IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

# Master's Thesis

# Implementation and Optimization of a TTA-based FGPA-Demonstrator for a Complex Filter System

Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Mikroelektronische Systeme
Fachgebiet Architekturen und Systeme

presented by

Tuan Anh Dang
Matriculation Number 3218080
born on: 18.06.1997 in: Hildesheim

| | |
|---|---|
| First Examiner: | Prof. Dr.-Ing. Holger Blume |
| Second Examiner: | Prof. Dr.-Ing. Bernhard Wicht |
| Supervisor: | M. Sc. Jens Karrenbauer |

Hanover, February 11, 2022

# Aufgabenstellung

Diese Seite wird gegen die offizielle Aufgabenstellung ersetzt!

## Erklärung

Ich versichere hiermit, dass ich die vorstehende Arbeit selbständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und sowohl wörtliche, als auch sinngemäßentlehnte Stellen als solche kenntlich gemacht habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Hannover, den 11. Februar 2022

# Contents

**Acronym Index** **VI**

**List of Figures** **VII**

**List of Tables** **IX**

**1. Introduction** **1**

**2. Fundamentals** **2**
2.1. Order Analysis . . . 2
2.1.1. The Keyphasor . . . 3
2.1.2. Traditional Order Tracking . . . 3
2.1.3. Computed Order Tracking . . . 4
2.2. Transport Triggered Architecture . . . 5
2.2.1. Programming . . . 6
2.2.2. In comparison with conventional VLIW architectures . . . 6
2.2.3. TTAs as Application-Specific Instruction-Set Processors . . . 7
2.2.4. TTA based Co-Design Environment . . . 8
2.3. Fixed Point . . . 10
2.3.1. Fixed Point Arithmetic . . . 10
2.4. FIR Filter . . . 14
2.5. Field Programmable Gate Array . . . 16

**3. Structure of the Filter System** **17**
3.1. Overview . . . 17
3.1.1. Input Values . . . 17
3.1.2. Signal Flow . . . 17
3.2. Band-pass Filter . . . 19
3.3. Envelope Detector . . . 19
3.4. Decimation Filter . . . 20
3.5. Interpolation Filter . . . 21
3.6. Fast Fourier Transform . . . 22
3.7. Filter System Parameters . . . 26

**4. Implementation** **29**
4.1. Overview . . . 29
4.2. Hardware Optimization . . . 30
4.2.1. Fixed Point Multiplier . . . 30
4.2.2. TTA Custom Function Units . . . 32
4.3. Streaming I/O . . . 36

4.4. TTA Configuration Implementation . . . . . . . . . . . . . . . . . . . . . . . . 37
    4.4.1. Further Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.5. Top-Level Design Entity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

**5. Evaluation** **41**
5.1. Evaluation Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
    5.1.1. Test Data Sets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
    5.1.2. Filter System over different TTA Configurations . . . . . . . . . . . . . 42
5.2. FPGA Synthesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
    5.2.1. Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
    5.2.2. Resource Utilization . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
    5.2.3. Power Consumption . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**6. Conclusion** **54**

**A. Appendix** **55**

**Bibliography** **60**

# Acronym Index

| | |
|---|---|
| ADD | Adder, Addition |
| ASIC | Application Specific Integrated Circuit |
| ASIP | Application Specific Instruction-Set Processor |
| BOOL | Boolean, Boolean RF |
| DSP | Digital Signal Processor |
| FIR | Finite Impulse Response |
| FPGA | Field Programmable Gate Array |
| FPMAC | Fixed Point Multiply-Accumulate |
| FPMUL | Fixed Point Multiplier |
| FU | Function Unit |
| GCU | Global Control Unit |
| GPP | General Purpose Processor |
| HDL | Hardware Description Language |
| I/O | Input/Output |
| I2C | Inter-Integrated Circuit |
| IMEM | Instruction Memory |
| IP | Intellectual Property |
| LSU | Load-Store Unit |
| LUT | Lookup Table |
| MAC | Multiply-Accumulate |
| MUL | Multiplier, Multiplication |
| RAM | Random Access Memory |
| RF | Register File |
| TTA | Transport Triggered Architecture |
| VHDL | Very High Speed Integrated Circuit Description Language |

# List of Figures

2.1. Keyphasor transducer [1] . . . 3
2.2. Equipment used for traditional order tracking [2] . . . 4
2.3. Equipment used for computed order tracking [2] . . . 5
2.4. Trade-off between efficiency and flexibility of different hardware architectures . . 8
2.5. Simple TTA design . . . 9
2.6. Binary number in integer and Q6.10 fixed point format . . . 10
2.7. Multiplication of 5.6103 · 5.1513 = 28.9003 in Q6.10 format and extracting the Q6.10 result . . . 12
2.8. Division of 5.6103 ÷ 5.1513 = 1,0888 in Q6.10 format . . . 12
2.9. Rotation Mode of CORDIC . . . 13
2.10. A direct form discrete-time FIR filter of order $N$ . . . 15
2.11. Moving averaging filtered signal (blue) on a noisy sinus signal (red) . . . 15

3.1. Overview of the filter system [3] . . . 18
3.2. Hilbert coefficients for the envelope detector . . . 20
3.3. Variable speed wave sampled at constant increments of time [4] . . . 21
3.4. Variable speed wave sampled at constant increments of shaft angle [4] . . . 22
3.5. Signal-flow graph for a radix-2 FFT with length $N = 8$ after the first decomposition in $U(k)$ and $V(K)$ . . . 23
3.6. Temporary signal-flow graph for a radix-2 FFT with length $N = 8$ after the three decomposition . . . 24
3.7. Butterfly diagram of a radix-2 FFT . . . 24
3.8. Signal-flow graph for a radix-2 FFT with length $N = 8$ [5, Bild 4-4] . . . 25

4.1. Workflow for implementing the TTA processor . . . 30
4.2. Multiplier with pre- and post-complement [6, Bild 3.3.11] . . . 31
4.3. Ripple Carry Array Multiplier [6, Bild 3.3.2] . . . 32
4.4. Adding the operations FPMUL and FPMAC to a new function unit in ProDe . . 34
4.5. Adding the HDL implementation for the FPMUL/FPMAC FU to the hardware database . . . 35
4.6. Section of the basic configuration . . . 37
4.7. Section of the pruned_2xlevel_8 configuration . . . 39
4.8. Top-level design entity for the FPGA synthesis . . . 39

5.1. Number of samples at each component . . . 42
5.2. Average cycles per sample for the basic configuration on test setups A and B . . 42
5.3. Average cycles per sample for configuration 1xlevel_1 on test setups A and B . . 43
5.4. Average cycles per sample for configuration 1xlevel_16 on test setups A and B . 44
5.5. Effects of the number of multiplier stages on the amount of clock cycles, max. clock frequency and performance in $[\frac{runs}{s}]$ using the 1xlevel_k configurations . . 45

5.6. Effects of the number of multiplier stages on the amount of clock cycles, max. clock frequency and performance in $[\frac{runs}{s}]$ using the 2xlevel_k configurations . . 46
5.7. Area and performance of the TTA configurations using different amounts of multiplier stages. The triangle and label (p) imply a pruned configuration. . . . . 48
5.8. Photo of the measuring station. The Evaluation Board is powered by an external power supply. The power consumption is measured using a multimeter. . . . . . 49
5.9. Power consumption and performance of the TTA configurations using different amounts of multiplier stages. The triangle and label (p) imply a pruned configuration. 52
5.10. Power consumption and area of the TTA configurations using different amounts of multiplier stages normalized to the performance. The triangle and label (p) imply a pruned configuration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

# List of Tables

3.1. Bit reversal for the decimation-in-time radix-2-FFT . . . . . . . . . . . . . . . . . 26

5.1. Resource utilization and instruction size of selected configurations with different multiplier stages on the FPGA . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.2. Power consumption of selected configurations with a clock frequency at 50 MHz 50
5.3. Overview of the power consumption of the individual components for configuration 2xlevel_8 taken from the power report of Vivado . . . . . . . . . . . . . . . . . . 51

A.1. Overview of the evaluated TTA configuration. . . . . . . . . . . . . . . . . . . . . 56
A.2. Information of the implemented TTA configuration. Values from the FPGA synthesis are taken using test setup A. . . . . . . . . . . . . . . . . . . . . . . . . 58

# 1. Introduction

Nowadays, production machines are becoming increasingly larger and more complex. To monitor the conditions of these ever more interconnected machines, they are equipped with numerous sensors. The sensor data are analyzed to predict wear and defects of the machinery without shutting down or disassembling the machine. In case excessive wear on a component is detected, the damaged part can be easily swapped out before it can further aggravate problems. As such, equipment failures and unscheduled downtime can be reduced to minimize the costs while maintaining the product quality.

In case of rotating machinery, the analysis of the vibrations produced by the machine can convey a lot of information. For example, local faults on rolling element bearings produce a series of impacts that repeat periodically. Due to the characteristic frequencies, it can even be identified if the fault is on the ball itself or on the outer- or inner-race.[7] The vibration analysis used in this thesis is known as *computed order tracking*. Instead of using analog instrumentation like in ordinary order tracking methods, computed order tracking is mostly digital and reduces some disadvantages of traditional order tracking.[2]

To perform computed order tracking on rotating machinery, on-site processing of the data is often required, because the moving and metallic components make it difficult to transport the sensor data over either cables or wireless connections. However, due to the environmental conditions, the space and energy available are limited for the architecture implementing the computed order tracking. An application-specific processor offers high efficiency and should provide the necessary performance for the analysis through specialized functional units.

The goal of this work is to implement and optimize a filter system for computed order tracking on an FPGA using the **transport triggered architecture (TTA)**, which due to its easy configurability is suitable for specialized applications. Using the data obtained by the FPGA synthesis, the different TTA configurations implemented are to be evaluated in terms of performance, power consumption and resource utilization of the FPGA.

The following chapter explains the prerequisites necessary to understand the subsequent chapters. This includes the fundamentals of the order analysis and the transport triggered architecture. Chapter 3 introduces the reference filter system with its individual components and functions. Chapter 4 presents the course of actions to implement and optimize the reference filter system for the TTA. Especially, the addition of custom hardware units will be explained in more detail. The evaluation of the implemented filter system and the implemented TTA configuration will be performed in chapter 5. Finally, chapter 6 summarizes the results and concludes the thesis.

# 2. Fundamentals

This chapter introduces the fundamental concepts used in the later chapters. This includes an overview of the primary topics of this thesis, order analysis and the TTA (transport triggered architecture). Following that, an introduction to fixed point numbers, including both the representation and arithmetic, and a general overview about FIR filters and FPGA is given.

## 2.1. Order Analysis

The analysis of the vibrations produced by rotating machinery is important to predict wear and defects of the machinery. Rotating machinery analysis determines when repairs are required, without shutting down or disassembling the machine. Due to vibration analysis being non-intrusive, the otherwise undetected defects can be discovered while the machinery is operating and repaired preemptively, before the failure of the machine, thus reducing the repair, operating and maintenance costs. This vibration analysis can be performed in either the time or frequency domain.

Every rotating machine produces its own vibrations made up of the combined vibrations of its individual parts. In case of the machine being unbalanced or a component wearing out, characteristic frequencies are being produced, which a proper vibration analysis can identify. For example with rolling element bearings there are 4 possible failing frequencies. Using these frequencies it can be identified if the defects lies on either the outer or inner ring, on the bearing ball itself or even on the cage.[7]

The vibration in the time domain is the overall vibration of the machine. Using a time domain analysis, the motion of particular machine components beneath the transducer can be indicated. In case of the frequency domain, the individual components are being isolated, and the individual frequencies contained in the vibration signal can be revealed. This allows the recognition of the sources of defects and can be used to help plan repairs.

A special case of a frequency domain vibration analysis that is often used is called Order Tracking. Instead of using absolute frequency units (Hertz), order tracking uses multiples of the base speed (orders). While there are other vibration analysis techniques, such as time and frequency analysis, the advantage of order tracking is the analyzing of non-stationary noise and vibration, which vary in amplitude and frequency with the rotational speed. Because it relates the vibration signal to the frequency ratio instead of the absolute frequency, it is insensitive to the variations in shaft speed. Here, order refers to a frequency of a vibration, which is a certain multiple of the reference speed, and can, in this way, be easily identified. An order of 3 would mean a vibration signal with a frequency equal to thrice the reference rotational frequency.[8]

### 2.1.1. The Keyphasor

A common component of most analysis methods is the keyphasor. It is a transducer that produces a voltage pulse for each turn of the shaft. This keyphasor signal is used primarily to measure rotating shaft speeds and serves as a reference for measuring vibration phase angles. The keyphasor transducer is typically either a proximity probe, as seen in figure 2.1, an optical pickup or a magnetic pickup. In case of the proximity probe, a notch is required which provides a trigger for the signal pulse once per full rotation of the shaft.[1]

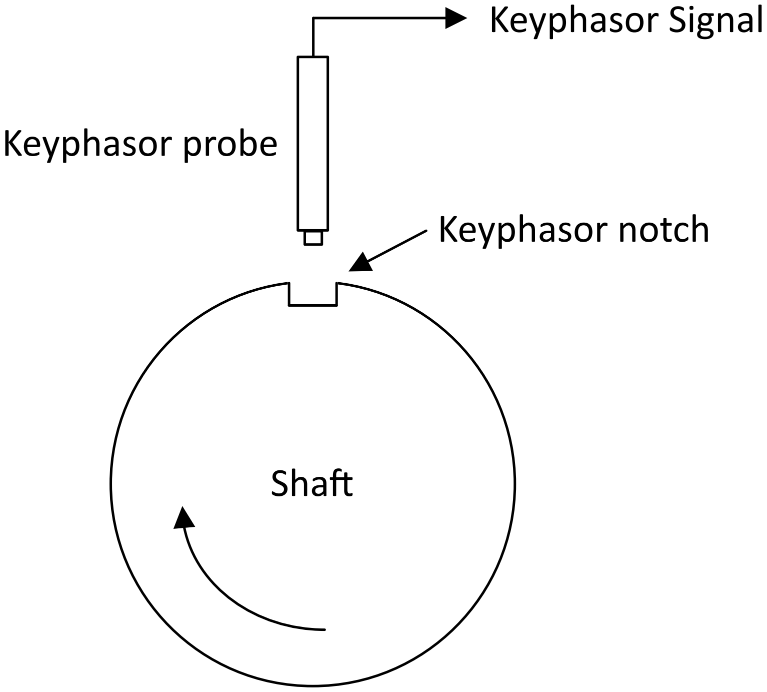


Figure 2.1.: Keyphasor transducer [1]

### 2.1.2. Traditional Order Tracking

Traditionally, order tracking has been performed using analog instrumentation to directly sample the analog vibration signal at constant increments of the shaft angle.[2] A schematic of a traditional order tracking is shown in figure 2.2. It includes a ratio synthesizer, an anti-aliasing tracking filter and a frequency counter to monitor the shaft speed.

Using the keyphasor signal, the ratio synthesizer generates a signal which is proportional to the shaft speed of the machine. This signal is used to control the sampling rate and cut-off frequency of the analog tracking filter. After the data is sampled at a constant shaft angles, a Fast Fourier Transform (FFT) is calculated per block, resulting in an order spectrum.

As one can see, the associated cost and complexity restrict its use for widespread usage of this technique. Furthermore, the analog approach is also prone to error; the equipment used for analog order tracking is known to have troubles when following rapidly changing shaft speeds.[9]

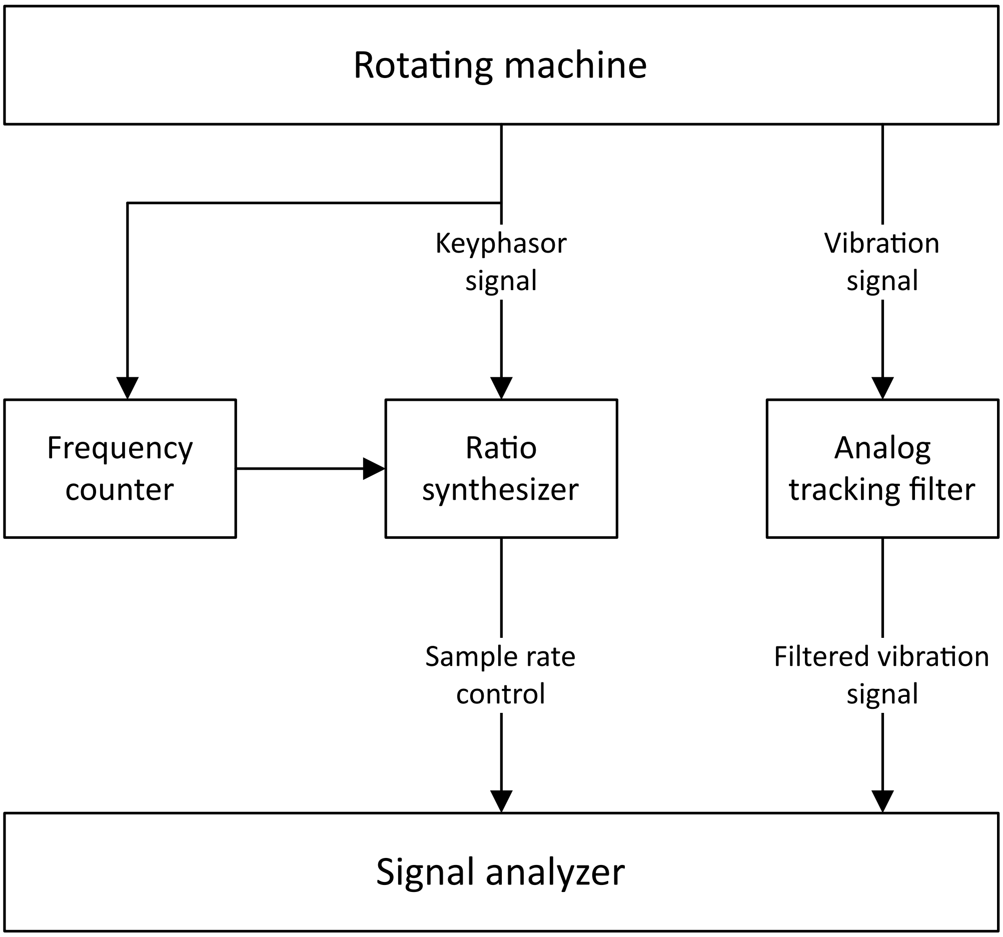


Figure 2.2.: Equipment used for traditional order tracking [2]

### 2.1.3. Computed Order Tracking

In 1989 a computational method for performing order tracking analysis was introduced by Hewlett Packard. The method reduces the dependencies on specialized equipment used in traditional order tracking and claimed to have a higher accuracy compared to analog methods.[10] It uses a method of resampling the signal by interpolation filtering. Soon after, Bruel & Kjaer introduced a different computational method, using decimation and interpolation for resampling the signal, while achieving essentially the same result.[11]

A schematic of a computed order tracking (COT) is shown in figure 2.3. In contrast to the traditional order tracking method, COT is mostly fully digital.

The vibration signal and the tachometer pulse are at first recorded at constant time intervals using conventional hardware. The vibration signal then passes through a low-pass filter and is sampled at constant time increments $\Delta t$. Up to this point, the method resembles a traditional frequency method more than order tracking. The signal will be resampled by software using the keyphasor signal to provide a signal, which is independent of the rotational speed of the machine. The signal is resampled at constant angular increments and is thus unrelated to speed. From this resampled data, the order spectrum will be calculated by applying an FFT, where frequency variances due to varying rotation speeds have been excluded. This way, the frequency domain is changed to the order domain.[2][4]

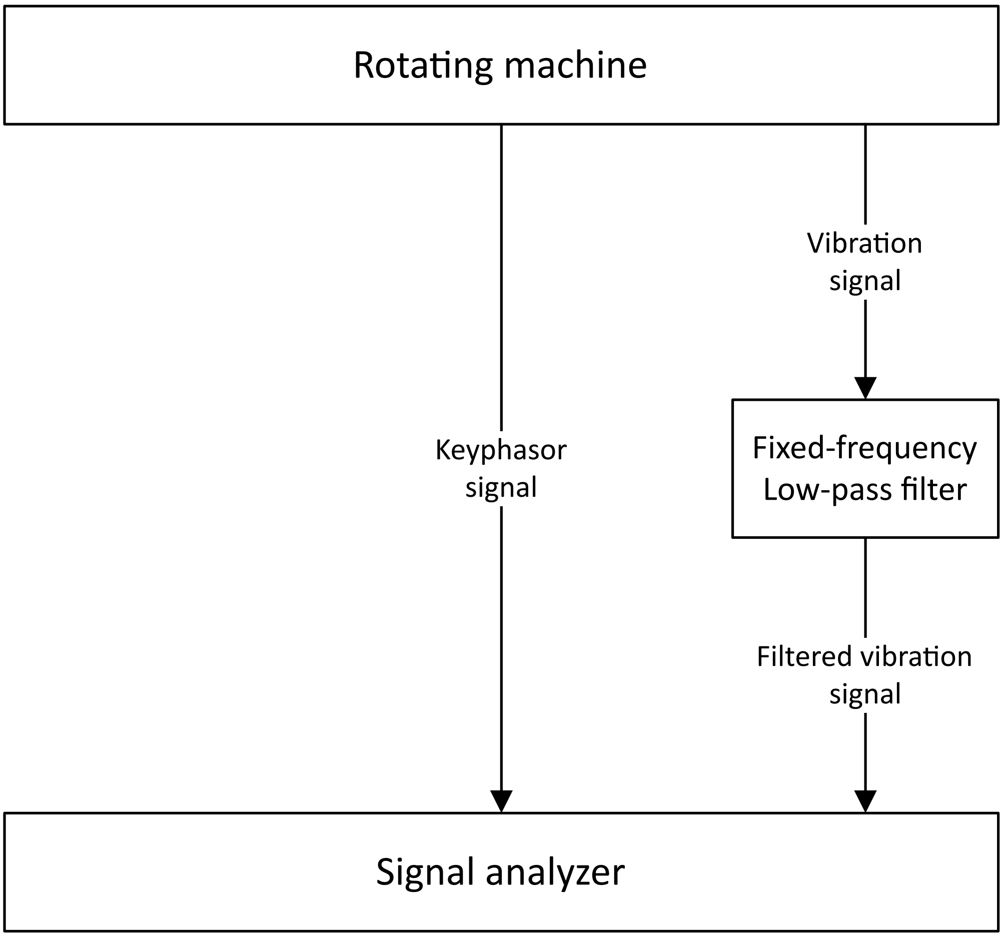


Figure 2.3.: Equipment used for computed order tracking [2]

## 2.2. Transport Triggered Architecture

Transport Triggered Architecture, also known as TTA, is a processor design philosophy, where the internal data paths are exposed in the instruction set and the program directly controls the internal control buses of the processor. A TTA processor consists of Function Units (FUs), a Global Control Units (GCU), Register Files (RFs) and the Interconnection Network (IC), which itself consists of buses and sockets. Sockets are used to connect the ports of FUs and RFs to the transport buses. Through controlling the transport buses and writing data into the triggering port of a FU, the program executes the operations as a side effect of these data transports, also called *moves*.

Usually, a TTA processor has multiple function units and transport buses. The instruction word of the processor consists of one move slot per transport bus, so that per instruction word a transport bus can execute one move each, introducing instruction level parallelism to the processor when using multiple transport buses. While a bus can be connected to every single FU and RF, TTA does not require bus sharing in any way. If they are not optimal for the implementation or use case, the designer can decide to use a less connected bus or even a bus only connecting two units.

### 2.2.1. Programming

In more traditional processor architectures, like the RISC architecture, the processor is typically programmed by defining the executed operations and the operands. The programming model can be more easily shown with the following assembly code example:

1. Traditional "operation triggered"

```
ADD R1, R2, R3
MUL R4, R1, R5
```

The first operation simply adds the value of the registers R2 and R3 and stores the result of the addition into register R1. Here, the execution of the instruction would be translated to control signals, which would result into transferring the current values of registers R2 and R3 into a function unit capable of executing the add operation, which could be the Arithmetic-Logic Unit (ALU). Then a control signal would select and trigger the addition operation in the ALU and finally the result would be transferred back to register R1. Following that, the result saved in register R1 would be multiplied with the value in register R5 and saved in register R4.

In comparison, the TTA does not define the operations, but only the data transport and through that the buses needed to write and read the operand values. Operations are only executed as a side effect though moving data to a triggering port of a function unit. Therefore, executing an addition in TTA requires moves. Each move defines the starting and endpoint for the data transport taking place in the transport bus:

2. Transport triggered

```
R2 -> ADD.OPERAND, R3 -> ADD.TRIGGER
ADD.RESULT -> R1
R1 -> MUL.OPERAND, R5 -> MUL.TRIGGER
MUL.RESULT -> R4
```

Here the TTA allows for some software optimizations such as a register bypassing, so that results can directly be transported to the destination function unit without saving it in a register, if the Add FU and the Mul FU are directly connected:

3. Transport triggered with software bypassing

```
R2 -> ADD.OPERAND, R3 -> ADD.TRIGGER
ADD.RESULT -> MUL.OPERAND, R5 -> MUL.TRIGGER
MUL.RESULT -> R4
```

The result from the adder is directly moved to the operand port of the multiplier, removing the need for register R1 in this example and reducing register pressure.[12]

### 2.2.2. In comparison with conventional VLIW architectures

The goal of Very Long Instruction Word (VLIW) is a higher performance by using instruction level parallelism by running multiple instructions in parallel.

In a VLIW architecture, the compiler encodes multiple operations as an instruction word, at least one operation for each execution unit of a device. This results in a higher performance, smaller hardware complexity, but also a higher complexity of the compiler, who has to schedule the different operations.[13] VLIW suffers from *register file complexity bottleneck*. A register file has to support two parallel reads and a parallel write for each parallel function unit.[14] Moreover, VLIW requires additional hardware logic to enable bypassing, which increases the required resources and reduce the efficiency of the processor.

The TTA exposes more of its internal architecture in the instruction set compared to VLIW. Because the transport buses are exposed, the processor can be so optimized, that unnecessary transport buses and ports for the application are removed, resulting in a smaller and more energy-efficient processor.

As shown above, on TTA register bypassing is done explicitly by programming a data transport directly from one function unit to another. In comparison, VLIW register bypassing is either not done or done indirectly by using extra hardware and instructions. This leads to a lower register file complexity for the TTA, while not requiring additional hardware for the bypassing.

TTA also has other advantages over conventional microprocessors regarding the hardware architecture and compiler optimizations. TTA is flexible, new function units can be easily added to the interconnection network, which can be a custom hardware block with any number of inputs and outputs. Using moves as the means of a data transport gives the compiler also more scheduling freedom.

The main disadvantages of TTA is its low instruction density.[14] This leads to a need for a larger instruction memory compared to other architectures. Other disadvantages are inefficient support for variant immediate values and dynamic-power wasting due to separate scheduling of source operands.[15]

### 2.2.3. TTAs as Application-Specific Instruction-Set Processors

Because there are different types of processor architectures available, there are also different ways to implement the complex filter system. Usually there is always an exchange between flexibility, performance, chip area and power consumption of the processor. Figure 2.4 shows this trade-off between different architectures.

On the upper left side, the General Purpose Processor (GPP) is shown. It provides a huge flexibility for the programmer in exchange for a slower performance. Thus, it is often not able to run applications in real time and not a good choice for a complex filter system, which would rather record the data in real time and display problems on the machines to recognize errors. Furthermore, because of the high flexibility, many different hardware elements are required, resulting in a bigger processor in comparison to the other processor architectures. This results also in a higher power consumption and cost.

While digital filters are usually implemented using Application Specific Integrated Circuits (ASIC) and Digital Signal Processors (DSP)[16], for ASICs a custom chip would have to been designed and manufactured, which would implement the targeted application in hardware resulting in a very high performance at the cost of a very low flexibility. So each time an update would have to be made, a new chip would have to be manufactured. On the other hand, DSPs, though often used for digital filters, have are designed to be flexible to reduce the design costs and to enable

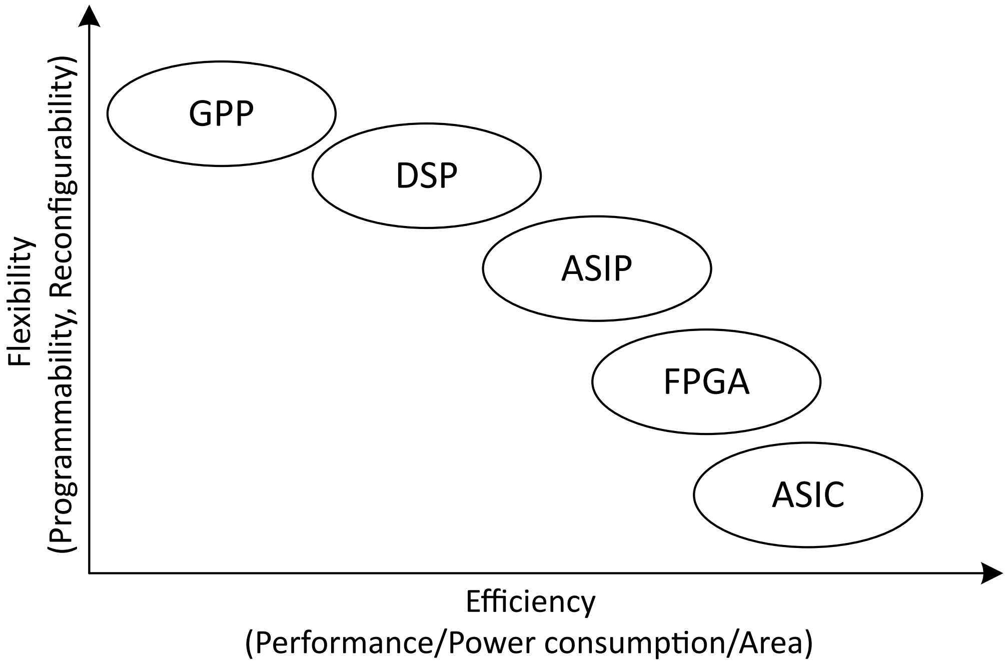

GPP
DSP
ASIP
FPGA
ASIC
Flexibility
(Programmability, Reconfigurability)
Efficiency
(Performance/Power consumption/Area)

Figure 2.4.: Trade-off between efficiency and flexibility of different hardware architectures

them for many different applications. This comes with the disadvantage, that they don't have the necessary instruction sets available for special operations, resulting in a longer cycle count for the operation and, thus, lowering the performance. Additionally, some instruction sets are not needed for the application. Parts of the hardware are not utilized, which also increases the power consumption more than necessary.

Thus, when knowing in advance which kind of application is going to be executed, there is also the option to design an Application-Specific Instruction-Set Processor (ASIP). It is a customized processor which can execute a wider variety of programs, but specially targeted to execute the used application with high performance. The advantage of this approach is, that it is easier to apply new features, adapt the applications or update the supported set of functions without having to design a new integrated circuit.

TTA processors can be easily customized by adding additional function units and register files, creating new function units specially customized for the application and by adding new transport buses and interconnections to improve its capabilities in instructions level parallelism. Thus, TTA is a good template for the design of ASIPs, where custom hardware operations are often important for the performance.

## 2.2.4. TTA based Co-Design Environment

**TTA based Co-Design Environment** (TCE) is an *open application-specific instruction-set toolset*. Developed by the **Customized Parallel Computing** (CPC) group at the Tampere University, Finland, it is used to efficiently design customized processors based on TTA.[17] TCE provides different tools to design and optimize TTA designs and to generate the TTA processors, so that they can be implemented on hardware. A few tools will be introduced in this section, which were used in this work to implement, optimize and generate the TTA processor:

**Processor Designer (ProDe)** is a graphical application to create and edit TTA designs and to print so-called processor architecture definition files (adf).

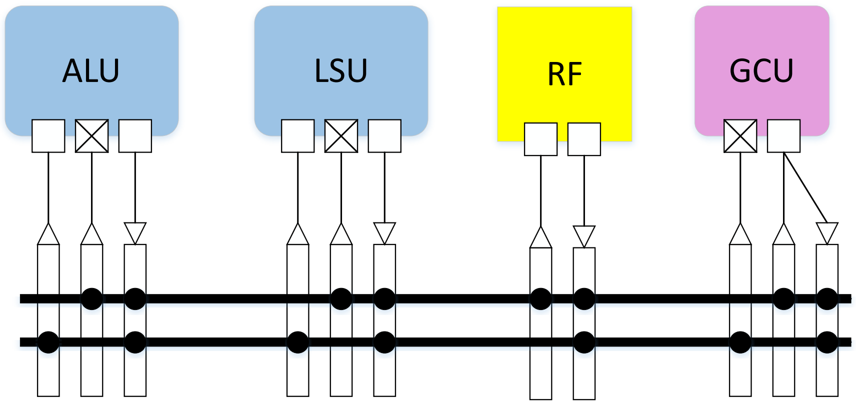


Figure 2.5.: Simple TTA design

Figure 2.5 shows the essential components of a TTA processor. Only the function units with their ports, and the transport buses are visible. The connections of the transport buses to the sockets are shown as black dots. The designer can add additional register files, already existing function units, or even newly created function units specialized for certain functions. Because the designer can choose whether to enable the connection between the transport buses and the sockets, there are exponentially amounts of possibilities of the design.

Using ProDe the designer can select implementations for each component of the processor, which can vary on their latency. Problems can occur here when creating a custom function unit with an unreasonable latency. The stated latency of the custom FU will be used for the simulation, but it will not be checked if the latency of the custom FU will work for the hardware implementation. Timing is completely the responsibility of the programmer. As such, when implementing the TTA on an FPGA, errors can occur here due to not meeting the latency constraints.

The **Hardware Database Editor (HDB)** is a graphical interface to create and modify TTA hardware databases (hdb-files). Using it, new components can be created to be used in ProDe. The programmer can add the hardware description language (HDL) definitions of TTA components to the database (function units, register files, buses and sockets).

**Processor Generator (ProGe)** produces a synthesizable hardware description of the TTA processor using the adf-file and idf-file (implementation definition file), which has the description of the assignment of the implementations to use for each component in the architecture. Using the idf-file, the Processor Generator knows which HDL to use for each component in the adf.

Together with the **Program Image Generator (PIG)**, which generates the bit image, which are to be uploaded to the target machine's memory for execution, the TTA can be synthesized and implemented on an FPGA. In this work, the TTA was synthesized on the FPGA using Vivado.

Other tools of the TCE can be looked up in the TTA-based Co-design Environment User Manual.[18]

## 2.3. Fixed Point

In computing, the usual method to calculate with fractional numbers is with floating point numbers. Nowadays, modern general-purpose CPUs have specially designed floating point units (FPUs) to carry out operations on floating point numbers. The computation on said FPUs is very fast, and most programming languages support data types for this format.

The issue in embedded platforms is, that many embedded processors lack a floating point unit. Integer arithmetic units require fewer logic gates in comparison and consume a smaller chip area than an FPU. While there are options like software emulation for floating point arithmetic, such options are usually too slow for most applications. Instead, the complex filter system will use another option, the fixed point format.

The fixed point format works with fractional numbers, but still uses the integer hardware for the arithmetic. By placing a virtual point after a fixed number of bits and storing a fixed amount of fractional digits of the number, one is able to represent fractional numbers with the integer format, as one can see in figure 2.6.

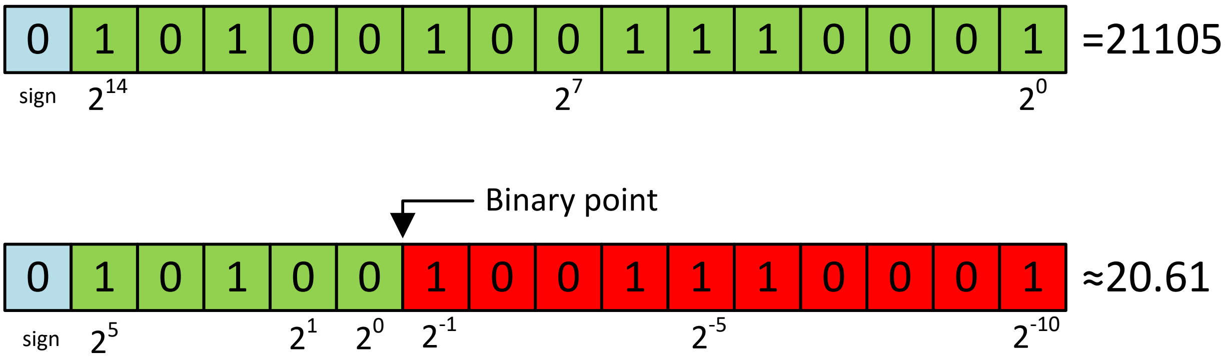


Figure 2.6.: Binary number in integer and Q6.10 fixed point format

The position of the so-called binary point determines the *fixed point format*. The fixed point format declares the number of bits to the left of the binary point, i.e. the number of the integer bits, and the number on the right of the binary points, which are the fractional bits. On the assumption that the total amount of bits stays the same, there is a trade-off between the possible range of the values that can be represented and the precision of the fractional part.

Though, there are multiple notations for fixed points formats, this thesis uses the "Q$I.F$" notation to denote a signed number with $I$ integer bits and $F$ fractional bits. In case of an unsigned number, the notation used would instead be UQ$I.F$.

### 2.3.1. Fixed Point Arithmetic

Up until this point, the difference between fixed point and integer format was purely a matter of interpretation, since the fixed point representation of a fractional number is essentially an integer, which is to be implicitly multiplied by a fixed scaling factor. For example, the value 0.75 can be stored as the integer value 75 with the scaling factor of 1/100. The inherited differences will become more apparent when performing arithmetic on the fixed point numbers.

### Addition and Subtraction

The integer hardware is used to perform arithmetic on the fixed point numbers. Because of that, the hardware will interpret the numbers as integer values. This means that if the programmer uses a fixed point number a, the hardware will interpret this as the same value as $a \cdot 2^F$, with F as the number of fractional bits of the fixed point format. Assuming two fixed point values a and b with the same format Q*I.F* are added or subtracted, it will be performed as followed:

$$a \cdot 2^F \pm b \cdot 2^F = s \cdot 2^F \tag{2.1}$$

One can see, that to add or subtract two fixed point numbers with the same format, it is sufficient to add or subtract the interpreted integers of the hardware. The result will have the same scaling factor $2^F$ and thus has the same fixed point format as the two operands. Unless the input values differ on their fixed point formats, there is no need for an additional modification to obtain the correct result and the result can be stored back in the same program variables as the operand. In case the values have a different scaling factor and thus a different fixed point format, they must be converted to a common fixed point format before the operation can be executed. A downside is, that because using a fixed point format means converting possible integer bits to fractional bits, overflows will happen at lower values.

### Multiplication

The multiplication of fixed point numbers is the first operation that differs from integer arithmetic. Because the hardware interprets the numbers as integers, the hardware will perform the multiplication of fixed point numbers as followed:

$$(a \cdot 2^F) \cdot (b \cdot 2^F) = p \cdot 2^{2F} \tag{2.2}$$

As shown in the equation 2.2, when multiplying two fixed point numbers with format Q*I.F*, the product will have the format Q*2I.2F*. The first thing to note is, that the number of bits to store the result in is doubled, which is not specific to the fixed point format, but a general property of integer multiplication.

The second thing is, that the result of the fixed point multiplication has a different format than the format of the operands. If one wants the same fixed point format as the operands, then the bits need to be shifted to the right by F, which is, mechanically, a simple and fast process in most computers. Additionally, I bits on the left have to be truncated, so that the result will have the same fixed point format as the operands. Figure 2.7 shows this process for a 16 X 16 bit multiplication.

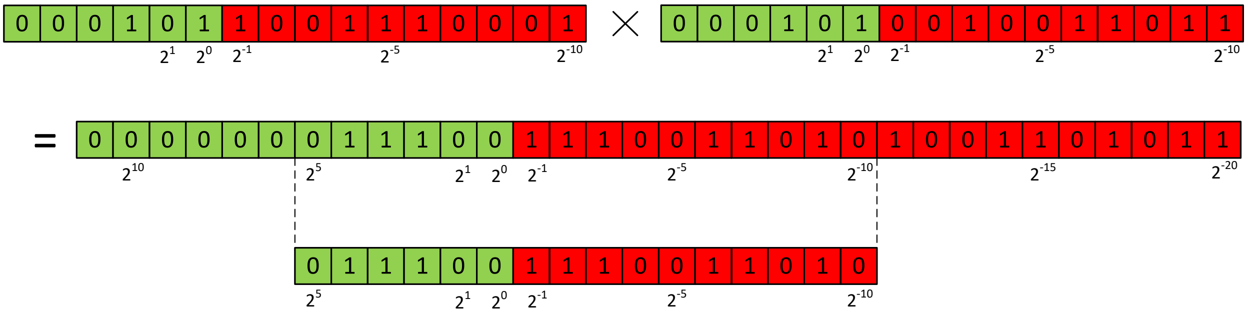


Figure 2.7.: Multiplication of $5.6103 \cdot 5.1513 = 28.9003$ in Q6.10 format and extracting the Q6.10 result

### Division

Another operation that differs from the integer operation is the division of fixed point numbers. Using integer hardware the following calculation is performed:

$$\frac{a \cdot 2^F}{b \cdot 2^F} = q \tag{2.3}$$

As shown in equation 2.3, division reduces the number of fractional bits to 0 of the result. A solution to this would be to shift the numerator by F bits before the division, so that the result would have the correct format, which would result to equation 2.4

$$\frac{a \cdot 2^F \cdot 2^F}{b \cdot 2^F} = q \cdot 2^F \tag{2.4}$$

Figure 2.8 shows this for a division of two 16-bit numbers in fixed point format Q6.10. Something to note is the possibility of an overflow of the numerator after being shifted by F bits, when using a datatype with a lower number of bits than the numerator after being shifted.

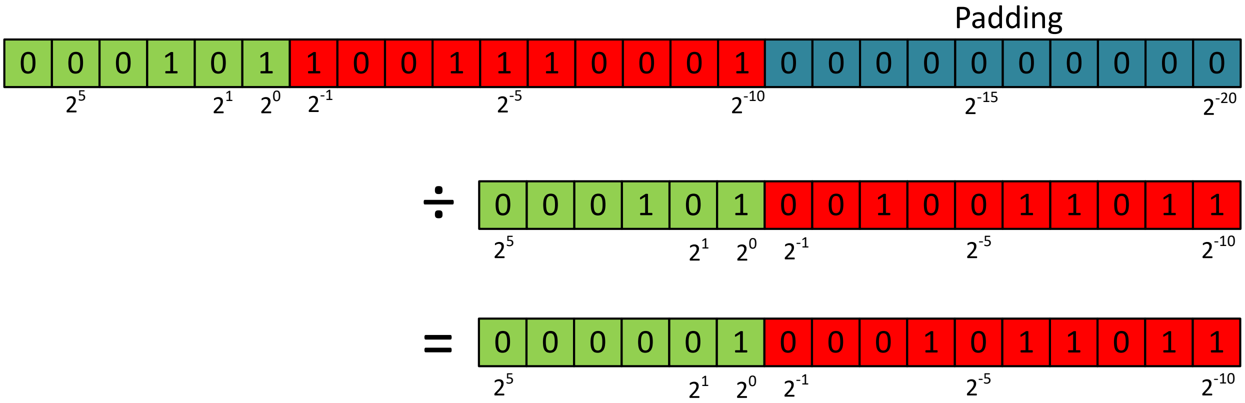


Figure 2.8.: Division of $5.6103 \div 5.1513 = 1{,}0888$ in Q6.10 format

### CORDIC algorithm

Using the above-mentioned methods, the elementary operation can be performed with fixed point numbers. But for more complex functions, the Coordinate Rotation Digital Computer

(CORDIC) algorithm is used. Using an iterative algorithm, the calculation of trigonometric function, hyperbolic functions, square roots and logarithms is possible. It is commonly used in simple microcontrollers and FPGAs when no hardware multipliers are available, due to it only requiring additions, subtractions, bit shifts and lookup tables.

There are two modes in CORDIC, rotation mode and vectoring mode. In rotation mode, the coordinate components of the vector and the desired angle of rotation are given. What has to be computed are the components of the vector, after being rotated by the desired angle. While in vectoring mode, the coordinate components are given at the start. Instead, the magnitude and angular argument of the original vector are computed.[19]

The calculation of the sine or cosine of an arbitrary angle $\theta$ will be shown in an example using the rotation mode. Figure 2.9 shows the basic principle of the rotation mode.

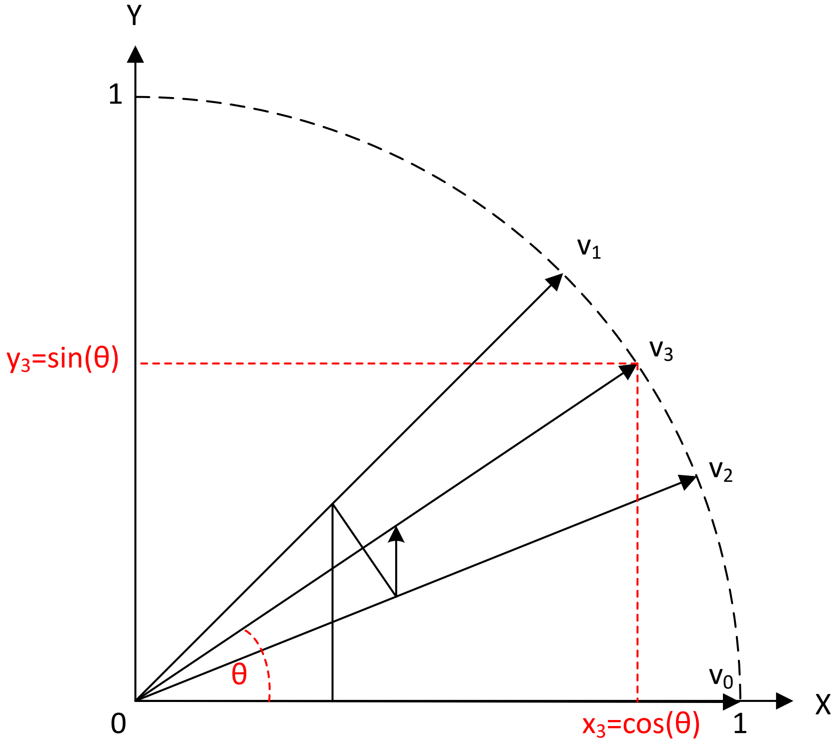


Figure 2.9.: Rotation Mode of CORDIC

The basic idea is to achieve the desired angle, by multiple, size-decreasing iterations of rotations with the size of the rotations being $\arctan 2^{-i}$ for $i = 0, 1, 2, \ldots$. For the first CORDIC iteration with $i = 1$, that would mean rotating the vector by 45° counterclockwise to get vector $v_1$.

Let $v_0$ be a vector in 2D-space with:

$$v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{2.5}$$

Using the rotation matrix $R_i$:

$$R_i = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix}, \tag{2.6}$$

the vector $v_{i+1}$ is calculated as:

$$v_{i+1} = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}. \tag{2.7}$$

Using the trigonometric identity:

$$\cos(\theta_i) = \frac{1}{\sqrt{1+\tan^2(\theta_i)}}, \tag{2.8}$$

equation 2.7 can be simplified to:

$$v_{i+1} = \frac{1}{\sqrt{1+\tan^2(\theta_i)}} \begin{bmatrix} 1 & -\tan(\theta_i) \\ \tan(\theta_i) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \tag{2.9}$$

While the sin and cos function were replaced by tan, choosing particular values for $\theta_i$ allows the calculation of equation 2.9 by using additions and bit shifts. Let the angle $\theta_i$ be chosen, so that $\tan(\theta_i) = \pm 2^{-i}$, now the multiplication with the tangent can be replaced by a simple bit shift operation in hardware. Following this, equation 2.9 is simplified to:

$$v_{i+1} = \frac{1}{\sqrt{1+2^{-2i}}} \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}, \tag{2.10}$$

with $\sigma_i$ is used to control the rotation direction. In case the angle $\theta_i$ is positive, i.e. the rotation should be counter-clockwise, then $\sigma_i$ is $+1$, otherwise -1.
Now, let $K_i$ be the factor $\frac{1}{\sqrt{1+2^{-2i}}}$. During the process of the CORDIC algorithm with $N$ iterations, $K_i$ can be at first ignored and afterwards applied as the scaling factor $K_N$:

$$K_N = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}}. \tag{2.11}$$

In case the number of iteration is fixed beforehand, the scaling factor can be calculated ahead-of-time and stored in a look-up table. In case of large values of $N$ the scaling factor tends towards $K_\infty \approx 0.6072529$.[20] For most ordinary purposes, 40 iterations are sufficient to obtain a result with 10 correct decimal places.

## 2.4. FIR Filter

In signal processing, a **finite impulse response (FIR) filter** is a filter whose impulse response is of finite duration, which means that after a finite time it will become zero. It stands in comparison to an infinite impulse response (IIR) filter, which may have internal feedback to continue to respond indefinitely. FIR filters will be used for the implementation of the filter system, which will be described in chapter 3.

The output signal $y(n)$ of a FIR filter of order $N$ is calculated as the weighted sum of the input values with the value of the impulse response:

$$\begin{aligned} y(n) &= b_0 \cdot x(n) + b_1 \cdot x(n-1) + \cdots + b_N \cdot x(n-N) \\ &= \sum_{i=0}^{N} b_i \cdot x(n-i), \end{aligned} \tag{2.12}$$

with $x(n)$ being the input signal and $b_i$ the value of the impulse response at the i'th instant. In case of the filter being in direct form, then $b_i$ is also one of the filter coefficient. The $x(n-i)$ is commonly referred as *taps*. As one can see in figure 2.10, the output signal $y(n)$ is only calculated using the input signals and the filter coefficients without any internal feedback. The top part is a $N$-stage delay line constructed out of $N+1$ taps.

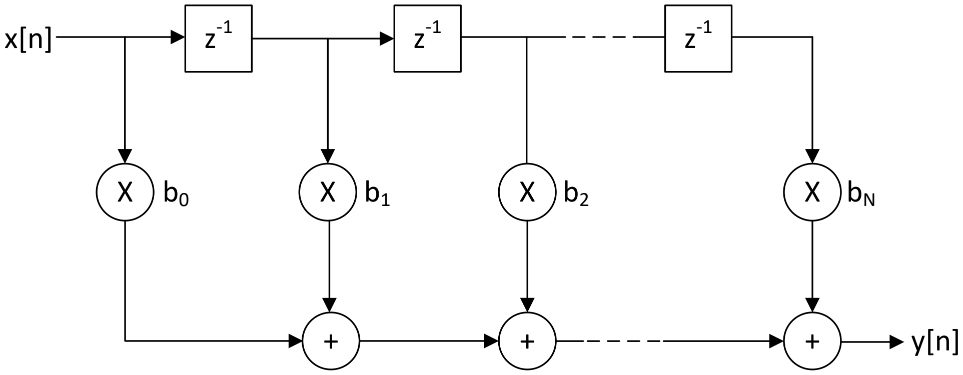


Figure 2.10.: A direct form discrete-time FIR filter of order $N$

Through careful selection of the filter coefficient $b_i$ one can realize different behaviours. An example to showcase this is the moving average filter. Given a series of numbers and a fixed window size $k$, the first output value is obtained by calculating the unweighted mean of the first $k$ consecutive data points. The next value is obtained by shifting the window by one, i.e. the first value will be excluded and the value at position $k+1$ included. The result is, that the output will change only slowly over time. It can no longer keep up with rapid changes, while still allowing slower changes. Using such coefficients, a low-pass filter can be realized. It would dampen the high frequencies while transmitting the lower frequencies unhindered.[21]

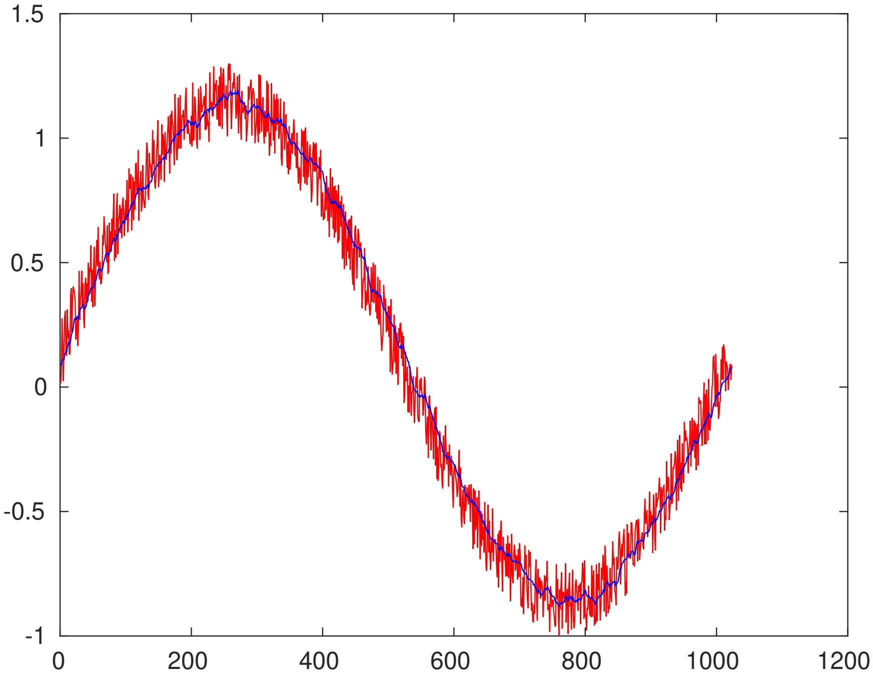


Figure 2.11.: Moving averaging filtered signal (blue) on a noisy sinus signal (red)

Using different coefficients other filter behaviours can be realized aside from a low-pass filter like a high-pass, band-pass or band-stop filter.

## 2.5. Field Programmable Gate Array

FPGAs are integrated circuits designed to be reconfigurable and reprogrammable. The flexibility and programmability of an FPGA is ideal to test and demonstrate logic devices and prototypes. Errors are easily fixed and updates can be performed after detecting errors. Therefore, FPGAs are often used in the industrial sector.

The main advantage of an FPGA over a programmable processor is its speed. Instead of processing programs sequentially, the program can run in parallel on an FPGA. While an ASIC can improve its speed even more by manually optimizing the design, the disadvantage is its long design time and low flexibility. Due to the fixed implementation, changes can only be made by creating a new design. In comparison to ASIC, the effort required for creating a design for a FPGA is lower. As such, it can also be worthwhile to use FPGAs for small quantities.

Design can be implemented on the FPGA using a hardware description language (HDL) such as VHDL or Verilog, or a schematic.

# 3. Structure of the Filter System

This chapter will describe the filter system, which will implement the computed order tracking. An overview is given first with the later sections describing the individual components in more detail.

## 3.1. Overview

The filter system performs Computed Order Tracking and consists of multiple components which form a filter chain together. It consists of a band-pass filter, an envelope detector, two decimation filters, an interpolation filter and a Fourier Transformation. An overview will be given in the following chapter.

### 3.1.1. Input Values

Through the following sections, the terms x- and y- values will be used to differentiate between the two input data, that the filter system receives. They are called as such, due to their use in the interpolation filter in section 3.5. The y-values are taken from one of the three axes from an accelerometer. The filter system of an order spectrum uses the angular displacement associated with a y-value as the x-value. In case of a frequency spectrum, the timestamps of the y-values would be used.

Most of the components of the filter system only deal with the y-values. The x-values are simply delayed and passed along to ensure that they still are aligned with their corresponding y-values.[3]

### 3.1.2. Signal Flow

A high-level overview of the filter system is shown in figure 3.1, including the configuration parameters for the different components.

At first, the y-values are passed to the band-pass filter (section 3.2) to select the desired frequency range. After the band-pass filter, an envelope detector (section 3.3) can be optionally used for certain applications. In case of rolling-element bearings, the order analysis is performed on the envelope of the signal[7], while other applications use the raw signal. Thus, a switch can be used to bypass the envelope detector.

The next step is using a decimation filter (section 3.4) to lower the signal rate for the upcoming interpolation. This is done to reduce the cost of processing the signal, which also results in a faster execution time.

The interpolation filter (section 3.5) uses the two input sequences to create "in-between" samples. It interprets the inputs as a single signal with a non-constant sampling rate and transforms it to a

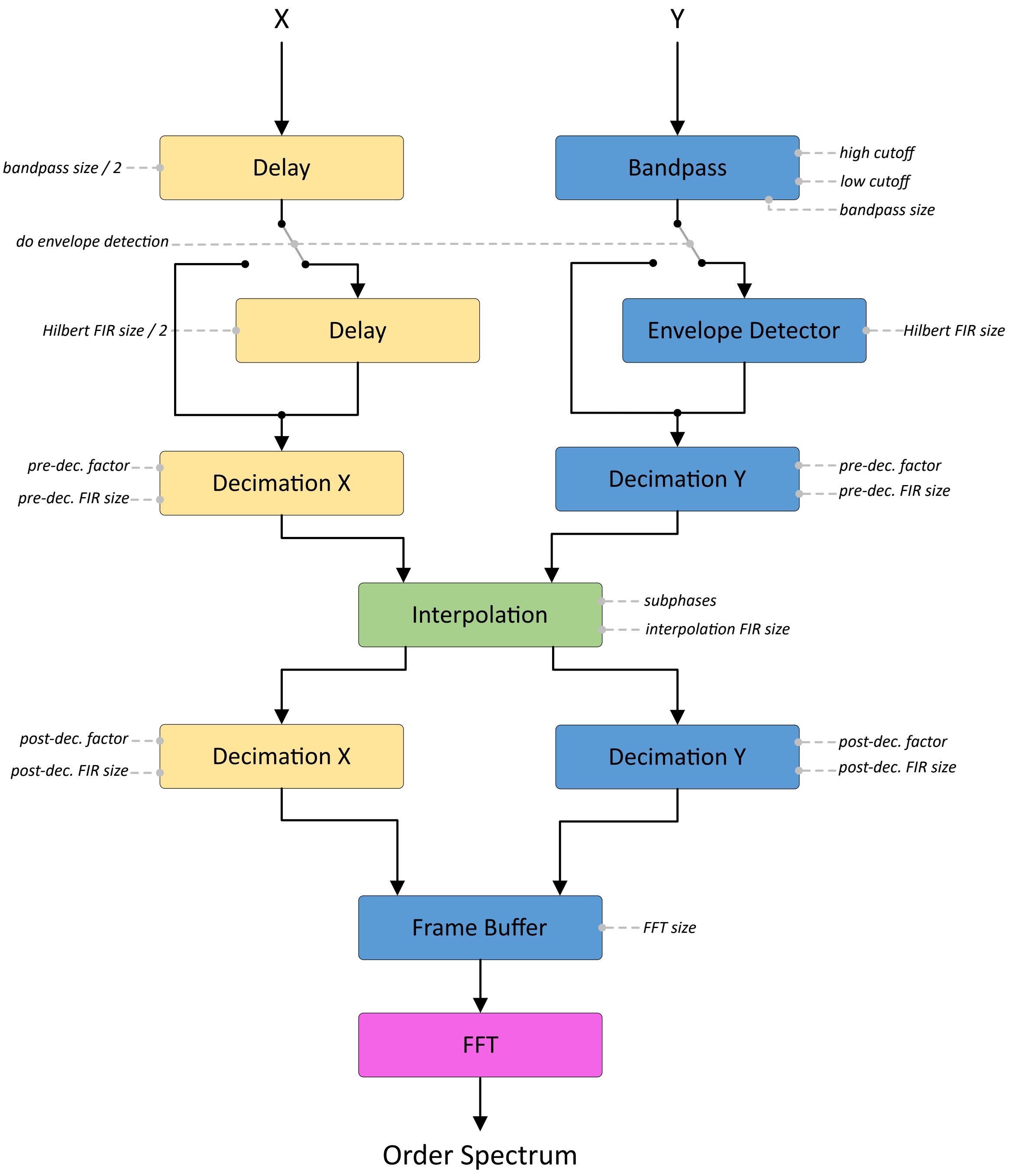


Figure 3.1.: Overview of the filter system [3]

signal with a constant angle-rate, required for the order spectrum.[3] The interpolation is followed by another decimation filter to lower the sampling rate, which was increased in the process of the interpolation.

The last two components transform the output into frequency space. The first component is a buffer. It collects the incoming data until enough samples are collected to perform an FFT and forwards it to the FFT component (section 3.6).

## 3.2. Band-pass Filter

The band-pass filter is the first component of the filter system. It is required for the Envelope Detector in section 3.3 to select the desired signal, or else the detector will simultaneously demodulate several signals. The filter is created by taking the difference of two FIR low-pass filters. The filter coefficients $h_n$ are calculated as:

$$h_n = W(n)(LP(f_{high})_n - LP(f_{low})_n) \tag{3.1}$$

with $W(n)$ being a window function of a Blackman-Harris-Window and $LP(f)$ being the coefficients of a low-pass filter with cutoff frequency $f$.

The low-pass filter is implemented using a $sinc$ function. The coefficients need to be shifted so that they can be stored into an array, where all indices are positive. Because of that, the filtered signal will be delayed by $(N_{bp}$-1$)/2$ with $N_{bp}$ being the length of the band-pass filter. To compensate the delay of the y-values, the x-values have to be delayed as well.

## 3.3. Envelope Detector

After band-pass filtering the desired signal, the envelope detection can be executed. This is required to identify faults, e.g. in rolling element bearings, where local failures produce periodically repeating series of impacts. The information of interest, in this case, is contained in the repetition frequency of the impact series and not in the frequency spectrum. The *envelope order analysis* enables easier detection of defects in these rolling element bearings.[7]

One way of creating an envelope detector is using the Hilbert Transform to create the analytic signal of the input signal. The Hilbert transform filter will be used to compute the imaginary part $\hat{s}(t)$ of the analytic signal $s_a(t)$ given its real part $s(t)$:

$$s_a(t) = s(t) + j\hat{s}(t) \tag{3.2}$$

This creates the analytic signal, a signal with no negative frequency components. All that is needed to calculate the envelope or instantaneous amplitude $s_m(t)$, is to take the absolute value of the analytic signal:

$$s_m(t) = |s_a(t)| \tag{3.3}$$

Using the idea of using the Hilbert Transform for the envelope detection, one is able to calculate the Hilbert filter coefficients. The exact reasoning and proof can be looked up in [3]. The Hilbert filter coefficients can be calculated using the following equation:

$$h_n = \begin{cases} 0, & \text{for even } n \\ \frac{2}{\pi n}, & \text{for odd } n, \end{cases} \tag{3.4}$$

which are shown in figure 3.2.

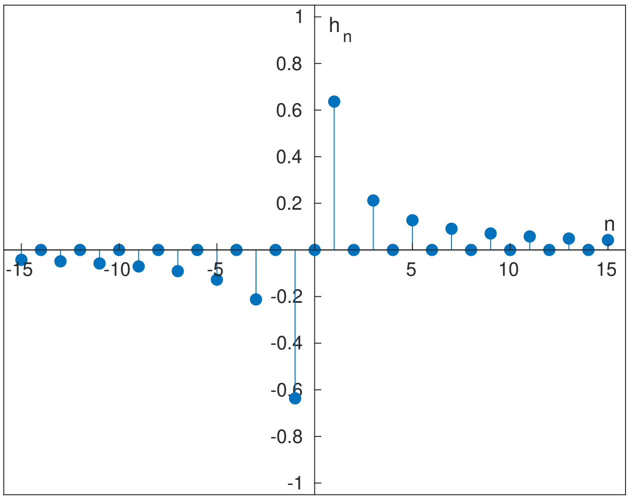


Figure 3.2.: Hilbert coefficients for the envelope detector

Similar to the band-pass filter in chapter 3.2, a Hilbert filter with length $N_H$ also has to be shifted by $(N_H - 1)/2$ to include the negative values of $n$, which means the x-values are being delayed again.

## 3.4. Decimation Filter

The decimation filter is used to downsample the incoming signal for the later components. It is used to reduce the cost of processing the signal. With the workload for the calculation and the required memory being proportional to the sampling rate, lowering the sampling rate results in a cheaper implementation. To reduce the sampling rate of a signal by an integer factor $N$, two steps have to be executed:

1. Pass the signal through a digital low-pass filter to avoid aliasing effects.
2. Decimate the filtered signal to keep only every $N^{\text{th}}$ sample.

Trying to only do a decimation on high-frequency signals may result in aliasing effects, due to violating the *Nyquist-Shannon sampling theorem*. The signal may only contain frequency components smaller than the Nyquist frequency. However, after decimating the signal, frequencies, which are now higher than the Nyquist frequency, are misinterpreted as lower frequencies. In such cases, using an anti-aliasing filter suppresses these distortions to an acceptable level.

## 3.5. Interpolation Filter

The interpolation filter is one of the more important components in the filter system, since without it the final output would not be as order spectrum. Until this point, the signal was recorded at constant time intervals (see figure 3.3). Applying an FFT on such a signal will introduce errors due to the fact that the frequency may not stay constant during the FFT frame. To prevent such errors, the frequency has to be viewed proportional to the shaft angle or rotational speed of the machine, which is the task of the interpolation filter.

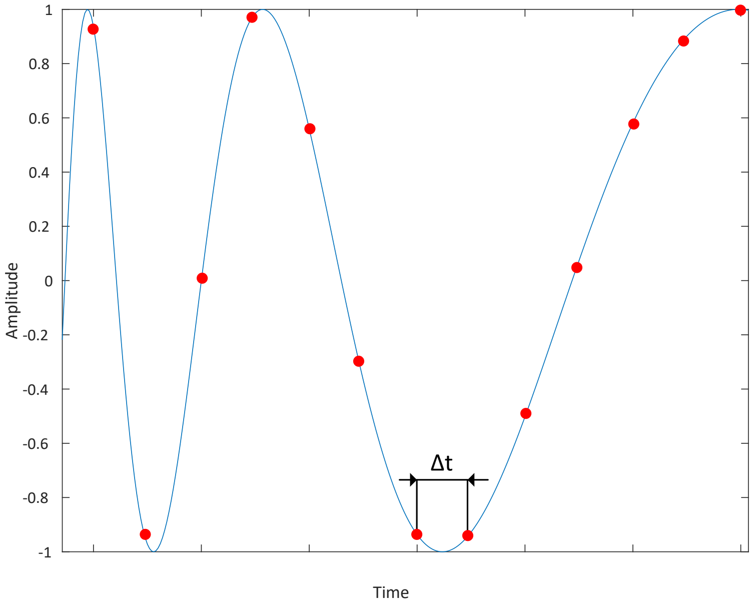


Figure 3.3.: Variable speed wave sampled at constant increments of time [4]

The position of the y-values is given by the corresponding x-values, and as such do not have a constant rate. The incoming y-values need to be re-sampled using the interpolation filter to provide the desired constant angle $\Delta\theta$ data for the order spectrum, based on the x-values, as shown in figure 3.4.[2]

To achieve this, the y-values in-between two samples have to be calculated. Since the input signal is in a discrete time-domain representation, such in-between values do not exist.

The implementation uses so-called *Fractional Delay Filters* to achieve this. The idea is, that while it is not possible to get the value of a fractional index in a discrete signal, it is possible to shift the samples by a fractional amount, so that the desired fractional index gets shifted to an integer index. More information about fractional delay filters and its algorithm used in this filter system implementation can be looked up in [3].

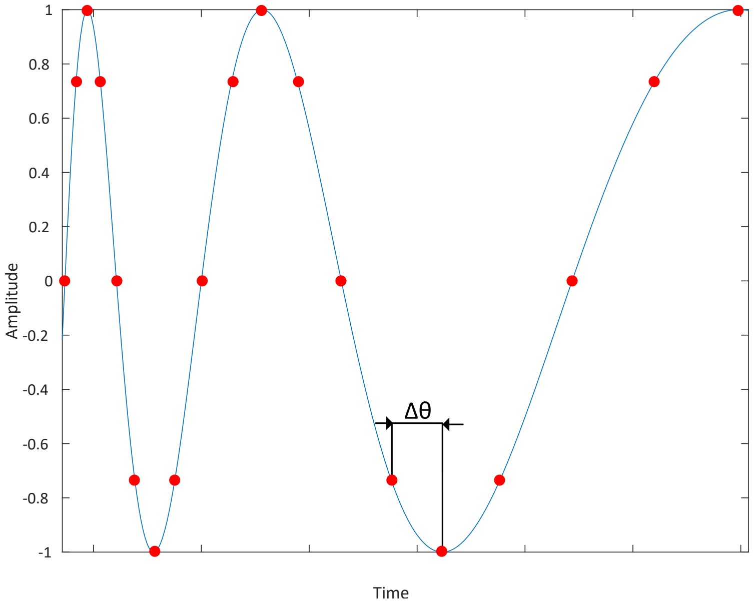


Figure 3.4.: Variable speed wave sampled at constant increments of shaft angle [4]

## 3.6. Fast Fourier Transform

The final component of the filter system for the computed order tracking is a Fast Fourier Transform (FFT) of the interpolated signal.

The most commonly used FFT algorithm is the *Cooley-Tukey algorithm*. It breaks down the discrete Fourier transform (DFT) of any composite size $N = N_1 N_2$ into many smaller DFTs of sizes $N_1$ and $N_2$. An explanation is given using a radix-2 FFT as an example. The discrete Fourier transform is defined as:

$$X(k) = \sum_{n=0}^{N-1} x_n w_N^{nk}, \tag{3.5}$$

with $k$ as an integer ranging from 0 to $N-1$ and the complex factor:

$$w_N^{nk} = e^{-\frac{2\pi i}{N} nk} \tag{3.6}$$

One can see that, for the calculation of each of the $N$ DFT-coefficients, $N$ multiplications of the complex sequence elements with the complex factors and $N-1$ additions have to be performed. The computational effort for the DFT increases quadratic with the length of the DFT.[5]

To reduce the computational effort, the radix-2 algorithm splits the DFT into even-indexed inputs $(x_{2m} = x_0, x_2, \ldots, x_{N-2})$ and odd-indexed inputs $(x_{2m+1} = x_1, x_3, \ldots, x_{N-1})$. The DFT of function $x_n$ will be rearranged into the two parts of Fourier transformations, with each being half the length of the original transformation:

$$X(k) = \sum_{m=0}^{M-1} x_{2m} w_N^{(2m)k} + \sum_{m=0}^{M-1} x_{2m+1} w_N^{(2m+1)k} \tag{3.7}$$

with $M = N/2$. Here, $N$ is assumed to be a power of two, but that is not an important restriction, since $N$ can be chosen freely when using methods like zero-padding. From the second sum, the so-called **twiddle factor** $w_N^k$ can be factored out:

$$X(k) = \sum_{m=0}^{M-1} x_{2m} w_N^{mk} + w_N^k \sum_{m=0}^{M-1} x_{2m+1} w_N^{mk} \tag{3.8}$$

The same steps can now be applied recursively to the two new transforms. After one step we receive four Fourier transform with each being a quarter of the original length. This is repeated until each sum has the length of two, which will be calculated directly then. The following example shows the method for a DFT length $N = 8$. Using the substitutions $u_m = x_{2m}$ and $v_m = x_{2m+1}$ we have:

$$X(k) = \underbrace{\sum_{m=0}^{M-1} u_m w_N^{mk}}_{\mathrm{U}(k)} + w_N^k \underbrace{\sum_{m=0}^{M-1} v_m w_N^{mk}}_{\mathrm{V}(K)} \tag{3.9}$$

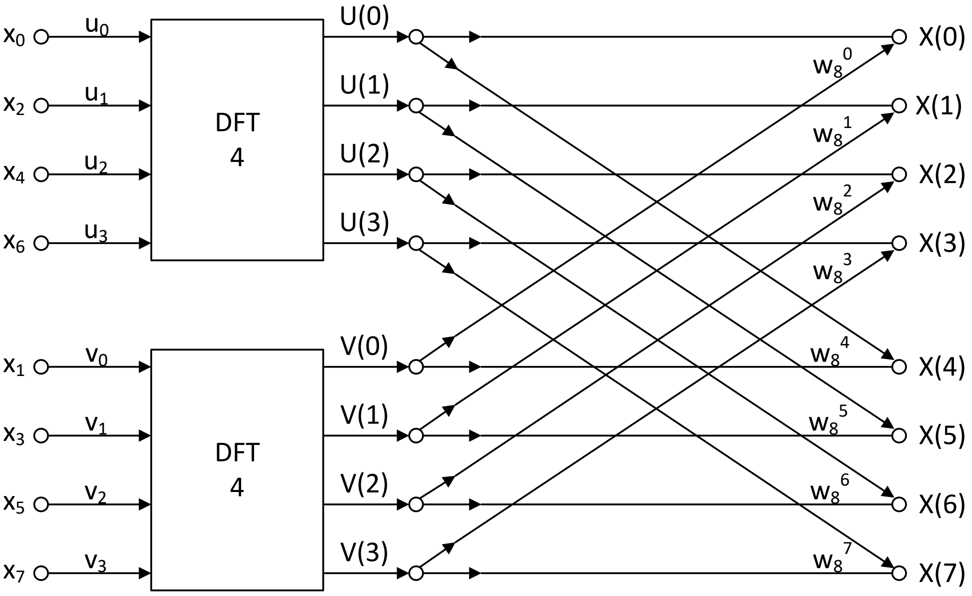


Figure 3.5.: Signal-flow graph for a radix-2 FFT with length $N = 8$ after the first decomposition in $U(k)$ and $V(K)$

Figure 3.5 shows the signal-flow graph after the first split. The weight is 1 if it is not explicitly stated.

This process is called decimation-in-time decomposition, due to the samples being rearranged in alternating groups.[5] Because of the periodicity with $M$ frequency samples of these length-$M = N/2 = 4$ DFTs, $U(k)$ and $H(K)$ can be used to compute two of the length-$N$ DFT frequencies, namely $X(k)$ and $X(k+M)$, but with a different twiddle factor.

The signal-flow graph after two further decomposition is shown in figure 3.6. It is shown that for each decomposition step $N$ complex multiplications and additions have to be performed. In case of the length of the DFT being a power of 2, then there are exactly $\log_2 N$ decomposition. The computational effort is reduced to $O(n \log n)$.

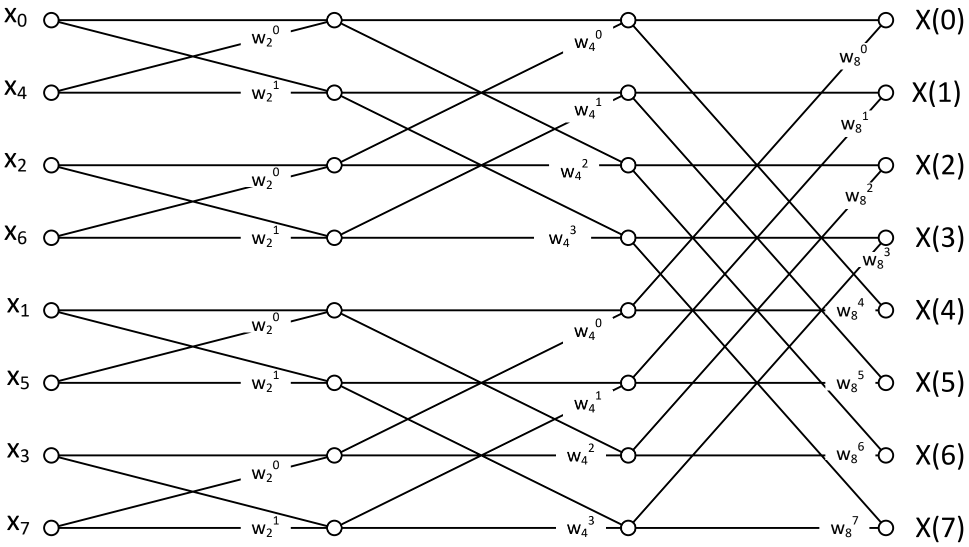


Figure 3.6.: Temporary signal-flow graph for a radix-2 FFT with length $N = 8$ after the three decomposition

In further analysis of the diagram, the complexity can be further reduced. For example, from the inputs $x_0$ and $x_4$ exactly two values will be calculated. After the calculation, the two values won't be required anymore and can be overwritten by their results.

Further optimization can be done using the equations:

$$w_N^{N/2} = -1 \tag{3.10}$$

$$w_N^{N/2+k} = -w_N^k \tag{3.11}$$

Using equation 3.10 reduces the two complex multiplication for each of decomposition step by half. For example, for the four butterfly (see. figure 3.7) radix-2-FFTs in figure 3.6 each weight $w_2^1 = -1$.

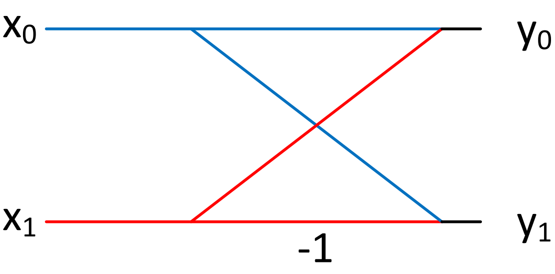


Figure 3.7.: Butterfly diagram of a radix-2 FFT

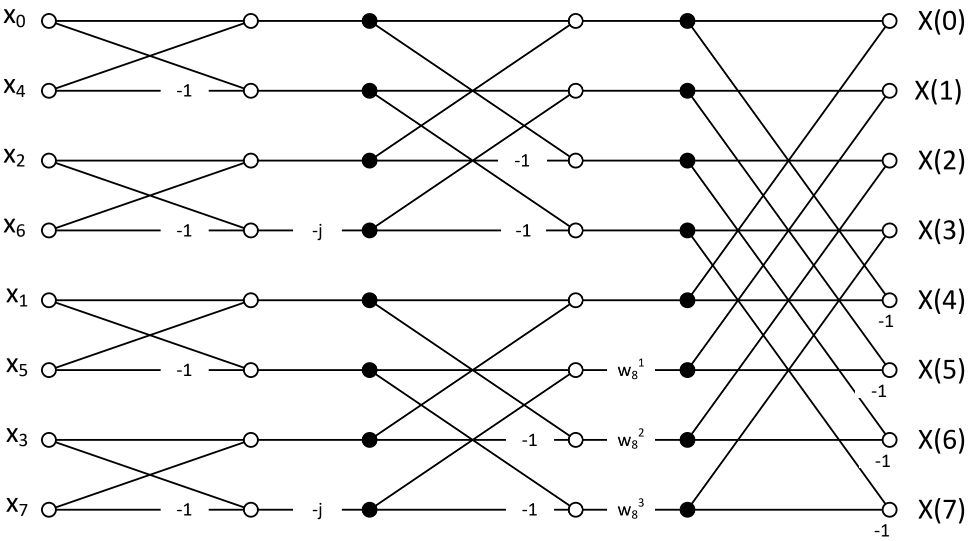


Figure 3.8.: Signal-flow graph for a radix-2 FFT with length $N = 8$ [5, Bild 4-4]

It is clear, that the first two of the three steps of figure 3.8 won't require any complex multiplications, which can further reduce the complexity of the algorithm. When implementing such an FFT one has to pay attention to the values, which have to be reordered. While the input values have to be reordered in the decimation-in-time decomposition, there is also the variant of the decimation-in-frequency where the order of the input values is kept, but instead the order of the output is changed.

For a DFT with length $N = 8$ the order of the input values can be seen in figure 3.8. The order can be obtained for higher radix values, by reversing the bits of the indices. For a radix value r the reordering of the input values is performed by writing out each index in base r and reversing the bits. In case r is a power of two, this can be represented by using the binary representation and looking at the groups of bits. For example, the index $1_{10} = 0001_2$ is reversed to $1000_2 = 8_{10}$. Table 3.1 shows the reordering for a radix-2-FFT.[5]

Table 3.1.: Bit reversal for the decimation-in-time radix-2-FFT

| Index | Binary | Bit reversed binary | Reordered index |
|---|---|---|---|
| 0 | 0000 | 0000 | 0 |
| 1 | 0001 | 1000 | 8 |
| 2 | 0010 | 0100 | 4 |
| 3 | 0011 | 1100 | 12 |
| 4 | 0100 | 0010 | 2 |
| 5 | 0101 | 1010 | 10 |
| 6 | 0110 | 0110 | 6 |
| 7 | 0111 | 1110 | 14 |
| 8 | 1000 | 0001 | 1 |
| 9 | 1001 | 1001 | 9 |
| 10 | 1010 | 0101 | 5 |
| 11 | 1011 | 1101 | 13 |
| 12 | 1100 | 0011 | 3 |
| 13 | 1101 | 1011 | 11 |
| 14 | 1110 | 0111 | 7 |
| 15 | 1111 | 1111 | 15 |

## 3.7. Filter System Parameters

As one can see in figure 3.1, each individual component of the filter system can be configured using some parameters. These parameters are either set by the user or calculated by these set values.

### Band-pass filter parameters

The band-pass filter has three configurable parameters. The lower and upper cutoff frequencies $f_{low}$ and $f_{high}$ are directly provided by the user. The FIR filter length of the band-pass filter $N_{bp}$ is also set to a constant value at compile time.

### Envelope detector parameters

The envelope detector has only the Hilbert FIR filter length as a configurable parameter. The parameter has to be set according to the incoming frequencies of the envelope detector. In this implementation, a Hilbert filter length $N_H$ of twice the period of the lowest frequency component is used. Since the only component before the envelope detector is the band-pass filter, the lower cutoff frequency of the band-pass filter is chosen as the lowest frequency component. As such, the Hilbert FIR filter length is calculated as:

$$N_H = 2T_{low} = 2\frac{f_s}{f_{low}}, \tag{3.12}$$

with $f_s$ being the sampling rate of the input signal. The functionality of the decimation filter with twice the lower cutoff period of the band-pass filter has been confirmed through testing in [3].

### Pre-decimation filter parameters

The first decimation filter is controlled by the filter length $N_{pre}$ and the decimation factor $q_{pre}$. The decimation factor in turn is determined using the maximum tracking order $O_{max}$ of the filter system, which is set by the user. Using the maximum order the minimum required sampling rate for the decimation filter can be calculated.

An order of $k$ means having $k$ times the frequency of the reference rotational frequency, which is in this case the input signal. The highest theoretical frequency that can occur is therefore the maximum order, times the maximum rotational speed $R_{max}$, which are both given by the user. Due to the Nyquist–Shannon sampling theorem, the required sampling rate to sample this frequency is calculated as:

$$f_{pre} = 2 \cdot O_{max} \cdot R_{max} \tag{3.13}$$

Using this frequency, the decimation factor can be calculated as:

$$q_{pre} = \left\lfloor \frac{f_s}{f_{pre}} \right\rfloor = \left\lfloor \frac{f_s}{2 \cdot O_{max} \cdot R_{max}} \right\rfloor \tag{3.14}$$

The filter length $N_{pre}$ is then calculated from the decimation factor as:

$$N_{pre} = 64 \cdot q_{pre} - 1 \tag{3.15}$$

The filter length is calculated from the decimation factor to let the filter have a constant overhead. With a higher decimation factor, more samples are skipped, and the filter length increases proportionally. To avoid a fractional delay, which would occur with an even filter length, the filter length is made odd by shortening it by 1.[3]

### Interpolation filter parameters

The interpolation filter is configured by the number of subphases and the interpolation FIR length, which both are provided by the user, and the resolution of the output grid of the interpolation filter $f_{int}$. $f_{int}$ itself is calculated by the oversampling factor $q_{int}$. At first, the oversampling factor is calculated by determining the highest frequency used in the filter system until now for the y-values, which is either the higher cutoff frequency of the band-pass filter and the first decimation filter or the sample rate of the input signal $f_{sample}$. Due to the sampling theorem, the sampling rate required is calculated as:

$$f_{max} = \min\{f_{pre}, 2f_{high}, f_{sample}\} \tag{3.16}$$

Here, only $f_{high}$ has to be multiplied with 2, because the $f_{pre}$ and $f_{sample}$ are already sampling rates of the highest frequencies of their components. From hereon, the oversampling factor $q_{int}$ can be calculated as:

$$q_{int} = \left\lceil \frac{f_{max}}{f_{min}} \right\rceil \tag{3.17}$$

with $f_{min}$ being the lowest output rate. Because the output rate at any point is calculated as the current rotational speed times the maximum order, the lowest output rate occurs when the slowest rotational speed $R_{min}$ is happening:

$$f_{min} = 2 \cdot O_{max} \cdot R_{min} \tag{3.18}$$

Finally, the resolution of its output grid can be calculated with:

$$f_{int} = 2 \cdot O_{max} \cdot q_{int} \tag{3.19}$$

### Post-decimation filter parameters

Like the first decimation filter, the second decimation filter is also controlled by its filter length $N_{post}$ and the decimation factor $q_{post}$. The second decimation filter is used due to the interpolation oversampling the signal. Due to this, the decimation factor is equal to the oversampling factor from the interpolation filter:

$$q_{post} = q_{int} \tag{3.20}$$

Similarly to equation 3.15, the decimation filter length is calculated as:

$$N_{post} = 64 \cdot q_{post} - 1 \tag{3.21}$$

### FFT parameters

The final component of the filter system is the FFT, with its only configurable parameter being the frame length $N_{FFT}$. The value is calculated from the maximum order and the number of rotations per frame $R$, which are both set by the user:

$$N_{FFT} = 2 \cdot O_{max} \cdot R \tag{3.22}$$

Something to note is, that the frame length $N_{FFT}$ has to be a power of 2.

# 4. Implementation

This chapter will present the course of actions of the implementation and optimization of the Complex Filter System for a TTA-based FPGA-Demonstrator.

## 4.1. Overview

The initial reference implementation was written in C++ and designed to run on an ARM® Cortex® M4-based microcontroller. This work uses the modified implementation of the Complex Filter System-Code, which was used in the Master's Thesis **Evaluation and Optimization of a Complex Filter System Using Different Application-Specific Instruction-Set Processor Configurations** by Jonas Rinke, a port designed to run on Xtensa LX7-based architectures and written in C11 using Cadence® Tensilica® Xtensa® Xplorer™ IDE (Xtensa IDE).[3]

A difference from the filter system of chapter 3, introduced in the modified implementation in [3], is skipping the first decimation filter in case of a decimation factor of 1. The decimation filter is used to downsample the incoming signal by a decimation factor $N$. The implementation skips the decimation filter in those cases, due to every sample of the input signal being passed on. It would reduce the decimation filter to a low-pass filter with a cutoff frequency slightly below the Nyquist-limit and does not contribute to the overall output of the system. As such, the decimation filter will be skipped in those cases.

To increase the performance for the TTA, the code was optimized by making use of instruction parallelism. Other changes include a hardware based fixed point multiplier, instead of a software-based implementation. Through loop unrolling and term splitting into temporary variables, the implementation of the FIR filter can also be simultaneously performed using multiple custom function units. Finally, instead of reading the input data through a file, a streaming FU was used to receive the data.

Figure 4.1 shows the workflow of the implementation for a TTA processor. Using the filter system implementation in C++ and the different configurations, which will be introduced later in this chapter, a processor simulation can be performed. The programmer receives feedback from the simulation in the form of the total cycle count and resource utilization and he can further optimize the processor. Finally, a processor can be generated along with its instruction and data memory image. Using Xilinx Vivado[22] the TTA processor can be synthesized on an FPGA, which will return the power consumption, resource utilization, and the timing report of the individual configuration.

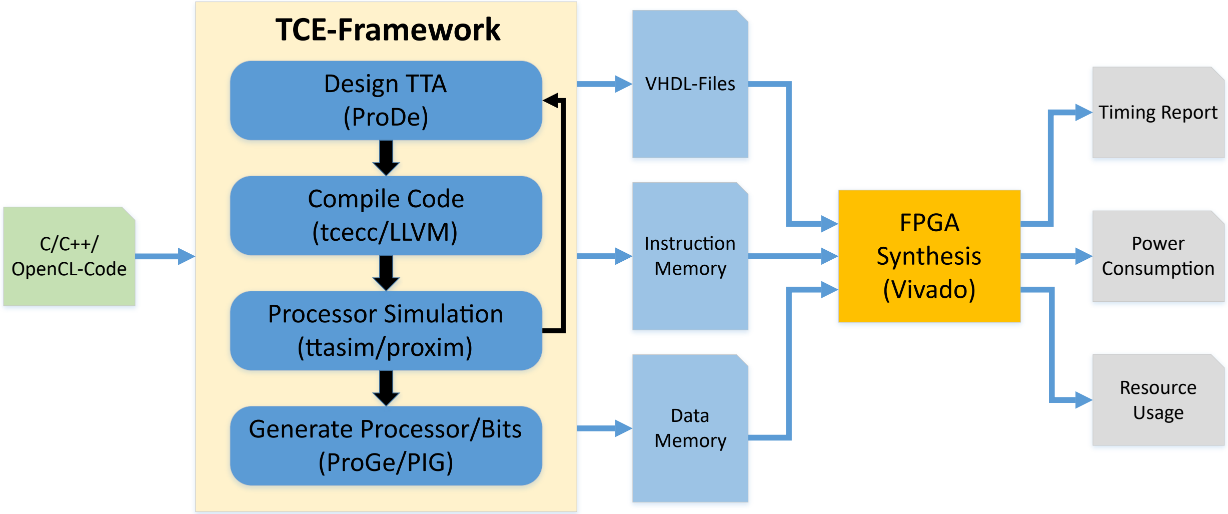


Figure 4.1.: Workflow for implementing the TTA processor

## 4.2. Hardware Optimization

The requirements for the processor architecture offer many approaches for optimization. Especially, when using fixed implementations of software in embedded systems, the required functions can be accelerated. In the case of TTA, existing or custom function units can be used according to the functions used.

### 4.2.1. Fixed Point Multiplier

In many FPGAs there are DSP (digital signal processor) blocks to perform multiplications quickly. Similarly, TTA offers multiplication function units, which are capable of doing 32-bit and 64-bit multiplications. Unfortunately, these function units are not suitable for this implementation. In the case of the 32-bit multiplier, the function unit only returns the lower 32 bits of the result. However, this application uses a fixed point implementation for the filter system. Depending on the fixed point format, specific 32 bits of the 64-bit result are desired to be returned.

The 64-bit multiplier also returns the lower 64 bits of the product. While this function unit could be used for this application after selecting the desired 32 bits, the function unit implementation would be rather big. Furthermore, the data width would also be required to be extended from 32 to 64 bits. Due to this, a custom fixed point multiplier function unit was implemented.

Speed and area requirements are important properties of hardware structures. The fixed point multiplier function unit was implemented using an array multiplier. An array multiplier has the advantage of reusing components due to calculating the same operations multiple times. Because of that, it is possible to reduce the area required for the hardware multiplication and to make an exchange between area, power consumption and performance.[23]

The multiplier can be reconfigured to change the latency of the operation. The lower the latency of the multiplier is, the bigger the area will be for the array multiplier.

Through pre- and post-complement the signed multiplication will be calculated. Let multiplicand $A$ and multiplier $B$ be $n$-bit two's complement. In case either of the numbers is negative, then either $a_{n-1}$ or $b_{n-1}$ would be 1. In that case, the absolute value would be calculated by complementing bitwise and adding 1 to the result. The product of the absolute values $|A| \cdot |B|$ will receive the sign:

$$a_{n-1} \oplus b_{n-1} \tag{4.1}$$

In case exactly one of them is negative, then the result will also become negative. The two's complement of the result has to be calculated, in that case, and the value will be negated. The layout of the multiplier is shown in figure 4.2.

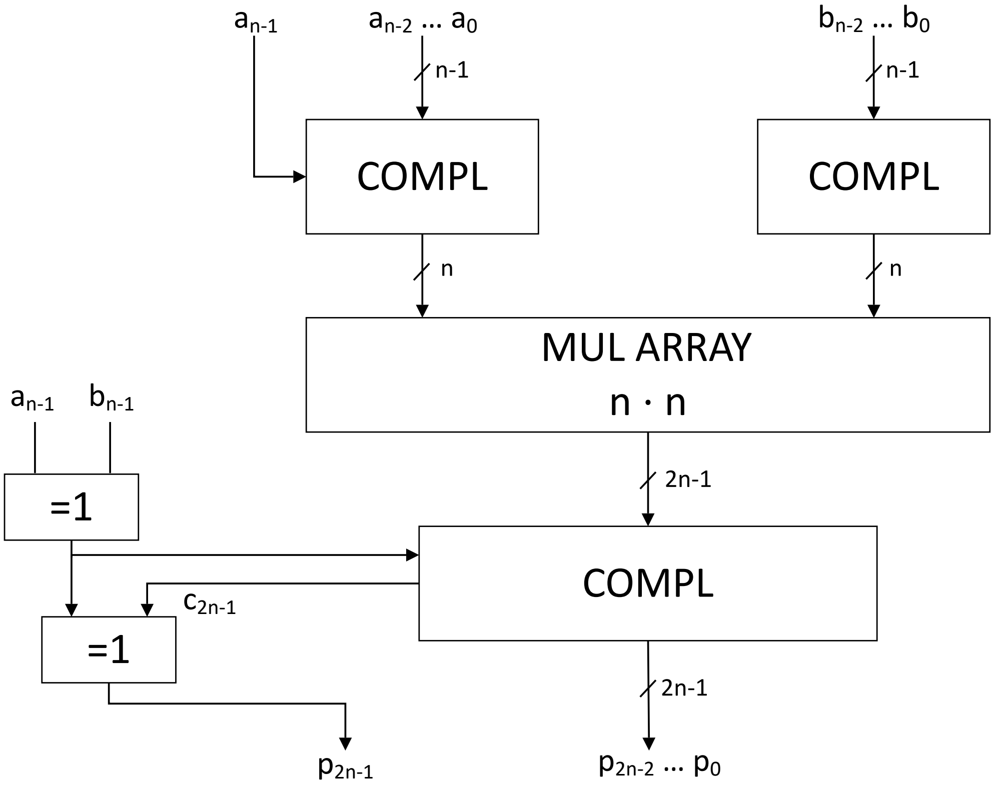


Figure 4.2.: Multiplier with pre- and post-complement [6, Bild 3.3.11]

The parallel multiplication uses many additions. In the case of only one array element with a full adder, the multiplication of two 32-bit numbers would require $32 \times 32 = 1024$ cycles. When using a 32-bit adder, the cycle count would be reduced to 32 cycles required to multiply the array with each of the 32 bits of the other operand. The least amount of cycles is 1 cycle for the whole multiplication. This is possible when using a $32 \times 32$ array component for the multiplication of two 32-bit numbers.

In figure 4.3 a 4-bit ripple carry array multiplier can be seen, which can be used for the multiplier component in figure 4.2. In some cases the full adder can be switched out with either a half adder cell, when either s'=0 or c'=0, or can be skipped completely if s'=c'=0.[6]

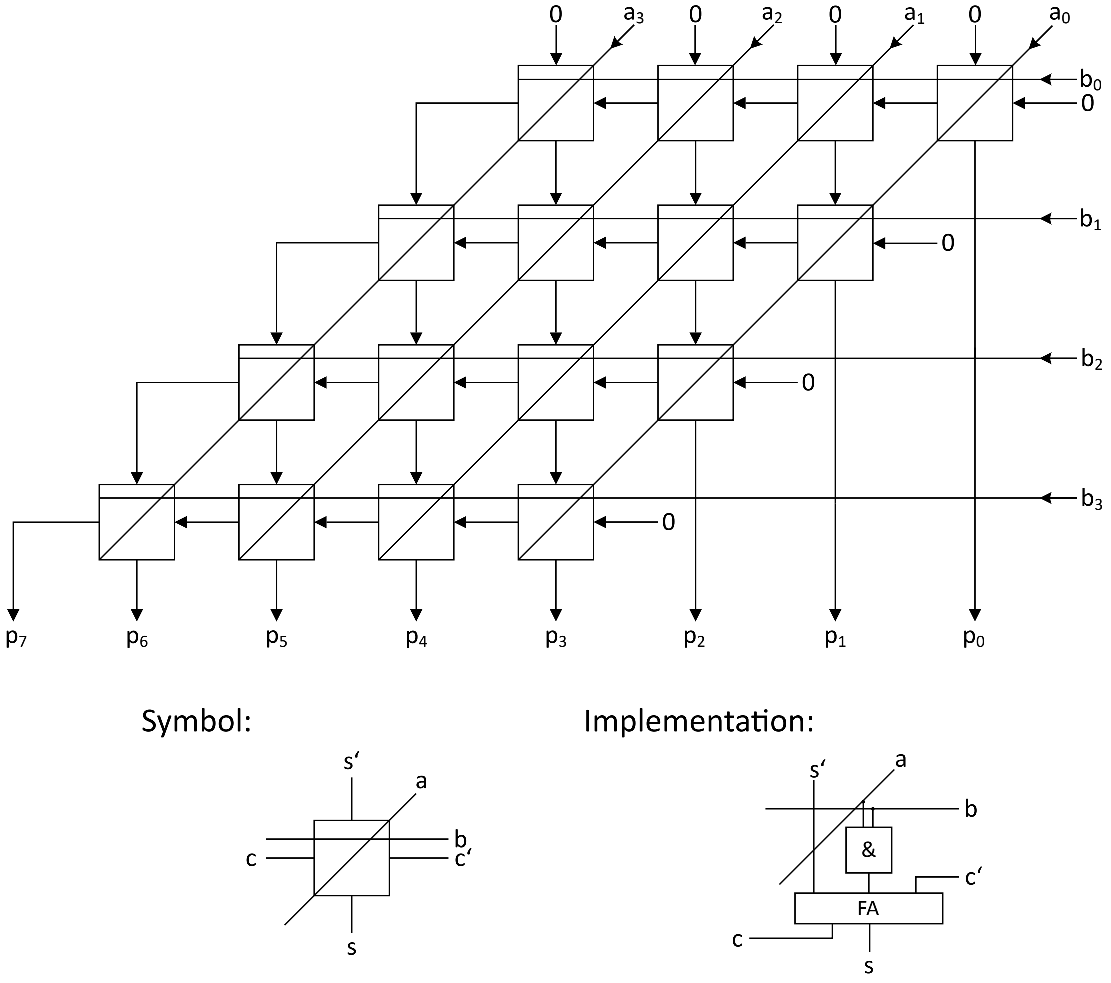


Figure 4.3.: Ripple Carry Array Multiplier [6, Bild 3.3.2]

The critical path of the multiplier begins with the cell with the inputs of $a_0$ and $b_0$ and ends at $p_7$. In the case that n=m, the delay is calculated using equation 4.2.[6]

$$T_{D,MUL} = (62,4n - 72,4)\tau_L \tag{4.2}$$

One can see, that the time for the operation is linearly dependent on the amount of bits $n$. The projection onto n rows provides scaling of the multiplier. The time required for the multiplication can be scaled between 1 and the maximum number of rows $n$. Through multiple use, the required number of multiplier rows and the required area can be reduced at the expense of the latency.[23]

This multiplier calculates all 64 bits of the $32 \times 32$ multiplication. Finally, the 32 bits of interest, which depend on the fixed point format, have to be selected for the result.

### 4.2.2. TTA Custom Function Units

After creating the implementation of a fixed point multiplier in VHDL and the verification of the functionality using the simulator Mentor Graphics ModelSim SE-64, custom function units can be added to the TTA processor design.

An example of creating and using a custom function unit will be shown using the FPMUL/FPMAC function unit, which enables the operations for a fixed point multiplication and a fixed point multiply-accumulate operation using the same hardware, which both use the fixed point multiplier implementation of chapter 4.2.1.

### Simulation behaviour definition

To use a custom function unit, the basic properties of the operations and a definition of its simulator behaviours have to be included to the operation database with **Operation Set Editor (OSEd)**:

**Algorithm 1** Simulation behaviour for the fixed point multiplication in OSEd

```
OPERATION(FPMUL)
TRIGGER
    int64_t in1 = static_cast<int64_t>(INT(1));
    int64_t in2 = static_cast<int64_t>(INT(2));
    int64_t c = in1 × in2;
    IO(4) = static_cast<SIntWord>(c≫(32 - INT(3)));
END_TRIGGER
END_OPERATION(FPMUL)
```

**Algorithm 2** Simulation behaviour for the fixed point multiply–accumulate in OSEd

```
OPERATION(FPMAC)
TRIGGER
    int64_t in1 = static_cast<int64_t>(INT(1));
    int64_t in2 = static_cast<int64_t>(INT(2));
    int64_t in3 = static_cast<int64_t>(INT(3));
    int64_t c = in1 × in2;
    IO(5) = static_cast<SIntWord>(c≫(32 - INT(4)) + in3);
END_TRIGGER
END_OPERATION(FPMAC)
```

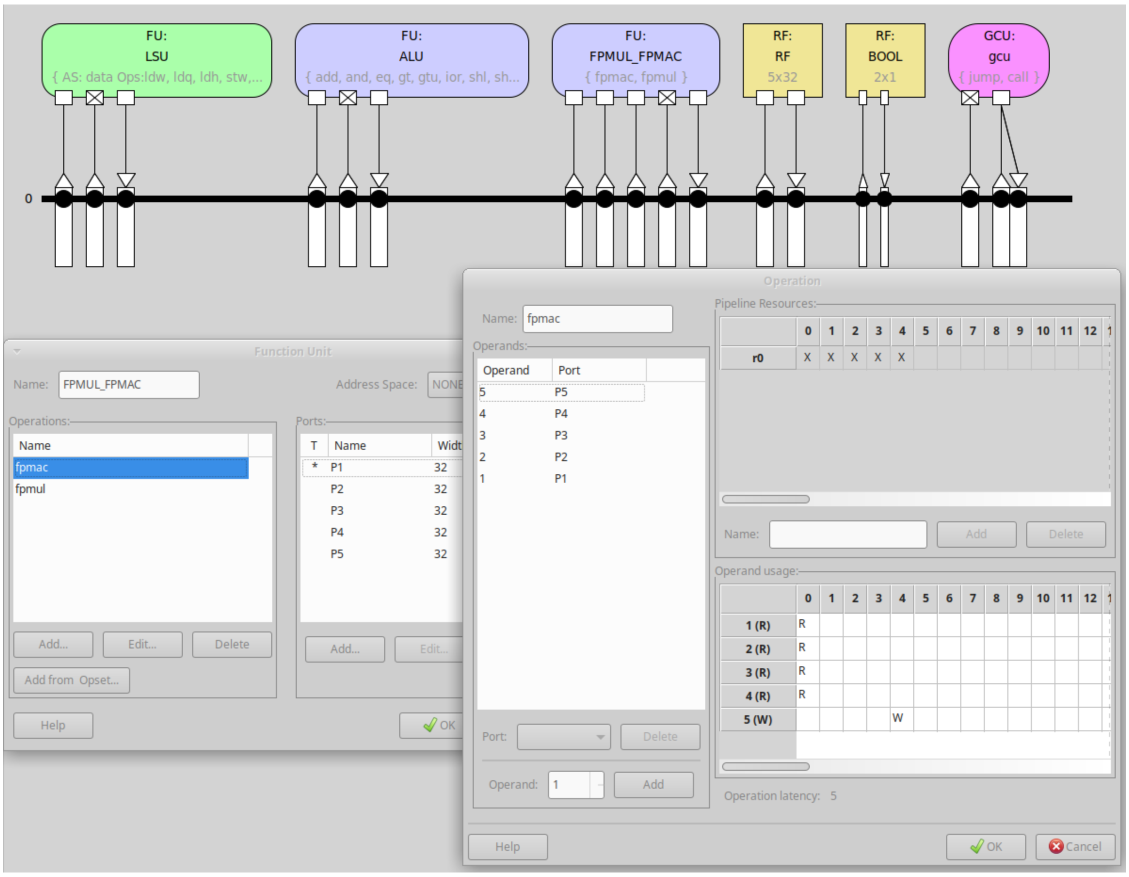
Algorithm 1 shows the simulation behaviour for the FPMUL operation. The operation casts the 32-bit inputs to 64-bit integers so that the result has enough bits to avert an overflow of the product. After the multiplication, the product will be shifted to the right, depending on the amount of fractional bits (see section 2.3.1). The result now has the original fixed point format and can be written back after being cast back to 32 bits.

Similarly algorithm 2 shows the behaviour for the FPMAC operation. The difference is, that after being shifted the result will be added with the remaining input.

### Addition of the custom function unit to the architecture

After the operation definition of the custom operation has been added to the Operation Set Abstraction Layer (OSAL) database using OSEd, the architecture needs at least one function unit which implements the FPMUL and FPMAC operations, so that it can be used in the processor design.

This can be accomplished using ProDe mentioned in section 2.2.4:



Function Unit

Name: FPMUL_FPMAC

Address Space: NONE

Operations:

| Name |
|---|
| fpmac |
| fpmul |

Ports:

| T | Name | Widt |
|---|---|---|
| * | P1 | 32 |
| | P2 | 32 |
| | P3 | 32 |
| | P4 | 32 |
| | P5 | 32 |

Operation

Name: fpmac

Operands:

| Operand | Port |
|---|---|
| 5 | P5 |
| 4 | P4 |
| 3 | P3 |
| 2 | P2 |
| 1 | P1 |

Port:

Operand: 1

Pipeline Resources:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| r0 | X | X | X | X | X | | | | | | | | |

Name:

Operand usage:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 (R) | R | | | | | | | | | | | | |
| 2 (R) | R | | | | | | | | | | | | |
| 3 (R) | R | | | | | | | | | | | | |
| 4 (R) | R | | | | | | | | | | | | |
| 5 (W) | | | | | W | | | | | | | | |

Operation latency: 5

Figure 4.4.: Adding the operations FPMUL and FPMAC to a new function unit in ProDe

As seen in algorithm 2 the FU requires 4 input ports and 1 input port. When executing the FPMUL operation, the port for the additional addition is unused.

Something to note is that the programmer has to configure the latency of the function unit himself. The simulator of the TTA doesn't care about the plausibility of the latency and would simulate the program with a stated latency of 1, even if the actual latency is higher. As long as the VHDL implementation of the operation is incapable of working on such a latency, the program would never run on an FPGA. As such, the latencies for the FUs using different multiplier configurations were calculated and tested using ModelSim. They can be calculated as followed:

$$\text{latency}_{\text{k-stage multiplier FU}} = \frac{32}{k} + 1 \tag{4.3}$$

In the case of a 8-stage multiplier implementation, the multiplication itself would require $32/8 = 4$ cycles. Another cycle is added on top of that for the FU to read the inputs and write the results.

Because both operations use the same hardware resources, the user also has to enter the pipeline resource usage as shown in figure 4.4, because it is not possible to execute the operations FPMUL and FPMAC on the same function unit at the same time.

### TCE custom operation intrinsic

The simulation behaviour was defined and a function unit capable of using the custom operations was added to the design. The last thing to do is to use the custom operation intrinsic to let the compiler know, when the added custom hardware will be used. Usage of the TCE custom operation intrinsic for the fixed point multiply-accumulate is as follows:

```
_TCE_FPMAC(m1, m2, a, s, r),
```

with m1 and m2 being the multiplicands of the multiplier, a the addend, s the shift value, which would usually be the number of fractional bits, and r the result.

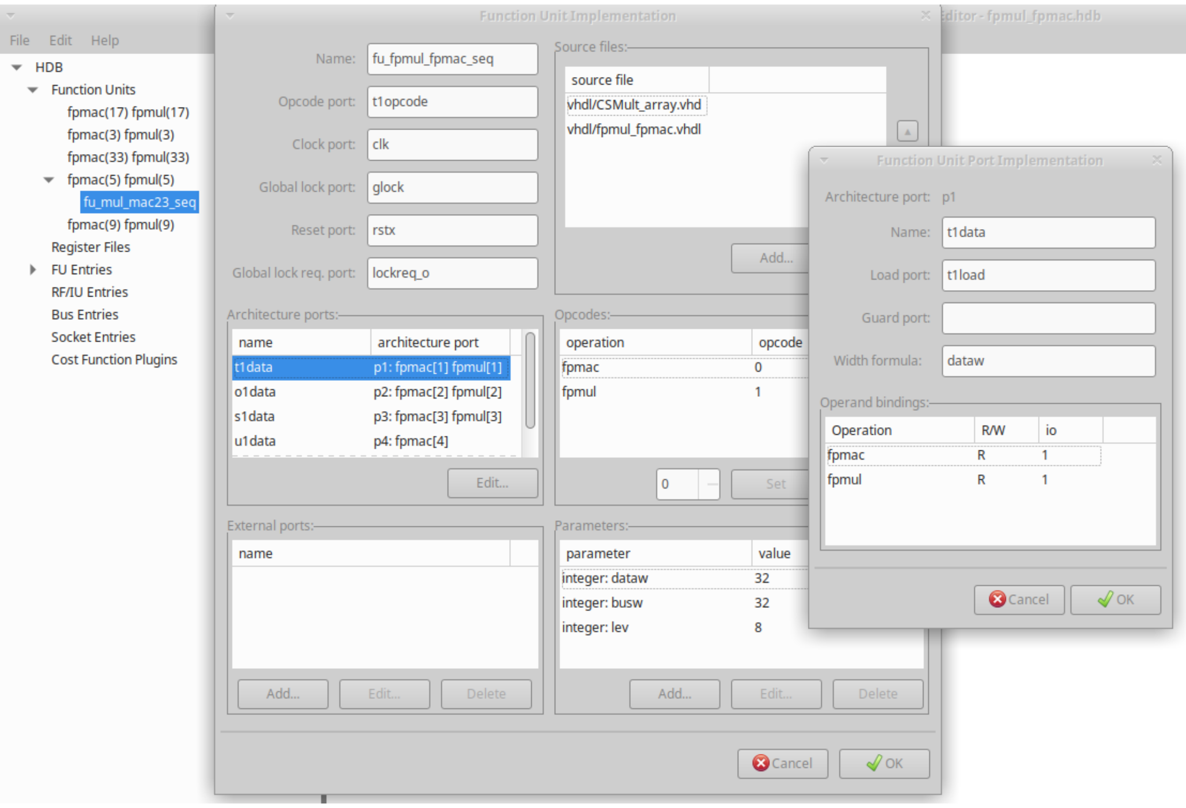
Similarly the intrinsic for the fixed point multiplier operation is:

```
_TCE_FPMUL(m1, m2, s, r),
```

except that there is not an additional input for the addition.

### Adding HDL implementation of the FU to the hardware database (HDB)

Subsequently, the HDL implementation of the function unit has to be added to the hardware database using the HDB Editor, so that the VHDL files of the processor architecture can be generated.



Name: fu_fpmul_fpmac_seq
Opcode port: t1opcode
Clock port: clk
Global lock port: glock
Reset port: rstx
Global lock req. port: lockreq_o

Source files:

| source file |
|---|
| vhdl/CSMult_array.vhd |
| vhdl/fpmul_fpmac.vhdl |

Architecture ports:

| name | architecture port |
|---|---|
| t1data | p1: fpmac[1] fpmul[1] |
| o1data | p2: fpmac[2] fpmul[2] |
| s1data | p3: fpmac[3] fpmul[3] |
| u1data | p4: fpmac[4] |

Opcodes:

| operation | opcode |
|---|---|
| fpmac | 0 |
| fpmul | 1 |

External ports:

| name |
|---|

Parameters:

| parameter | value |
|---|---|
| integer: dataw | 32 |
| integer: busw | 32 |
| integer: lev | 8 |

Architecture port: p1
Name: t1data
Load port: t1load
Guard port:
Width formula: dataw

Operand bindings:

| Operation | R/W | io |
|---|---|---|
| fpmac | R | 1 |
| fpmul | R | 1 |

Figure 4.5.: Adding the HDL implementation for the FPMUL/FPMAC FU to the hardware database

As mentioned in 4.2.1 the multiplier is configurable, with each configuration having a different latency for the operation. For each configuration an HDL implementation has to be added to the hardware database, as seen in figure 4.5.

To enable a single FU to execute the two different operations FPMUL and FPMAC the *opcode* (operation code) is used to differentiate between the two in the HDL implementation. The opcode port is usually bound to the trigger port and is in this implementation called t1opcode. In case of the operation FPMAC the opcode 0 is selected and for operation FPMUL the opcode is 1.

This example showed the creation and usage of the FPMUL/FPMAC function unit. Furthermore, because the fixed point format doesn't change while running the program, the performance of the FU can be further enhanced.

Each time the FU executes an operation, the same shift value will be loaded to the shift input port. This requires an additional move instruction, which can be omitted by dropping the shift input port and hard-coding the shift value into the VHDL files. The FPMUL/FPMAC FUs used in the evaluation in chapter 5 are optimized for the chosen fixed point format and have only 4 ports.

In this way, custom function units for specific fixed point formats for the multiplier and the multiply-accumulate operations were implemented.

## 4.3. Streaming I/O

While the reference implementation used an environment, where the input data were read using csv-files, TTA/TCE is an environment without operating system. There is also no file system available for implementing file-based I/Os. To simulate the filter system on the TTA, a stream-based I/O was implemented. With the intrinsic operations provided by the TCE environment it is possible to implement streaming I/O using specific stream FUs:

**Algorithm 3** Streaming I/O example on TTA

```
char byte;
int status;
_TCE_STREAM_IN_STATUS(0, status);
if (status == 0)
    break;
_TCE_STREAM_IN(0, byte);
_TCE_STREAM_OUT(byte);
```

Algorithm 3 uses the streaming TCE operations to read a single byte from the input stream and write it back to the output stream. When simulating the TTA the input are read from the *input.in* file, while the outputs are written to the *output.out* file.

The above-mentioned implementation using the stream FUs is sufficient for simulation purposes of the TTA. But when synthesizing the TTA processor on the FPGA, the input data still has to be loaded to the block ram. While VHDL implementations and HDB entries for the streaming I/Os are included in TCE, a buffer block ram is required to store the inputs, which results in storing the data on the FPGA. Because the block ram size of the FPGA is limited and due to the many coefficients of the filters saved in the data memory, large amounts of samples cannot be loaded to the FPGA in this implementation.

To simplify the process of reading the input samples, when synthesizing the TTA for the FPGA synthesis, the samples are directly read from the data memory and the streaming I/O is not used for the FPGA verification.

## 4.4. TTA Configuration Implementation

There are many ways to configure a TTA due to the high customizability. To optimize a TTA processor a basic TTA configuration was created. As shown in figure 4.6 the basic configuration has two transport buses and has an instruction width of 50 bits. It includes the basic FUs, the LSU responsible for executing all load and store instructions and an ALU capable of the basic arithmetic operations, addition and subtraction, shift operations and comparison operations. A simple multiplier FU was included to the TTA and a streaming FU to read the input data. It has two RF, one for booleans and another to store 32-bit data. Both are capable of storing up to 16 entries. What is not shown in figure 4.6 are the GCU, an immediate FU and a few FUs for debugging purposes.

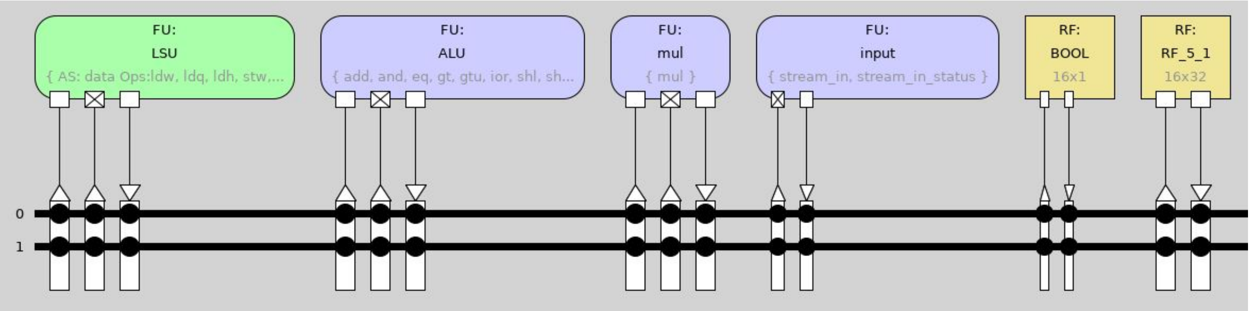

FU:
LSU
{ AS: data Ops:ldw, ldq, ldh, stw,...
FU:
ALU
{ add, and, eq, gt, gtu, ior, shl, sh...
FU:
mul
{ mul }
FU:
input
{ stream_in, stream_in_status }
RF:
BOOL
16x1
RF:
RF_5_1
16x32
0
1

Figure 4.6.: Section of the basic configuration

Using the basic configuration as the basis, the configurations 1xlevel_k, 2xlevel_k and 4xlevel_k_large were created with each having 5 different multiplier implementations. The configurations 1xlevel_k and 2xlevel_k are extended by 2 additional transport buses compared to the basic configuration, which results in an instruction width of 82 bits. An additional adder and either 1 or 2 FPMUL/FPMAC FUs with a fixed point multiplier implementation were included. Due to them only differing on an additional FPMUL/FPMAC FU, comparing the 1xlevel_k and 2xlevel_k configurations should show if and how much the FIR filter implementation can be parallelized and the effects on the performance.

Configuration 4xlevel_k_large instead has in total 6 transport buses and 4 FPMUL/FPMAC FUs. The exact details for each configuration can be seen in table A.1 of the appendix.

The multiplier was implemented with either 1, 2, 4, 8 or 16 multiplier stages, which are called levels in the configuration names, with a multiplier with $k$ stages meaning the execution of the multiplication in 32/k cycles, as already told before. A multiplier with 2 stages means that of the 32 arrays of the $32 \times 32$ array multiplier, only 2 are implemented in hardware, which are being used iteratively. Due to only implementing 2 of the 32 stages, the required area is reduced, while the multiplication itself requires 16 cycles.

A comparison between the configurations itself should also show which of the five multiplier implementations is the best in terms of computational performance. While the amount of clock cycles should sink with an implementation of more multiplier stages, due to the longer critical path the maximum possible clock frequency should also be lower.

As already told, configuration series 4xlevel_k_large has 4 FPMUL/FPMAC FUs. To allow the processor to more effectively use these FUs the TTA configuration was further extended to 6 transport buses instead of 4, which is also the reason it is called "large". Due to the higher amount of buses, the instruction width is also increased to 114 bits.

### 4.4.1. Further Optimizations

Lastly, using the configuration 4xlevel_8_large with an 8-stage multiplier implementation, a design space exploration was performed. Adding additional RFs or simple FUs can have a big influence on the performance of the TTA processor. Adding another RF can avoid unnecessary storing and loading of data to the data memory. Another adder for example can lead to more parallel instructions and an additional transport bus also allows potentially more simultaneous data transports per instruction. Nevertheless, one has to consider the cost of extra hardware. Using an additional bus extends the instruction width. Because of the many additional socket connections, the size of the processor grows larger and a higher power consumption also has to be expected.

These configuration with additional hardware compared to configuration 4xlevel_8_large, are called variants in this work.

#### Connection pruning

The above-mentioned modifications try to optimize the processor by reducing the total amount of clock cycles for the cost of a bigger processor. Naturally, it is also possible to reduce the size of the processor to reduce the resource utilization.

The configurations mentioned above are all implemented using a fully connected interconnection network. The high amount of connections should bloat the size of the configurations, which would result in a higher power consumption and resources used for the configuration, with the configurations 4xlevel_k_large being the largest of the non-variant configurations. Furthermore, a fully connected network on a bigger architecture tends to lead to a lower maximum clock frequency.[18]

TTA allows the designer to reduce the interconnection network. Pruned configurations were created using the configurations 2xlevel_8, 4xlevel_8_large. The pruned versions were created using the following strategy:

1. The register files are fully connected
2. All output ports are fully connected
3. Each input port has only one connection to the IC
4. The input port connections of the different FUs are evenly distributed on the transport buses

The RFs being fully connected allows them being reachable from every other FU. They have a direct connection with each input port, which allows for a relatively high speed. This is also the reason for strategy 2, so that register bypassing is allowed, since if they wouldn't have a direct connection to another FU, they would have to transfer the data to the RF first. Point 3 should decrease the overall size of the configuration and strategy 4 is taken, so that multiple transport buses can be used at the same time. In case every FU would have their connection on the same bus, then they would have to wait if it were occupied by another instruction.

An example can be seen in figure 4.7, which shows a section of the pruned_2xlevel_8 configuration. The mul23_mac23 FU is a FPMUL/FPMAC FU with a fixed shift value of 23 bits.

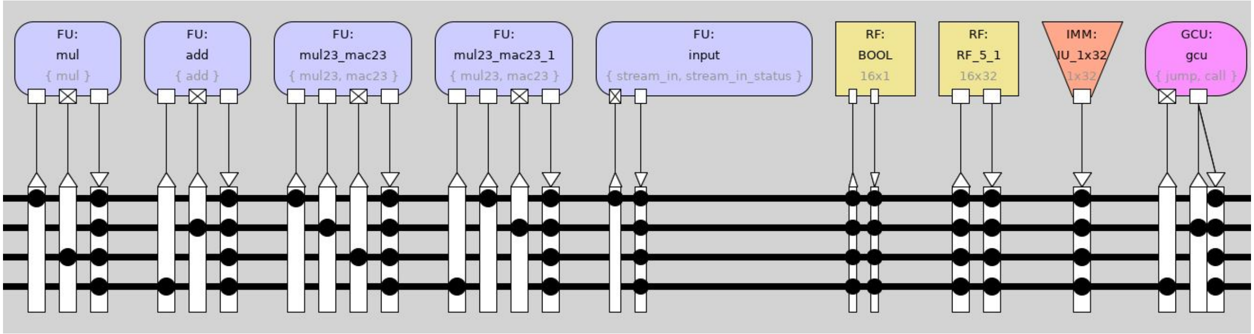


Figure 4.7.: Section of the pruned_2xlevel_8 configuration

## 4.5. Top-Level Design Entity

For an FPGA synthesis, a top-level design entity is required to connect the generated TTA core with its necessary components. The design is shown in figure 4.8.

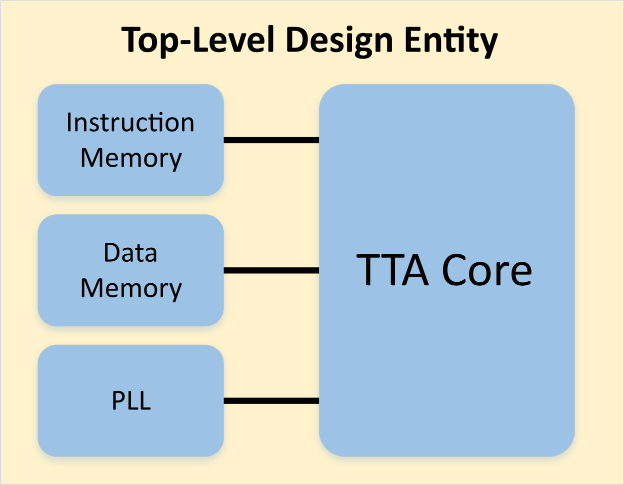


Figure 4.8.: Top-level design entity for the FPGA synthesis

The TTA core is generated using the Processor Generator introduced in chapter 2.2.4. Both bit images for the instruction memory and data memory components are generated with the Program Image Generator. The Block Memory Generator of the Vivado IP Catalog is used to generate the data memory component, which loads the data memory image file. The Clocking Wizard generates the PLL of the design, which receives its input clock signal from the FPGA board and distributes signals with different clock speeds. Finally, with the instruction memory component, taken from the TCE Manual[18], the top-level design entity can connect the different components.

A script was written to create a Vivado project for each TTA configuration, which includes generating the necessary IP components with their settings and the pin assignments for the FPGA board. After the synthesis of the whole design, the FPGA board can either be programmed with the generated bit file or can be evaluated using generated reports from Vivado. This includes a timing, power or utilization report, which can give information regarding possible timing errors, the power consumption of the different components and the resource utilization of the implemented design.

# 5. Evaluation

Based on the in chapter 4 presented modifications and extension of the basic architecture, different configurations of the TTA were created, which can be looked up in the appendix' table A.1. Using the in chapter 3 introduced filter system, the different TTA configuration are to be evaluated. Firstly, the evaluation setup, along with the test data sets, are presented. Following this, using a few of the selected TTA configuration, a short evaluation of the individual components of the filter system is done. Further evaluation regarding the performance, resource utilization and power consumption of the TTA configurations will be done using data determined by an FPGA synthesis.

## 5.1. Evaluation Setup

The implementation is evaluated through the tools of the TCE. Enabling breakpoints on the Processor Simulation tool ttasim allows to print out the starting cycle for each component per frame. Using this data, the total amount of cycles per component is calculated. Because of this, certain parts of the implementation are not included in the cycle count. This includes the initialization for each component. The I/O-related parts, using either the stream FU or just reading the inputs from the memory, are also not included.

### 5.1.1. Test Data Sets

The proposed implementation is evaluated using two different data sets. The first data set contains 9216 samples and records a coast-down of a machine. It is the shorter one of the two data sets and is also used for the FPGA synthesis, where the samples are loaded to the block memory. The filter system configuration features a decimation factor of 1 for the pre-decimation and as a result skips it. The oversampling factor of the interpolation factor is 6. Therefore, the second decimation filter also has a decimation factor of 6. This setup will be referred as *test setup A* in the following sections.

The second data set contains 255488 samples. The filter system configuration for the data set configures a pre-decimation factor of 31, unlike in test setup A. As such, the anti-aliasing filter of the decimation step is very large. The raw oversampling factor is 1,068, but gets rounded up to 2 according to equation 3.17. This setup will be referred as *test setup B*.

The actual numbers of samples that pass through the different components of the filter system vary. Figure 5.1 shows the number of samples that gets passed through the components for both test setups. During the time, when the FIR filter fills its internal state, the initial part of the output, also known as the filter transient, gets discarded by the implementation. This results in the band-pass filter and envelope detector dropping some samples.

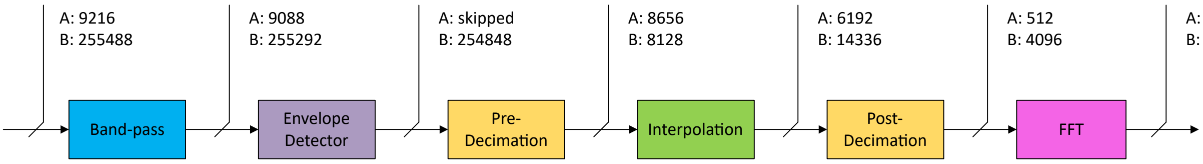


Figure 5.1.: Number of samples at each component

### 5.1.2. Filter System over different TTA Configurations

Using the above-mentioned test setups, the different TTA configurations mentioned in section 4.4 were evaluated. Figure 5.2 shows the average cycle count per sample for the configuration basic for both test setups A and B. The cycle count for the band-pass is nearly the same, because the components only uses the FIR filter with no additional computation and a fixed filter length of 128. The envelope detector requires fewer samples overall because of its shorter filter length. Furthermore, there is a noticeable difference between both test setups for the envelope detector. This is because the filter length is dependent on the filter system configuration. Instead of a hilbert filter length of 47 in test setup A, setup B only has a length of 31.

The component that requires the most cycles per sample is the interpolation filter. While the interpolation filter in test setup A requires slightly fewer cycles per sample than the band-pass filter, the interpolation in test setup B requires more than double the amount of cycles.

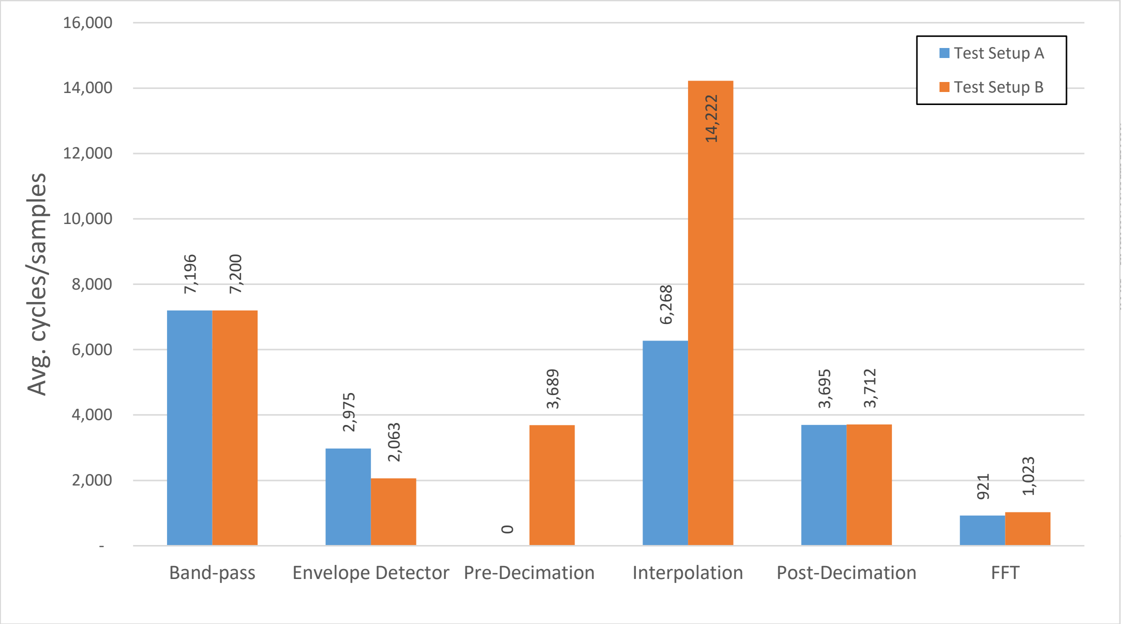


Figure 5.2.: Average cycles per sample for the basic configuration on test setups A and B

While for test setup A the pre-decimation was skipped, the post-decimation filter shows little difference between the two test setups like the band-pass filter. One might think that test setup B should require more cycles due to the large anti-aliasing filter. The decimation factor of test setup B has a decimation factor of 31, which results in a anti-aliasing length of $31 \times 64 - 1 = 1983$, due to equation 3.15. But, due to the decimation factor of 31, only every 31st sample is actually calculated, which results in effectively calculating a 64-tap FIR filter. This effect can be seen when comparing the band-pass and the decimation filter. The decimation filter requires half the cycles per sample than the band-pass with a filter length of 128 taps, which is double the length.

Figure 5.3 shows the average cycle count per sample for configurations 1xlevel_1. Comparing the basic configuration with configuration 1xlevel_1, the cycle count per sample is dropped by about 40% for each component, while the ratios between test setup A and B stayed the same. The cycle count can be further reduced by selecting a multiplier with more multiplier stages and, as such, a shorter latency.

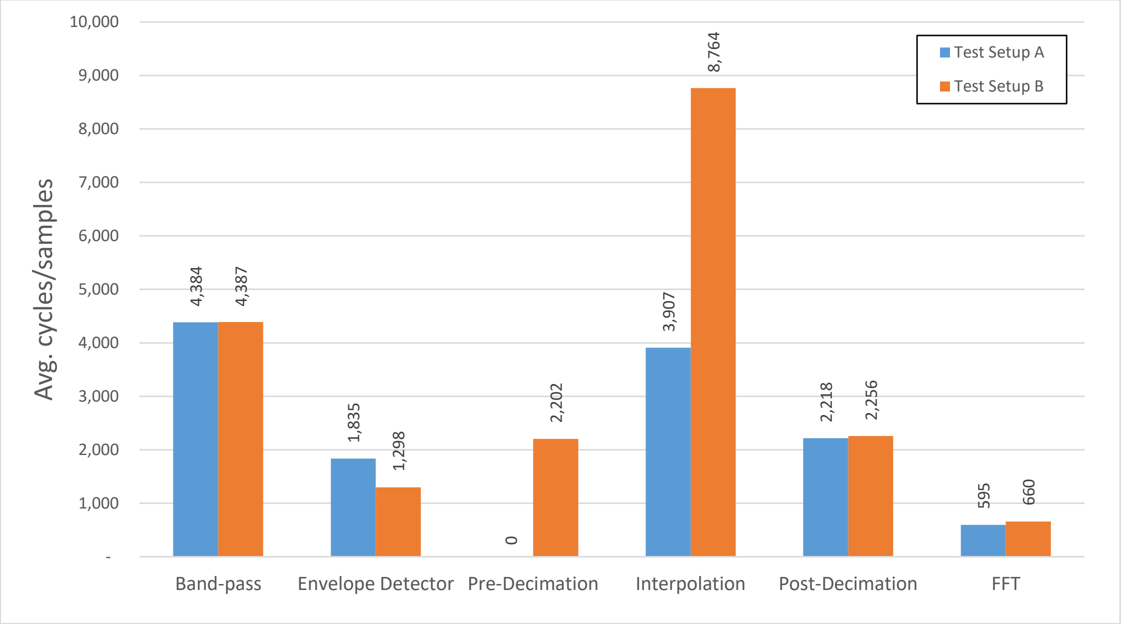


Figure 5.3.: Average cycles per sample for configuration 1xlevel_1 on test setups A and B

Configuration 1xlevel_16 switches out the multiplier with 1 stage, which has a latency of $32+1=33$, for one with 16 stages and a latency of $2+1=3$. As such, one could expect a speedup of around 90%, but due to other computations, the highest speedup achieved is around 86% at the band-pass filter. The other components, which calculate using an FIR filter, achieve a speedup of at least 72% with the envelope detector of test setup B being the slowest. Only the FFT doesn't calculate a FIR filter. Instead, it has to calculate many fixed point multiplication, but also many other operations and can only achieve a speedup of 54% for both setups.

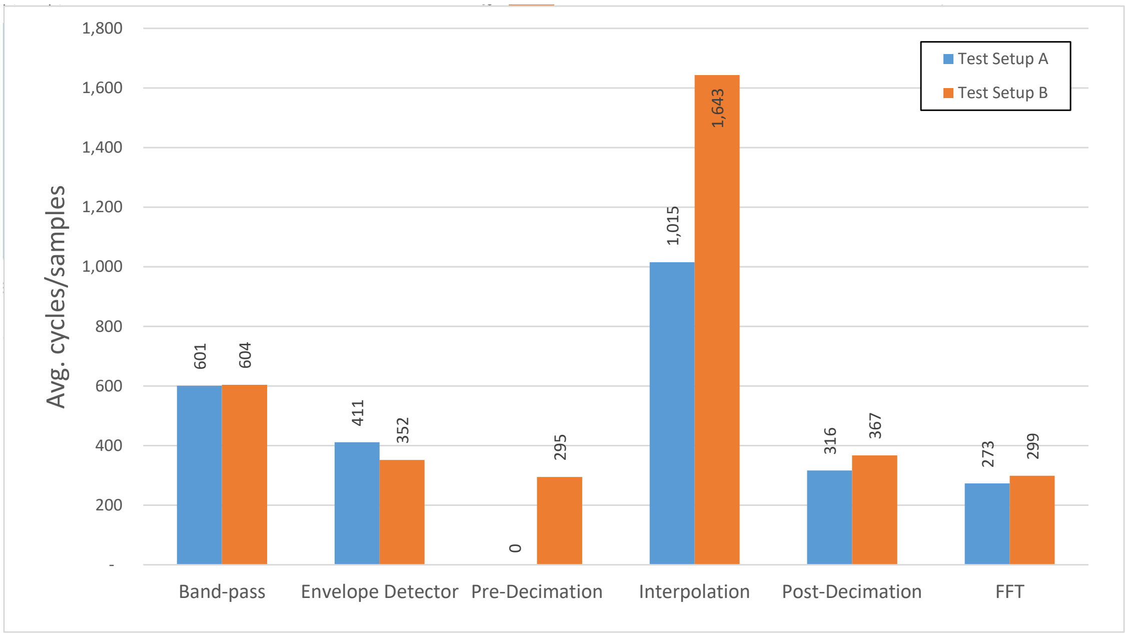


Figure 5.4.: Average cycles per sample for configuration 1xlevel_16 on test setups A and B

## 5.2. FPGA Synthesis

The FPGA synthesis will be performed on the Xilinx Zynq-7000 SoC ZC706 Evaluation Kit. The power consumption will be estimated using the synthesis software, due to the short execution time of the filter system and the many configurations. Using actual measured values the ratios of the power consumption between the different configuration can be verified. The fixed point multiplication instruction will be implemented using the array multiplier of the TTA core.

The frequency used for each configuration is chosen by at first configuring a clock frequency of 100 MHz. After the synthesis a negative WNS (worst negativ slack) indicates how much the maximum possible frequency is. Thereafter, the frequency can be changed using the PLL and the synthesis has to be done again. This step was repeated until the WNS became positive and the timing requirements were met for each configuration.

For the verification of the filter system on the FPGA, the test data set A with 9289 samples was stored into the block ram of the ZC706, which in turn increased the necessary area and power consumption. In a real use-case the filter system would not store the input signals in the block ram, but would receive them from e.g. an UART connection, which, in turn, would lower the block RAM size. After the calculation of the filter system control LEDs indicate a correct result.

### 5.2.1. Performance

The performance of a processor is measured by its speed. The speed corresponds to the number of program runs per second, which is also the time it takes for the filter system to process the 9289 samples of test setup A. The speed is proportional to the clock frequency and inversely proportional to the number of clock cycles required per program run.

$$\text{Performance}\left[\frac{\text{runs}}{\text{s}}\right] = \frac{\text{clock frequency}\left[\frac{\text{clock cycles}}{\text{s}}\right]}{\text{total number of clock cycles}\left[\frac{\text{clock cycles}}{\text{runs}}\right]}$$

In this evaluation, the clock frequency for each configuration is selected by using the maximum possible clock frequency, which meet the timing constraints. As such, the following evaluation is performed using the values obtained from the reports of the FPGA synthesis, which can be looked up in table A.2 in the appendix.

For the multiplier, the higher the number of stages of the multiplier implementation, the lower the required latency and the lower the number of cycles for the operation. Conversely, the clock frequency and the number of clock cycles increase with a smaller number of multiplier stages.

The performance of the configurations differ strongly with the configurable multiplier implementations. An evaluation of the effects regarding the multiplier stages on the amount of clock cycles, the maximum clock frequency and the performance was conducted using test setup A and the TTA configurations 1xlevel_k and 2xlevel_k. The results can be seen in figure 5.5 and 5.6.

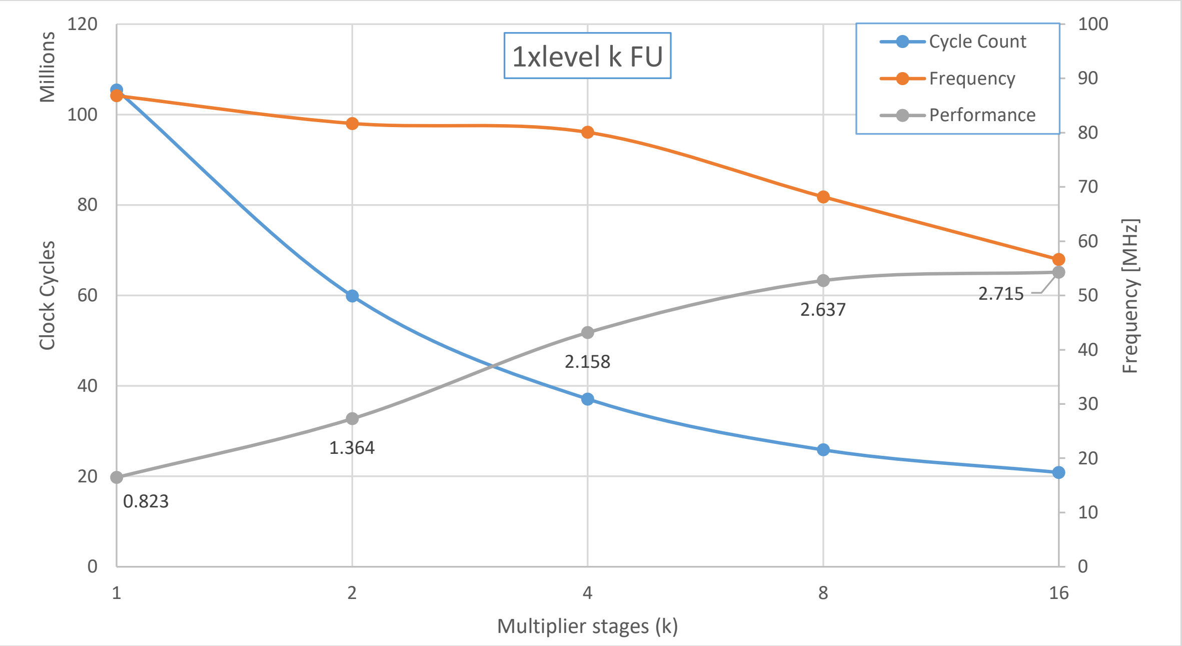


Figure 5.5.: Effects of the number of multiplier stages on the amount of clock cycles, max. clock frequency and performance in $[\frac{runs}{s}]$ using the 1xlevel_k configurations

As one can see, the more stages of a multiplier are implemented, the lower the latency of said multiplier and the lower the total amount of clock cycles. Furthermore, because of the bigger multiplier implementation, the critical path becomes longer and the maximum possible frequency gets lower. The configurations 1xlevel_k only have one FPMUL/FPMAC FU. Therefore, it has to calculate the FIR filter sequentially and cannot effectively utilize the four transport buses.

It is noteworthy that, while the configurations with 8 and 16 multiplier stages have nearly the same performance, configuration 1xlevel_8 with a performance of 2,637 $[\frac{runs}{s}]$ has 3,2 times the performance of configuration 1xlevel_1 with 0,823 $[\frac{runs}{s}]$. While the frequency only dropped by about 22% from 87 MHz to 68 MHz, the difference of the amount of clock cycles is huge. Configuration 1xlevel_1 with a cycle count of $105 \times 10^6$ requires more than four times the amount of clock cycles than configuration 1xlevel_8 with $25 \times 10^6$ cycles.

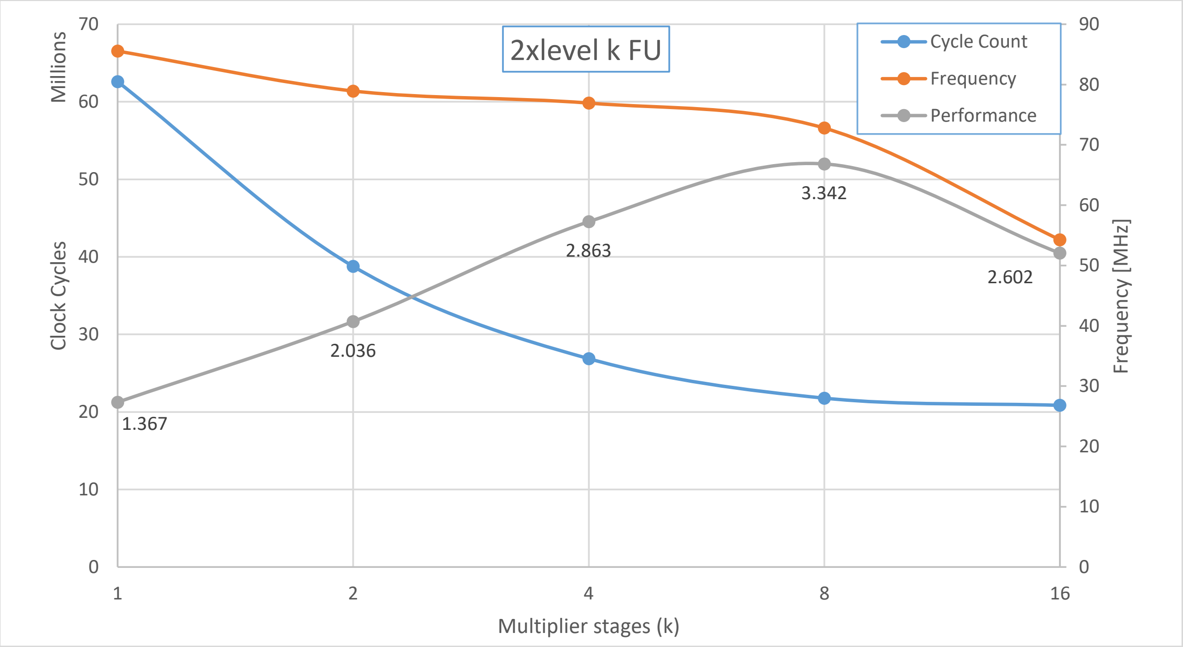


Figure 5.6.: Effects of the number of multiplier stages on the amount of clock cycles, max. clock frequency and performance in $[\frac{runs}{s}]$ using the 2xlevel_k configurations

Compared to the configurations 1xlevel_k, 2xlevel_k has in total two FPMUL/FPMAC FUs. Due to this additional FU, the FIR filter can be executed in parallel. Comparing configurations 1xlevel_1 and 2xlevel_1, the amount of clock cycles dropped by 40% from $105 \times 10^6$ to around $62 \times 10^6$ clock cycles. This results in a performance increase of around 65% to 1,367 $[\frac{runs}{s}]$. A performance increase can also be seen for the configurations with 2, 4 and 8 multiplier stages.

Noteworthy here is, that the performance of configuration 2xlevel_16 is lower than 1xlevel_16. While the maximum possible clock frequency decreased after adding another FPMUL/FPMAC FU, the total amount of clock cycles stayed around the same at $20,85 \times 10^6$. This means that configuration 2xlevel_16 cannot effectively utilize the second FPMUL/FPMAC FU. It seems its latency of 3 ($2$ for the multiplier $+ 1$) is too low. While executing the second FPMAC operation for the FIR filter, the FPMUL/FPMAC FU used for the first FPMAC operation already finished its calculation and can be used again. The second FPMUL/FPMAC FU would never be used or in case it is used, the compiler would also be able to use the first FU. Nevertheless, because of its lower frequency, the performance is lower compared to configuration 1xlevel_16 and also lower than 2xlevel_4. The more FPMUL/FPMAC FUs are added, the higher this difference should be shown.

Figure 5.6 clearly indicates, that the best performance for the filter system, at least in this implementation, is achieved using a multiplier implementation using 8 stages.

### 5.2.2. Resource Utilization

The resource utilization of an FPGA can be represented by the amount of LUTs used on the FPGA board. The number of LUTs varies with different configuration using different amount of multiplier stages. For the configurations 1xlevel_k using a multiplier with 32 stages requires 4057 LUTs, 1 stage 3595 LUTs and 4 stages 3690 LUTs. The number of transport buses and additional FUs also increase the amount of required LUTs. The resource utilization of selected configurations with different number of multiplier stages is shown in table 5.1.

Table 5.1.: Resource utilization and instruction size of selected configurations with different multiplier stages on the FPGA

| configuration | Mul stages | used LUTs | used registers | used Block RAMs | instruction width | instruction size [B] |
|---|---|---|---|---|---|---|
| basic | - | 2 217 | 1 259 | 306 | 50 | 194 106 |
| 1xlevel_k | 1 | 3 595 | 1 733 | 338 | 82 | 272 086 |
| | 8 | 3 775 | 1 729 | 338 | 82 | 251 306 |
| | 16 | 4 057 | 1 729 | 338 | 82 | 239 235 |
| 2xlevel_k | 1 | 4 169 | 2 041 | 338 | 82 | 260 043 |
| | 8 | 4 593 | 2 035 | 338 | 82 | 240 188 |
| | 16 | 5 030 | 2 033 | 338 | 82 | 239 256 |
| 4xlevel_k_large | 1 | 6 924 | 2 739 | 368 | 114 | 348 726 |
| | 8 | 7 644 | 2 723 | 368 | 114 | 327 693 |
| | 16 | 8 648 | 2 723 | 368 | 114 | 326 596 |
| pruned 2xlevel_8 | 8 | 3 849 | 1 994 | 335 | 79 | 242 135 |
| pruned 4xlevel_8_large | 8 | 5 809 | 2 647 | 364 | 109 | 335 884 |
| available resources<br>Device | | LUTs | registers | Block RAMs | | |
| XC7Z045 FFG900–2 | | 218 600 | 437 200 | 545 | | |

The configuration 4xlevel_16_large uses nearly four times the amount of LUTs compared to the basic configuration. Furthermore, the size of the instructions, which have to be loaded to the FPGA, is increased by about 70%, while the instruction width is more than doubled.

Because the configurations 1xlevel_k and 2xlevel_k only differ on an additional FPMUC/FPMAC FU with a fixed shift amount and its socket connections, the amount of LUTs and registers required for the additional FU can be inferred. The FU with 1 multiplier stage on 2 transport buses, while being fully connected, requires $4169 - 3595 = 574$ LUTs and $2041 - 1733 = 308$ registers. Using 16 multiplier stages would require 973 LUTs and 304 registers.

The XC7Z045 FFG900–2 has in total 545 of 36-Kb block RAMs, with each configuration occupying around 60% of it. While the 9216 pairs of input samples occupy a huge part of it, the many coefficients for the filter components and twiddle factors for the FFT also require a lot of memory.

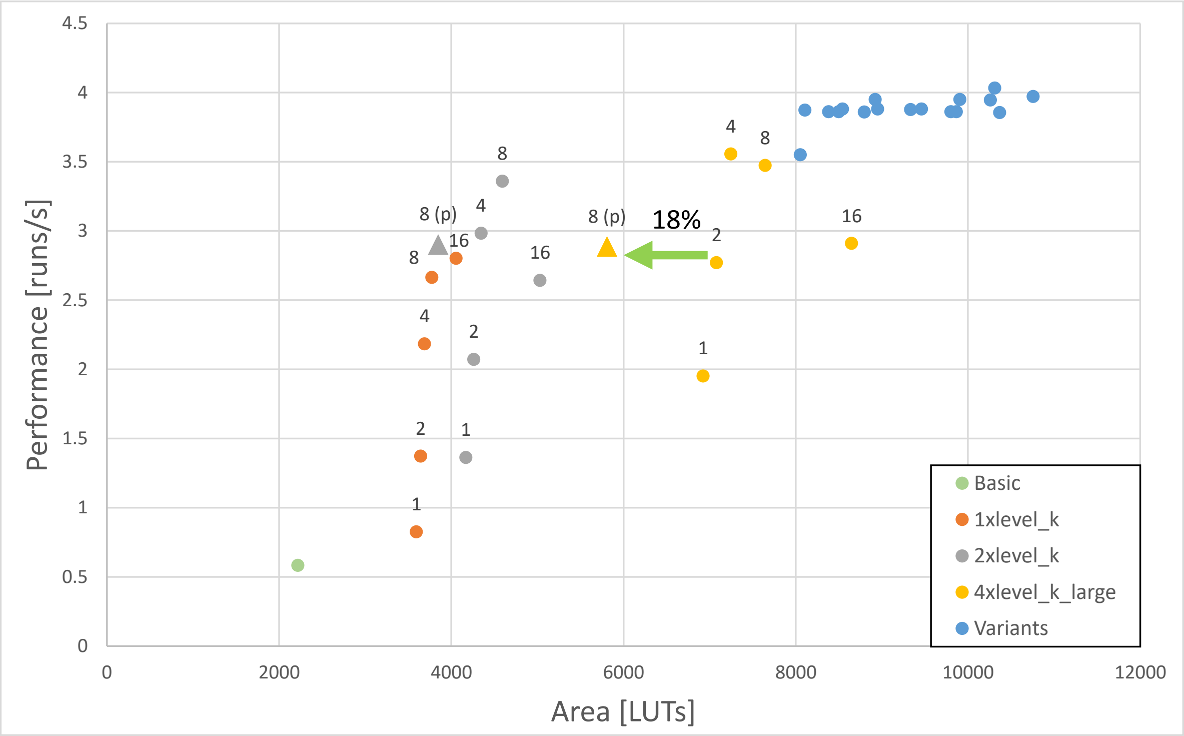


Figure 5.7.: Area and performance of the TTA configurations using different amounts of multiplier stages. The triangle and label (p) imply a pruned configuration.

Figure 5.7 shows a comparison between the area required, represented by the number of used LUTs, and the performance of each configuration. It is observable by comparing the configurations 1xlevel_k and 2xlevel_k, that with an additional fixed point multiplier the number of LUTs used increases. The performance also increases for all configurations, except for configuration 2xlevel_16, where it instead is lower.

Comparing the configurations with different amounts of multiplier stages, multiplier implemented with more stages require more LUTs. For example for the configurations 1xlevel_k, each time a bigger multiplier is implemented, its point on the diagram is on the right side compared to the ones with the smaller multiplication implementation. What is also noticeable is that the variant configuration with 8 multiplier stages do not change much in terms of performance, even though there are up to three additional RFs, 2 transport buses and an additional FPMUL/FPMAC FU.

Regarding the pruned configurations, one can see that though the performance dropped in comparison to their un-pruned versions, they require much less LUTs. This effect is less noticeable with configuration pruned_2xlevel_8, since compared to configuration pruned_4xlevel_8_large, it only has 4 buses. Compared to configuration 4xlevel_8_large with 7644 LUTs, the pruned version only requires 5809 LUTs, which is a reduction of 24%. Even more important is, that, while having a slightly better performance than configuration 4xlevel_2_large, it needs 18% less LUTs.

Regarding the variant configurations in the upper right corner of the diagram, while many different options were tried by adding FUs, transport buses or RFs, the performance stayed around the same level. This means that the program can not efficiently use the additional hardware.

The most number of LUTs requires variant 17 with 10759 LUTs, which is more than 4,5 times the basic configuration, which is the smallest one with 2217 LUTs. Meanwhile, the best performance has variant 12 with 4,034 runs/s which is an increase of 590% of the basic configuration.

### 5.2.3. Power Consumption

In this chapter, the power consumption of the TTA processor will be evaluated. Using an FPGA board, there are two methods this can be done. First, using the FPGA synthesis, Vivado can report estimated values for each configuration. The other method is to directly measure the power consumption using e.g. a multimeter as seen in figure 5.8. The power consumption measured not only is the power for the processor, but also includes the power for the fans and other components.

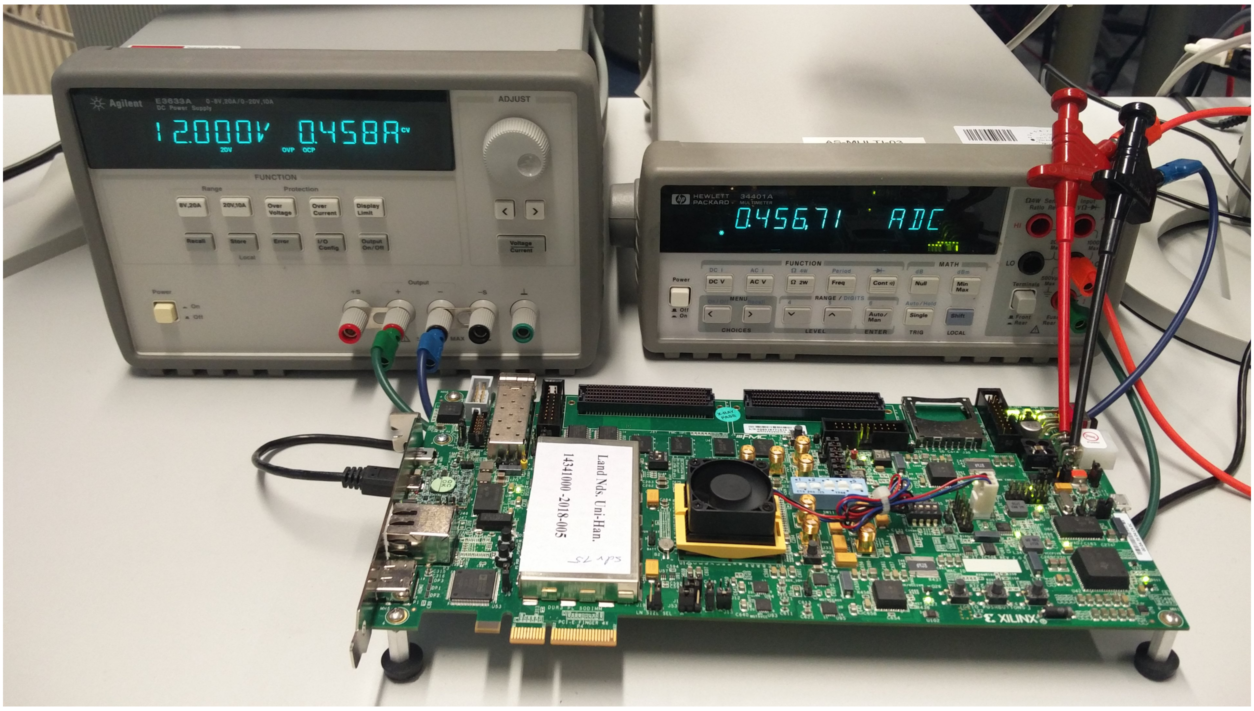


Figure 5.8.: Photo of the measuring station. The Evaluation Board is powered by an external power supply. The power consumption is measured using a multimeter.

The total power consumption of the evaluation board is calculated as:

$$P_{total} = P_{TTA} + P_{idle}, \tag{5.1}$$

with $P_{TTA}$ being the power of the TTA processor which is programmed on the evaluation board and $P_{idle}$ being the idle power of the board.

The overall power consumption for the TTA processor from Vivado is an estimate calculated from the mean switching activity and is calculated from the static $P_{stat}$ and dynamic power $P_{dyn}$:

$$P_{TTA} = P_{stat} + P_{dyn} \tag{5.2}$$

Figure 5.8 shows the setup to measure the power consumption of the evaluation board using a multimeter. Because the filter system is executed in a very short time, an alternative implementation was used with the filter system using an infinitive repeating input of test setup A. Using the directly measured power and the estimated power consumption from Vivado, an estimated $P_{idle}$ can be calculated:

Table 5.2.: Power consumption of selected configurations with a clock frequency at 50 MHz

| configuration | measured P$_{total}$ | estimated P$_{TTA}$ | calculated P$_{idle}$ |
|---|---|---|---|
| 2xlevel_1 | 5 438 mW | 438 mW | 5 000 mW |
| 2xlevel_8 | 5 450 mW | 440 mW | 5 010 mW |
| 2xlevel_16 | 5 438 mW | 442 mW | 4 996 mW |
| 4xlevel_1 | 5 478 mW | 489 mW | 4 989 mW |
| 4xlevel_8 | 5 501 mW | 498 mW | 5 003 mW |
| 4xlevel_16 | 5 484 mW | 502 mW | 4 982 mW |
| pruned_2xlevel_8 | 5 440 mW | 434 mW | 5 006 mW |
| pruned_4xlevel_8 | 5 474 mW | 465 mW | 5 009 mW |

As seen in table 5.2 the calculated $P_{idle}$ using the estimated values tends to be around 5000 mW. This shows, that while the accuracy of the estimated values of Vivado are not proven, the comparison of the configurations itself can tell which processor requires more power. Furthermore, the power report of Vivado gives detailed information about the power consumption for each component. A detailed report for the configuration 2xlevel_8 is shown in table 5.3:

Table 5.3.: Overview of the power consumption of the individual components for configuration 2xlevel_8 taken from the power report of Vivado

| | Power (mW) |
|---|---|
| $P_{TTA}$/Total On-Chip Power | 530 |
| Device Static | 222 |
| Dynamic | 307 |
| Instruction memory | 110 |
| Data memory | 15 |
| PLL | 141 |
| TTA core | 41 |
| ALU | 2 |
| LSU | 2 |
| MUL FU | 1 |
| FPMUL/FPMAC FU(1) | 6 |
| FPMUL/FPMAC FU(2) | 6 |
| RTC_TIMER FU | 2 |
| Interconnection Network | 5 |
| Instruction Decoder | 6 |
| Instruction Fetch | 6 |
| RF | 4 |

The values of the TTA core's components that are listed don't add up to 41 mW, since smaller FU like the ADD FU are not listed. Nevertheless, one can see that the TTA core itself only consumes a small fraction of the total power. Most of the dynamic power consumption stems from the instruction memory and the PLL for the configurable clock. A simple way to reduce the power consumption, is by omitting the PLL entirely, since the input clock of the PLL itself stems from a programmable user clock of the evaluation board, whose frequency can be changed through an $I^2C$ interface.[24]

### Evaluation on the power consumption of the different TTA configurations

The power required for the TTA processor increases linearly with the size of the processor and the clock frequency used. Larger configuration with 16 multiplier stages have a lower power consumption due to their lower clock frequency than smaller multipliers. Figure 5.9 shows a comparison between the power consumption and the performance of the TTA configurations. A low power consumption with a high performance (upper left corner in the diagram) corresponds to the optimum. The dashed line indicates the same power consumption/performance efficiency.

The configurations with the lowest power consumption are the basic configuration at 466 mW and configuration 1xlevel_16 and 2xlevel_16, both at 470 mW. The configuration with the highest power consumption is variant 15 at 678 mW. Noticeable is, that, though the basic configuration and 2xlevel_16 consume nearly the same power, the performance difference is 4,5 times higher. The power consumption of 2xlevel_16 is nearly the same as the basic configuration, even with its

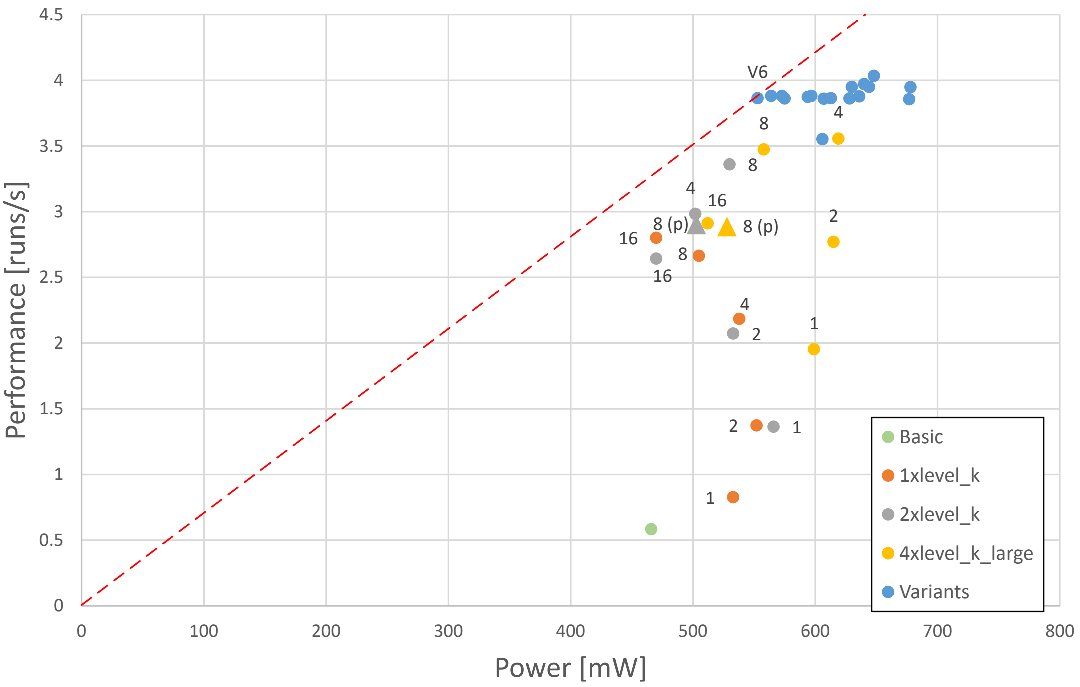


Figure 5.9.: Power consumption and performance of the TTA configurations using different amounts of multiplier stages. The triangle and label (p) imply a pruned configuration.

additional hardware, because of the lower clock frequency used. It has to be remembered, that each configuration can run on a lower frequency. While the performance would be lower, the power consumption would also decrease, so it is possible to configure the frequency, so that all configurations have the same power consumption.

Using the estimated values, it is noticeable that the configuration with only 1 and 2 multiplier stages are less efficient than the other ones. They have the same or even a higher power consumption than the configuration with 8 or 16 multiplier stages of the same series, but their performance is noticeably worse. One might think, that they should require less power, due to a smaller multiplier implementation. In fact, that would be true, if the configurations would be run using the same clock frequency, but they are all using their maximum possible frequency. Because of this reason, the configurations with 1 and 2 multiplier stages consume more power.

While the pruned versions of configuration 2xlevel_8 and 4xlevel_8_large consume less power, they have also a worse performance. In fact, pruned_4xlevel_8_large has a worse power consumption/performance efficiency than configuration 4xlevel_8_large. The best power consumption/performance efficiency has variant 6 with an additional RF and an Add/Mul/Sub FU compared to 4xlevel_8_large.

Figure 5.10 shows the power consumption and the area normalized to the performance of each configuration. It shows how efficiency each configuration is in terms of power and area or rather number of LUTs. A processor close to the origin (0,0) corresponds to the optimum. The further left in the diagram a processor configuration is, the better the energy efficiency. Similarly, the further down, the better the area efficiency.

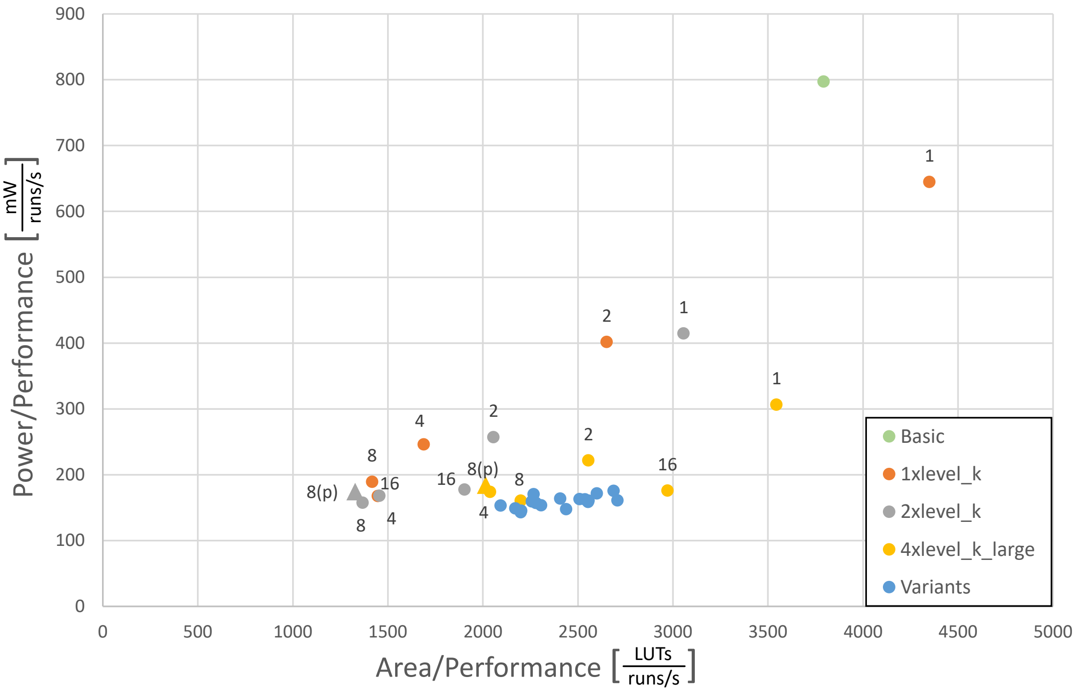


Figure 5.10.: Power consumption and area of the TTA configurations using different amounts of multiplier stages normalized to the performance. The triangle and label (p) imply a pruned configuration.

The configurations 2xlevel_8 and the pruned version of it show the best ratio between area and power consumption normalized to the performance show. The worst efficiency are shown by the basic configuration and configuration 1xlevel_1.

While many configurations have a power/performance value of 170 $\frac{mW}{runs/s}$, they vary greatly on their area/performance efficiency. One reason for that is, because the power consumption values for this evaluation include not only the TTA core, but also the PLL, instruction memory and data memory component. As such, the differences between the different configurations of the TTA core's power consumption are not highlighted as much, because an offset is always included in the power values.

# 6. Conclusion

Using the TTA based Co-Design Environment (TCE) multiple TTA configurations were created for a complex filter system implementing a computed order analysis in fixed point. Opportunities for further configuration adjustments were made possible by additional function units and application-specific additions. Some examples are a hardware fixed point multiplier/multiply-accumulate and additional register files and transport buses. The hardware multiplier was designed to be configurable, so that the operation can be executed in 1 until 16 clock cycles. These configurations were synthesized on the Xilinx Zynq-7000 SoC ZC706 Evaluation Kit. Using the data of the FPGA synthesis, an evaluation was performed regarding their power consumption, performance and required area, which is represented here by the number of used LUTs.

The computational performance increases with the addition of the custom fixed point multiplier function unit. With more complex TTA configurations, the number of LUTs increases as far as sevenfold compared to the basic configuration. Meanwhile, the power consumption of each configuration lie in the range of 466 mW and 678 mW. A big difference in the power consumption between the configurations cannot be seen, because the values include the power for the PLL, instruction memory and data memory components, with the TTA core only consuming a small part of the total power. Due to this, only a fraction of the total power is influenced by the configured TTA processor design.

Case studies on configurations with 6 transport buses and 4 fixed point multiplier FUs show, that through pruning a configuration, which contains a bigger multiplication implementation, a more efficient configuration compared to one with a smaller multiplication implementation can be obtained. For example the pruned version of the configuration with a 8-stage multiplication implementation (pruned 4xlevel_8_large) has even a slightly better performance than the configuration with only a 2-stage multiplication implementation, while in total requiring 18% less LUTs.

The configuration 2xlevel_8 and its pruned version have the best ratio between area and power consumption efficiency regarding their performance. Meanwhile, the best power/performance efficiency has variant 6. Naturally, by adding additional hardware, the performance of the processor can be increased. But, as shown for the many variant configurations, just adding FUs, transport buses, and RFs will not lead to a good efficiency. While the area and power required increased with the additional hardware, the performance stayed around the same level. The program can not efficiently use this additional hardware in this implementation and for these configurations.

Still, the Transport Triggered Architecture is well suited for application-specific tasks. This is expressed in simple parameterization and a large design space with regard to performance, area and power consumption.

# A. Appendix

Table A.1.: Overview of the evaluated TTA configuration.

| **Configuration**[1] | **Mul rows** | **FPMUL/ FPMAC FU** | **Bus**[2] | **RF**[3] | **BOOL** | **Add FU** | **Add/Mul/Sub FU** |
|---|---|---|---|---|---|---|---|
| basic | - | 0 | 2 | 1 | 1 | 0 | 0 |
| 1xlevel_k | 1 | 1 | 4 | 1 | 1 | 1 | 0 |
|  | 2 | 1 | 4 | 1 | 1 | 1 | 0 |
|  | 4 | 1 | 4 | 1 | 1 | 1 | 0 |
|  | 8 | 1 | 4 | 1 | 1 | 1 | 0 |
|  | 16 | 1 | 4 | 1 | 1 | 1 | 0 |
| 2xlevel_k | 1 | 2 | 4 | 1 | 1 | 1 | 0 |
|  | 2 | 2 | 4 | 1 | 1 | 1 | 0 |
|  | 4 | 2 | 4 | 1 | 1 | 1 | 0 |
|  | 8 | 2 | 4 | 1 | 1 | 1 | 0 |
|  | 16 | 2 | 4 | 1 | 1 | 1 | 0 |
| 4xlevel_k_large | 1 | 4 | 6 | 1 | 1 | 1 | 0 |
|  | 2 | 4 | 6 | 1 | 1 | 1 | 0 |
|  | 4 | 4 | 6 | 1 | 1 | 1 | 0 |
|  | 8 | 4 | 6 | 1 | 1 | 1 | 0 |
|  | 16 | 4 | 6 | 1 | 1 | 1 | 0 |
| variant 1 | 8 | 4 | 6 | 2 | 1 | 1 | 0 |
| variant 2 | 8 | 4 | 6 | 3 | 1 | 1 | 0 |
| variant 3 | 8 | 4 | 6 | 1 | 1 | 1 | 0 |
| variant 4 | 8 | 4 | 7 | 2 | 1 | 1 | 0 |
| variant 5 | 8 | 4 | 7 | 3 | 1 | 1 | 0 |
| variant 6 | 8 | 4 | 6 | 2 | 1 | 1 | 1 |
| variant 7 | 8 | 4 | 6 | 3 | 1 | 1 | 1 |
| variant 8 | 8 | 4 | 7 | 2 | 1 | 1 | 1 |
| variant 9 | 8 | 4 | 7 | 3 | 1 | 1 | 1 |
| variant 10 | 8 | 4 | 8 | 3 | 1 | 1 | 1 |
| variant 11 | 8 | 4 | 8 | 3 | 2 | 1 | 1 |
| variant 12 | 8 | 4 | 8 | 4 | 2 | 1 | 1 |
| variant 13 | 8 | 5 | 7 | 2 | 1 | 1 | 0 |
| variant 14 | 8 | 5 | 7 | 3 | 1 | 1 | 0 |
| variant 15 | 8 | 5 | 7 | 4 | 1 | 1 | 0 |
| variant 16 | 8 | 5 | 8 | 3 | 1 | 1 | 0 |
| variant 17 | 8 | 5 | 8 | 4 | 1 | 1 | 0 |

[1] Besides the units in the table, each configuration has an additionally LSU, ALU, GCU, immediate unit, Mul FU and a stream FU as basic components and a WRITE_LEDS, RTC_RTIMER and STDOUT FU for debugging purposes.

[2] Each RF can store up to 16 32-bit words.
[3] Each BOOL RF can store up to 16 booleans.

Table A.2.: Information of the implemented TTA configuration. Values from the FPGA synthesis are taken using test setup A.

| Configuration | Mul Config. | Cycle Count | | Power [mW] | Periode [ns] | LUTs | Registers | Performance [runs/s] | Instruction Size [B] |
|---|---|---|---|---|---|---|---|---|---|
| | | A | B | | | | | | |
| basic | - | 171 079 403 | 3 480 285 349 | 466 | 10,00 | 2 217 | 1 259 | 0,585 | 191 106 |
| 1xlevel_k | 1 | 105 006 439 | 2 120 092 249 | 533 | 11,52 | 3 595 | 1 733 | 0,827 | 272 086 |
| | 2 | 59 428 968 | 1 162 809 261 | 552 | 12,24 | 3 644 | 1 733 | 1,374 | 254 190 |
| | 4 | 36 644 781 | 684 212 564 | 538 | 12,49 | 3 690 | 1 730 | 2,185 | 245 354 |
| | 8 | 25 586 156 | 452 756 493 | 505 | 14,67 | 3 775 | 1 729 | 2,665 | 241,306 |
| | 16 | 20 198 423 | 339 504 105 | 470 | 17,66 | 4 057 | 1 729 | 2,803 | 239 235 |
| 2xlevel_k | 1 | 62 724 567 | 1 219 783 025 | 566 | 11,69 | 4 169 | 2 041 | 1,364 | 260 043 |
| | 2 | 38 082 349 | 708 170 319 | 533 | 12,67 | 4 261 | 2 037 | 2,073 | 248 081 |
| | 4 | 25 766 855 | 452 418 276 | 502 | 13,00 | 4 345 | 2 035 | 2,985 | 242 228 |
| | 8 | 21 660 485 | 369 190 533 | 530 | 13,74 | 4 593 | 2 035 | 3,361 | 240 188 |
| | 16 | 20 520 526 | 346 422 800 | 470 | 18,43 | 5 030 | 2 033 | 2,644 | 239 256 |
| 4xlevel_k_large | 1 | 42 661 639 | 792 740 694 | 599 | 12,00 | 6 924 | 2 739 | 1,953 | 348 726 |
| | 2 | 28 485 270 | 503 947 184 | 615 | 12,67 | 7 080 | 2 731 | 2,771 | 336 186 |
| | 4 | 21 441 522 | 360 531 592 | 619 | 13,11 | 7 248 | 2 727 | 3,557 | 330 073 |
| | 8 | 19 185 071 | 316 136 216 | 558 | 15,00 | 7 644 | 2 723 | 3,475 | 327 693 |
| | 16 | 18 080 503 | 294 053 741 | 512 | 19,00 | 8 648 | 2 723 | 2,911 | 326 596 |
| variant 1 | 8 | 17 178 189 | 282 741 214 | 594 | 15,03 | 8 107 | 3 253 | 3,874 | 309 311 |
| variant 2 | 8 | 17 258 948 | 282 611,366 | 575 | 15,00 | 8 383 | 3 789 | 3,863 | 304 366 |
| variant 3 | 8 | 19 204 804 | 316 136 216 | 606 | 14,66 | 8 053 | 2 742 | 3,552 | 373 701 |
| variant 4 | 8 | 17 175 289 | 282 717 029 | 564 | 15,00 | 8 545 | 3 274 | 3,882 | 352 739 |
| variant 5 | 8 | 17 254 719 | 282 537 319 | 630 | 14,67 | 8 922 | 3 811 | 3,951 | 363 035 |
| variant 6 | 8 | 17 254 366 | 283 830 805 | 553 | 15,00 | 8 498 | 3 367 | 3,864 | 311 662 |
| variant 7 | 8 | 17 271 538 | 282 424 171 | 607 | 15,00 | 8 796 | 3 902 | 3,860 | 319 902 |
| variant 8 | 8 | 17 252 239 | 283 813 700 | 597 | 14,93 | 8 953 | 3 390 | 3,882 | 355 453 |
| variant 9 | 8 | 17 267 969 | 282 391 390 | 636 | 14,93 | 9 334 | 3 928 | 3,879 | 365 568 |
| variant 10 | 8 | 17 259 000 | 282 142 017 | 628 | 15,00 | 9 801 | 3 960 | 3,863 | 410 729 |
| variant 11 | 8 | 17 258 403 | 282 132 753 | 644 | 14,67 | 9 908 | 4 003 | 3,951 | 428 320 |
| variant 12 | 8 | 16 898 510 | 274 350 758 | 648 | 14,67 | 10 311 | 4 534 | 4,034 | 427 200 |
| variant 13 | 8 | 17 175 289 | 282 717 029 | 573 | 15,00 | 6 461 | 3 584 | 3,882 | 352 723 |

continued on next page

continued from last page

| Configuration | Mul Config. | Cycle Count | | Power [mW] | Periode [ns] | LUTs | Registers | Performance [runs/s] | Instruction Size [B] |
|---|---|---|---|---|---|---|---|---|---|
| | | A | B | | | | | | |
| variant 14 | 8 | 17 254 719 | 282 573 319 | 613 | 15,00 | 9 866 | 4 121 | 3,864 | 363 035 |
| variant 15 | 8 | 16 855 998 | 273 779 574 | 678 | 15,03 | 10 262 | 4 659 | 3,948 | 377 951 |
| variant 16 | 8 | 17 254 824 | 282 579 001 | 677 | 15,03 | 10 368 | 4 152 | 3,857 | 407 841 |
| variant 17 | 8 | 16 855 998 | 273 779 574 | 640 | 14,93 | 10 759 | 4 690 | 3,973 | 425 600 |
| pruned_2xlevel_8 | 8 | 23 527 204 | 402 926 345 | 503 | 14,67 | 3 849 | 1 994 | 2,898 | 242 135 |
| pruned_4xlevel_8_large | 8 | 23 128 124 | 397 863 813 | 528 | 15,00 | 8 809 | 2 647 | 2,882 | 335 884 |

# Bibliography

[1] R. J. M. Rao, "What is keyphasor? how does keyphasor works?" https://instrumentationtools.com/keyphasor/ Last accessed: 24.01.2021.

[2] K. Fyfe and E. Munck, "Analysis of computed order tracking," *Mechanical Systems and Signal Processing*, no. 11(2), p. 187–205, 1997.

[3] J. Rinke, "Evaluation and optimization of a complex filter system using different application-specific instruction-set processor configurations," Master's thesis, Leibniz Universität Hannover, 2021.

[4] P. Saavedra and C. Rodriguez, "Accurate assessment of computed order," *Shock and Vibration*, no. 13, pp. 13–32, 2006.

[5] M. Werner, *Digitale Signalverarbeitung mit MATLAB®: Grundkurs mit 16 ausführlichen Versuchen*, 01 2019.

[6] P. Pirsch, *Architekturen der digitalen Signalverarbeitung*. Vieweg+Teubner Verlag, 1996.

[7] J. Courrech and M. Gaudel, "Envelope analysis - the key to rolling-element bearing diagnosis," 2001.

[8] C. Stoll, "What is order analysis?" https://vibrationresearch.com/blog/what-is-order-analysis/ Last accessed: 20.01.2021.

[9] C. N. Tan and J. Mathew, *Monitoring the Vibrations of Variable and Varying Speed Gearboxes*, 1990. [Online]. Available: https://search.informit.org/doi/10.3316/informit.439010843027615

[10] R. Potter and M. Gribler, "Computed order tracking obsoletes older methods," *SAE Transactions*, vol. 98, pp. 1099–1103, 1989.

[11] H. Herlufsen, *Order Tracking Analysis*, 1995. [Online]. Available: https://www.bksv.com/media/doc/bv0047.pdf

[12] P. Jääskeläinen, "Tce project: Co-design of application-specific processors with llvm-based compilation support," 2010, http://blog.llvm.org/2010/06/tce-project-co-design-of-application.html Last accessed: 10.02.2021.

[13] J. Nurmi, *Processor Design: System-On-Chip Computing for ASICs and FPGAs*. Springer, 2007.

[14] P. Jääskeläinen, A. Tervo, G. Payá Vayá, T. Viitanen, N. Behmann, J. Takala, and H. Blume, "Transport-triggered soft cores," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 83–90.

[15] Y. He, D. She, B. Mesman, and H. Corporaal, "Move-pro: A low power and high code density tta architecture," in *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, 2011, pp. 294–301.

[16] P. Priya and S. Ashok, "Iir digital filter design using xilinx system generator for fpga implementation," in *2018 International Conference on Communication and Signal Processing (ICCSP)*, 2018, pp. 0054–0057.

[17] T. U. Customized Parallel Computing (CPC) research group, "Tta-based co-design environment (tce) tools," http://openasip.org/ Last accessed: 16.01.2021.

[18] T. U. Customized Parallel Computing research group, *TTA-based Co-design Environment*, v1.22 ed., 2020, http://openasip.org/user_manual/TCE.pdf Last accessed: 07.01.2021.

[19] J. Volder, "The cordic computing technique," in *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, ser. IRE-AIEE-ACM '59 (Western). New York, NY, USA: Association for Computing Machinery, 1959, p. 257–261. [Online]. Available: https://doi.org/10.1145/1457838.1457886

[20] J. Muller, *Elementary Functions: Algorithms and Implementation*. Birkhäuser Boston, 2016. [Online]. Available: https://books.google.de/books?id=eNaCDQAAQBAJ

[21] S. W. Smith, *Moving Average Filters*, 1999. [Online]. Available: https://www.analog.com/media/en/technical-documentation/dsp-book/dsp_book_Ch15.pdf

[22] "Xilinx vivado," https://www.xilinx.com/products/design-tools/vivado.html Last accessed: 07.02.2021.

[23] S. Gesper, "Implementierung eines vliw-mips-prozessors für hochtemperaturanwendungen mit compilerunterstützung," Master's thesis, Leibniz Universität Hannover, 2018.

[24] Xilinx, *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC User Guide*, v1.8 ed., 2019, https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf Last accessed: 08.02.2021.