Specification of APERTIF Polyphase Filter Bank in ClaSH

Rinse Wester\textsuperscript{a}, Dimitrios Sarakiotis\textsuperscript{a}, Eric Kooistra\textsuperscript{b}, Jan Kuper\textsuperscript{a}

University of Twente, Enschede
ASTRON, Dwingelo

August 27, 2012
Contents

• Introduction
• Background
• Describing the Filter bank using CλaSH
• Results
• Conclusions & Future Work
Introduction

- What is C\(\text{aSH}\)?
  - Functional Language and Compiler for Concurrent Digital Hardware Design

- Motivation?
  - Testing C\(\text{aSH}\) on real life complex application

- Why APERTIF Polyphase filter bank?
  - Strict specification on Throughput, Area and clock frequency
Background

• CλaSH

• A functional language and compiler for digital hardware design

• On the lowest level, everything is a Mealy machine \[ f(s,i) = (s',o) \]

• A CλaSH description is purely structural i.e. all operations are performed in a single clock cycle

• Simulation is cycle accurate
Background

\[
\text{mac } (\text{State } s) (a, b) = (\text{State } s', \text{out})
\]

\[
\text{where } \\
\begin{align*}
s' &= s + a \times b \\
\text{out} &= s'
\end{align*}
\]
Background

\[
\text{fir } cs \ (\text{State } us) \ \text{inp} = (\text{State } us', \ out)
\]

where

\[
us' = \text{inp } \triangleright\triangleright \ us
\]

\[
ws = \text{vzipWith } (*) \ us \ cs
\]

\[
out = \text{vfoldl } (+) \ 0 \ ws
\]
Background

APERTIF Polyphase Filter bank

• Increasing field of view of Westerbork telescope using small array

• Each antenna in the array requires a polyphase filter bank

• Goals: $F_{clk} = 200 \text{ MHz}$ and throughput = 800 MS/s
Background

- APERTIF Polyphase Filter bank
- Polyphase FIR filter
- FFT

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 C\textsubscript{aSH} 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 consists 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.

The polyphase filter consists 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 \( (C_0, C_1, ..., C_M) \). In front of the filters, a decimation step \#\( M \) 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.
Describing the PFB

- Design method
- Polyphase filter
- FFT pipeline
Describing the PFB

- Design whole Architecture first in plain Haskell
- Perform small modification such that the code is accepted by ClaSH
  - Lists are replaced by vectors: lists with fixed length
- Fixed point representation for numbers
- A clear division between structure and low level hardware details
Describing the PFB

- A set of FIR filters sequentially activated
- Parallelization of P=4 needed

\[
pfs \ css \ (\ uss, \ cntr) \ inp = ((\ uss', \ cntr'), \ out)
\]

where

- \( cntr' = (cntr + 1) \mod (\text{length} \ css) \)
- \( us = uss!cntr \)
- \( cs = css!cntr \)
- \((us', \ out) = \text{fir} \ cs \ us \ inp \)
- \(uss' = \text{replace} \ cntr \ us' \ uss \)
Describing the PFB

- A set of FIR filters sequentially activated
- Parallelization of P=4 needed

\[
pfs css ún\ s cntr\ inp = ((ús', \ cntr'), \ uts)
\]

\[
where
\begin{align*}
\text{cntr}' &= (\text{cntr} + 1) \mod (\text{length css}) \\
\text{ús} &= \text{ús} \lookup \text{cntr} \\
\text{cs} &= \text{css} \lookup \text{cntr} \\
(\text{ús}', \ uts) &= \text{fir} \ cs \ us \ inp \\
\text{ús}' &= \text{replace} \ \text{cntr} \ us' \ uts
\end{align*}
\]
Describing the PFB

- A set of FIR filters sequentially activated
- Parallelization of P=4 needed

\[
pfs\ css\ (uss,\ cntr)\ inp = ((uss',\ cntr'),\ out)\]

\[
\text{where} \quad cntr' = (cntr + 1) \mod (\text{length css})
\]
\[
us = uss ! cntr
\]
\[
cs = css ! cntr
\]
\[
(us',\ out) = fir\ cs\ us\ inp
\]
\[
uss' = \text{replace}\ cntr\ us'\ uss
\]
The last step is to apply a few changes to the Haskell description such that the design is distributed among these four Polyphase Filters. The parallelization is depicted in Figure 5.

As can be seen in Listing 5, the parallel Polyphase Filter accepts three arguments: a list of tuples (parpfs csss states inps)

Listing 5

\[
\text{parpfs csss states inps} = (\text{states}', \text{outs})
\]

where

\[
\text{res} = \text{zipWith3 pfs csss states inps}
\]

\[
(\text{states}', \text{outs}) = \text{unzip res}
\]
Describing the PFB  

FIR filter: Haskell → C\aSH

fir :: [Double] → (State [Double]) → Double → (State [Double], Double)

fir cs (State us) inp = (State us, out)

where

us = inp >>= us
ws = zipWith (*) uscs
out = foldl (+) 0 ws
Describing the PFB

**FIR filter: Haskell → aSH**

\[
\text{fir} :: [\text{Double}] \to (\text{State} [\text{Double}]) \to \text{Double} \to (\text{State} [\text{Double}], \text{Double}) \\
\text{fir} \ cs \ (\text{State} \ us) \ inp = (\text{State} \ us, \ out) \\
\text{where} \\
us = inp \mapsto us \\
ws = \text{zipWith} (*) \ uscs \\
out = \text{foldl} (+) 0 \ ws \\
\]

\[
\text{fir} :: \text{Vector D}16 \ S \to (\text{State} \ (\text{Vector D}16 \ S)) \to S \to (\text{State} \ (\text{Vector D}16 \ S), S) \\
\text{fir} \ cs \ (\text{State} \ us) \ inp = (\text{State} \ us, \ out) \\
\text{where} \\
us = inp \mapsto us \\
ws = \text{vzipWith} \text{‘fpmult’} \ uscs \\
out = \text{vfoldl} (+) 0 \ ws \\
\]

---

**Description**

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 Bank.

As shown in Figure 3, the FIR filter structure consists of an input, a set of filter coefficients, and an output. The filter coefficients are passed pairwise to the elements of the two lists, and there is no need for special notation. Also, dependencies and no sequential ordering. Therefore, the description is implicitly parallel, and there is no need for special notation. Also, dependencies and no sequential ordering. Therefore, the description is implicitly parallel and there is no need for special notation.
Describing the PFB  

FIR filter: Haskell → CλaSH

\[
\text{fir} :: [\text{Double}] \rightarrow (\text{State} [\text{Double}]) \rightarrow \text{Double} \rightarrow (\text{State} [\text{Double}], \text{Double})
\]

\[
\text{fir} \ c s \quad (\text{State} \ u s) \quad \text{inp} \quad = \quad (\text{State} \ u s, \text{out})
\]

where

\[
\begin{align*}
us &= \text{inp} \gg \text{us} \\
ws &= \text{vzipWith} (\ast) \ u s c s \\
\text{out} &= \text{foldl} (+) 0 \ w s
\end{align*}
\]

\[
\text{fir} :: (\text{Vector} \ D16 \ S) \rightarrow (\text{State} (\text{Vector} \ D16 \ S)) \rightarrow S \rightarrow (\text{State} (\text{Vector} \ D16 \ S), S)
\]

\[
\text{fir} \ c s \quad (\text{State} \ u s) \quad \text{inp} \quad = \quad (\text{State} \ u s, \text{out})
\]

where

\[
\begin{align*}
us &= \text{inp} \gg \text{us} \\
ws &= \text{vzipWith} \ 'fpmult' \ u s c s \\
\text{out} &= \text{wfoldl} (+) 0 \ w s
\end{align*}
\]
Describing the PFB  

FIR filter: Haskell → CλaSH

\[
\text{fir :: } \text{Double} \rightarrow (\text{State } \text{Double}) \rightarrow \text{Double} \rightarrow (\text{State } \text{Double}, \text{Double})
\]

\[
\text{where}
\]

\[
\begin{align*}
us &= \text{inp} \gg\gg \text{us} \\
ws &= v\text{zipWith} (\ast) \text{uscs} \\
\text{out} &= \text{foldl} (+) 0 \text{ws}
\end{align*}
\]

\[
\text{fir :: } (\text{Vector } D16 \text{ S}) \rightarrow (\text{State } (\text{Vector } D16 \text{ S})) \rightarrow \text{S} \rightarrow (\text{State } (\text{Vector } D16 \text{ S}), \text{S})
\]

\[
\text{where}
\]

\[
\begin{align*}
us &= \text{inp} \gg\gg \text{us} \\
ws &= \text{vzipWith} \text{fpmult} \text{uscs} \\
\text{out} &= \text{vfoldl} (+) 0 \text{ws}
\end{align*}
\]
The FFT pipeline is built according to the same procedure followed for the Polyphase Filter, except 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 C\texttt{aSH} are changed such that hardware can be generated with C\texttt{aSH}.

### 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](image)

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](image)

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 $out$.

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).

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 consists 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.

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 parameter).
Describing the PFB

• FFT is implemented using pipeline

• Two types of butterflies and Complex multiplier
Describing the PFB

The FFT pipeline is based on [10] and utilizes a radix-2 algorithm which requires the same application. As can be seen in Figure 6, the size of the memory, used for intermediate results, 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 C since the length is encoded in aSH compiler doesn’t accept that the FFT is only shown with parallelization factor 8.

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 consists of a counter and one with an additional multiplier. Implementing the architecture of Figure 8 in Haskell is now straightforward (Listing 9).

Listing 8 shows the implementation as it is written in Haskell.

Describing the PFB
Describing the PFB

The FFT pipeline is based on [10] and utilizes a radix 2 algorithm which requires the same butterfly structure of a radix 4 algorithm but has the same butterfly operation in complex numbers. As can be seen in Listing 8, the implementation is straightforward (Listing 9). Specifying the architecture from Figure 7 in Haskell is easy since Haskell supports composition of functions. By combining the complex multiplier, BF2I and BF2II into a single function, a basic instance of the FFT is implemented in Haskell as shown in Listing 10.

### 2.2.1. Haskell description

Implementing the architecture of Figure 8 in Haskell is now straightforward (Listing 9). The first butterfly operation is implemented in Haskell as follows:

```hs
bf1state, bs1state = 
bf2state, bs2state = 
```

where

- `bf2state = cmult ws cntr inp`
- `bf1state = cmult ws cntr inp`
- `ws1 = ws * w`
- `cntr' = (cntr + 1) \ mod \ n`
- `w = ws ! cntr`
- `out = inp * w`

### 2.2.2. Translation to C

Listing 11

```hs
basic building block of the FFT
cmult ws cntr inp = (cntr', out)
where
n = length ws
|cntr' = (cntr + 1) 'mod' n
w = ws ! cntr
out = inp * w
```
Describing the PFB

fftb \( w_s \) \( (bf1state, bf2state, cmstate) \) \( inp = ((bf1state', bf2state', cmstate'), out) \)

where

\[
\begin{align*}
(bf1state', a) &= bf2i \ bflstate \ inp \\
(bf2state', b) &= bf2ii \ bf2state \ a \\
(cmstate', out) &= cmult \ w_s \ cmstate \ b
\end{align*}
\]
Describing the PFB

**fftchain** \((ws1, ws2, \ldots) (bb1state, bb2state, \ldots)\) \(inp = ((bb1state', bb2state', \ldots),\ out)\)

where

\[
(bb1state', d1) = \text{fftbb} \ ws1 \ bb1state \ inp \\
(bb2state', d2) = \text{fftbb} \ ws2 \ bb2state \ d1
\]

\[
(bbNstate', \ out) = \text{fftbb} \ wsN \ bbNstate \ d9
\]
Describing the PFB

**FFT BF2I: Haskell → CλaSH**

\[
\text{bf2i} \ (cntr, lst) \ \text{inp} = ((cntr', lst'), \ \text{out})
\]

**where**

\[
\begin{align*}
\ n &= \text{length} \ lst \\
\ cntr' &= (cntr + 1) \mod n \\
\ lst' &= \text{lstin} \rightarrow \rightarrow lst \\
\ (out, \ \text{lstin}) &= \text{if} \ cntr \geq n \\
\ &\quad \text{then} \ (\text{lstout} + \ \text{inp}, \ \text{lstout} - \ \text{inp}) \\
\ &\quad \text{else} \ (\text{lstout}, \ \text{inp}) \\
\ \text{lstout} &= \text{last} \ lst
\end{align*}
\]

\[
\text{bf2i\_clash} \ (cntr, lst) \ \text{inp} = ((cntr', lst'), \ \text{out})
\]

**where**

\[
\begin{align*}
\ n &= \text{vlength} \ lst \\
\ cntr' &= \text{cntr} + 1 \\
\ lst' &= \text{lstin} \rightarrow \rightarrow lst \\
\ (out, \ \text{lstin}) &= \text{if} \ cntr \geq n \\
\ &\quad \text{then} \ (\text{lstout} + \ \text{inp}, \ \text{lstout} - \ \text{inp}) \\
\ &\quad \text{else} \ (\text{lstout}, \ \text{inp}) \\
\ \text{lstout} &= \text{vlast} \ lst
\end{align*}
\]
Describing the PFB

Therefore, we have chosen to describe the FFT chain in Haskell by separately defining each problem for lists in Haskell, it will be for vectors in C. As can be seen in Figure 6, the size of the memory, used for intermediate results support recursion (yet). Furthermore, the length of the FFT is fixed in the previous sections, the FFT chain consists of a set of basic building blocks chained building block.

Listing 11 shows how the aforementioned components are chained in the basic argument). Listing 10 shows how the aforementioned components are chained in the basic

bf2ii (cntr, lst) inp = ((cntr', lst'), out)
where
\[n = \text{length } lst\]
\[cntr' = (cntr + 1) \mod n\]
\[lst' = \text{lstin } \rightarrow \text{lst}\]
\[(\text{out, lstin}) = \text{if } cntr \geq n\]
\[\text{then (lstout } + \text{inp, lstout } - \text{inp)}\]
\[\text{else (lstout, inp)}\]
\[\text{lstout } = \text{last lst}\]

bf2i _clash (cntr, lst) inp = ((cntr', lst'), out)
where
\[n = \text{vlength } lst\]
\[cntr' = cntr + 1\]
\[lst' = \text{lstin } \rightarrow \text{lst}\]
\[(\text{out, lstin}) = \text{if } cntr \geq n\]
\[\text{then (lstout } + \text{inp, lstout } - \text{inp)}\]
\[\text{else (lstout, inp)}\]
\[\text{lstout } = \text{vlast lst}\]
Describing the PFB

Therefore, we have chosen to describe the FFT chain in Haskell by separately defining each
the type resulting in a different type depending on the position of the butterfly in the chain.
in the butterfly, is different depending on the position in the chain. Although this is not a
support recursion (yet). Furthermore, the length of the FFT is fixed
together. This could be written down using recursion, however, the C
in the previous sections, the FFT chain consists of a set of basic building blocks chained
argument). Listing 11 shows how the aforementioned components are chained in the basic

Listing 9

bf2i (cntr, lst) inp = ((cntr', lst'), out)

where
n = length lst

\underline{cntr'} = (cntr + 1) 'mod' n

\underline{lst'} = lstin ++ lst

(out, lstin) = if cntr \geq n

\underline{then} (lstout + inp, lstout - inp)

\underline{else} (lstout , inp )

lstout = last lst

Listing 10

bf2i_clash (cntr, lst) inp = ((cntr', lst'), out)

where
n = vlength lst

\underline{cntr'} = cntr + 1

\underline{lst'} = lstin ++ lst

(out, lstin) = if cntr \geq n

\underline{then} (lstout + inp, lstout - inp)

\underline{else} (lstout , inp )

lstout = vlast lst
Results

- Polyphase filter bank has been fully implemented using \texttt{C}\texttt{laSH}
- Simulation shows that the PFB operates correctly
- Synthesis revealed some limitations of the current compiler

<table>
<thead>
<tr>
<th></th>
<th>Polyphase filter (256 elements)</th>
<th>1k-points FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic utilization</td>
<td>91%</td>
<td>6%</td>
</tr>
<tr>
<td>blockRAMS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DSP blocks</td>
<td>128</td>
<td>70</td>
</tr>
<tr>
<td>Max. (F_{clk})</td>
<td>114 MHz</td>
<td>195 MHz</td>
</tr>
</tbody>
</table>
Conclusions

- The complete Polyphase Filter Bank has been implemented
- Haskell code needs only small modifications before it is accepted by the CλaSH compiler
- The description is purely parallel (structural) and cycle accurate
- Shortcomings of CλaSH compiler
  - Large coefficient vectors not supported
  - BlockRAM not supported, limiting $F_{clk}$
Future Work

✦ Develop area vs time time trade off based on functional description
✦ Improvements for the ClaSH compiler
✦ Support for blockRAMs on FPGA
✦ Support for memory initialization files for coefficient vectors
Questions ?