High-Level Synthesis for Efficient Design and Verification

Alfred Johansson
elt13aj2@student.lu.se
John Johansson
elt13jjo@student.lu.se

Department of Electrical and Information Technology
Lund University EITM01/EITM02

Supervisor: Liang Liu

Examiner: Erik Larsson

February 19, 2020

LUND UNIVERSITY

LTH
FACULTY OF ENGINEERING

ERICSSON

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Designing hardware using *High Level Synthesis* automates parts of the digital hardware design process. By automating the process control is passed from the designer to the tool, thus it is highly important that the tool generates high performance hardware in terms of area and speed. This thesis explores the tool performance of Vivado HLS using two designs implemented anew with *High Level Synthesis* and *Hardware Description Language*. The evaluations are done based on hardware performance and functional verification times and how these scale to larger designs.

When using *High Level Synthesis* one should have a good idea of what hardware that is ideal for the given design in order to design high performance hardware. The synthesis process of generating *Register Transfer Level*-code from C or C++ is highly dependent on syntax, especially as designs grow larger. This could be satisfied by having a good balance of pre-defined libraries and design specific code and keeping native C data types for high functional verification speed.

There are different ways of designing using *High Level Synthesis* this thesis aims to explore these and highlight their pros and cons. Thus providing guidelines and ideas for how to work with *High Level Synthesis* in different situations.

Keywords: **HLS, HDL, VHDL, C, C++, Vivado, FPGA, Xilinx**

Popular Science Summary

Newer digital circuit technologies are rapidly arising which allow for higher clock frequency, better energy efficiency and a higher number of transistors in a given area. All these enhancements opens up for more complex designs. This will in turn increase the development, simulation and verification time taken by the designers. The marketing window for a product is usually short and products needs to be ready for marketing within that window. Otherwise it might lead to huge economical set backs for the developers of the product.
Initially, digital circuit design took place at transistor level, which meant that designers had to place every single transistor which is a time consuming process. As the designs became more complex the development time increased a lot. A hardware description language called VHDL, at first used for document the behaviour of ASIC's (Application Specific Integrated Circuit) became very popular for also describing hardware. This reduced the development time a lot since every line of VHDL corresponds to several transistors. Today HDL coding languages like VHDL and SystemVerilog are the contentious industry standard and the preferred way of designing hardware. But as the designs have gotten even more complex and demanding large bases of VHDL code, techniques to design hardware at a higher abstraction levels have emerged. This way of designing hardware has been around for some years and has been met with some scepticism. So we might be facing a another shift in industry standard if it turns out to give the same benefits as VHDL did back then.
HLS (High Level Synthesis) is a way which lets the development take place at a high level language instead of a low level one. This will speed up development time since functions does not have to be implemented from scratch and also because it requires little knowledge of the hardware compared to HDL coding.
HLS has been around since 1994 but was not seen as matured enough for producing effective hardware in terms of resources and speed. Today there exists a "smörgåsbord" of different vendors which offers HLS, to mention some Cadence's Stratus, Xilinx's Vivado HLS, Mathworks HLS and Mentor's Catapult with more. This thesis will focus on Xilinx's Vivado HLS and evaluate if it can bee seen as mature enough for competing with traditional HDL coding and if there will be any time gain during verification.

Acknowledgements

Looking back at the last six months it has been an interesting learning experience in many different ways. We want to dedicate this section to the people who helped and supported us throughout it. First of all we wish to express our sincere appreciation of our supervisor Liang Liu from the department of Electrical and Information Technology who has supported and guided us throughout the entire process. The technical guidance and support from our company supervisor Kevin Cushon has been crucial for the thesis to succeed and for ensuring the use of forensic methods, and for that we express our deepest gratitude. We also want to thank Sacki Agelis for the administrative- and general support that has been crucial for the process and completion of this thesis. We also want to express our thanks to Ericsson and the department of Electrical and Information Technology at LTH for the opportunity to execute on this thesis. At last we want to express our sincere appreciation to our families and friends who has supported us.

Acronyms

ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
CCY	Clock Cycle
CFG	Control Flow Graph
CLK	Clock
CROM	Cosine ROM
CPU	Central Processing Unit
DSP	Digital Signal Processing
FF	Flip-Flop
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FT	Frequency Translator
HDL	Hardware Description Language
Hz	Hertz
HLS	High Level Synthesis
I/O	Inputs and Outputs
X_i	$X_{imaginary}$
IP	Intellectual Property
LUT	Look-Up Table
MSB	Most Significant Bit
MUX	Multiplexer
N/A	Not Applicable/Available, No Answer
NCO	Numerically Controlled Oscillator
PWM	Power Meter
X_r	X_{real}
RAM	Random Access Memory
RMS	Root Mean Square
ROM	Read Only Memory
RTL	Register Transfer Level
SROM	Sine ROM
UVM	Universal Verification Methodology
VHDL (VHSIC-HDL)	Very High Speed Integrated Circuit HDL
VIO	Virtual Input Output
XOR	eXclusive OR

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Automation of industries could be argued, together with IT, as one of the biggest drivers of the modern economy. Automation could simply be put as the aim to complete a given task with as little human interaction as possible. Achieving this has several benefits' perhaps the most sought after would be to free up human attention and effort from more mundane tasks to work on innovation and progress. The word automation could refer to many different areas and implementations. It could refer to anything as simple as software program performing a set sequence of tasks on a computers file system to advanced mechatronic implementations in modern manufacturing processes.

The potential gains of automation are large but while there are many gains of automation it too has its downsides. The lack of contextual understanding that machines or tools, posses could affect the result of the task it is to perform in a negative way. Where the lacking of contextual awareness yield a sub-optimal result the automation process require more sophistication. Generally where understanding why something is performed would increase the quality of the result by adapting to the specific circumstances of the specific task at hand. In such a case the progress of automation could be staggered. To surpass this problem most commonly the context of the problem at hand needs to be integrated into the automation process. Regarding simple and isolated tasks this is possible. As for more complex tasks integrating all possible outcomes and context becomes almost impossible which would yield in worse performance for some situations.
The gain in execution speed and versus the loss of contextual awareness is perhaps the most common dilemma one would face when when deciding if automation is the right course of action. The maturity of the automation process do often weigh heavy in these cases. Introducing a new way of working should yield in a more efficient way of working then what it previously was.

Automation affects almost all industries including digital hardware design. The most common way to design digital hardware would be with *Register Transfer Level*- (RTL) coding, that is describing digital hardware through code on a cycle- and bit accurate level. This was introduced with HDL-Synthesis in the mid 1980s [5]. HDL-Synthesis [5] tools aimed to take an RTL design and transfer it to a gate-level ASIC netlist consisting of logical gates. In doing so it automated the process of digital hardware design from the placement of logical gates and transistors to designing it in RTL-code.

The next step of abstraction for digital hardware design could be *High Level Synthesis*, or *HLS*. Which is a way of designing hardware by using the same coding syntax that is normally used for software design. HLS is an attempt to automate parts of the hardware design process by increasing the abstraction level from the traditional ways of designing hardware using RTL to designing hardware by describing its functionality in classical software syntax.

This thesis is aimed to be an analysis of *HLS*. The analysis will be done using *Vivado HLS 2018.3*. A development environment from Xilinx which uses C-, C++- or System C syntax to design hardware. The hardware results presented in this thesis is not guaranteed to be repeatable using any other tool or target hardware. But the methodologies used when working with *Vivado HLS* could be applicable when working with other HLS vendor tools. This should be achieved by giving an insight into working with HLS, what the benefits and drawbacks there are and how they can be leveraged to fit into different situations.

HLS has the potential to bridge the knowledge gap to hardware design, pushing design up on a system level requiring only software, or less hardware, knowledge. To do this the tool needs to prove sufficient performance that it can, in a trusted manner, produce hardware as good, or better, than with RTL. If this is not the case the tool needs to provide additional benefits, not existing with RTL, that can make up for the loss of performance. By analysing the state of Vivado HLS this thesis hopes to show the state of the tool but also HLS in general.

The analysis will be conducted by creating a digital hardware design in RTL and then the design should then be recreated using HLS. The HLS designs will be divided into two designs. One design should mimic the RTL-codes functionality and behaviour by coding as close to bit and cycle accurate as possible using C++. This design aims to recreate the RTL design but using the HLS tool. The other HLS design aims to maximize abstraction by using provided libraries with a more software approach to designing.

Designing with HLS can produce hardware with results much resembling those generated using RTL-coding. If the goal is to design high performance hardware in terms of area and timing, the logic functionality and hardware should be determined before the design is done in HLS. The C/C++ syntax should then be written so that the functionality resembles the goal design. This pushes the tool to generate hardware using the same logic and timing as to goal design. HLS could

also generate hardware using normal software syntax with the right settings. Designing using normal software syntax in general produces digital hardware that consumes more logic with worse timing results. HLS is sensitive to syntax and using pre-defined libraries makes it easier for the tool to synthesize the hardware but it can also decrease the performance when simulating in C.

It should be noted that qualities of designs is heavily dependent on the designer and the experience they posses. Before this thesis neither of the authors has done any previous work using HLS and their experience with RTL is restricted to academics only.

1.1 Thesis Structure

This thesis aims to compare the results produced as well as the workflow of Vivado to that of RTL when designing digital hardware aimed for FPGAs. It should also provide guidelines for people designing in HLS with an analysis of how the tool performs in different scenarios. Even if the design and practical work is done using Vivado HLS methodologies and workflow practices used throughout this thesis should be applicable, to some extent, if one should choose to design in HLS using different tools.

This thesis will be structured into six different parts, excluding the introduction.

1. Background - This chapter aims to explain why and what that was conducted throughout this thesis. It will also cover some basic digital hardware concepts the reader should be familiar with to understand the results and analysis. It will also give a brief introduction to RTL design concepts and how it was conducted throughout this thesis.

2. Xilinx Vivado HLS - This chapter should serve as an general introduction to HLS, more specifically to Vivado HLS. The aim is to get the reader familiarized with HLS design concepts and also how the differ as well as connects to digital design using RTL.

3. Case Study - Is about the hardware components, or designs, used throughout this thesis. Their specifications and an overview of their functionality will be presented as well as how they are intended to be used to analyse and test the Vivado HLS tool.

4. Results - In this chapter the quantifiable results are presented. It presents graphs and tables of the different designs implementations as well as the results of the design process it self. It aims to only present quantifiable and measurable results for the reader to extract.

5. Analysis - Aims to present the reader with a more in depth analysis of the results. Why the designs ended up the way they did and how they differ from each other. It will also bring up problems and hurdles one could face when designing hardware using HLS and how these could be prevented or solved.

6. Conclusions - Should conclude the thesis results and analysis with a concise description of HLS and its current state. In this chapter the authors opinions and experiences will be presented. These are not to be taken as facts but the opinions of two master students using HLS for digital hardware design for the first time.

The thesis is aimed at Vivado HLS and how it can be used for hardware design with measurable results as the main resource of analysis. Throughout the thesis the impact HLS might have on workflow methodologies in different parts of digital hardware design will be covered. Even so it is important as a reader to keep in mind that this is not an evaluation of methodologies used when designing hardware but the hardware produced as a result. The methodologies are brought up because they present one of the differences, and potential gains, when designing digital hardware in HLS compared to RTL.

Chapter 2
Background

To follow along in this thesis a basic knowledge of hardware design, how it is done and what's the reason behind it, is recommended. Even so the following chapter will briefly cover the methods, concepts and theory used throughout this thesis. The content of this chapter should be kept in mind when reading the rest of this thesis, which will hopefully explain the reasoning and decisions made throughout this work.

2.1 Motivation

Hardware development is in many aspects costly and resource demanding. Due to modern tools requiring a lot of detail to create functionality the design aspects can take a lot of time, get large and slow down development. Developing using HLS has the potential to allow the focus to be shifted from a bit-wise design to a more modular design approach. Which has the potential to reduce the development time and speed up the time to market for hardware and IP:s if it can reach sufficient performance compared to the tools used today. Other than design speed, HLS has the potential to speed up verification by moving it from an RTL environment to system-C/C. This could speed up the verification process significantly since the entire RTL model does not have to be simulated in order to do a functional verification.

High Level Synthesis tools have several potential benefits that could speed up the design of hardware. They include:

- Higher abstraction level for easier design
- Faster design verification from higher level simulation

While High Level Synthesis has several potential benefits for both hardware design and verification, it also presents potential drawbacks. By increasing the abstraction level the designer gives up control of design aspects that modern HDL tools provide. This could potentially lead to different types of implementation problems where design architectures are not as efficiently, in terms of area and timing of the circuit, implemented as they could be had the designer been given more control.

To evaluate the performance of Vivado HLS this thesis aims to use three different approaches.

- Basic RTL design using VHDL
- C++ design using an bit exact coding approach
- C++ design using a "Ultra High Level" -style of coding

These design ways do represent three abstraction levels of coding, ranking from lowest to highest in the list. A higher level of abstraction would generally be seen as better from a design speed perspective, but with a lower level of customization and performance.

Exploring different abstraction levels when designing could provide important insight to how well the tool performs in different scenarios. Even though a higher abstraction level would make hardware design more accessible. An approach to hardware design using C/C++ while having a goal implementation in mind, in terms of hardware as well as functionality, could have potential benefits of efficient hardware- and DSP design together with the design speed potential of software.

Through out this thesis the quality and efficiency of the hardware produced by different design methods, using high level synthesis and VHDL, will be analysed to determine how well they work. For each level of abstraction the design will be evaluated on the following parameters.

- Area
- Speed
- Time for Synthesis
- Simulation Time

This should provide a better view of the current state of the tool and what trade-offs that is being made when used in comparison to traditional RTL design using Verilog or VHDL.

2.2 FPGA Fabric

To understand the measurement of area efficiency, which is a very important indicator of the quality of the generated hardware, it is important to understand the basics of how hardware is mapped onto the FPGA. The target FPGA circuit for this thesis is the Xilinx Zync Ultrascale ZU27DR [9]. The FPGA holds a set of standard building blocks and programmable interconnections to design different logic implementations. The three basic building blocks discussed throughout this thesis are:

- DSP48 - The DPS48 slice in the Ultrascale architecture is the main slice for multiplication. The DSP contains a multiplier, adders, a multi-purpose ALU, and pipeline registers. By default multiplications are mapped to the DPS-slice, if possible the multiplication can be combined with other basic arithmetic operations [1].
- Block RAM - RAM's with programmable width and depth. Block RAM usually used for storing large amount of data such as comprehensive look up tables and large registers, data which would only need to be partly updated or accessed. This ultrascale+ arcitecture feature 38 Mb of block RAM [3].
- CLB - Is the main resource to implement general-purpose sequential- and combinational circuits on and FPGA [2]. It holds several blocks of hardware functionalities such as *Look-up Tables*(LUT), *Shift Register Logic*(SRL), *Flip Flops*(FF) and *High Speed Carry Logic*(Carry8) to implement logic and arithmetic operations. All this logic is contained within one CLB '*slice*', which then has several interconnects to other CLB-slices as well as other parts of the fabric to create the FPGA-architecture.

These are the standard FPGA fabric components used to build up different hardware logic and will be used in order to measure the area efficiency of Vivado HLS. The number of logic blocks used in a design matters also for timing reasons. If the logic needs more LUT's or FF's it could effect the timings of the circuit. The reason why it could affect timing is that if the design does not fit in CLB some of it might have to be spread over several CLB's were communication delay for the signal to travel to the next CLB can affect timing in a negative manner. The routing of the FPGA fabric allows for easier communication within a block than between several of them because there can not always be an available CLB close and a longer communication route may therefore have to be taken. Inefficient usage of the block RAM- or DSP- fabric could be even worse than CLB mapping, since the placement of these are less frequent on the FPGA-board and there are a limited number of them. If something does not map to a RAM or DSP, it might have to be built from CLB's leading to a increase in utilization for RAM case. In the DSP case it will mainly be the timing that will be affected, since DSP are rather complex and developed to be efficient at their capabilities it will require a high number of CLB's to perform this, see [1] and [16].

Different FPGA manufacturers has different fabric architectures and building blocks. Thus the same design might have different utilization depending on the FPGA provider. During this thesis Xilinx Vivado HLS is used to program a Xilinx

FPGA. This could cause a biased view and different result might be obtained if using FPGA-boards from different vendors, due to inefficient mapping and other mismatches.

It is important to understand these basic FPGA architecture blocks to understand the analysis of the area efficiency. Because of the structure of filed programmable gate arrays the number of blocks, or slices, used to implement the hardware functionality will determine how area efficient the final design is. The number of slices used is also affected on how the hardware algorithm is optimized and implemented but these are kept constant as much as possible through out this thesis.

2.3 Abstraction Levels and Design Hierarchies

This section should briefly clarify some terms used throughout this thesis. It is mainly to those how are not as familiar with digital hardware design and the different terms one usually uses to describe components of a design. The section also covers briefly what *abstraction levels* are referring to which is used frequently throughout this thesis to describe the design methods.

2.3.1 Design Hierarchies

The digital designs used in this project can be divided into two parts, *top-module* or *system level* and *sub-module* or *component*. An example of these can be seen in figure 2.1 where the left is a component usually performing a function or arithmetic operation with a set of I/Os. The right part of figure 2.1 is the system level or the top-module used to describe how different components are connected as well as the top level inputs and outputs.

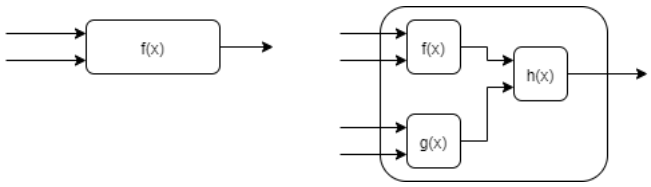

Figure 2.1: Design hierarchy levels used in this thesis component (left) and system (right).

It should be noted that functionality can be divided into several sub-modules, or components. How this division is done is decided by the designer.

2.3.2 Abstraction Levels

Abstraction levels are used to describe the different ways of designing hardware that was used throughout this thesis. Where the lower abstraction levels are designs where the functionality and logical hardware are more meticulously expressed. Higher abstraction levels indicates a reduction of the level of detail required, leaving the left out details to be performed by the tool. An example of the abstraction levels can be seen in appendix B.

To compare the HLS implementations with the RTL implementations three abstraction levels will be used. *VHDL* is the abstraction level closest to hardware. *VHDL* is design on a logic level and after HDL-synthesis it is converted to a netlist. The netlist is mapped onto physical hardware units which will be used to measure the utilization efficiency of HLS. The *VHDL* model will be used as the baseline performance measurement as it is a form of RTL design which is one of the most common ways to design hardware today.

The HLS implementations will be divided into two abstraction levels, *Bit-Exact* and *Ultra High Level*. The goal of the *bit-exact* model is recreate the hardware described in this chapter and to recreate or surpass, if possible, the VHDL design using HLS. Using syntax that both in timing and functionality is intended to force the HLS-synthesis to create RTL that function and maps to hardware as the VHDL design does. This should be achieved by coding as close to clock- and bit accurate as possible.

The Ultra High Level model goal is to demonstrate the performance of a design using higher abstraction level style of coding. In this case higher abstraction level is achieved by using libraries, syntax and arithmetic operations as one would in classic software design. It should be noted that the syntax is altered in a way so that it generates functional hardware and uses directives to improve HLS-synthesis result so that the design meets the initial requirements, see section 4.2.

2.4 Hardware Optimization and Structure

In digital hardware design there are several different optimization methods that could be utilized in order to increase the performance of a circuit. Throughout this thesis some of this optimizations will be implemented to meet specific hardware requirements. Therefore this section of the report aims to present some of these optimization methods, why they are used and how they affect the circuit they are applied to. A brief introduction to hardware hierarchy is also presented in this section. Even though the hierarchy is not a optimization method by itself it is important to understand the basic concept to fully grasp the results presented later in the report.

2.4.1 Pipelineing

Pipelineing is the insertion of registers at strategic places in a circuit to divide its functionality over as many clock cycles as the number of inserted registers [12]. As an example of this see figure 2.2 showing a made up datapath and figure 2.3 showing the same datapath with inserted registers. If in this dataflow graph representation of a fictitious circuit uses addition that requires two nanoseconds to complete and multiplication that requires four nanoseconds the circuit in figure 2.2 would take ten nanoseconds in total to complete. Which would require the datapath to have a clock cycle with at least a ten nanosecond period, if register setup and hold times are discarded. One way of pipelineing the datapath can be seen in figure 2.3 where, if once again register setup and hold times are discarded, a clock period of four nanoseconds would be possible. With the use of pipelineing the clock period were decreased which is one of the main advantages of this optimization method. Introducing pipeline stages in the datapath has one other advantage. It exploits hardware parallelism, meaning that the next operation in the datapath show in figure 2.3 can start after one clock cycle. Effectively allowing the hardware to have three active calculations in parallel which allows the circuit to produce one result every fourth nanosecond instead of every tenth. One should also note that in this case the total execution time increases from ten nanoseconds to twelve because the additions execute during one clock cycle even if it is finished before increasing the time compared to the non-pipeliened version of the datapath. This example summarises the main pros, clock speed and parallelism, and cons, increased hardware and possible longer execution time, of pipelineing a circuit.

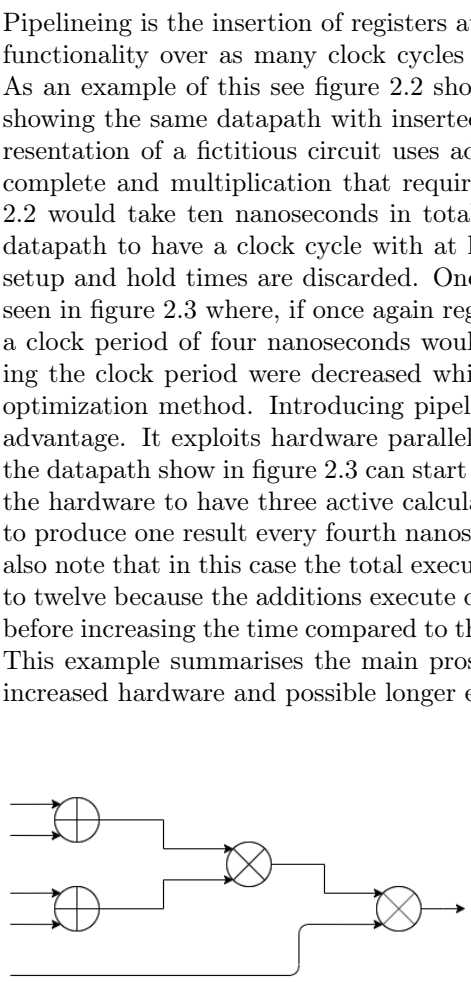

Figure 2.2: Dataflow graph example with no pipelineing.

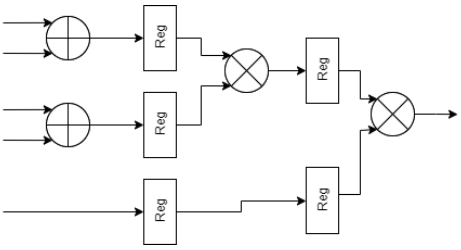

Figure 2.3: Dataflow graph example with pipelineing.

2.4.2 Loop Unrolling

Unrolling is an optimization method to reduce the number of clock cycles required to implement a iterative algorithm. Unrolling is done by performing several iterations each cycle, or if the loop is completely unrolled all iterations are performed in parallel [12]. An example of a loop that could be unrolled can be seen in figure 2.4. This loop could be implemented using a single adder which looks like figure 2.5 calculating one addition per cycle. By unrolling this loop the resulting hardware could look like that of figure 2.6

```
void top_function(int& A)
{
        for(int i = 1; i < 4; i++)
        {
                A += i;
        }
}
```

Figure 2.4: Example of software loop that could be unrolled.

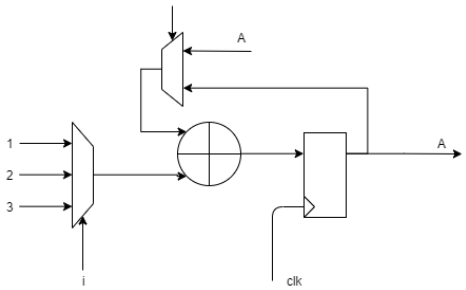

Figure 2.5: Hardware implementation of loop before unrolling.

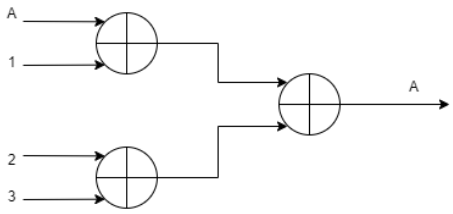

Figure 2.6: Hardware implementation of loop after unrolling.

The normal benefits of unrolling is a lower latency circuit that has to utilize more hardware to perform its calculations.

Chapter 3
Workflow and Xilinx Vivado HLS

This chapter will introduce Vivado HLS to the reader and give basic knowledge of the tool. Presenting briefly how the tool works and the process of generating RTL from C/C++. It also aims to present the workflow used in this thesis, both for the RTL- and HLS design and verification. Understanding the workflow and how it changes when designing in HLS compared to RTL is an important aspect to consider when evaluating the tools because the workflow can affect the design- and verification process and their speed.

3.1 Xilinx Vivado HLS

The tool used to design HLS throughout this thesis was Xilinx Vivado HLS and the tool used to do RTL design was Xilinx Vivado. The most significant difference between the two, that Vivado HLS is using C - or C++ code and Vivado using HDL code. It is assumed that the reader has some experience or previous knowledge of designing hardware using HDL and this thesis will not cover that in any detail.

3.1.1 HLS-Synthesis

High Level Synthesis is used to generate functional RTL from C/C++ source code. The synthesis itself is when the source code is interpreted and used to generate functional RTL. This process of generating RTL-code will throughout this thesis be referred to as *HLS-Synthesis* to differentiate it from HDL-Synthesis which is used to generate a netlist from RTL-code [5].

The synthesis process is the critical factor that enables HLS to be used for hardware design. The synthesis process could be described briefly in three parts. First the tool identifies key parts of the code. There are usually six attributes of the code that the tool has to identify, *Functions*, *Top Level I/O*, *Types*, *Loops*, *Arrays* and *Operators* [8]. These part all needs to be handled to correctly to synthesize the C/C++ code. Second the tool has to identify the control parts of the C-code. This is usually the start and end of a function call or internal loops. Using these sequences the tool creates a CFG to determine the order of operations. The tool then identifies the different operators used in the code and they are then placed within the different parts of the *Control Flow Graph* (CFG) [8] so that they are performed

in the correct order. Using this CFG with the different operators integrated the tool can then structure RTL to replicate the C functionality in hardware. Third the tool has to schedule which operations that should be performed on which clock cycle.

The structure of the CFG and its internal operators can be changed using directives, or C-pragmas [8]. This could tell the tool to implement different hardware optimization options, such as unrolling a loop (see section 2.4.2) for lower latency or pipelineing (see section 2.4.1) a function to increase the throughput of the hardware.

3.1.2 Customisation, Optimization and Constraints

Vivado HLS has several features and options to optimize and customize the hardware design implementation [8]. Note that all of them wont be covered in this thesis but this section aims to present the ones that one needs to know to understand the thesis results and analysis.

Arbitrary Precision

Arbitrary precision data types are provided from a Vivado HLS library. They are used in Vivado HLS for doing bit exact models and calculations. Assigning a variable to be of arbitrary precision type allows for more exact design beyond the standard C/C++ data types [14]. The arbitrary data types can be of *ab_fixed<WL, N>* or *ab_ufixed<WL, N>* where the data type can be set with word length, *WL*, and integer length, *N*, to create a fixed-point or unsigned fixed-point implementation. The arbitrary precision data types *ab_int<N>* and *ab_uint<N>* which will create a *N*-bit integer number [8].

Directives or Pragmas

By default the key attributes from the HLS-Synthesis are synthesised in standard ways. An array is for example by default synthesised as a BRAM [8]. How the tool should synthesise the source code could be modified with pragmas or directives. This allows the designer to perform different optimization methods by telling the tool specifically how a part of the code should be synthesised, or mapped to hardware. The directives mostly commonly used throughout this project are *Pipeline*, *Unroll*, *Array Mapping*. Pipeline has been explained in section 2.4.1 and are applied to functions to tell the tool that all the operations in that function should be pipelined. Unroll has been explained in 2.4.2 and is applied to loops to reduce the loop-iterations. Array mapping is used to change how an array is synthesised in HLS. This was often used to force the tool to synthesise arrays into registers instead of BRAM. For a full explanation of which directives there are and how they should applied we refer to the Vivado HLS user guide [8].

Constraints

A constraint when writing HLS is that memories has to be defined on compile time, i.e. you can't use memories with dynamic range, since it is not possible to synthesise a memory of unknown size. Therefore one cannot use pointers with offsets to access values. Normal C/C++ code would most likely contain a pointer to an array index then with the use of offset different values can be accessed. This is not the case in HLS where pointers are allowed but arrays has to be statically defined at compile time in some part of the code in order for the code to pass synthesis.

3.2 Design and Verification Workflow

To evaluate the HLS effect on design and verification it's important to have a good overview of the work flow when designing in HLS and in VHDL. In this chapter the design methodology used for this thesis is presented first and why it was chosen. Then a version of a VHDL workflow is presented, as a starting point, to be compared with the recommended HLS workflow. In the end of this chapter the differences will be compared and the possible impact these might have will be discussed.

3.2.1 Design

Hardware design-flow is usually done in one of two ways, using a top-down- or bottom-up design methodology. The bottom-up methodology is used because it was regarded as a preferable methodology when working with HLS design. In a bottom-up methodology the system functionality is divided up in smaller blocks which are designed separately first and then connected afterwards using a predefined interface.

There are especially two advantages of this design methodology. First the bottom-up methodology will allow the designer to evaluate the performance of each individual block by itself, thus knowing its limitation before it's implemented inside the entire system, or top model. This will make it easier for the designer to know what performance limitations that are existing before the top model implementation is done.
Secondly the saturation of *Moore's Law* [10] causes higher frequencies to cost more to achieve, which in turn is pushing the development towards more parallel, or multi-core, computing. This would be better utilized by running the synthesis of several design blocks in parallel instead of in one single core.

VHDL

The RTL workflow used in this thesis starts with a system description, overall and for each individual part. Using the system description a RTL-design and a RTL-testbench are designed using VHDL. The two designs are then used for behavioral simulation to verify that the design is logically correct. If the design does not

perform as expected the design has to be modified and the behavioral simulation re-run until the design works correctly. If the design functions as expected in the behavioral synthesis it goes through HDL-synthesis to generate a netlist. During synthesis the design is tested for timing errors and latches [15]. Should these test fail one has to redo the RTL-design once again. If the HDL-synthesis passes it generates a functional netlist that can be used in Place and Route where the design is mapped to a FPGA circuit. Should this fail one has to go back to the RTL-design step, if it passes a bit-stream to program the hardware onto the FPGA can be generated. An overview of the HDL workflow can be seen in figure 3.1.

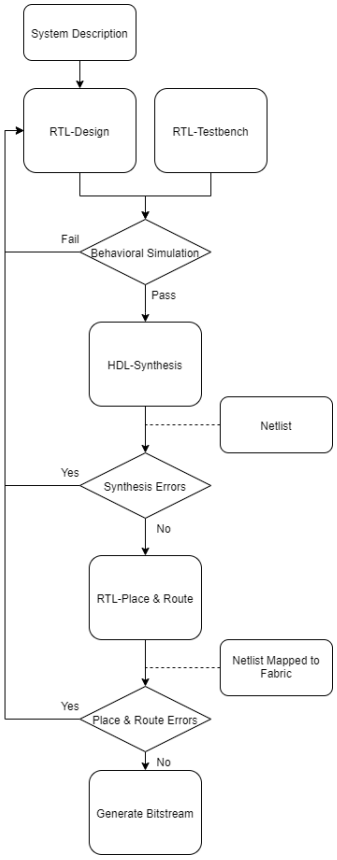

Figure 3.1: Description of the HDL workflow used in this thesis.

Vivado HLS

Designing in HLS starts with the system description of the design which describes the goal functionality. The system description is then used, together with existing software libraries, to create a system design and testbench in C or C++. Running the testbench as a main function the functionality of design is verified. Should it not pass changes have to be made to the design, if it passes it can go through HLS-synthesis. HLS-synthesis generates RTL-code and provides an estimate of the

hardware performance, timing and utilization. If the performance estimate are not good enough, directives [8] can be applied to improve them and change the hardware. In the worst case the C design has to be changed to improve the estimates. Once the estimates are good enough C/RTL-Cosimulation is performed, which verifies the RTL with the same testbench. Should the C/RTL-Cosimulation fail the directives or C design has to be altered to solve it. Once C/RTL-Cosimulation passes the design RTL can be exported to go through the flow of figure 3.1. The difference here is that the functional verification of the RTL is automated and done with a testbench written in C/C++. An overview of the HLS-workflow can be seen in figure 3.2.

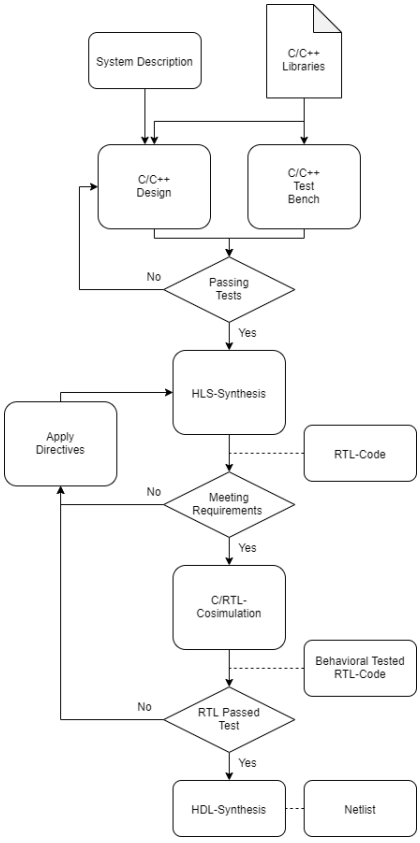

Figure 3.2: Description of the HLS workflow used in this thesis.

3.2.2 Verification

In this section of the report the verification process used throughout this thesis is presented. Verification could differ between different industries and even between different projects therefore the verification method, specification and process used are presented in this section. This should give some understanding of how verification could be performed and aims to improve the understanding of the verification results and analysis.

Verification is an integral part of hardware design. Before hardware can be sent into production or to be put in commercial use comprehensive testing has to be done in order to ensure design functionality and rated performance. The verification process is designed to emulate, even if limited, the process used in modern industry. It has to be noted that due to the time limiting factor of this thesis more advanced verification methods, which is a industry standard, for testing integrated circuits such as UVM [13] could not be used to test verification with HLS.

The metric used for evaluating verification is the simulation time for designs. These times will give a quantitative measurement on the time reduction when simulating functionality on a higher abstraction level in C/C++ versus RTL-simulation. This will be done by simulating for both smaller design on an IP-level as well as on a larger system level. The goal is to describe the verification processes and measure time differences to show how verification could change, for better or worse, with HLS.

To analyse the verification performance of HLS the main focus is the verification time i.e. how fast can the circuit be simulated and the logic functionality verified compared to traditional hardware design verification in HDL. This is of high relevance to industrial companies since larger hardware designs often contains large quantities of logic thus use up large areas on FPGA fabric and takes longer time to simulate.

Verification Flow

Verification for the traditional hardware design languages, VHDL or Verilog, are usually done on three levels which are listed below and shown in figure 3.3.

1. Component Level where each component or block is tested individually.

2. Verify System Level functionality with a mathematical model emulating the desired functionality of the implemented RTL design.

3. Advanced software from Xilinx, or similar vendors, which is used to toggle all switches and explore all branches in the given design.

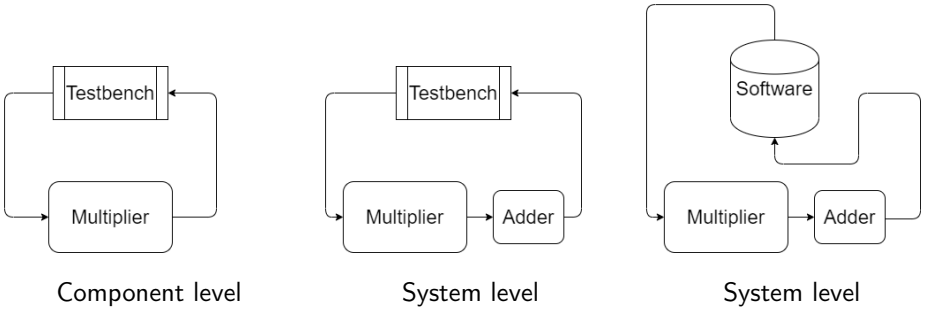

Figure 3.3: Verification levels

The verification in HLS is conducted with the same methodology as described in figure 3.3. The difference is that the testbench and design are written in C/C++. The top of figure 3.1 and 3.2 shows the behavioral verification, on each verification level, it is done in C/C++ for HLS and VHDL for the HDL design.

Verification Specification

The low level component verification is done using small test benches in VHDL for each component. These test benches uses static inputs to verify the basic functionality that is correct outputs for given inputs, latency of the circuit matches expectation and that enable-, reset- and valid signals behave as expected with correct timings.

Design in HLS will not use this verification step in the same way. In HLS design the code will be verified on a module level where each sub component input and output could be compared but this will not include any timing- or control signal verification since it's expected that the tool handles that when the code is synthesised. The timing- and control signal are mostly automated, or abstracted away, giving only limited customization options, mainly through directives.

The top design verification is done using both functional verification, for design specific cases, as well as constrained input testing, for regions more commonly used for frequency translators. The functionality tested for are:

- Fixed Operation Testing - where the component is feed with static inputs, either fixed or zero, to verify its behaviour for the edge cases.
- Functional Testing - Verifying the function of the component by feeding it a set of known inputs, and comparing it with a set of known outputs. This should be done for different variations of inputs to cover as many cases as possible.
- Hardware Functionality - Verify behavior at edge cases which causes overflow. As well as verification of hardware rounding due to the limited number of bits in the data-path.

- Random Vector Testing - Generate random inputs and run through a working software model. Use the same inputs and the generated outputs to verify the functionality of the hardware.

Verification Method

To do verification a model of the system has been implemented in python together with code for generating system inputs- and outputs then saving them to a file. The python model also contains a main part that is used to define various test cases which are used to verify the functionality of different parts of the system.

The RTL verification for the VHDL-design will be done using a VHDL test-bench reading an input file then computing the output using the RTL-logic then writing the logic-output to an output file. The output file will be compared with the one that was generated using the python script, see appendix **??**], for all the implemented designs, including HLS. Doing the HLS verification the same input- and output files, from the script, will be used to verify but the test-bench used is designed in C++. Because Xilinx HLS RTL co-simulation verifies the generated RTL-logic with the same stimuli that is used for the C-verification earlier in the HLS work flow, figure 3.2, the input- and output files can be used to verify both C functionality as well as HLS generated RTL.

Chapter 4
Case studies

In order for Vivado HLS to be evaluated it has to be used for designing hardware. This section describes the hardware design used throughout this thesis and explains why they were chosen. This chapter is also presenting the specification of the hardware that is to be used to have common design goals for each of the abstraction levels that are tested.

4.1 Modules

In this section the different modules and hierarchies used throughout this thesis are presented. The design is divided according to section 2.3.1 to divide the functionality into separate modules in order to ease the design process. The modules where chosen because they contains complex arithmetic operations in order to test the tool. They also uses all the main resources of an FPGA, DSP's, BRAM's which allows analysis of utilization choices. The two models can also be chained together in order to create a large design.

4.1.1 Frequency Translator Module

The frequency shift module is used to modify an input frequency by shifting it up or down. Frequency translators are commonly used in communication applications to shift signals to other frequency bands. Its basic functionality can be described with a block-diagram, see figure 4.1.

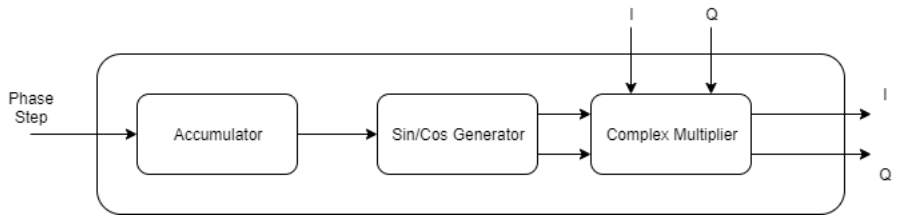

Figure 4.1: Block-Diagram for an overview of the frequency translator.

The sine and cosine generator uses one LUT for cos angles and one for sin angles respectively. To clarify the LUTs and their outputs will be referenced to as *SROM*, for the sine values, and *CROM*, for the cosine values, in the upcoming sections.

Accumulator

The accumulator and the cosine and sine generator together works as a digital oscillator. It will rotate its phase with a static step size (see figure 4.2) depending on the phase resolution in the sine- and cosine generator ROMs address-space. Throughout this thesis they have been separated to ease the analysis of each hardware implementation.

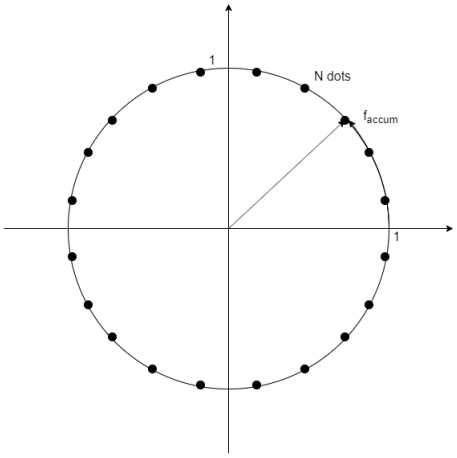

Figure 4.2: Schematic picture of the accumulators function.

The phase step size can not be smaller than the highest possible phase resolution, which will be the same as an accumulation of one, and has to be a multiple of the resolution in the *SROM* and *CROM*. The frequency generated by the accumulator can be described with equation 4.1

$$f_{accum} = \frac{f_{clk} \cdot N}{(2^{pwl} \cdot 2^{awl-pwl})} \tag{4.1}$$

where f_{accum} is generated frequency, f_{clk} is the on chip clock frequency, pwl is the number of bits used for the address space in the *SROM* and *CROM* respectively and awl is the accumulator word-length. N is the integer step size with which the accumulator is increasing. The *SROM* and *CROM* size has to be multiplied with $2^{awl-pwl}$ in equation 4.1 because the memory represents one of eight segments for an entire 2π rotation and the bits which are not used to address the memories are used in the truth table 4.1. The circuit itself is a simple accumulator circuit which can be seen in figure 4.3.

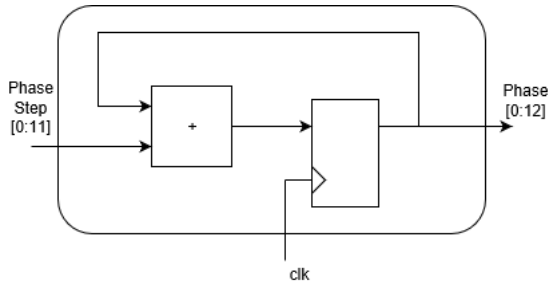

Figure 4.3: Description of the accumulator circuit used for calculating the next phase value.

Sine and Cosine Generation

The sine and cosine values are generated using a ROM, where function values of cosine and sine between zero and forty-five degrees are stored then modified according to table 4.1 to represent a full rotation. The whole sine and cosine generator can be seen in figure4.4. The 13-bit input phase will be split up so that the three MSB's will be used to determine what segment the angle is in, see table 4.1, and the lower 10-bits will be addressing the space according to table 4.1. The outputs from the *SROM* and *CROM* is then used to determine the sine and cosine value from the input phase, see table 4.1.

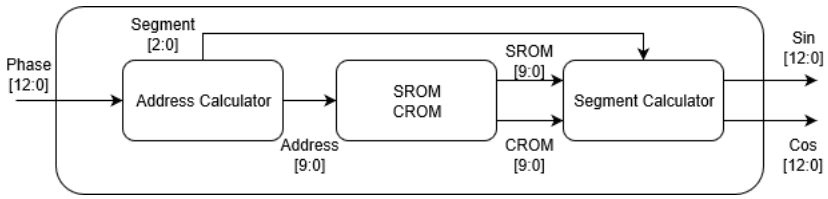

Figure 4.4: Block-Diagram overview of the cos-/sin Generator

Segment Address	ROM Address	COS	SIN
0	k	CROM	SROM
1	$k_0 - k$	SROM	CROM
2	k	-SROM	CROM
3	$k_0 - k$	-CROM	SROM
4	k	CROM	-SROM
5	$k_0 - k$	-SROM	-CROM
6	k	SROM	-CROM
7	$k_0 - k$	CROM	-SROM

Table 4.1: Truth table for address and value representation depending on the segment address.

The cosine and sine generator together with the accumulator is often refereed to as an *Numerically-Controlled oscillator* or *NCO*. In this report the accumulator and trigonometric calculations are separated into different components to make it easier to analyse the different parts of the frequency translator.

Complex Multiplication

The complex multiplication is used to mix the I and Q values from the sine and cosine generator with a fixed complex number $z_i = Q_i + jI_i$. The complex multiplication uses the three multiplication algorithm [11], thus it needs three DSP-blocks on the FPGA-fabric to be represented. The complex multiplier is designed using the Xilinx template in order to optimize the mapping to the FPGA-fabric, thus giving the best possible benchmark to be compared with the HLS. The real and imaginary calculations can be seen in equations 4.2 and 4.3.

$$I = a_r(b_r + b_i) - b_i(a_r + a_i) \tag{4.2}$$

$$Q = a_r(b_r + b_i) + b_r(a_i - a_r) \tag{4.3}$$

There are other ways of doing complex multiplication in hardware but it requires more multiplications and thus more DSP-units to keep the throughput to one.

4.1.2 Power Meter

The other module that were implemented is a power meter. A power meter calculates the power of the I and Q signals.
The power is calculated by squaring I and Q and adding them up according to the formula 4.4, this gives the instantaneous power. The average power is then calculated by shifting in the instantaneous power values into a shift register which holds the eight previously calculated instantaneous power values. A rolling average is then calculated by summing up all of the eight entries and dividing them by eight to get the average, the division is in this case a three step right bit-shift since eight is a power of two.
A schematic picture of the module can be seen in figure 4.5.

$$P_{Peak} = I^2 + Q^2 \tag{4.4}$$

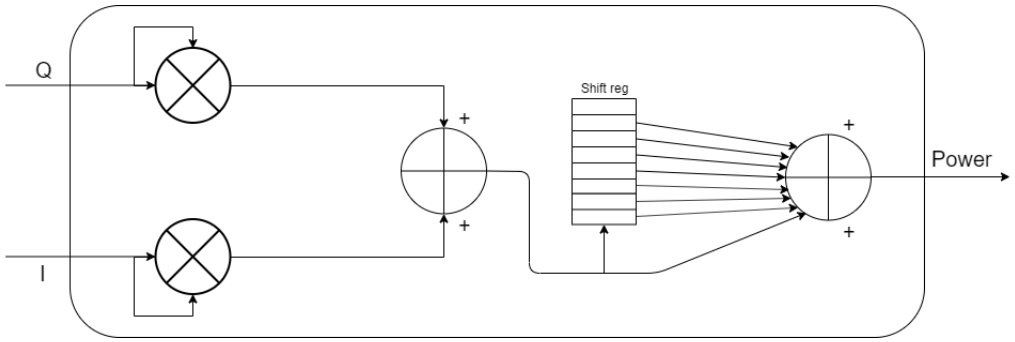

Figure 4.5: Block-Diagram overview of the power meter

4.1.3 Top-Module

The top-module which ties the power meter together with the frequency translator is show in figure 4.6. The "Valid_FREQ" indicates when valid data from the frequency translator can be obtained at "I_out" and "Q_out", whereas the "Valid_PWR" indicates when valid data from the power meter can be obtained at "PWR_out".

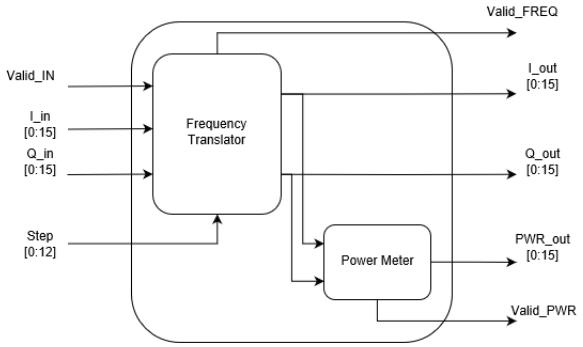

Figure 4.6: Block-Diagram overview of the top module

Note the top-model is used to describe an implementation that is connecting the frequency translator and power meter together. The purpose of the top-model design is to be used in the large system simulation experiment conducted to test simulation- and synthesis times of HLS when using large systems.

4.1.4 Large System

The large system is implemented by connecting n modules from figure 4.6 in series where I_out and Q_out from each model are connected to the next I_in and Q_in. The first I_in and Q_in as well as n inputs of *step* are used in the large system as inputs. The I_out and Q_out from the last component as well as n outputs of PWR_out are used as the large system outputs. This allows for

separate reading of the signal power after each component. Figure 4.7 shows the design that is to be used for testing larger simulations using Vivado HLS. The *Comp. Block* are the module show in figure 4.6. The controller is supposed to set all step sizes as well as the first real- and imaginary input and to read all power calculations done in the design.

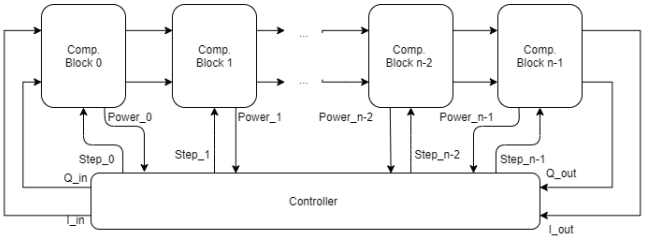

Figure 4.7: The large system design used to test long simulations.

The actual design done in HLS is the *Comp. Block* show in figure 4.7 connected in series with n step inputs, n power outputs as well as the real- and imaginary inputs and outputs. The *controller*-unit is a behavioral test bench designed in C++.

4.2 General Design Specifications

All the design should keep consistent goals so that each of the designs can be compared in a fair manner. This should make it so that the hardware keep approximately the same performance, if it is possible, and the difference will be mainly hardware utilization and design aspects. The design have the following hardware specification, see 4.2.

Clock Frequency	400 MHz
Throughput	1 cycles/output
Goal Latency Freq. Translator	6 cycles
Goal Latency Power Meter	1 cycle
Data-path Specification	
16-bit	fixed point signed with 6 integer-, 10 decimal-bits
Overflow	saturate
Rounding	truncate
Address-path Specification	
13-bit	unsigned integer
Overflow	wrap around
Rounding	not needed

Table 4.2: Design specification

The address width for the frequency translator is 13-bits to represent the phase which was chosen so that the cosine and sine memories would have a 10-bit address space which would match the memory modules on the Xilinx FPGA-fabric [3]. The full 13-bits of the phase will not be used for memory addressing, it can be read more about in this section 4.1.1.

Chapter 5
Results

Throughout this chapter the results of the case studies are presented using graphs and tables with brief descriptions of their content. The results will be divided into four main parts first comparing HLS- and HDL-synthesis then the simulation and synthesis times. The third section will cover the timing results from the case studies and the forth section will cover the hardware utilization and how the implementations mapped onto hardware. This chapter only aims to present the result of the different measurements that were done in this thesis. The discussion and analysis of the tool are presented in chapter 6. All designs were done targeting the FPGA circuit ZU27DR FPGA [7]. The results were generated by a RTL simulator from Xilinx using the Vivado design environment and the libraries for the ZU27DR FPGA.

The simulation-, synthesis- and place and route times are all measured in CPU time. This was deemed to be a good way of comparing the different design moments against each other since elapsed time is not as repeatable and would depend on what system the design is done on.

All the functional verification have been done using the same input and output samples generated from a Python reference model. The input stimuli has been randomly generated from normal distributions. The input frequency used was generated from a normal distribution using normal value $\mu = 5MHz$ and standard deviation $\sigma = 100KHz$. Where as the input step, generating the internal frequency, for the frequency translator used a normal value $\mu = 50$ and standard deviation $\sigma = 10$ where, from equation 4.1, each step corresponds to an internal frequency step of approximately $49KHz$.

When presenting the results there are differences between the VHDL design and the HLS designs. The choice was made to not design RTL testbenches for the HLS generated design but to use the C/RTL-Cosimulation provided by Xilinx Vivado HLS to verify the behaviour of the generated RTL.

5.1 HLS- versus HDL-Synthesis

It is important to emphasize on the difference between HLS- and HDL synthesis. Where HLS-Synthesis is used to transform C or C++ to RTL-code and HDL-Synthesis is used to transform RTL-code into a netlist which can be mapped onto hardware. Vivado HLS provides estimates of hardware performance after HLS-Synthesis. These estimates usually contains information if the design is going to meet timing and the logic consumption of the circuit. In table 5.1 the estimates of frequency translator and power meter implementations can be seen.

	Bit-exact		**Ultra High Level**	
	HLS-synthesis	**HDL-synthesis**	**HLS-synthesis**	**HDL-synthesis**
	Frequency translator			
Clock Period	2.5 ns	2.5 ns	2.5 ns	2.5 ns
Slack	0.311 ns	0.576 ns	0.311 ns	0.615 ns
LUT's	661	128	746	188
FF's	399	165	915	199
	Power meter			
Clock Period	2.5 ns	2.5 ns	2.5 ns	2.5 ns
Slack	0.211 ns	0.445 ns	0.148 ns	0.657 ns
LUT's	420	179	437	234
FF's	156	156	163	163

Table 5.1: Result-matrix of the comparison between the HLS-synthesis and the HDL-synthesis.

The estimation are, and was throughout, underestimating the performance of the hardware that the generated RTL was going to implement. In table 5.1 the hardware estimates shows an overhead of 200% for some cases. This is mainly due to HLS-Synthesis not making any hardware optimizations such as removing signals bound to zero or, implementing division or multiplication with a power of two with shifts.

HLS-Synthesis did also provide timing information which for the single component implementations ensured that timing was going to be met after HDL-Synthesis as well. This was not the case for the large system which passed HLS-Synthesis timings but not HDL-Synthesis which contained to long physical paths and limited the large system design.

5.2 Synthesis- and Simulation Times

In this section the synthesis- and simulation times will be presented for each of the case studies. The section will cover the performance for each case study and the abstraction levels related to it. Each of the simulations are done with one million input samples for each respective abstraction level and component.

The C/RTL-Cosimulation times, presented in tables 5.2 and 5.3, shows the CPU time measured from the C-testbench. The CPU time to run the actual C/RTL-Cosimulation after it has synthesised is down to ms.

5.2.1 Frequency Translator - Synthesis- and Simulation Times

The C-simulation times for the frequency translator is approximately eleven times faster for the bit-exact implementation, see table 5.2. This is mainly due to the use of arbitrary precision datatype library used so that datatypes are synthesised into fixed-point. The bit-exact C simulation is also faster than the behavioral simulation.

	VHDL RTL	**HLS Bit-Exact**	**HLS Ultra High Level**
C simulation	N/A	00:00:03	00:00:33
RTL CO-Simulation	N/A	00:01:16	00:01:11
Behavioral RTL-simulation	00:00:25	N/A	N/A
HLS-synthesis	N/A	00:00:30	00:01:19
HDL-Synthesis	00:00:52	00:00:47	00:00:46
Place and route	00:01:27	00:02:30	00:02:43

Table 5.2: Result-matrix of the comparison between CPU time of the simulation and build times of the frequency translator, format hh:mm:ss.

5.2.2 Power Meter - Synthesis- and Simulation Times

The times presented in table 5.3 shows no noticeable differences in any of the synthesis- and simulation times for the power meter. The power meter implementation shows a reduction between functional simulation using C compared to behavioral simulation.

	VHDL RTL	**HLS Bit-Exact**	**HLS Ultra High Level**
C simulation	N/A	00:00:10	00:00:12
RTL CO-simulation	N/A	00:00:26	00:00:26
Behavioral RTL-simulation	00:00:23	N/A	N/A
HLS-synthesis	N/A	00:01:12	00:01:10
HDL-Synthesis	00:01:05	00:01:02	00:00:59
Place and route	00:01:12	00:01:49	00:01:40

Table 5.3: Result-matrix of the comparison between CPU time of the simulation and build times of the power meter

5.2.3 Large System - Synthesis- and Simulation Times

In table 5.4 the functional verification- and synthesis times of a large system is presented. The goal of these measurements are to represent the verification- and synthesis times that one could expect when designing a larger and more complex system that utilizes more hardware where *Number of devices* indicates the number of frequency translator and power meter pairs used.

The results for the HLS bit-exact C simulation (see table 5.4) time is taken from simulations which were using C data types. That is *double* for floating point calculations and *int* for the integer calculations. Should the same C-simulation be done with the Vivado HLS libraries for fixed point and integer it would increase the simulation time for the bit-exact model as for the ultra high level implementation. The ultra high level is forced to use the arbitrary precision libraries because the Vivado HLS libraries for complex multiplication and cosine and sine generator are using them. Thus the C-simulation times are higher for the ultra high level design than the bit-exact. The difference between behavioral simulation for RTL and C simulation (using native C data types) is grows larger as the design size increases. This can be seen in table 5.4 which has seven minutes and seventeen seconds compared to thirteen second difference in table 5.3.

	VHDL RTL	**HLS Bit-Exact**	**HLS Ultra High Level**
Number of devices	100	100	100
Number of inputs	1 000 000	1 000 000	1 000 000
C-Simulation	N/A	00:00:20	00:20:01
HLS-synthesis	N/A	00:09:24	00:13:30
Behavioral-RTL CO-simulation	00:07:37	N/A	N/A
HDL-Synthesis	00:07:13	00:08:42	00:05:43
Place and route	00:15:35	00:15:20	00:06:48

Table 5.4: Result-matrix of the comparison between CPU time of the simulation and build times of the big design

The HLS-synthesis time varied depending on which of the abstraction levels used. This is shown in figure 5.1 where the HLS-synthesis time is plotted against the number of components, or top-modules from section 4.1.3. The difference between the bit exact implementations are that the *control_none* uses pipeline directive in the top-function and singe data points as inputs and outputs and directive for no interface logic. The bit-exact *stream* uses the dataflow directive in the top-function with the Vivado library for streams as inputs and outputs and handshake directive for interface logic.

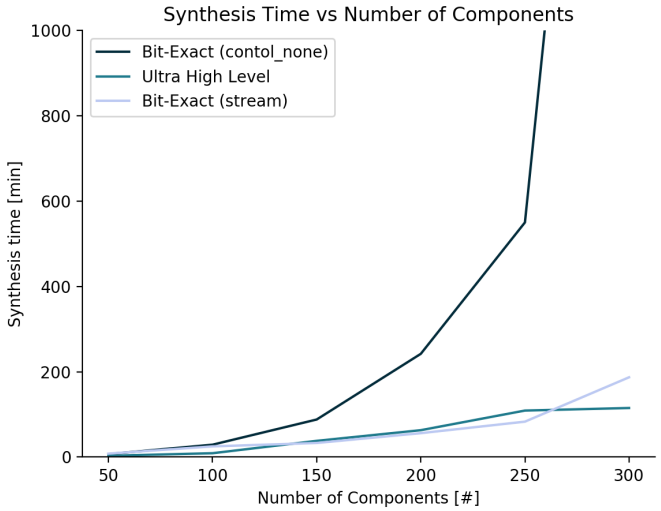

Figure 5.1: Plot of how the HLS-Synthesis time changes with the number of components.

5.3 Timing Results

The following section will present the timing results achieved for each of the case studies. The timing goals for the hardware can be found in table 4.2.

5.3.1 Frequency Translator

The frequency translator met the timing goals for the lower abstraction level, namely the VHDL and bit-exact implementations, see table 5.5. The ultra high level implementation did not meet the timing goals where a latency of ten clock cycles was achieved. The increase in latency was due to algorithmic differences in the cosine and sine calculations.

	VHDL RTL	**HLS Bit-Exact**	**HLS Ultra High Level**
Frequency [MHz]	400	400	400
Slack [ns]	0.340	0.576	0.615
Latency (CCY's)	6	6	10

Table 5.5: Result-matrix of the timing results for the frequency translator.

5.3.2 Power Meter

The power meter met the timing goals that were stated, see table 5.6. The bit-exact design had the smallest margin to meet the timing requirement of 400MHz.

	VHDL RTL	HLS Bit-Exact	HLS Ultra High Level
Frequency [MHz]	400	400	400
Slack [ns]	0.691	0.445	0.657
Latency (CCY's)	1	1	1

Table 5.6: Result-matrix of the timing results for the power meter.

5.3.3 Large System

The timing results from the large system can be seen in table 5.7 and 5.4. All designs meet the timing after place and route. The VHDL margin remained with a higher number of components where the HLS implementation failed timing. Resulting in a upper limit of 100 components for the large system after which timing would no longer pass for the HLS implementations.

	VHDL RTL	HLS Bit-Exact	HLS Ultra High Level
Frequency [MHz]	400	400	400
Slack [ns]	0.054	0.02	0.138
Latency (CCY's)	500	899	507

Table 5.7: Result-matrix of the timing results for the large system.

The latency of the VHDL- and ultra high level implementation where less than expected, see table 5.7. This is assumed to be because of the tool manage to optimize the design, since it is a chain of the same functionality.

5.4 Hardware Mapping and Area Consumption

In this section of the results the utilization of the FPGA fabric and the how the different abstraction levels mapped onto hardware will be presented. All of area usage is presented with the post-synthesis results to make a cross comparison between all implementations. The HDL-synthesis results are also all run with -*mode out_ of _ context* settings to ignore mapping to the FPGA I/O as they are not used in the case studies. Each sub-section will cover the area utilization and hardware mapping for each case study and its respective abstraction level. The results will be structured by case study, first the frequency translator results will be presented then the power meter and last the large system design results.

5.4.1 Frequency Translator - Hardware Mapping and Area Consumption

The hardware utilization of the designs are increasing as the abstraction level is increased but the overhead for the bit-exact design is not as large as for the ultra high level. The bit-exact implementation functions in a very similar way as the VHDL implementation thus the smaller overhead, which is mainly due to differences in control signals and implementation of quantization logic after the complex multiplication. The number of components used in the frequency translator as a whole can be seen in table 5.8.

Looking at table 5.8 the ultra high level implementation has more overhead than both the VHDL- and bit-exact implementation. Compared to the VHDL model the overhead is 71% for the LUT's, 75% for the FF's, three more DSP's and half of the BRAM's. In table 5.11 the number of LUT's are 36 more and the number of FF's are twelve less. While using three DSP compared to none for the VHDL model and only one BRAM not two. The increase in DSP units are most likely due to the trigonometric calculations being done with a version of Taylor expansion using a Xilinx library function instead of a LUT. The reason to the reduction in number of FF's are due to the fact that HLS-synthesis did not divide the cosine and sine generator into a clearly defined sub module and most likely some of the leaf cell FF's in table 5.11 belongs to the cosine and sine generator module.

	VHDL RTL	**HLS Bit-Exact**	**HLS Ultra High Level**
LUT's	110	128	188
FF's	114	165	199
DSP48's	3	3	6
BRAM's	2	2	1

Table 5.8: Result-matrix of the Number of components used for the different abstraction levels of the frequency translator.

VHDL Implementation of Frequency Translator - Area Consumption and Hardware Mapping

The **VHDL** implementation functionality mapping can be seen in figure 5.2. It is implemented according to the reference figure and aims to use as few components as possible. The step input of the frequency translator is connected directly to the accumulator then feedback through a set of registers.

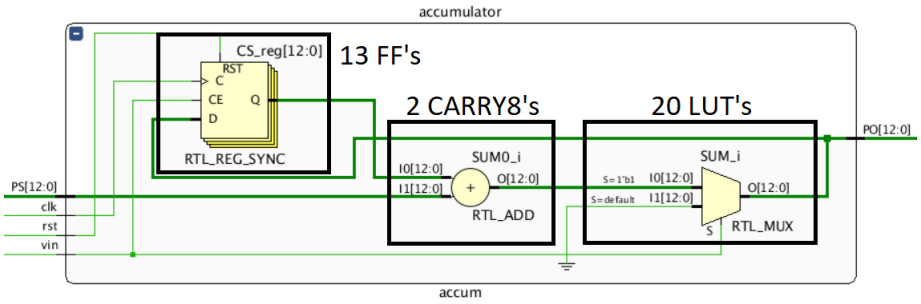

Figure 5.2: RTL Analysis of the VHDL Accumulator implementation.

The **VHDL** model implementation of the *cosine and sine generator* can be seen in figure 5.3, which is an hardware implementation much like the goal design shown in figure 4.4.

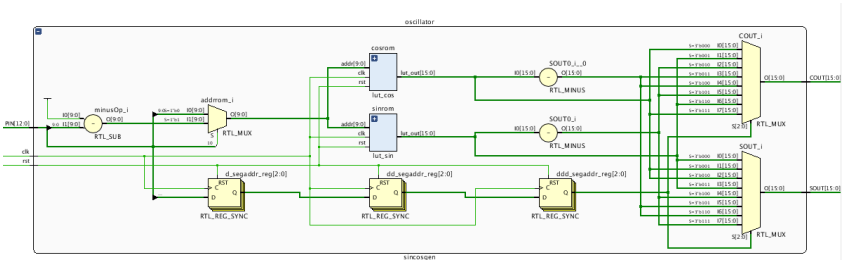

Figure 5.3: Functional RTL Analysis of the VHDL model for Cosine & Sine Calculations.

The complex multiplication was done using three DSP-slices structured so that the common factor was calculated in one DSP (see figure 5.4) and the two unique factors was calculated (see figure 5.5 and 5.6) separately in one DPS each. The structure of the three DSP-slices can be seen in figure 5.7. The **VHDL** model implements the same equation as described in the case study chapter, see equation 4.2 and 4.3.

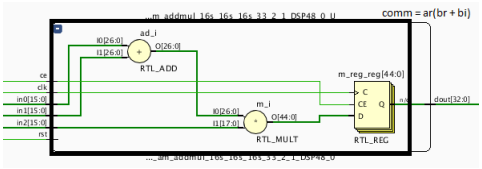

Figure 5.4: Functional RTL-Analysis of the common factor calculation.

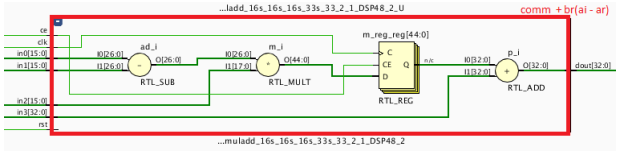

Figure 5.5: Functional RTL-Analysis of the complex multiplication for the imaginary output.

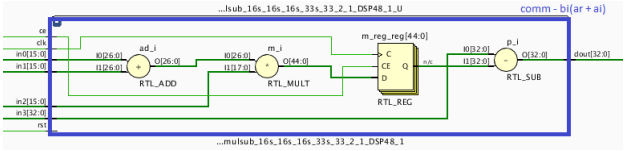

Figure 5.6: Functional RTL-Analysis of the complex multiplication for the real output.

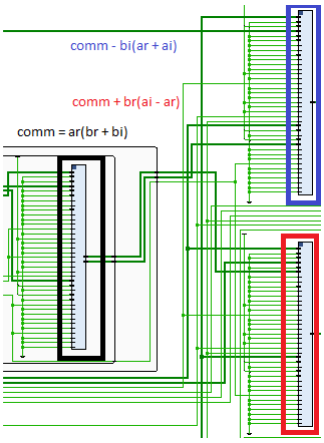

Figure 5.7: Three DSP48-Slice implementation representing the architecture used for all implementations.

A more detailed description of the logic components distribution used for the **VHDL** implementation of the frequency translator can be seen in table 5.9 and figure 5.8. The majority of the logic LUT's are used in the *cosine and sine generator* and all the BRAM's. The majority of the registers are placed within the *complex multiplication* as it is needs three clock cycles, while the *accumulator* and *cosine and sine generator* needs one and two cycles respectively. Leaf cells, in table 5.9, refers to logic that is not used in any sub-module of the system.

	Accumulator	Cosine and Sine Calc.	Complex Multiplier	Leaf Cells
LUT's	20	69	16	5
FF's	13	39	48	14
DSP	0	0	3	0
BRAM	0	2	0	0
CARRY8	2	0	0	0
SRL	0	0	0	0

Table 5.9: Detailed result-matrix for the VHDL Frequency translator showing the number of components.

In figure 5.8 the utilization of the **VHDL** implementation is displayed as a bar chart to visualize how the logic is distributed.

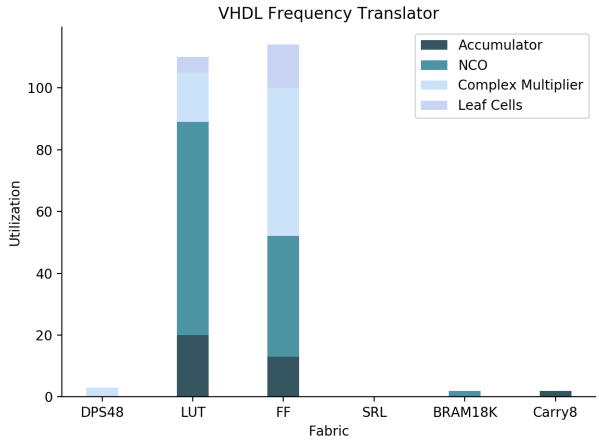

Figure 5.8: Bar chart showing the number of components for the VHDL implementation of the frequency translator.

Bit-Exact Implementation of Frequency Translator - Area Consumption and Hardware Mapping

The **bit exact** model implemented the accumulator in the expected way as the VHDL model. It uses 13 registers, 13 look-up tables and two carry8 units, see table 5.10, and functions as figure 5.9.

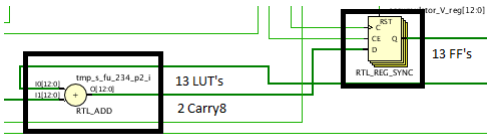

Figure 5.9: RTL Analysis of the BE Accumulator implementation.

The cosine and sine generator for the **bit-exact** implementation is much harder to analyse and to display any functional results from the algorithms. Exactly which function that maps to hardware is hard to determine. Because the HLS modules are cluttered with logic AND-gates and multiplexers they become hard to interpret. One can assume that the hardware has much in common with the VHDL implementation because the area consumption are much alike, see table 5.9 and 5.10. The **complex multiplication** is done in the same way for the **bit-exact** implementation as it was in the VHDL implementation, see figure 5.7.

In table 5.10 a more detailed distribution of the **bit-exact** frequency translator implementations utilization of the FPGA-fabric is presented. This can be compared to the number of components used in table 5.8. Leaf cells, in table 5.10, are harder to map, but the increase in leaf cells compared to the VHDL implementation (see table 5.9) comes from an increase in logic for control- and quantization

logic. The increase in FF's in the complex multiplier is due to less registers being absorbed into the DSP slices. This is assumed to be because circuits timing which are less relaxed because of the control signals in the bit-exact model require the registers to stay outside the DSP48-slice.

	Accumulator	Cos and Sin Calc.	Complex Multiplier	Leaf Cells
LUT's	13	86	1	38
FF's	13	46	64	42
DSP	0	0	3	0
BRAM	0	2	0	0
CARRY8	2	0	0	0
SRL	0	0	0	0

Table 5.10: Detailed result-matrix for the bit-exact Frequency translator showing the number of components.

In figure 5.10 the utilization of the **bit-exact** implementation is displayed as a bar chart to visualize how the logic is distributed.

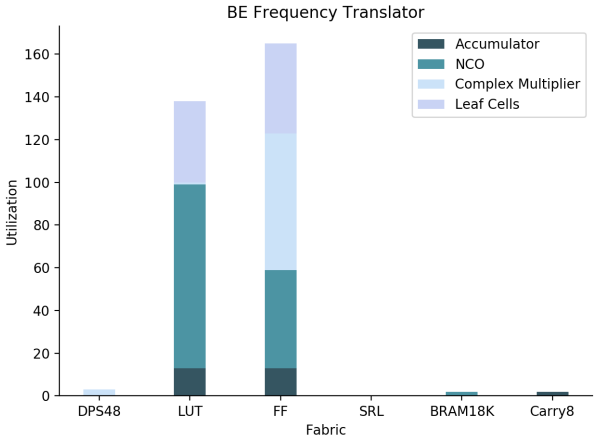

Figure 5.10: Bar chart showing the number of components for the bit-exact implementation of the frequency translator.

Ultra High Level Implementation of Frequency Translator - Area Consumption and Hardware Mapping

The **ultra high level** implementation of the accumulator is implemented in the same way as the VHDL- and bit-exact implementation. The only difference is the registers used at the bottom left in figure 5.11 which are used as input registers. This is assumed to be the reason the accumulator uses twice the number of LUT-, FF- and CARRY8 units, see table 5.11. The extra registers are assumed to be added from the library function used for cosine and sine calculation which includes the accumulator.

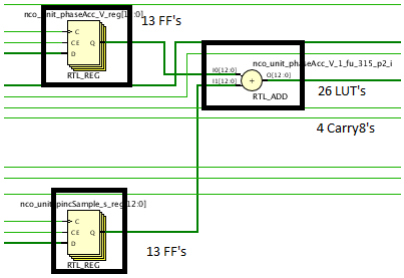

Figure 5.11: RTL Analysis of the Ultra High Level Accumulator implementation.

The cosine and sine generator was implemented using a CORDIC-like iterative algorithm. The exact hardware mapping is not included because it is too large to include in the report in a good way. The complex multiplication is done in the same way as was done in the VHDL implementation and the bit-exact implementation, see figure 5.7.

The **ultra high level** implementation of the frequency translator used the most logic of the the frequency translator implementations, see table 5.8. The more detailed logic distribution of the ultra high level (see table 5.11) shows that much of the increase in LUT's comes from the differences the cosine and sine calculations. The increase in FF's is mainly due to the longer latency of the circuit (see table 5.5) requiring more registers to keep it pipelined.

	Accumulator	Cos and Sin Calc.	Complex Multiplier	Leaf Cells
LUT's	26	105	0	46
FF's	26	27	0	146
DSP	0	3	3	0
BRAM	0	1	0	0
CARRY8	4	0	0	0
SRL	0	16	0	17

Table 5.11: Detailed Result-matrix for the ultra high level frequency translator showing the number of components.

The extra number of DSP48-slices (see table 5.11) are due to the difference in how cosine and sine are calculated when using the Vivado HLS library. These results were obtained using a 13-bit wordlength for the address and a 10-bit wordlength for the phase resolution.

Figure 5.12 visualise the area consumption of the **ultra high level design**.

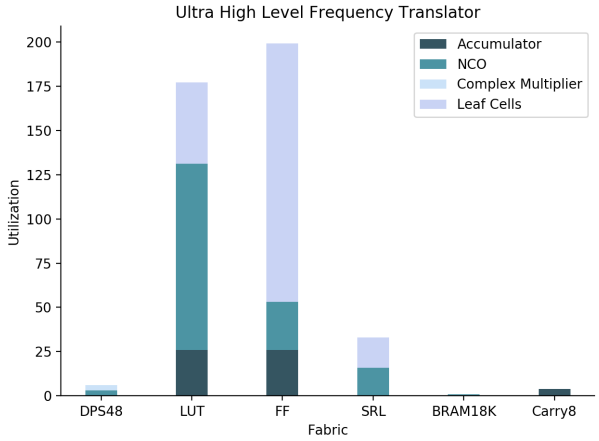

Figure 5.12: Bar chart showing the number of components for the ultra high level implementation of the frequency translator.

5.4.2 Power Meter - Area Consumption and Hardware Mapping

In table 5.12 the number of components used by each abstraction level implementation the power meter is presented. The most efficient implementation in terms of area is the VHDL implementation, followed by the bit-exact and the ultra high level has the least efficient implementation.

	VHDL RTL	**HLS Bit-Exact**	**HLS Ultra High Level**
LUT's	115	179	234
FF's	113	156	163
DSP48's	2	2	2
BRAM's	0	0	0

Table 5.12: Result-matrix of the area consumption of the power meter for all three abstraction levels.

Figure 5.13 shows the multiplier block used in both HLS designs, see figure A.2 and A.3 for where the blocks are used. This block consists as seen of a multiplier and register which are both absorbed into a DSP48.

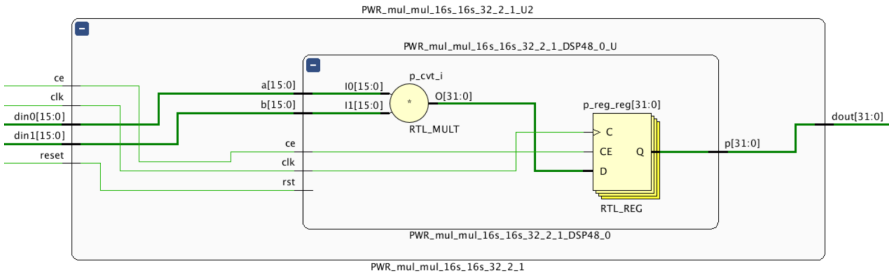

Figure 5.13: The common multiplier block used in both HLS designs

VHDL - Power Meter Area Consumption and Hardware Mapping

Here table 5.13, 5.14 and 5.15 reports the number of components used for the power meter post-synthesis for the VHDL, bit-exact and ultra high level design respectively. The schematics for the VHDL implementation of the power meter design can be seen in Appendix A, see figure A.1.

	Sum	**Square+add**	**shift-register**	**Adder tree**	**Control logic**	**Leaf Cells**
LUT's	115	16	0	67	32	0
FF's	113	0	112	0	1	0
DSP	2	2	0	0	0	0
BRAM	0	0	0	0	0	0
CARRY8	9	0	0	9	0	0

Table 5.13: Detailed result-matrix for the VHDL power meter showing the number of components.

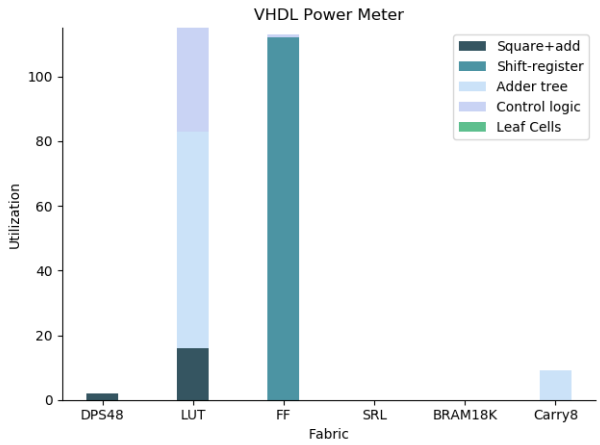

Figure 5.14: Bar chart showing the number of components for the VHDL implementation of the Power meter.

In table 5.13 and figure 5.14 one can see the number of components used for the power meter and in figure A.1 the corresponding schematic.

The two registers seen in figure A.1 after the multiplications which squares I and Q are absorbed into the DSP48, which also is the case for the following addition which is done in the DSP48 where Q is being squared. The overflow detection which is the equal block also done within the DSP48. One can also see in figure A.1 that the final division with the number of 8 entries in the shift-register is done with a 3 step right-shift which equals a division of 8.

To summarize the schematic seen in figure A.1, the control logic consists of a register delaying the "valid" signal one clock cycle for it to be in synchronization with the "powerout" and it also controls a LUT at the input which switches the inputs of I and Q to naught if they are not valid. The data path of I and Q then continues from the LUT to a DSP48 respectively to be squared, from their DSP's they are added and checked for overflow by checking if the most significant bits are equal to zero. The result of the most significant bits controls a LUT which saturates if overflow has occurred. The sum is then passed from the LUT to register "cshiftreg_reg[0]" and to the last addition where its added with the sum of the entries in shift register. The shift register consists of seven serialized registers which are starting from "cshiftreg_reg[0]" which output is coupled to an adder and the input of the next register "cshiftreg_reg[1]", the result of the adder goes to the next adder to be added with the output of the next register and so on.

Bit-Exact - Power Meter Area Consumption

A detailed distribution of the logic used for the bit-exact implementation can be seen in table 5.14. It can be seen that the utilization of the bit-exact design in table 5.14 and figure 5.15 is higher, the leaf cells here are also assumed to be used for control, quantization and saturation logic. It can be seen in the corresponding schematic A.2 that it has logic for detecting and control of overflow after every addition and also control logic for enabling registers and data paths, additional registers can also be seen.

For a summation of the schematic for the bit-exact in figure A.2, the control logic consists of a "ap_start" signal which starts the system by enabling registers. The control signal also features some latching capabilities. The bit-exact hardware mapping can be found in appendix A, see figure A.2.

	Sum	**Square+add**	**shift-register**	**Adder tree**	**Control logic**	**Leaf Cells**
LUT's	179	81	0	93	5	67
FF's	156	0	105	49	2	0
DSP	2	2	0	0	0	0
BRAM	0	0	0	0	0	0
CARRY8	19	6	0	13	0	10

Table 5.14: Detailed result-matrix for the Bit-exact HLS power meter showing the number of components.

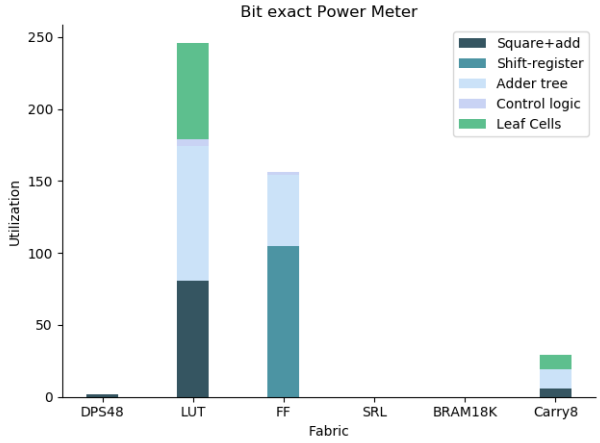

Figure 5.15: Bar chart showing the number of components for the Bit-exact HLS implementation of the Power meter.

The data path begins with I and Q getting squared the most significant bits are then checked for overflow by a series of XOR-, AND- and OR-gates which is implemented by a look up table which controls a multiplexer to saturate if overflow. The I and Q are then both added together and added one by one last at the adder tree. The shift register consists of serialized registers whose outputs are grouped ad summed up and checked for overflow by letting the most significant bit control a multiplexer which can saturate if overflow.

Ultra High Level - Power Meter Area Consumption and Hardware Mapping

The number of components used for the ultra high level design listed in table 5.15, which shows a more detailed distribution of the used logic, and figure 5.16 has even higher utilization due to higher bit-width to some additions, compare its corresponding schematic A.3 to the schematic of the bit-exact A.2. The hardware mapping of the ultra high level power meter implementation can be found in appendix A, see figure A.3.

	Sum	**Square+add**	**shift-register**	**Adder tree**	**Control logic**	**Leaf Cells**
LUT's	234	66	0	162	6	137
FF's	163	0	112	49	2	0
DSP	2	2	0	0	0	0
BRAM	0	0	0	0	0	0
CARRY8	25	4	0	21	0	18

Table 5.15: Detailed result-matrix for the ultra high level HLS power meter showing the number of components.

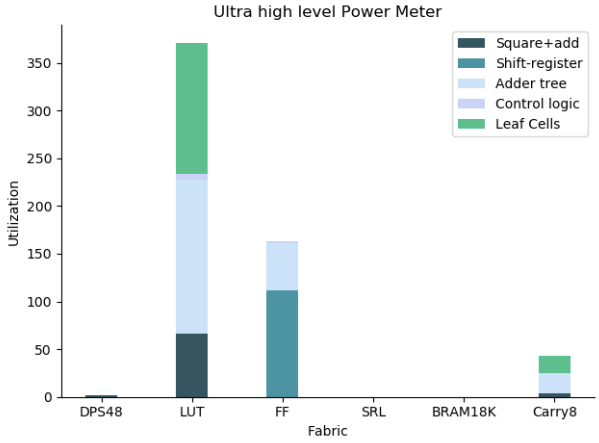

Figure 5.16: Bar chart showing the number of components for the ultra high level HLS implementation of the Power meter.

The control logic of the ultra high level design are almost the same as the bit exact with the only difference that the ultra high level has one additional LUT, see schematic in figure A.3. The beginning of I and Q data path are also the same until after they are added together and added last to the adder tree. The shift register for the ultra high level also consists of the same configuration of serialized registers, the difference here is that the bit width for the first additions in the adder tree are one bit more.

5.4.3 Large System Area Consumption

The large system were limited by the HLS implementations where neither of the abstraction level managed to meet the timing when using more components. The implementation of the large system synthesis reveled that the quantization method used can have a large impact on the HLS-synthesis result. Running HLS-synthesis using 100 components (see figure 4.7) yielded in a big difference depending on which quantization method used. Changing from *saturation* to *wrap around* brought down the utilization of LUT's from 89% to 32%. This is because *wrap around* is the simplest way of handling overflow and uses no extra logic. To implement *saturation* for arithmetic operation, depending on the datapath specification, requires significantly more logic to check for different types of overflow. This scales out of hand as the design grows larger and shows the importance of knowing how these things are implemented in hardware and how using as simple representation as possible reduces the logic utilization.

In table 5.16 the utilization of the large system is presented. The bit-exact implementation has the worst result in terms of area where the overhead for the bit-exact is 28% compared to the VHDL implementation. This is assumed to be because of both the original overhead from the single component as well as the control- and interface logic which took the most time of the HLS-synthesis for the large design.

	VHDL	**Bit-Exact**	**Ultra High Level**
LUT's	25104	32153	19605
FF's	25100	37205	19355
DSP	500	500	600
BRAM	200	200	102

Table 5.16: Detailed result-matrix for the large system showing the number of components.

It is worth noting that the ultra high level design actually has the least area consumption for the large system, see table 5.16. This is most likely arose from the HLS tool being able to optimize the chain in the large system in a way that was not done in the other implementations.

Chapter 6
Analysis

This chapter aims to represent the four main points of investigation for this thesis and present them accordingly. This is done by analysing and discussing the results from the case studies in more detail. This chapter will be divided into four sections of analysis, synthesis- and simulation time, hardware utilization, timing and workflow.

6.1 Simulation- and Synthesis Times

The HLS-synthesis process is important as it is used to generate RTL code from the C source code. There are mainly two things that affect the synthesis and simulation times, C/C++ syntax and libraries. The syntax affects what hardware that is generated as well as its performance but can also have an impact on synthesis- and simulation times. The usage of libraries affects the HLS-synthesis results and time in different ways depending on how they are used. In this section the synthesis and simulation times are analyzed discussed as well as the performance estimates and the readability of the generated RTL.

6.1.1 Syntax Effect on Synthesis

The syntax of the C/C++ code can have an impact on how the source code is synthesised into RTL. False dependencies causes the tool to reduce timing performance and latency by implementing dependencies in hardware that was not intended.

6.1.2 Libraries and Their Affect on Synthesis- and Simulation Times

The libraries used provided by Vivado can affect the synthesis and simulations in different ways. Often they affect the synthesis, times and results, in a positive way and the C run times in a negative way depending on the case. In this section the library effect on simulation and synthesis are analysed.

Arbitrary Precision Data Types Pros and Cons

Using Vivado HLS-libraries can be beneficial by forcing the tool to synthesise in certain ways. The library for arbitrary precision data types allows the designer

to specify the number of bits that should be used after synthesis [8]. Which is crucial for designers because using more bits than required can have large effects on design performance and area.

An unwanted effect is that **the C-simulations becomes slower**. Thus affecting the faster iteration times for functional verification in a negative way. **Using native C data-types do decrease the simulation times** of larger systems, see table 5.4, when comparing behavioural RTL simulation to C-simulation. Keeping native data types throughout the iterative design process, only verifying the code with arbitrary precision types, would keep the fast iteration times promoted by HLS advocates.

Streaming Interface to Reduce HLS-Synthesis Time

Vivado HLS provide a library for streaming- inputs and outputs. As for the case of the arbitrary precision library this provides good effect on how the code is synthesised into RTL. It adds input and output registers as well as easier interactions with the interface directives which allows for integration in larger system using the standard interfaces provided by Vivado HLS. Using the streaming interface together with the dataflow directive [8] improves the HLS-synthesis times (see figure 5.1) compared to using single data point references as function attributes. As the dataflow is used for task level parallelism it performs much better when used in top-functions compared to the pipeline directive which works best for components, or sub-functions.

The stream library has had one drawback that stands out during C-testbench simulation during C/RTL-Cosimulation. When simulating with a larger number of inputs, such as the one million used for the case study simulation, holding all inputs in the stream causes the tool to simulate slower. While holding only ten thousand inputs in the stream sped up the simulation by a considerable amount. These result were never measured in CPU time. In elapsed time, however, simulating ten thousand inputs using a stream holding a million input variables took approximately ten minutes. While altering the test bench to alternate between reading and simulating ten thousand at the time reduced the elapsed time down to seconds. This problem could possibly be solved by allocating more memory to the simulation.

6.1.3 Use Cases for Directives

Using loops that uses the loop index to access parts of an array. These could synthesize with dependencies so that what ever operation that is done on the current index has to be done before the next iteration can start, thus preventing pipelineing on operations done on the same array. In this case the dependencies can be removed in synthesis with the use of the directive *dependence* which would cause the tool to remove dependencies either between loop iterations with *inter* or inside loop iterations *intra*.

Digital Signal Processing (DSP) applications often use the same hierarchy as described in section 2.3.1. In the large system design for this thesis it was noted that using the **dataflow directive for the top-module to implement task level parallelism**, and input- and output streams, resulted in a better synthesis result both in terms of hardware utilization and HLS-synthesis time than the pipeline directive did. Using the pipeline directive on the top-module with single data points as I/O increased the number of LUT's in the design significantly. It also caused the HLS-synthesis times to rise significantly as the design grew larger, see figure 5.1.

6.1.4 HLS-Synthesis Estimations

When using the Vivado HLS tool for hardware design C/C++-code is synthesised into RTL-code, either VHDL or Verilog. It is important to note that there is a difference between HLS synthesis, from C/C++ to RTL, and RTL synthesis, RTL to netlist. Vivado HLS provides a performance estimate during HLS-Synthesis. This estimate predicts initial timing and utilization numbers. The timing estimates have been accurate, i.e. if a HLS-synthesised design meets the timing requirements it does so in RTL-Synthesis as well. The accuracy of the utilization estimates are not as good. It has for the duration of the thesis work **always reported too high estimates**. This is assumed to be of design where the tool gives a pessimistic view which then is improved during RTL-synthesis. But the overhead on the could be large, more than 200% on the look up tables and the flip flops, see table 5.4.

6.1.5 Readability of Generated RTL

The Readability of the generated RTL during HLS-Synthesis is subjective but also an important part of the workflow and should therefore be considered during the analysis. The section about readability is based on the authors opinion and experience and should not be considered as scientific quantifiable result. In the ideal case one should not have to modify the generated RTL and in many cases it probably wont be necessary. In specific cases however changing a certain signal inside the RTL might be needed and thus the readability comes in to play.

The RTL generated by Vivado HLS could in a crude way be divided into two type of files, top- and functional-files. The functional-files are for example the files where the complex multiplication or, for the bit exact case, the trigonometric memories are defined. These are quite easy to read and one could understand the logic and follow the functionality without to much effort. The top-files, are connecting functionality-files and eventual control-logic, could be quite bloated. They usually contain a lot of signals with names that are hard to interpret and logic could be hard to follow. If one do not have a clear view of what a signal is meant for it could be really hard to understand it without a schematic drawing of the signals. An rule of thumb for designing in HLS is to give all functions and variables as unique names as possible. Doing so can make the analysis result provided by Vivado HLS and the RTL more readable.

Synthesis of Design Hierarchy

In complex systems the top-function do in many cases consist of sub-functions. By default Vivado HLS tries to reuse hardware for different operations meaning that if the same sub-function is called twice in one top-function it will try to use the same hardware for both calls. Depending on the timing requirements of the circuit this could be beneficial but in some cases it is not.

6.2 Area Consumption and Hardware Mapping

This section of the report the hardware utilized on the FPGA board analysed and discussed. It will cover the algorithmic mapping from the different parts of the designs done in the case study as well as how the control-, quantization and rounding logic generated automatically by the tool affects the on board utilization. It will also cover how directives change the synthesis process by applying, or removing, different hardware optimizations as well as how syntax can be modified to improve utilization.

6.2.1 Algorithmic Mapping on to Hardware

In this section the hardware mapping of the different components in the case study will be analysed. In general functional implementation the bit-exact and the VHDL implementations are much alike. The ultra high level implementation differs mainly because the library solution source code used do not implement the same algorithm.

Complex Multiplier Mapping

The complex multiplication is implemented using the same DSP-structure in all three of the abstraction levels, see figure 5.7. The difference is how the complex multiplications were implemented where the bit-exact model was implemented using equations 4.2 and 4.3 and the ultra high level using an library implementation. The library implementation uses arbitrary precision data types which have been affecting the C simulation times in a negative way, thus the bit-exact implementation is recommended so that functional verification can be done using native C data types. In the schematic the only noticeable difference is that the bit exact HLS model has a single LUT implemented for the common term in the three multiplication algorithm, see equation 4.2 and 4.3, connected to the control signals for empty input and output streams.

Between the bit exact model and the RTL model there is a quite large difference in in the number of FF:s required in the system. One set of registers where not absorbed into the DSP unit in the bit-exact when comparing to the VHDL implementation. This is assumed to be because of the control signals constraining

the timing so that the registers had to be used outside the DSP units. Four registers arises from the control signals for the input- and output FIFO control signals. The ultra high level frequency translator uses more registers than the other which arises from a longer latency from a fully pipelined circuit.

Cosine and Sine Generator Mapping

The bit exact model does not map and separate the 13-bit input address in the same way as it was done in the VHDL design. In the VHDL model the three most significant bits are used to determine which of the eight segments that the current output phase should be in. In VHDL this can be done by selecting specific bits with the *downto*-statment. When trying to mimic that functionality in C++ the bit wise operator *&* was used with the decimal values so that the upper tree bits always were zero for the address and the lower ten bits always were zero for the segment. This did not carry out in logic similar to the VHDL model where the segment is delayed and then used in a MUX to select the correct output. Instead all 13-bits of the input address is forwarded in parallel with the 10-bits used when addressing the BRAM. The forwarding of these bits will use more registers in the final design than if it only would use the three bits as in the VHDL design. These bits do get optimized away after HDL-synthesis as they are always zero. But this is one of the reasons that the overhead in table 5.1 is less after HDL-synthesis compared to HLS-synthesis.

Analysing the generated RTL from the bit-exact model the address input to the cosine and sine generator are connected as selection bits to a MUX with inputs that are either grounded or set bound to one. These bits are then connected to a tree of and-gates together with the control logic. Exactly what this does is hard to interpret. The cosine and sine generator is also synthesised as a top-model with sub-blocks and therefore it has its own interface logic implemented with internal control logic.

An interesting calculation done in the frequency translator is the sine and cosine calculation and that is the main reason a frequency translator was implemented. The trigonometric calculation done in the VHDL model, see figure 5.3, is the same as the hardware goal design in section 4.1.1 after implementation. This implementation utilizes the hardware on the FPGA fabric by mapping cosine and sine segments to one BRAM respectively. The bit-exact model which aims to recreate the hardware of the design specification using HLS utilizes almost the same amount of hardware as the VHDL model with 16% overhead on FF's and 45% overhead on the LUT's. The number of DSP's and BRAM's are the same for the two models, see table 5.8. Looking at the more detailed tables 5.9 and 5.10 the cosine and sine generator uses seven more FF's and 17 more LUT's. The post implementation schematics suggests that one of the reasons for this should be the product of more developed control logic, i.e. valid- and enable signals. The VHDL model did, for example, not implement chip enable logic for the cosine and sine generator while this was the case for the bit exact model.

In general for all three implementations were that the increase or decrease in FF's mainly depends on the latency of the design. This is because it is pipelined to have a throughput of one causing one clock cycle latency to increase the number of FF's with at least 36, same as the number of bits in the data path.

Power Meter Mapping

In HDL the addition of the squared I and Q takes place within the second DSP48 or the one which squares Q and requires therefore no extra hardware. Whereas in both HLS designs its done outside with extra dedicated hardware which is one good reason why the HLS has higher numbers for the utilization. The first addition of the squared I and Q is not reused so another one is implemented for storing the value in the register, increasing LUT utilization even more. The bit-exact and ultra high level HLS designs both implements three groups of extra registers for storing a sub part of the addition before being added together, see figures A.2 and A.3. This will increase the use of FF's compared to the VHDL since it does not implement registers in the addertree. It can be seen in table 5.14 and 5.15 that the ultra high level design has a greater LUT and CARRY8 utilization than the bit exact and this is due to the increased bit width of the early state adders in the addertree, see figures A.2 and A.3 in appendix A.

The last division in the power meter where the summation is divided by the number of entries in the shift register are done as a bit shift in the VHDL design and in both HLS designs. This can already be seen at RTL-level in the block diagrams A.1, A.2 and A.3 where there are no extra hardware implemented for the division. However in both HLS-designs one had to specify that one wanted it done as a bit-shift since the HLS tool did not convert the division to a bit shift despite it being a power of two.

6.2.2 Quantization and Rounding Affect on Utilization

The HLS designs implements logic for checking overflow after every addition which increases the use of LUT's. This is done in a different way in the HDL were the check is only necessary after the two multiplications and the addition of them since the bit width of the bus is increased by 3 bits for the other additions which end with a 3 step right shifts for dividing with the number of 8 entries, overflow can therefore not occur there. Both of the HLS designs also implements a series of XOR-, AND and OR-gates for checking for overflow which is implemented with LUT's. In HDL this done by checking if the six most significant bits are equal to "000000" or "111111" which will indicate if an overflow has occurred, this operation is done within the DSP which by an output signal called "PATTERNDETECT" to control some LUT's configured as a MUX to saturate if overflow has occurred.

The output of complex multiplier in the bit-exact HLS model is connected to 38 LUT's which are assumed to be used for the Quantization- and Overflow methods defined for the fixed point data path in the HLS model. Comparing with the VHDL model the number of LUT's after the complex multiplication are less. This is the main reason to why the number of LUT's are higher for the bit-exact than the VHDL model, see table 5.10.

To have rounding integrated in the circuit is one aspect of HLS that could both speed up design time as well as reducing the number of errors introduced in the design even before verification has begun. At the same time the designer has to be aware and keep in mind the extra logic required when implementing different operations using these Quantization- and Overflow methods. If not it could potentially cause the final design to be unnecessarily bloated using logic that would not be required to meet the requirements of the system.

6.2.3 Automatic Implementation of Control Logic and its Affects on Design Results

Through out the HLS designs control logic is used to enable different parts of the circuit. These control signals are separated, one for the real and one for the imaginary input, even though both inputs only could be valid or invalid at the same time. In some cases this might be useful and would enforce that all inputs are valid in order for the circuit to produce valid outputs. For the circuit described in chapter 4 one signal is enough for both the real- and imaginary input signal. There could be directives or solutions that only uses one valid signal, but that has not been uncovered in this thesis.

In the VHDL model of the frequency translator one set of imaginary input registers are absorbed into the internal registers of the DSP units used for complex multiplication. In the HLS more registers are kept outside the DSP-units. This is assumed to be because of timing reasons where the synchronous input signals, both real and imaginary, of an empty input queue is used to control the calculations of the complex multiplier. In the HDL design the valid input signal is not used to enable the multiplier. It is only used to signal if the output is valid, i.e. if the inputs the calculations where done on where valid.

The first big difference is that both HLS designs implements control logic that will enable all registers and some data-paths and it also features a latching capability which will keep it going until the present inputs have passed through. This control logic is controlled by the input signal "ap_start" and output signal "ap_idle", were a one in at "ap_start" starts the system and a one out at "ap_idle" indicates that the system is idling hence no new values are being processed. Wheres in the HDL design the registers and data-paths are always enabled. The HDL design has one register for delaying the valid signal 1 clock cycle for syncing it with the output. The HDL however has 32 LUT's at the input for keeping the data valid so no undefined data are stored in the shift-register, these might however be removed if the hardware which is responsible for the input data gives valid data when the

valid in is not set to one.

The control logic seems to be depending on the interface and syntax of designs. The individual component designs yielded in less utilization and better latency for the bit-exact models (see tables 5.8 and 5.12 for utilization and tables 5.5 and 5.6 for timing) but the large design utilization was worse for the bit exact implementation, see table 5.16. It is assumed to arise from the control logic implementation as well as rounding and quantization. The logic implemented by the libraries does at scale outperform the hardware generated by the bit-exact because of this.

Throughout the thesis work, especially for the big design, false dependencies and feedback has been a returning concern. The way these could arise are many. Reuse of hardware could be one of them, if-statements another. One has to be aware of these kind of issues and how they can arise and be solved to efficiently design in HLS. In some cases it could be solved by changing the syntax or trying a different algorithmic approach. Other times it is solved by using directives telling the tool how to synthesise in order for the sought after results to be achieved.

6.2.4 Data Typecasting to Reduce Logic Utilization

If for example a design like the power meter that has signed inputs, but has unsigned right after the multiplier due to the power of two on both the I and Q inputs. One should **specify that the output is unsigned or it is automatically interpreted as signed which will result in extra logic being used**. The tool does not optimize the for unsigned despite that it is not being used, this will decrease area and improve timing, see figure 6.1 where TC_OU is an unsigned data type.

```
process_sqrI:{II_out = TC_0U(I_in*I_in);
}

process_sqrQ:{QQ_out = TC_0U(Q_in*Q_in);
}
```

Figure 6.1: Cast to unsigned

In general it is a good practice to **use unsigned over signed data types as well as integer over fixed point when it is possible**. This reduces the logic needed to perform arithmetic operations and as the complexity of the hardware increases this makes a difference in the performance of the generated RTL and hardware.

6.3 Aspects of Designing Which Affect Timing

There are different aspects of a design that can affect the timing performance of the final implementation. The two main ways are the use of directives and how

the code is structured in the implementation, both of which will be covered in this section.

6.3.1 Syntax Affect on Timing

The structure of the code can have an impact on the timing performance of its final implementation. This subsection aims to present the ones that was encountered during this thesis.

False Dependencies Reduces Timing Performance

When writing HLS code the compiler will synthesise a *if, else if, else* - statement as a continuous datapath. This yields in hardware, after HLS-synthesis, where each if-condition is tested each passing of the implemented datapath. To avoid this, and reduce circuit latency, it is possible to use a *switch-case* - statement. One can draw the comparison between the switch-case in C and the case statements in, for example, VHDL. The tool behave like this when the same variable is used in all the boolean conditions for the if-statements. **The *switch-case* - statement gets implemented with the same performance as a MUX in hardware**. Using the *default* - state in a switch gives rise to a false dependency and should be avoided by specifying all cases if possible. Throughout this thesis conditional statements has been used over *if, else* statements as it can have a similar effect on the HLS-synthesis. But this has not been the case for all implementations.

Arrays are implemented either as registers or as BRAM in HLS. When implemented as BRAM arrays can cause false dependencies when they are used at both sides on an assign clause or twice in the same loop [17]. This implements a feedback where the operation on the next index can not start before the former has finished. This could be solved by using one array for reading one for writing, or by using the *dependency* directive. If two sequential loops operates on the same array this could require one loop to finish before the next starts. This dependency can be avoided by using the *dataflow* directive allowing task level parallelism.

Syntax Affect on Timing

When writing code for HLS one needs to consider in what order inputs are ready, take for example an add function. If one input is from a multiplier and n other inputs is from different registers, the one from the multiplier will arrive later in the current clock cycle while the ones from the registers are available from the start. If the one from the multiplier is written first then it will be first in the addertree which means it will have to go through the whole adder chain and will therefore increase the time required for the output signal to stabilize which will decrease the slack. So signals which are to be expected later during the current clock cycle are to be put last in the addition to prevent them from ending up first in the adder chain, see where "II" and "QQ" are in figure 6.2.

In order to create a flattened addertree and not a long chain of adders the tree structure had to be manually written in C++. When done on a single line in the

C++ code the adders would always be synthesised as a chain of adders which, for the power meter implementation, were sub-optimal and meant that the tool could not meet the timing requirements. When the additions were written in parallel as in figure 6.2 like an addertree, the tool managed to synthesise it correctly. But it did not when written like the commented "process_sum" in figure 6.2.

```
add1 = shrg1;
add2 = shrg2 + shrg3;
add3 = shrg4 + shrg5;
add4 = shrg6 + shrg7;


PWRavg = (add1 + add2 + add3 + add4 + II + QQ)>>3;
//process_sum:{PWRavg = (PWRinst_in + shrg1 + shrg2 + shrg3 + shrg4 + shrg5 + shrg6 + shrg7)>>3;//
//}
```

Figure 6.2: Syntax for writing an adder-tree

Using Variables to Implement Registers

A variable declared as static implements a register but using it causes that variable name to be occupied for the design. Using the same variable name in any other part of the C-code will refer to that variable, or register, even if it is called from a different sub-function. In figure 6.3 the sub-functions *foo_1* and *foo_2* should be synthesised as two separate components connected under *top_function*. In each function call one would like to increase the internal registers of the foo functions by one and set them to the output. If the static variables share the name *var2*, it will be equal to two using the syntax in figure 6.3. In figure 6.4 each static variable has a unique name an they won't share register i.e. $var1 = var2 = 1$.

```
void foo_1(int& var)
{
        static reg = 0;
        reg += 1;
        var = reg;
}


void foo_2(int& var)
{
        static reg = 0;
        reg += 1;
        var = reg;
}


void top_function(int& var1, int& var2)
{
        foo_1(var1);
        foo_2(var2);
}
```

Figure 6.3: Example code showing how static variables could be shared.

```
void foo_1(int& var)
{
        static reg_1 = 0;
        reg_1 += 1;
        var = reg_1;
}


void foo_2(int& var)
{
        static reg_2 = 0;
        reg_2 += 1;
        var = reg_2;
}


void top_function(int& var1, int& var2)
{
        foo_1(var1);
        foo_2(var2);
}
```

Figure 6.4: Example code showing how to avoid static variables sharing.

Using temporary variables within algorithms causes the tool to implement registers in the places where the variable is implemented if it is needed to meet timing. Coding with temporary variables also allows for different data type declarations which can be useful to force different parts of the code to be synthesised with different bit-lengths, quanitzation- and overflow methods. The usage of temporary variables are also implemented in the provided libraries and could then, perhaps, be seen as good practice.

Shift Register Issues With Multiple Read

There is a library for a shift register provided by Xilinx called "SRL IP Library" [8], this should have been something for the ultra high level design. Unfortunately this library turned out to be very time inefficient when one wanted to use the function to read the entries of the shift register and there were therefore not possible to develop a design which would have met timing despite many attempts, so this library had to be discarded for another implementation.

6.3.2 Directives Affect on Timing

Using the latency directive could force the tool to reduce the latency of the circuit. In both the single component cases one could decrease the number of clock cycles by setting an upper limit on the latency. This forced the tool to generate hardware with lower latency which also reduced the number of FF's needed in the data-path.

When using the directives to reduce latency to six for the frequency translator design the HLS synthesis latency was reported as six clock cycles. After C/RTL-Cosimualtion the latency of the circuit was reported as five cycles. But when exporting the RTL into the Vivado HLS editor and running an RTL-Analysis the datapath contained six registers as do the enable-signal path. The latency error of the tool could perhaps be critical in some applications and why there is an difference between the C/RTL-Cosimualtion and the RTL code generated is unclear.

The large system was limited by the HLS designs not meeting the timing requirements as more components was connected together. The failing paths was due to path delays over $2ns$ for synchronization signals implemented automatically by the HLS tool.

6.4 Changes in Workflow from HDL to HLS

This section of the analysis will cover a short analysis of noticeable workflow changes both for design and verification when switching from HDL to HLS. This has not been the main focus of the thesis but it is worth a mentioning to provide guidelines and information how HLS can change the approach to hardware design.

6.4.1 Design Differences

Perhaps the largest difference in projects when the design is done in HLS versus classic HDL is that less human interaction is required to create the same functionality. As an example quantization and rounding logic do not need manual implementation throughout the design. This removes the possibility for bugs and errors to be introduced in these parts of the design. While HLS is not as effective as RTL designs when it comes to optimizing hardware, in terms of the speed and area see table 5.8 and 5.12, it do automate parts of the workflow reducing the possibility of design errors.

A change in verification and workflow is that for HLS the system model could be the same as the hardware design model because both can be written in C or C++. This would allow the system designer to simultaneously develop both the system model and the hardware design model. The source code can not be written as if it was a normal software program or model but has to be developed with the hardware implementation in mind, as discussed earlier in this chapter. In larger system designs this would allow the system and hardware design to be done by the same person even for larger systems.

6.4.2 Verification Differences

The method of verification is changing when transferring from HDL- to HLS designs. In traditional HDL the circuit functionality is verified using RTL behavioral simulations. Meaning that the whole circuit is emulated in the computer running the simulation. The behavioral simulations could be very time consuming for large RTL simulations, see table 5.4, which could slow down the verification of the functionality. With the HDL-workflow, see figure 3.1, the iterative process of verifying and updating the functionality of the design is slowed down significantly. If it is possible to use the simulation times of the bit-exact model, see table 5.4, and solving the problem with HLS-synthesis time the behavioral verification could be speed up by a significant margin. The gain of performing functional verification scales with the design, as can be seen between the single component simulations in tables 5.2 and 5.3 and the large system simulations from table 5.4.

When designing digital hardware using RTL system design the system model is usually implemented in a high level programming language such as MATLAB or Python. These languages are almost exclusively using floating point for all their arithmetic operations and has limited support for fixed point binary operations. This could cause problems when an RTL design is verified by a system model using floating point precision. Since floating point is in most cases more accurate than fixed point implementations this could cause a miss match between the system model and the HDL implementation. See figure 3.1 and 3.2 for an overview of the workflows. Because HLS is entirely written in C/C++, both design and testbench, the design could be verified using both floating point precision and fixed point precision just by changing the data types in the header-file.

Chapter 7
Conclusions

HLS has potential to increase the abstraction level of digital hardware design compared to HDL which is often used today. But to design in HLS do require previous knowledge of hardware design to achieve good results. In this chapter the conclusions from this thesis are presented together with areas for future work and ending with the opinions of the authors.

In software design it could be argued that good code is easy to read and implemented in as few lines as possible. Only in more demanding cases, if at all depending on the compiler, does a software designer have to consider optimization methods such as loop-unrolling and how things actually would map onto hardware. Using C/C++ to design hardware is different compared to software development in that case. **To design efficient hardware with C/C++ and HLS the designer needs to have the hardware implementation, and its underlying logic, in mind when writing the code**. In general one has to keep the target device in mind when designing and HLS is no exception. Disregarding the target FPGA fabric and the logic implementation will lead to inefficient hardware and performance.

7.1 Synthesis- and Simulation Times

The HLS-libraries are useful to ensure the tool synthesises the design in a correct way. They automate part of the RTL design process such as rounding and quantization logic after arithmetic operations, control signals throughout the design and ensure timings are meet with registers. While they are beneficial for the HLS-synthesis they have a downside for the C-simulations. **Using non-native data types and other library tools provided by Xilinx affects the simulation times for C-simulations in a negative way**. It is possible to work around these in different ways, trying to get the best of both worlds, but one has to be aware that these problems do exist. Many times **using library components removed the possibility to change data types and speed up C-simulations** which argues for using a bit-exact way of designing.

The estimation of the timing done by the HLS-tool is conservative when compared to the estimation done by the HLx-tool after export. This is also true for the utilization estimates where there is a large overhead after HLS-synthesis compared to RTL-synthesis, see table 5.1. These estimates are partly due to **optimizations done in HDL-synthesis such as removing signals bound to zero and division into shift is not implemented in HLS-synthesis**. These optimizations is done when the generated RTL is synthesised into a netlist which is the next step of HLS workflow, see figure 3.2.

Due to the very long synthesis times for connecting several sub-components, see table 5.4, it is better to use HLS to design logical components to a system separately. The tool is not equipped to synthesise the interconnections and timing between several components in a large system in an effective way. These components could instead be connected manually using another tool, for example RTL, where as HLS seems better equipped to deal with DSP implementations where it do not have to take control and timing of designs that utilizes the majority of the FPGA fabric into consideration.

The C/C++ syntax has impact on how the source code is synthesised to RTL. There is a trade-off to be made when designing hardware using Vivado HLS. The bit-exact design which is more based on normal C/C++ simulate faster, see table 5.4, but perform worse than the ultra high level in HLS-synthesis, see figure 5.1. **The ultra high level model is based heavily on libraries provided by Vivado and are better suited for HLS-synthesis than the bit-exact model**, see figure 5.1. For the individual component designs these times were not as noticeable and the utilization were better for the bit-exact model, which were almost as good as the VHDL design, see table 5.8 and 5.12. This shows that HLS has potential for fast synthesis as well as good hardware utilization if one can has a good symbiosis between library functions and bit-exact hardware design. To keep down the simulation times it is important to use native C data types [14]. But using them collides with some of the provided libraries that uses arbitrary precision types as default which removes the possibility to speed up C-simulations. One has to be aware of the fact that the Vivado libraries improve HLS-synthesis times but reduce C-simulation times and not using them could have the reversed effect.

7.2 Hardware Utilization

The hardware mapping is good for the HLS implementations when writing in a bit-exact way where **most of the overhead comes from control-, interface-, overflow- and quantization logic**. The ultra high level model is not specified in such a detailed manner and the resulting hardware perform worse for both case study components.

The tool generates control-, interface-, overflow- and quantization logic automatically. This is in many cases a good feature and an example of an area where the design and verification is made faster by always implementing this thereby reducing the number of errors that can be introduced. At the same one has to be aware of this fact and what method is used to implement this logic because when not used as intended it introduce a large overhead on the hardware utilization.

Using the **arbitrary precision library improves the hardware utilization** overall and allows the designer to chose quantization and rounding methods. This allows for more accurate designs but as a designer one has to be aware of these functionalities as well. The large system results show this in a clear manner where **changing quantization method from *saturation* to *wrap around* [8] reduced the hardware utilization of LUT's from 89% to 32%**. These functionalities are implemented by default in Vivado HLS and if one is not aware of these hardware solutions and when they are implemented could cause sever reduction in hardware performance.

The ultra high level design had the largest overhead for the single component after HDL-synthesis implementations of the modules, see table 5.8 and 5.12. The bit-exact implementations had a better result for these modules with less overhead as well as more flexibility around the design and simulation to reduce the time required

In order to generate efficient designs using HLS a bit-exact design approach, with a previously defined hardware structure, is preferable to a ultra high level approach in terms of hardware utilization as well as simulation- and design speed. Overall using a bit-exact HLS approach includes the benefits of designing and verifying with C/C++ while keeping, almost, VHDL utilization performance. The ultra high level approach do overall perform worse than the bit-exact approach for the individual components. Using existing libraries and a standard C way of coding is more beneficial if how the hardware should be implemented is unknown but the system functionality is defined. It could be used for prototyping- and proof of concept designs where the optimization of the hardware is not as high of a priority.

7.3 Timing

In general HLS did not have any problem to meet timing for the individual components. In the case study the timing where meet both for the bit-exact- and the ultra high level designs. Despite that they were both pipelined to achieve a relatively high clock frequency the designs had a throughput of one. The bit-exact did even manage to have the same latency as the VHDL implementation and the ultra high level difference was mainly due to a different algorithmic implementation. Using Vivado HLS also **removed the possibility to implement latches** in the design, which is a common mistake in RTL-design.

There were problems with timings after implementation, HDL-synthesis and place and route, as the design grew larger that arose even if the timings were met after HLS-synthesis. Timing issues at this stage can be hard to debug as one has to go back and change in the source code or directives to solve it. **Except for the large system design the timing errors got caught by the synthesis tool** before the HDL-synthesis and implementation.

7.4 Workflow

Automating parts of the design process such as quantization and overflow increase design speed and reduce the possibilities of introducing errors. In the case studies much of the overhead from the bit-exact design arose from these type of logic implementation that was automatically implemented. While it is a good feature that these are done automatically **as a designer you have to be aware of how functionality is implemented in hardware in order to achieve as good performance as possible**.

The functional verification in C is faster than behavioral simulations especially as the design grew larger, see table 5.4. This has a big impact on functional verification as the behavioral simulation was speed up significantly. This allows for faster verification but can also lead to a better coverage as more test cases can be investigated due to the fast simulation times.

7.5 Future work

It has been observed during this thesis that more bugs are more easily found when one simulates the RTL with the C/C++ based test bench, it would therefore be interesting to see if it is possible to somehow measure the increase in fault coverage.

An investigation in how other HLS tool's such as Cadence Stratus and Mentors Catapult compare to Xilinx's Vivado HLS tool in terms of usability and the ability to create efficient hardware designs. Together with the possibility of implementing HLS that is vendor agnostic would be an interesting continuation of this thesis.

An interesting analysis for future work would be to analyse if HLS-Synthesis behave differently when written in C or System-C as all impelemtations were done using C++. This is also true for the target RTL implementation that has been VHDL throughout this thesis. The efficiency of VHDL implementations versus Verilog implementations has not been evaluated either.

7.6 Authors Opinions

This section goes through our thoughts and experience after working with HLS for the duration of this thesis. It will cover the upsides downsides and how we think the future of HLS will look like.

Designing in HLS definitely has its advantages such as **fast iteration times for functional verification**, see table 5.5, 5.6 and 5.4, while cutting down on implementation tasks by automatically applying rounding, quantization and control-logic. There is also the design speed, which is much more subjective to the designers, where we believe HLS is better especially for DSP-design and algorithmic implementations. The **possibility to quickly change and try optimization methods** such as pipelineing or loop unrolling is definitely an advantage and allows for more flexibility in case any alterations would be required.

When the designs are implemented the tool enforces good practice implementation of surrounding logic which we considered a good feature that removes the possibility of introducing errors and mistakes but it does have down sides as well. In this thesis the complexity of the components were kept low which reduced the need for this type of logic and that we did not manage to do as much for the HLS designs. As designer one has to be careful so that the tool do not implements more control logic than necessary.

Implementing designs, especially large designs, you as a designer **need to be aware of surrounding logic such as control and quantization logic**. When and where it is implemented or the logic utilization can get much larger than intended. We think much of the overhead, in HDL-synthesis times and area, of the large system were caused by these type of logic implementation which did not exist in our sparse VHDL implementation. This is true for arithmetic calculations as well where integer calculations are easier than fixed point thus one should try to do as much as possible with integers.

Vivado HLS **works best for sub-modules and components**, using it to generate and implement a large system by its entirety slowed down the GUI of the tool considerably and HLS-Synthesis for larger designs were slower than for HDL-Synthesis for the same design, see table 5.4. Vivado HLS can effectively implement and synthesis a design from C- to RTL-code while the scale of the design is kept at a moderate level. We think the **interconnections and control logic of a larger design is better left for an other tool** or even to be done in conventional HDL if the design is using a bit-exact approach. As was more evident designing the large system where the majority of the HLS-Synthesis time was to implement the interface, i.e. control logic, then the sub-modules and functionality unless the interface was designed using pre-defined libraries.

The usage of **libraries in the design** had both positive and negative effects throughout the work on this thesis. While the arbitrary precision library **improved HLS-synthesis result** it did **reduce the efficiency of verifying functionality with C** as the simulation times increased. One can work around this and use native C data types for functional verification and only switch to arbitrary precision data types in the later stages and for synthesis. It is problematic that some libraries from Vivado only functions with arbitrary precision data types thus the functional verification speeds are lost. Other libraries such as streams can be

used all the time and should be used for the top-function in a data-flow application since they improve the HLS-synthesis time for larger systems.

Overall HLS feels like a tool that follows the direction of the trends in the industry, and of automation in general. Where it is pushing the abstraction levels of hardware design up, trading design specific control for speed. While HLS has been around for a while and adopted by some we think it will only grow larger as it get more users, leading to improved performance and better features. As the demand for more complex designs grows fast and at some point there just wont be enough designers to satisfy these demands. Combine this with the saturation of *Moore's Law* [10], we think could point towards higher demands on system designers. HLS is a possible solution to this reducing the total number of designers needed for a complex design allowing designers to focus more on system level features and verification while leaving lower level synthesis to automation.

References

[1] Xilinx. Ultrascale Architecture DSP Slice, *UG579*. September 20 2019. [Online]. Available:
https://china.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[2] Xilinx. Ultrascale Architecture CLB Slice, *UG574*. February 28 2017. [Online]. Available:
https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[3] Xilinx. Ultrascale Architecture Memory Resources, *UG573*. Feburary 4 2019. [Online]. Available:
https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf

[4] Xilinx. Vivado Design Suite Tutorial, *UG871*. May 22 2019. [Online]. Available:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug871-vivado-high-level-synthesis-tutorial.pdf

[5] Erik Seligman, Tom Schubert, M V Achutha Kiran Kumar. Formal Verification An Essential Toolkit for Modern VLSI Design. 2015

[6] Xilinx. ZCU102 Evaluation Board Documentation, *UG1182*. June 12 2019. [Online]. Available:
https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/ug1182-zcu102-eval-bd.pdf

[7] Xilinx. FPGA Xilinx Zync Ultrascale+, *Zynq UltraScale+ RFSoC*. 2017-2019. Accessed: Nov.26, 2019. [Online]. Available:
https://www.xilinx.com/support/documentation/selection-guides/zynq-usp-rfsoc-product-selection-guide.pdf

[8] Xilinx, Vivado Design Suite User Guide High-Level Synthesis, *UG902*, December 20 2018,
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf

[9] Xilinx. FPGA Xilinx Zync Ultrascale+, *Zynq UltraScale+ RFSoC*. 2017-2019. Accessed: Dec.10, 2019. [Online]. Available:
xilinx.com/support/documentation/selection-guides/zynq-usp-rfsoc-product-selection-guide.pdf

[10] M. Mitchell Waldrop. 09 February 2016. The chips are down for Moore's law. [Online]. Available:
https://doi.org/10.1038/530144a

[11] Xilinx. Complex Multiplier v6.0, *PG104*. 18 November 2015. [Online]. Available:
https://www.xilinx.com/support/documentation/ip_documentation/cmpy/v6_0/pg104-cmpy.pdf

[12] John Hennessy & David Patterson. *Computer Architecture: A Quantitative Approach*. 6th edition, Morgan Kaufmann, 2017.

[13] Doulos by John Aynsley. *UVM Verification Primer*. June 2010. [Online]. Available:
https://www.doulos.com/knowhow/sysverilog/uvm/tutorial_0/

[14] Stanley B. Lippman , Josee Lajoie , Barbara E. Moo. *C++ Primer, Fifth Edition*. Addison-Wesley Professional, 2012

[15] Pong Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006

[16] Umer Farooq, Zied Marrakchi, Habib Mehrez. *Tree-Based Heterogeneous FPGA Architectures*. Springer Science+Business Media New York, May 14 2012, DOI: 10.1007/978-1-4614-3594-5_2

[17] Xilinx. Vivado HLS Optimization Methodology Guide, *UG1270*. April 4, 2018. [Online]. Available:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf

Appendix A

Schematic of Power Meter Implementations

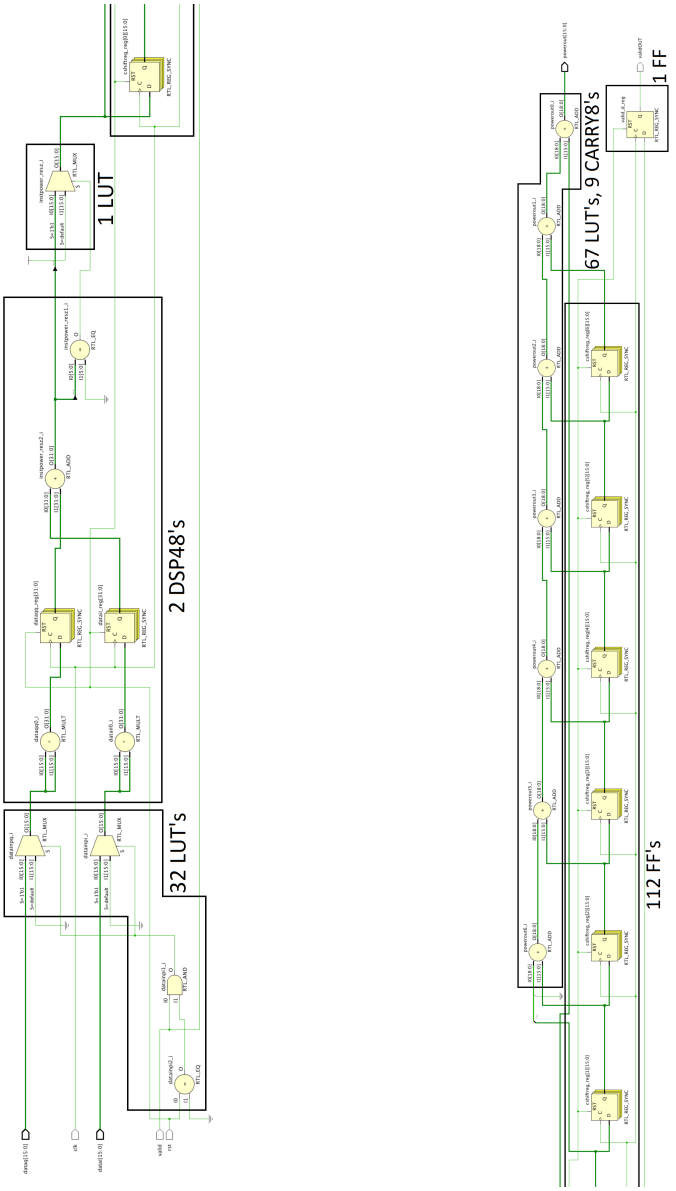

Figure A.1: Block diagram for VHDL design of the power meter

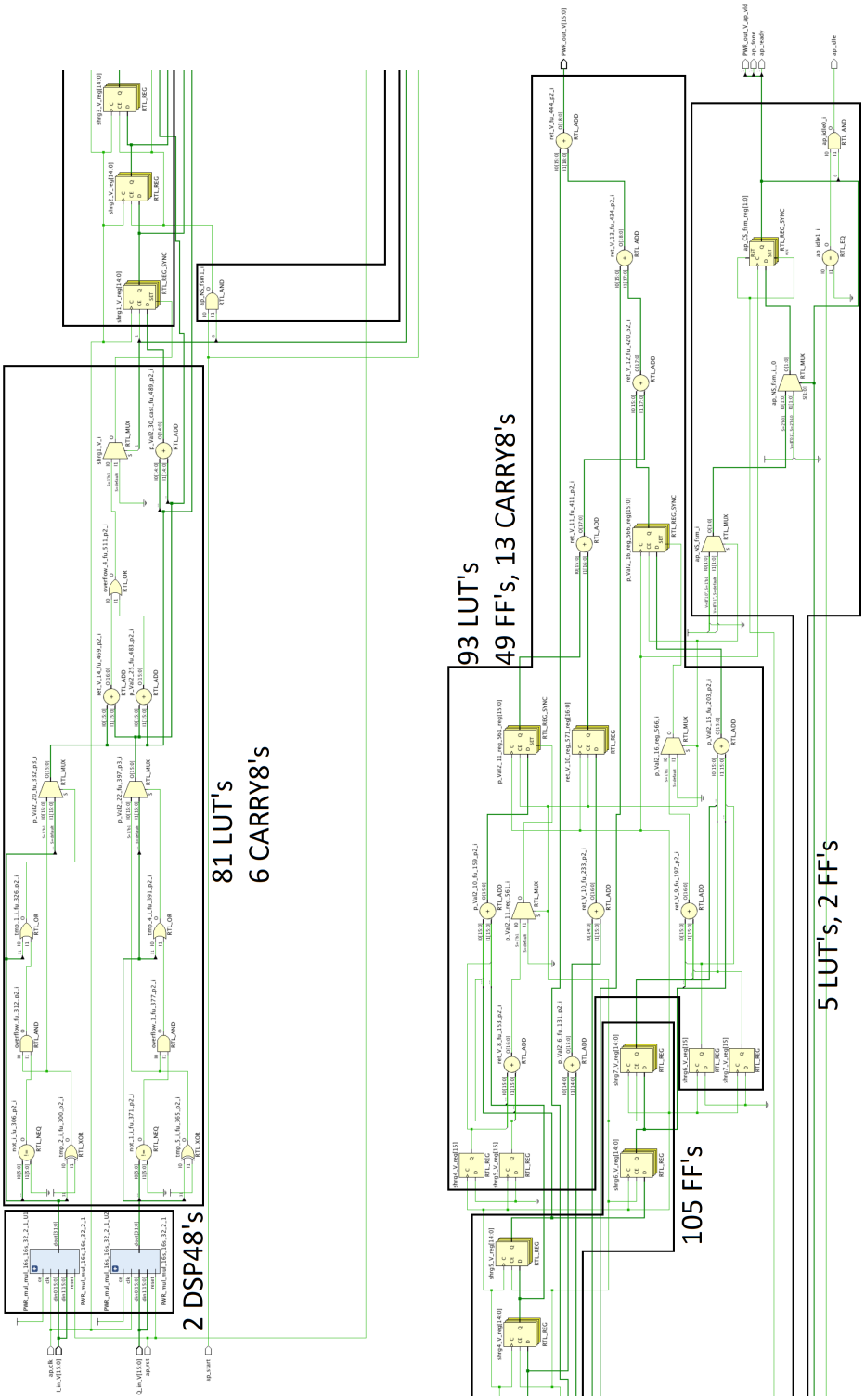

Figure A.2: Block diagram for bit-exact HLS design of the power meter

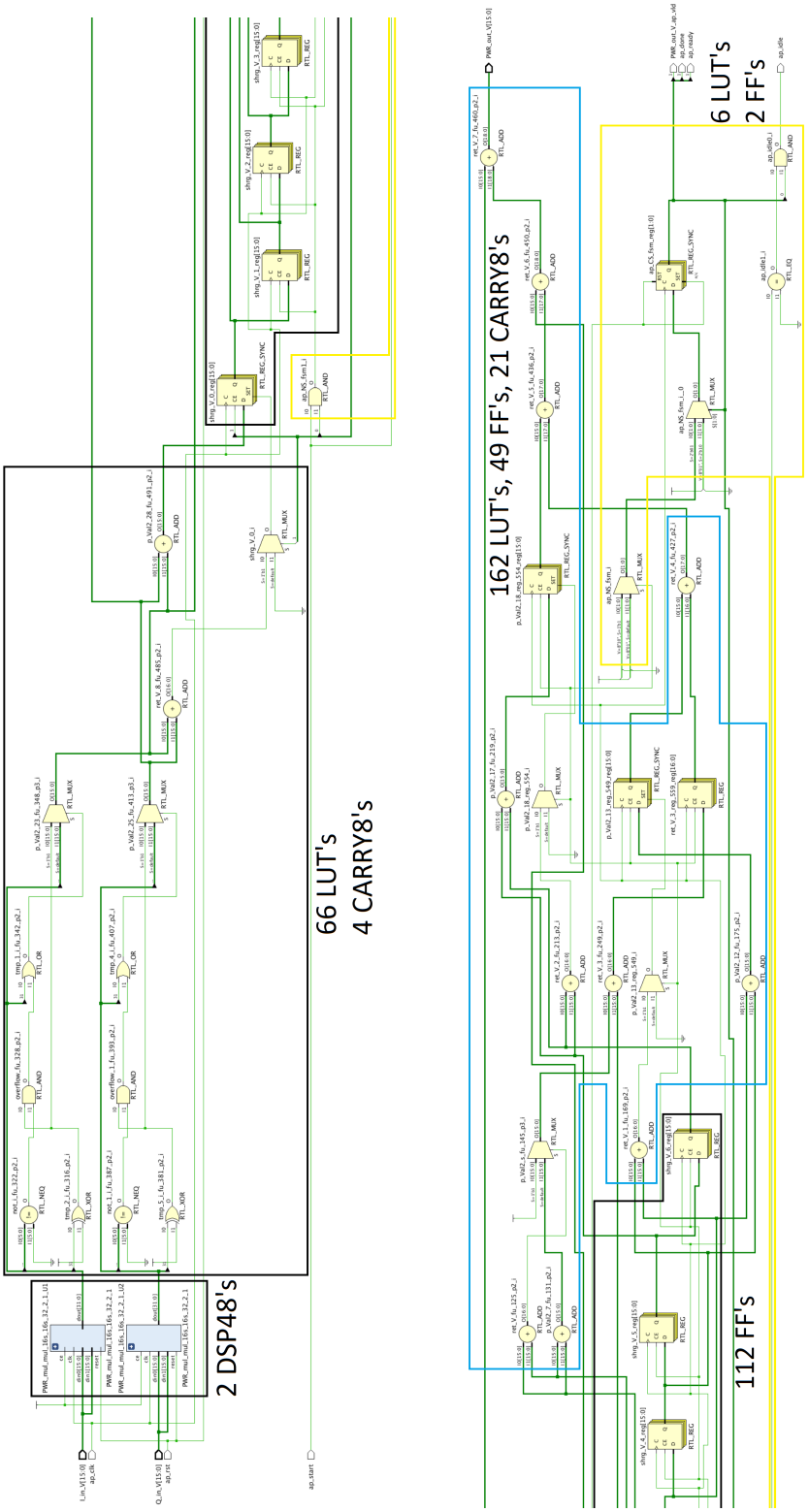

Figure A.3: Block diagram for TLM HLS design of the power meter

Appendix B

Example of Abstraction Levels

As an example of the difference in abstraction levels between designing hardware in HLS and VHDL consider the following circuit, see figure B.1.

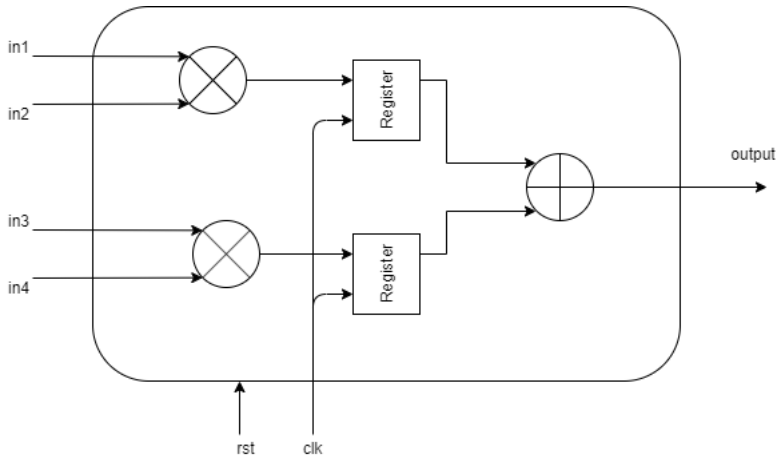

Figure B.1: Example component for abstraction level example.

This could be implemented using VHDL in the following way, see figure B.2.

```
entity top_function is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           in1 : in STD_LOGIC_VECTOR (15 downto 0);
           in2 : in STD_LOGIC_VECTOR (15 downto 0);
           in3 : in STD_LOGIC_VECTOR (15 downto 0);
           in4 : in STD_LOGIC_VECTOR (15 downto 0);
           output : out STD_LOGIC_VECTOR (15 downto 0));
end top_function;

architecture Behavioral of top_function is

signal  next_inter_a, curr_inter_a : SIGNED(32 downto 0);
signal  next_inter_b, curr_inter_b : SIGNED(32 downto 0);

begin

process(in1, in2, in3)
begin
        next_inter_a <= SIGNED(in1)*SIGNED(in2);
        next_inter_b <= SIGNED(in3)*SIGNED(in4);
        output <= STD_LOGIC_VECTOR(curr_inter_a + curr_inter_b);
end process;

process(rst, clk)
begin
    if(clk = '1' and clk'event) then
        if(rst = '0') then
            curr_inter_a <= (others => '0');
            curr_inter_b <= (others => '0');
        else
            curr_inter_a <= next_inter_a;
            curr_inter_b <= next_inter_b;
        end if;
    end if;
end process;


end Behavioral;
```

Figure B.2: Example component implemented using VHDL.

Implementing the same functionality in HLS could be done in the following way, see figure B.3.

```
void top_function(short& in1, short& in2, short& in3, short& in4, short& output)
{
        #pragma HLS PIPELINE
        ouput = in1*in2 + in3*in4;
}
```

Figure B.3: Example component implemented using HLS.

It is evident that the complexity of the code is reduced by letting the tool implement the RTL, see figure B.2, from the HLS implementation, see figure B.3. **OBS** running the design in figure B.3 wont generate the code in figure B.2 they are both designed for this example explicitly.