High Level Synthesis for ASIC and FPGA

MALIN HEYDEN

MASTER´S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

High Level Synthesis for ASIC and FPGA

Malin Heyden
malin.heyden@gmail.com

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu

Examiner: Erik Larsson

June 11, 2023

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This thesis aims to evaluate the performance of Siemens' High Level Synthesis (HLS) tool Catapult. HLS can be considered the next step up in abstraction level from writing traditional Register Transfer Level (RTL) code which is time consuming and error prone. The promise of HLS is to speed up the process of designing integrated circuits by lifting the abstraction level so the design can be coded fully in a high-level programming language such as C++. RTL code can be generated from the high-level description and then synthesized. The goal of this thesis is to evaluate if HLS can be used to efficiently share code for different target platforms. The compared platforms are Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA). Being able to use the same code base for both could enable prototyping on FPGA and faster development cycles. To evaluate the suitability of sharing code this way, a case study of a symmetric Finite Impulse Response (FIR) filter was conducted. This is a suitable design to compare because it has clear best implementations that differ between ASIC and FPGA. For FPGA it should be implemented using DSP-slices. Since an ASIC is fully custom it should be implemented with multipliers and adders. To achieve this, the same C++ source code was used with different pragmas depending on platform and the generated RTL was either implemented on FPGA or estimated area for ASIC. No backend implementation was done for ASIC. The result showed a need for slightly different code in the two cases to get the intended implementation. DSP-slices were not generated for FPGA below certain bit-widths. The results varied greatly depending on the language chosen for the generated RTL-code, VHDL or Verilog. In conclusion the C++ source code could only be optimized for one of the targets at a time. Using the same source for FPGA and ASIC would result in worse results for one of them.

Popular Science Summary

Advancements in technology in the past 100 years have been huge and affected our way of living a lot. More and more things are becoming "smart", it's not only our phones, but also things like lighting, cars, household appliances and much more. An important building block behind these inventions is the integrated circuit (IC). The innovation that lets multiple transistors be integrated on the same piece of silicon. This makes the chip smaller, faster, and more energy efficient. As a result the integrated circuits could also do more for the same cost. The development has continued rapidly with new smaller and even faster techniques to enable better products to manufacturers and then customers. But this fast development has also made the design of the circuits so much more complex. Chips can consist of billions of transistor, which is a staggering number. To be able to handle this increasing complexity computer aided tools are needed. It is simple impossible to do without them. The development of these tools is in some way limiting what can be achieved in the designing of an integrated circuit, and therefore essential to the further improvements. This thesis examines one of these tools, Siemens' Catapult High Level Synthesis tool. The purpose of this tool is to speed up the development of IC's by letting the designers specify in a higher level programming language than what has been done traditionally. Another use of this could be to unify code targeted at different platforms to reduce effort and code maintenance. Two fundamentally different target platforms that are compared in this thesis is FPGA and ASIC. These two are quite different from each other, but one of the key differences is that an ASIC can never be changed after production. It will contain exactly the functionality that was designed. An FPGA on the other hand, is designed to be highly configurable and can be reprogrammed freely to implement new functionality. ASICs are also faster and more efficient than FPGA, but on the backside also much more expensive and take longer to develop. Being able to use the same code for both of these platforms would mean that a design could be prototyped on FPGA before producing an ASIC, thus reducing the risk of bugs in the more expensive chip. This thesis uses a case study of a common digital signal processing block, a symmetric finite impulse response filter, to explore whether the same code can be efficiently used for both platforms.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Digital hardware design is a constant fight to keep improving. The industry has come a long way from the times of hand placing every single transistor of a design, much accredited to the help of computer tools. As costumers demands are always increasing for faster computers and power-efficient gadgets of all kinds, the efficiency of the engineering must also increase. Hardware design is a very costly endeavour. The more time and resources that can be saved here, the better, keeping in mind that performance of the design doesn't diminish.

As mentioned, the key to this is computer aided design. A new paradigm in these tools is based on the concept of high level synthesis. These tools aim to synthesize fully functioning digital circuits from a high level programming language such as C or C++. Lots of research, both academic and in industry, is underway to develop this technique. There are many successful examples, one being a complete graphics processing application being synthesized already in 2011.[1] This thesis will investigate a specific tools' capabilities of synthesizing digital filters for both ASIC and FPGA targets.

The thesis is conducted in cooperation with Ericsson. Ericsson is at the forefront of telecommunications and 5G. Both ASICs (Application Specific Integrated Circuit) and FPGAs (Field Programmable Gate Array) are used in different parts of the infrastructure so it would be useful to be able to maintain one common codebase for both. The symmetric finite impulse response filter (SFIR) is a often utilised component when it comes to frequency manipulation.[2]

The goal of this project is to develop an SFIR filter in C++ code and synthesize it with Mentor's Catapult HLS targeting both ASIC and FPGA.[3] An evaluation will be done to see if the design can be implemented for both targets with a good result in terms of area, components used, and allover implementation structure. Other important points of interest are the complexity of developing, debugging and maintaining the code and also ease of use and flexibility of the tool.

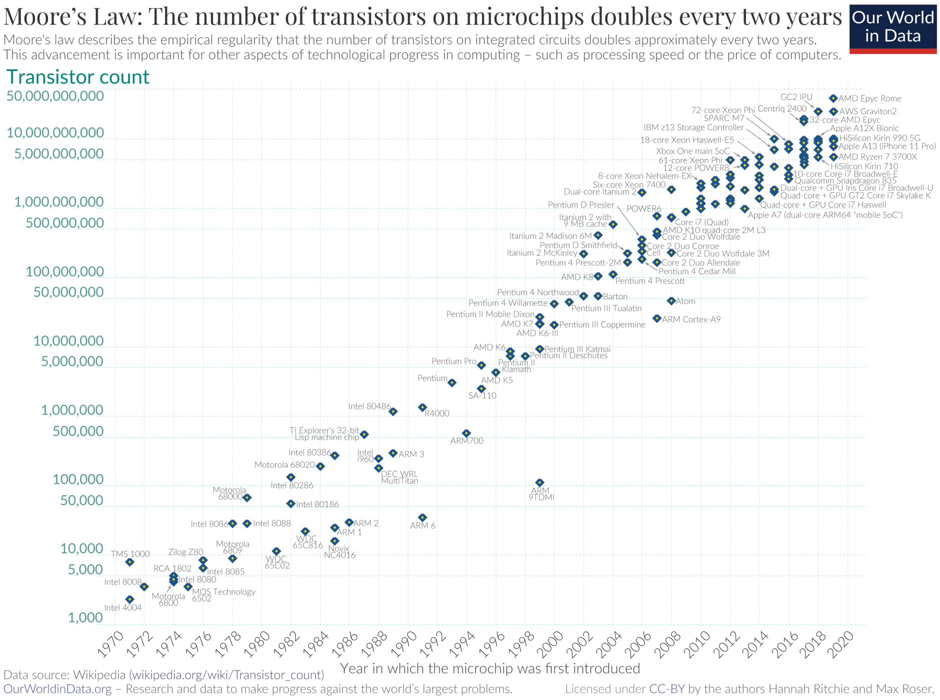
Chapter 2
Background

Digital systems have emerged and evolved rapidly in the last century. The first building block in digital systems was the vacuum tube. It was invented in 1904 and was prevalent during the first half of the century in applications such as radio, telephone, and radar. The first reprogrammable digital computer, ENIAC, was built with vacuum tubes and completed in 1945. This computer was indeed very computationally powerful at the time, but this came at a steep cost. ENIAC weighed over 27 000 kgs, occupied 167m^2, and consumed 150 kW of power.

Figure 2.1: Vacuum tubes, the predecessor to transistors
[4]

With the invention of the transistor in 1947, vacuum tubes were soon obsolete. Transistors could provide the same functionality with smaller size and less energy consumption. The further development of the MOSFET (transistor) made it possible to build integrated circuits with a high density of transistors. The digital systems have been continually growing in transistor count. As formulated by Gordon Moore, Moore's law states that the transistor count on a single chip doubles about every two years. As seen in figure 2.2 this projection holds up well. At some point it is of course impossible to place such a number of transistors by hand, and so computer aided design tools has been developed in parallell. Electronic design automation (EDA) is a whole separate business area.

The currently most used way of designing digital systems is by using a chain of EDA tools. The input is often a hardware description language, such as Verilog or VHDL. These are used to describe the structure and behaviour of the system



Moore’s Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world’s largest problems.

Figure 2.2: Graph showing Moore’s Law
[5]

at the register-transfer level (RTL). The RTL code is then synthesised by some tool into a netlist. The netlist contains all the information of the system such as the components and interconnects between them. The next steps are mapping to the desired technology, and physical layout and routing. Throughout the process simulation tools are used to make sure the design works as intended.

The development of digital systems is a complicated and time-consuming task that needs many engineers of different specializations. As time-to-market is important, there are constantly efforts to increase productivity. One bid to achieve increased productivity is the concept of High-level Synthesis (HLS) which will be the topic of this thesis. HLS aims to lift the abstraction level from describing the system at the register-transfer level to the algorithmic level, using languages such as C and C++. By describing a system at the algorithm level, more implementation details are removed from the designer’s hands, instead being implemented by an EDA tool. This lets engineers focus on system level decisions such as architecture and functionality instead of details. Research into HLS started as early as the 1970s, but it was not commercially viable for a long time[6]. It is now starting to become so.

This thesis will evaluate using pragmas in Catapult HLS to easily make the same code applicable for both ASIC and FPGA platforms. This will be done by implementing an example module using HLS, and following the design flow. Different pragmas will be used to try to influence the generated RTL.

This section aims to cover some concepts in hardware design to help the unfamiliar reader in understanding the topics, and provide technical specifics to the more familiar readers.

2.1 Integrated circuits

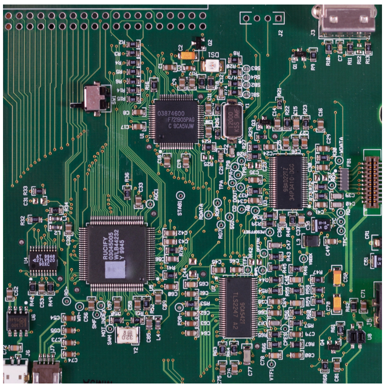


Figure 2.3: Several integrated circuits on a printed circuit board [7]

Hardware design has moved from discrete components to integrated circuits. This means that several billions of transistors are manufactured on the same chip using modern technology. This is possible because the abstraction level has been raised with the help of computer aided tools. Instead of designing at transistor level, the current status quo is designing at the register-transfer level. As mentioned in the previous section, lifting the abstraction level like this removes some details from the control of the designer, and in return facilitates a higher level of the design. Meaning higher transistor counts, which in turn give more complex integrated circuits. Integrated circuits really outperform their discrete counterparts in performance. The performance is better because of the small size of the transistors which give them fast switching times and shorter interconnects, while also consuming less power. Another advantage is that the complexity of circuits that can be achieved with integration is wildly higher than what would be possible with discrete components.[8] There are two main classes of integrated circuits, ASIC and FPGA. They will be described in the following sections.

2.1.1 ASIC

ASIC stands for Application Specific Integrated Circuit. It is a circuit that is designed for one specific purpose. It isn't flexible like the FPGA, rather it's functionality is permanent. The advantage of this is that an ASIC can be heavily optimized for its single functionality to achieve faster, smaller, and more power efficient performance than one could get from an FPGA. ASICS are implemented using a HDL just like an FPGA, but the backend implementation differs.

2.1.2 FPGA

FPGA stands for Field Programmable Gate Array. FPGAs are ready-made circuits that can be bought from a number of vendors, for example Xilinx and Intel. An FPGA is a reprogrammable integrated circuit.

The main building blocks are configurable logic blocks (CLB), interconnects and I/O. Since the demand for FPGAs have been very high and the development focused on what customers want, FPGAs have evolved to include also more specialised blocks apart from these basic building blocks. Some examples of this are on-chip memory and DSP-blocks.[9]

As the name suggests, FPGAs are field programmable. This means that the circuit can be re-programmed at any time to implement a different functionality. This is possible because of the CLBs and the programmable interconnects.

2.1.3 Configurable Logic Block

CLB, or configurable logic blocks, make up the heart of the FPGA. The CLBs are interconnected and programmed to form the logic and mathematical functions that are implemented. They are constructed in a generic way in order to allow them to implement a large range of functions.

The internal structure of a CLB consist of a LUT(look-up table), a d-flop, and a multiplexer. A d-flop is the smallest memory element available on the FPGA. In its simplest form it is capable of saving one bit in a stable state. The multiplexer is used to pass by the flip flop if that is necessary. The LUT actually saves and contains all possible outputs for all possible inputs for the given function. The operation of the LUT is fast since it is not actually calculating the result, but referencing it.[9]

2.1.4 Interconnects

Interconnects are an important part of the FPGA. This is the routing network that connects together different parts of the board to enable the correct functionality. There are both short and long interconnects. The CLBs are connected directly by short interconnects to other CLBs that are in the direct vicinity. They are also connected to the switching matrix that routes the long interconnects. All interconnects can be modified as they are controlled and switched by transistors. This gives a great flexibility in the circuit. The long interconnects can be used to connect parts that are physically distanced, and also to implement internal buses.[10] The interconnect delay is a critical factor of the performance of an FPGA. The propagation delay depends on the physical distance between the connected blocks as well as the number of switch matrices crossed. Therefore the assignment of functions to resources on the FPGA is a very important step as it can dramatically change the performance and timing metrics. The placement is usually done sufficiently well by software, but in some cases it may have to be aided by the designer to get the right layout, especially if resources are scarce. To avoid timing issues, the clock signal usually has dedicated interconnect lines that cover all or most of the board. [9]

2.1.5 I/O

At the boundary of the FPGA fabric is the input and output ports. The interconnects connect to the IO to make it possible to communicate with the outside of the FPGA. They usually have buffers. [9]

2.1.6 DSP block

The basic blocks of the FPGA logic, CLBs, is built to implement a large range of functions. Because it is so generic, it can't be optimal for any one functionality. Because of consumer demand to add more specialized parts of hardware, many FPGAs contain specialized blocks for handling digital signal processing, or DSP. While it is entirely possible to implement these DSP functions on the regular fabric, implementing them on the dedicated DSP block lead to better utilization, performance, and power consumption. DSP blocks can vary some in implementation but usually consists of a dedicated multiplier and an accumulator. Some can also have a pre-adder. This thesis will focus on the Xilinx UltraScale architecture, which includes a pre-adder. The pre-adder is used to save resources when implementing symmetric filters. The multiplier is for multiplying with the coefficient, and the accumulator is used to chain multiple DSP slices together and accumulate the values from the taps. The basic structure can be seen in Xilinx documentation at [11].

2.1.7 Development time

FPGAs are commonly programmed using a hardware description language (HDL) such as VHDL or Verilog. The backend is usually handled by a tool, like Xilinx Vivado. The backend in FPGA mainly involves mapping to the available resources on the platform. The development cycle of FPGA systems is considerably shorter than that of an ASIC, measured in terms of weeks or months while asic projects commonly take 18-24 months[12]. This makes FPGAs suitable for not only projects but also prototyping of new functionality.

2.2 Design abstraction levels

An abstraction is a model of a system in the desired amount of detail. A higher level of abstraction means only overall functionality is taken into account. A lower level means more implementation details are visible, which means a more complex but also more accurate view.[13]

There are multiple different layers of abstraction in hardware design. The four main levels are transistor, gate, RTL, and algorithm. This is not a definitive list, as abstractions can be arbitrarily decided upon. There are also possible intermediates between the layers. However, these are the most important ones and they give a clear picture of the concept of abstraction levels.[14]

The lowest abstraction level is the transistor level. Building blocks at this level are transistors, and also passive components such as resistors. The input-output characteristics of the transistor can be described by differential equations of a current-voltage diagram. At this level, the transistor is viewed as an analog circuit element. This entails that all signals are time-varying and continuous.[14]

The next step is the gate level. In this level, no notice is paid to the individual transistors, it is only the function of the overall gate that is important. The physical aspect has been abstracted away. The basic building blocks are simple gates such as AND, OR, XOR, and multiplexers. At this level, the signal values are assigned 1 or 0 based on if they are above respectively below certain thresholds. That means it is now considered in the digital domain. The input-output characteristics is described by boolean functions.[14]

The next level is the register-transfer level, RTL. In this layer components such as multipliers, adders, and flip-flops can be utilized. Again, these components are made up of gates, but that is not important at this level. Only the behaviour of the block is. At this level, signals are often grouped together and interpreted as data types such as unsigned integers, or system state. An important feature at the register transfer level is the system clock. The clock signal is common to the blocks in this level and controls when signals are sampled and held in storage component. The existence of this clock signal lets the design be measured in clock cycles instead of propagation delays.[14]

The algorithm level is used to describe the overall functionality of the block or system. This is often used to compare different implementations to see which fulfill the specification. The chosen algorithm will then be further implemented in the lower abstraction layers. This further implementation is possible to do by hand, but this is where the benefit of the EDA-tools come in. In a traditional RTL-flow, the algorithm description is translated by hand into RTL, and the RTL is then passed into the tool for further synthesizing.[13] In an HLS-flow on the other hand, it may be sufficient to only change the data type into bit-accurate types and then let the tool handle the rest.

2.2.1 High Level Synthesis

High level synthesis denotes the process of converting an algorithmic description of a system into an RTL architecture. Research into HLS can be traced back to the 1970's, but this was purely research-oriented and had very little impact on the industry. This was a time when the extent of the commercial EDA products were limited to physical layout. Research continued throughout the years and the turn of the century saw several EDA companies offer their own versions of commercial HLS tools. However, this second generation of tools was ultimately a failure. The tools simply weren't good enough to motivate designers to make the switch. RTL coding was the standard and the synthesis tools for RTL were still improving rapidly. The tools used a version of HDLs as input language, thus competing directly with traditional RTL, the results were hard to validate, and additionally some tools required synthesizing all the way down to gate-level, making it impossible to integrate into existing RTL-flows.

A third generation of tools emerged in the early 2000s. The vendors now seem to have found the right market as well as improved the tools to be competitive to RTL. The tools are more specific to certain domains were they perform reasonably well, with many focusing on data flow, DSP, and algorithm design. Another thing

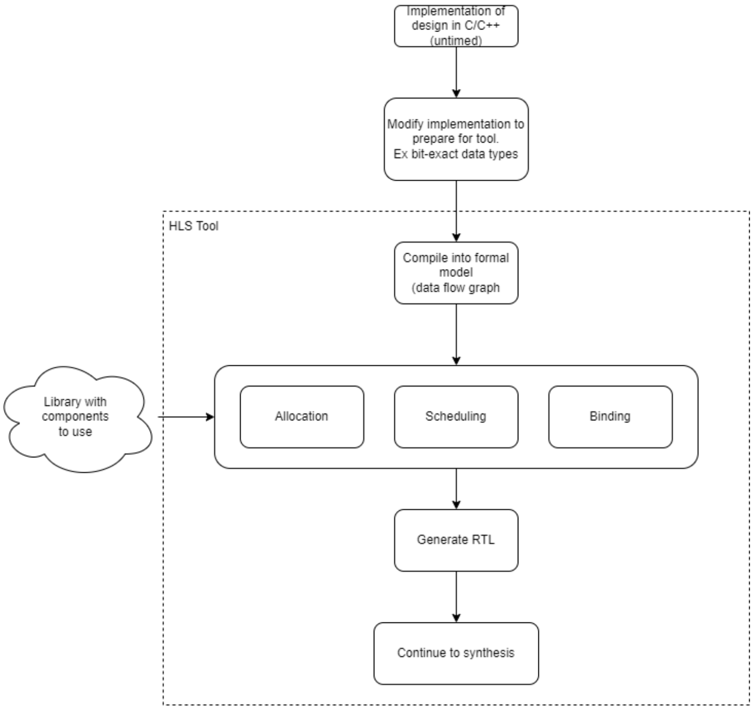

Implementation of design in C/C++ (untimed)
Modify implementation to prepare for tool. Ex bit-exact data types
HLS Tool
Compile into formal model (data flow graph
Library with components to use
Allocation
Scheduling
Binding
Generate RTL
Continue to synthesis

Figure 2.4: Description of the design steps in high level synthesis.

supporting the success of the current generation is the usage of FPGAs. With their differences from ASICs, they are perfect for quick prototyping. Combined with HLS, they provide a very fast way of testing out algorithms on hardware. [6]

The design process starts with an untimed software model. This can be written in any high level software languague and is used to find and test the appropriate algorithm to be used for the system. In a perfect scenario, this code would be the direct input to the HLS tool, but in reality some changes need to be made. One being that data types must be converted into bit-accurate data types of a specific length, and with sufficient accuracy. HLS tools then transforms this untimed model into a fully timed implementation consisting of datapath, controller, memory, interfaces, and any other functionality specified. This is done through a series of steps shown in figure 2.4 and described in the following section.

In the first step the specification is compiled into a formal model. This step can include some optimizations such as removal of dead code, elimination of false dependencies, constant folding and loop transformations. The output of the compilation is typically a data flow graph. The next steps are allocation, scheduling and binding. These steps are interdependent so in order to successfully optimize, all these steps should be optimized together. However, they are usually executed sequentially to manage the computational complexity. For these steps a library is used. The library contains information about the available resources that can be used, along with information about their area and delay. Allocation decides the type and number of resources needed to satisfy the constraints of the design. Components are picked from the library. The components can for example be functional units, storage elements, and connectivity components. In the scheduling steps, all operations are assigned to one or more clock cycles. Using the characterization from the library, the scheduler knows how long the operations will take on the specified resource. For each operation, the operands need to be fetched from either storage elements or another functional unit, the operation needs to execute, and the result has to either be stored or propagated to another functional unit. Depending on the delay in this chain, the operation is scheduled for the needed amount of cycles. The operands don't have to pass through a storage element, they can be directly chained between functional units. If there are no data dependencies between operations, they can be scheduled in parallel, given there are enough resources to do so. In the binding phase variables that save values over multiple clock cycles must be bound to a storage element. The usage of storage elements can also be optimized by analysing the variables lifetime. If certain variables exist at different times, with no overlap, they can be bound to the same storage. Every operation also needs to be bound to a functional unit that can execute the specified operation. There is also connectivity binding, which makes sure all storage and functional units are connected in the appropriate way.

When these tasks are completed, the RTL code is generated using the information. [15]

2.2.2 Advantages

One of the main advantages of HLS is that it can reduce hardware design time. However, it can also reduce the verification time and facilitate power analysis flows. [15] Another advantage is in terms of faster design space exploration - it is possible and even easy to generate microarchitectures with different characteristics using the same behavioural description. [16] An advantage that stems from the higher abstraction level is that algorithm code is technology-independent. [15] With the source code being inherently untimed, a wide range of interfaces and architectures can be generated from the same source code, and it helps avoid error introduced by manual coding of such structures. Instead these are generated by synthesis directives which the designer control. [15]

2.2.3 Limitations

The design space exploration for complex systems quickly become unsustainable as the runtime of the HLS tools increase with the size of the set of possible solutions. However, this problem can be overcome using clever algortihms that quickly find the important possible solutions.[17] One limitation with HLS is that the code must be written in a specific style in order to infer the suited hardware. It can be easy to slip into the software mindset when one is writing C-code, but it is still describing hardware. Without the correct specific style, the tools may still translate into hardware, but it will be bloated and poor performing. Traditional software code is inherently sequential, while hardware is parallel. This means synchronisation is a much bigger task, and the complexity can lead to problems. The RTL code that is the output of the HLS tool is not human-readable. This shouldn't be a problem since the point of HLS is to cut out the RTL step, but if verification shows any bugs within the code, it can be near impossible to find and fix the cause in the RTL. It can also be very difficult to find the corresponding source of the bug in the C-code. [18]

Chapter 3

HLS Design and Optimization Methods

The goal of this master thesis consist of developing a symmetric FIR filter of variable length and data-types, that is designed to be used both in ASIC and FPGA applications. The idea behind this is that one could then use the same code base for projects in both technologies, thus saving time and effort to remake the entire design. Instead only minor tweaks might be necessary. The symmetric FIR filter is an interesting component to use for this since it has clear differences in how it is best implemented depending on if it will be for ASIC or for FPGA.

3.1 SFIR

FIR stands for finite impulse response. It is a type of filter where the impulse response is of finite duration. This means that the output of the filter will settle to zero at some time after the input has settled to zero. The opposite of FIR is IIR, infinite impulse response filters. IIRs contain feedback, which FIRs does not.

The output of the FIR filter is calculated as the sum of the most recent inputs weighted by the coefficients, as described by the following formula:

$$y_n = h_0 x_n + h_1 x_{(n-1)} + ... + h_N x_{n-N} = \Sigma_{i=0}^{N} h_i \cdot x_{n-i} \tag{3.1}$$

A regular direct form finite impulse response filter with 6 taps is shown in figure 3.1. Here it can be seen that the 5 most recent inputs are saved in the delay line, and each input is in turn multiplied by the coefficients of the filter and lastly summed up to produce the output.

In some applications, the coefficients of the filter are symmetrical. That is, in the example of the six-tap filter, $h_0 = h_5, h_1 = h_4$, and $h_2 = h_3$. In this case, the filter can instead be implemented as shown in figure 3.2.

An N-tap direct form FIR filter will have N multipliers and N-1 adders. In the symmetrical configuration, the number of multipliers is reduced to N/2, and the amount of adders are still N-1. In the 6-tap filter, the number of adders stay the same, but the number of multipliers have been reduced from six to three. This is a great improvement since multipliers are very costly to implement in hardware, much more so than adders.

The structure shown in figure 3.2 is a good way to implement in an ASIC design, but in an FPGA if would be better to utilize the DSP-slices. Since these

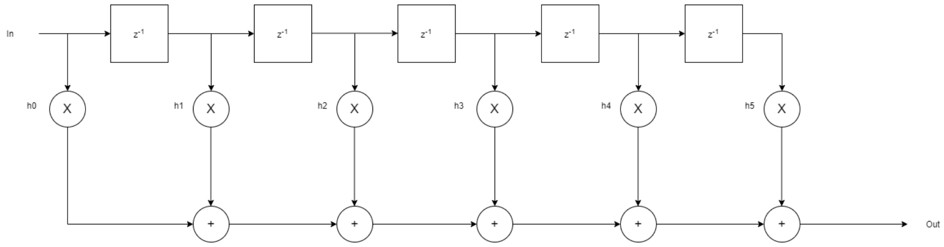


Figure 3.1: Block diagram of a direct form FIR

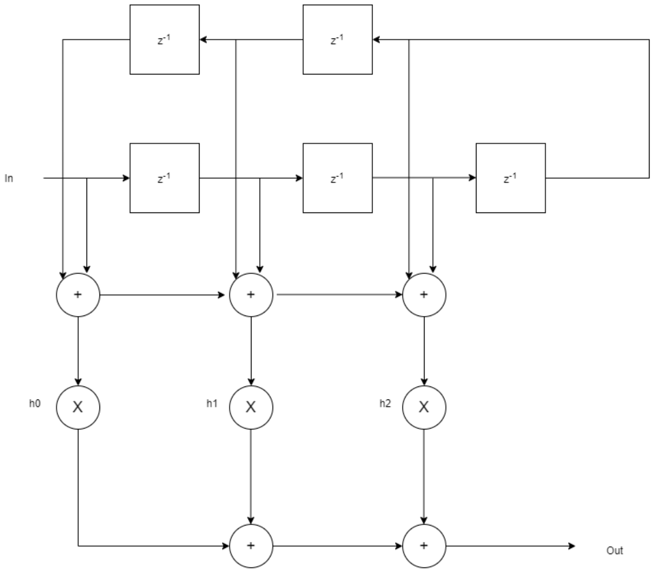


Figure 3.2: Block diagram of a symmetric FIR

have dedicated adders and multipliers it can save a lot of the other fabric resources. A figure showing this configuration can be seen here: [11].

3.2 HLS Flow

Just like in more traditional chip development flow, the first step is the specification. This is usually written in a high-level language such as python or C++. This specification of the design would usually converted to RTL code, by writing it in a HDL such as Verilog or VHDL. For HLS, under the right circumstances the specification can be reworked slightly to be used as input for the next stage.

First of all it needs to be written in a language that the HLS tool support as input. This could be C++ or SystemC or many other languages depending on the tool.

Theoretically this is all that's necessary. The specification can be run through the tool that will produce usable RTL or something even further in the flow like a netlist. In reality, optimizations are needed. A first step will be to convert all variables to bit-exact types. Using an "int" may result in 64bits, but in reality only 10 bits might be needed for example. That needs to be specified to get a good result.

The design source file can be loaded into the tool, and a target library can be chosen. This is one aspect of the HLS flow that makes design space exploration easier. It's easy to change tech library to explore options.

Two commonly used techniques for further optimizations are loop unrolling and loop pipelining. These are both controlled through pragmas, or directives in the code.

3.2.1 Loop unrolling

Usually some loop unrolling is done. Consider a for loop in the high level code. It loops for a number of iterations and some code is executed in the loop body every time. By unrolling the loop, the number of iterations is decreased, and multiple iterations' worth of instructions are executed in the loop body. An example of this is a shift register. It is modeled by a loop, where each register gets its value from the previous register. However the data cannot be propagated one register per cycle, that is too slow. So by unrolling the loop completely, in each cycle all registers are assigned the value of the previous one. That's the shift register functioning as expected.

Loop unrolling can also be utilized in other contexts. It can be partially or fully unrolled. Depending on the application this needs to be decided what gives the best outcome. Fully unrolling a loop makes it possible to execute the entire set of iterations in one cycle, but this might not always be possible. For example if some part of the loop body has a dependency on previous iterations, or other signals that needs to be timed correctly. Unrolling a loop is done by replicating the hardware needed for an iteration a number of times depending on the loop unrolling factor. Thus, loop unrolling results in larger area.

Figure 3.3 shows an example of no unrolling, partial unrolling, and full unrolling of a loop. The image shows an operation that needs to be executed 4 times

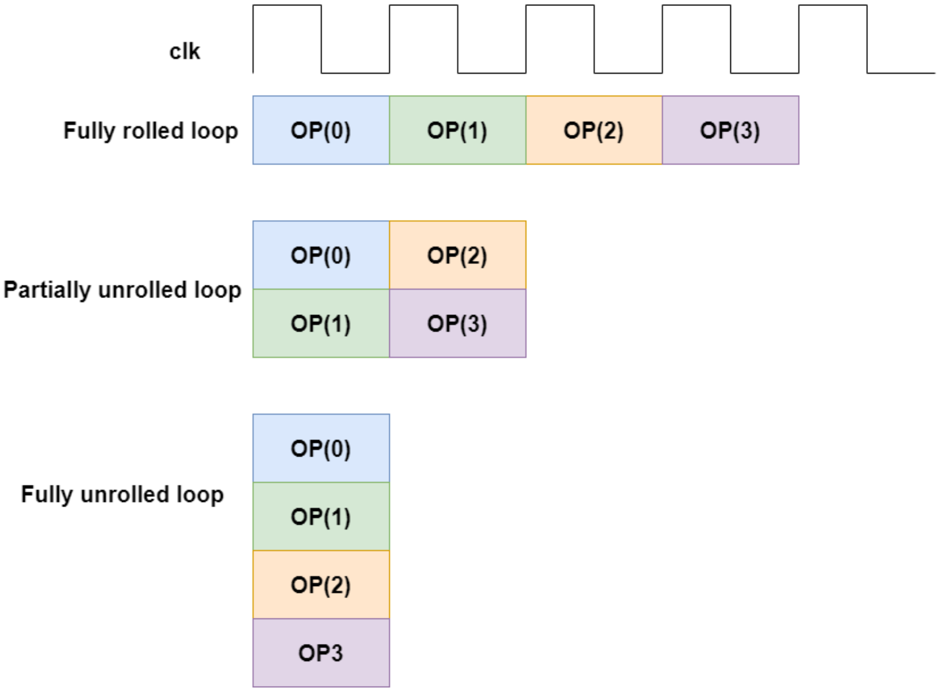


Figure 3.3: Loop unrolling

with different inputs, each operation taking one clock cycle. In the rolled loop this takes four iterations of the operation, meaning four clock cycles. In each iteration, the operation is executed. Since only one operation is executed at a time, the hardware for this operation is only needed once.

In the partially unrolled loop, OP(0) and OP(1) are performed at the same time in the first iteration, and in the second iteration OP(2) and OP(3) is performed. This then requires two times the hardware required for the operation, but it only takes two clock cycles.

In the fully unrolled loop, all 4 operations are performed at the same time. The loop now only takes one iteration to complete, however it requires four times the hardware.

3.2.2 Pipelining

An importand metric for a function or loop is the number of clock cycles before it can accept new input [19]. To decrease that number, loop pipelining can be done. As seen in figure 3.4, a loop containing 3 operations is run three times. Without any pipelining, the execution will look like the top image. When all three operations of the first iteration are finished, the second iteration can begin. The initiation interval (cycles between accepting input) is 3.

If this loop is fully pipelined as in the bottom image, however, it will be able to accept a new input every clock cycle.

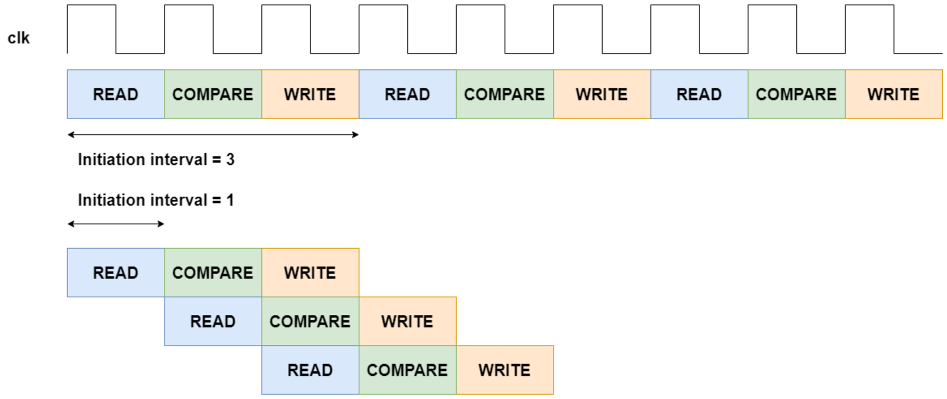


Figure 3.4: Pipelining

Pipelining is done by inserting registers between the operation, and it increases the overall throughput.

Chapter 4

Results

In this section the results are presented. The end result of the project consists of one piece of code that can be optimized for FPGA implementation or optimized for ASIC implementation. The FPGA-optimized version was implemented all the way through placing and routing in Vivado, but it was not uploaded onto a real FPGA. The metrics are taken from the reports in Vivado post PnR. For the ASIC-optimized version the metrics are taken from Catapult after the initial synthesis result. The reason behind this is that the further implementation for ASIC requires additional tools and considerable effort, and is not within the scope of this thesis.

The results were achieved by developing the code in several steps. The first step was making a simple FIR consisting of a shift register variable and a for loop for the calculations. This code was verified to be a functionally correct implementation of a FIR filter with some additional software test cases.

Starting from this code some modifications were made in Catapult. Namely, the main function was pipelined to achieve a functioning pipelining of successive values. Without this the resulting circuit would only be able to process one value at a time no matter how much the inner loops were pipelined.

The next step was unrolling the loop describing the shift register. This was done to aid implementation and help the tool implement it in an efficient way, i.e. as an actual shift register of the required bit width.

One last modification at this point was to increase the bit widths of the data types. For smaller bit widths, for example data types of 5 bits, Catapult wouldn't infer DSP even if everything else hinted at it. The exact limit for when DSPs are used was not determined, but a bit width of 16 for the input and 13 for the coefficients did suffice when targeting DSP generation.

Both versions of the code (ASIC/FPGA-optimized) were run for both targets. This was to investigate whether the optimizations made a difference to the end result.

4.1 Verilog vs VHDL

A major difference was found between using VHDL or Verilog as the target language of the implementation. With the exact same input C++-code, the two languages resulted in drastically different implementations. The schematics can be seen in figure 4.1 for Verilog as target language and figure 4.2 for VHDL.

	CLB as LUT	CLB as register	CARRY8	CLB	LUT as logic	LUT as memory	DSP
VHDL	472	452	56	63	448	24	16
Verilog	18	36	0	9	2	16	16
Small bit widths	558	390	52	109	558	0	0

Table 4.1: Summary of components used for verilog, VHDL, and verilog with small bit widths

	CLB as LUT	CLB as register	CARRY8	CLB	LUT as logic	LUT as memory	DSP
FPGA	18	36	0	9	2	16	16
ASIC	8	160	0	19	0	8	10

Table 4.2: Differently optimized code implemented on FPGA

These were implemented with FPGA as target. The Verilog version is the desired outcome consisting of chained DSP slices, as well as some supporting logic implemented in the CLBs. The VHDL version also has DSP slices but they are not cascaded in the correct way and there is a considerable amount of extra logic implemented in the CLBs.

Table 4.1 summarises the difference between the Verilog and VHDL targets. It also includes an implementation with smaller bit widths, 5 bits for coefficient and 8 for input values, also with Verilog as target language. These widths did not result in using DSP slices. The larger size was 16 bits for input data and 13 bits for coefficients. The VHDL and Verilog implementations both use 16 DSP slices, but the VHDL implementation uses considerably more of all the other components. The small bit width version has no DSP slices, and also more of the other components than the Verilog of larger bit widths.

4.2 FPGA target

This section describes the differences of the ASIC- and FPGA-optimised version when FPGA was the target. The FPGA version is shown in figure 4.1 and the ASIC version can be seen in figure 4.3. The FPGA-optimized version was the Verilog version described in section 4.1. This implementation contains cascaded DSP-slices and a minimal amount of supporting logic implemented in CLB's. The ASIC-optimized code implemented on FPGA does contain DSP slices, but they are not cascaded. It also contains much more logic implemented in CLB's, resulting in a implementation but not an optimal one. Table 4.2 summarizes the difference between the two.

4.3 ASIC target

This section will describe the difference between implementing the ASIC-optimized version and the FPGA-optimized version with ASIC as the target.

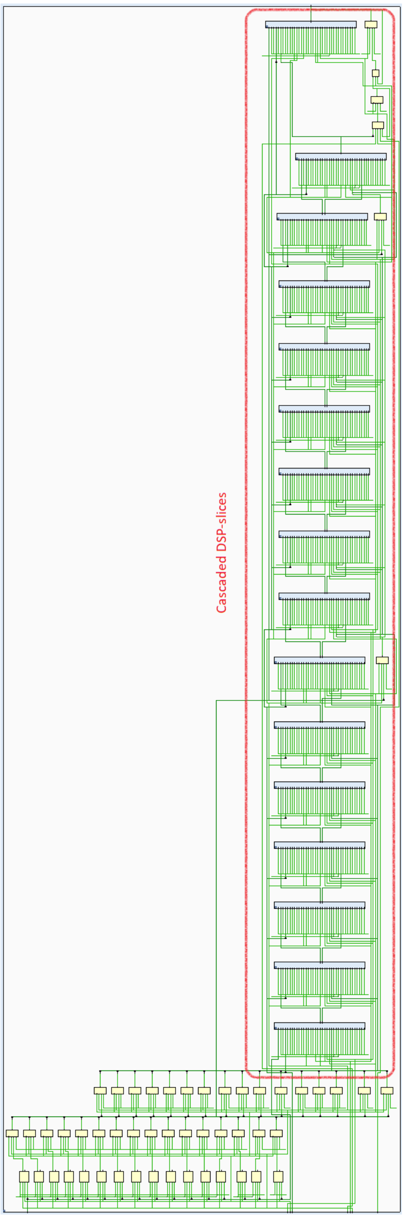


Figure 4.1: SFIR using verilog as target language

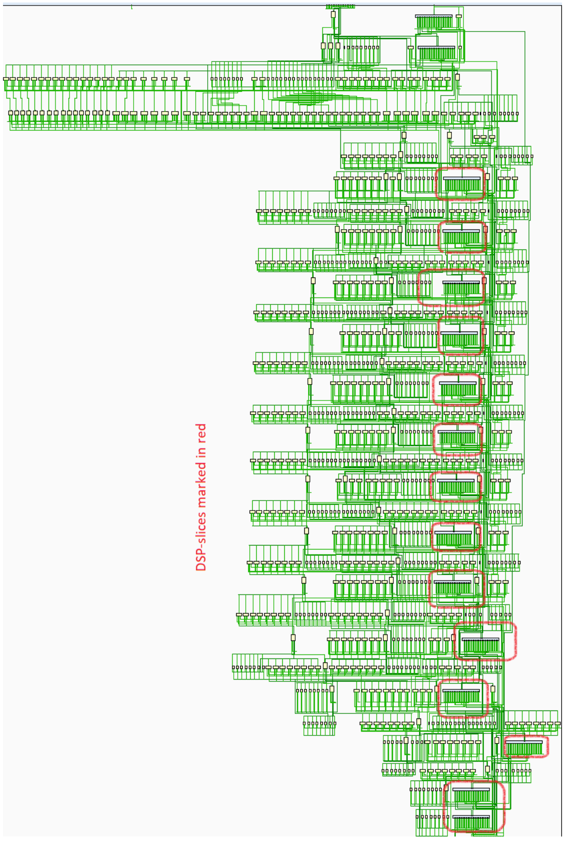


Figure 4.2: SFIR using VHDL as target language

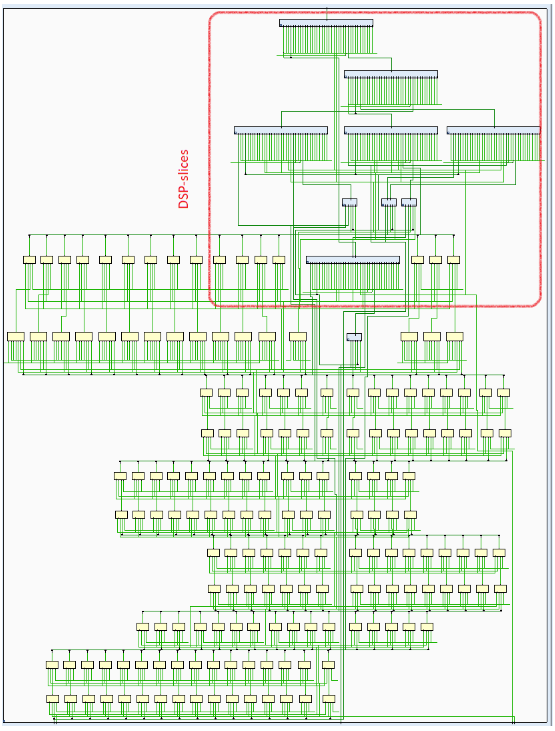


Figure 4.3: ASIC optimised code implemented on FPGA

Component name	Area score	# used post assign
Lib: ccs_ioport		
ccs_in(1,16)	0	1
ccs_in(2, 104)	0	1
ccs_in(6,1)	0	1
Lib: mgc_ioport		
mgc_out_dreg(3,32)	0	1
Lib: nangate-45nm_beh		
mgc_add(16,1,16,1,17,4)	69.617	8
mgc_add(30,1,30,1,31,4)	129.655	4
mgc_add(31,1,31,1,32,3)	211.014	2
mgc_add(32,0,32,0,32,3)	210.504	1
mgc_and(1,2,1)	1.064	1
mgc_mul(13,1,17,1,30,6)	1087.920	8
mgc_reg_pos(16,0,0,1,1,1,1,4)	102.144	15
mgc_reg_pos(32,0,0,1,1,1,1,4)	204.288	1
Total area after assignment	12148.958	

Table 4.3: Bill of materials for ASIC-optimized version implemented on ASIC platform

4.3.1 ASIC-optimized version

The schematic can be seen in figure 4.4. Note that this shows only the primary logic of the filter. Some details were left out for clarity. It was implemented with the Oasys RTL nangate sample library provided by Catapult.

The implementation contains a shift register of length 15. This represents the delay line necessary for this filter. There are eight adders used for "pre-adding". This is because of the symmetry of the filter coefficients, two inputs are to be multiplied by the same coefficient. After the adders there are multipliers which carry out the multiplication by the coefficients. Lastly there is a balanced adder tree for adding up the branches of the filter and an output register.

In table 4.3 all components used and their area scores are listed. There are 3 input ports which are register inputs. There is one output which corresponds to the final result of the filter. Next there are 4 different adders. The difference between the adders is that they have different bit widths. In total 15 adders are used. The 8 multipliers are all of the same size, and so are the 15 registers that form the shift register. Lastly the final register is the output register. The total area score after assignment is 12148.958.

4.3.2 FPGA-optimized version

In this section the FPGA-optimized version implemented on ASIC-platform will be presented. A schematic overview is shown in figure 4.5. This schematic only

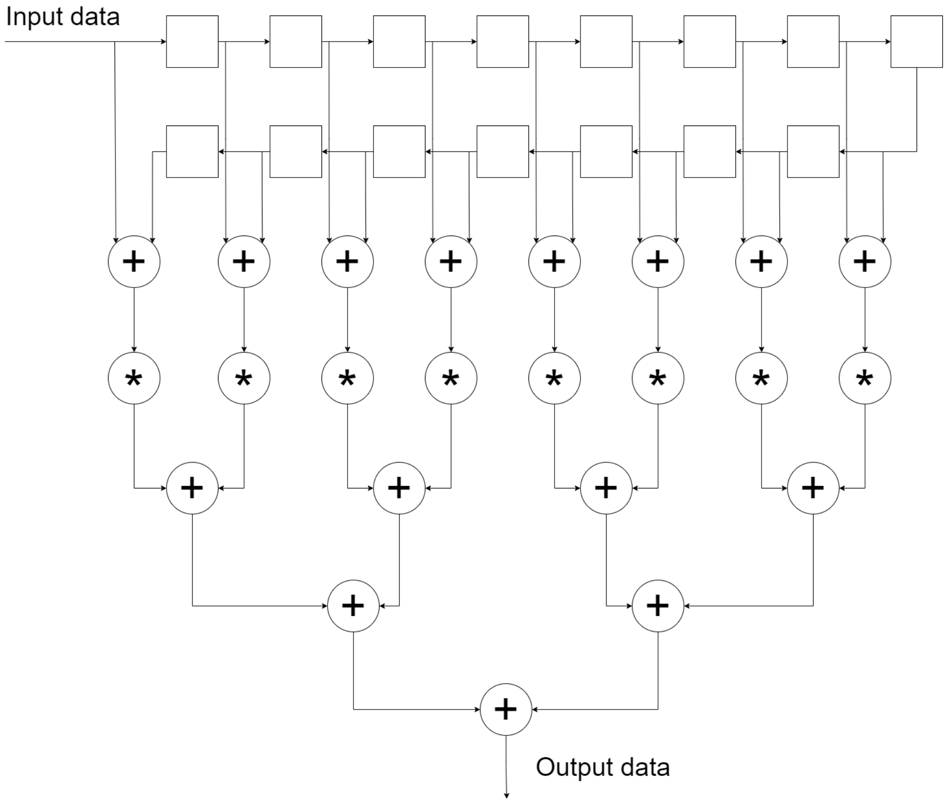


Figure 4.4: The schematic of the ASIC-optimized code implemented on ASIC

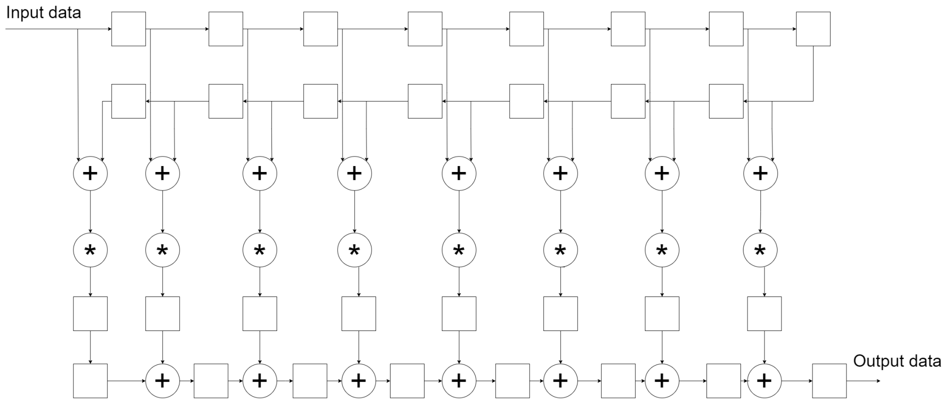


Figure 4.5: Schematic of FPGA-optimized code implemented as ASIC

includes the primary logic of the filter. Some details were left out for clarity. This implementation is very similar to the ASIC-optimized one. The difference is in the output stage. Instead of an adder tree, there is a plain accumulative adding with extra registers.

Table 4.4 shows the components used in this implementation. There are four registered inputs and one output. There are two differently sized adders and one type of multiplier. The are a total of 16 adders and 8 multipliers. There are 19 registers in addition to the 15 registers of the shift register, totaling 34 registers. The total area score of the implementation is 15266.221.

4.3.3 Comparison

Table 4.5 summarizes the differences between the ASIC- and FPGA-optimized versions implemented on ASIC-platform. The input and output ports and and-gates have been left out as they are of minor importance in this scope. The biggest difference is in the registers. The FPGA version has slightly more than double the amount of registers, one additional adder and the same number of multipliers. The difference in area score is about 25%.

Component name Lib: ccs_ioport	Area score	# used post assign
ccs_in(1,16)	0.000	1
ccs_in(2,104)	0.000	1
ccs_in(4,32)	0.000	1
ccs_in(7,1)	0.000	1
Lib: mgc_ioport		
mgc_out_dreg(3,32)	0.000	1
Lib: nangate-45nm_beh		
mgc_add(16,1,16,1,17,4)	69.617	8
mgc_add(32,0,30,1,32,4)	135.982	8
mgc_and(1,2,1)	1.064	1
mgc_and(1,3,1)	1.330	1
mgc_mul(13,1,17,1,30,6)	1087.920	8
mgc_reg_pos(1,0,0,1,1,1,1,4)	6.384	2
mgc_reg_pos(16,0,0,1,1,1,1,4)	102.144	15
mgc_reg_pos(30,0,0,1,1,1,1,4)	191.520	8
mgc_reg_pos(32,0,0,1,1,1,1,4)	204.288	9
Total area after assignment	15266.221	

Table 4.4: Bill of materials for FPGA-optimized version implemented on ASIC platform

Component	# in ASIC	# in FPGA	diff
Register	16	34	+112.5%
Adder	15	16	+6.7%
Multiplier	8	8	+0%
Area score	12148.958	15266.221	+25.7%

Table 4.5: Summary of the most important differences between ASIC- and FPGA-optimized version implemented on ASIC-platform.

Chapter 5

Discussion

This section will aim to summarize and analyse the findings presented in the previous section. The analysis will focus on the difference in implementation depending on target platform and the resulting architectures.

5.1 The effect of target language

As was seen in the results section, the target language had a major impact on the resulting architecture. Using VHDL as target language resulted in a jumbled mess. As this author was mostly familiar with VHDL beforehand, that was the chosen language. Because of this, a lot of time was spent debugging and trying to figure out why the results were far from the intended. This is obviously a limitation of the tool itself.

However, since the goal of using HLS is a fully synthesized design, the HDL code is merely an intermediate stage which the designers would rather not have to analyse at all. After all, one of the main selling points of HLS is the move away from complex HDL languages. With this in mind, whether the target language is Verilog or VHDL or something entirely different, does not really hold any importance.

It is important to be aware of this effect. There may be projects or specific modules which benefit from using VHDL instead of Verilog. In the end this is one more level of complexity that the intermediate language has such a big effect on the result.

5.2 From C-code to gate-level

The experienced reader might notice that the resulting code doesn't quite look like pure C-code. One necessary modification is using bit-exact types for all variables. Where generic C-code may use for example an int, that doesn't work when writing for hardware. An int is not a set number of bits. It's often representative of 32 bits, but that is not always true as it can vary between different architectures. In hardware, every bit corresponds to something physical, like a wire. And correspondingly an addition of to values requires more gates for more bits. Therefore it is quite important to use as few bits as necessary when designing hardware since more wires and transistors equals more area, higher cost, and more power consumed.

There are also pragmas inserted into the code. Pragmas are directives to the compiler on how to handle certain constructs. For example the pragma hls_unroll yes tells the compiler that the shift loop is should actually be implemented as a shift register. That means the shift operation through the register only takes 1 clock cycle. Without the pragma it would only perform one bits shifting each cycle, meaning a total of N cycles for N bits.

5.3 Platform-independent code

One big hope for HLS is the possibility to use the same code for developing both ASIC and FPGA. One aspect of this point is that ASICs are produced by a lengthy and very expensive process, wherein any minor undetected fault or bug can possibly result in respins of the chip, costing additional millions of dollars. FPGAs on the other hand are much more flexible and can be reprogrammed at will. If the same code could be used for both, prototyping on an FPGA could be more widely used to show both proof of concept and actually, that the design is functional. Verification would of course still be needed, but having the exact same code functioning on an FPGA would increase the reliability and decrease the risk for ASIC development.

However, as can be seen in the code for the filter, it is not as simple as that. Different versions has to be used, together with a switch to tell which is the intended platform. Of course, if the only motivation for having it implemented on FPGA is to show a working design, it might not need to be completely optimized. On the other hand, using the ASIC-optimized version of the code on FPGA cause serious bloating of the components used, and if the intention is to prototype an entire complex design, an incredibly big FPGA, or multiple, might be necessary, thus offsetting the improvement.

And if the intention is to use the same design on FPGA as well as ASIC, the architecture will need to be optimized for both cases. Then every component of the design will need to be evaluated for both platforms anyway, to make sure the intended structure results in both cases. It is true that some of the code can be common between the two, but one needs to make sure which parts cannot, and then work on those separately. This further requires knowledge in both ASIC and FPGA to make the best design decisions.

5.4 Verification

The only verification done for this project was functionally verifying the C-code and cross-referencing the outputted values with those from the ASIC and FPGA designs. In reality, there is much more to be done to consider a design fully verified. This could be something for future projects to explore.

Chapter 6

Conclusions

HLS holds a very tempting promise of decreasing the time and cost of producing integrated circuits. Even though the concept has been around for decades, it has not yet reached its full potential. More development and more research needs to be done before HLS can be widely adopted in industry. There needs to be sufficient evidence that the swap will be beneficial, since learning new technologies and ways of working will in and of itself take time. For specific projects using highly specialized tools for the very purpose, it may be beneficial, but as a general tool for any project I find it lacking still. It could be possible to start using HLS on a small scale at first, by evaluating it for a subset of modules in a system that seem promising for use with HLS. By doing the evaluation, you would gain a lot of knowledge about the tools and techniques, and simultaneously start the shift from RTL to HLS. It is not necessary to switch over the entire project at once, since the HLS-generated code should be possible to integrate into hand-written code.

6.1 Guidelines for HLS

The following section gives a list of some tips and tricks for future assessments of HLS.

- Use bit-exact data types
- Don't be afraid to experiment with all settings, even ones that seem obscure.
- In Catapult, the main function needs to be pipelined to achieve II=1.
- A shift register loop needs to be fully unrolled to work as intended.
- The GUI is useful for testing new options or pragmas, but when a solution is found, these should be inserted into the code or the tcl script that runs the program.

Bibliography

[1] Thomas Bollaert. “High-Level Synthesis Walks the Talk: Synthesizing a Complete Graphics Processing Application”. In: *DVCon* (2011). URL: `https://dvcon-proceedings.org/document/high-level-synthesis-walks-the-talk-synthesizing-a-complete-graphics-processing-application/`.

[2] JULIUS O. SMITH III. “Introduction to digital filters With audio applications”. In: 2007. Chap. Symmetric Linear-Phase Filters. URL: `https://www.dsprelated.com/freebooks/filters/Symmetric_Linear_Phase_Filters.html`.

[3] Siemens. URL: `https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/`.

[4] Stefan Riepl (Quark48). *Vacuum tube.* This file is licensed under the Creative Commons Attribution-Share Alike 2.0 Germany license. 2022. URL: `https://en.wikipedia.org/wiki/Vacuum_tube`.

[5] Hannah Ritchie Max Roser. *Moore’s law.* This file is licensed under the Creative Commons Attribution 4.0 International license. Dec. 2022. URL: `https://en.wikipedia.org/wiki/Moore%5C%27s_law`.

[6] Grant Martin and Gary Smith. “High-Level Synthesis: Past, Present, and Future”. In: *IEEE Design Test of Computers* 26.4 (2009), pp. 18–25. DOI: `10.1109/MDT.2009.83`.

[7] Miguel Á. Padriñán. *Green circuit board.* URL: `https://www.pexels.com/photo/green-circuit-board-343457/`.

[8] Jan M. Rabaey. “1”. In: *Digital Integrated Circuits: A design perspective.* Prentice Hall, 1996, pp. 1–15.

[9] Juan Jose Rodriguez Andina, Eduardo de la Torre Arnaz, and Maria Dolores Valdes Peña. *FPGAs. [Elektronisk resurs] Fundamentals, Advanced Features, and Applications in Industrial Electronics.* CRC Press, 2017. ISBN: 9781439896990. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat07147a&AN=lub.6202673&site=eds-live&scope=site.

[10] Bob Zeidman. *FPGA programmable interconnect.* 2006. URL: https://www.eetimes.com/all-about-fpgas/#.

[11] Xilinx. *UltraScale Architecture DSP Slice User Guide.* 2020. URL: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[12] Sejal Singh. *FPGA VS ASIC Design [Comparison].* 2021. URL: https://www.logic-fruit.com/blog/fpga/fpga-vs-asic-design/.

[13] John Sanguinetti. "Abstraction and Standardization in Hardware Design". In: *IEEE Design Test of Computers* 29.2 (2012), pp. 8–13. DOI: 10.1109/MDT.2012.2184074.

[14] Pong P. Chu. *RTL hardware design using VHDL. [Elektronisk resurs] coding for efficiency, portability, and scalability.* John Wiley Sons, Inc., 2006. ISBN: 0471786411. URL: http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat07147a&AN=lub.5496260&site=eds-live&scope=site.

[15] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design Test of Computers* 26.4 (2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[16] Farah Naz Taher, Mostafa Kishani, and Benjamin Carrion Schafer. "Design and Optimization of Reliable Hardware Accelerators: Leveraging the Advantages of High-Level Synthesis". In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS).* 2018, pp. 232–235. DOI: 10.1109/IOLTS.2018.8474222.

[17] Guanwen Zhong et al. "Design space exploration of multiple loops on FPGAs using high level synthesis". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD).* 2014, pp. 456–463. DOI: 10.1109/ICCD.2014.6974719.

[18] D.G. Bailey. "The advantages and limitations of high level synthesis for FPGA based image processing." In: *ACM International Conference Proceeding Series.* Vol. 08-11-Sep-2015. 9th International Conference on Distributed Smart Cameras, ICDSC 2015. School of Engineering and Advanced Technology, Massey University, 2015, pp. 134–139. URL: https://ludwig.lub.lu.se/login?url=https://

search.ebscohost.com/login.aspx?direct=true&AuthType=ip,uid&db=edselc&AN=edselc.2-52.0-84958259629&site=eds-live&scope=site.

[19] Niansong Zhang. *Loop optimization in HLS*. Mar. 2021. URL: https://www.zzzdavid.tech/loop_opt/.

Printed by Tryckeriet i E-huset, Lund 2023

LUND UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2023-925
http://www.eit.lth.se