2012

Specification of APERTIF Polyphase Filter Bank in Cλash

Master's Thesis
of
Dimitrios Sarakiotis

Computer Architecture for Embedded Systems (CAES)
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Department of Electrical Engineering

Examination Committee
Prof.dr.ir. Gerard Smit
Dr.ir. Jan Kuper
Rinse Wester, MSc.
Ir. Eric Kooistra
Ir. André Gunst
Dr.ir. Jan Broenink

University of Twente
22 May 2012

Abstract

The complexity of nowadays architectures has made the traditional hardware description languages inadequate and the need of having a more abstract description of the hardware a necessity. Therefore the exploit of any possible high level abstraction mechanism is more than just a luxury but has become necessary.

Functional hardware description languages are a class of languages specialized in hardware descriptions which exploit the ability to express higher level structural properties, such as parameterization and regularity. Due to features as higher-order functions and polymorphism, parameterization in functional hardware description languages becomes easier than in other hardware description languages, like VHDL.

In the CAES group at Twente University a new functional hardware description language, Cλash, has been developed. Cλash borrows both the syntax and semantics from the general-purpose functional programming language Haskell.

In this Thesis the APERTIF Polyphase Filter Bank, which has been used at the LOFAR telescope, is being attempted to be implemented using Cλash. The Filter Bank as part of the Beam-Former of the LOFAR telescope serves as a spectrum analyzer. It mainly consists of a FIR Pre-Filter Structure for complex inputs and a complex FFT. The particular Filter Bank has been chosen due to its complexity and its subcomponents which provide a suitable case study for the exploit of high level abstraction in hardware designing procedures and the use of functional HDLs.

The primary goal of this Thesis is to investigate whether or not Cλash, as a newly developed language, can be used to specify and describe complex architectures, like an FFT, a Filter Bank etc.

As a second goal, a list with the most important attributes that can be improved, so that Cλash will evolve in a more dependable and adequate HDL which can be used in more complicated and demanding designs, should be derived.

Additionally, if time allows, a comparison should be performed between the traditional design approach, which is being followed by the more common HDL languages (VHDL, Verilog), and the functional approach, which is being followed by the functional HDLs (Cλash, Lava).

Acknowledgments

First of all I would like to express my gratitude to my thesis project supervisors Dr. Ir. Jan Kuper and Rinse Wester (MSc) from the University of Twente and CAES group and Ir. Eric Kooistra and Ir. André Gunst from ASTRON, for the valuable time that they have dedicated to this project and to me personally. Their continued interest in this project, their guidance and their insightful comments, made the completion of this thesis project possible. Also I would like to thank the whole CAES group for their help whenever I asked it for.
Finally, I would like to thank from the bottoms of my heart, my family, for their moral, emotional and material support throughout this whole period of my Master of Science studies.

Table of Contents

List of Abbreviations

- **APERTIF** : **APER**ture **TI**le **I**n **F**ocus
- **ADC** : **A**nalogue to **D**igital **C**onvertor
- **CFA** : **C**ommon **F**actor **A**lgorithm
- **DFT** : **D**iscrete **F**ourier **T**ransformation
- **DSP** : **D**igital **S**ignal **P**rocessing
- **FIR** : **F**inite **I**mpulse **R**esponse
- **FFT** : **F**ast **F**ourier **T**ransformation
- **FPGA** : **F**ield **P**rogrammable **G**ate **A**rray
- **HDL** : **H**ardware **D**escription **L**anguage
- **LFSR** : **L**inear **F**eedback **S**hift **R**egister
- **LOFAR** : **LO**w **F**requency **A**rray
- **I/O** : **I**nput/**O**utput
- **PFB** : **P**olyphase **F**ilter **B**ank
- **PFS** : **P**re-**F**ilter **S**tructure
- **R2^{2}SDF** : **R**adix-**2**^{2} **S**ingle **D**elay **F**eedback
- **SKA** : **S**quare **K**ilometer **A**rray
- **WSRT** : **W**esterbork **S**ynthesis **R**adio **T**elescope

Chapter 1 – Introduction

The latest project of global radio-astronomy community is the Square Kilometer Array (SKA) telescope. The major improvement that SKA telescope promises is that the field of vision will be much larger than what is possible with any other radio-telescope until now. For increasing the field of view of the telescope there are several ways, one of them is the APERTIF project. APERTIF is a project which explores one of the technologies to improve the field of vision of radio-telescopes. In more detail it aims to increase the field of vision of the Westerbork Synthesis Radio Telescope (WSRT) [15].
For achieving a large field of vision whether we have to construct a huge parabolic radio telescope or construct an array of regular size radio telescopes. For combining all the received signals to one, as it would be if we had a single telescope, the time delay between the telescopes of the array need to be compensated. Therefore beamforming becomes necessary.
But for performing beamforming in an optimum way the incoming signal needs to be narrow band, this is the so-called "narrow band assumption". Therefore for satisfying this assumption the incoming signal needs to be split in narrow band signals and that is done with the APERTIF Polyphase Filter Bank. The PFB plays an essential role at the DSP structure of a telescope and allows beamforming to be applied in an optimum way.
The complexity of such a Filter Bank makes almost necessary to move from the traditional hardware description languages to a new language, which will help to reach a higher level of abstraction. This new language is Cλash, a newly developed functional HDL.

1.1) Cλash

Hardware Description Languages (HDL) are languages specially designed for formal descriptions of digital and electronic circuits. A special class of HDLs which prioritize abstraction and parameterizations is the so-called functional HDLs.
Functional HDLs emphasize on that ability, to express higher level structural properties, such a parameterization and regularity. Due to features as higher-order functions and polymorphism, parameterization in functional hardware description languages is more natural than the parameterization support found in the more traditional hardware description languages, like VHDL and Verilog. Cλash is a newly developed functional hardware description language, which borrows both the syntax and semantics from the general-purpose functional programming language Haskell. Furthermore an extra feature of Cλash is that it is able not only to specify and simulate hardware, but also generate the actual hardware from that description. [4]

1.2) Beamforming

In astronomy having telescopes with large field of view are essential and the same goes for radio-astronomy. For increasing this field of view large physical parabolic telescopes has to be used, but the construction of such telescopes is limited. For further increasing the field of view arrays of telescopes are used.
According to this concept an array of antennas, or in our case parabolic telescopes, is combined to create a larger antenna, which otherwise would be impossible to physical construct it. For combining multiple antennas into one single beamforming is needed.
Beamforming is a general signal processing technique used to control the directivity of the reception or transmission of a signal on a transducer array. [5]
In APERTIF project the so-called *phase-shift* beamforming is used, where phase shifts are used to simulate time delays and therefore to compensate for the difference in the arrival of the signal at each antenna.
The phase shift that it is required for each antenna of the array is being calculated from the following schematic, which shows the time delay that needs to be achieved at each antenna.

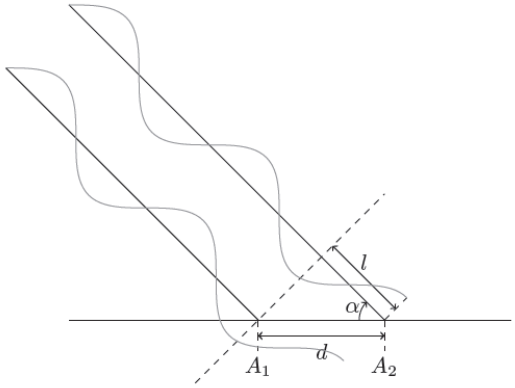

Figure 1: beamforming delay for two consecutive antennas (A1,A2) [5]

It has to be pointed out that beamforming gives better results for narrow band signals. A narrow band signal is defined as:

$$BW = \frac{F_H - F_L}{(F_H + F_L)/2} \leq 1\% \qquad (1.1)$$

where BW is the bandwidth of the narrow band signal.

Therefore before the beamformer a filter bank is placed so that the initial signal to be split in narrow subbands. In the case of the APERTIF PFB that splitting to narrow subbands is performed from an FFT.

1.3) APERTIF Filter Bank Description

At this Thesis the implementation of APERTIF Polyphase Filter Bank in Cλash, which has been used at the LOFAR telescope, will be discussed.
ASTRON Netherlands Institute for Radio Astronomy has designed and built the LOw Frequency Array (LOFAR). This telescope is able to observe at very low radio frequencies (10-240 MHz). It is an important scientific and technological pathfinder for the next generation of radio telescopes, like the Square Kilometer Array (SKA). LOFAR was developed by a consortium of knowledge institutes, universities and industrial parties, led by ASTRON [14].

The APERTIF PFB will serve as a spectrum analyzer and is located before the beamforming structure of the LOFAR telescope. In more detail it is placed between the ADC, which transforms the analogue input to digital, and the actual beamforming structure. Its main purpose is to split the input spectrum into smaller subbands, so that the narrow-bandwidth assumption is satisfied and beamforming becomes possible.

Figure 2: Signal processing chain of LOFAR telescope

LOFAR telescope is actually an array of antennas and beamforming needs to be performed at the signal of every antenna, therefore for each antenna an ADC and a filter bank is required and every filter bank has to be interconnected with every beamformer of the system. An overview of the LOFAR signal processing system is shown in Figure 3,

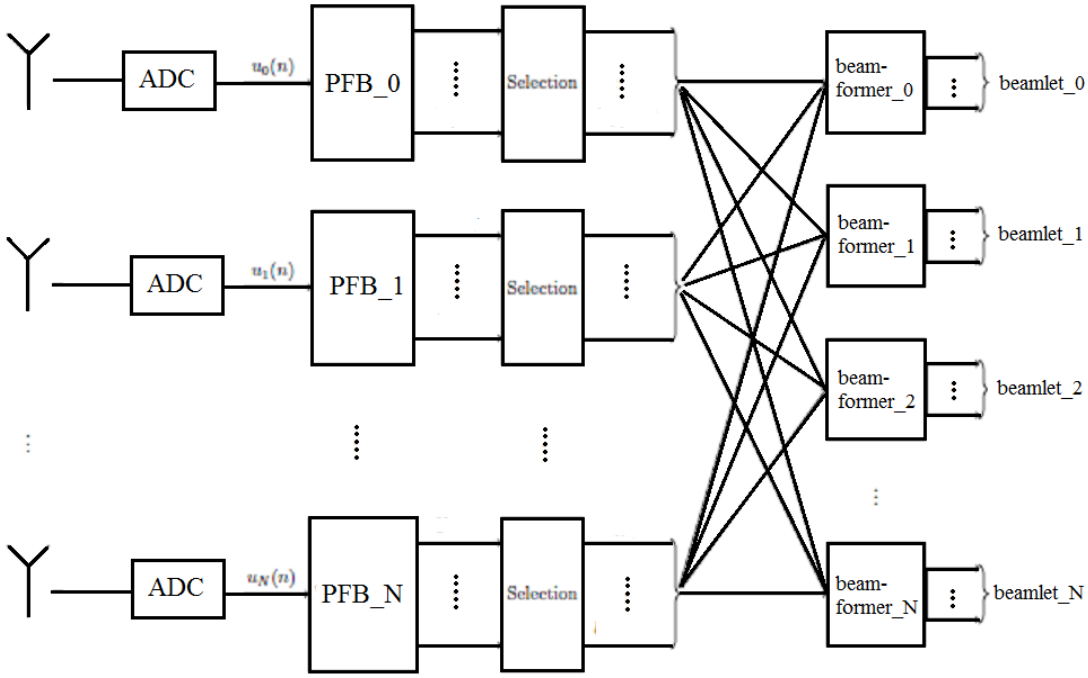

Figure 3: LOFAR telescope digital signal processing system with beamformers [ref]

From Figure 3 it can be deduced that the PFB is a separate system from the beamformer and each PFB needs to be interconnected with all the available beamformers.
The PFB which is going to be implemented can be seen in Figure 4

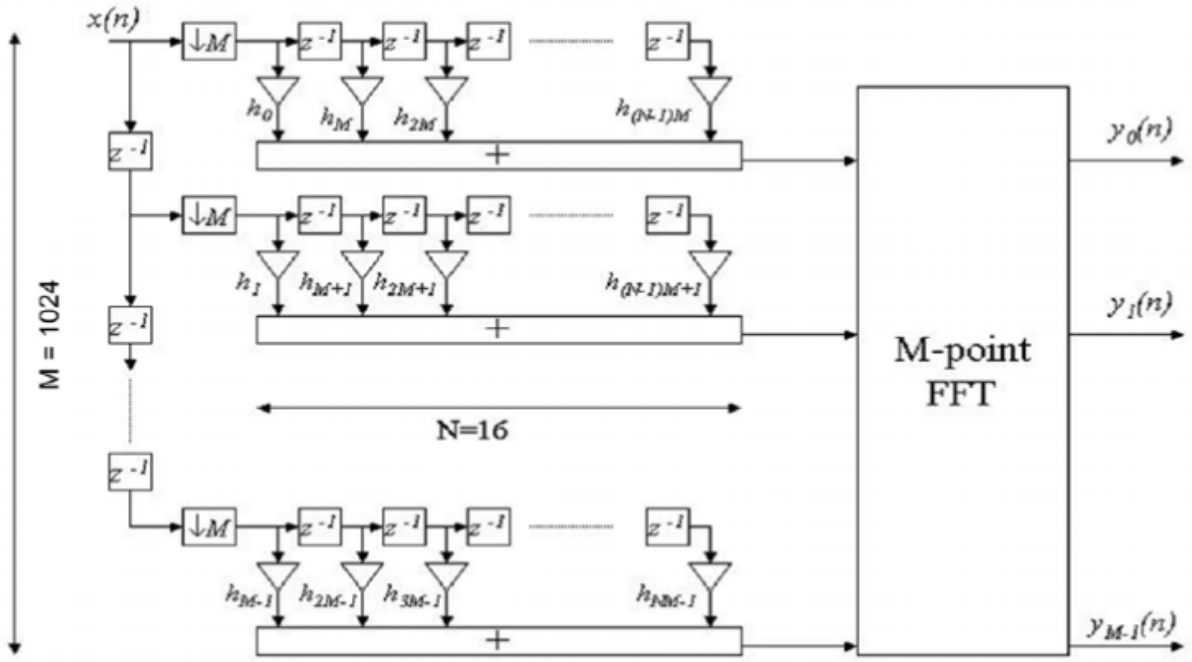

Figure 4: schematic of APERTIF Polyphase Filter Bank [ref]

The PFB consists of 2 parts, the Pre-Filter Structure (PFS) and the FFT. The specifications set from ASTRON for the whole PFB [13] are,

- The PFB should be able to fit in a single FPGA
- The PFB must be synthesized for the Stratix IV FPGA (type EP4SGX230KF40C2) using the Quartus II synthesis tool
- The PFB must be able to process 800 Msps using a clock frequency of 200 MHz
- The PFS is consisted from 1024 16-tap FIR-filters using 18-bit numbers
- The FFT must be an 1024-points wideband complex FFT

1.4) Thesis Goals and Research Questions

Cλash as a newly developed language isn't by any mean a dependable and complete language and of course it can accept a series of optimizations and fixes. Furthermore, Cλash has only been tested on a few cases. Therefore, the primarily goal of this project is to determine whether or not Cλash can be used to specify complex architectures like a Filter Bank. Furthermore since the specific architecture of the Polyphase Filter Bank has been implemented in VHDL directly, the resulted VHDL description from Cλash can be used to compare those two descriptions and draw conclusions about the optimality of the description and how easy was to produce each description.
Secondary, this thesis serves as some sort of debugger of Cλash, since the resulted description and the whole procedure which has been followed can highlight most of the bugs and rough-ends of the Cλash. These conclusions can be used to improve Cλash, so that it will become a more adequate and complete HDL.
To conclude the research questions which are attempted to be answered by such a Thesis are:

- Can Cλash, in the state that it is now, describe complex architectures?
- Which are these attributes of Cλash and how they need to be changed so that Cλash will evolve in a more adequate and complete HDL?

1.5) Overview

In this thesis we follow the structure of the implemented PFB. In chapter 2 the Pre-Filter Structure (PFS) is analyzed. We start with the Haskell implementation of the PFS and then we continue with the Cλash implementation and the synthesis results and finally conclusions are being drawn regarding the implementation.
In Chapter 3 the same structure is followed for the FFT. First the Haskell implementation is analyzed, followed by the Cλash implementation and finally the synthesis and performance results are being discussed. At the end of Chapter 3 once again conclusions about the resulting Cλash and VHDL description are being drawn.
Finally in Chapter 4 general conclusions about the implemented PFB and about Cλash are drawn and suggestions to improve Cλash as a functional HDL are proposed.

Chapter 2 – Pre-Filter Structure for APERTIF filter bank

2.1) Introduction

The Polyphase Filter Bank will be used in the APERTIF beam-former and its main purpose is to perform spectrum analysis. As part of the beam-former the input signal needs to be split in several subbands so that the narrow-bandwidth assumption is satisfied and thus beamforming is possible.
The Pre-Filter Structure is the first part of that PFB structure and its goal is to split the input spectrum into M number of subbands. For that reason it is being constructed by a set of 1024 16-tap bandpass FIR-filters.

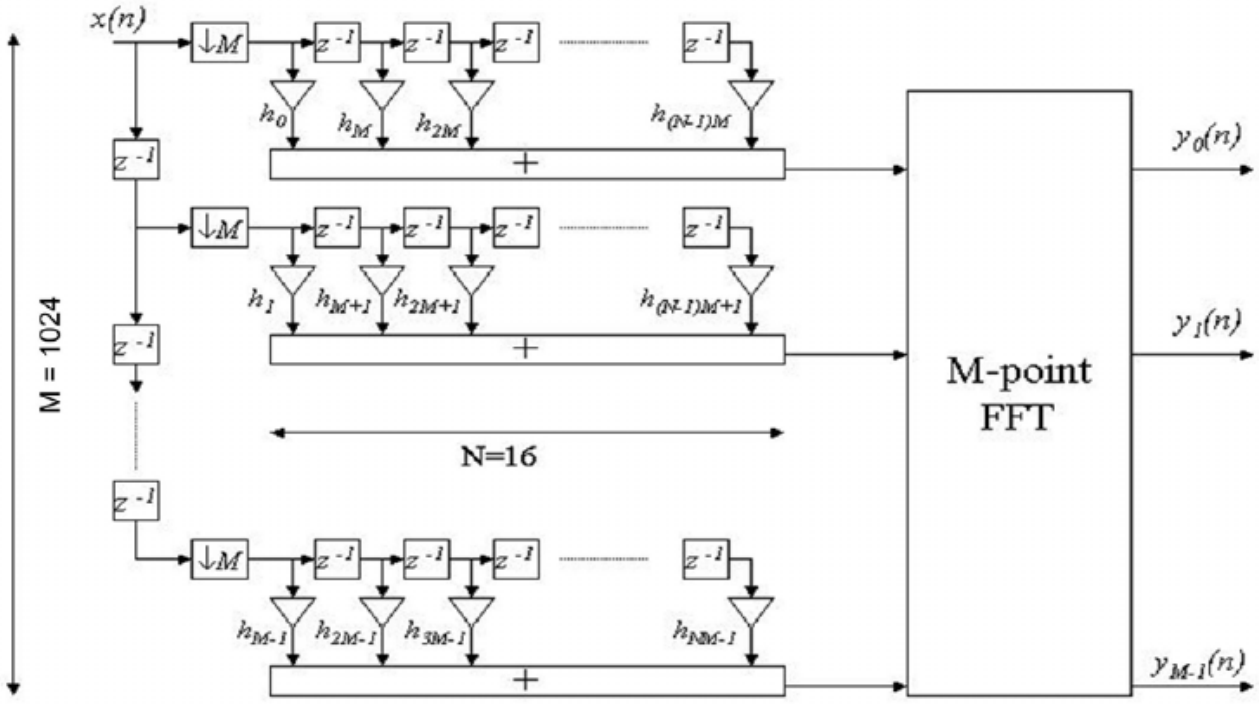

Figure 5: Polyphase Filter Bank (PFB) used in LOFAR [13]

Before starting with the analysis of the Pre-Filter Structure, which is going to be implemented as part of the Polyphase Filter Bank (PFB), it is better to start analyzing the implementation of the components of the Pre-Filter Structure in a functional programming language (Haskell).

2.2) Pre-Filter Structure implemented in Haskell

In general, a Pre-Filter Structure for a Polyphase Filter Bank is being described by:

$$P(z) = \sum_{q=0}^{M-1} z^{-q} H_q(z^M) \tag{2.1}$$

Where $H(z^M)$ is the transfer function describing the FIR-filters which consist the filter bank. [2]
From equation (2.1) the resulting structure will be:

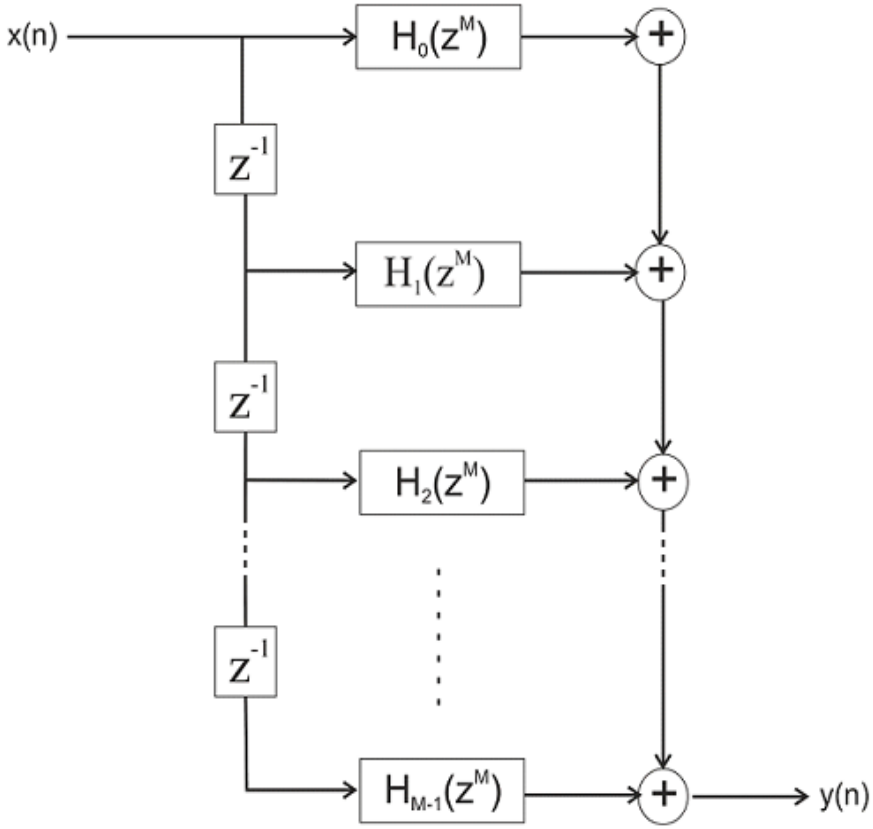

Figure 6: Schematic diagram of Pre-Filter Structure of M filters [2]

Regarding the APERTIF Polyphase Filter Bank, because the input sample rate and the FFT clock frequency is different, downsampling before the FFT block is necessary. The downsampling rate that is necessary is of a factor M. That M factor is determined by the size of the FFT. Since the PFB is intended to be used in beamforming applications then the size of the FFT has to be that so the narrow bandwidth assumption is satisfied. For our case the size of the FFT and thus the M factor has been set to **M = 1024.** [13]

The PFB structure with downsampling is being shown in Figure 7,

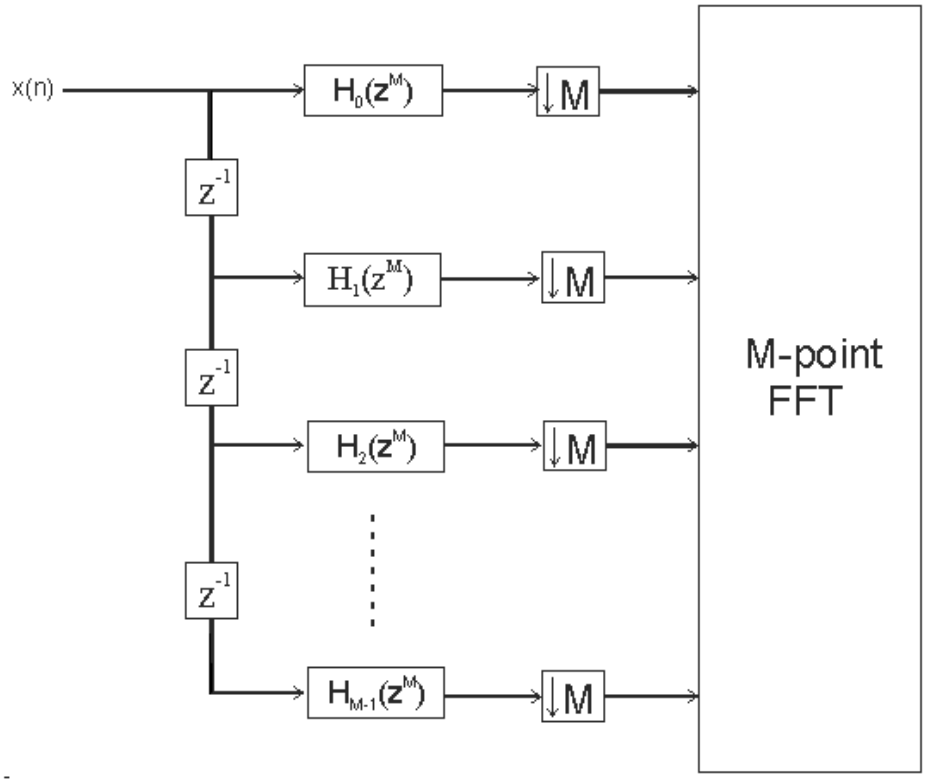

Figure 7: schematic of PFB with downsampling

It is being proved in [2] that downsampling can be moved in front of the filter with changing the filter's transfer function as it is being illustrated in Figure 8,

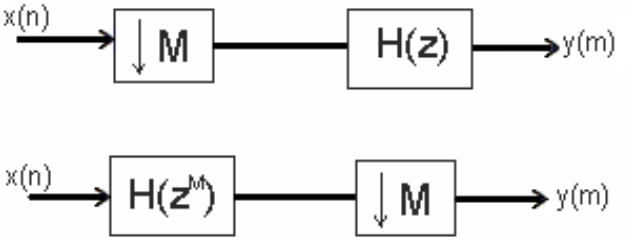

Figure 8: equivalent downsampling systems

For our case the first transformation is used since like that the hardware complexity of the FIR-filter is being reduced.

In more detail if the downsampling is performed after the FIR-filter, the transfer function of the filter has to be $H(z^M)$, which means that between two consecutive taps M registers are needed. Respectively if the downsampling is located in front of the FIR-filter then the transfer function of the filter will be H(z) and that means that only a single register is needed between two concequitive taps. So by placing the downsampling in front it reduces the required number of registers.

The combination of downsampling and a register, as it is being shown in Figure 5, is actually form a switch [2]. This means that in the Pre-Filter Structure only one filter will be active at every clock cycle. Figure 9 shows a schematic representation of the PFB structure with the set of registers and downsampling replaced by a switch,

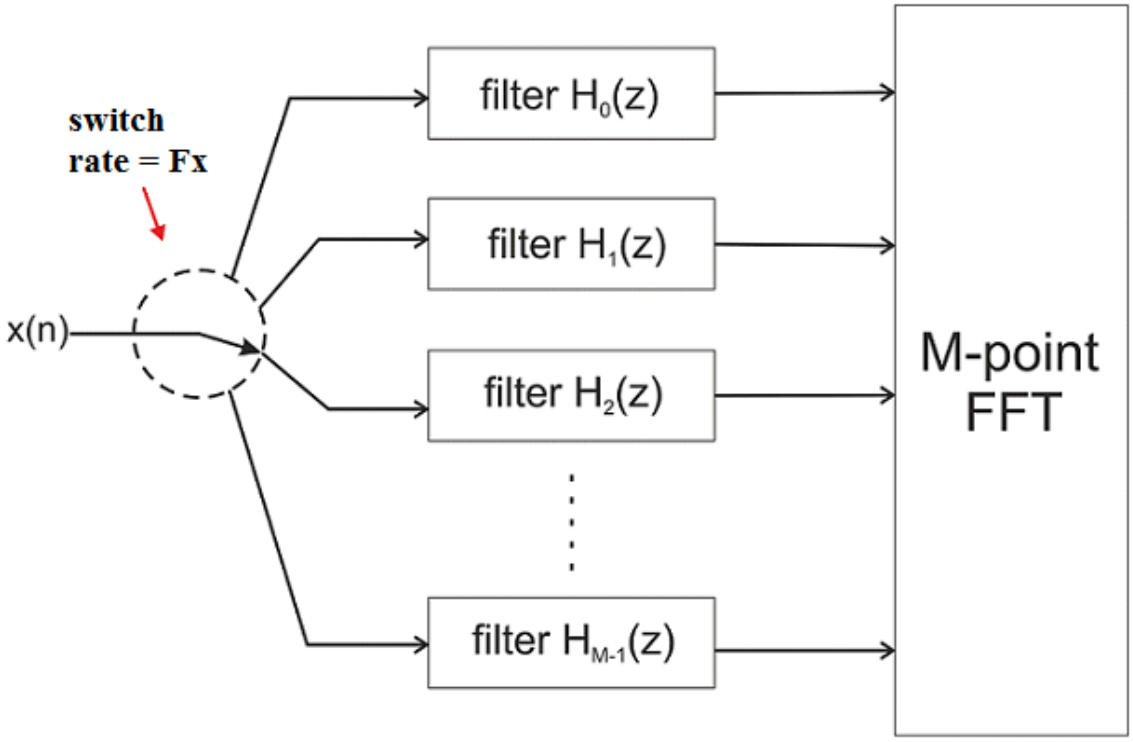

Figure 9: PFB structure with delay registers and downsampling replaced by a switch

Figure 9 shows that at every clock cycle only 1 FIR-filter will be active. Since the Pre-Filter Structure should be able to fit in one single FPGA and since only one of the filters is going to be active at every time, then all the M FIR-filters can be merged into a single filter. The states of that filter will be pulled out and that means that the states need to be supplied at every clock cycle. Furthermore at every clock cycle the proper coefficients needs to be supplied to the filter as extra input arguments.

Such a structure is being shown in Figure 10:

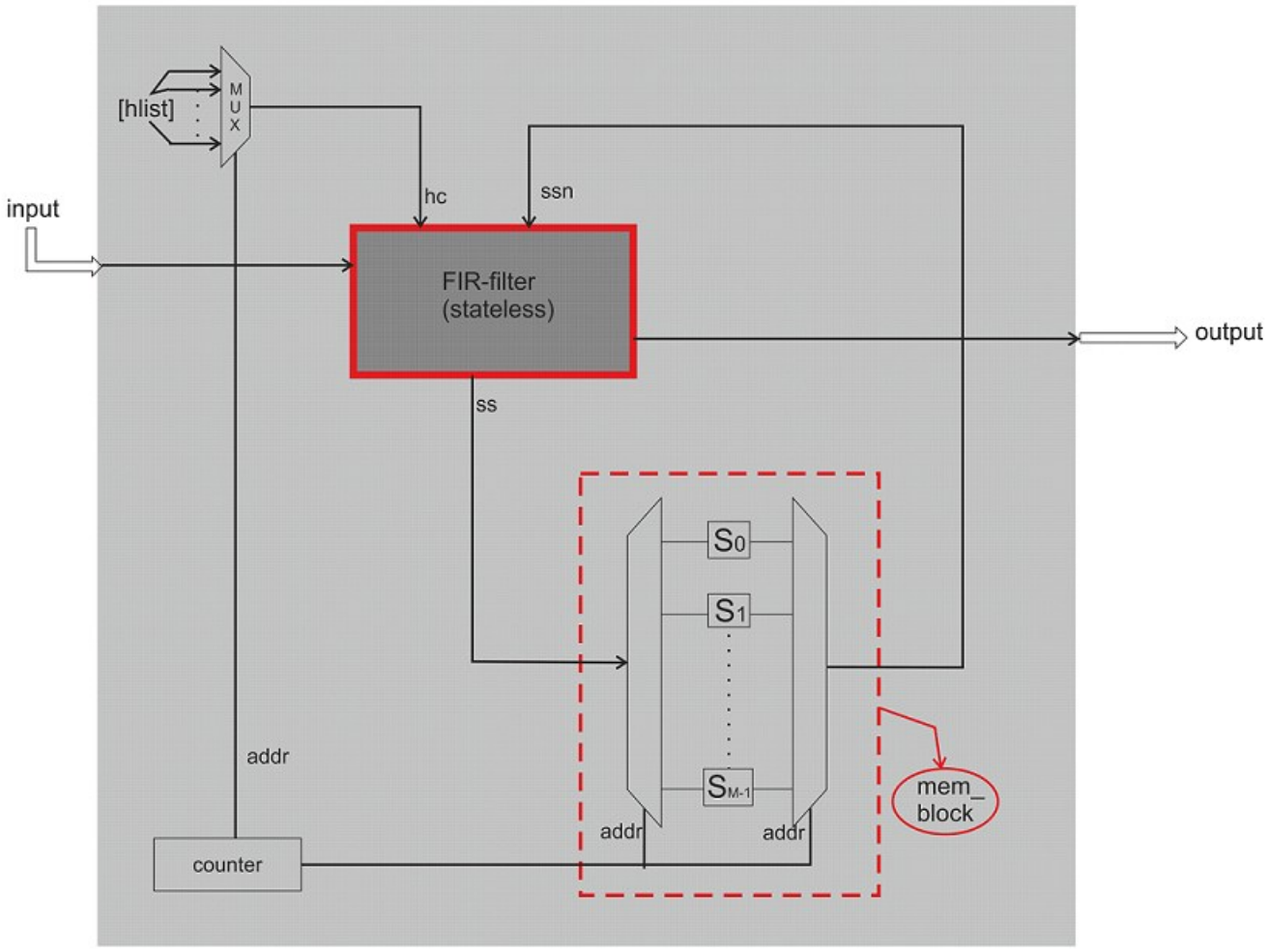

Figure 10: Block diagram of Pre-Filter Structure for the PFB

With such a structure instead of having M filters downloaded/build at the FPGA as part of the Pre-Filter Structure now we have only one. The coefficients are stored in a dedicated memory block and are being loaded in the filter through the multiplexer *(MUX)*. The states of the filter are stored and loaded from the memory block *(mem_block)*. That means that the states of the FIR-filter are being pulled out, transforming the FIR-filter to a stateless structure. In order to synchronize everything and make sure that the correct coefficients and the correct states are loaded at the correct time, a counter has been added which is responsible for selecting the proper states and set of coefficients

2.2.1) FIR-filter implemented in Haskell

In general, a FIR-filter is being described from the following difference equation,

$$y(n) = \sum_{k=0}^{L-1} h_k x(n-k) \tag{2.2}$$

or expressed in the z-domain

$$H(z) = \sum_{k=0}^{L-1} h_k z^{-k} \tag{2.3}$$

From equation (2) it is obvious that an FIR-filter will be composed by a set of delays and by a set of multipliers and adders. The number of delays, multiplications and additions is directly determined from the number of taps of the filter [2].

By using the z-expression of the FIR-filter we can derive the direct form realization. That form is being shown in Figure 11

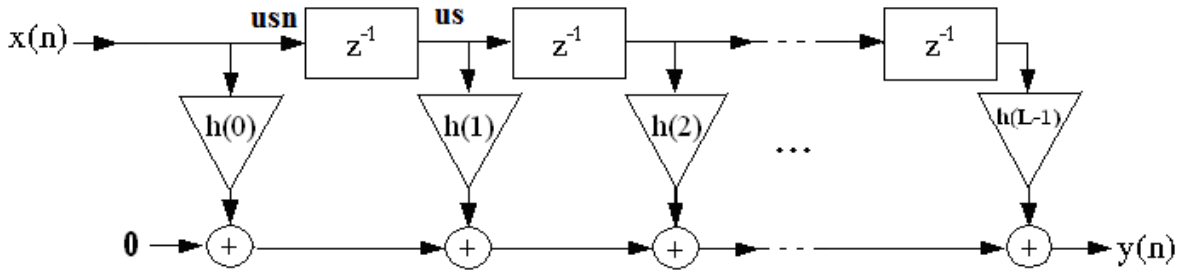

Figure 11: direct-form realization of an FIR-filter [2]

By checking Figure 11 it can deduced that the structure of an FIR-filter consists of a basic building block which has been replicated M times, where M is the number of the FIR coefficients or the number of taps of the FIR-filter.
The code for the implementation of the FIR-filter in a Haskell can be directly derived from the structure in Figure 11.
The resulting code is:

```
fir :: [Int] -> [Int] -> Int -> ([Int], Int)
fir hc us input = (usn, output)
                  where
                  usn = [input] ++ (init us)
                  m = zipWith (*) us hc
                  output = foldl (+) 0 m
```

Figure 12: Haskell code of FIR-filter (direct form)

A FIR-filter structure is a series combination of a single basic building block. Therefore the Haskell code describing a FIR-filter could be written using recursion. But, at least for the moment, Cλash is not supporting generic recursion and since it is intended to implement the PFS also to Cλash all the

functions need to be implemented in a form which is accepted from Cλash. Therefore in the end higher order functions are going to be used, which are supported from Cλash.
Also from Figure 12 it can be deduced that the states of the filter are considered to be a list and in every clock cycle that list is being recreated. For the output of the filter the "zipWith" and "foldl" functions have been used, which are standard functions for computations on lists. So in the end we can say that the FIR-filter has been implemented from a list's perspective since it is considered to be consisted from two lists (the states and the coefficients) and that way of implementing it is exactly the difference between VHDL and Haskell.

Apart from the direct form of the FIR-filter there is the transpose form. That form is being shown in Figure 13

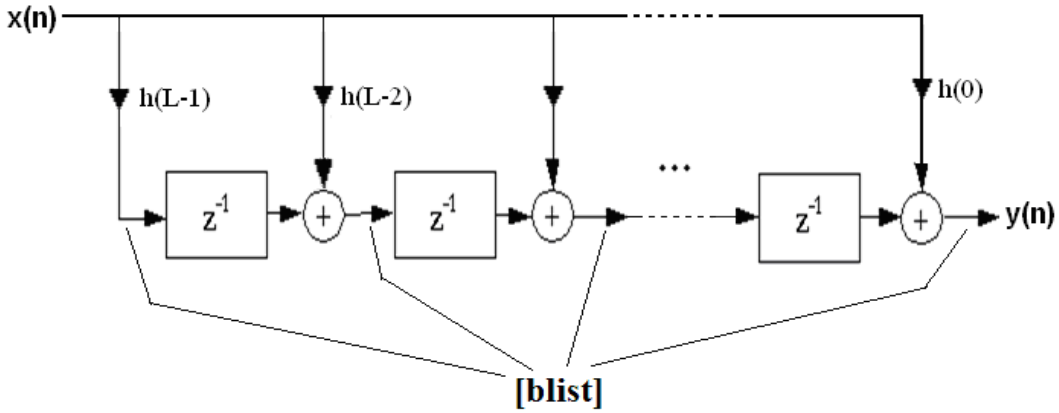

Figure 13: transpose form of a FIR-filter [18]

By comparing the two forms of FIR-filters (direct and transposed) it can been seen that the direct form contains a combinatorial path which goes through the first multiplier (h(0)) and all the adders. On the other hand at the transposed form the longest combinatorial path is limited to a multiplier and an adder. From that it can be concluded that the transposed form of the FIR-filter can achieve higher frequencies than the direct form.

By following the same procedure as before the resulting Haskell code for the transpose form is:

```
firT :: [Int] -> [Int] -> Int -> ([Int], Int)
firT hc us input = (usn, output)
                 where
                 ws = map (*input) hc
                 usn    = zipWith (+) ([0] ++ (init us)) ws
                 output = last us
```

Figure 14: Haskell code of transposed form of a FIR-filter

It has to be pointed out that for both forms the Haskell code has been derived directly from the schematic structure and not from the mathematical description. This is actually the usual procedure, from the mathematical description the schematic representation is derived and from that the Haskell code is derived.

2.2.2) Multiplexing function implemented in Haskell

A Multiplexer is simple to implement in a functional programming language, like Haskell. In Haskell there is a predefined operator, the **"!!"**, with which an element can be chosen from a list. That functionality is actually the function of a multiplexer. So a multiplexer in Haskell is implemented using the following line of code,

- output = input_list !! sel

where input_list: inputs of the multiplexer
sel: is the select signal/argument by which a specific input is being directed to the output

2.2.3) Counter function implemented in Haskell

A counter in functional programming can be implemented in several ways. For the particular case it has been chosen to be implemented by using modulo function so that the resulting code to be as generic as possible. The resulting Haskell code is,

- cntrN = cntr % length

the operator **"%"** is actually the modulo function and length is the size of the index that the counter will count.

2.2.4) Memory Block (mem_block) implemented in Haskell

From Figure 10 it can be deduced that the functionality of the memory block is to store the states of the active filter in a specific address and to load the proper states for the next active filter. That block can be seen as a combination of a de-multiplexer, a set of registers which form the actual memory and a multiplexer. A schematic of that combination is shown in Figure 15,

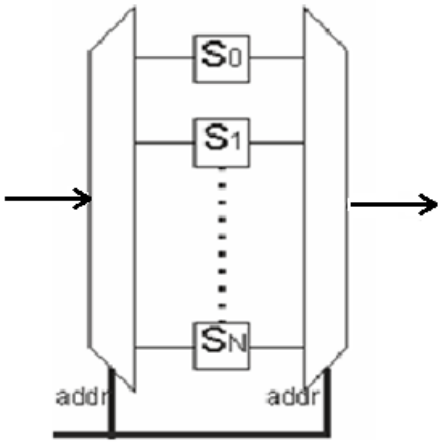

Figure 15: schematic representation of memory block used in PFS

This structure can also be considered from a functional point of view. That means that only the function which is performed from the block is being considered and not the components that will perform that function. From that point of view the function of mem_block is simply to write the input in one of the registers (S_0, S_1,…, S_N) and to output the contents of one of these registers according to the signal *addr*. So by viewing the memory block in that way the following Haskell code is being produced,

```
mem_block :: [[a]] -> ([a],Int) -> ([[a]],[a])
mem_block ss (ins,addr) = (ssn,outs)
                  where
                  outs = ss!!addr
                  ssn = first ++ [ins] ++ second
                  first = take addr ss
                  second = drop (addr + 1) ss
```

Figure 16: Memory Block for storing the FIR-filter's states implemented in Haskell

From the resulting code it can be deduced that the memory block is actually a list of lists and that at every clock cycle that list is being recreated.

2.2.5) Pre-Filter Structure implemented in Haskell

Now that all the basic blocks/functions have been implemented the actual implementation of the Pre-Filter Structure shown in Figure 10 is just a simple matter of interconnect these blocks. The Haskell code for that is,

```
prefilter :: (Int, [[a]]) -> a -> ((Int, [[a]]),a)
prefilter    (countstate,regstate) input = ((countstaten, regstaten), output)
               where
               countstaten = countstate % (length hlist)
               addr = countstate
               h = hlist !! addr
               (regstaten, ssn) = mem_block regstate (ss, addr)
               (ss, output) = firT h ssn input
```

Figure 17: Haskell code for Pre-Filter Function

The code of the PFS is derived directly from the structure itself [Figure 10]. Each of the components used in the structure are defined as a single function and all are combined in a higher order function to form the final Pre-Filter Structure.

2.2.6) Simulation Results

In order to test the Pre-Filter Structure and to show how the structure cycles through all the FIR-filters, a simulation of a filter bank with 4 FIR-filters of 4-tap each has been performed.
The results of that simulation are:

hlist = [[1,2,3,4],[5,6,7,8],[2,4,6,8],[3,5,7,9]]

Input = [1,1,1,1,1,1,1,1]

Output = [0, 0, 0, 0, ⟶ 1st iteration

1, 5, 2, 3 ⟶ 2nd iteration

3, 11, ...] ⟶ 3rd iteration

Normal FIR-filter

hlist = [[4,3,2,1],[8,7,6,5],[8,6,4,2],[9,7,5,3]]

Input = [1,1,1,1,1,1,1,1]

Output = [0, 0, 0, 0, ⟶ 1st iteration

1, 5, 2, 3 ⟶ 2nd iteration

3, 11, ...] ⟶ 3rd iteration

Traspose FIR-filter

In order to have the coefficients of the filters outputted from the PFS and like that to be able to check which filter is active in each clock cycle we have to determine the step response. That can be done by feeding a step input at the structure.
From the resulting outputs it becomes clear that after 4 clock cycles the first coefficient of the first filter is outputted and then the first coefficient of the 2nd filter and so on. That shows that the filters are becoming active consecutively and that only one filter is active at each clock cycle. Furthermore, from the results of the 3rd iteration it follows that the state of each filter is stored in mem_block and is loaded correctly.

2.2.7) Extension of Pre-Filter Structure

According to the specifications of the APERTIF filter bank firmware the PFB has to be able to process 800 Msps using a clock of 200 MHz [13]. That means that the rate of sample frequency divided by the processing frequency (P) will be 4. So by having only one active filter at each clock cycle that requirement cannot be met. In order to meet the requirements, the Pre-Filter Structure needs to be enhanced so that it will be able to have 4 filters active in parallel at every clock cycle.

The structure of the extended Pre-Filter Structure now becomes,

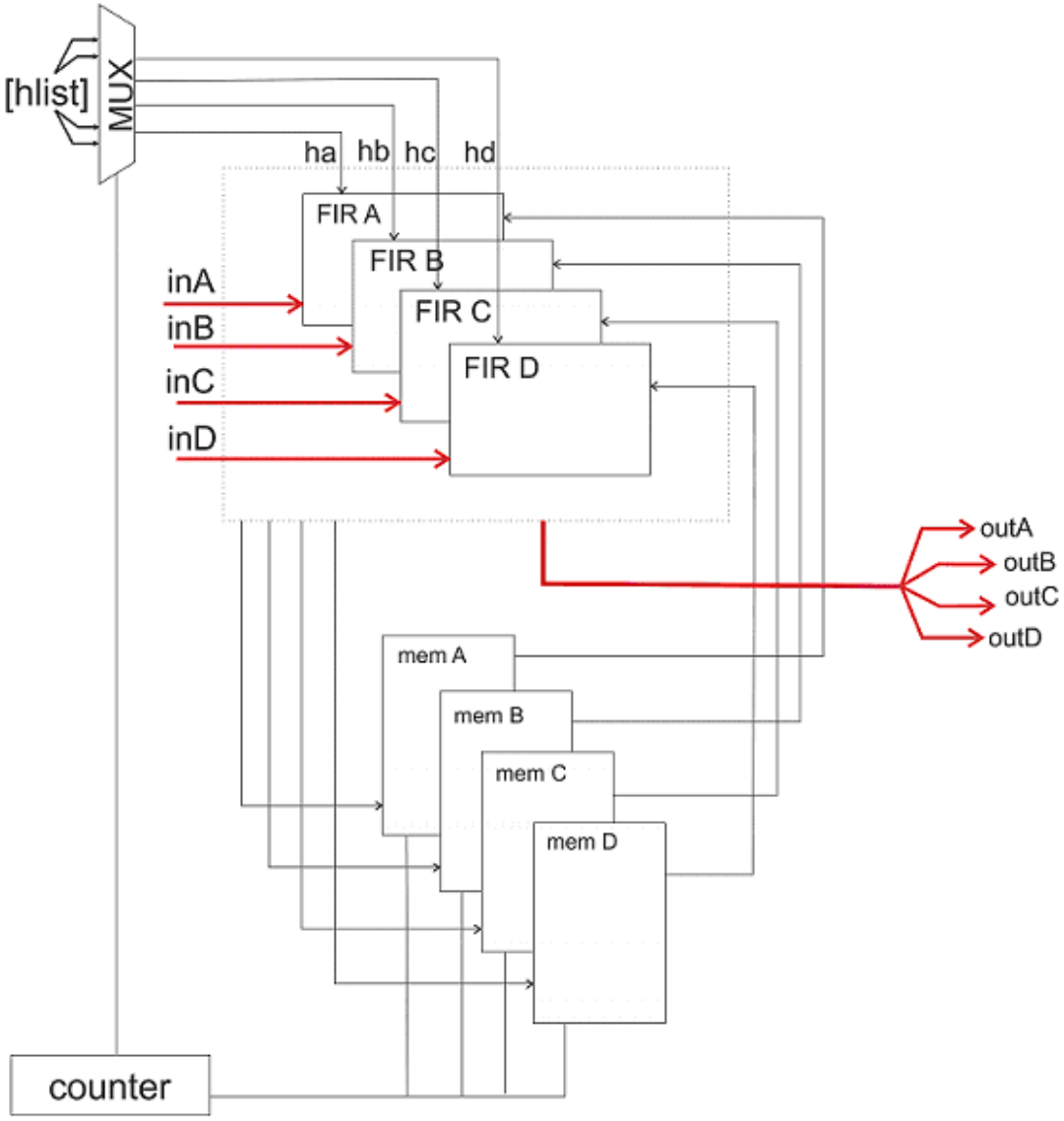

Figure 18: extended Pre-Filter Structure for the case of 4 active filters at each clock cycle

From Figure 18 it is clear that the new extended PFS is just a replication of the previous PFS with 1 FIR-filter and 1 mem_block. That means that the *FIR-filter* function and *mem_block* function doesn't change but the *mux* function has to change in order to give not only 1 output but 4, in other words an M-to-4 multiplexer needs to be implemented for the specific structure.

The Haskell code for that general M-to-N multiplexing function is:

```
mux_gen :: [[a]] -> (Int,Int) -> [[a]]
mux_gen    inlist (sel, p) = outs
              where
              bs = div (length inlist) p
              index = map (bs*) [0..p-1]
              hs = map (sel+) index
              outs = map (inlist!!) hs
```

Figure 19: Haskell code for an M-to-N multiplexer

The extra attribute that has been added to this multiplexing function is that an extra input argument (p) has been added. From this additional argument the number of outputs is determined and also the factor by which the input list (inlist) will be divided is determined.

As it has been mentioned already 4 filters needs to be active at each clock cycle. Therefore the whole list of filters will be divided in 4 sublists. The pattern of the active filters is predetermined and is the following:

- 1st clock cycle: FIR_0, FIR_1, FIR_2, FIR_3 → active
- 2nd clock cycle: FIR_4, FIR_5, FIR_6, FIR_7 → active
- 3rd clock cycle: FIR_8, FIR_9, FIR_10, FIR_11 → active
- …
- …
- 256th clock cycle: FIR_1020, FIR_1021, FIR_1022, FIR_1023 → active

As it will be explained later on that pattern is important for the FFT.

The final code for the extended Pre-Filter Structure of the PFB is:

```
-- basic building block of PFS --
---------------------------------
prefilter_bb :: (Firfilter) -> [[Int]] -> (Int,(Int,[Int])) -> ([[Int]], Int)
prefilter_bb ff regstate (inp,(addr,hc)) = (regstaten, outp)
                where
                (regstaten,ssn) = mem_block regstate (ss, addr)
                (ss, outp) = ff hc ssn inp

-- generalized architecture description of prefilter structure --
------------------------------------------------------------------
prefilter_ext2 :: (Firfilter) -> (Int, [[[Int]]]) -> ([Int], Int) -> ((Int, [[[Int]]]), ([Int]))
prefilter_ext2 ff (countstate, regss) (inps, p) = ((countstaten, regssn), (outps))
                  where
                  countstaten = count countstate (div (length hlist) p)
                  addr = countstate   -- addr: num
                  hc = mux_gen hlist (addr,p)  -- hc: list
                  addrlist = replicate (length hc) addr
                  bbinp = zip inps (zip addrlist hc)
                  list = zipWith (prefilter_bb ff) regss bbinp
                  regssn = map fst list
                  outps = map snd list
```

Figure 20: Haskell code of basic building block of the PFS (prefilter_bb) and final extended Pre-Fitler Structure (prefilter_ext2)

It has to be noted that the *FIR-filter* function and *the mem_block* function have remained the same and that they have been combined to create the *prefilter_bb* function and also that once again the Haskell code has been directly derived from the schematic.
Furthermore, the Pre-Filter Structure has been fully parameterized, meaning that for terms of simulation the user can decide how many and of which form (normal/transpose) FIR-filters will be active just by giving the proper input arguments and without any need to change the code itself.

2.3) Cλash Implementation of the Pre-Filter Structure

After the implementation of the PFS in Haskell the next step is to implement it also in Cλash. According to the initial specifications all data of the PFB have to be encoded as 18-bit Signed numbers [13]. Therefore the PFS will be consisted from 1024 16-tap FIR-filters using 18-bit Signed numbers.
In the specific implementation instead of having just 1 single 1024-to-4 MUX for providing the coefficients at the FIR-filters we decided to have 4 256-to-1 MUXs. That means that our idea for the final hardware is to have 4 separate RAMs dedicated for storing the FIR coefficients.
So the Cλash code for the 256-to-1 MUX is,

```
mux_256 :: WordSet256_16 -> Unsigned D8 -> WordLst16
mux_256    inlist sel = output
              where output = inlist ! (fromUnsigned sel)
```

Figure 21: Cλash code for a 256-to-1 Multiplexer

Comparing with the Haskell implementation of a MUX the only differences can be found in the data types of the input and output arguments. In Cλash, instead of using infinite lists, vectors of predefined length have to be used (WordSet256_16 and WordLst16). In more detail,
WordSet256 → Vector of 256 vectors
WordLst16 → Vector of 16 elements

From the FIR-filter structure it becomes obvious that a set of multiplications needs to be performed. As it has been point out already Cλash accepts only vectors of predefined length. Due to the way Cλash evaluates a multiplication,
In Cλash twice the number of bits is needed to perform a multiplication. For example for a multiplication of two 6 bit numbers in Cλash those numbers need to be represented as 12 bit numbers. So a multiplier in Cλash except from the multiplication unit also needs a set of resize and shifting blocks. A schematic of such a multiplier is being shown in Figure 22:

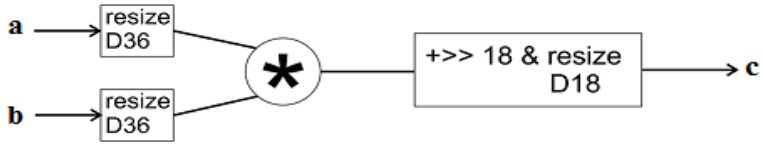

Figure 22: Schematic of an 18x18 multiplier implemented for Cλash

So according to that schematic the Cλash code of the multiplier used in the FIR-filter is,

```
fir_cmult :: Word -> Word -> Word
fir_cmult    a b = c
             where
             cr = ((resizeSigned a)::Signed D36) * ((resizeSigned b):: Word
             c = (resizeSigned (shiftR cr 18)) :: Word
```

Figure 23: Cλash code for multiplier used in the FIR-filter
***) The data type "Word" is nothing else than an 18-bit Signed type.**
The only reason for using the Word type is because it is easier to change the type in the code if there is a need to.

The memory block description where all the states of our Fir-filters will be stored in Cλash becomes much simplier.

```
mem_block :: WordSet128_16 -> (WordLst16, Unsigned D7) -> (WordSet128_16, WordLst16)
mem_block    ss (ins, sel) = (ssN, outs)
                where
                outs = ss ! (fromUnsigned sel)
                ssN = vreplace ss (fromUnsigned sel) ins
```

Figure 24: Cλash code of memory block of Pre-Filter Structure

We see that now the whole list of FIR states is not being recreated, as it was done for the Haskell implementation of *mem_block*, but instead the *vreplace* function is being used. With the specific function simply a specific element (vector in this case) is being replaced with a new one leaving intact the rest of the elements of the states vector.

The direct and the transposed form of the FIR-filter implementation in Cλash will be,

```
fir :: WordLst16 -> WordLst16 -> Word -> (WordLst16, Word)
fir    hc us input = (usN, output)
            where
            usN = input +>> us
            m = vzipWith (fir_cmult) us hc
            output = vfoldl (+) 0 m


firT :: WordLst16 -> WordLst16 -> Word -> (WordLst16, Word)
firT    hc us input = (usN, output)
             where
             ws = vmap (fir_cmult input) hc
             usN = vzipWith (+) (0 +>> us) ws
             output = vlast us
```

Figure 25: Cλash code for normal and transposed form of Fir-filter

From that we see that the FIR-filter implementation in Cλash remains the same with the one in Haskell, except the data types and the functions specialized in vectors (*vmap, vzipWith, vlast* etc.)

After transforming all the needed functions we can create the basic building block of our PFS consisting of a FIR-filter and a memory block for the FIR states.

```
prefilter_bb :: (Filt_type) -> WordSet128_16 -> (Word,(Unsigned D7,WordLst16)) -> (WordSet128_16, Word)
prefilter_bb    ff regstate (inp,(addr,hc)) = (regstaten, outp)
                where
                (regstaten,ssn) = mem_block regstate (ss, addr)
                (ss, outp) = ff hc ssn inp
```

Figure 26: basic building block of PFS consisting from a memory block and a FIR/FIRT filter

That basic building block is nothing more than a simple interconnection between the filter (FIR or FIRT) and the memory block.

So with the use of that building block the extended PFS consists of 4 of those building blocks and a counter which controls them. So the Cλash code for the extended PFS will be,

```
prefilter_ext    ff (cntr, (regs_a,regs_b,regs_c,regs_d)) ((inA,inB,inC,inD),(hc_a,hc_b,hc_c,hc_d))
                 = ((cntrN, (regsN_a,regsN_b,regsN_c,regsN_d)), (outA,outB,outC,outD))
                      where
                      cntrN = cntr + 1
                      (regsN_a, outA) = prefilter_bb ff regs_a (inA,(cntr,hc_a))
                      (regsN_b, outB) = prefilter_bb ff regs_b (inB,(cntr,hc_b))
                      (regsN_c, outC) = prefilter_bb ff regs_c (inC,(cntr,hc_c))
                      (regsN_d, outD) = prefilter_bb ff regs_d (inD,(cntr,hc_d))
```

Figure 27: Cλash code for the final extended Pre-Filter Structure

It is clear that the extended PFS (*prefilter_ext* function) accepts as inputs first the type of FIR-filters that will be used (normal or transposed), the actual inputs of the FIR-filters (inA,inB,inC,inD) and the FIR coefficients for each of the 4 active filters (hc_a,hc_b,hc_c,hc_d). That means that the coefficients are being provided at every clock cycle to the PFS but they are not provided externally. A separate subsystem with the 4 dedicated memories will be responsible for providing the proper set of coefficients at each time. Another way to view the Pre-Filter Structure is that it consists of 2 separate parts, the coefficient memories (*coef_struct*) and the extended prefilter (*prefilter_ext*).

2.4) Generation of VHDL code via Cλash

Since the Haskell code of the PFS has been transformed to Cλash code the next step is to use the "*:vhdl"* command to produce the synthesized VHDL code of the Pre-Filter Structure.
But by doing that we run into a problem, the Cλash compiler runs out of memory. That is happening due to the way that Cλash generates the VHDL code. In more detail before generates the VHDL description of any structure, Cλash transforms the initial Cλash description to an intermediate language and then it used that language to generate the VHDL code. During the translation to this intermediate language the vectors of our structure tends to become extremely big and therefore the compiler runs out of memory.
In order to acquire a hardware description of our structure so that we can get an idea about the space consumption and the frequency performance we have to use a trick to go around that problem.
There are mainly two tricks to go around that problem. One is to feed the coefficients externally. That means that the FIR coefficients will be stored in an external RAM or memory to the FPGA itself and it will be given to the Pre-Filter Structure as extra inputs at every clock cycle. But in that case the required I/O pins will exceed the available number of pins.
The second trick is to replace the coefficient structure with a set of Linear Feedback Shift Registers (LFSR) to produce a set of random coefficients at every clock cycle. An LFSR is a shift register with a special feedback circuit. That feedback circuit performs XOR operations on specific bits of the register and like that it forces the register to cycle through a set of unique states. So it is obvious that the

LFSRs can be used for generating pseudo-random numbers. Like that the 4 sets of FIR coefficients needed from the 4 active FIR-filters will be produced from these 4 sets of LFSRs.
In the specific case we will use the so-called *Fibonacci* implementation of an LFSR. According to which an LFSR can be described from a characteristic polynomial.
In our case we need one LFSR for every FIR coefficient and since these coefficients have to be coded as 18-bit numbers, which means that 18-bit LFSRs will be used.

The characteristic polynomial of an 18-bit LFSR is $x^{18} + x^{11} + 1$ with a period of **262143** states which means that it can produce 262143 unique states. A schematic of the 18-bit LFSR is shown in Figure 28:

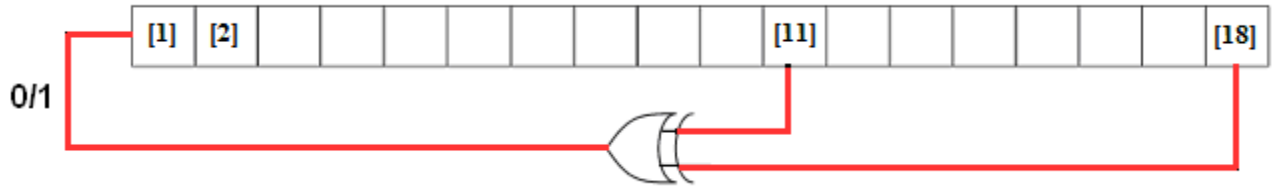

Figure 28: 18-bit LFSR (Fibonacci implementation) with the feedback circuit

The Cλash code for an 18-bit LFSR is:

```
vlfsr18 :: Word -> (Word, Word)
vlfsr18     st = (stN, out)
             where
             st_bits = s2bv st
             n_bit = hwxor (st_bits ! 10) (st_bits ! 17)
             st_shift_b = s2bv (shiftR st 1)
             stN = bv2s (vreplace st_shift_b 0 n_bit)
             out = st
```

Figure 29: Cλash code of an 18-bit LFSR (Fibonacci form)

The Cλash code is directly derived from the schematic structure of the 18-bit LFSR [Figure 28]. The only thing that might seem a bit confusing is that the output and the state need to be 18-bit Signed numbers and the operation that needs to be done in order to acquire the new state has be performed in bit level. For that reason the *s2bv* and *bv2s* functions are being used, which simply transform a signed number to bit vector and a bit vector to a signed number respectively.
Since the FIR-filters are 16-tap filters each of these filters needs a set of 16 FIR coefficients. So in order to produce random sets of coefficients 16 LFSRs are being needed.
The final coefficient structure will consists of 4 LFSRs. To produce the necessary FIR coefficients 64 18-bit LFSRs will be needed.

The Cλash code of the function which will generate the set of FIR coefficients and the final coefficients structure is shown in Figure 30,

```
fir_coef :: WordLst16 -> (WordLst16,WordLst16)
fir_coef    hc_st = (hc_stN,out)
            where
            coef_vec = vmap vlfsr18 hc_st
            hc_stN = vmap fst coef_vec
            out = vmap snd coef_vec

coef_struct :: (WordLst16,WordLst16,WordLst16,WordLst16)
               -> ((WordLst16,WordLst16,WordLst16,WordLst16),(WordLst16,WordLst16,WordLst16,WordLst16))
coef_struct    (hca_st,hcb_st,hcc_st,hcd_st) = ((hca_stN,hcb_stN,hcc_stN,hcd_stN),(hc_a,hc_b,hc_c,hc_d))
               where
               (hca_stN,hc_a) = fir_coef hca_st
               (hcb_stN,hc_b) = fir_coef hcb_st
               (hcc_stN,hc_c) = fir_coef hcc_st
               (hcd_stN,hc_d) = fir_coef hcd_st
```

Figure 30: Cλash code of the fir_coef function which will produce the 16 coefficients for the FIR-filters and the final coefficient structure (coef_struct) which simply contains 4 replicas of the fir_coef function/block.

The PFS structure with the coefficient structure consisting of LFSRs is being shown in the schematic of Figure 31:

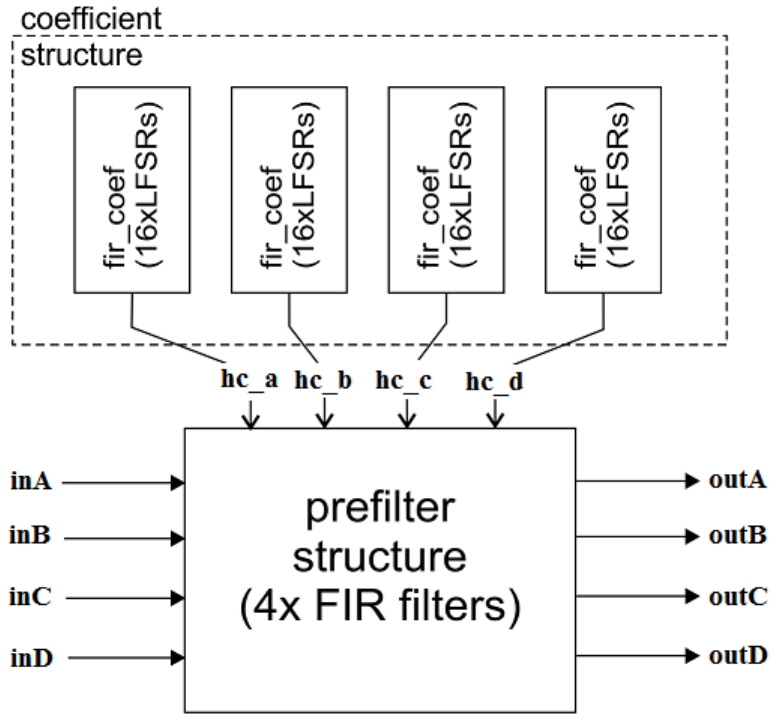

Figure 31: schematic of Pre-Filter Structure consisting of coefficient_structure and prefilter_structure

and the Cλash code of that final Pre-Filter Structure is:

```
pfs_1k    ff (State (pf_cntr,(hca_st,hcb_st,hcc_st,hcd_st),(regs_a,regs_b,regs_c,regs_d))) (inA,inB,inC,inD)
          = ((State (pf_cntrN,(hca_stN,hcb_stN,hcc_stN,hcd_stN),(regsN_a,regsN_b,regsN_c,regsN_d))),(outA,outB,outC,outD))
          where
          ((hca_stN,hcb_stN,hcc_stN,hcd_stN),(hc_a,hc_b,hc_c,hc_d)) = coef_struct (hca_st,hcb_st,hcc_st,hcd_st)
          ((pf_cntrN, (regsN_a,regsN_b,regsN_c,regsN_d)), (outA,outB,outC,outD))
          = prefilter_ext ff (pf_cntr, (regs_a,regs_b,regs_c,regs_d)) ((inA,inB,inC,inD),(hc_a,hc_b,hc_c,hc_d))
```

Figure 32: Cλash code of Pre-Filter Structure consisting of prefilter_structure and coefficient_structure with LFSRs

With the use of LFSRs it becomes possible to produce the VHDL code of the PFS. It has to be pointed out that the output data of the PFS are not valid anymore since the FIR coefficients are just randomly generated and that the only reason for using this specific coefficient structure is to be able to produce the VHDL code with Cλash, and getting at least an idea about the resource consumption and the frequency performance of the PFS.

The targeted FPGA is the Stratix IV (EP4SGX230KF40C2) and according to its data sheet it contains 273600 logic registers. [16]
From the synthesis analysis the PFS needs 296072 registers for memory usage. This means that the PFS cannot fit in the selected FPGA. The reason behind this excessive need for registers from the PFS is that the generated VHDL code from Cλash instantiates simple registers instead of block RAMs. This results in a very large set of registers which simply cannot fit to the selected FPGA.

So just for getting an idea about the space and resource usage and the frequency performance of the PFS the number of FIR-filters will be decreased to 256 from 1024.
The results from the synthesis and timing analysis of that smaller PFS are:

- Logic utilization → 91%
- # of dedicated logic registers used → 74886 (41%)
- # of block memories used → 0 (0%)
- # of DSP block 18-bit elements → 128
- f_{MAX} for slow 900mV 0C model → ≈ 114 MHz

From these results it is clear that the resulting hardware doesn't contain any block RAMs and that all the memory elements of the structure are implemented by a set of registers.
Furthermore by locating the worst delay path we see that it propagates through one of the FIR-filters and its memory block. This memory block is just a large set of registers.
To conclude, the bottleneck in the frequency performance of the PFS is the interconnections between the components of the structure itself. Also since the memory of the system consists from simple registers placing them close enough to the filter structure, so that the combinatorial path between memory and filter to be minimum, is rather difficult to be done automatically from Quartus II.

2.5) Conclusions

From the Haskell implementation of the Pre-Filter Structure it is clear that the resulting code is concrete and rather condensed. That makes it easy to obtain an overview of the whole code of the Pre-Filter Structure. For the resulting Haskell code only the functionality of each component was considered without taking into consideration any subcomponents with which these PFS components are being constructed, for example at the case of the FIR-filter we didn't start by considering that n multipliers and adders will be needed but start with thinking that a dynamic length list needs to be multiplied with another list and that a summation of those elements needs also to be performed.. With that way of viewing the hardware the resulting code remains generic and not limited to specific components and furthermore the functionality of the structure can be derived directly by reading the code itself without the need for further information about the structure.

The transition from Haskell to Cλash is done by transforming every dynamic length list in finite length vector, removing recursion and transforming every function so that it can accept finite length numbers. With these transformations, the body of the functions/components sometimes becomes simpler. For example the memory block (*mem_block*) of the FIR-filters and in some other cases some extra functionality needs to be added, for example the multiplier used in the FIR-filters. In any case though the basic idea of each function/component remains intact and the transformations don't change that. That means that as soon as we have the functional description of a system the code translation to Cλash code is just a matter of performing these transformations.

The generated VHDL code from the Cλash compiler, although it is being generated automatically without the user to have to worry about that, in the end that seems to be its basic drawback. In more detail, from the synthesis and timing results is being shown that the memory blocks of the PFS are not instantiated as block RAMs, as it was desired, but just as large sets of registers. Due to that the initial Pre-Filter Structure with 1024 FIR-filters cannot even fit in a single FPGA and furthermore due to these large sets of registers Quartus is not able to place them in an optimum way and minimize the interconnection paths between these registers and the rest of the PFS's components. That exactly is the major bottleneck in the frequency performance of the Pre-Filter Structure. To conclude the fact that no block RAMs are being instantiated is not only the major bottleneck in the performance of the system but also the reason that the PFS cannot even fit in one FPGA, something which was one of the basic requirements for the Pre-Filter Structure.

Chapter 3 - M-Point FFT for APERTIF filter bank

3.1) Introduction

At this Chapter the implementation of the M-point FFT of the PFB will be analyzed. In more detail first we will implement the FFT structure in plain Haskell and afterwards we will transform the Haskell code to Cλash. Afterwards the resulting VHDL description of the FFT structure will be analyzed and finally conclusions will be drawn regarding those two implementations and the resulting VHDL description.

As it is stated before, the Polyphase Filter Bank (PFB) consists of the PFS and the M-point FFT,

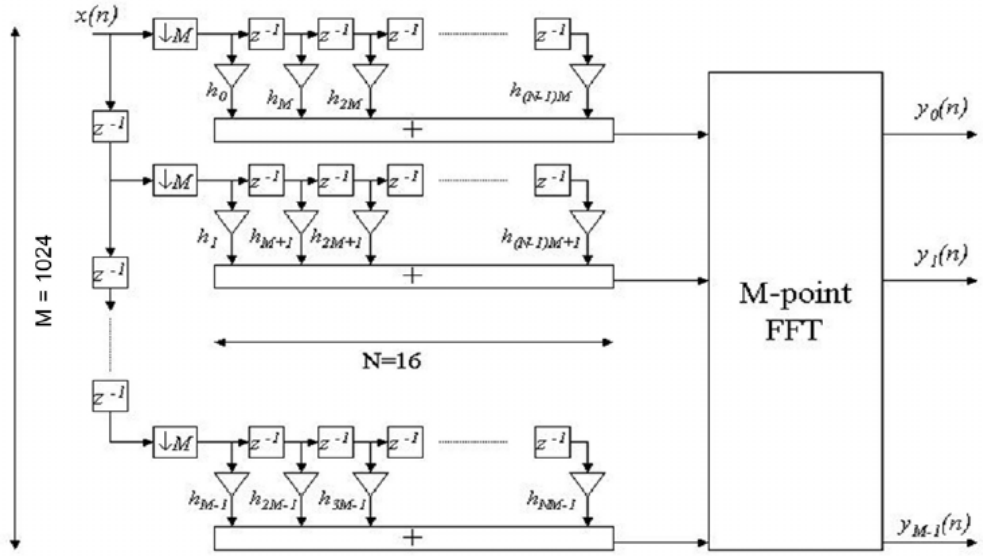

Figure 33: Polyphase Filter Bank (PFB) used in LOFAR

For the FFT of the APERTIF Polyphase Filter Bank, a radix-2^2 Single-path Delay Feedback (R2^2SDF) architecture has been chosen. The particular architecture has been chosen mainly because of its minimal hardware requirements. Furthermore the radix-2^2 architecture has the same number of non-trivial multiplications as a radix-4 architecture and at the same time it uses the butterfly structure of radix-2 architecture and that results in the same number of multiplications but less additions.
That particular architecture has also been used in the Polyphase Filter Bank for LOFAR.

3.2) Radix-2^2 DIF FFT Algorithm

The general equation describing an N-point DFT is:

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk} \quad , \quad 0\le k\le N \tag{3.1}$$

where $W_N^{nk}=\exp\left(j\frac{2\pi nk}{N}\right)$ are the twiddle factors

Since the DFT is intended to be implemented in Hardware there is need of an FFT, so that computations and thus hardware to be minimized. There is a big set of FFT algorithms (radix-2, radix-4, radix-2^2, split radix, etc) which can be distinguished coarsely as Common Factor Algorithms (CFA). With the CFA algorithms what is actually being done is to map a one-dimensional map into two or multi-dimensional representation. With that way the congruence property of the DFT is being exploited and the necessary computations are being simplified [6], [7].

For the specific case a radix-2^2 FFT algorithm is being used and for the specific algorithm a 3-dimensional linear index map has been chosen to be applied at **n** and **k** parameters. That mapping results to the following relations:

$$\begin{aligned} n &= \langle \frac{N}{2}n_1+\frac{N}{4}n_2+n_3\rangle \\ k &= \langle k_1+2k_2+4k_3\rangle \end{aligned} \tag{3.2}$$

From that linear indexing the general DFT equation gets the following form:

$$\begin{aligned} X(k_1+2k_2+4k_3) &= \sum_{n_3=0}^{\frac{N}{4}-1}\sum_{n_2=0}^{1}\sum_{n_1=0}^{1}x(\frac{N}{2}n_1+\frac{N}{4}n_2+n_3)W_N^{(\frac{N}{2}n_1+\frac{N}{4}n_2+n_3)(k_1+2k_2+4k_3)} \\ &= \sum_{n_3=0}^{\frac{N}{4}-1}\sum_{n_2=0}^{1}\left\{B_{\frac{N}{2}}^{k_1}(\frac{N}{4}n_2+n_3)W_N^{(\frac{N}{4}n_2+n_3)k_1}\right\}W_N^{(\frac{N}{4}n_2+n_3)(2k_2+4k_3)} \end{aligned} \tag{3.3}$$

where $B_{N/2}^{k_1}\left(\frac{N}{4}n_2+n_3\right)=x\left(\frac{N}{4}n_2-n_3\right)+(-1)^{k_1}x\left(\frac{N}{4}n_2-n_3+\frac{N}{2}\right)$ and it represents the butterfly structure used in the FFT algorithm [1].

The basic idea behind any CFA algorithm is to proceed with the second step of the decomposition to the rest of the DFT coefficients before the construction of the next butterfly. In more detail that means that to proceed with the second step of decomposition to the remaining DFT coefficients before the construction of the next butterfly. With that way the exceptional values in multiplication can be exploited

With some further simplification equation (3.3) can written as:

$$X(k_1+2k_2+4k_3)=\sum_{n_3=0}^{\frac{N}{4}-1}\left[H(k_1,k_2,n_3)W_N^{n_3(k_1+2k_2)}\right].W_{\frac{N}{4}}^{n_3k_3} \tag{3.4}$$

where $H(k_1,k_2,n_3)$ is:

$$H(k_1,k_2,n_3)=\overbrace{\left[x(n_3)+(-1)^{k_1}x(n_3+\frac{N}{2})\right]}^{\text{BF I}}+(-j)^{(k_1+2k_2)}\overbrace{\left[x(n_3+\frac{N}{4})+(-1)^{k_1}x(n_3+\frac{3}{4}N)\right]}^{\text{BF I}} \tag{3.5}$$

BF II

From (3.4) it can be deduced that by applying the CFA algorithm once, it results in 4 sets of N/4-point DFTs. In other words after each application of the CFA algorithm the initial size N-point DFT breaks down to 4 independent N/4-points DFTs.
In Figure 34 the application of the CFA can been seen for the case of a 16-point radix-2^2 FFT algorithm. In more detail by using the index mapping described in equations (3.2) for the first level of the 16-point DFT what is actually happening is that the complexity of the computations that need to be performed is decreased by a factor of (1/4). That is actually the goal of the CFA, to decrease the amount of computations that need to be performed during a DFT computation.

The structure of the 1st phase of decomposition that is shown in Figure 34 and is derived directly from equation (3.4) and (3.5). Particularly (3.5) states the mathematical description of the BFI and BFII butterfly structures.

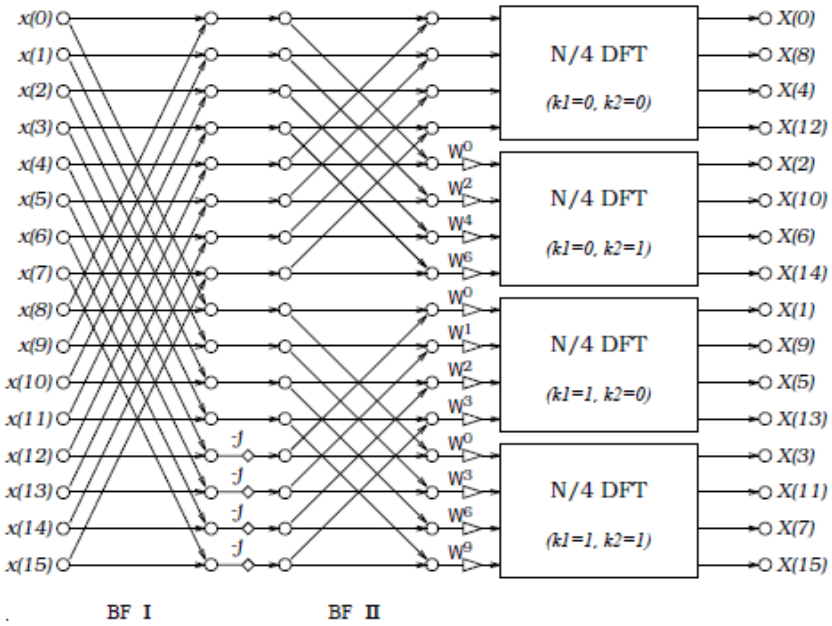

Figure 34: Butterfly with decomposed twiddle factors [1]

In order to obtain the complete radix-2^2 FFT algorithm the CFA algorithm has to be applied recursively to each of the resulting N/4 DFTs.

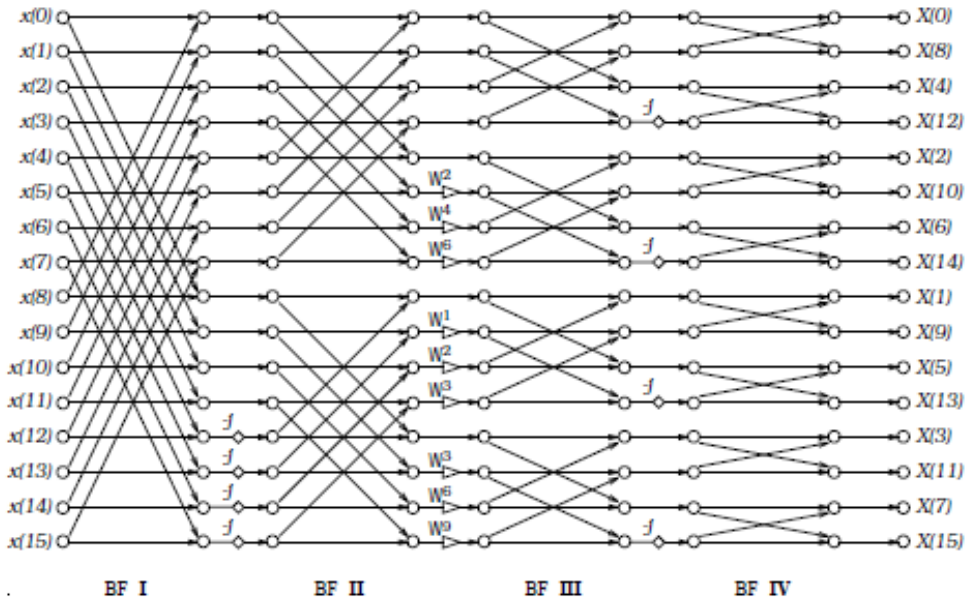

Figure 35: Radix-2^2 DIF FFT flow graph for N=16 [1]

It has to be noted that the order of the twiddle factors after that first stage of decomposition is different than the order in the radix-4 algorithm since the radix-2^2 FFT algorithm is using the same butterfly structures as radix-2 but it follows the computation scheme of radix-4.

3.3) Implementation of Radix-2^2 Single Delay Feedback (R2^2SDF) architecture

As it is been stated already the chosen architecture for the FFT will be a Radix-2^2 Single Delay Feedback (R2^2SDF) architecture. It has been chosen a pipeline architecture because eventually the FFT as part of the PFB will be downloaded to an FPGA and for that reason the optimum usage of the hardware is desired. A general schematic of SDF architectures is shown in Figure 36:

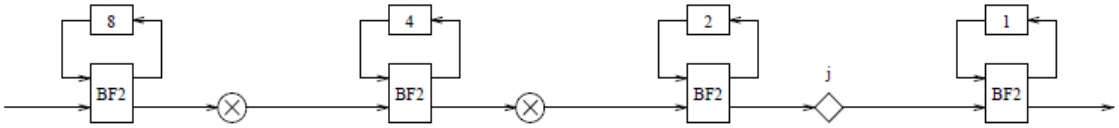

Figure 36: schematic representation of 16-point Single Delay Feedback (SDF) architecture

With this specific architecture, the usage of the registers is more efficient compared to other pipeline architectures. Furthermore, the number of multipliers and the needed memory size is the minimum compared to other common pipeline architecture [1].

	Complex multipliers	Complex adders	Memory size	Control logic	Comp. Utilization add/sub	Comp. Utilization Multiplier
R2SDF	$\log_2 N - 2$	$2\log_2 N$	$N-1$	simple	50%	50%
R4SDF	$\log_4 N - 1$	$8\log_4 N$	$N-1$	medium	25%	75%
R4SDC	$\log_4 N - 1$	$3\log_4 N$	$2N-2$	complex	100%	75%
R2^{2}SDF	$\log_4 N - 1$	$4\log_4 N$	$N-1$	simple	75%	75%
R2MDC	$\log_2 N - 2$	$2\log_2 N$	$3N/2-2$	simple	50%	50%
R4MDC	$3(\log_4 N - 1)$	$8\log_4 N$	$5N/2-4$	medium	25%	25%

Table 1: Hardware requirement comparison between common pipeline architectures

Following from Table 1, Radix-2^2 Single Delay Feedback (R2^{2}SDF) architecture seems to be the most reasonable choice for the M-point FFT of the PFB.

A 256-point R2^{2}SDF pipeline architecture is being shown in Figure 37:

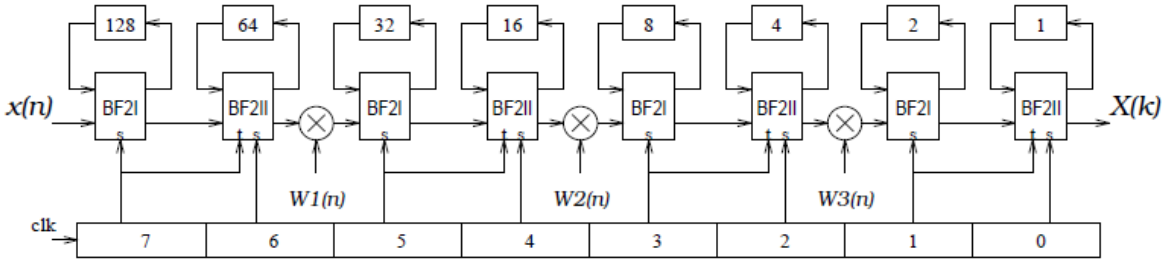

Figure 37: R2^{2}SDF pipeline FFT architecture for N=256 [1]

The pipeline SDF architecture is directly related with the data-flow diagram of an FFT (like the one in Figure 35) through the number of butterfly decomposition stages. That means that the number of butterfly stages needed for a complete N-point FFT is the same as the number of sets of butterfly modules. Let's consider the case of a 16-point radix-2^2, from Figure 35 it is clear that for the complete FFT, 2 stages of butterfly computations are needed. In this case an R2^{2}SDF chain diagram will be,

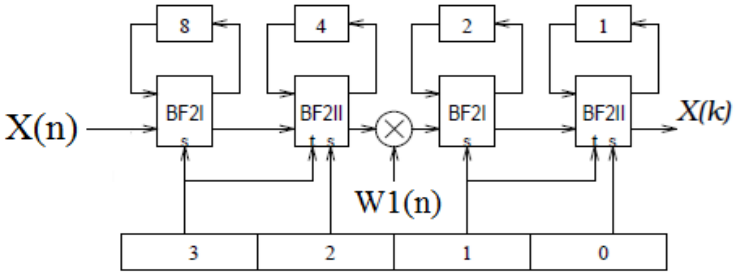

Figure 38: FFT chain diagram for the case of 16-point R2^{2}SDF architecture

It is now obvious that every stage of butterfly computation results in a set of butterfly modules in the SDF chain.

Furthermore, the SDF architecture in general can be viewed as a recursion of the same basic structure which is consisting of a butterfly structure and a multiplier.

3.3.1) Butterfly Modules implementation

As is explained before, the R2^{2}SDF architecture is a repetition of the same basic building block. That block consists of two butterfly modules (BF2I and BF2II) and a multiplier. These two butterfly modules can be seen in Figure 39,

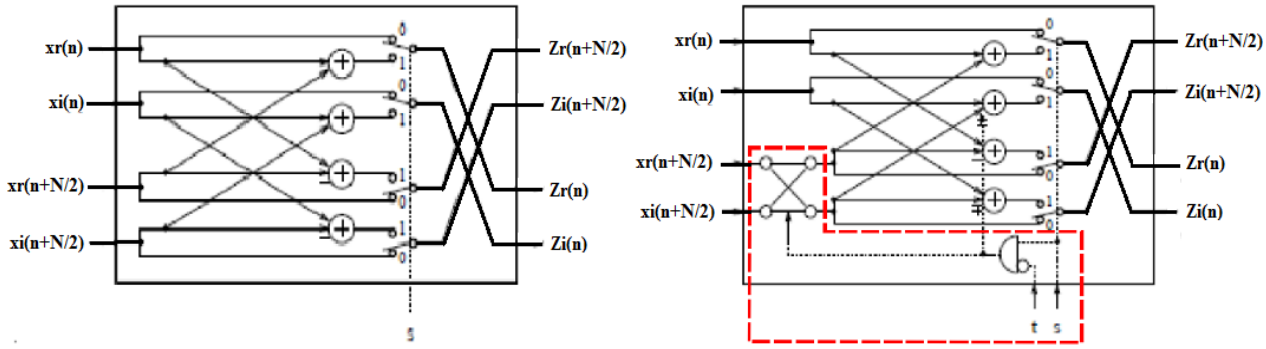

Figure 39: Butterfly structure for R2^{2}SDF FFT processor [1]

With these two types of butterflies, the operation of the R2^{2}SDF chain is as follows:
For the first butterfly (BF2I) on the first N/2 clock cycles, the multiplexers are turn to position '0' and the butterfly is idle. That means that data from the input is forwarded to the shift registers which is connected to the butterfly. That state of the butterfly will be called "*F-state*". During the next N/2 clock cycles, the multiplexers turn to position '1' and the 2-points DFT is being computed with inputs, the input data and the data stored in the shift register. That state of the butterfly will be called "*B-state*".
The functionality of the second butterfly (BF2II) is basically the same but now there is an extra state, the "*Bj-state*". During this state the 2-point DFT computation takes place as in B-state but additionally the "-j multiplication" is also being performed [1].

As can been seen in Figure 37 and Figure 39, for controlling the multiplexers in the butterflies two signals (s and t) are being used. In order to produce those signals a dedicated clock structure has been added. In order to avoid that clock structure we are going to add an extra counter which will be responsible for controlling the multiplexers and the "-j multiplication". In other words that counter will determine the state of each butterfly.

A schematic for both butterflies is shown in Figure 40.

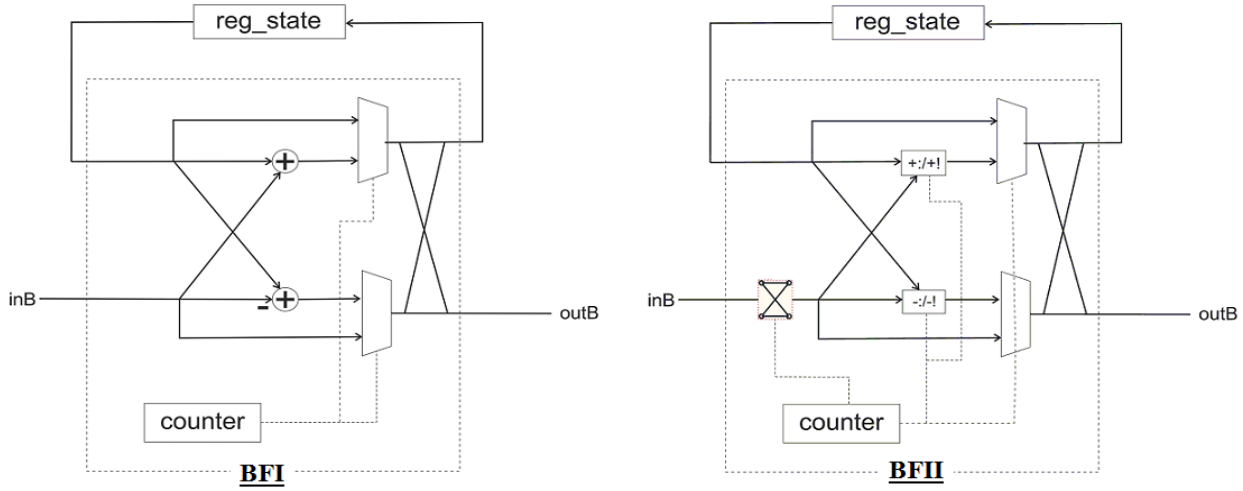

Figure 40: butterfly structures with embedded control counter

Something that needs to be pointed out, is that especially at the BF2II at the two adders is not performed exactly addition/subtraction but a slightly different computation.
In more detail those computations are:

- **[+:]** → (ReA + ReB, ImA + ImB)
- **[-:]** → (ReA - ReB, ImA - ImB)
- **[+!]** → (ReA + ReB, ImA - ImB)
- **[-!]** → (ReA - ReB, ImA + ImB)

That means that during the *Bj-state*, a slightly different computation is being performed inside the 2nd butterfly.

Now follows the Haskell description of these butterflies. Once again only the functionality was taken into consideration.
According to the schematics the resulting Haskell code is:

```
-- the first butterfly module BF2I
bf1 :: BFState -> Sample -> ( BFState, Sample)
bf1     (reg_state,cnt_state) inB = ((reg_stateN,cnt_stateN),outB)
                    where
                    (reg_stateN, inT) = shift_reg reg_state outT
                    n = length reg_state
                    cnt_stateN = (cnt_state + 1) `mod` (2*n)
                    (outT, outB) = if (cnt_state >= n)
                                   then (inT - inB, inT + inB) -- butterfly state
                                   else (inB, inT) --Forward state
```

Figure 41: Haskell code for butterfly module 1 (BF2I)

```
-- the second butterfly module BF2II
bf2 :: BFState -> Sample -> (BFState, Sample)
bf2 (regs, cntr) input = ((regsN, cntrN), output)
  where
    n = length regs
    minj = (0 :+ (-1))
    cntrN = (cntr + 1) `mod` (4 * n)
    (output, regsin) = if (cntr >= n) && (cntr < 2*n)
                       then (regsout + input, regsout - input) --butterfly state
                       else if cntr >= 3*n
                            then ( regsout + minj * input, regsout - minj * input ) --butterfly and j state
                            else (regsout, input) -- forwarding state
    (regsN, regsout) = (regsin +>> regs, last regs) -- functionality of shift register
```

Figure 42: Haskell code for butterfly module 2 (BF2II)

For the resulting Haskell code for these two butterfly modules the inner components, like the adders and the multiplexers, haven't been considered but once again the functionality of each butterfly as a block has been considered. Exactly as was done for the FIR-filter and the rest of the blocks/functions of the PFS.

Table 2 and Table 3 shows the operation of each of the two butterfly modules and how the register in being filled and how the butterfly goes through its states,

cc	inB	reg_stateN	cnt_stateN	bf_state	outB	outT
1	1	1 -- 0	1	F	0	1
2	2	2 -- 1	2	F	0	2
3	3	(-2) -- 2	3	B	4	-2
4	4	(-2) -- (-2)	0	B	6	-2
5	5	5 -- (-2)	1	F	-2	5
6	1	1 -- 5	2	F	-2	1

Table 2: functionality of BF2I for a 2-bit shift register

cc	inB	reg_stateN	cnt_stateN	bf_state	outB	outT
1	1	1 -- 0	1	F	0	1
2	2	2 -- 1	2	F	0	2
3	3	(-2) -- 2	3	B	4	-2
4	4	(-2) -- (-2)	4	B	6	-2
5	5	5 -- (-2)	5	F	-2	5
6	1	1 -- 5	6	F	-2	1
7	2	(5 + j2) -- 1	7	Bj	(5 - j2)	5 + j2
8	3	(1 + j3) -- (5 + j2)	0	Bj	1 - j3	1 + j3
9	4	4 -- (1 +j3)	1	F	5 + j2	4
10	5	5 -- 4	2	F	1 + j3	5

Table 3: functionality of BF2II for a 2-point shift register

In the specific example both butterflies have the same size registers and the main purpose of such an example is to illustrate the function which is being performed at each state that the butterflies can be into.
In more detail, for the BFI from Table 2 it can been seen that for two cycles the input data are being forwarded to the shift register and for the next two clock cycles the butterfly operation, as it is described in Figure 40.
For BFII there is an extra state that it can be into. The computation which is performed in that specific state is being illustrated at the 7^{th} and 8^{th} clock cycle of Table 3, where the butterfly is at Bj-state and in that state the "*–j multiplication*" is being performed. That means that the imaginary part of the input is being swapped with the real part. That's why a complex number appears in the shift register and the output of the butterfly, although the inputs so far were only real numbers.

3.3.2) Twiddle Factor Multiplier implementation

Every DFT computation involves a number of complex multiplications. Therefore also in the case of a radix-2^2 FFT algorithm there is the need of a multiplier so that the necessary multiplications to be performed.
The Haskell code for such a multiplier is:

```
-- complex multiplier with the twiddle factors embedded
tw_multi :: [Sample] -> Int -> Sample -> (Int, Sample)
tw_multi    ws cnt_state input = (cnt_stateN, output)
                    where
                    cnt_stateN = (cnt_state + 1) `mod` (length ws)
                    output = (ws !! cnt_state) * input
```

Figure 43: Haskell code for the multiplier used at the twiddle factors multiplication

As can be seen in Figure 43, the specific multiplier accepts a list of complex numbers as one of its input arguments. That list represents the list of the twiddle factors. From Figure 37 it becomes clear that at every stage of the FFT chain, a different list of twiddle factors is needed. Instead of instantiating a new multiplier for every stage the twiddle factor list is given as an input argument to the multiplier itself.
Also at the multiplier an extra feature has been included. That extra feature is that with a counter a specific element of the twiddle factors list is being send for multiplication at every time. The reason for adding the specific feature will be explained in detail later on.

3.3.3) Basic Building Block implementation

Now the next step is to combine those two butterfly modules with a multiplier into one block/function which will be the basic building block of the R2^{2}SDF FFT architecture. That basic building block is simply an interconnection between the two butterfly structures and a multiplier.

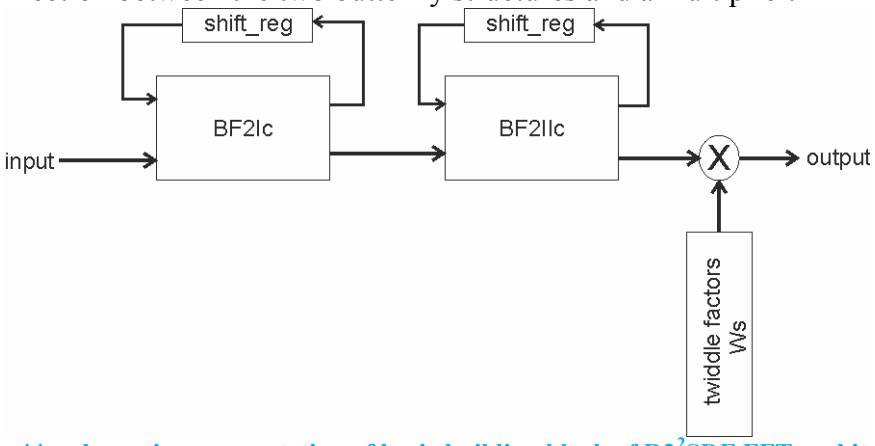

Figure 44: schematic representation of basic building block of R2^{2}SDF FFT architecture

So the Haskell code for that block is pretty much straightforward,

```
-- basic building block of FFT chain
bf12m :: [Sample] -> (BFState, BFState, Int) -> Sample -> ((BFState, BFState, Int), Sample)
bf12m    ws (bf1st, bf2st, mcntr) input = ((bf1stN, bf2stN, mcntrN), output)
                        where (bf1stN, bf1out) = bf1 bf1st input
                              (bf2stN, bf2out) = bf2 bf2st bf1out
                              (mcntrN, output) = tw_multi ws mcntr bf2out
```

Figure 45: Haskell code for basic building block of R2^{2}SDF FFT architecture

The specific block accepts three input arguments,

1) ws → the list of the twiddle factors
2) (bf1st,bf2st,mcntr) → the states of the block (the states of the butterflies and the state of the counter of the multiplier)
3) Input → the actual input of the block

From that it becomes clear that the specific building block of the FFT accepts the twiddle factors as an extra input and that means that the twiddle factors need to be provided externally to the FFT.

3.3.4) Butterfly and Twiddle Factors Synchronization

Now by reconsidering the data-flow diagram of the radix-2^2 FFT algorithm, as it is for all the FFT algorithms, there are data dependencies between the data of each computation stage and especially for the case of a radix-2^2 where the basic butterfly structure is being consisted from 2 separate butterflies there is also data dependencies between the butterflies of the same computation stage. That means that not only the stages but also the butterflies of each stage needs to be synchronized in order to function correctly and these data dependencies not to be violated.
For example let's consider the case of a 16-point radix-2^2 FFT flow graph,

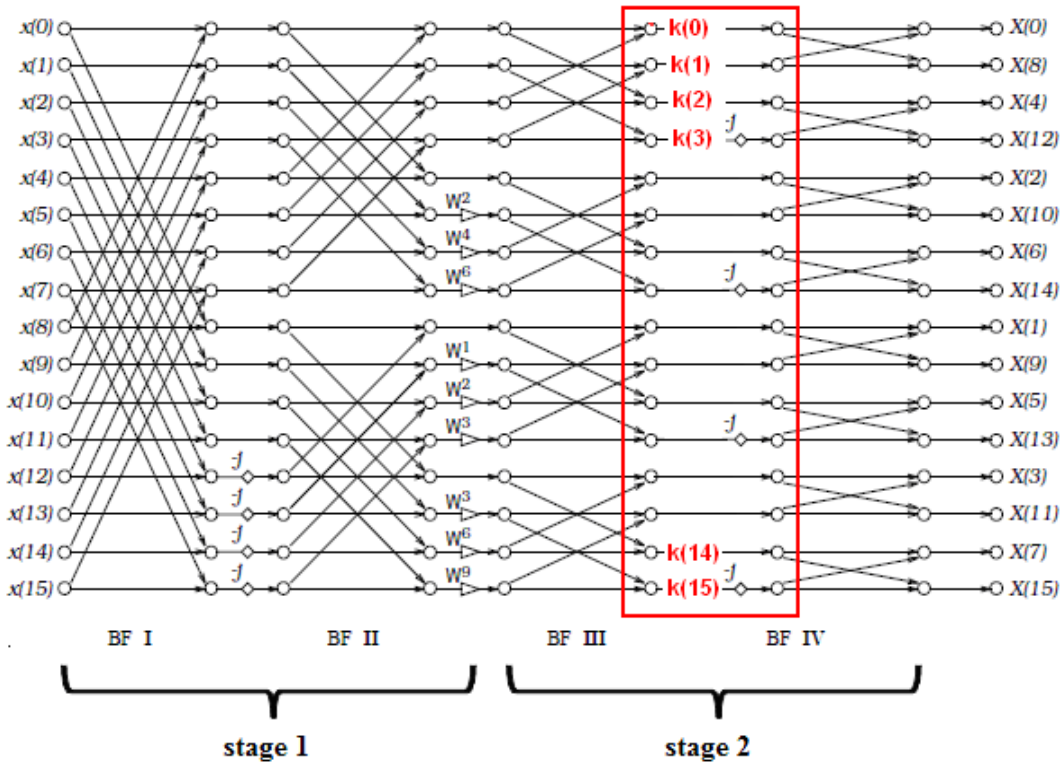

Figure 46: 16-point radix-2^2 FFT flow graph [1],[3]

As it already has been explained each butterfly stores data to the registers for N/2 clock cycles, where N: length of the register and after that it computes the butterfly. That means that a delay is being inserted from every butterfly module and that the butterflies of each stage needs to be synchronized in order to compensate for that delay.
For example at stage 2 the first butterfly module (BFIII) will forward data to the shift registers for 2 clock cycles and then will be at butterfly state for another 2 clock cycles. At the same time the second butterfly module (BFIV) will forward data for 1 clock cycle, be at butterfly state for 1 clock cycle, forwarding data for another clock cycle and finally be at Bj-state. But from Figure 46 it can been seen that there data dependencies between these two butterfly modules. So they had to be synchronized so that when valid data are being produced from the first butterfly the second butterfly to be in the correct state to accept them.

Table 4 shows, for the case of a 16-point FFT, the state pattern of each butterfly in order to be able to maintain the data dependencies,

	Stage 1		Stage 2	
cc	**BF_I**	**BF_II**	**BF_III**	**BF_IV**
1	F	F	F	F
2	F	F	F	Bj
3	F	F	B	F
4	F	F	B	B
5	F	Bj	F	F
6	F	Bj	F	Bj
7	F	Bj	B	F
8	F	Bj	B	B
9	B	F	F	F
10	B	F	F	Bj
11	B	F	B	F
12	B	F	B	B
13	B	B	F	F
14	B	B	F	Bj
15	B	B	B	F
16	B	B	B	B

Table 4: butterfly states for case of 16-point radix-2^2 FFT

From Table 4 it becomes clear that when the first butterfly of every stage is at *B-state,* the second butterfly of the same stage needs to be in *F-state*.

As it has been mentioned before the butterflies of each stage needs to be synchronized in such a way that the delay of the data computation in each butterfly will be compensated. This synchronization can be done by initializing the counters of each butterfly so the desired state pattern will be achieved. That pattern is being derived by checking the flow graph of the FFT as the one in Figure 46, where it can been seen when each of the butterflies will start producing valid data. From that it can be deduced the state that each butterfly needs to be at every clock cycle so that the data dependencies won't be violated.

For example the case of the 16-point FFT is being considered, from Table 4 it can been see that in stage 2 the BF2II module (BF_IV in the table) has to cycle through its states in the pattern: **[F → Bj → F → B]** and the BF2I module (BF_III in the table) has to follow the pattern: **[F → B]** so that the data dependencies between these two butterflies won't be violated. From that it becomes clear that for the specific stage the initial values of the butterfly counters have to be: $\mathbf{CNTR_{BF\_III} = 0}$ **and** $\mathbf{CNTR_{BF\_IV} = 2}$.

By initializing the counters in such a way that the initial state of the butterfly is being determined from that value and with that way the desirable state pattern is being achieved.

By following the same procedure for stage 1 the initial values for the counters are, $\mathbf{CNTR_{BF\_I} = 0}$ **and** $\mathbf{CNTR_{BF\_II} = 8.}$

Regarding the data dependencies between two consecutive stages from Table 4, it seems that there is no need for further synchronization since stage 1 is at *B-state* or *Bj-state* and producing valid data stage 2 is at *F-state*.

By repeating the same procedure for bigger FFTs (with more computation stages) we realize that at each stage the BF2I counter will be initially 0 and the BF2II counter will be initialized to the size of the biggest register of that specific stage, which actually will be the size of the register of the BF2I butterfly.
Table 5 shows the initial values for the counters in the case of a 256-point R2^{2}SDF FFT,

Stage 1		**Stage 2**		**Stage 3**		**Stage 4**	
BF2I	**BF2II**	**BF2I**	**BF2II**	**BF2I**	**BF2II**	**BF2I**	**BF2II**
0	128	0	32	0	8	0	2

Table 5: initial counter values for N=256 R2^{2}SDF FFT

For the general case of an N-point R2^{2}SDF FFT only the data dependencies between the butterflies of the same computation stage needs to be considered since due to the structure of the SDF architecture and the radix-2^2 FFT it seems the stages are already synchronized and the butterfly counters of each stage needs to be initialized as it has been explained before.

Apart from the butterflies, also the twiddle factor multipliers need to be synchronized and that because of the delay that is being inserted to the FFT chain from the functionality of the butterflies. By going back to the FFT chain graph, it can be seen that at the end of every stage a twiddle factor multiplication needs to be performed. Since the data from the butterflies is delayed due to the way the butterflies operate, that means that also the twiddle factors need to be delayed also so that the correct twiddle factor to be multiplied with the correct output of the butterflies. Since the twiddle factors of each stage is given to the multiplier in the form of list, the synchronization with the butterfly outputs can be done in 2 ways.
One is to reorder the twiddle factor list itself so that the correct twiddle factor to be present in the multiplier. The other, is to use the counter of the multiplier as it has been done with the butterfly modules.
For the initialization method of the multiplier counter, the flow graph of Figure 46 needs to be revised. From such a graph, the delay of the data can be calculated and figure out at which position the twiddle factor list should be initialized in order to have the proper twiddle factor present in the multiplier after that initial delay. For example, in the case of a 16-point FFT, for stage 1 valid data will come out of the butterflies after 12 clock cycles. That means that the first of the twiddle factors (W_{16}^{0}) needs to be at the multiplier at that specific clock cycle. That can be done if the counter will be initialized with the value **4,** because after 12 clock cycles the counter will be pointing in the first element of the list. By using the same reasoning, the following formula is derived to determine the initial value of the multiplier counter,

$$multi\_cntr_{initial} = \frac{N}{(2^2)^{(S-1)}} - (buttterfly\_delay)$$, where N: size of the FFT

S: stage of the FFT

3.3.5) Twiddle factors list creation in R2^{2}SDF architecture

For every stage of the FFT a different twiddle factor list is needed. These lists tend to become relatively large very quick, for example consider the case of a 64-point FFT, the twiddle factor list of stage 1 [Figure 47] is a list of 64 elements. Furthermore, by checking the FFT data-flow graph it can be seen that there is a certain grouping at the twiddle factor lists. So there is the need of having an abstract way to create the list of the twiddle factors for each stage of the R2^{2}SDF architecture.
Regarding the size of the initial FFT the number of computation stages can be calculated. This calculation can be done by the following formula:

$$N = \left(2^2\right)^S \quad \text{, where N: points of the DFT} \qquad (3.6)$$
S: number of the required stages

For example in the case of a 64-point FFT we will need 3 stages because $64 = (2^2)^3$.

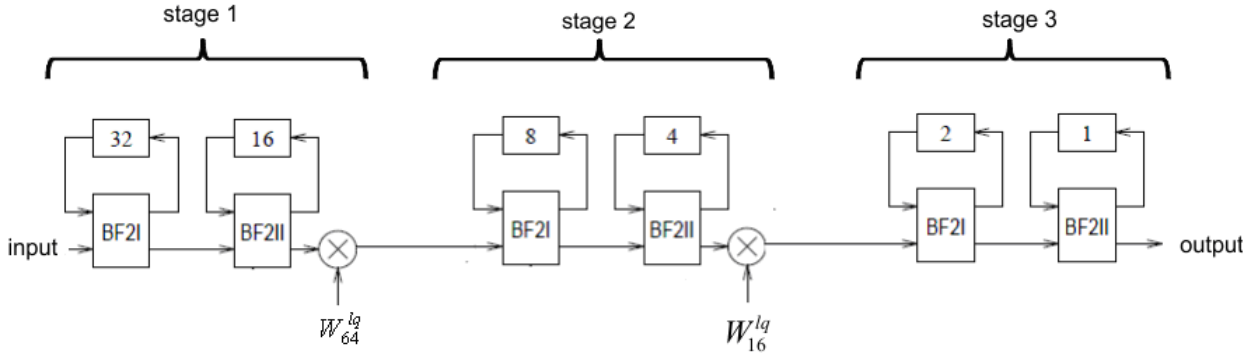

Figure 47: 64-point R2SDF FFT architecture

The number of the necessary stages gives the number of the lists of twiddle factors that are needed and also how many times the CFA algorithm is needed to be applied in order to acquire the complete FFT algorithm.
The basic radix-2^2 butterfly is being shown in Figure 48:

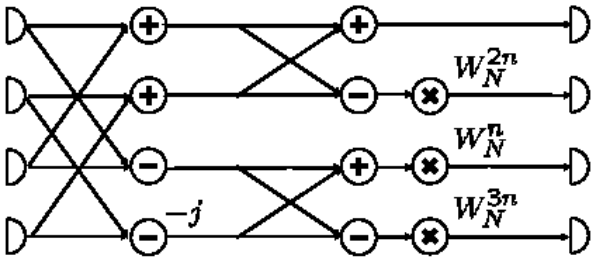

Figure 48: Radix-2^2 butterfly

From the data-flow graph [Figure 49] it can be seen that the twiddle factors in the specific FFT algorithm are being grouped in a specific pattern [0, multiple of 2 (W^{2n}), multiple of 1 (W^{n}) and multiple of 3 (W^{3n})]. That means that for a bigger FFT the list of the twiddle factors will follow the same pattern but each group will be consisted of more twiddle factors.

In more detail in that list 4 equally sized groups can be distinguished,

1) group of 1s
2) group of multiple of 2
3) group of multiple of 1
4) group of multiple of 3

That grouping of the twiddle factors is illustrated in Figure 49:

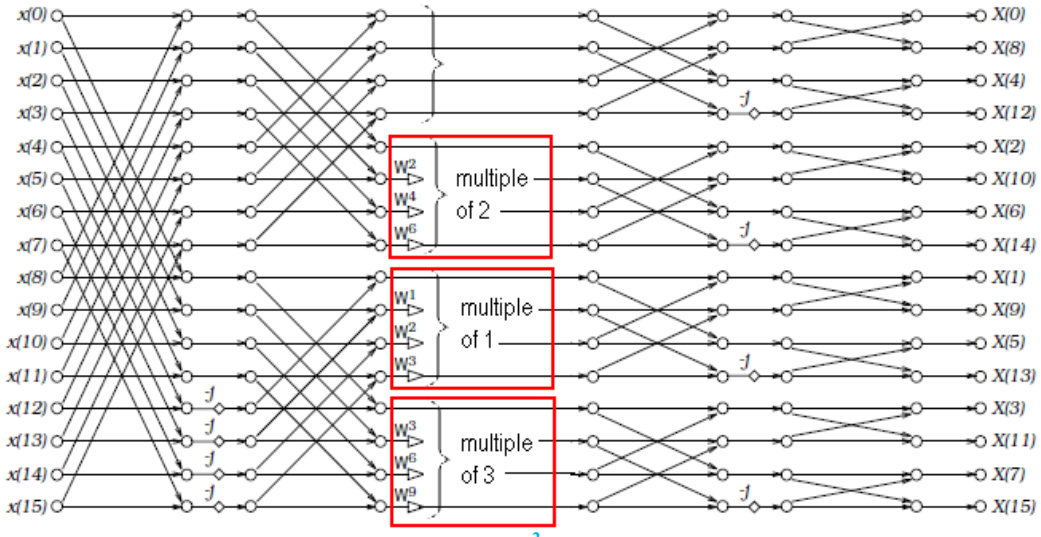

Figure 49: 16-point radix-2^2 FFT with twiddle factors

By taking this kind of grouping into account and the fact that every time the CFA algorithm is applied, 4 DFTs of N/4 points are produced. The following formula is derived for the twiddle factors of each stage as a function of the initial size of the FFT and the stage,

$$\text{Twiddle factor} \Rightarrow W_M^{lq}, \text{ where M} = \frac{N}{4^{(S-1)}}$$
$$q = 0,2,1,3 \qquad (3.7)$$
$$l = 0,1,2,3,\ldots,\left(\frac{N}{4^S}-1\right)$$

For example let's consider the case of a 16-point FFT. That means that 2 stages of butterfly computation are needed ($16 = (2^2)^2$). By using the formula (3.2.2):

- **Stage 1** → $M = \frac{16}{4^{(1-1)}} = 16$ and $l = 0,1,2,3$ since $\left(\frac{16}{4^1}-1\right) = 3$

  $\Rightarrow \quad \mathbf{W_{stage\ 1}} = \left[1,1,1,1,1,W_{16}^2,W_{16}^4,W_{16}^6,1,W_{16}^1,W_{16}^2,W_{16}^3,1,W_{16}^3,W_{16}^6,W_{16}^9\right]$

- **Stage 2** → $M = \frac{16}{4^{(2-1)}} = 4$ and $l = 0$ since $\left(\frac{16}{4^2}-1\right) = 0$

  $\Rightarrow \quad \mathbf{W_{stage\ 2}} = [1,1,1,1]$

Implementation in functional language (Haskell)

Now that the mathematical description of the twiddle factor lists has been derived it should be also implemented in Haskell so that it can be combined with the rest of the FFT.
For that implementation a function, which will accept as inputs an integer which will represent the size of the FFT and an integer that represents the stage of the FFT computational flow, is being needed. The Haskell code for that function is being shown in Figure 50:

```
tfls:: Int -> Int -> [Sample]
tfls   n s = (ws)
       where
       m = fromIntegral (n `div` (4^(s-1)))
       qs = [0,2,1,3]
       ls = [0..(n `div` (4^s))-1]
       lqs = concat (map (\q -> map (*q) ls) qs)
       ws_pol = map (((-2*pi)/m)*) (map fromIntegral lqs)
       ws = map cis ws_pol
```

Figure 50: Haskell code of twiddle factors lists generation function

At *tfls* function (n) is the size of the FFT and (s) is the stage of computation at the FFT data flow graph. The resulted twiddle factor list (ws) is a list of complex numbers (a :+ b) where a: real part of W_S and b: imaginary part.

3.3.6) Tests and simulations

Let's consider the case of a 64-point R2^{2}SDF FFT in order to verify that the implementation works correctly. First the FFT is going to be tested with an impulse response as an input. The resulting output is,

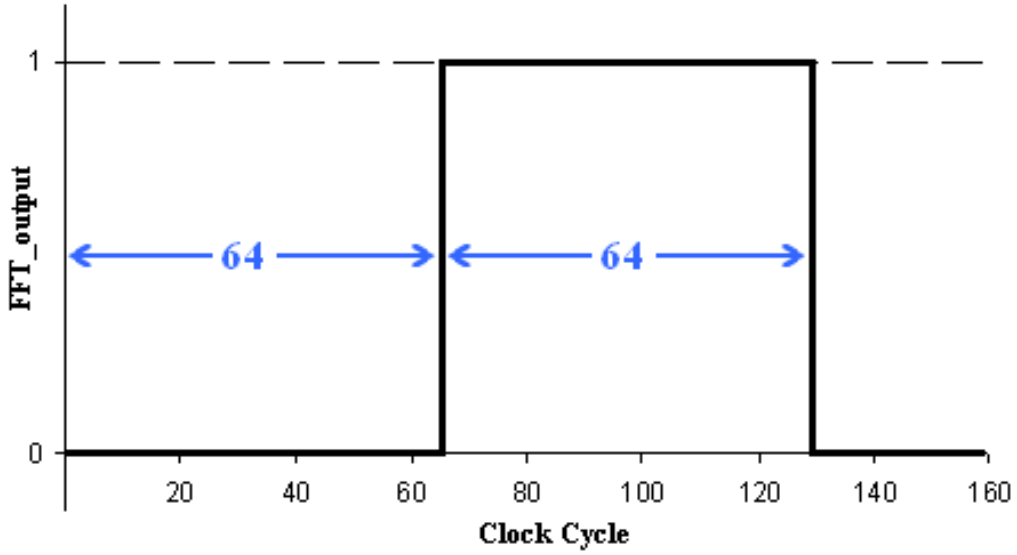

Figure 51: output of fft_64 with impulse input

From Figure 51 it can be seen that for the first 64 clock cycles the output remains 0 and after that 64 1s is produced. That was expected since the initial delay for the 64-point R2^{2}SDF FFT architecture is 64 clock cycles and that means that valid data will start to output from our FFT at the 65th clock cycle.

From a data-flow graph, like Figure 46, it can be seen that the outputs in radix-2^2 algorithm are not in normal order. If the index of the outputs is being expressed in binary code it can be seen that the outputs are in **bit-reverse** order. In order to verify that the outputs of a radix-2^2 algorithm will always be in bit-reverse order regardless the size of the FFT let's consider the case of a 64-point FFT. In order to be able to track down the outputs we need that the values of each output to be the index of the specific output. That means that we need the following pattern fro the output data,
X(0) = 0, X(1) = 1, X(2) = 2, X(3) = 3, …, X(N-1) = N-1, X(N) = N.
For finding out the input pattern in order to acquire a pattern like that in the output of the FFT the iFFT (inverse FFT) function in MatLab is being used.

By giving these inputs in the 64-point R2^{2}SDF FFT that results to the following output sequence,

#	output	out_ rounded	#	output	out_ rounded
0	0.0 :+ 0.0	0	**32**	0.99969673 :+ 1.2516975·10^{-6}	1
1	32.0 :+ 0.0	32	**33**	33.000046 :+ 2.9802322·10^{-7}	33
2	15.999802 :+ 0.0	16	**34**	17.000061 :+ (-6.556511·10^{-7})	17
3	48.0002 :+ 0.0	48	**35**	48.999985 :+ (-6.556511·10^{-7})	49
4	8.000496 :+ 3.4968238·10^{-7}	8	**36**	8.999926 :+ (-5.443843·10^{-7})	9
5	39.999905 :+ 3.4968238·10^{-7}	40	**37**	41.00052 :+ 1.3629643·10^{-6}	41
6	24.000097 :+ 1.2715478·10^{-7}	24	**38**	24.999977 :+ 1.6172679·10^{-6}	25
7	55.999504 :+ (-8.2651957·10^{-7})	56	**39**	56.999947 :+ (-2.1974292·10^{-6})	57
8	4.000141 :+ 4.7683716·10^{-7}	4	**40**	4.999962 :+ 2.3245811·10^{-6}	5
9	35.99973 :+ 4.7683716·10^{-7}	36	**41**	36.999825 :+ (-5.364418·10^{-7})	37
10	20.000202 :+ (-4.7683716·10^{-7})	20	**42**	20.999834 :+ (-5.9604645·10^{-8})	21
11	51.999928 :+ (-4.7683716·10^{-7})	52	**43**	52.99978 :+ (-1.013279·10^{-6})	53
12	12.0000725 :+ 0.0	12	**44**	13.000125 :+ (-2.9802322·10^{-7})	13
13	43.999798 :+ 0.0	44	**45**	44.999996 :+ (-2.9802322·10^{-7})	45
14	28.00027 :+ 0.0	28	**46**	29.000345 :+ 6.556511·10^{-7}	29
15	59.99986 :+ 0.0	60	**47**	60.999977 :+ (-1.2516975·10^{-6})	61
16	2.000042 :+ 1.9947713·10^{-6}	2	**48**	3.0000286 :+ 2.1457672·10^{-6}	3
17	34.00012 :+ (-1.8199258·10^{-6})	34	**49**	34.99965 :+ 2.3841858·10^{-7}	35
18	18.000093 :+ 8.742278·10^{-8}	18	**50**	19.0 :+ 7.1525574·10^{-7}	19
19	49.9999 :+ 8.742278·10^{-8}	50	**51**	50.99987 :+ (-2.1457672·10^{-6})	51
20	9.999747 :+ 1.6292216·10^{-6}	10	**52**	11.000219 :+ 2.0186194·10^{-6}	11
21	41.999496 :+ 6.755473·10^{-7}	42	**53**	43.00016 :+ 1.064945·10^{-6}	43
22	26.0004 :+ (-2.3864573·10^{-8})	26	**54**	27.000174 :+ (-1.11270765·10^{-7})	27
23	58.000202 :+ (-1.9312133·10^{-6})	58	**55**	59.00004 :+ (-3.925968·10^{-6})	59
24	5.999799 :+ 1.462298·10^{-6}	6	**56**	7.000057 :+ 4.053116·10^{-6}	7
25	37.999603 :+ 1.462298·10^{-6}	38	**57**	39.000023 :+ 2.3841858·10^{-7}	39
26	22.000498 :+ (-1.398725·10^{-6})	22	**58**	22.999481 :+ (-7.1525574·10^{-7})	23
27	54.00025 :+ (-1.398725·10^{-6})	54	**59**	55.000076 :+ (-3.5762787·10^{-6})	55
28	14.000103 :+ 1.939135·10^{-6}	14	**60**	15.000015 :+ 1.9073486·10^{-6}	15
29	45.999905 :+ 3.17865·10^{-8}	46	**61**	46.99994 :+ (-1.9073486·10^{-6})	47
30	29.999882 :+ (-1.8755621·10^{-6})	30	**62**	30.999952 :+ 1.9073486·10^{-6}	31
31	61.999958 :+ (-9.218878·10^{-7})	62	**63**	63.000305 :+ (-1.9073486·10^{-6})	63

Table 6: resulted outputs from 64-point R2^{2}SDF FFT architecture

Table 6 gives a basic idea regarding the accuracy of our FFT. In more detail we see that the resulted outputs are pretty close to the expected ones. For further conclusion regarding the accuracy of our architecture more tests and comparison with other architectures needs to be done.

If the outputs are being represented in binary coding system and compare them with the binary representation of the index of the outputs then, as it can been seen in Table 7, the output is again in bit-reverse order.

	binary #	output	binary	#	binary #	output	binary
1	"000000"	0	"000000"	**33**	"100000"	1	"111110"
2	"000001"	32	"100000"	**34**	"100001"	33	"000001"
3	"000010"	16	"010000"	**35**	"100010"	17	"100001"
4	"000011"	48	"110000"	**36**	"100011"	49	"010001"
5	"000100"	8	"001000"	**37**	"100100"	9	"110001"
6	"000101"	40	"101000"	**38**	"100101"	41	"001001"
7	"000110"	24	"011000"	**39**	"100110"	25	"101001"
8	"000111"	56	"111000"	**40**	"100111"	57	"011001"
9	"001000"	4	"000100"	**41**	"101000"	5	"111001"
10	"001001"	36	"100100"	**42**	"101001"	37	"000101"
11	"001010"	20	"010100"	**43**	"101010"	21	"100101"
12	"001011"	52	"110100"	**44**	"101011"	53	"010101"
13	"001100"	12	"001100"	**45**	"101100"	13	"110101"
14	"001101"	44	"101100"	**46**	"101101"	45	"001101"
15	"001110"	28	"011100"	**47**	"101110"	29	"101101"
16	"001111"	60	"111100"	**48**	"101111"	61	"011101"
17	"010000"	2	"000010"	**49**	"110000"	3	"111101"
18	"010001"	34	"100010"	**50**	"110001"	35	"000011"
19	"010010"	18	"010010"	**51**	"110010"	19	"100011"
20	"010011"	50	110010"	**52**	"110011"	51	"010011"
21	"010100"	10	"001010"	**53**	"110100"	11	"110011"
22	"010101"	42	"101010"	**54**	"110101"	43	"001011"
23	"010110"	26	"011010"	**55**	"110110"	27	"101011"
24	"010111"	58	"111010"	**56**	"110111"	59	"011011"
25	"011000"	6	"000110"	**57**	"111000"	7	"111011"
26	"011001"	38	"100110"	**58**	"111001"	39	"000111"
27	"011010"	22	"010110"	**59**	"111010"	23	"100111"
28	"011011"	54	"110110"	**60**	"111011"	55	"010111"
29	"011100"	14	"001110"	**61**	"111100"	15	"110111"
30	"011101"	46	"101110"	**62**	"111101"	47	"001111"
31	"011110"	30	"011110"	**63**	"111110"	31	"101111"
32	"011111"	62	"111110"	**64**	"111111"	63	"011111"

Table 7: binary representation of output from 64-point radix-2^2 FFT

So from that is being verified that regardless the size of the FFT the output data will always be in **bit-reverse order**.

3.4) 1k-points FFT implementation in Cλash

The FFT of the PFB will be a 1k-points FFT, but as it is described at Chapter 2, the PFS produces 4 parallel outputs. Therefore the final FFT needs to be able to process 4 parallel inputs. By reconsidering the CFA and a data-flow diagram, as the one in Figure 34, it becomes clear that after the first CFA application 4 N/4-points DFTs are produced. If we translate that in the case of a 1k-points FFT, after the first application of CFA 4 parallel 256-points DFTs will be produced.
Furthermore from the data-flow diagram it becomes clear that the initial stage of an FFT, regardless size, cannot be split in 4 parallel FFTs. Therefore the final 1k-points FFT will consists of two parts, the initial stage and 4 256-points FFTs which will be implemented using the R2^{2}SDF architecture.

3.4.1) 256-point FFT implementation in Clash

The 256-point FFT R2^{2}SDF chain in being shown in Figure 52:

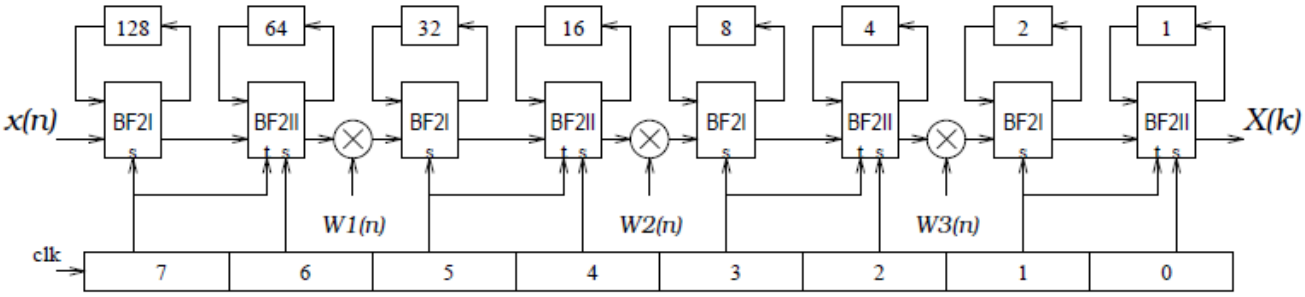

Figure 52: SDF pipeline architecture of a 256-point FFT

This specific architecture has to be implemented in Cλash. According to the requirements the input and output signals and the twiddle factors will be 18 bits signed numbers [13].
The code in general won't change from the one implemented for Haskell but a number of changes has to be done in order to be accepted by Cλash.

Cλash doesn't recognize the Haskell notation of complex numbers {C = Re(C) **:+** Im (C)} [11]. Therefore from now on the complex numbers will be represented as a tuple {C = (Re(C), Im(C)}. Due to this representation, the basic calculations between complex numbers (plus, minus, multiplication) need to be rewritten to be able to accept tuples as input arguments.

```
-- basic calculus functions for complex numbers (Re,Im)
(!+!) :: (NaturalT s) => (Signed s,Signed s) -> (Signed s,Signed s) -> (Signed s,Signed s)
(!+!)    (a,b) (c,d) = (a+c, b+d)
(!-!) :: (NaturalT s) => (Signed s,Signed s) -> (Signed s,Signed s) -> (Signed s,Signed s)
(!-!)    (a,b) (c,d) = (a-c, b-d)
(!*!) :: (NaturalT s) => (Signed s,Signed s) -> (Signed s,Signed s) -> (Signed s,Signed s)
(!*!)    (a,b) (c,d) = (a*c-b*d, ((a+b)*(c+d)-a*c-b*d))
```

Figure 53: basic calculus functions for complex numbers represented as tuples

By using these operators, the two butterfly modules used in the FFT architecture can be implemented so that they accept complex numbers as inputs. But since finite length numbers are used, it has to be made sure that there is no way to run into any overflow problems. During the butterfly operation the only computation that can lead to overflow is the addition. In order to avoid any overflow to occur the inputs of the butterfly first needs to be resized from 18-bits to 19-bits and then perform the necessary computations (addition/subtraction).
In our case the FFT is a pipelined FFT with several computation stages. In order to avoid overflow problems that would mean that at every stage the inputs should be resized to +1-bits from the ones in the previous stage. Like that in the end of the FFT chain we would end up with outputs (18+N)-bits, where N is the number of the stages of the FFT chain. In order to avoid that and the output of the FFT chain to be an 18-bits number in every butterfly after the computations the result is resized back to 18-bits. With that way the input and output of every butterfly and consequently of every stage would remain the same (18-bits). But due to resizing to less bits a shifting to the right needs to be performed before that to keep the most significant bits of the resulted number. The amount of bits that needs to be shifted is the difference of bits between the resized numbers, so in our case it is 1-bit since we resize from 19 to 18-bits.
It has to be pointed out that shifting is the same by dividing with 2^b, where b is the number of bits that we shift. That means that the output of every butterfly will be half of the expected one and consequently the final FFT output will be $1/2^s$ of the expected one, where s is the number of shifting operations which have been performed. In order to avoid any overflow problems, the resulting output of the FFT will be a scaled version of the expected one, this is preferable from the overflow problems.
Furthermore it becomes obvious that the resize and shifting should be done only at the butterfly path and not in the forward path. That is because if the shifting operation is performed also to the forward path that would mean that the stored data on the shift registers of the butterfly would be half and that would result in wrong value.
The two butterfly modules with the necessary resizing and shifting blocks are shown in Figure 54 and Figure 55,

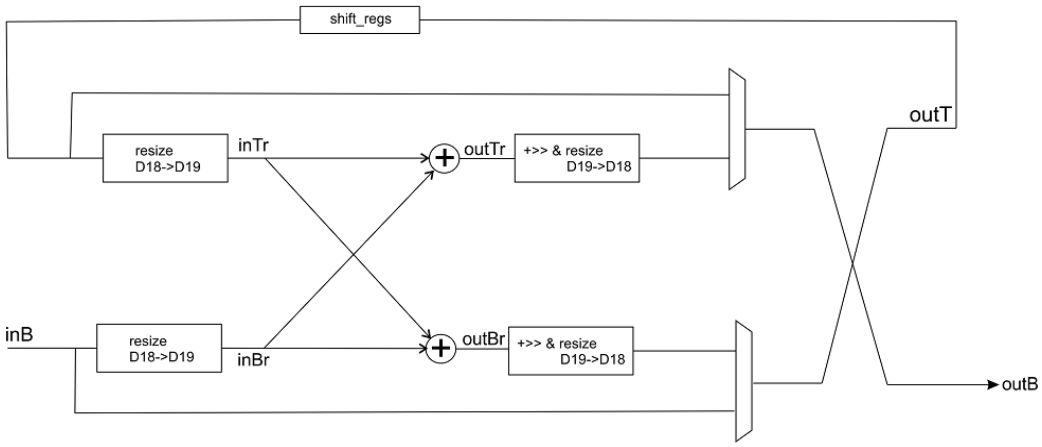

Figure 54: Schematic representation of BF1

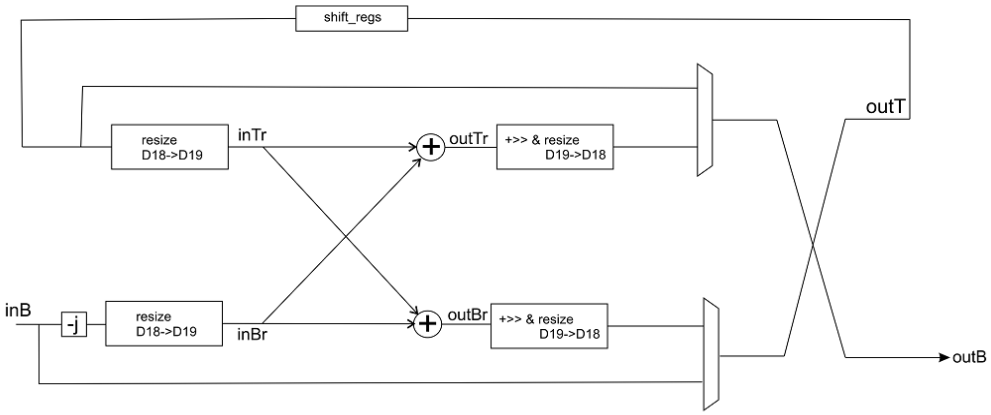

Figure 55: Schematic representation of BF2

So by adding the resizing and shifting operations as it is shown in the schematic representations of the butterflies the following Cλash code is produced,

```
-- abstract fixed point implementation of BF2I
vbf1 n (regs,cntr) inB = ((regsN,cntrN), outB)
                    where
                    cntrN = cntr + 1
                    inTr = (resizeSigned (fst inT), resizeSigned (snd inT))::ComplexD19
                    inBr = (resizeSigned (fst inB), resizeSigned (snd inB))::ComplexD19
                    outTr = inTr !+! inBr -- butterfly function
                    outBr = inTr !-! inBr -- butterfly function
                    (outT, outB) = if (cntr >= n)
                                   then ((resizeSigned (shiftR (fst outBr) 1), resizeSigned (shiftR (snd outBr) 1))::ComplexD18,
                                         (resizeSigned (shiftR (fst outTr) 1), resizeSigned (shiftR (snd outTr) 1))::ComplexD18)
                                   else (inB, inT) -- forward state
                    (regsN, inT) = (outT +>> regs, vlast regs) -- functionality of shift register
```

Figure 56: Cλash code for fixed point implementation of bf1

```
-- abstract fixed point implementation of BF2II
vbf2 n (regs,cntr) inB = ((regsN,cntrN), outB)
                   where
                   minj = (0, -1) :: ComplexD19
                   cntrN = cntr + 1
                   inTr = (resizeSigned (fst inT), resizeSigned (snd inT))::ComplexD19
                   inBr = (resizeSigned (fst inB), resizeSigned (snd inB))::ComplexD19
                   (outTb, outBb) = (inTr !+! inBr, inTr !-! inBr) -- butterfly function
                   (outTbj, outBbj) = ( inTr !+! (minj !*! inBr), inTr !-! (minj !*! inBr) ) -- butterfly & -j mult function
                   (outT, outB) = if (cntr >= n) && (cntr < 2*n)
                                  then ((resizeSigned (shiftR (fst outBb) 1), resizeSigned (shiftR (snd outBb) 1))::ComplexD18,
                                        (resizeSigned (shiftR (fst outTb) 1), resizeSigned (shiftR (snd outTb) 1))::ComplexD18)
                                  else if (cntr >= 3*n)
                                         then ((resizeSigned (shiftR (fst outBbj) 1), resizeSigned (shiftR (snd outBbj) 1))::ComplexD18,
                                               (resizeSigned (shiftR (fst outTbj) 1), resizeSigned (shiftR (snd outTbj) 1))::ComplexD18)
                                         else (inB, inT) -- forwarding state
                   (regsN, inT) = (outT +>> regs, vlast regs) -- functionality of shift register
```

Figure 57: Cλash code for fixed point implementation of bf2
***) ComplexD18 and ComplexD19 => a tuple of 18-bit and 19-bit Signed numbers respectively**

At the butterfly functions (vbf1 and vbf2) an extra input argument has been added **(n)**. This argument is used in the if-statements where the state of the butterfly is determined. An argument like that is necessary because in the Haskell implementation that argument was derived directly from the size of the shift registers but something like that in Cλash is not possible.

As mentioned before Cλash uses vectors instead of lists. That means that in each stage of the FFT, each of the butterflies will need a different size vector as its shift register. In order to be as generic as possible and not having to declare a set of functions with the same body and functionality and with the only difference the size of the shift register polymorphism is going to be used.
This means that in the initial declarations of the butterfly functions no specific types for their arguments have been set and that is because with that way we can instantiate a different butterfly structure for every stage of the FFT chain without repeating the whole function body for every stage. That is being done with the following piece of code,

```
-- 256-point FFT fixed point implementations of butterflies
vbf1s1 :: Unsigned D8 -> (ComplexD18Vect128,Unsigned D8) -> ComplexD18 -> ((ComplexD18Vect128,Unsigned D8), ComplexD18)
vbf1s1 = vbf1

vbf2s1 :: Unsigned D8 -> (ComplexD18Vect64,Unsigned D8) -> ComplexD18 -> ((ComplexD18Vect64,Unsigned D8), ComplexD18)
vbf2s1 = vbf2

vbf1s2 :: Unsigned D6 -> (ComplexD18Vect32,Unsigned D6) -> ComplexD18 -> ((ComplexD18Vect32,Unsigned D6), ComplexD18)
vbf1s2 = vbf1

vbf2s2 :: Unsigned D6 -> (ComplexD18Vect16,Unsigned D6) -> ComplexD18 -> ((ComplexD18Vect16,Unsigned D6), ComplexD18)
vbf2s2 = vbf2

                                                    ⋮
```

Figure 58: instantiation of the butterflies of each stage
***) ComplexD18Vect128 and so on in a vector of 128 numbers of ComplexD18 type**

With this way the whole code of the butterfly functions doesn't need to be repeated for every stage which reduces our code and makes it more compact and more readable.

From complex number theory the multiplication of two complex numbers can be done in 2 ways,

1) $x * y = (a \cdot c - b \cdot d) + j(a \cdot d + b \cdot d)$
2) $x * y = (a \cdot c - b \cdot d) + j[(a+b) \cdot (c+d) - a \cdot c - b \cdot d]$

where x = a +jb and y = c +jd.
It is obvious that for the 2nd multiplication scheme only 3 trivial multiplications are needed and that results in a complex multiplier which consists of 3 hardware multipliers. But with a closer look to that multiplication scheme it can be seen that the number of calculations for the real part and the imaginary part is different. On the other hand in the 1st scheme the number of calculations for the real and the imaginary part is the same but 4 trivial multiplications are needed. That difference between these two multiplication schemes is important because if the two paths (real and imaginary) aren't of equal lengths, that might result in timing issues. That is because when the number of computations is not the same, the computation time also would be different for either the real or the imaginary part and that could result in timing issues during the operation of the complex multiplier.

For that reason for the implementation of the complex multiplier used for the twiddle factor multiplications the 1st multiplication scheme has been chosen although it requires 4 trivial multipliers instead of 3 and thus more hardware.
A schematic representation of the complex multiplication scheme is shown in Figure 59:

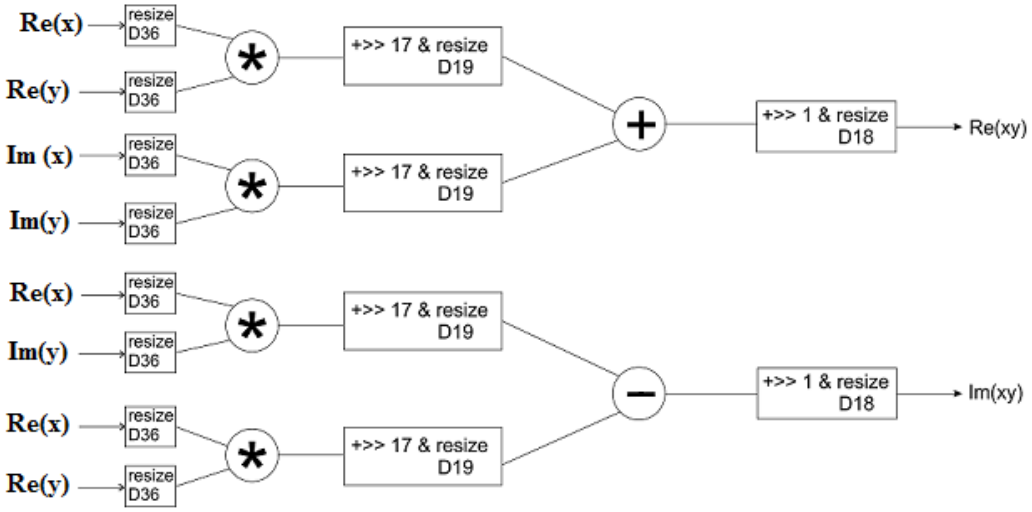

Figure 59: schematic representation of the used complex multiplier

In Figure 59 it can be seen that the inputs of the complex multiplier are resized to 36-bits. That is necessary only due to the way Cλash is performing a multiplication. Furthermore, because an addition/subtraction needs to be performed the necessary resize and shifting also needs to be added.

The resulting Cλash code for the complex multiplier is:

```
-- complex vector multiplier with twiddle factors list embedded
v_twmulti wsv cntr (inR, inI) = (cntrN, output)
                    where
                    cntrN = cntr + 1
                    (wRe, wIm) = wsv ! (fromUnsigned cntr)
                    (wRe36,wIm36) = (resizeSigned wRe, resizeSigned wIm)::ComplexD36
                    (inR36, inI36) = (resizeSigned inR, resizeSigned inI)::ComplexD36
                    a = resizeSigned (shiftR ( inR36 * wRe36 ) 17) :: Signed D19
                    b = resizeSigned (shiftR ( inI36 * wIm36 ) 17) :: Signed D19
                    c = resizeSigned (shiftR ( inR36 * wIm36 ) 17) :: Signed D19
                    d = resizeSigned (shiftR ( inR36 * wRe36 ) 17) :: Signed D19
                    output = (resizeSigned (shiftR (a-b) 1), resizeSigned (shiftR (a+b) 1))
```

Figure 60: Cλash code for fixed point implementation of complex multiplier

Regarding the twiddle factors, unfortunately Cλash doesn't recognize floating type numbers. That means that the twiddle factors has to be transformed into a type that Cλash recognizes. The only possible type is Integers.
The biggest value a twiddle factor can have is 1, (W_N^0 = 1+j0, 0+j1, -1+j0, 0-j1), all the other twiddle factors are smaller that these 4 values. From that fact it can be realized that instead of using 1 bit for

the integer part and 17 bits for the fractional part all the 18 bits can be used for representing the fractional part only. By doing that, W_N^0 is not represented as 1 anymore but as 0.99999237.

That representation is not entirely correct but with that way an extra bit for the presentation of the fractional part of the twiddle factor is being earned and furthermore the error in the representation of W_N^0 is smaller than 1%. For transforming the twiddle factors from floating type numbers to integers for the case of W_N^0 the number has to be multiplied with 2^{17}-1 and for all the other twiddle factors just with 2^{17}. For example let's consider the case of a 16-point FFT, then the twiddle factors of stage 1 will be:

Ws	Fractional representation	Integer representation
W0 = 1	(1, 0)	(131072, 0)
W0 = 1	(1, 0)	(131072, 0)
W0 = 1	(1, 0)	(131072, 0)
W0 = 1	(1, 0)	(131072, 0)
W0 = 1	(1, 0)	(131072, 0)
W2 = exp(-jπ/4)	(0,707, -0,707)	(92668, -92668)
W4 = exp(-jπ/2)	(0, -1)	(0, -131072)
W6 = exp(-j3π/4)	(-0,707, -0,707)	(-92668, -92668)
W0 = 1	(1, 0)	(131072, 0)
W1 = exp(-jπ/8)	(0,923, -0,382)	(120979, -50069)
W2 = exp(-jπ/4)	(0,707, -0,707)	(92668, -92668)
W3 = exp(-j3π/8)	(0,382, -0,923)	(50069, -120979)
W0 = 1	(1, 0)	(131072, 0)
W3 = exp(-j3π/8)	(0,382, -0,923)	(50069, -120979)
W6 = exp(-j3π/4)	(-0,707, -0,707)	(-92668, -92668)
W9 = exp(-j9π/8)	(-0,923, 0,382)	(-120979, 50069)

Table 8: twiddle factors transformation from fractional to integers for a 16-point FFT case

As it has been referred previously Cλash doesn't recognize the Haskell notation of complex numbers (a :+ b). As it has been done for the basic calculus functions the function for generating the twiddle factors has also to be changed so that the resulted twiddle factors to be expressed in tuple form.

The Haskell code for the fixed point twiddle factors generation function is:

```
import Data.List

type Sample = Complex Float

tfls:: Int -> Int -> [Sample]
tfls   n s = (ws)
       where
       m = fromIntegral (n `div` (4^(s-1)))
       qs = [0,2,1,3]
       ls = [0..(n `div` (4^s))-1]
       lqs = concat (map (\q -> map (*q) ls) qs)
       ws_pol = map (((-2*pi)/m)*) (map fromIntegral lqs)
       ws = map cis ws_pol

tfls2fp :: [Sample] -> [(Int, Int)]
tfls2fp ss = ssT
  where
    r2i r = if ((abs r) == 1.0)
               then round (r * 2^17) - 1
               else round (r * 2^17)
    ssT = map (\c -> (r2i (realPart c), r2i (imagPart c))) ss
```

Figure 61: Haskell code of the changed twiddle factors generation function

The **tfls** function, which actually produces the twiddle factors, has remained unchanged. The transformation from fractional to integer numbers and the representation in tuple form is performed by the **tfls2fp** function.
Having transformed the twiddle factors to integers and represented them in tuple form, they can be imported in the Cλash code and be used.

The butterflies can be combined with the complex multiplier to create the basic building block of each stage of our FFT chain. The same procedure with the one for the butterflies will be followed for creating the basic building block of each stage. So for example, the code for the building block of stage 3 will be,

```
-- stage 3 of FFT_clash chain
vbf12ms3 :: ComplexD18Vect16 -> (Unsigned D4, Unsigned D4) -> (((ComplexD18Vect8,Unsigned D4), (ComplexD18Vect4,Unsigned D4), Unsigned D4)) -> ComplexD18
          -> (((ComplexD18Vect8,Unsigned D4), (ComplexD18Vect4,Unsigned D4), Unsigned D4), ComplexD18)
vbf12ms3    wsv (n1, n2) (vbf1st, vbf2st, mcntr) input = ((vbf1stN, vbf2stN, mcntrN), output)
                  where
                  (vbf1stN, bf1out) = vbf1s3 n1 vbf1st input
                  (vbf2stN, bf2out) = vbf2s3 n2 vbf2st bf1out
                  (mcntrN, output) = v_twmulti wsv mcntr bf2out
```

Figure 62: Cλash code for building block of stage 3 for the FFT chain

Finally after all the building blocks of all stages of our FFT have been instantiated they are simply being combined in one component or function (if you prefer) to create the 256-point FFT.

As a last remark, we can say that for producing the Cλash code first we are creating all the necessary functions without determine the types of the input/output arguments and afterwards we instantiate each of these functions with specific types to create all the necessary components of our final circuit. Like that the final code becomes more compact and easier to read.

3.4.2) Implementation of Partially Parallelized 1k-points FFT

As it is described in Chapter 2, from the PFS structure 4 parallel outputs are being produced. So the final FFT has to be able to process 4 parallel inputs at once. That means that the 1k-point FFT needs to be parallelized or in other words to be split in 4 parallel 256-point FFTs.
Ideally we would like to use the already implemented 256-point radix-2^2 SDF FFT as one of the 4 parallel FFTs in order to create the final 1k-point FFT.
If the CFA algorithm and the flow graph of an 16-point FFT are reconsidered, it can be realized that after the first butterfly decomposition the FFT itself is being split in 4 data independent FFTs of size N/4, where N: the initial size of the FFT. That splitting is valid for every size. That means that whichever the size of the FFT the first stage of butterfly decomposition will result in 4 independent N/4-point FFTs which can run in parallel. This is shown in Figure 63:

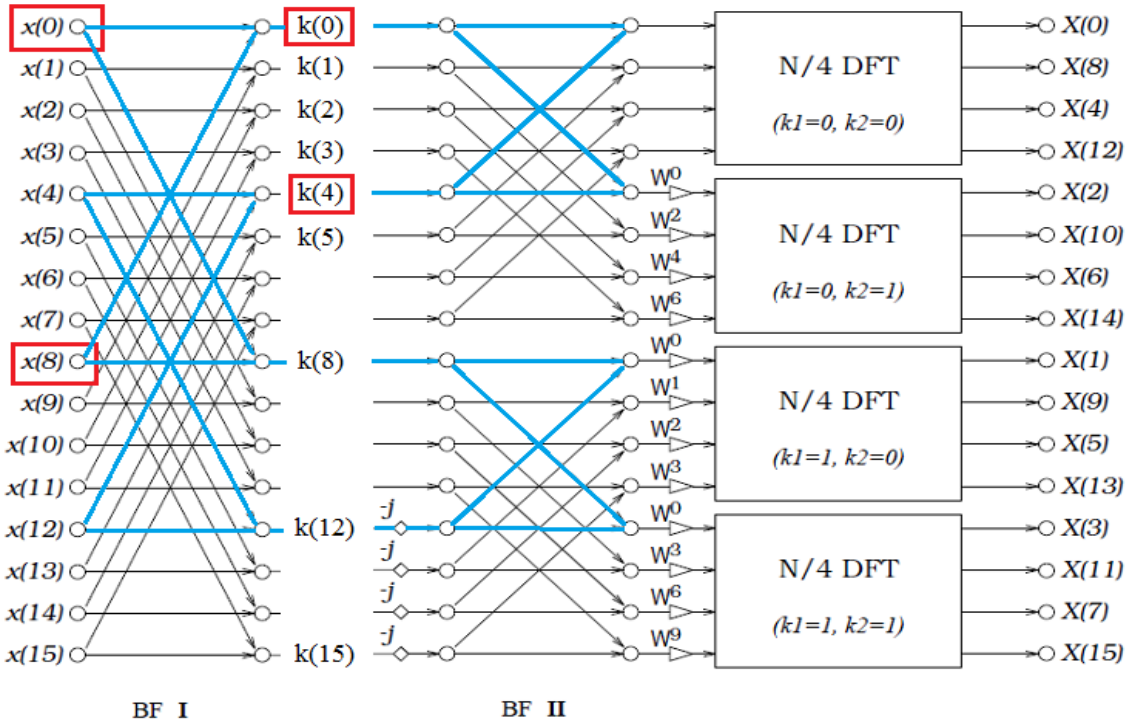

Figure 63: illustration of FFT splitting from the CFA algorithm

So since the FFT is being split in 4 parallel FFTs after its first stage it now remains to split that first stage into 4 parallel structures each one of which will be combined with one of the 4 independent N/4-points FFTs. But as it can be seen in Figure 63 this first stage cannot be split in 4 independent parts and that is due to the data dependences throughout the whole structure. For example let's consider the BFI and the first butterfly operation, it requires x(0) and x(8) signals in order to be computed. That means that those signals should be included in the same FFT structure, but if we move to BFII for the first butterfly operation the k(0) and k(4) are being required and thus those signals also should be included in the same FFT. So finally it seems that there is a pattern to split the BFI phase in 4 parallel phases which are independent, but the BFII phase cannot be split because it requires data from different FFT structures from BFI phase.

Therefore the final 1k-points FFT cannot be fully parallelized. That means that it will consists of 4 256-points FFTs using the structure discussed at §3.4.1 and an initial stage which will be directly derived from the data-flow diagram. Figure 64 shows a schematic overview of this final 1k-points R2^{2}SDF partially parallelized FFT,

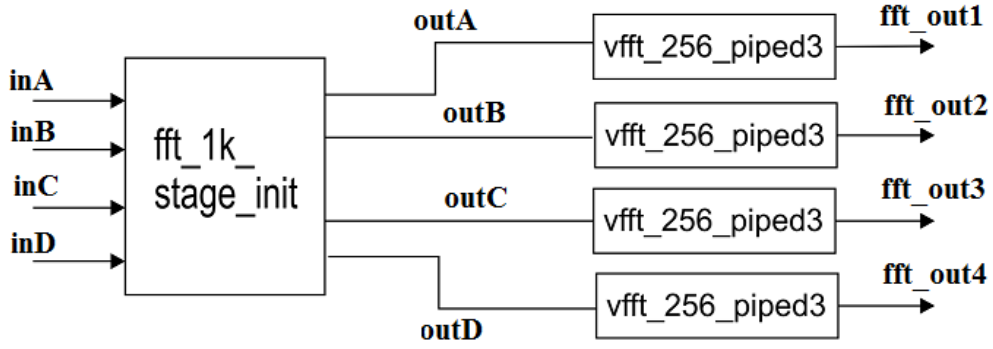

Figure 64: combination of initial stage and 4 256-points FFT for the parallelized 1k-points FFT

The initial stage will be implemented directly as it can be seen in the flow graph, without using the butterfly modules which have been used for the 256-point R2^{2}SDF FFT. In Figure 65 it is shown the schematic representation of that initial stage of the parallelized 1k-point R2^{2}SDF FFT,

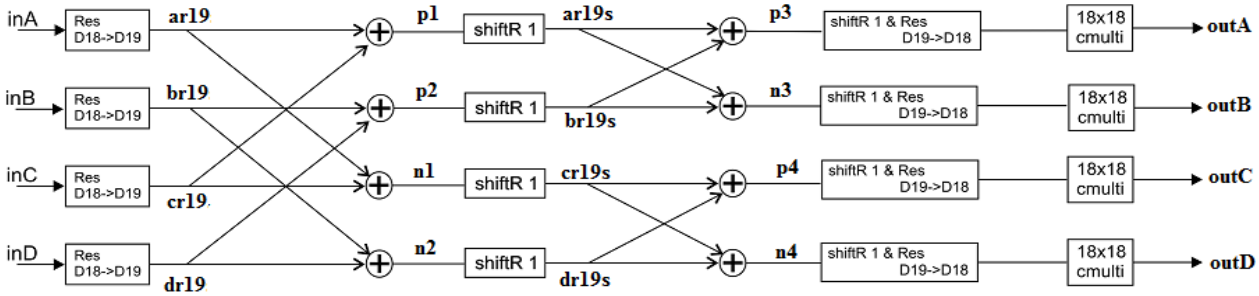

Figure 65: schematic of the initial stage of the FFT implemented in Cλash

The structure shown in Figure 65 is actually the basic butterfly structure of the radix-2^2 algorithm. The first stage of butterfly decomposition at the data-flow diagram [Figure 63] is a simple repetition of that basic butterfly structure. So the whole data-flow diagram can transformed to that structure where the inputs are the following sequences,

- inA :: [x(0), x(1),…, $x\left(N-1/4\right)$]
- inB :: [x(4), x(5),…, $x\left(N-1/4+4\right)$]
- inC :: [x(8), x(9),…, $x\left(N-1/4+8\right)$]
- inD :: [x(12), x(13),…, $x\left(N-1/4+12\right)$]

Regarding the twiddle factors, which are needed at the end of the initial stage for the complex multiplications, in the specific stage instead of 1 list with the necessary twiddle factors 4 different lists are needed. Those 4 lists are derived from the bigger twiddle factors list by dividing that list in 4 parts

and assign each part to one of the complex multipliers respectively. So for an FFT of size N these 4 lists will be:

- $W_A = \left[W^{0\cdot0}, W^{0\cdot1}, W^{0\cdot2} \ldots W^{0\cdot\left(N/4-1\right)}\right]$ → multiple of 0
- $W_B = \left[W^{2\cdot0}, W^{2\cdot1}, W^{2\cdot2} \ldots W^{2\cdot\left(N/4-1\right)}\right]$ → multiple of 2
- $W_C = \left[W^{1\cdot0}, W^{1\cdot1}, W^{1\cdot2} \ldots W^{1\cdot\left(N/4-1\right)}\right]$ → multiple of 1
- $W_D = \left[W^{3\cdot0}, W^{3\cdot1}, W^{3\cdot2} \ldots W^{3\cdot\left(N/4-1\right)}\right]$ → multiple of 3

Furthermore it has to be pointed out that with this structure no delay is introduced to the butterfly operation as it is done from the structure with the two butterflies (BF2I and BF2II) which has been used as the basic building block for the 256-point FFT so far.
Also the necessary resize and shifting operations has been added to avoid overflow problems, as it has been done to the previous FFTs implementations.

The Cλash code for the initial stage of the 1k-point FFT is:

```
bf_stless (inA,inB,inC,inD) = (outA,outB,outC,outD)
              where
              minj = (0, -1) :: ComplexD19
              ar19 = (resizeSigned (fst inA),resizeSigned (snd inA))::ComplexD19
              br19 = (resizeSigned (fst inB),resizeSigned (snd inB))::ComplexD19
              cr19 = (resizeSigned (fst inC),resizeSigned (snd inC))::ComplexD19
              dr19 = (resizeSigned (fst inD),resizeSigned (snd inD))::ComplexD19
              p1 = ar19 !+! cr19
              p2 = br19 !+! dr19
              n1 = ar19 !-! cr19
              n2 = br19 !-! dr19
              ar19s = (shiftR (fst p1) 1,shiftR (snd p1) 1)
              br19s = (shiftR (fst p2) 1,shiftR (snd p2) 1)
              cr19s = (shiftR (fst n1) 1,shiftR (snd n1) 1)
              dr19s = (shiftR (fst n2) 1,shiftR (snd n2) 1)
              p3 = ar19s !+! br19s
              n3 = ar19s !-! br19s
              p4 = cr19s !+! (minj !*! dr19s)
              n4 = cr19s !-! (minj !*! dr19s)
              outA = (resizeSigned (shiftR (fst p3) 1),resizeSigned (shiftR (snd p3) 1))::ComplexD18
              outB = (resizeSigned (shiftR (fst n3) 1),resizeSigned (shiftR (snd n3) 1))::ComplexD18
              outC = (resizeSigned (shiftR (fst p4) 1),resizeSigned (shiftR (snd p4) 1))::ComplexD18
              outD = (resizeSigned (shiftR (fst n4) 1),resizeSigned (shiftR (snd n4) 1))::ComplexD18
```

Figure 66: Cλash code for the initial stage of the 1k-point R2^{2}SDF FFT (without the complex multipliers)

From the code it becomes clearer that the specific structure has no internal states and therefore no delays during the computations. That means that the results of this stage are being produced during the same clock cycle in which the input data arrives at the inputs of the stage.

3.4.3) Reordering RAMs

The data from PFS are produced in four parallel independent streams. But as in most of the FFT architectures, the incoming data needs to be given in a specific order. Therefore there is the need of a reordering structure, which will reorder the incoming data from PFS to the desired order for the FFT. In more detail that desired order is:

CC	255	254	….	11	10	9	8	7	6	5	4	3	2	1	0
INA	255	254	….	11	10	9	8	7	6	5	4	3	2	1	0
INB	511		….	267	266	265	264	263	262	261	260	259	258	257	256
INC	711		….	523	522	521	520	519	518	517	516	515	514	513	512
IND	1023		….	779	778	777	776	775	774	773	772	771	770	769	768

Table 9: order of data from the PFS structure needed in the 1k-point FFT

But the data from the PFS are produced with the following order,

CC	255	254	….	11	10	9	8	7	6	5	4	3	2	1	0
PFS_1	1020	1016	….	44	40	36	32	28	24	20	16	12	8	4	0
PFS_2	1021	1017	….	45	41	37	33	29	25	21	17	13	9	5	1
PFS_3	1022	1018	….	46	42	38	34	30	26	22	18	14	10	6	2
PFS_4	1023	1019	….	47	43	39	35	31	27	23	19	15	11	7	3

Table 10: order of outputted data from the PFS structure

For the reordering structure the most reasonable choice seems to be 4 RAM memories, one for each output of the PFS or one for each of the channels of the FFT.
For those RAMs we created our own function. The code of that RAM memory is:

```
-- homemade RAM --
vRAM mem_st rdaddr wraddr data_in = (mem_stN, data_out)
            where
            data_out = mem_st ! rdaddr
            mem_stN = vreplace mem_st wraddr data_in
```

Figure 67: Clash code for the RAM memory used in the reordering structure

The implementation of the RAM memory is pretty much the standard one.
It has 3 input arguments,

- rdaddr: the address from which the output is being read
- wraddr: the address in which the input is being written
- data_in: the input of the memory

a state argument,

- mem_st: a vector with the contents of all the addresses of the memory

and one output,

- data_out: the output of the memory

Usually there is also a write enable signal but in our case we are always writing to the memories so we chose to skip it since it will always be "High".
From Table 9 and Table 10 it can be realized that the data which are coming at the same time from the PFS needed to be placed in the same channel of the FFT. That means that if simply one RAM is connected with one of the channels of the FFT, it would mean that 4 different values will have to be written at the same time in one of the RAMs, but that is impossible since the available RAMs at the FPGA are dual-port RAMs. That means that it is possible to write and read simultaneously but only at a single address and not to 4 addresses. Therefore a special reordering needs to be done in order to overcome this problem.
That problem can be solved by using a set of multiplexers before and after the RAMs.

In order to explain how and why this reordering needs to be done let's consider the case of a 64-point FFT from now on. So the data that comes from the PFS will be now,

CC	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFS_1	60	56	52	48	44	40	36	32	28	24	20	16	12	8	4	0
PFS_2	61	57	53	49	45	41	37	33	29	25	21	17	13	9	5	1
PFS_3	62	58	54	50	46	42	38	34	30	26	22	18	14	10	6	2
PFS_4	63	59	55	51	47	43	39	35	31	27	23	19	15	11	7	3

Table 11: order of outputted data from the PFS structure in which 64 FIR-filters have been implemented

and the expected data from the FFT is respectively,

CC	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INA	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INB	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
INC	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
IND	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48

Table 12: order of data from the PFS structure needed in the 64-point FFT

So if we consider that for each channel of the FFT (inA, inB, inC, inD) we are going to use one RAM we realize that for example during the 1st cc the 4 outputs from the PFS needs to be written in the same memory, but that is impossible since our RAMs can write only at one address at each time. That means that those data, which arrive in the same clock cycle, needs to be distributed to all the 4 memories and also they need to be distributed in a periodic scheme so that it would be possible to read them and send them to the FFT afterwards.

Table 13 and Table 14 show the data that are needed from the FFT and the memory in which each of them is written. Color coding has been used to give a better overview of the periodicity of the distributed data at the 4 memories.

CC	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INA	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INB	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
INC	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
IND	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48

Table 13: order of data needed in the FFT with color coding for each channel of the FFT

#ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RAM_A	31	46	61	12	27	42	57	8	23	38	53	4	19	34	49	0
RAM_B	47	62	13	28	43	58	9	24	39	54	5	20	35	50	1	16
RAM_C	63	14	29	44	59	10	25	40	55	6	21	36	51	2	17	32
RAM_D	15	30	45	60	11	26	41	56	7	22	37	52	3	18	33	48

Table 14: order of data with color coding written in the 4 RAMs

In more detail, in Table 14 we see that the first 4 outputs of the PFS are being distributed to the 4 RAMs respectively. With that way the problem of multiple writes in a single RAM at the same time is being bypassed. Also due to the periodicity, of which the data are written at the RAMs, it becomes possible to read them afterwards simply by using a set of multiplexers and not any kind of complex structure.
Another way of deriving Table 14 from Table 13 is by taking each column of Table 13 and rotating it downwards. The number of rotations for each column is *R = (number of column) mod 4*. For example the 5th column of Table 14 is derived from the respective column of Table 13 after rotating it downwards 1 position, since for this case it is R = 5 mod 4 = 1.

As it has been referred before in order to be able to write different PFS outputs to each of the RAMs and respectively to be able to read them and send them to the FFT a set of multiplexers at the input and the output of the RAMs will be used.

A schematic diagram of these multiplexers and how they are interconnected with the RAMs is being shown in Figure 68,

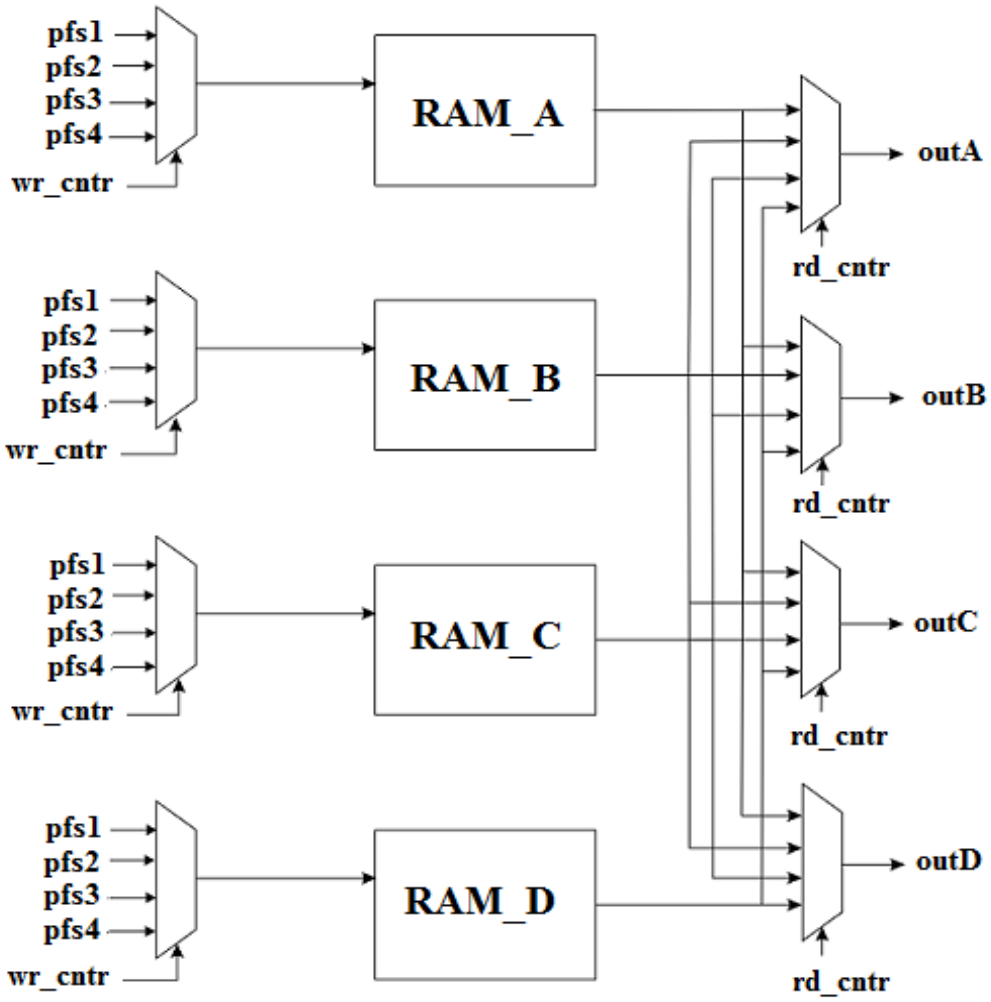

Figure 68: schematic diagram of MUXs set and how they are interconnected with the RAM memories

From Figure 68 we understand that between the PFS and the RAMs there are 4 multiplexes and that each output of the PFS is connected to the 4 RAMs through these 4 MUXs. Like that any PFS output can be directed to any of the RAMs. The same applies for the MUXs after the memories. These multiplexers are located between the memories and the FFT and as shown in Figure 68, any of the RAM's outputs can be directed to any of the 4 channels of the FFT.

Figure 69 shows how the input MUXs and the output MUXs of the reordering structure have been implemented by using *case statements,*

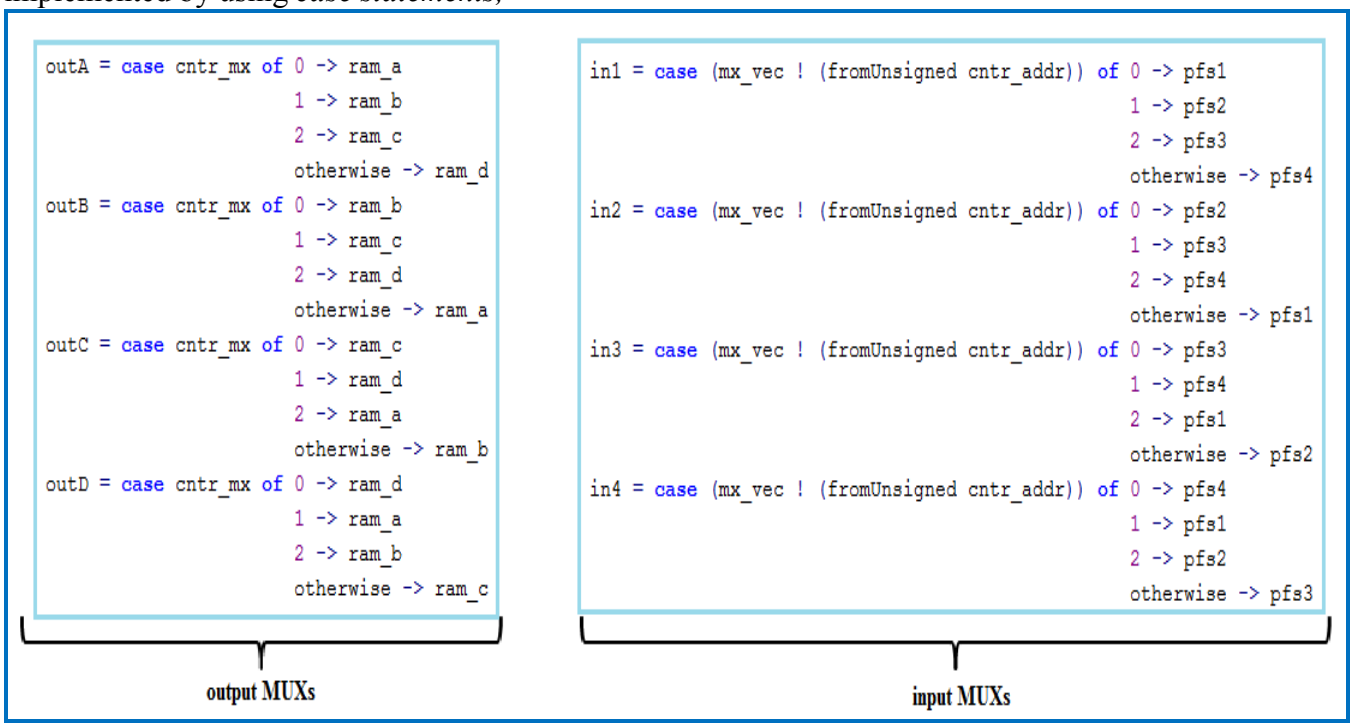

Figure 69: case statement use for implementing the input and output multiplexers of reordering RAMs structure

At the input MUXs the "*fromUnsigned cntr_addr"* is necessary because the *mx_vec* argument in a vector and the *cntr_addr* is an unsigned number.

Furthermore, from both Figure 68 and Figure 69 it becomes obvious that only one counter has been used for the input multiplexers and also only one counter for the output multiplexers instead of instantiate a different counter for each multiplexer.
Let's consider the input MUXs and especially **in1** and **in2**. We see that for in1 the possible cases are [pfs1, pfs2, pfs3, pfs4] and that which of these cases will be outputted to **in1** depends on the counter value (" *mx_vec ! (fromUnsigned cntr_addr* "). Also depending on the same counter the possible cases for **in2** are [pfs2, pfs3, pfs4, pfs1]. So by using the same counter and by altering the sequence of the possible cases we can achieve the desirable output pattern from those multiplexers.

From the implementation of *vRAM* function [Figure 67] it can be seen that there are 2 input arguments *wraddr* and *rdaddr* which give the choice to write and read specific addresses of the memory. So for the implementation of the whole reordering structure a sequence of *wraddr* and *rdaddr* needs to be derived in order to be able to write the data which are coming from the PFS not only to the desirable RAMs but also to the specific addresses of those RAMs.

By checking Table 11 and Table 14 again it can be seen that the exact moment that the data arrive from the PFS and in which RAM and in which address of the specific RAM are being written. For example let's consider again the case of the PFS with 64 FIR-filters and the addresses of RAM_A,

CC	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFS_1	60	56	52	48	44	40	36	32	28	24	20	16	12	8	4	0
PFS_2	61	57	53	49	45	41	37	33	29	25	21	17	13	9	5	1
PFS_3	62	58	54	50	46	42	38	34	30	26	22	18	14	10	6	2
PFS_4	63	59	55	51	47	43	39	35	31	27	23	19	15	11	7	3

#ADDRESS	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RAM_A	31	46	61	12	27	42	57	8	23	38	53	4	19	34	49	0
RAM_B	47	62	13	28	43	58	9	24	39	54	5	20	35	50	1	16
RAM_C	63	14	29	44	59	10	25	40	55	6	21	36	51	2	17	32
RAM_D	15	30	45	60	11	26	41	56	7	22	37	52	3	18	33	48

Table 15: pattern of PFS outputs being written in RAM_A

Table 15 shows which output of the PFS is being send to RAM_A and in which specific addresses is being written. So for RAM_A the sequence of wr_addr will be,

- wr_addr_1 → [0,4,8,12,3,7,11,15,2,6,10,14,1,5,9,13]

By doing the same for the rest of the memories we end up with the following patterns for the wr_addr arguments,

- wr_addr_1 → [0,4,8,12,3,7,11,15,2,6,10,14,1,5,9,13]
- wr_addr_2 → [1,5,9,13,0,4,8,12,3,7,11,15,2,6,10,14]
- wr_addr_3 → [2,6,10,14,1,5,9,13,0,4,8,12,3,7,11,15]
- wr_addr_4 → [3,7,11,15,2,6,10,14,1,5,9,13,0,4,8,12]

Furthermore with a closer look to these patterns it can be realize that they are not randomly produced but they are simply a circular combination of 4 sets of addresses,

- [0,4,8,12]
- [3,7,11,15]
- [2,6,10,14]
- [1,5,9,13]

Figure 70 shows how this circular combination is being structured,

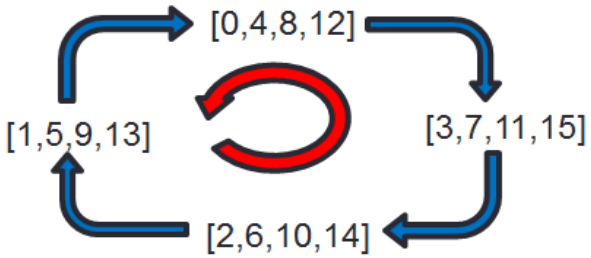

Figure 70: circular combination of the 4 sets of wr_addr used to the 4 RAM memories

In more detail the outer circle (blue arrow) shows in which direction to cycle through the sets in order to create the *wr_addr* sequences for the 4 RAMs.
The inner circle (red arrow) show which set will be the initial set of the sequence.
From the *wr_addr* sequences we see that the initial sets for the 4 sequences are:

- wr_addr_1 → [0,4,8,12]
- wr_addr_2 → [1,5,9,13]
- wr_addr_3 → [2,6,10,14]
- wr_addr_4 → [3,7,11,15]

For example consider the wr_addr_3 (the write address sequence for RAM_C), it is the 3rd sequence so the initial set will be: [2,6,10,14] and after that by following the blue circle we end up with the [3,7,11,15,2,6,10,14,1,5,9,13,0,4,8,12] sequence.

The *rd_addr* sequences are the same for all the 4 RAMs and by checking Table 13 and Table 14 it becomes obvious that the *rd_addr* sequence is **[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]**. That sequence is obvious, because from the columns in these two tables we see that the data are correct but in a different order. But that order is being taken care of from the multiplexers between the memories and the FFT. So in order to send over the correct data from the memories to the FFT we simply have to read all the addresses of the memories consecutively.
Because the final implementation is for a 1024-points FFT and that means that 1024 FIR-filters will be included at the PFS that means that those sequences (*rd_addr* and *wr_addr*) needs to be expanded for that case. Therefore the RAM memories will be of N/4 = 1024/4 = 64 addresses each so that means that the sequences needs to be expanded in such a way in order to cover that range of addresses.

Figure 71 shows these expanded sets for the *wr_addr* sequences.

Figure 71: circular combination of 4 the expanded sets of wr_addr for the case of 1024 FIR-filters

Figure 71 actually shows that the combinations remain the same and only the 4 sets of subsequences are being expanded in order to cover the new range of the addresses.
The *rd_addr* sequence respectively now becomes ***rd_addr*** → **[0,1,2,3,4,…,255].**

3.5) Synthesis results

As been described in the previous section, the final 1k-points FFT block for the Polyphase Filter Bank is composed of the actual FFT and the Reordering Structure.
As it was done for the PFS also the FFT block will be synthesized for the Stratix IV (type EP4SGX230KF40C2) using the Quartus II synthesis tools.

3.5.1) Synthesis Results of the FFT

[a] 256-points R2^{2}SDF FFT

The generation of the hardware can be done by using the "**:vhdl**" command in the Cλash compiler. The resulted VHDL code can be used directly to be synthesized with the *Quartus synthesis tool*.
In Figure 72 is being shown the image from the chip-planner tool of the synthesized hardware in the FPGA,

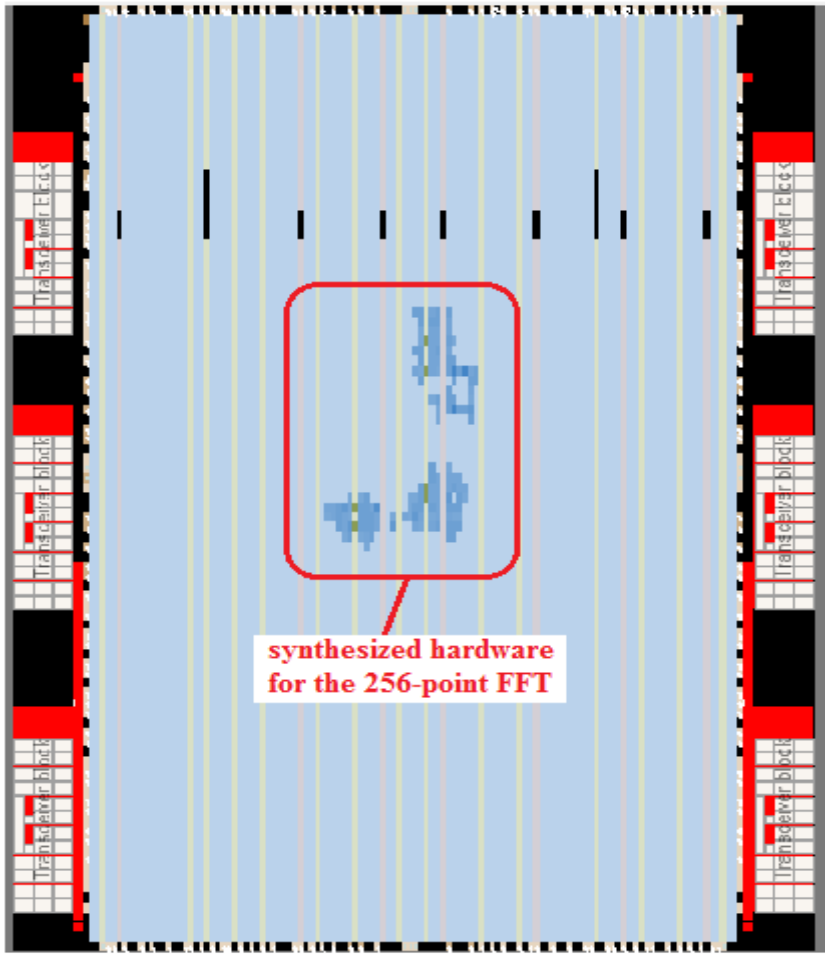

Figure 72: chip planner view of the synthesized hardware for the 256-point FFT

From Figure 72 we observe that the circuit has been placed more or less in the center of the targeted FPGA and also that it covers a relatively small area comparing to the available area of the FPGA.

From the overview summary for the FFT circuit we get the following characteristics:

- # DSP Blocks = 14 (18x18 trivial multipliers)
- Memory = 8640 Bits = 1.05 KBytes
- f_{MAX} = 29 MHz

From Figure 52 we realize that in last stage of the FFT there is no actual need for a complex multiplier since there the twiddle factors are just 1s. The only reason that a complex multiplier has been added in that stage is only because for the implementation of the 256-point FFT a basic building block has been used, which includes a complex multiplier in the end. So in the end only 12 (4 multipliers for every complex multiplier) multipliers are needed and not 16. We expected from Quartus II to had optimize the last multiplier and not include him at all, but it seems that it had done that optimization by half and that is the reason why in the end 14 DSP blocks have been used for the 256-point FFT.
Furthermore we see that the maximum frequency that our FFT can operate is only 29 MHz, which is way too low comparing to the 200 MHz which is the goal frequency according to the specifications.

Figure 73 shows the worst case delay path of the 256-points FFT,

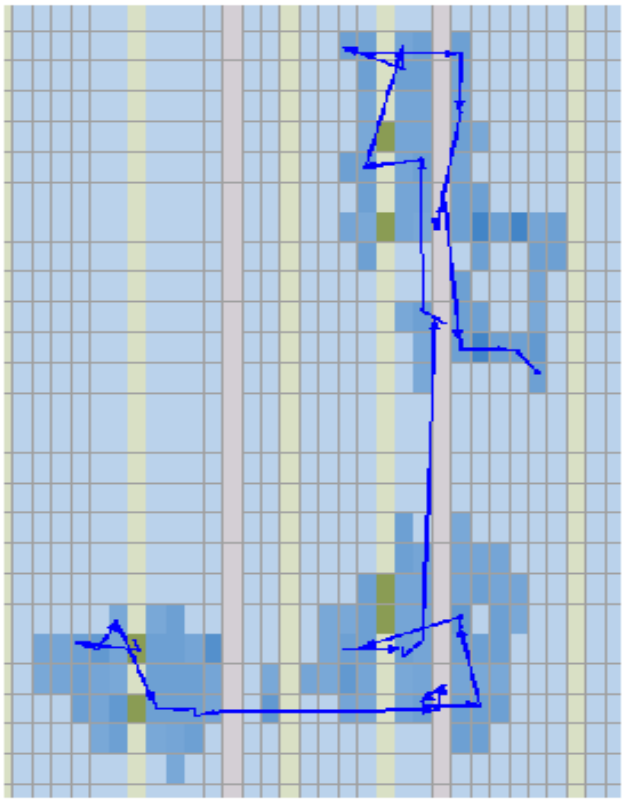

Figure 73: worst case delay path or path with the biggest negative clock slack for the 256-point FFT

From Figure 73 it can be conducted that the worst case delay path is distributed across the whole circuit and it's not concentrated around a specific unit/block of the synthesized circuit. Now if the schematic of the 256-point FFT chain [Figure 52] is being reconsidered then it can be seen that there is a combinatorial path through the whole FFT chain. So if we want to be able to achieve higher clock frequencies for the FFT that combinatorial path has to be shrunk down or split up in smaller ones. That

can easily be done by adding extra pipeline stages. These pipeline stages there are nothing more than simple shift registers.
With a closer look at the FFT chain [Figure 52] it becomes clear that the first place these extra pipeline stages needs to be added is around the complex multipliers. That is because usually multipliers add a relative big delay in any kind of circuit/system and also the complex multipliers are being placed at the end of every computation stage at the FFT chain. So by adding pipeline registers around the complex multipliers the longest combinatorial path is being limited between two consecutive complex multipliers and also each complex multiplier is being isolated.
With these pipeline registers around every complex multiplier the timing/frequency analysis of the circuit is being redone and the maximum operating frequency that the FFT circuit can achieve becomes $\mathbf{f_{MAX} = 145\ MHz}$ which is a rather big improvement but still it's lower than the frequency goal of 200 MHz.
For further increasing the maximum frequency the longest combinatorial path needs to be decreased. In order to do that extra pipeline registers needs to be added between the butterflies of each stage. Like that the longest combinatorial path is being limited between 1 of the butterfly modules. Now the maximum frequency becomes $\mathbf{f_{MAX} = 159\ MHz.}$
From Figure 72 it is obvious that the whole circuit is placed in the center of the FPGA. That means that there should be a relatively long combinatorial path from the I/O pins of the FPGA to the actual inputs/outputs of the circuit. That long path needs to be decreased and for that pipeline registers are being added between the I/O pins and the input/output of the circuit. Also an extra shift register between the butterflies of each stage is being added, which means that now there are 2 shift registers between the butterfly modules at each stage. That extra register is necessary because during the FFT computation a –j multiplication needs to be done. That multiplication is being done by the second butterfly BF2 in each stage. With that extra pipeline register the synthesis tool can pull the registers which are around the butterfly inside the structure and wrap them around the –j multiplier.
In Figure 74 is being shown the chip planner view where the shift registers between the I/O pins and the input/output of the circuit are being clearly indicated.

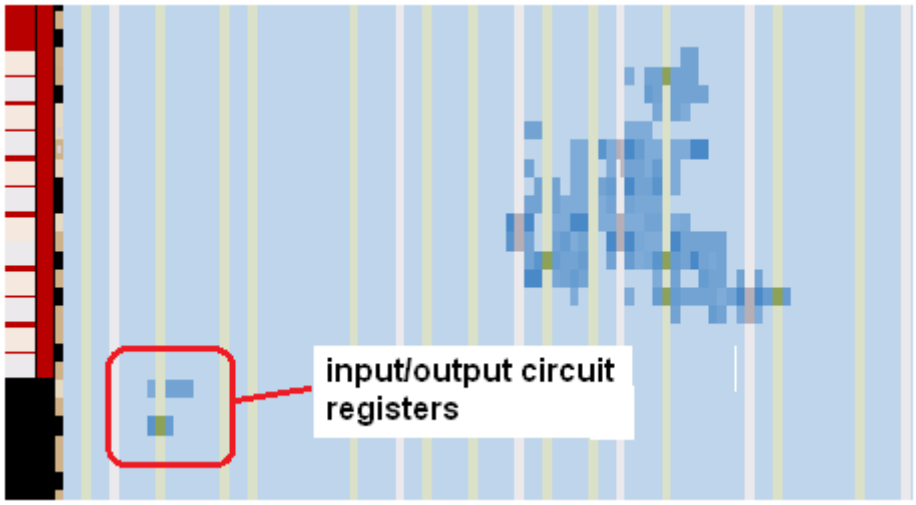

Figure 74: chip planner view of the 256-point FFT with extra pipeline stages between the I/O pins and the input/output of the circuit

Finally with these extra shift registers the maximum operating frequency which our circuit can achieve is $\mathbf{f_{MAX} = 209\ MHz.}$ So the frequency goal finally has been achieved.

In Figure 75 is being shown a schematic representation of the 256-point FFT with the extra pipeline stages,

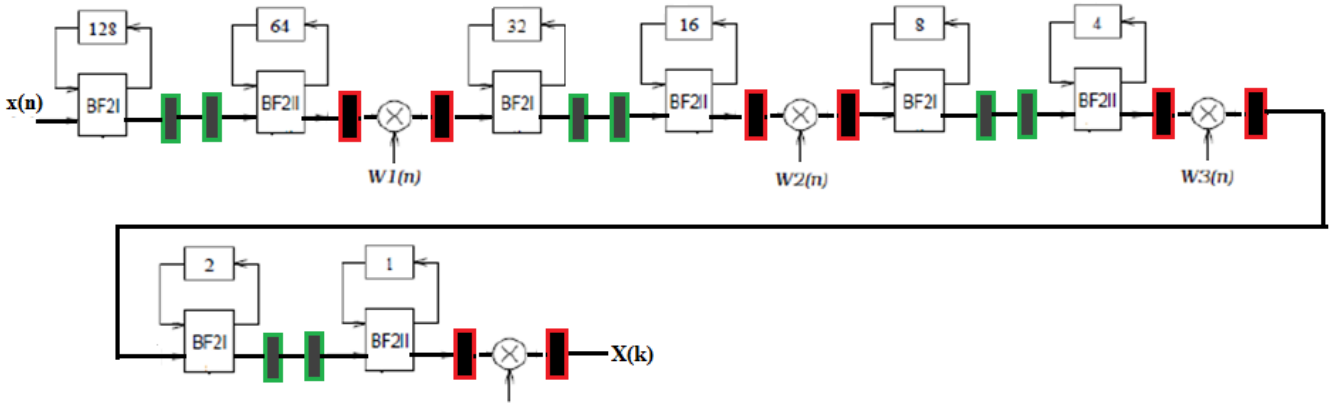

Figure 75: schematic of a 256-point FFT SDF chain with extra pipeline stages

From that schematic [Figure 75] it is clear that with these pipeline stages additional delay is being inserted in the FFT chain and thus all the counters of the butterflies and the multipliers needs to be resynchronized in order to compensate for that extra delay that has been inserted. That synchronization is rather simple. At every counter the total delay which is left to the counter is being subtracted from the initial value of the counter which is being calculated before. In Table 16 are being shown the initial values of all the counters of a 256-point FFT and the new values for the case with the extra shift registers [Figure 75],

	Stage 1			**Stage 2**			**Stage 3**			**Stage 4**		
	BF1	**BF2**	**cmulti**	**BF1**	**BF2**	**cmulti**	**BF1**	**BF2**	**cmulti**	**BF1**	**BF2**	**cmulti**
Initial counters	0	128	64	0	32	16	0	8	4	0	2	2
Subtracted delay	-4	-6	-7	-8	-10	-11	-12	-14	-15	-16	-18	-17
With extra pipeline stages	252	122	57	56	22	5	4	10	5	0	0	3

Table 16: counter values before and after pipelining the FFT chain

We have to keep in mind that the value of each counter in finite, for example the counter of the cmulti in stage 3 can take values from 0 to 15. So in the case where the delay is bigger than the initial value of the counter after reaching 0 it cycles through the whole list and continue the subtraction from the maximum index of the counter. Another way to consider this subtraction is to consider each counter as a list and by starting from the initial value to go upwards to the list as many positions as the delay. At the case where the subtracted delay is bigger than the length of the list and we reach the top of the list we just return to the bottom and continue the subtraction from there. As an example let's consider the case where the initial value of the cmulti counter of stage 3 is 4 and the delay is -15. The specific counter has a range of 16 (from 0 to 15). That means that the new initial value for that counter after subtracting the delay from the pipelining will be 5.

This computation is being shown in Figure 76

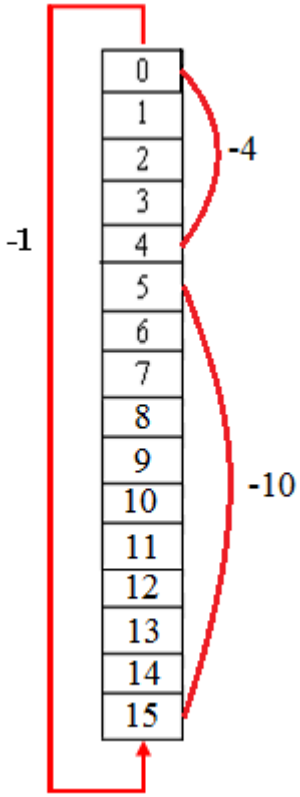

Figure 76: schematic representation of delay subtraction from initial values of counters

[b] 1k-points FFT partially parallelized

The synthesis gives the following results regarding the logic elements and memory usage,

- Logic utilization = 6 %
- # of dedicated logic registers = 5762 (3%)
- # of DSP blocks = 70 (18-bit elements)

Regarding the timing analysis if everything was ideal the fully parallelized FFT circuit, since the initial block doesn't introduce any kind of delay and it is simply the basic butterfly structure of the radix-2^2 algorithm, that would mean that the synthesis results would be the desired ones but unfortunately that is not the case. From the synthesis results for the combined circuit the maximum frequency which the FFT can achieve has dropped to **140 MHz**. That is below the 200 MHz which is the goal frequency for the final FFT. So it seems that the initial stage also has to be further pipelined. As it was done for the 256-points FFT the same reasoning on adding the necessary pipeline registers will be followed in this case. So we end up with the following pipelined structure of the initial stage,

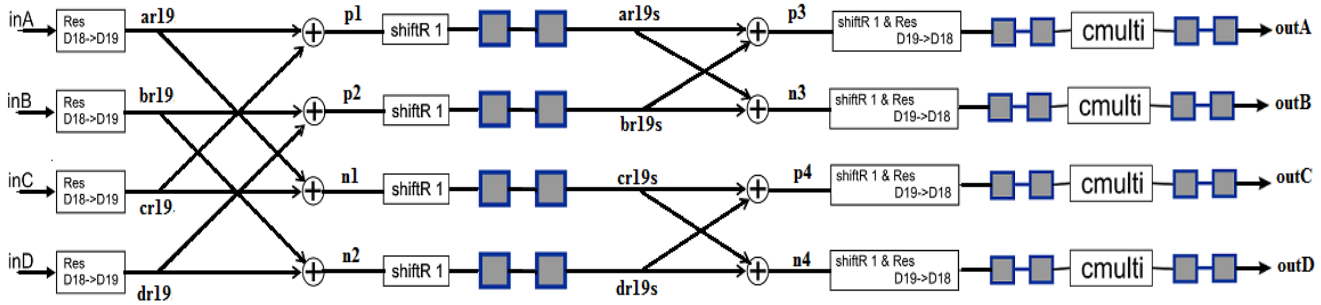

Figure 77: initial stage of the 1k-points FFT with pipeline registers

In Figure 77 it can be seen that 2 shift registers has been added in the middle of the butterfly structure of the initial stage (between BFI and BFII) and 4 shift registers has been wrapped around each complex multiplier. Around the complex multipliers 4 registers has been added specifically so the synthesis tool will be able to pull 2 of them inside and wrap them around the inner multipliers.
But even with these pipeline registers the maximum achievable frequency is **195 MHz**, which is almost the frequency goal.

From the synthesis results we can see that after pipelining the worst case delay path for the circuit is being located between registers and adders. In more detail that path was actually the interconnection between 2 adders and that means that there is a relatively long combinatorial path between those two adders. So now the bottleneck of the system is not a specific component (multiplier or adder) as it was for the circuit of 256-points FFT but it is the interconnections between those components or the registers of them which are being used in the circuit.

3.5.2) Synthesis Results of Reordering RAMs block

As it was done for the case of the Pre-Filter Structure where all the memory elements were translated to a set of registers and not to RAM blocks, as it was expected, also in the case of the reordering RAMs no actual RAMs have been instantiated from the Cλash code. By doing some calculations about the memory that it will be needed from the reordering RAMs we end up with the following,
256 addresses **x** 18-bits = 4608 Bits → 4 RAMs x 4608 Bits = 18432 Bits (2.25 Kbytes)

If only the reordering RAMs will be synthesized will get the following synthesis results,

- Logic Utilization → 24 %
- # of dedicated logic registers → 36866 (22%)
- # of memory blocks → 0 (0%)

So from the synthesis results no RAM blocks have been instantiated and that the logic elements utilization of the FPGA has reached 24%. In other words almost ¼ of the total logic elements have been used.
From the timing analysis the Reordering RAMs block can achieve operating frequencies up to **225 MHz**.

So from the synthesis and timing analysis results it can be concluded that once again the fact that instead of RAM blocks registers are being instantiated comprises the bottleneck in the performance of our system because for a block like that which is being composed by 4 block memories the maximum achievable frequency would be expected to be much higher than the 225 MHz.

3.6) Conclusions

From the Haskell implementation of the FFT and based to our former experience, it can be concluded that once again the resulted code is relatively small and condensed. That makes it easier to read it and keep a better overview of the whole code.
The Cλash description of the FFT (256-point FFT) also didn't need a lot of changes comparing to the Haskell description. Of course the necessary blocks (resize, shift) need to be added in order to prevent overflow problems but something like that was more or less expected when you go from generic descriptions to finite length descriptions.
Something that needs to be pointed out is that the clock which will be used in the final FFT circuit is being added automatically from the Cλash compiler without the user having any kind of control on that. That means that during the production of the Haskell/Cλash code the user only has to consider the phases or modes of functionality in which each component of the FFT can be and how these modes needs to be combined in order to achieve the desired functionality.

The expected maximum frequency was ~400 MHz, since the used FPGA (Staratix IV) has DSP blocks which are considered rather advanced and are able to operate in relatively high frequencies (~600 MHz) and furthermore the FFT structure itself is not a complex one, actually it is simply a repetition of the same building block.

But from the timing analysis it seems that for the 256-point R2^{2}SDF FFT the highest achieved frequency is around 210-220 MHz and for the final fully parallelized 1k-points FFT the frequency goal couldn't be achieved. Furthermore the worst case delay path for both circuits is located at the interconnections between DSP blocks and registers. From that it can be concluded that in the end the DSP blocks that are instantiated are the simplest ones and the advanced and fast ones are not being used at all. Also like in the case of the Pre-filter Structure no block-RAMs have been instantiated and all the memory elements are actually a set of registers.
In order to improve the speed of the FFT first of all these special DSP blocks needs to be used and in general the available hardware on the FPGA needs to be used in a more optimal way. That can be done basically by explicitly instantiating certain hardware blocks which are known that can operate in higher frequencies and thus will improve the speed of the FFT.
But with that way the Haskell/Cλash code stops being generic anymore and it only can be applied to a specific FPGA chipset.

Unfortunately due to lack of time the 1k-points FFT has not been combined with the reordering RAMs. Therefore we don't have any synthesis results about the combined FFT of the PFB. Nevertheless, in the end that is not that important since the 1k-points FFT failed to reach the 200 MHz operating frequency goal.

Apart from that, the major bottleneck in the performance of the FFT remains the fact that no block-RAMs have been instantiated. Like in the case of PFS, also at the FFT instead block-RAMs, large sets of registers have been instantiated and therefore the maximum frequency, which the FFT can achieve, is limited.
To conclude for achieving higher operating frequencies it is essential to use block-RAMs and all these advanced DSP blocks, which are available in the FPGA.

Chapter 4 – Conclusions & Future Work

General Conclusions

The primary goal of this thesis is, by using a real life application, to test and determine whether or not Cλash is able to specify and describe such a complex architecture.

Apart from that, in this project we are using a functional programming language (Haskell) to introduce an alternative approach at hardware design of systems. The basic idea behind this approach is to focus on the functionality of the hardware itself and create an abstract description of the hardware. In more detail, as it already shown throughout the whole thesis, the resulting code has been derived directly from the schematic representations of the hardware, which schematic has derived directly from its mathematical description. In other words the basic procedure for producing the functional code of a piece of hardware is the following,
Mathematical Description ⇨ Schematic Representation ⇨ Functional Code
With such an approach the user/designer gains a level of abstraction at the hardware description and thus the resulting code can become as generic as possible.
.
From the synthesis results it is shown that in the end the implemented Polyphase Filter Bank cannot fit in a single FPGA, something that was one of the basic requirements. The main reason for that is that Cλash is not able to instantiate RAM blocks. Instead of a RAMs, sets of registers are being instantiated. That increases the memory usage of the system dramatically and it comprises the major bottleneck at the performance of the PFB. In more detail due to these big sets of registers it becomes really difficult for Quartus to place them at the FPGA in an optimal way and thus the performance of the PFB diminishes.

In the end it can be concluded that by using a functional approach in hardware design, a rather generic description can be achieved without a lot of effort and the resulting code is rather condensed and easy to understand, as soon as someone familiarizes himself with the language notation.
Furthermore by using Cλash to produce the VHDL description of our hardware is relatively easy and the resulting description is acquired also without a lot of effort. But from the synthesis results, it can be concluded that the resulting VHDL description is not the most optimum and that is being reflected to the overall performance of the implemented PFB.

Future Work/ Suggestions

The secondary goal of this thesis is by using Cλash in a real life application to come up with a number of suggestions for improvement of Cλash, so that it will be able to evolve in a more dependable and adequate HDL..
From this work there are basically 4 aspects that Cλash can be improved.

[1]. RAM blocks instantiation

It has been shown that the major bottleneck in the performance of the implemented system is that no RAM blocks have been instantiated. So another scope for improvement o is that the translation of memory elements from Cλash to VHDL would result in RAM blocks. That can be done either by changing the way any memory blocks are being translated from Cλash to VHDL code or by giving the control of the memory blocks that are being instantiated to the user, so that he will be responsible for choosing which memory elements to be translated in a specific type of memory

[2]. More efficient way of code evaluation

During the translation to VHDL code of the Pre-Filter Structure it was shown that for the initial size of the PFS (1024 FIR-filters) the generation of the VHDL code wasn't possible due to lack of memory. The reason behind that is the way that Cλash is evaluating any function. So in order to be able to implement systems which are that big as the PFS and even bigger either the way that functions are being evaluated or the compiler itself to be able to provide more available memory during the evaluation

[3]. More readable generated VHDL code

Cλash compiler makes sure that the used names for any kind of signals at the VHDL code are unique and random, that results in rather big names with no logical consistency. But that is making the resulting VHDL code almost impossible to read. Ofcourse the idea behind Cλash is that the user won't need to read the VHDL code, but there might be a case where the VHDL code needs to be altered in order to have a more robust description of the hardware, like for example at the case of the FFT block where the VHDL code can be altered so that specific IP blocks to be instantiated and improve the performance of the circuit. So a more readable VHDL code would give in a general that scope for improvement to the user/designer.

[4]. Choice of special IP blocks instantiation and more readable VHDL code

From the synthesis results of the PFB it has been shown that the instantiated IP blocks were the simplest ones and that if the more advanced blocks of the selected FPGA were used then higher operation frequencies could be achieved. So another scope for improvement is exactly that choice. In more detail the user should have control on the IP blocks that will be instantiated and like that to be able to choose certain IP blocks, which are known that are more suitable for the specific application.

References / Bibliography

[1]. **A New Approach to Pipeline FFT Processor**
Shousheng He, Mats Torkelson
"Parallel Processing Symposium, 1996., Proceedings of IPPS '96, The 10th International"
[http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=508145]

[2]. **Digital Signal Processing - principles, algorithms and applications**
John G. Proakis and Dimitris G. Manolakis
4th ed., chap 1-10, Prentice-Hall, 2007

[3]. **Understanding Digital Signal Processing**
Lyons, Richard G.
Addison Wesley 1997

[4]. **CλasH: From Haskell To Hardware**
Baaij C.P.R
Master Thesis, University Twente, 2009
[http://eprints.eemcs.utwente.nl/17922/]

[5]. **Power efficient parallel signal processing chip for radio astronomy**
Gerard Bos
Master Thesis, Univeristy Twente, 2010

[6]. **A new Prime Factor FFT Algorithm and its Index Mapping**
Rong Zheng
Industrial Technology, 1994. Proceedings of the IEEE International Conference on
[http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=467059]

[7]. **A systolic Array Implementation of Common Factor Algorithm to Compute DFT**
Shousheng He, Mats Torkelson
Parallel Architectures, Algorithms and Networks, 1994, (ISPAN)

[8]. **Parallelism in FFT Hardware**
Ben Gold, Theodore Bially
IEEE transactions, Audio Electroacoustics, AU-21, 5-16 (1973)

[9]. **CλasH: From Haskell To Hardware**
C.P.R Baaij
Master's Thesis, University of Twente, (2009)

[10]. **Learn You a Haskell for Great Good - a Beginners Guide**
Miran Lipovaca
[http://learnyouahaskell.com/]

[11]. **A tutorial to Cλash: From Haskell to Hardware**
Christiaan P.R Baaij
[http://clash.ewi.utwente.nl/ClaSH/Documentation_files/clashreference.pdf]

[12]. **Functional Specifications of Hardware Architecture**
Jan Kuper

[13]. **APERTIF Filter Bank Firmware Specification**
Eric Kooistra, Andre Gunst
ASTRON-RP-474

[14]. **ASTRON LOFAR telescope info**
[http://www.lofar.org/]

[15]. **APERTIF project**
[http://www.astron.nl/general/apertif/apertif]

[16]. **Stratix IV Devices: Documentation**
[http://www.altera.com/literature/lit-stratix-iv.jsp]

[17]. **Differences between direct and transpose form of FIR-filters (Thread)**
[http://www.edaboard.com/thread30463.html]

[18]. **Direct and Transposed form of FIR-filters**
[http://cnx.org/content/m11918/latest]