the functions $zipWith$ and $foldl$ only have to be replaced by their vector counterparts ($vzipWith$ and $vfoldl$). As mentioned before, CλaSH has no support for floating point numbers and therefore special care has to be taken for the multiplications in the FIR filter. The APERTIF Polyphase Filter Bank uses a Kaiser window for the filter coefficients which are taken to be less than 1.0. Therefore, a fixed point implementation doesn't need an integer part and the fixed point coefficient c_{FP} can be found by (1) (assuming 18 bits sample size).

$$c_{FP} = \text{round}(c * 2^{17}) \tag{1}$$

The function to perform a fixed point multiplication, $fpmult$, is shown in Listing 6 where both operands (18 bits unsigned numbers) are first resized to 36 bit signed numbers. Then, the actual multiplication is performed after which the result is divided by 2^{17} (shifting 17 bits to the right) and resized (back to 18 bits) to keep the number of significant bits the same.

Listing 6 Fixed point multiplication in CλaSH

```
fpmult :: Unsigned D18 → Unsigned D18 → Unsigned D18
fpmult a b = c
  where
    a' = resizeSigned a :: Signed D36
    b' = resizeSigned b :: Signed D36
    c' = a' * b'
    c  = resizeSigned (c' `shiftR` 17) :: Signed D18
```

All necessary changes are now made to the Haskell description such that it is accepted by the CλaSH compiler. The final code is shown in Listing 7.

Listing 7 Complete Polyphase Filter in CλaSH

```
fir cs (State us) inp = (State us', out)
    where
      us'  = inp +>> us
      ws   = vzipWith fpmult us cs
      out = vfoldl (+) 0 ws
  pfs css (State (uss, cntr)) inp = (State (uss', cntr'), out)
    where
        -- same as Listing 4 but with vzipwith
  parpfs csss (State states) inps = (State states', outs)
    where
      res             = vzipWith3 pfs csss states inps
      (states', outs) = vunzip res
```

Again, all functions in Listing 7 are distinct components of the hardware which shows the implicit parallelism.

2.2. FFT pipeline

The FFT pipeline is built according to the same procedure followed for the Polyphase Filter, accept that the FFT is only shown with parallelization factor $P = 1$. First, the basic building blocks are built in Haskell and combined into a full pipeline. Secondly, parts of the code that are not supported by CλaSH are changed such that hardware can be generated with CλaSH.

2.2.1. Haskell description

The FFT pipeline is based on [10] and utilizes a radix 2^2 algorithm which requires the same number of multipliers as a radix 4 algorithm but has the same butterfly structure of a radix 2 algorithm. This pipeline uses two types of butterfly blocks and a complex multiplier as depicted in Figure 6.

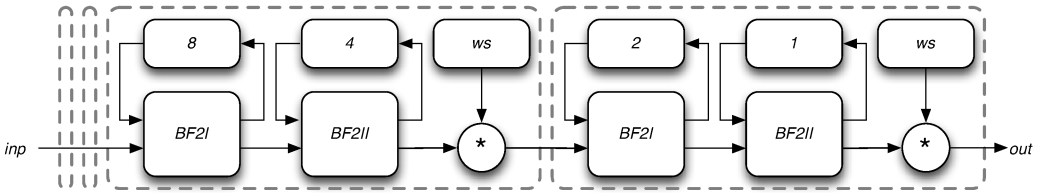

Figure 6. FFT pipeline

The first butterfly operation *BF2I* has two modes: a stage where data is simply forwarded to the memory located above and a stage where the butterfly operation is performed. All operations are performed on complex numbers which results in the schematic shown in Figure 7.

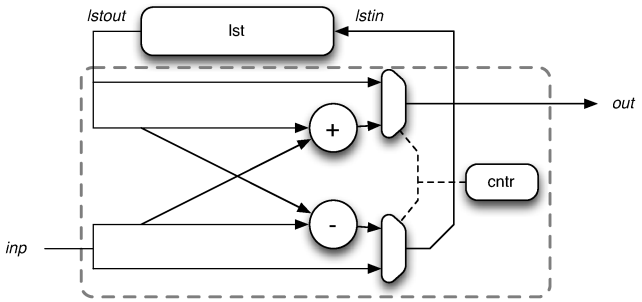

Figure 7. BF2I butterfly structure

Specifying the architecture from Figure 7 in Haskell is easy since Haskell supports complex numbers. As can be seen in Listing 8, the $bf2i$ operation accepts a single input inp, has a state consisting of a counter cnt and a list for memory lst and a single output out.

Listing 8 BF2I butterfly operation in Haskell

```
bf2i (cntr, lst) inp = ((cntr', lst'), out)
  where
    n           = length lst
    cntr'       = (cntr + 1) `mod` n
    lst'        = lstin +>> lst
    (out, lstin) = if cntr >= n
                   then (lstout + inp, lstout - inp)
                   else (lstout      , inp         )
    lstout      = last lst
```

Also the second butterfly operation *BF2II* has a stage where data are stored and a stage where the butterfly computation is performed. However, the butterfly operation in *BF2II* comes in two variations: a butterfly operation as in *BF2I* and one with an additional multiplication with the complex number $-j$. Figure 8 shows the structure of *BF2II*.

Implementing the architecture of Figure 8 in Haskell is now straightforward (Listing 9).

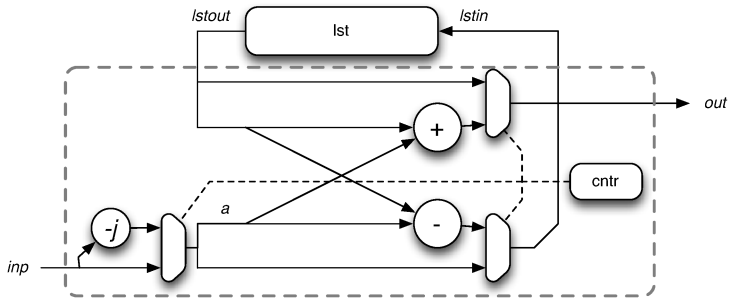

Figure 8. BF2II butterfly structure

Listing 9 BF2II butterfly operation in Haskell

```
bf2ii (cntr, lst) inp = ((cntr', lst'), out)
  where
    n           = length lst
    cntr'       = (cntr + 1) mod n
    lst'        = lstin +>> lst
    (out, lstin) = if (n <= cntr < 2 * n) || (3 * n <= cntr < 4 * n)
                   then (lstout + a, lstout - a)
                   else (lstout    , a)
    lstout      = last lst
    a           = if cntr >= 3 * n
                  then inp * (-j)
                  else inp
```

The last component to describe is the complex multiplier which multiplies every incoming sample with a twiddle factor. The state of the complex multiplier only consist of a counter $cntr$ to select the correct twiddle factor w from the list of twiddle factors ws (first parameter). Listing 10 shows the implementation as it is written in Haskell.

Listing 10 Complex multiplier in Haskell

```
cmult ws cntr inp = (cntr', out)
  where
    n    = length ws
    cntr' = (cntr + 1) `mod` n
    w    = ws !! cntr
    out  = inp * w
```

By combining the complex multiplier, BF2I and BF2II into a single function, a basic building block for the FFT pipeline is formed. The states of all the building blocks are simply combined in a single tuple and the twiddle factors are given as an extra input (the first argument). Listing 11 shows how the aforementioned components are chained in the basic building block.

The final step is to create a function that describes the full FFT chain. As is mentioned in the previous sections, the FFT chain consists of a set of basic building blocks chained together. This could be written down using recursion, however, the CλaSH compiler doesn't support recursion (yet). Furthermore, the length of the FFT is fixed $M = 1024$ for the application. As can be seen in Figure 6, the size of the memory, used for intermediate results

Listing 11 Basic building block of the FFT

```
fftbb ws (bf1state, bf2state, cmstate) inp = ((bf1state', bf2state', cmstate'), out)
  where
    (bf1state', a)   = bf2i bf1state inp
    (bf2state', b)   = bf2ii bf2state a
    (cmstate', out) = cmult ws cmstate b
```

in the butterfly, is different depending on the position in the chain. Although this is not a problem for lists in Haskell, it will be for vectors in CλaSH since the length is encoded in the type resulting in a different type depending on the position of the butterfly in the chain. Therefore, we have chosen to describe the FFT chain in Haskell by separately defining each stage of the FFT in a single function such that the result resembles the eventual hardware more accurately (avoiding this repetition of code still remains future work). Listing 12 shows the Haskell description of this function.

Listing 12 Haskell code of FFT chain

```
fftchain (ws1, ws2, ...) (bb1state, bb2state, ...) inp = ((bb1state', bb2state', ...), out)
  where
    (bb1state', d1)  = fftbb ws1 bb1state inp
    (bb2state', d2)  = fftbb ws2 bb2state d1
      ∘
      ∘
    (bbNstate', out) = fftbb wsN bbNstate d9
```

2.2.2. Translation to CλaSH

After having fully specified the FFT pipeline in Haskell, it is time to modify the code such that it is accepted by the CλaSH compiler. As was the case for the Polyphase Filter, all lists have to be replaced by vectors and all floating point numbers have to be replaced by an appropriate fixed point implementation. Special care must be taken in the two butterfly functions (*BF2I* and *BF2II*) in order to prevent any overflow. Therefore, data is first resized from 18 to 19-bits before starting the butterfly computation. After the butterfly computation, the result is shifted one position to the right to retain the same number of significant bits. Finally, the result is resized back to 18 bits again before it is sent to the next component. To hide these low level details, the $+$ and $-$ operators are overloaded by a function that performs the fixed point operation (the $(+)$ function in Listing 13). Listing 13 shows how the *BF2I* butterfly operation is implemented in CλaSH.

The changes for the *BF2II* butterfly and complex multiplier are performed in the same way and therefore not further elaborated. By combining both butterflies and the complex multiplier, we create the basic building block for the FFT chain. This function $fftbb$, is the first function in Listing 14 and accepts the list of twiddle factors ws as argument from outside of the function. The FFT chain itself, is composed only of building blocks composed in a single function $fftchain\_clash$.

3. Results

The full Polyphase Filter Bank of APERTIF has been implemented using and simulated using CλaSH. The resulting CλaSH description is concise and cycle accurate. Simulation is

Listing 13 BF2I butterfly operation in CλaSH

```
bf2i_clash (cntr, lst) inp = ((cntr', lst'), out)
  where
    n           = vlength lst
    cntr'       = cntr + 1
    lst'        = lstin +>> lst
    (out, lstin) = if cntr >= n
                   then (lstout + inp, lstout - inp)
                   else (lstout,       inp)
    lstout      = vlast lst
(+) a b = c
  where
    a' = resizeSigned a :: Signed D19
    b' = resizeSigned b :: Signed D19
    c  = resizeSigned ((a + b) shiftR 1) :: Signed D18
```

Listing 14 Basic building block of the FFT and FFT chain function in CλaSH

```
fftbb_clash ws (bf1state, bf2state, cmstate) inp = ((bf1state', bf2state', cmstate'), out)
  where
    (bf1state', a)   = bf2iclash bf1state inp
    (bf2state', b)   = bf2iiclash bf2state a
    (cmstate, out) = cmult ws cmstate b
fftchain_clash (ws1, ws2, ...) (bb1state, bb2state, ...) inp = ((bb1state', bb2state', ...), out)
  where
  (bb1state', d1)   = fftbb_clash ws1 bb1state inp
  (bb2state', d2)   = fftbb_clash ws2 bb2state d1
    o
    o
  (bbNstate', out)  = fftbb_clash wsN bbNstate d9
```

performed using the builtin *sim* function. This function accepts two arguments: a function representing the architecture to be simulated and a list of input values and produces a list of tuples as an output $[(states', outs)]$, where $states'$ are the new states and $outs$ are the actual outputs of the simulated architecture. The specification presented in this paper has been simulated both in Haskell and CλaSH and verified for functional correctness using Matlab by comparing it with the output of the standard functions *fft* and *filter*.

Besides simulation, CλaSH has also been used to generate hardware (although the M had to be reduced to 256 for the Polyphase Filter to make the design fit in the FPGA). The resulting VHDL code has been synthesized using Altera Quartus and the results are shown in Table 1.

4. Conclusions and Future Work

A complex digital signal processing algorithm, the APERTIF Polyphase Filter Bank, has been built using CλaSH. The resulting design has been simulated and behaves correctly. This shows that CλaSH is an appropriate tool to developed complex parallel architectures like a Polyphase Filter Bank since the description is fully parallel and cycle accurate. Also simula-

Table 1. Synthesis results of the Polyphase Filter Bank

	Polyphase Filter(256 elements)	*1k-points FFT*
Logic Utilization	91%	6%
number of dedicated logic registers	74886 (41%)	5762 (3%)
number of block-RAMs	0	0
number of DSP blocks (18-bit elements)	128	70
fmax for slow 900mV 0C model	114 MHz	195 MHz

tion is relatively fast since CλaSH code is valid Haskell code and therefore easy to compile and run. Furthermore CλaSH itself pushes the user to exploit any available parallelization of the structure that is describing. Therefore it is a suitable hardware description language for describing parallel structures, since CλaSH designs are implicitly parallel.

During hardware generation, it has been shown that the current version of CλaSH was not able to instantiate blockRAMs. Therefore, all filter coefficients and twiddle factors have to be hardcoded into the CλaSH description which requires a lot of memory during compilation. Not being able to use blockRAMs results in a lot of register consumption which reduces the performance of the circuit significantly due to excessive routing. Therefore, the hardware results are not realistic. Currently, a new version of CλaSH is being developed with proper blockRAM support which should give realistic numbers.

In the future, CλaSH could really benefit from IP (Intellectual Property) support since the resulting hardware would have more performance in term of area and clock frequency.

References

[1] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. 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*, pages 714–721, USA, September 2010. IEEE Computer Society.

[2] J. Kuper, C. P. R. Baaij, M. Kooijman, and M. E. T. Gerards. Exercises in architecture specification using CλaSH. In *Proceedings of Forum on Specification and Design Languages, FDL 2010, Southampton, England*, pages 178–183, Gières, France, September 2010. ECSI Electronic Chips & Systems design Initiative.

[3] A. Niedermeier, R. Wester, K. C. Rovers, C. P. R. Baaij, J. Kuper, and G. J. M. Smit. Designing a dataflow processor using CλaSH. In *28th Norchip Conference, NORCHIP 2010, Tampere, Finland*, page 69. IEEE Circuits and Systems Society, November 2010.

[4] ASTRON. APERTIF Project, 2012. `http://www.astron.nl/general/apertif/apertif/`.

[5] The GHC Team. The Glasgow Haskell Compiler, 2012. `http://www.haskell.org/ghc/`.

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

[7] Nelio Muniz Mendes Alves and Sergio de Mello Schneider. Implementation of an Embedded Hardware Description Language Using Haskell. 9(8):795–812, aug 2003.

[8] Richard G. Lyons. *Understanding Digital Signal Processing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[9] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Pearson Education, Inc., fourth edition, 2007.

[10] Shousheng He and M. Torkelson. A new approach to pipeline FFT processor. In *Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International*, pages 766 –770, apr 1996.