DEGREE PROJECT

Evaluation of high-level synthesis tools for generation of Verilog code from MATLAB based environments

Carl Bäck

Engineering Physics and Electrical Engineering, master's level
2020

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

LULEÅ UNIVERSITY OF TECHNOLOGY

Abstract

FPGAs are of interest in the signal processing domain as they provide the opportunity to run algorithms at very high speed. One possible use case is to sort incoming data in a measurement system, using e.g. a histogram method. Developing code for FPGA applications usually requires knowledge about special languages, which are not common knowledge in the signal processing domain. High-level synthesis is an approach where high-level languages, as MATLAB or C++, can be used together with a code generation tool, to directly generate an FPGA ready output. This thesis uses the development of a histogram as a test case to investigate the efficiency of three different tools, HDL Coder in MATLAB, HDL Coder in Simulink and System Generator for DSP in comparison to the direct development of the same histogram in Vivado using Verilog. How to write and structure code in these tools for proper functionality was also examined.It has been found that all tools deliver an operation frequency comparable to a direct implementation in Verilog, decreased resource usage, a development time which decreased by 27% (HDL Coder in MATLAB), 45% (System Generator) and 64% (HDL Coder in Simulink) but at the cost of increased power consumption. Instructions for how to use all three tools has been collected and summarised.

Keywords: HLS, System Generator for DSP, Histogram, Xilinx Zynq UltraScale+, FPGA design workflow, Hardware Description Language Coder, HDL Coder, Field Programmable Gate Arrays, Image processing

Sammanfattning

I ingångssteget på ett mätsystem är det av intresse att använda en FPGA för att uppnå höga hastigheter på de oundvikliga datafiltrering och sorterings algoritmer som körs. Ett problem med FPGAer är att utvecklingen ställer höga krav på specifik kunskap gällande utvecklingsspråk och miljöer vilket för en person specialiserad inom t.ex. signalbehandling kan saknas helt. HLS är en metodik där högnivåspråk kan användas för digital design genom att nyttja ett verktyg för automatgenerering av kod. I detta arbete har utveckling av ett histogram använts som testfall för att utvärdera effektivitet samt designmetodik av tre olika HLS verktyg, HDL Coder till MATLAB, HDL Coder till Simulink och System Generator for DSP. Utvecklingen i dessa verktyg har jämförts mot utvecklingen av samma histogram i Vivado, där språket Verilog använts. Arbetets slutsater är att samtliga verktyg som testats leverar en arbetsfrekvens som är jämförbar med att skriva histogrammet direkt i Verilog, en minskad resursanvändning, utvecklingstid som minskat med 27% (HDL Coder i MATLAB), 45% (System Generator) och 64% (HDL Coder i Simulink) men med en ökad strömförbrukning.En sammanställning av instruktioner för utveckling med hjälp av verktygen har även gjorts.

List of Figures

2.1 General FPGA architecture. 5
2.2 Layout of of a CLB section. 6
2.3 DSP block in Xilinx UltraScale+ devices. 7
2.4 Typical FPGA design flow. 8
2.5 Comparison between a sequential and a pipelined workflow. . 14
2.6 Histogram calculation with bins stored in RAM. 15
2.7 Histogram calculation with bins stored in registers. 16
2.8 Schematic view of a register based counter. 16

3.1 Schematical view of SFIR. 20

4.1 General HLS tool workflow. 24
4.2 System overview. 34

List of Tables

2.1 Specification of the target device. 7

4.1 Non optimized test case. 23
4.2 Data types supported by the Fixed-Point converter tool. . . . 26
4.3 Data types supported in HDL Coder. 27
4.4 Arithmetic operators supported in HDL Coder. 27
4.5 Logical operators supported in HDL Coder. 28
4.6 Relational operators supported in HDL Coder. 28
4.7 Relational operators supported in System Generator. 31
4.8 Logical operators supported in System Generator. 32
4.9 xfix-type unique functions. 32
4.10 MATLAB functions supported in System Generator. 33
4.11 Est. development time for the histograms. 34
4.12 Non and default optimized histogram - MATLAB. 35
4.13 Non and default optimized histogram - Simulink. 36
4.14 Default optimized histogram - System Generator. 36
4.15 Pipeline optimization - MATLAB. 37
4.16 Pipeline optimization - Simulink. 37
4.17 Pipeline optimization - System Generator. 38
4.18 Tool comparison of the pipelined versions. 38

6.1 HLS tool ranking. 49

Acronyms

ASIC Application Specific Integrated Circuit.

BRAM Block RAM.

CE Clock Enable.

CLB Configurable Logic Block.

DRAM Dynamic RAM.

DSP Digital Signal Processing.

EDA Electronic Design Automation.

FF Flip-Flop.

FPGA Field-Programmable Gate Array.

HDL Hardware Description Language.

HLL High-Level Language.

HLS High-Level Synthesis.

I/O Input and Output.

IP Intellectual Property.

LM List Manager.

LUT Look-Up Table.

MA Memory Allocator.

MI Mutual Information.

MUX Multiplexer.

RAM Random Access Memory.

SFIR Symmetric Finite Impulse Response.

TE Transfer Entropy.

WNS Worst Negative Slack.

Table of Contents

1 Introduction **1**
1.1 Previous studies . . . 2
1.2 Problem definition and delimitations . . . 2
1.3 Ethical considerations . . . 4

2 Background **5**
2.1 FPGA . . . 5
2.1.1 Configurable Logic Block . . . 6
2.1.2 Programmable interconnects . . . 6
2.1.3 Hard modules . . . 6
2.1.4 Input and Output . . . 7
2.1.5 Xilinx Zynq UltraScale+ target platform . . . 7
2.2 FPGA design methodology . . . 8
2.2.1 Design Entry . . . 8
2.2.2 Synthesis and Mapping . . . 8
2.2.3 Place and Route . . . 9
2.2.4 Bitstream Generation . . . 9
2.2.5 Verification . . . 9
2.3 High-Level Synthesis . . . 10
2.3.1 Commonly used HLLs for HLS . . . 10
2.3.2 A brief overview of a few different HLS tools . . . 12
2.3.3 HDL Coder . . . 12
2.3.4 System Generator for DSP . . . 13
2.3.5 Common optimization techniques in HLS tools . . . 13
2.4 Histogram . . . 15

3 Method **19**
3.1 Literature review . . . 19
3.2 Preliminary tool evaluation . . . 19
3.2.1 Key points gathering . . . 20
3.3 Verilog histogram implementations . . . 20
3.4 HLS histogram implementations . . . 21
3.5 Evaluation criteria . . . 21

4 Results **23**

4.1 Preliminary investigation of the tools 23
4.2 Implementation methods and key points to be aware of 24
4.2.1 General design flow 24
4.2.2 MATLAB method 24
4.2.3 Simulink method 28
4.2.4 System Generator method 30
4.3 Histogram implementations 33
4.3.1 Development time for the different systems 34
4.3.2 Non and default optimized histogram versions 34
4.3.3 Optimized histogram versions 36

5 Discussion **39**
5.1 Preliminary investigation of the tools 39
5.1.1 Formula for maximum possible clock frequency 39
5.1.2 Major differences between the tools 39
5.2 Implementation methods and key points to be aware of 40
5.2.1 Fixed point conversion 40
5.2.2 Supported functionality 41
5.2.3 Timing 42
5.2.4 Usability 42
5.3 Histogram implementations 44
5.3.1 Development time for the different tools 44
5.3.2 Non and default optimized histogram versions 44
5.3.3 Optimized histogram versions 45

6 Conclusion **49**
6.1 Future Work 50

Chapter 1

Introduction

As soon as any measurements are taken signal conditioning is performed to properly capture the characteristics of the given signal. Starting with an analog low pass filter stage the signal then passes through an analog to digital converter and a second batch of conditioning is performed through signal conditioning algorithms. It is critical that these digital stages are performed quickly, so that the sampling speed of the device is not reduced. For this reason Field-Programmable Gate Arrays (FPGAs) are interesting to use for this purpose. Running signal processing algorithms on an FPGA instead of on a regular processor core can improve the operation frequency, as the dedicated hardware resources in FPGAs are usually faster at processing this type of math intensive calculation.

To implement any sort of algorithm on an FPGA requires a good understanding of Hardware Description Languages (HDLs), as e.g. Verilog or VHDL, and development in any of these languages usually takes a lot of time. For a person working on the development of signal processing algorithms this is probably not familiar grounds. Going from algorithm testing in e.g. MATLAB to real life testing on the FPGA is usually not feasible by the same person. A signal processing engineer would have to hand their work over to an engineer with a digital design background, as they have the knowledge required for further development targeting the FPGA.

To simplify progression between different parts of the development cycle, tools called High-Level Synthesis (HLS) tools has been developed. Examples being MathWorks “HDL Coder” and Xilinx “System Generator for DSP” add-on for Mathworks Simulink.

Grepit AB is a high-tech development firm specialised in the development of embedded systems. One of their projects concern the development of a pulse detection and classification system, where a peak detection algorithm has been developed and is currently running on an FPGA. The final output of the system is a histogram displaying these peaks but the frequency accumulation of said peaks are currently being performed on a Linux based computer

system. To increase throughput of this system the frequency accumulation should be moved to the FPGA. The idea of this thesis is to use the development of this histogram as a test case to compare different HLS tools, to find out how effective they are in generating HDL code. Usability for someone without a deeper understanding of the tools will also be investigated.

1.1 Previous studies

Using HLS workflows instead of traditional manual development of FPGA code is something that has been evaluated in earlier studies and was e.g. done by Sarge [1] who investigated the performance and viability in using HDL Coder in Simulink (based on MATLAB version 2017a) with a polynomial nonlinear equalization application as her test case. Her main conclusion were that while performance was worse using the automatically generated code the differences were small enough for it to be of practical use. A similar study was performed by Shah et. al. [2] who developed the MIPI Low Latency Interface using MATLAB 2016b and Simulink as well as Verilog for comparison. Results show a decreased development time at the cost of decreased performance in area and power usage as well as operation frequency.

As there exists a lot of different HLS tools on the market studies has been performed comparing the efficiency of some of these. In one of these studies Baguma [3] studied the implementation of an IIR filter using HDL Coder in Simulink and then compared this to implementation using the C based Vivado HLS tool. He concluded that the Vivado based tool was superior as the HDL Coder implementation did not meet his timing specifications.

The test case for this thesis is the development of a histogram and this is a common application within the area of computer vision, more precisely used e.g. for real-time visual matching and object positioning in autonomous vehicles. Geninatti and Boemo [4] has e.g. developed luminosity histograms, which can be used for image comparison, using two different hardware schemes. One utilizing embedded Random Access Memory (RAM) blocks and one structure of parallel accumulators.

1.2 Problem definition and delimitations

This thesis aims at examining these tools in a similar manner as has previously been done, by developing the same system using several different tools and comparing the results. To keep a clear focus in the report only one application will be developed and that being the histogram. It aims to utilize

newer versions of the tools as to see if they have become more efficient as well as to include more tools for a larger base of comparison. The following specifies the problem definition and delimitations used during this thesis.

Problem definition

- Which of the tools gives the most optimized FPGA implementation of the histogram algorithm?
  - HDL Coder based on a MATLAB function.
  - HDL Coder based on a Simulink model.
  - Xilinx System Generator for DSP.
- How should a .m program be structured and written as to be able to use it as input to HDL Coder or System Generator for DSP?
- Is it appropriate to use a HLS tool to implement HDL code based on these results?

Delimitations

- In this thesis only HLS tool capable of taking .m files and Simulink diagrams as input will be investigated.
- The target platform is a Xilinx Zynq UltraScale+ device but no testing will be performed on real hardware, only Vivado simulations will be performed.
- Methods for code implementation will only be investigated for Xilinx FPGAs.

No hardware has been available and thus the delimitation exists that only simulations will be used. Results should not be much affected by the fact as the Vivado output is what would define the configuration of the output and the amount of resources used should thus be the same. Values for estimated maximum frequency and power usage could differ but as all implementations should be equally affected this should not affect the results.

As the test case used in this thesis is a histogram it should be noted that as the area of Digital Signal Processing (DSP) is very large it spans a lot of different applications and possible use cases, some with very different requirements to the one investigated here. As of this the results stated in this thesis are mainly applicable to problems in the areas of image processing and computer vision, where histograms are heavily used in a manner similar to the development in this thesis.

1.3 Ethical considerations

Auto generation of code structures in the way it is performed today, as a tool to speed up development time but where the process still requires verification of the final results is in itself not an ethical challenge. Strict regulations exist on verification standards for areas of interest to e.g. human life thus the question of ethical considerations does not apply to this area of research.

Chapter 2

Background

2.1 FPGA

FPGAs were introduced in 1984 by a company called Xilinx [5]. FPGAs are used to implement logic functions and they are very efficient at performing calculations on problems which are of a parallel nature, as e.g. linear algebra calculations or where data can be streamed through the device at a high rate as in the case of e.g. video processing [6]. A modern FPGA consists of four main parts, the logic blocks (called Configurable Logic Block (CLB) in the case of a Xilinx device or Adaptive Logic Modules in the case of an Intel device), the programmable interconnects, the hard modules and the Input and Outputs (I/Os). A Xilinx device will be used in this thesis and the naming used will be CLB for the logic unit. A typical FPGA layout can be seen in Fig. 2.1.

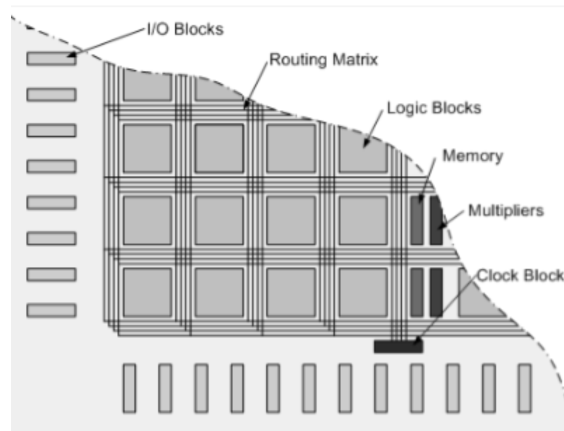


Fig. 2.1: *General FPGA architecture [7].*

2.1.1 Configurable Logic Block

A CLB is the basic logic unit used inside an FPGA to perform its given task. A CLB consists of eight smaller but identical pieces and this smaller group can be seen in Fig. 2.2 [8].

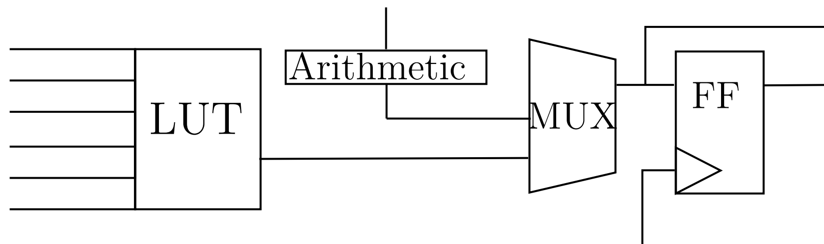


Fig. 2.2: *The layout of $\frac{1}{8}$ of a CLB. Picture adapted from Prof. D. Maskells lecture slides [7].*

2.1.2 Programmable interconnects

The interconnects are the grid of wires surrounding the CLBs as well as the connection matrices, which can be seen as darker grey squares where the routing lines intersect in Fig. 2.1. This network can connect any and all parts of the FPGA in an almost endless number of ways. Certain parts have a dedicated wiring system to increase speed even further, but for the most parts the interconnections are configured by the design tool as needed [7].

2.1.3 Hard modules

In Fig. 2.1 blocks labeled ”Memory” and ”Multipliers” can be seen. These constitute some of the embedded hard modules which can be found in modern FPGAs and which can increase the processing speed of certain tasks. Dedicated RAM sections, called Block RAM (BRAM), exists so that CLBs does not need to be used to implement memory in the Dynamic RAM (DRAM) fashion. This is done to increase speed while keeping the energy consumption low [9]. The multiplier blocks are a lot more advanced than just being multipliers and often go under the name of DSP blocks, see Fig. 2.3 for an overview of the contents of a DSP block in Xilinx UltraScale+ devices. They are efficient to use when implementing signal processing algorithms as they, according to Xilinx user guide on DSPs [10], can implement ”custom parallel algorithms” with the help of different Multiplexer (MUX) devices and options for the arithmetic logic unit.

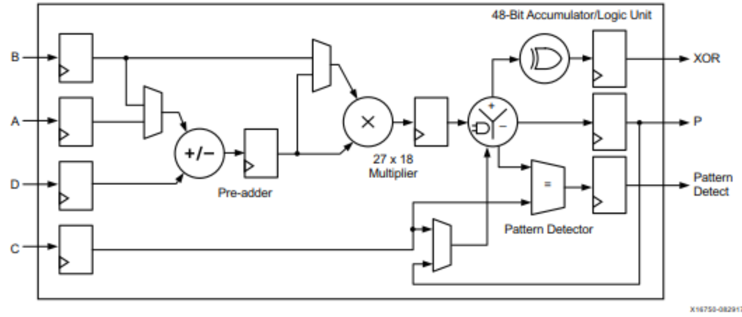


Fig. 2.3: *Layout of DSP block in Xilinx UltraScale+ devices [10].*

2.1.4 Input and Output

The I/O is the part of the FPGA communicating with the outside world. They connect internal registers and CLBs to e.g. sensors and external communication channels to receive or transmit signals. The I/O of a modern FPGA is capable of being configured for single-ended or differential communication over a wide range of different voltage standards. Picking the settings used for the generation of I/O interfaces can allow the designer to implement the physical layer of most communication standards available today, and to change them with a software update if needed. An improvement over using external devices specifically for certain types of communication standards [6].

2.1.5 Xilinx Zynq UltraScale+ target platform

Relevant specifications of the device used as a target platform in Vivado are summarized in Table 2.1.

Table 2.1: *Maximum available resources of the Xilinx FPGA, Zynq UltraScale+ xczu6eg-ffvc900-1-i, used as the target device in Vivado [11].*

Resource	Number of units
LUT	214604
LUTRAM	144000
DSP blocks	1973
BUFG	404
Flip-Flops	429208

2.2 FPGA design methodology

Implementing circuits on an FPGA is done with the help of Electronic Design Automation (EDA) tools. These tools help the designer with transforming the code, written using an HDL, into a netlist. The netlist is then further processed down into the bitstream which is used to program the FPGA itself. General flow followed when designing a circuit for an FPGA can be seen in Fig. 2.4, starting from the design entry block.

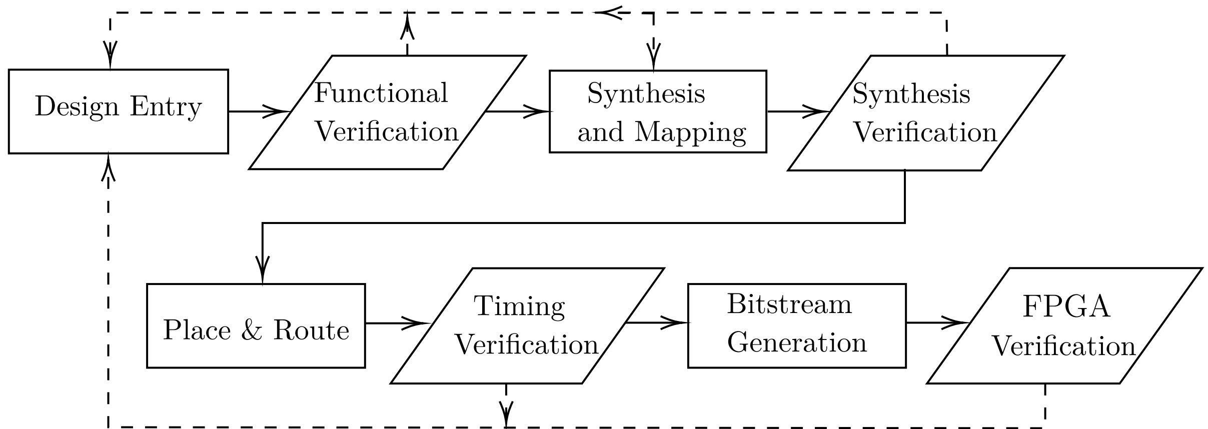


Fig. 2.4: *Typical FPGA design flow [12]. Filled lines represent work flow and dashed are what happens when a verification step fails.*

The workings of each part in Fig. 2.4 will be further described in headings 2.2.1 - 2.2.5.

2.2.1 Design Entry

The design entry stage is where the algorithm that is to be run on an FPGA is first designed. The architecture is defined and things like timing specifications are formulated [7]. The architecture is then described in the EDA tool, e.g. as a block diagram or HDL code [12].

2.2.2 Synthesis and Mapping

The synthesis and mapping stage of the design flow is where the implemented design gets translated into a low level circuit description, called a netlist. The synthesis part generates the logic translation of the circuit and in this stage

combinational logic sections are minimized. Hard modules are inferred as to speed up arithmetic operations [7].

The mapping is the physical translation of the synthesis stage. In this stage the tool infers what components need to be used on the target device to implement the circuit described by the netlist from the synthesis stage. Combinational logic is converted into LUTs, synchronous components are mapped to registers and the hard modules defined in the netlist are mapped to specific types of hard modules on the target [7].

2.2.3 Place and Route

Place and route is where data from the mapping stage gets translated to the exact locations of the target device. Blocks are mapped to specific areas on the silicon and the routing between these parts are defined. This is the computationally heaviest part of the operation and it starts from a randomized placement, it then gets iteratively improved until an optimal placement is found based on user constraints [7].

2.2.4 Bitstream Generation

The bitstream is the binary code which configures the FPGA so that it fulfills the intended purpose. LUTs are given proper values and the routing matrices are defined [12].

2.2.5 Verification

Verification are the stages where the circuits are simulated to check that they are behaving as expected, and that they fulfill the specifications defined before starting. The different forms of verification mentioned in Fig. 2.4 are described below.

Functional

The functional verification checks that the circuit implemented in the design entry stage correctly describes the algorithm it is supposed to. This is done by testing individual components while they are being developed, as well as testing the complete functionality of the circuit once it has all been assembled. If it does not match a return to the design entry stage is required to fix the bugs [7].

Synthesis

Synthesis verification checks that the netlist from the synthesis and mapping stage is synthesised correctly and implementable in the hardware being targeted. It also checks that no functionality has been unintentionally removed. If something fails the designer returns to the design entry stage to fix the error, settings in the synthesis and mapping tool stage can also address some issues [7].

Timing

The final verification stage performed before hardware testing commences is the timing verification. Timing verification calculates the timing characteristics of the implemented circuitry so that the designer can check if the design fulfills the timing specifications. If the specifications are not met, the designer will return to the design entry stage and redo the implementation with e.g. more pipeline stages as to increase the speed of the circuit [12].

FPGA

Hardware verification on the target device. Laboratory tests are performed to verify that the behaviour matches the expected one.

2.3 High-Level Synthesis

HLS tools aim to increase the effectiveness of HDL development by allowing the designer to work in an High-Level Language (HLL) like C++ or MATLAB to implement algorithms, then to let the tool automatically generate an implementation in the chosen HDL. This code can then be utilized in an EDA as e.g. Vivado to complete the design flow with further simulations and finally the bitstream generation [13].

HLS tools has a history of not being very effective at generating HDL code and is therefore not used everywhere, but as the tools increase in performance so will their use [5]. Hitachi has e.g. found a use for the HDL Coder tool when combined with Mathworks Simulink model based design principles were they have managed to collect everything from high-level specifications down to the development and testing into one system. This has allowed them to improve inter-team communications and decrease development time [14].

2.3.1 Commonly used HLLs for HLS

Some of the most common HLLs used in HLS are C-based, SystemC or MATLAB [15]. Some comparative data relating to the C-based and the

SystemC are presented here based on findings by K. Georgopoulos et. al. [16].

C-based

The C-based HLLs are typically C or C++. They are basic languages for many software engineers and they also share strong similarities between original untimed algorithms and the versions usable as input to HLS tools. These two factors grant the C-based languages a shorter learning path until code can be produced in comparison to SystemC. Math libraries, such as math.h, is typically supported in full by any C-based HLS tool [16].

Using data from Table 1 in K. Georgopoulos et. al. [16], stating their development time for different algorithm implementations, the average development time for the C-based version of the algorithms was 9h for the Mutual Information (MI) and Transfer Entropy (TE) algorithms, and 11h for the List Manager (LM) and Memory Allocator (MA) algorithms.

SystemC

SystemC is a system-level modeling language typically used for performance modeling, functional verification and HLS. Being based on C++ macros and classes it adds an environment for simulating concurrent threads and their interactions [17]. Whilst the learning curve on how to apply SystemC code in an HLS environment is steeper than the C-based systems, the end product is typically more robust and comprehensive. HLS tools using SystemC as its input typically does not include support for the entirety of its math libraries and these might thus have to be implemented by the designer, leading to an increased development time [16].

Using data from Table 1 in K. Georgopoulos et. al. [16] the average development time for the SystemC version of the algorithms was 46h for the MI and TE algorithms (adding averages of both math and design time) and 27h for the LM and MA algorithms.

MATLAB

MATLAB is a language commonly used in data analytics, wireless communication, deep learning, robotics and many others [18]. Its uses in the signal processing domain makes it a suitable language to use as input to an HLS tool, as algorithms to be used on an FPGA might very well have been developed in this language already.

2.3.2 A brief overview of a few different HLS tools

There exists a number of different HLS tools on the market, as well as even more from academia. A brief summary of some notable commercial systems are mentioned here. Tools used in this thesis will be covered more in depth under the headings 2.3.3 and 2.3.4.

Catapult-C

Catapult-C is developed by Calypto Design Systems and was originally intended to be used for Application Specific Integrated Circuit (ASIC) development, but today it supports both this and FPGA development. It takes as its input C, C++ or SystemC and can generate either VHDL, Verilog or SystemVerilog as its output. Optimization options includes loop pipelininig and unrolling [13].

VivadoHLS

VivadoHLS is developed by Xilinx and takes as its input C, C++ or SystemC and can generate either VHDL, Verilog or SystemVerilog as its output. Optimization options includes loop pipelininig, unrolling and operation chaining [19].

Synphony C

Synphony C is developed by Synopsys and takes as its input C or C++ and can generate either VHDL, Verilog or SystemVerilog as its output. Optimization options includes loop pipelininig and unrolling [19].

2.3.3 HDL Coder

HDL Coder is a tool developed by Mathworks and takes MATLABs .m files, Simulink models or Stateflow charts as input. It generates either Verilog or VHDL as its output and can be used for either FPGA or ASIC development. Optimization techniques included in the tool are e.g. pipelining and resource sharing. Test benches for the generated HDL code can also be automatically created and test vectors can thus be reused from the simulation stages. The generated HLS code fulfills standard rules from industry. It is possible to keep traceability between different levels of the design, e.g. by automatically adding the equivalent MATLAB code as comments and to keep a connection from high-level requirements down to the HDL code itself. This simplifies the fulfillment of safety standards in the aviation, automotive, machinery and industrial automation sectors [20].

2.3.4 System Generator for DSP

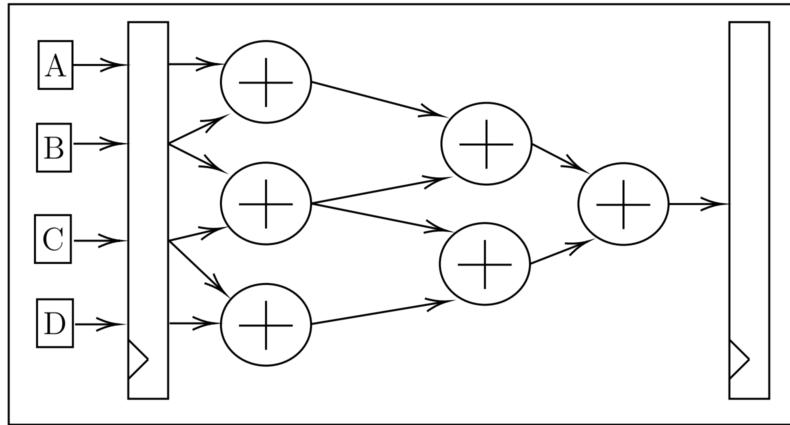
System Generator for DSP is a tool developed by Xilinx and it works as an add-on for Simulink defining its own block library. This library includes blocks which generate Intellectual Property (IP) optimized for the target device. Code generation outputs a stand-alone IP for use in Vivado, an HDL netlist to replicate the design or a synthesized checkpoint to use in Vivado. It can generate all these options in either Verilog or VHDL. Test benches for the generated HDL code can also be automatically created and test vectors can thus be reused from the simulation stages [21].

2.3.5 Common optimization techniques in HLS tools

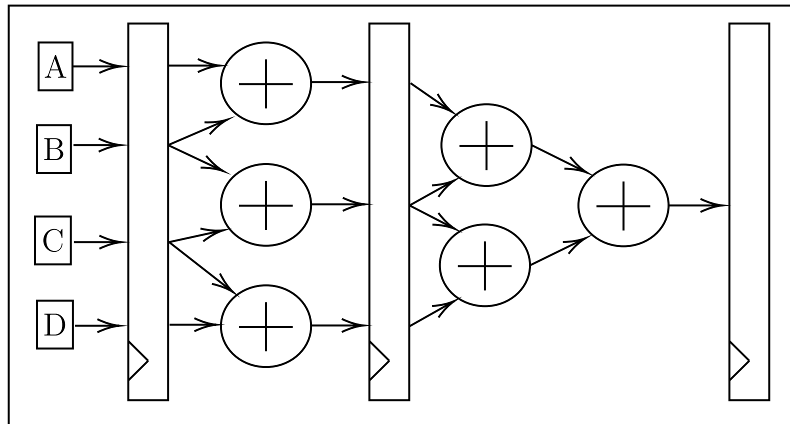
One of the features supplied by HLS tools are the automated optimization techniques applied to generated code. Following is a brief explanation of two of the capabilities available in the tools used in this study.

Pipelining

As described in Hennessy and Patterson [22] pipelining is "An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line". In an FPGA setting this corresponds to using registers to shorten the critical path of a calculation. It results in an increased throughput, number of samples processed per clock cycle, while also possibly increasing latency, the delay between input and its corresponding output. The maximum frequency of a circuit can be calculated as $f_{max} = \frac{1}{criticalpath}$, where the critical path is given in seconds. This formula originates in that inputs can only be supplied at a frequency matching the slowest path of the circuit to not lose any data. Fig. 2.5 provides an example of this phenomenon [7].



(a) *Sequential case. Critical path is 3ns,* $f_{max} = 333MHz$ *and the latency is one clock cycle,* $latency = 1 * 3ns = 3ns$.



(b) *Pipelined case. Critical path is 2ns,* $f_{max} = 500MHz$ *and the latency is now two clock cycles,* $latency = 2 * 2ns = 4ns$.

Fig. 2.5: *Comparison between a sequential and a pipelined workflow [7]. Throughput and latency values based on the assumption that an adder has a delay of 1ns to produce its results.*

Resource sharing

Resource sharing is the concept of using the same hardware resources, for instance DSP blocks, for several calculation steps of an algorithm. Resulting in a decreased area usage of the design but at the expense of lowering the maximum throughput. As algorithms sometimes requires more resources, especially multipliers and other hard modules might be in short supply, than what is available on the device used, it might become a necessity as to be

able and run the algorithm at all [23].

2.4 Histogram

A histogram is a statistical tool used to gather frequency data about a given data set. It find uses in several domains, relevant to this thesis is e.g. computer vision and object detection [24].

Calculation of a histogram on an FPGA typically involves the use of one of the two methods described in Baileys book ”Design for Embedded Image Processing on FPGAs” [25], working on data stored in RAM or in local registers. Input data to these histograms are typically the highest data bits of the data to be placed in the histogram, how many bits are used depends on the amount of bins in the histogram. Fig. 2.6 shows a schematic view of the RAM based implementation.

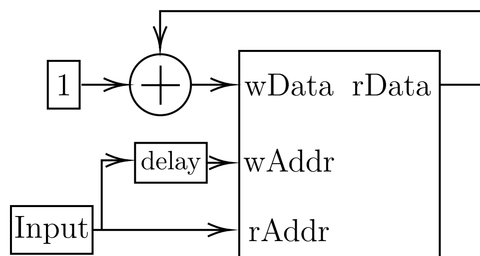


Fig. 2.6: *Histogram calculation based on dual-port RAM. The data is used as the address to read from in the RAM. The data read is increased by one and then stored back in RAM on the same address, utilizing a delayed version of the data as write address.*

An advantage of the RAM version is that it only utilizes a small amount of resources this is as it only requires a RAM block with some minor logic around it for control and updating the data. On the downside it requires the delay logic to be properly configured as to match the RAMs read/modify/write cycle as the memory is using this update routine it will also perform at a lower maximum speed than the register based version [25].

Fig. 2.7 shows an equivalent schematic of the register based implementation.

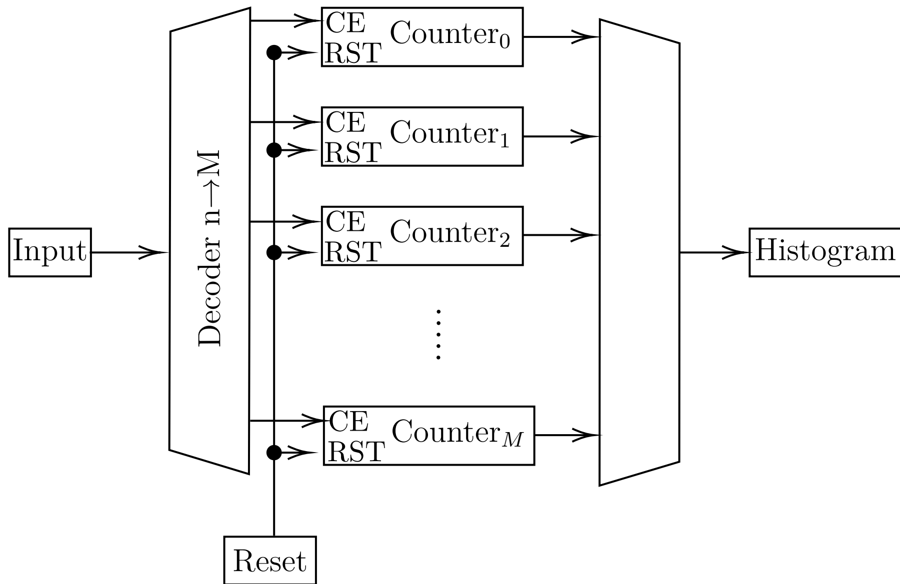


Fig. 2.7: *Histogram calculation based on register counters, the input data is used to address and activate the Clock Enable (CE) input of the relevant counter to increase it by one. $M = 2^n - 1$.* *Picture adapted from Bailey [25].*

Summarized as "Counter$_M$" in Fig. 2.7 is a counter using registers to store the current count, see Fig. 2.8 for an overview of the implementation.

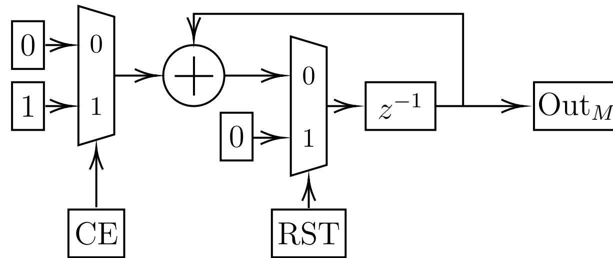


Fig. 2.8: *Schematical view of a register based counter.*

The register version uses a lot of resources, especially if implementing large design, as it uses one counter per histogram bin with each counter being implemented as seen in Fig. 2.8. The decoder circuit is typically built by LUTs and even if it is easy to design it grows large for big systems [25]. The main advantages of the register version is that it does not require any complex timing logic and it is capable of reaching high speeds as the only

main delay in the system is the one in the decoder which would typically be in the nanosecond range [7].

Chapter 3

Method

3.1 Literature review

A review of literature has been performed where the following areas were investigated: FPGAs, HLS, HDLs, Mathworks HDL Coder software, Xilinx System Generator for DSP software and implementation strategies of histograms on FPGAs. These were investigated to form a basis of understanding about the areas covered in the rest of the thesis as well as to find previous studies covering the effectiveness of HLS tools to be used for comparison. Main sources of material were the IEEE databases, books, earlier thesis works and datasheets from Xilinx and Mathworks.

3.2 Preliminary tool evaluation

The three tools under consideration was first tested in a preliminary testing phase to gain an understanding on how they function, gather key points to follow during the histogram development as well as to see if any tool over or under performs in comparison to the rest.

The preliminary testing was performed by implementing a Symmetric Finite Impulse Response (SFIR) filter based on Mathworks example project on the topic [26]. The SFIR filter is implemented manually in all tools as to be as equal as possible and thus grant a better comparison. The SFIR was chosen as a benchmarking tool because it incorporates several different basic system blocks: addition, multiplication (making use of DSP blocks) and delay lines utilizing Flip-Flops (FFs). Mathworks version of this is readily available and could directly be used in MATLAB, adapting it to run in Simulink and System Generator was easily done based on the MATLAB version.

The example utilizes the symmetry of the FIR filter to decrease the number of operations necessary, a block based overview of the implementation can be seen in Fig. 3.1.

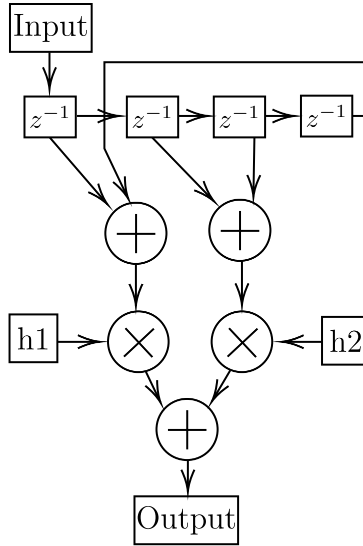


Fig. 3.1: *Schematic overview of the SFIR filter implemented for tool testing purposes [26]. h1 and h2 are the filter constants used in the filter.*

3.2.1 Key points gathering

Instruction sheets on HDL Coder and System Generator has been studied and compared to experiences gathered during the tool evaluation phase. These experiences has then been summarised as key points to take note of when developing code in the given HLS tool.

3.3 Verilog histogram implementations

The implementation method followed for the histogram is as seen in Fig. 2.7. This method has been chosen as we do not want to be limited in the maximum attainable clock speed and because we have a lot of resources available in the chosen FPGA. Another reason for this method to be chosen is to bypass the potential problems with the delay blocks timing logic as theses possible errors are not directly connected to the goals of this thesis and thus would only waste time. Implemented as 256 individual 32-bit wide clock enabled counters, according to project requirements. The counter to increase is activated by the use of an 8 to 256-bit decoder using the eight most significant bits of the data packet as address bits. A separate data conditioning function, with a basic move by offset x and scale by a factor y, was also designed and the logic required for that will be included in the total resource comparison.

3.4 HLS histogram implementations

Implementation of the hardware based algorithm was written directly in the HLS tools, avoiding any toolbox macros or similar. It was designed in the same way as the Verilog version as to generate an implementation that would be comparable. The tools used were MATLAB version 2018a [27], Simulink v. 9.1 [28], HDL Coder toolbox v. 3.12 [20] and Vivado with System Generator version 2018.1 [21].

3.5 Evaluation criteria

The HDL code generated by the HLS tools were all synthesized and implemented using the ”out of context” mode in Vivado. This mode generates an implementation that is intended to be used as part of another system and no external I/O ports are created. Some testing of the tools optimization options was also performed. The tools were then evaluated based on the following five factors:

- Resource usage.
- Estimated max frequency.
- Estimated power consumption.
- Usability.
- Major differences between the tools.

Chapter 4

Results

4.1 Preliminary investigation of the tools

Table 4.1: *Non optimized test case.*

Resource/Parameter	**MATLAB**	**Simulink**	**System Generator**
WNS (ns)	4,4	8,876	8,407
Estimated max frequency (MHz)	179	890	628
LUTs used	159	0	191
LUTRAM used	31	0	0
DSP blocks used	4	4	4
FFs used	364	96	294
Estimated Total On-Chip Power (W)	0,628	0,624	0,627

The tools were preliminary evaluated without speed optimization routines used and the results can be seen in Table 4.1. The optimization tools utilized by Vivado tries to meet the timing constraints set in the user supplied constraints file, set to 100MHz during simulations, and nothing more. Thus not much can be said about any theoretical maximum speed of the design from the Vivado simulation itself. Using the Worst Negative Slack (WNS) value the formula $f_{simMax} = \frac{1}{T_{simulationClk} - WNS}$, found on the Xilinx forums [29], is supposed to estimate this value, $T_{simulationClk} = 10ns$.

4.2 Implementation methods and key points to be aware of

Code implementation methodologies are discussed in the following chapter.

4.2.1 General design flow

The general design flow followed when developing code with an HLS tool can be seen in Fig. 4.1.

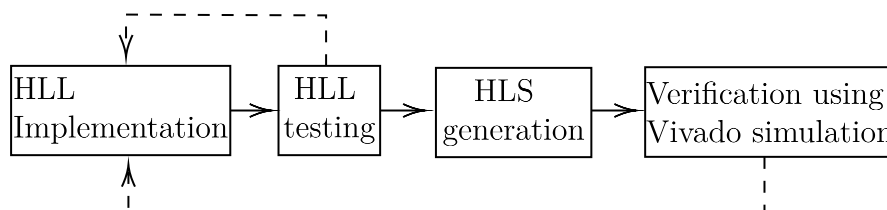


Fig. 4.1: *General HLS tool workflow.*

The last stage of Fig. 4.1 includes everything covered in Fig. 2.4. During development no iteration at this stage has been necessary and these last steps has been running smoothly.

4.2.2 MATLAB method

Writing efficient MATLAB code to be used with HDL Coder requires that the code is formatted along the idea that it is going to be implemented in hardware. This places requirements on the code relating to data types, variable sizes, fixed vs. floating point precision, data storage and architectures [30].

MATLAB applications developed to be used with the HDL Coder tool has to be written utilizing a main file which in turn launches any sub-functions used. The second component used by HDL Coder is the function test bench. This file has to present a test bench which covers all possible input scenarios expected of the function as it is used by both HDL Coder to define input types, as well as by the Fixed-Point converter tool to evaluate required word or fraction lengths of the variables in the function [30].

Test bench auto generation is an option available in the HLS tool and if this is activated the supplied test bench is also converted to the chosen HDL. It is to be used in a later stage, e.g. a Vivado simulation, to check the functionality

of the automatically generated HDL code with the same reference as was used in MATLAB [30].

HDL Coder has an option for automatically configuring Vivado projects so that opening the HLS output and verifying it in Vivado can easily be done after HDL Coder is finished. This functionality did not work properly and manual addition of any test related files was required [30].

Fixed point precision

By default a MATLAB function uses double precision floating point data (64-bit numbers with a floating decimal point) in its calculations. This infer problems when it is run on an FPGA as it requires a lot of resources [31]. In FPGAs fixed point precision is typically used to save on the amount of resources required for calculations.

In MATLAB conversion from floating point to fixed point is most easily done by the use of the Fixed-Point converter [32]. This tool automates the process of translating a function using floating point precision, into an equivalent function using fixed point precision. This is also a way to reduce data size as precision can be decreased from the default 64-bits to a smaller value.

The converter tool takes the function to convert and a test bench utilizing the function to run a simulation to determine range and precision required to cover the variables in question. Tool settings include standard values for word or fraction length, which one of the two to determine, how rounding and overflow is to be handled, precision in arithmetic operators during synthesis and if the variables proposed are to be signed, unsigned or automatically set depending on the simulation results. After simulation the tool proposes word or fraction length and the user can then with the press of a button convert the entire function to use the new proposed data types. The changed function is saved as a separate file as to retain the original for further development.

The Fixed-Point converter tool only supports the data types stated in Table 4.2 to be used for simulation and code generation.

Table 4.2: *Data types supported by the Fixed-Point converter tool.*

Type	Supported Data Types
Integer	unsigned data types, 1 to 128 bits signed data types, 2 to 128 bits
Real	single, double, scaled double
Logical	boolean

Serial data transfers

As MATLAB is a vector based programming language, transferring large data sets, in parallel, between functions is not uncommon. At the same time it is something which put hardware resources under large strain. Considering that a default variable in MATLAB is 64-bits wide every variable transferred to a function intended for hardware implementation would need 64 binary I/O ports. Comparing this to the available I/O units on a modern FPGA like e.g. the Zynq Ultrascale+ xczu6eg-ffvs900-1-i, which has 208 I/O units in total. It quickly becomes apparent that sending several variables or even full arrays of data, in parallel, to be processed is not a feasible task [11].

Converting the large floating-point variables into smaller fixed point variables alleviates some of this problem but when it comes to transferring large arrays of data a different approach is required. Serial data transfers are e.g. used to limit the I/O requirements of functions calls, as a reduced amount of ports are needed but at the cost of increased latency. In MATLAB serial data transfers are implemented using for loops, which calls the function in question several times with smaller data sets instead of calling it with the entire data set at once. If there is a need to compare several of these data sets to each other, data will have to be registered in the function between the calls so as to be available during following function calls.

Supported data types and functions

MATLAB functions supported while using the HDL Coder can be seen HDL Coder Users Guide chapter 1 [30]. Data types and operators supported by HDL Coder can be seen in Tables 4.3 - 4.6 as well as chapter 2 of the HDL Coder Users Guide, where applicable restrictions also can be seen [30].

Table 4.3: *Data types supported in HDL Coder.*

Type	**Supported Data Types**
Integer	uint8, uint16, uint32, uint64, int8, int16, int32, int64
Real	double, single
Character	char
Logical	logical
Fixed point	Scaled (binary point only), fixed point numbers, Custom integers (zero binary point)
Vectors	unordered {N}, row {1, N}, column {N, 1}
Matrices	{N, M}
Structures	struct
Enumerations	enumeration

Table 4.4: *Arithmetic operators supported in HDL Coder.*

Types	**Operator Syntax**
Binary addition	A+B
Matrix multiplication	A*B
Array wise multiplication	A.*B
Matrix power	A^B
Array wise power	A.^B
Complex transpose	A'
Matrix transpose	A.'
Matrix concatenation	[A B]
Matrix index	A(r c)

Table 4.5: *Logical operators supported in HDL Coder.*

Relation	Operator Syntax
Logical And	A&B
Logical Or	A|B
Logical Xor	A xor B
Logical And (short circuiting)	A&&B
Logical Or (short circuiting)	A||B
Element complement	~A

Table 4.6: *Relational operators supported in HDL Coder.*

Relation	Operator Syntax
Less than	A<B
Less than or equal to	A<=B
Greater than or equal to	A>=B
Greater than	A>B
Equal	A==B
Not equal	A~=B

Registers

Creating registers in MATLAB is done by the use of persistent variables and the "isempty" function to initialize the register, an example can be seen in the box below where ud1 and ud2 are defined as registers and preallocated with the value of zero [30].

```
1 persistent ud1 ud2;
2 if isempty(ud1)
3     ud1 = 0; ud2 = 0;
4 end
```

4.2.3 Simulink method

Designing Simulink code to be usable in HDL Coder follows the same notes as stated above in the MATLAB header concerning fixed point precision,

supported data types and registers. Topics required for the Simulink implementation, not covered in the MATLAB section above are listed in thew following headers.

Supported functions

The HDL Coder library, in the Simulink library browser, contains all functions supported by the HDL Coder tool [30].

HDL Coder usage

In a similar manner to MATLAB the code to be run through the HLS tool has to be packaged. Simulink uses the name ”subsystems” where in MATLAB it would have been called ”functions”. The subsystems reside in the main level of the model but its also possible to maintain several levels of hierarchy so as to increase readability. Outside of the HDL subsystem any sort of block can be added as to add stimulus and other simulation capabilities to the model [33].

Test bench auto generation is an option available in the HLS tool and if this is activated the entire system surrounding the HDL subsystem is also converted to the chosen HDL and utilized as a test bench in later stages, as e.g. Vivado. It is used to verify the generated code against the same reference as was used in Simulink during development [30].

HDL Coder has an option for automatically configuring Vivado projects so that opening the HLS output and verifying it in Vivado can easily be done after HDL Coder is finished. This functionality did not work properly and manual addition of any test related files was required [30].

Sample time

Input data in Simulink is either sampled directly at the input or passed through a rate transition block before entering the HDL subsystem to make sure that the correct sampling rate is used during simulation and in the final code generation.

External model interfaces

In the same way as the MATLAB method requires data to be transferred in a serial manner it is also required here. As the simulation model works in software it will not throw errors from transferring data packets which are to large for the hardware implementation to handle. Instead this has to be included in the design of the system to not encounter issues once it is running on the FPGA.

MATLAB function blocks

Simulinks normal MATLAB based function blocks are available which can be used to include MATLAB code directly as a new Simulink block. If anything was missing in the HDL Coder library, MATLAB code could be developed to cover this as long as this new code follows the limitations in function usage, types etc. mentioned in section 4.2.2.

4.2.4 System Generator method

Designing with the System Generator library takes place in the Simulink environment but it is launched as a separate application. Parts to be generated requires two specific blocks to define it. The Gateway for input and output definition, and the System Generator token for defining the FPGA technology. Subsystems to be generated can only utilise blocks from the System Generator library. The usual Simulink libraries are available and can be used for simulation purposes outside of this subsystem.

System Generator token

The System Generator token is used to define the FPGA technology for the targeted architecture. The token is also the main interface to the HLS tool itself and it grants access to settings for the code generation. Settings include FPGA version and model, what clock speed to be used on the FPGA, Simulinks system simulation period, defining the location constraints for the clock on the FPGA, defining target architecture, compilation goals and HDL languages to generate. Test benches based on the entire implemented system, performance tests, resource and timing estimates can be generated. The tool itself outputs an IP block holding the developed functionality as well as an example project for testing and verification purposes.

Gateways

System Generator uses a specific block, the Gateway, to define the external I/O interfaces for the function. Simulink libraries used as stimulus during simulations use double precision floating point data types as default but as System Generator blocks only work on boolean, arbitrary precision fixed point or in some cases floating point data the conversion from double precision to a suitable type is done in the input Gateway blocks. In a similar manner conversion back to double precision is performed when data passes through the output Gateway. Gateways are also what defines I/O during generation and the name of the Gateway will transfer to become port names in the generated code [34].

System Generator MCode blocks

System Generator can use MATLAB functions directly by using the function blocks called MCode. MCode blocks are meant to be used to implement functionality related to arithmetic functions, finite state machines and control logic and it is stated in the Vivado Reference Guide for Model-Based DSP Design Using System Generator that it ”[...] supports a limited subset of the MATLAB language [...]” [35, p.215]. The MCode block supports the following language constructs,

- Assignment statements.
- Simple and compound if/else/elseif statements.
- Switch statements.
- Arithmetic expressions involving only addition and subtraction.
- Addition.
- Subtraction.
- Multiplication.
- Division by a power of two [35].

Relational and logical operators supported in System Generator can be seen in table 4.7 and 4.8 as found in chapter 1 of the reference guide [35], covering the MCode block.

Table 4.7: *Relational operators supported in System Generator.*

Relation	**Operator Syntax**
Less than	A<B
Less than or equal to	A<=B
Greater than or equal to	A>=B
Greater than	A>B
Equal	A==B
Not equal	A∼=B

Table 4.8: *Logical operators supported in System Generator.*

Operation	**Operator Syntax**
And	A & B
Or	A | B
Not	$\sim$A

The MCode blocks can only utilize Xilinx own data type, the xfix, which can be either signed, unsigned or boolean. Table 4.9 states the special functions required in the MCode environment to work on this data types [35].

Table 4.9: *xfix-type unique functions.*

Command	**Functionality**
xl_nbits()	Returns number of bits
xl_binpt()	Returns binary point position
xl_arith()	Returns arithmetic type
xl_and()	Bit-wise and
xl_or	Bit-wise or
xl_xor()	Bit-wise xor
xl_not	Bit-wise not
xl_rsh()	Shift right
xl_lsh()	Shift left
xl_slice()	Slice
xl_concat()	Concatenate
xl_force()	Reinterpret
xl_state()	Internal state variables

Table 4.10 displays the basic MATLAB functions which are supported in the MCode blocks [35].

Table 4.10: *MATLAB functions supported in System Generator.*

Command	Functionality
disp()	Displays variable values
error()	Displays message and abort function
isnan()	Test whetever a number is NaN
NaN()	Returns Not-a-Number
num2str()	Converts a number to a string
ones(1,N)	Returns 1-by-N vector of ones
pi()	Returns π
zeros(1,N)	Returns 1-by-N vector of zeros
for	For loop

Pipelining

Pipelining or other forms of optimization routines are not readily available in the System Generator tool for automatic usage and thus any optimization has to be done manually. Certain blocks, like multipliers, contains a setting to add internal pipelining stages but more generally pipelining is inserted as registers or delay blocks in between subsystems or other components.

Xilinx Waveform viewer

The Xilinx Waveform Viewer is a tool used to manually evaluate generated waveforms in the simulation environment. It shares its layout with the Vivado simulation viewer and all waveforms added are shown at the same time in the same window for ease of comparison.

4.3 Histogram implementations

Fig. 4.2 shows an overview of the implemented system, the blocks inside the dashed lines are the ones developed as a test case for this thesis.

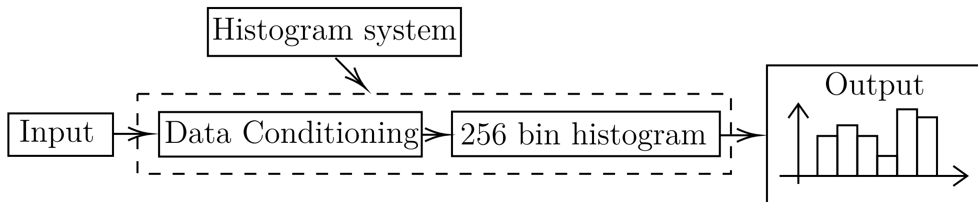


Fig. 4.2: *System overview.*

The HLS tools has been tested using their non optimized, default optimized and some optimized generation settings. Data presented has been collected from Vivados post-implementation project summary page. All histograms has been simulated after HLS generation using Vivados behavioural simulation and verified to produce equal histograms.

4.3.1 Development time for the different systems

An estimation of the total development time in the different tools has been summarized in Table 4.11. One tool was used at a time and the development in it was finished before moving on to the next one in the list. The tools has been used in the following order. Development started with Verilog followed by MATLAB, Simulink and then finally System Generator. Knowledge gained along the way was used in the later tools. In addition to the specific development time related to the histograms approximately another 5h has been spent on creating a MATLAB based generator for test data, the output of which has been utilized in the development of all histogram versions.

Table 4.11: *Estimated development time for the histogram in the different tools.*

Tool	**Estimated development time (h)**
Verilog	33
MATLAB	24
Simulink	12
System Generator	18

4.3.2 Non and default optimized histogram versions

MATLAB

The non optimized data is collected by turning off all settings relating to optimization in the HLS tools menus. Default optimized is here defined as keeping the HLS tools settings as they were when the tools are started for the first time in a new project. For MATLAB this means [20]:

- Always share multipliers.
- Map persistent arrays to RAM if they are larger than 256 elements.
- Loop optimizations are set to ”none”.

Table 4.12 displays the resulting data.

Table 4.12: *Non and default optimized histogram - MATLAB.*

Resources and Parameters	None	Default
WNS (ns)	1,913	4,988
Estimated max frequency (MHz)	124	200
LUTs used	1670	2298
LUTRAM	0	0
DSP blocks used	2	2
FFs used	1563	1545
BUFG units used	0	0
Estimated Total On-Chip Power (W)	0,664	0,656

Simulink

The non optimized data is collected by turning off all settings besides the ”balance delay” setting in Simulink (as the model would not generate anything without this active) settings relating to optimization in the HLS tools menus. In Simulink default settings mean [20]:

- Balance delays.
- Transform non zero initial value decay.
- Clock-rate pipelining.
- Adaptive pipelining.
- Always share multipliers, multiply-add blocks, atomic subsystems, MATLAB function blocks and Floating-point IPs.

Table 4.13 displays the resulting data.

Table 4.13: *Non and default optimized histogram - Simulink.*

Resources and Parameters	**None**	**Default**
WNS (ns)	1,62	1,62
Estimated max frequency (MHz)	119	119
LUTs used	15301	15301
LUTRAM	1	1
DSP blocks used	2	2
FFs used	16448	16448
BUFG units used	0	0
Estimated Total On-Chip Power (W)	0,873	0,873

System Generator does not support settings in the same way as MATLAB and Simulink. Stated in the User Guide for System Generator is that "the more complex IP blocks [...] are generated under the hood. They are provided as highly-optimized netlists [...]" and the output from the tool will be called default optimization [34]. Table 4.14 displays the resulting data.

Table 4.14: *Default optimized histogram - System Generator.*

Resources and Parameters	**Default**
WNS (ns)	3,088
Estimated max frequency (MHz)	145
LUTs used	583
LUTRAM	1
DSP blocks used	2
FFs used	8194
BUFG units used	1
Estimated Total On-Chip Power (W)	0,702

4.3.3 Optimized histogram versions

The optimization routines tested is the addition of input and output registers. Testing the tools capabilities of automatically inserting one stage of

pipelining. In the case of System Generator and HDL Coder for Simulink the addition of input and output registers was done manually as the tools did not have a setting for this. Tables 4.15 - 4.17 compares the tools default parameters to its pipelined form. Table 4.18 compares the different tools to each other as well as to the Verilog implementation.

Table 4.15: *One stage pipeline optimization - MATLAB.*

Resources and Parameters	Default	Pipelined
WNS (ns)	4,988	3,555
Estimated max frequency (MHz)	200	155
LUTs used	2298	2315
LUTRAM	0	0
DSP blocks used	2	2
FFs used	1545	3164
BUFG units used	0	0
Estimated Total On-Chip Power (W)	0,656	0,678

The code generated by HDL Coder in the pipelined MATLAB case has been inspected in the same way as the non and default versions. It has been confirmed that the code has been generated in the same way as in the default version, with the only difference being the extra registers on input and outputs.

Table 4.16: *One stage pipeline optimization - Simulink.*

Resources and Parameters	Default	Pipelined
WNS (ns)	1,62	2,998
Estimated max frequency (MHz)	119	143
LUTs used	15301	11009
LUTRAM	1	1
DSP blocks used	2	2
FFs used	16448	8280
BUFG units used	0	0
Estimated Total On-Chip Power (W)	0,873	0,811

The code generated by HDL Coder in the pipelined Simulink case has been

inspected in the same way as the non and default versions. The code has been generated in the same way as in the default version with two differences. The two differences are the extra registers on inputs and outputs and that the pipelined case lacks a type conversion on the output which is included in the default version.

Table 4.17: *One stage pipeline optimization - System Generator.*

Resources and Parameters	Default	Pipelined
WNS (ns)	3,088	3,579
Estimated max frequency (MHz)	145	156
LUTs used	583	583
LUTRAM	1	1
DSP blocks used	2	2
FFs used	8194	16436
BUFG units used	1	0
Estimated Total On-Chip Power (W)	0,702	0,751

Table 4.18: *Comparison of the one stage input and output register version of the HLS tools with the Verilog implementation.*

Resources and Parameters	Verilog	MATLAB	Simulink	System Generator
WNS (ns)	3,301	3,555	2,998	3,579
Estimated max frequency (MHz)	149	155	143	156
LUTs used	4683	2315	11009	583
LUTRAM	0	0	1	1
DSP blocks used	2	2	2	2
FFs used	16690	3164	8280	16436
BUFG units used	1	0	0	0
Estimated Total On-Chip Power (W)	0,66	0,678	0,811	0,751

Chapter 5

Discussion

5.1 Preliminary investigation of the tools

A brief discussion about the key points of the preliminary testing will be conducted before focus shifts to the implementation methods and the histogram implementation.

5.1.1 Formula for maximum possible clock frequency

Trying to calculate the maximum possible clock frequency that an application can run at is not an exact measure as it will depend on a number of different things, mainly the critical path and final location on the device. Wire lengths will differ depending on how the remainder of the system is implemented and this is connected to how filled a certain FPGA is at the time of placement, as this might block certain resources or paths from being utilized.

As the histograms developed within this thesis have been examined in the ”out of context” mode they do not include any I/O ports and are not placed in a specific place on an FPGA running the full system that eventually will use this functionality. The applications has not been tested on an real FPGA but only in simulation. All this translates to that the estimated max frequency values calculated for the different implementations are good approximations in regards to comparing the different versions on an equal ground but they should be verified before they are used in a live system.

5.1.2 Major differences between the tools

Based on the resulting data found in Table 4.1 the different tool implements widely different circuits, but where examining the simulated Vivado outputs shows that they behave in a similar manner. I find it safe to say that the same task can be completed in different ways and that even though the tools work on similar types of code internally they clearly function in different ways.

MATLAB

Examining the MATLAB based HDL Coder version shows a large difference in the estimated max frequency in comparison to the other two tools, while using approximately the same amount of resources as the System Generator version. A reason might be that as there is no inherent timing in MATLAB as a tool it just implements the combinational sequence designed without any clocks to run it at a certain speed, why this would result in worse estimated timing performances is unknown.

Simulink

Simulink can be seen to not require any LUTs to implement its functionality thus not using any logical implementations at all. This means that all arithmetic's are performed by the DSP units and that any intermediaries are registered in the FFs. Evaluating the generated code confirms that this is the case. This could explain the high estimated maximum frequency as the DSP units are running at high speeds and as the calculations are registered between every stage the critical path is short, and the maximum frequency is that of one DSP stage.

5.2 Implementation methods and key points to be aware of

Chapter 4.2 collects information about some of the key concepts to be aware of when writing code for HLS tools, the main differences in these will be discussed here.

5.2.1 Fixed point conversion

Fixed point conversion in MATLAB is straightforward as it is integrated into the HDL Coder tools interface, utilizing the Fixed-Point converter tool. In Simulink and System Generator the Fixed-Point tool is used for the conversion but it is not automatically included in the interfaces for their respective HLS tools and has to be used separately. It is used to set the input ports of the Simulink subsystem to fixed point precision data types matching the data used for simulation of the subsystem. As the subsystem itself has to be built using the HDL Coder library no conversion has to be made internally as these blocks are mostly using fixed point precision for their calculations. System Generator follows a similar manner as its blocks also run mostly on fixed point precision data types, just using the Xilinx Blockset library in-

stead, and its Gateway blocks include a setting for fixed point conversion of data passing through them.

5.2.2 Supported functionality

Somewhat different functionality can be found and utilized based on which tool is used.

MATLAB and Simulink tools

The MATLAB based HDL Coder covers a wide range of both data types and operators and it is in this tool that I had the least amount of limitations when I was working on the histogram. Functions outside of the basic toolbox are rarely supported directly, but as most things can be manually implemented, using basic functionality, this is not an issue in more than that it slightly increases development time.

The Simulink based tool is powerful in that its available block library includes a lot of functionality as well as its excellent integration with the MATLAB function blocks.

System Generator

The System Generator environment also covers a wide range of basic building blocks, in the same way as the Simulink one, but its integration with MATLAB through the MCode blocks is very limited in comparison. It is only meant to be used as a way to implement arithmetic functions, finite state machines and control logic, as mentioned in section 4.2.4, and this limitation was clearly seen when the histogram was developed. The histogram version uses an MCode block for register addressing, storage and incrementation.

Using a for loop that spans the 256 histogram bins and an if statement comparing the bin number to increase, which is taken as an input, to the current loop count. Using this scheme instead of just using the bin number as an address straight to the register, as was done in Simulink is because the MCode block requires register addresses to be fixed. Using the bin number input as an address raises an error as it is a non fixed input signal. Most likely because no checking is performed to see if it is out of bounds or not.

The MCode blocks does not support register arrays as outputs which forced me to make 256 output variables instead of one array, each variable reading the value from one of the functions internal 256 register locations. All resulting in a lot of extra manual work.

5.2.3 Timing

Timing is handled in different ways in the tools and this leads to some major differences in the generated result.

MATLAB

Timing of the circuit is where the MATLAB version lacks behind the other two. As MATLAB code is untimed inferring timing at the HLS stage required manual intervention. Clocked registers can be added as a way of manually inserting pipeline stages either in the MATLAB code directly, this allows them to be placed anywhere, or using settings in HDL Coder during the generation process which in turn limits them to be used on the inputs and outputs of the function. Without these manual additions the generated code will be asynchronous.

Simulink and System Generator

Simulink and System Generator are time based simulation tools which generate a synchronous output by default. Code which is synchronous is generally safer and would typically result in a more robust output. This is preferable here when the code is automatically generated as it should result in a less error prone output.

5.2.4 Usability

This part covers the perceived usability of the different tools and is thus more of a qualitative comparison then the the other parts of this thesis.

MATLAB

MATLAB is a tool which a lot of people have encountered during e.g. their studies and MATLAB code ready to be used with HDL Coder share large similarities with MATLAB code written for many other use cases. As of this the initial learning path is very short and flat until one is able to produce code for HLS purposes using MATLAB.

One downside of the MATLAB method is that it is harder to know what functions are available to use for development than in the other tools. Finding this information forces the developer to go through the documentation for HDL Coder in full as compared to just opening to correct library.

Requiring manual intervention to properly run simulations in Vivado, on top of the automatically generated projects, is not good in the sense of usability

and neither are the manual registers required to keep the code from being asynchronous.

Simulink

If the MATLAB method has a short learning path the Simulink learning path is probably even shorter as it only requires one to change which library is being used. Most function blocks from HDL Coder library looks very similar to its counterparts in the default Simulink libraries and changing from one to the other is an simple task. If one has used Simulink before it is often also in conjunction with MATLAB thus making the process of developing MATLAB function blocks easier for specific tasks.

In the same way as with the MATLAB version requiring manual addition of the simulation files to the automatic Vivado project lowers the usability of this tool, especially as this is not an obvious requirement from a users perspective.

System Generator

System Generator is the only tool not developed by Mathworks and it uses some other systems which was found to be harder to understand. Even though it runs on the Simulink environment and looks in the same way as its counterpart by Mathworks it uses its own system for generating the HDL code. The System Generator interface was found to be lacking all optimization settings found in the Mathworks HDL Coder interfaces leaving this to be added manually by the user. Some settings were implemented on function blocks directly but exactly what was done was unclear and what extra optimization could be useful was hard to understand.

A strange bug was also encountered where the System Generator token somehow lost its connection to the underlying library leading to that the entire project had to be remade. Problems were also encountered with one of the tools other main settings, the connection between the Simulink simulation time period and the corresponding time period to be used when running the generated code on an FPGA. Not properly balancing these time periods with the sampling times on input gateways led to critical errors in the tool. No further examination was performed in regards to what strengths could be found in these settings after these issues had halted work on the thesis for some time.

5.3 Histogram implementations

Using the histogram as a test case has worked well as its been a rather simple system to model and thus there has been time to solve issues that arose during development as well as to iterate on the HLS implementations to improve performance of the resulting system. One drawback in correlation to using it as a test case for HLS tools tightly coupled with signal processing is the lack of intensive calculations.

5.3.1 Development time for the different tools

Development in Verilog was the slowest option but even within the HLS tools the time difference was large. Development time decreased by 27% while using MATLAB, 64% while using Simulink and 45% while using System Generator. Knowledge was gained during development and solutions found while developing the MATLAB version could be used in Simulink and System Generator to decrease the time spent to develop the histogram using these tools. This is likely one of the reasons why there is a large difference in development time between MATLAB on one hand and Simulink and System Generator on the other. Development in System Generator took place last but as several issues was encountered and the framework with MCode blocks was not as allowing as was the case with Simulink, the development time in this tool increased in comparison to Simulink.

The results are consistent with what was found by Shah et.al. [2] in that using HLS tools decreases development time in comparison to developing code directly in a HDL.

To also note in Table 4.11 is that the estimates include time spent to translate the base code to Verilog for Vivado usage and then verifying the functionality in Vivado using behavioural simulations. Time spent in Vivado to simulate and test the generated outputs has been minimal and most of time has been in the HLS tools which is what can be expected using these.

5.3.2 Non and default optimized histogram versions

Comparing non and default optimized HLS output, Table 4.12 - 4.14, is done as to get a bottom line of comparison, how inefficient are the tools when no optimizations settings are used.

Of interest is that the baseline Verilog implementation is not the worst and not the best in either of the tables of interest which indicates that in its basic form none of the tools tested is exceptionally better or worse than

direct HDL implementation. As optimizations are part of the tools they will in most scenarios be used to improve the results.

MATLAB

Examination of the generated code has shown that using the non optimized settings has generated code where every stage of the calculation is stored in its own register. This leads to that the algorithm is performed in a sequential order with one instruction at a time. Using the default settings has allowed the tool to run some parts of the code in parallel before registering the results of that section, a new parallel section the commences.

Table 4.12 shows that utilizing the default optimizations increases estimated maximum frequency by 61% at the cost of an 38% increase in resource usage while the estimated energy consumption is almost the same. Analysing the schematic indicates that this is likely a result from the increased number of LUTs which seems to be used as storage elements instead of the FFs.

Simulink

The code generated by HDL Coder in Simulink has also been inspected and the resource usage is the same because the code is identical between the two. The settings utilized by Simulink in its default mode does not affect the generation of code for this test case.

The default optimization settings did not improve the results in comparison to the non optimized case. This is likely as no pipelining is done internally in the HLL function the settings related to pipelining are not relevant. It was assumed from the start that multiplier sharing should reduce the amount of DSP blocks used but this did not happen. Functions using the DSP blocks are located in different subsystems which might be why Simulink has not applied this setting to them.

5.3.3 Optimized histogram versions

To further test the tools their ability to automatically help with pipelining was tested. No tool was able to automatically add pipeline stages within a function or subsystem so I had to settle with trying the setting for automatically adding pipeline registers to the input and outputs. HDL Coder, both MATLAB and Simulink versions, did this with a setting in the tool whilst System Generator lacked this functionality and a register was manually added to be able to compare the tools on an equal level.

MATLAB

In the MATLAB case, Table 4.15, the addition of the pipeline stage did the opposite to what was expected, it lowered the estimated max frequency. One possible reasons for this is that as MATLAB is not inherently clocked this added layer of synchronization makes the compilation process implement something more complex that what is necessary. This is not the main question of this thesis and it will not be further investigated.

Simulink

As seen in Table 4.16 adding registers on input and output increases the estimated maximum frequency by 20% while power requirements decreases by about 7%. It is interesting to see that these large differences are found but it may be related to that in the default version a type conversion was required on the output data. The addition of the output registers makes this conversion unnecessary and the tool has omitted it in the pipelined version. This stage was performed by a combination of LUTs and FFs and now that it does not have to be executed these resources are free, decreasing power requirements and increasing the estimated maximum frequency.

System Generator

Table 4.17 shows that manually adding the registers on input and output side of the System Generator version of the histogram increases estimated maximum frequency by about 8% while approximately doubling the required amount of FFs, indicating that all inputs and outputs are also registered in the default version of the code. This conclusion is drawn from the fact that as the FFs doubles in quantity the same amount existed before, likely in the same locations as well as that the gain in estimated maximum frequency is not major. Power consumption increases by about 7% which likely is because of the added FFs requiring power to operate.

Tool comparison

Looking at Table 4.18 it can be seen that by utilizing these settings all different tools use different amount of resources but they operate at approximately the same estimated maximum frequency of around 150MHz. In the MATLAB section of the optimized discussion it can be noted that MATLABs maximum frequency dropped when adding these registers but what it is that gives this change is unknown and will not be investigated further here.

Resource usage to achieve these implementations vary widely between the tools but the maximum resource usage in comparison to what is available on the target system used in Vivado is still very small. The biggest resource users are Simulink and Verilog. Simulink still only uses 5% of the available

LUTs and Verilog uses only 4% of the available FFs.

As can be seen in Table 4.18 the Verilog version utilizes a BUFG, a global clock buffer, which none of the optimized HLS versions use. Examining the code shows that the BUFG is used to activate the output registers of the Verilog version when valid data is received at the histogram input. The same control signal is implemented in the HLS tools but it is not implemented in the same manner in the generated output as it is in the Verilog version. This might be because that the tools handle the clock generation in such a way so that these registers match the base clock of the system, but that this is not the case in the manual Verilog implementation.

The histograms developed follow the same hardware scheme as the Dedicated Registers Version implemented by Geninatti and Boemo [4] who manage to run their 256-bin histograms at 206MHz using the Spartan 6 FPGA. How they have worked with pipelining or what languages they are using has not been stated in their paper but it is clear that they have not used any HLS tool for development. As of this comparing the results is not straightforward but it seems like they might have worked more on hand optimization of their code and in that way reached a higher speed using equivalent resources.

The results of this thesis differs from Shah et.al. [2] in that it has been found that resource usage and performance does not have to be negatively affected by using HLS tools. A main reason to why different results has been found is probably that the two studies are targeting algorithms in different parts of the signal processing domain. Shah et.al. worked on a communication interface which faces different challenges to the histogram that has been developed in this thesis. Another reason might be that an updated tool set has been utilized in this thesis (MATLAB 2018a) in comparison to what was used by Shah's team (MATLAB 2016a). To be noted is that Sarge [1] used the MATLAB 2017a version and found improved results in comparison to Shah et.al. with a problem in a similar sector as they had been working, this could imply that the tools are the main factor and that they are quickly gaining in performance.

Chapter 6

Conclusion

Instructions has been summarised on how to structure and write code using all three different HLS tools to generate proper output with the intended functionality.

A summary of how the different tools compared (ranked best to worst) can be seen in Table 6.1. Numerical values are based on Table 4.18 data only as this was deemed to be the table giving the most identical comparison.

Table 6.1: *HLS tool ranking.*

Evaluation Criteria	**Best**		**Worst**
Resource Usage	MATLAB	System Generator	Simulink
Estimated max frequency	System Generator	MATLAB	Simulink
Estimated power consumption	MATLAB	System Generator	Simulink
Usability and functionality	Simulink	MATLAB	System Generator
Development time	Simulink	System Generator	MATLAB

Based on this test case, a histogram implementation, the tools are judged to work more or less equally well with Simulink faring worse as it was not as resource, speed and power efficient. At the same time I would say that the Simulink based tool functions way above the other two in regards of usability and functionality, which is likely the main reason to why it was the tool with

the shortest development time. Simulink would be my tool of choice based on that as well as that there are only minor differences in the other parameters.

The results of this thesis also shows that implementation utilizing any of these HLS tools results in an output which is comparable to directly designing the Verilog code itself in regards to operation frequency, the resource utilization is reduced and it has been shown that development time decreases when using a HLS tool during development. Power consumption is the one criteria that is not positively affected by the use of HLS tools according to this study.

6.1 Future Work

Areas not investigated in this thesis but which are related to the same topic and which could provide an approach to further studies are for example:

- Investigating ways of improving the power consumption of algorithms developed with the help of HLS tools.
- Testing newer version of these tools when available to investigate if the performance continues to improve.
- Implementation in resource confined environments and investigate the possibilities of the resource optimization settings in these tools.
- Working with computationally heavy algorithms and investigate how this affects the output.
- Comparing the efficiency of implementation of different types of algorithms, using the different tools (as in Georgopoulos et.al. [16]).
- Working with several different clock regions within one system and test the possibilities in the Simulink and System Generator tools connected with this.

Bibliography

[1] V. Y. Sarge, "Evaluating simulink hdl coder as a framework for flexible and modular hardware description," Master's thesis, Massachusetts of Technology, 2018.

[2] R. K. Shah, T. Kumar, A. Fell, M. S. Dohadwala, and R. Malik, "Executable model based design methodology for fast prototyping of mobile network protocol: A case study on mipi lli," in *2017 4th International Conference on Signal Processing and Integrated Networks (SPIN)*, Feb 2017, pp. 346–351.

[3] G. Baguma, "High level synthesis of fpga-based digital filters," Master's thesis, Uppsala University, 2014.

[4] S. Geninatti and E. Boemo, "A proposal of two histogram circuits to calculate similarities between video frames using fpgas," in *2019 X Southern Conference on Programmable Logic (SPL)*, April 2019, pp. 103–108.

[5] S. M. S. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, Spring 2018.

[6] J. Teubner and L. Wood, *Data Processing on FPGAs*. Morgan & Claypool, 2013.

[7] D. Maskell, "Ce2003: Digital systems design," 2019, as presented during spring term 2019 at NTU, Singapore. Request material via email, asdouglas@ntu.edu.sg.

[8] Xilinx, "Ug574 - ultrascale architecture configurable logic block, v1.5," 2019, accessed on the 28/8-2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[9] L. M. Hiot, "Ee2004: Digital electronics," 2019, as presented during spring term 2019 at NTU, Singapore. Request material via email, emhlim@ntu.edu.sg.

[10] Xilinx, "Ug579 - ultrascale architecture dsp slice, v1.8," 2019, accessed

on the 29/8-2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf

[11] ——, *DS891 - Zynq UltraScale+ MPSoC Data Sheet: Overview*, 2019.

[12] E. A. Bezerra and D. V. Lettnin, *Sythesizable VHDL Design for FPGA*. Springer International Publishing Switzerland, 2014.

[13] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Autom. for Emb. Sys.*, vol. 16, pp. 31–51, 09 2012.

[14] N. Kosugi, K. Hori, Y. Ishida, and M. Hasegawa, "Driving the adoption of model-based design for communications system development at hitachi," 2013, accessed on the 3/9-2019. [Online]. Available: https://www.mathworks.com/company/newsletters/articles/driving-the-adoption-of-model-based-design-for-communications-system-development-at-hitachi.html

[15] P. Coussy and A. Morawiec, *High-Level Synthesis - From Algorithm to Digital Circuit*. Springer, 2008.

[16] K. Georgopoulos, P. Malakonakis, N. Tampouratzis, A. Nikitakis, G. Chrysos, A. Dollas, D. Pnevmatikatos, and I. Papaefstathiou, "Comparing c and systemc based hls methods for reconfigurable systems design," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, N. Voros, M. Huebner, G. Keramidas, D. Goehringer, C. Antonopoulos, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2018, pp. 459–470.

[17] Accellera Systems Initiative., "About systemc," https://accellera.org/community/systemc/about-systemc, 2020, [Online; accessed 20-February-2020].

[18] MathWorks, "Matlab," https://se.mathworks.com/products/matlab.html?s_tid=hp_products_matlab, 2019, [Online; accessed 17-January-2020].

[19] R. Nane, V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct 2016.

[20] MathWorks, "Hdl coder v3.13," 2018, accessed on the 27/8-

2019. [Online]. Available: https://se.mathworks.com/products/hdl-coder.html?s_tid=srchtitle

[21] Xilinx, "System generator for dsp," 2019, accessed on the 4/9-2019. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html

[22] D. A. Patterson and J. L. Hennessy, *Computer Origanization and Design*, 4th ed. Morgan Kaufmann Publishers, 2009.

[23] B. Ronak and S. A. Fahmy, "Improved resource sharing for fpga dsp blocks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–4.

[24] K. E. B. Ahmed, R. A. Mokhtar, and R. A. Saeed, "A new method for fast image histogram calculation," in *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*, Sep. 2015, pp. 187–192.

[25] D. G. Bailey, *Design for Embedded Image Processing on FPGAs [Electronical resource]*. Hoboken: John Wiley & Sons, 2011.

[26] MathWorks, "Getting started with matlab to hdl workflow," 2019, accessed on the 15/10-2019. [Online]. Available: https://se.mathworks.com/help/hdlcoder/examples/getting-started-with-matlab-to-hdl-workflow.html

[27] ——, "Matlab 2018a," February 2018, accessed on the 27/8-2019. [Online]. Available: https://se.mathworks.com/products/matlab.html

[28] ——, "Simulink," February 2018, accessed on the 21/1-2020. [Online]. Available: https://se.mathworks.com/products/simulink.html

[29] X. f. m. arpansur, "determine maximum frequency at which a circuit can run," 2016, accessed on the 22/10-2019. [Online]. Available: https://forums.xilinx.com/t5/Timing-Analysis/determine-maximum-frequency-at-which-a-circuit-can-run/td-p/703734

[30] MathWorks, *HDL Coder User's Guide R2018a*, 2018.

[31] ——, *MATLAB Programming Fundamentals R2018a*, 2018.

[32] ——, "Fixed-point converter," 2018, accessed on the 23/10-2019. [Online]. Available: https://se.mathworks.com/help/releases/R2018a/fixedpoint/ref/fixedpointconverter-app.html?searchHighlight=fixed-point%20converter&s_tid=doc_srchtitle

[33] J. Erickson, "Hdl coder self-guided tutorial," 2019, accessed on the 30/8-2019. [Online]. Available: https://www.mathworks.com/matlabcentral/fileexchange/69651-hdl-coder-self-guided-tutorial

[34] Xilinx, *UG897 - Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator*, 2018.

[35] ——, *UG958 - Vivado Design Suite Reference Guide Model-Based DSP Design Using System Generator*, 2018.

Acknowledgement

I would like to thank my supervisor Per Lindgren for his support during this thesis and quick responses when questions have arisen.

I would also like to thank Grepit AB for granting me access to the server based work environment, as well as a software license for the Vivado tool, utilized during testing and development without which this thesis would not have been possible to conduct as has been done.

I would also like to thank my dear friend Jeffrey Duong-Boudrias for his help with proofreading parts of the manuscript.