Performance Evaluation of MathWorks HDL Coder as a Vendor Independent DFE Generation

ELISABETH PONGRATZ
ROSHAN CHERIAN
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

LUND UNIVERSITY
Lund University EITM01/EITM02
Master thesis report

August 27, 2019

Performance Evaluation of MathWorks HDL Coder as a Vendor Independent DFE Generation

Master thesis

By
Elisabeth Pongratz and Roshan Cherian
Department of Electrical and Information Technology
Faculty of Engineering, LTH, Lund University
SE-221 00 Lund, Sweden
2019

Elisabeth Pongratz elt14epo@student.lu.se
Roshan Cherian ro6211ch-s@student.lu.se

1 Abstract

This thesis aims to evaluate MathWorks HDL (Hardware Descriptive Language) Coder and compare the results with designs produced by its vendor dependent counterparts. The focus is mainly on evaluate the design effort needed to close timing and to get optimal resource mapping for a selected design. Different tests were carried out with the designs made in HDL Coder ranging from a small design to a complex digital front end design. Several logic tests were also carried out to understand the efficiency of the hardware produced from HDL Coder. To get our hands on with the tool, a power meter was undertaken as the design under test. Further, a down sampling filter chain was designed in HDL Coder and lastly, a branched filter was designed. Results have been presented for the unoptimized design and the various optimizations tried. An analysis of the different methodologies used is discussed and how the tool can make the design flow more effective. With the time invested in learning and using the tool, an experience of HDL Coder as a HLS (High Level Synthesis)) tool can be analyzed and compared to other HLS tools.

In conclusion, it was found out that in the bigger designs HDL Coder did not have a problem meeting the timing requirements for the selected designs and the results are comparable. However, it does not map optimally to the resources on the FPGA (Field Programmable Gate Array) fabric and it was difficult to change the mapping of resources according to constraints. On analysis, the HDL Coder blocks consume a lot of DSP (Digital Signal Processing) slices compared to the vendor dependent counterparts. Though, the tool is faster and easier to learn, work with and many optimization methods can be done automatically by the tool. Additionally, there is no need to run synthesis on a separate vendor cause this can be done by HDL Coder which improves the workflow. Using HDL Coder in a design methodology would increase productivity because of it being vendor independent. Whereas, the performance results can vary depending on design structure.

Keywords: **MATLAB**, **Simulink**, **HDL Coder**, **XSG**, **DSPB**.

2 Acknowledgements

We would like to thank LTH, Lund University and Ericsson AB for giving us this opportunity to pursue this Master thesis. In addition, we would like to express gratitude to a few people in particular. Special thanks to: our supervisor at the university, Professor Liang Liu for ensuring we ask the right research question and motivate our thesis around it. Also, great thanks to our supervisors Adeel Tajammul and Naeem Abbas at Ericsson for helping us out with their great knowledge, feedback and the review meetings. Similarly thanks to our manager, Sacki Agelis for giving us the support and the resources needed to fulfill this thesis. Thanks to Lakshmi Prasad for helping us out during the initial days to get started with the designs and continued help during the thesis.

We are also grateful for our parents who have been the pillars because of which we are here today. Additionally, in case we have missed out, we would also like to thank everyone who have contributed to this thesis in anyway possible.

Contents

List of Tables

List of Figures

Acronyms

ALM Adaptive Logic Modules.

ALUT Arithmetic LUT.

ASIC Application-Specific Integrated Circuit.

CC Carrier Combiner.

CLB Configurable Logic Block.

CSD Canonical Signed Digit.

DFE Digital Front End.

DSP Digital Signal Processor.

FCSD Factored Canonical Signed Digit.

FFT Fast Fourier Transform.

FIR Finite Impulse Response.

FPGA Field-Programmable Gate Array.

FT Frequency Translator.

HDL Hardware Descriptive Language.

HLT High Level Tool.

IIR Infinite Impulse Response.

IP Intellectual Property.

LUT Look Up Table.

MBD Model-Based Design.

mux Multiplexer.

NCO Numerically Controlled Oscillator.

RAM Random Access Memory.

RX Receiver.

TX Transmitter.

VGA Voltage Gain Amplifier.

3 Popular Science Summary

The number of transistors on a silicon chip has doubled every two years as per Moore's law and the design requirements have been exponentially increasing. Wireless systems are also getting more and more complex. With the development of 5G, this has been no exception. This complexity increases the design workload which makes the design flow multiple times longer and puts a lot of pressure on the designer. These complex systems generally contain a mix of DSP algorithms such as FFT's, IIR filters, FIR poly-phase filters. With the advancements in the FPGA technology, designs can be built on them with great flexibility. Since the time to market is critical, the development time window is reducing drastically. This is where High-Level Synthesis (HLS) and Model-Based Design (MBD) tools fit in. MBD is a methodology which lets the designer creates models of the design implementation and promotes the use of the high level synthesis within it to create RTL relatively easily.

HLS tools have been designed to automate and accelerate design by moving manual work on to a higher level. FPGA vendors have come out with high level tools(HLTs) optimized specifically for their hardware and are integrated within the MBD tool Simulink. These tools are vendor specific and the learning curve can be quite steep and a deeper understanding of hardware is needed to use them. One of the problems with this vendor specific approach arises if the same design needs to be ported on to different FPGA vendor platform. This causes the same design to be replicated and re-implemented in the other vendor specific HLT. This procedure is time consuming and requires knowledge of the new tool. One other tool which looks promising and produces vendor independent synthesizable HDL code is HDL Coder provided by MathWorks. In this thesis, Simulink is the MBD tool used along with the HLTs like HDL Coder, Xilinx SysGen and Intel DSP builder.

In this thesis, a few experimental designs of a complex filter chains is done with HDL Coder. HDL Coder like the other architecture based design tools is a HLT that can be used to generate HDL code from Simulink and MATLAB algorithms. The resulting HDL code can be further synthesized into any FPGA. The design and results from HDL coder is compared to the results from same design implemented in vendor dependent tools: Xilinx SysGen (XSG) and Intel DSP Builder(DSPB). All the HLTs are evaluated with a major focus on HDL Coder on the metrics of ease of implementation, RTL produced, design space exploration, resulting speed and area consumption as well as the learning curve.

4 Introduction

Currently, hardware design companies are running several products with several different vendors for which the digital designs are generated separately using vendor specific RTL generation tools. These tools are excellent in generating digital designs quickly compared to the handwritten HDL code.

HDL Coder from MathWorks is a HLT that is used to generate vendor independent HDL code from MATLAB and Simulink. This tool is evaluated in this thesis work and compared with the vendor dependent counterparts. The vendor dependent HLT tools almost always needs additional interconnections, also called as the glue logic which are not available within these blocks and require significant amount of time to design and verify. This results in structurally lowering the level of abstraction of the design. This costs a lot of working hours which could be reduced by reusing these blocks from a local block library which can be effectively done in MATLAB. These blocks have also been identified and HDL supported designs have been designed for use in the future.

HDL Coder can generate HDL for different FPGAs from the standard Simulink library blocks. Thus, in case there are similar requirements for a different FPGA, the designer can switch vendors for the same implementation of the design easily and would not have to redo the design and the verification for it. Since, many industries are increasing its FPGA product development due to its economic standpoint, investigating options to improving efficiency and productivity of development process is worth exploring.

The prior state of the art does not make comparisons between different HLT tools and have previously compared HDL Coder to HLS tools and hand written RTL code. Related thesis works have also set a foundation on how to use the HDL Coder and listed some of the best practices. These have come in handy and made the learning curve for the thesis smooth. Most of the previous work also does not quantify the performance aspects of the tools from the different vendors. This thesis aims to do that and suggest a methodology for a vendor independent use case.

4.1 Motivation

The motivation of this Master's thesis is based on the need to explore a performance effective tool to generate vendor independent HDL that could save development time and is comparable to its vendor dependent tools. HDL Coder fit into these tool requirements and is to be evaluated in this thesis where both the efficacy and efficiency of HDL Coder has been explored. A list of best practices and possible solutions to problems is also covered.

There exists vendor specific architecture-level design tools for implementing high performance DSP algorithms for a FPGA. The ones widely used in the industry nowadays are DSPB and XSG. Both of the these tools are vendor specific i.e. work on the FGPA's manufactured by them and operate more on the block level with highly optimized blocks for commonly used sub blocks. These tools are integrated in the Simulink MBD environment and are used seamlessly within this environment to produce designs. However,

the optimization methods on the system level are quite limited for these two tools. One other tool used with the Simulink MBD environment available in the industry is HDL Coder which is vendor independent has not been explored much at Ericsson for FPGA prototyping and design. HDL Coder also provides system level optimizations to explore the design space. In this thesis, one of the previously implemented designs on the Xilinx/Intel FPGAs is implemented using HDL Coder and various performance metrics involved with the design are evaluated and possible alternatives have been suggested. It also throws light into design productivity and IP reuse as the design made in HLT tools can be easily ported to new designs.

4.2 State of the Art

There has been prior work on similar topics. The learning from sources [2] and [14] have been used to pave the way for this thesis. The source [2] though a little outdated (2010), shows the flow and contains tips and tricks that can be used by the designer. It also discuses some of the optimization techniques tried by the user to meet the timing constraints. It describes the design and the implementation of software defined radio (SDR) with the HDL Coder based design implemented on Altera Cyclonce II platform. The conclusion for this thesis is that it is easy to understand the programmable logic tools but the result may vary a bit. Source [14] is relatively new (2018) and has taken a similar approach but is comparing the HDL Coder results to hand written HDL code. Valerie Youngmi Sarge claims in this thesis that HDL Coder design offered reasonable performance with greater ability to parameterize a design and easily visualize and re-use portions of an existing design. Traditional forms of hardware description still give a more highly optimized and greater degree of control over pipelining and reset and enable behaviour.

Reference [4] is another example which has compared the Vivado HLS tool with the HDL coder as an HLS tool. The thesis work is done by Gerald Baguma in 2014. It concludes that as a HLS tool Vivado HLS generated more efficient designs, because of it provide more granularity to control of scheduling and allocation, than HDL Coder. This thesis also list the various HLS tools that were available then. Additionally, it is concluded that the HLS tools provided today does not meet the requirements from FPGA/ASIC designers. The thesis is from 2014 and there has been a few updates to HDL Coder since then. There had been some options available then in HDL Coder which have been disabled or discontinued from this version which have been made note of. Also used is the source from 2014 [3] by Mathew S. Allen. This thesis is evaluating the performance of designs using HLT using HDL Coder and hand-written RTL code and the pros and cons of each. His conclusions were that HDL Coder has a high productivity once the tool is understood, though, he proves that it produces inefficient design. The design consumed almost double of the resources used for the manually optimized design.

Also a bit out dated (year 2015) is the thesis work for reference [12], made by Joonas Järviluoma. This thesis work is mostly evaluating HDL Coder as a HLS tool. Additionally, he is looking into the generation flow with HDL Coder for writing HDL, as for coding styles and synthesizable RTL code produced.

In the source [15] the people involved have made a comparison between HDL Coder and hand written RTL with the case of methodology and time spent. Their result is positive for HDL Coder, in the manner of time spent, though, negative for the other parameters as area, power consumption and operating speed.

To summarize, the current state of the art gives a brief introduction to HDL Coder and compares the results with HLS tools and handwritten HDL. The tool have also improved ever since and is fairly accepted in the industry with positive case studies and scenarios. There have not been any comparisons so far of the different HLT tools using the MBD workflow and this thesis aims to do that.

4.3 Thesis Structure

The new design workflow methodology and tools makes it a requirement to do an elaborate pre-study. The pre-study is made based on the HDL Coder toolbox, tutorials and previous thesis work and related work section 4.2 on the subject.

Chapter 5 gives the background needed to understand this thesis. It explains the current and the proposed workflow. It also gives a brief information of the tools used in this thesis and their optimization techniques. Section 6 provides the information needed to understand the HDL Coder tool that was used.

A design already implemented on the Xilinx and Intel FPGA was given. All of the produced designs in this thesis is done with MATLAB version 2018a and is used within a Linux virtual server based system. This design consists mainly of FIR filters, down samplers, decimators, interpolators and a NCO. Also, MathWorks offers support to assist in the design of optimized filter chains for different vendors. A common subsystem, the power meter, was observed that is present in almost all the designs. This power meter, which can be used across future designs was a good beginning to go through the flow and gain more hands on knowledge of the tool. An understanding of what each subsystem can do and how the logic can be recreated was needed. The power meter flow was documented in chapter 7. Once the experience with the new workflow and tools has been gained then the design of the filter can be created and tested on. This is covered in section 8. Further, a bigger branched design was made to investigate optimizations with a bigger design, see section 9. Further logistic test were done, which is explained in section 6.5.

Once all designs have been created and the optimization of each has been done, evaluation of the HDL Coder tool has been done. Further comparison of the tool with other work flow options, problems faced and work around are also suggested in this topic. This is documented in chapter 10. Also during the project, the time taken was recorded and evaluated to get a rough estimate of the time needed for research and learning so as to find out how much time is needed to get the HDL Coder up and running in terms of working man-hours, this can also be observed in section 10. Additionally, the hours used for optimization also needs to be documented. A summary of the results and this thesis are made in section 10.6. Finally, possible future work within this topic are discussed in the chapter 10.7.

5 Background

In this section the background information needed for the thesis is presented. First an explanation of the current methodology used for FPGA based design. Second, the proposed methodology with HDL Coder instead is described. Last, an explanation of the different FPGA fabrics concerning each FPGA.

5.1 Digital Front-End

For wireless base stations the Digital Front-End (DFE) is the most vital component. The DFE along with an analog to digital converter handles the path from radio and intermediate frequency process to the digital baseband process. The signal processing is done on a software programmable DSP before the baseband processing can be preformed. Digital signal processing is used because it can in real time reconfigure the radio frequency channels in the base station. Therefore, multiple signal conditions, compensations and limitations to channel non-linear responses can get implemented. [8] The functionality of a DFE can be derived from the characteristics of the input and output signals of the DFE. The output is a digital signal with a specific sample rate controlled by the DFE. Lastly, this signal can then be baseband processed. [9]

5.2 Current Methodology - Vendor Specific Workflow

The current methodology of FPGA based design go through the specific steps, as follows. First requirements for a particular design are set in a specification. Then the functionality is split into various functional blocks. These functional blocks require their own block specification from this a reference model for this design is made in Simulink. If the target FPGA is a Xilinx based device, the design is done in SysGen with Simulink and compared with a reference model in Simulink. The delays in the reference model are then adjusted to account for latency to make it cycle and bit accurate with respect to the reference model. Suppose if the same design is to be implemented on the Intel FPGA device then the same design has to be redesigned using DSP Builder because the earlier design is not valid in this case. Additionally, the delays are adjusted to make this design cycle and bit accurate as in the previous case. When the design is working or during the design time some optimizations has to be done. [15]

This methodology is time consuming and inefficient, involves repetition of the design process and needs a person to be skilled in all the three software working with the project. Therefore, there might be a need for more than one person working with the project with each having expertise in Simulink, XSG and Intel DSPB in order to make the designs work.

5.3 Proposed Methodology - Vendor Agnostic Workflow

The specifications for the design and blocks is set and the functionality in each block is decided. From these specification the designer can create a reference model in Simulink. The reference model from the current methodology can be directly taken and a new model can be made using only blocks supported by HDL Coder and/or HDL Coder MATLAB function blocks. The design then gets optimized to fit the requirements. Lastly, this model can generate HDL for both Xilinx and Intel FPGAs. In this methodology there is no need to redesign the same IP which saves a considerable amount of time. Moreover, the tools is known for its fast prototyping targeting toward FPGAs.

5.3.1 High Level Synthesis

HLS tools has provided the possibility of software programmers to target hardware, FPGA and ASICs more rapidly. Functionality or behavioural logic is written in software which then by the tool can be turned into RTL code, VHDL or Verilog. The tools provide scheduling, allocation of the logic onto the hardware and in most cases the tools automatically generate cycle-by-cycle details for the implementation. This essentially means that the designer do not have to be an expert in those fields and can focus on getting the functionality implemented. This gives us a tool with a higher level of abstraction than designer's hand written RTL.

Below are some of the many HLS vendors available today. They support different programming languages and provide different HDL outputs (RTL, verilog, VHDL or systemC) and constraints as fixed or floating point.[13]

- Bambu from PoliMi
- Stratus from Cadence
- C-to-Silicon from Cadence Design Systems
- Catapult C from Mentor Graphics
- Intel High Level Synthesis Compiler from Intel FPGA
- Synphony C from Synopsys
- Vivado HLS from Xilinx
- HDL Coder from MathWorks

As a HLS tool, HDL Coder workflow can produce results comparable with hand coding, given that the tool and it's optimization settings are correctly used. There is a possibility to write the functionality in a MATLAB function block and add it in the design and HDL Coder will provide the RTL for that logic. This method is much faster than the HLS tools available in the market. A HLS tool automatically takes care of the scheduling, resource allocation and more, which normally would take a lot of time for the person working with it. Though, this include that the person would have to trust the tool which

comes with continuous use and expect it to automatically provide somewhat good results. Otherwise, HDL Coder provide good traceability between the RTL code produced and the MATLAB function block. It is also possible to use HDL Coder to produce HDL Code from Mealy and Moore State Machines designed in Simulink with Stateflow.[13]

5.3.2 High Level Tools - HDL Code Generation

The core principle of MBD workflow is the capability of using a model in the design process to simulate and verify the functionality before the creating the real system which has been explored with the HLT listed below. In this thesis, Simulink from MathWorks is used as the MBD tool. LabVIEW from National Instruments is one another alternative but does not have any support from vendor specific HLT. Vendor dependent HLT are highly optimized for their own hardware especially for filter chains because the blocks are hardwired. HDL Coder on the other hand, considers the whole design and analyzes the functionality and then proceeds with the implementation.

- Xilinx System Generator
  Xilinx System Generator XSG is high-level software tool that enables the use of MATLAB/Simulink environment to create and verify hardware designs for Xilinx FPGAs quickly and easily. It provides a library of Simulink blocks bit and cycle accurate modelling for arithmetic and logic functions, memories, and DSP functions. It also includes a code generator that automatically generates HDL code from the created model. Generated HDL code can be synthesized and implemented in the Xilinx FPGAs. The XSG blocks are like standard Simulink blocks except that they can operate only in discrete-time.[17]

- Intel DSP Builder
  DSP Builder (DSPB) for Intel FPGAs is a digital signal processing (DSP) design tool that allows push-button HDL generation of DSP algorithms directly from the MathWorks Simulink environment on Intel FPGAs. The tool generates high quality, synthesizable VHDL/Verilog code from MATLAB functions and Simulink models. The generated RTL code can be used for Intel FPGA programming. DSP Builder for Intel FPGAs is widely used in radar designs, wireless and wireline communication designs, medical imaging, and motor control applications.

  DSP Builder for Intel FPGAs adds additional library blocks alongside existing Simulink libraries with DSP Builder for Intel FPGAs (Advanced Blockset) and DSP Builder for Intel FPGAs (Standard Blockset).[11]

- MathWorks HDL Coder
  HDL Coder generates portable, synthesizable Verilog and VHDL code from MATLAB functions, Simulink models, and Stateflow charts. The generated HDL code can be used for FPGA programming or ASIC prototyping and design. It also supports Hardware/Software co-design targeting SoC targets like Xilinx Zynq and Intel SoC. In this case both HDL and C code for embedded CPUs are generated.

  HDL Coder provides a workflow advisor that automates the programming of Xilinx, Microsemi, and Intel FPGAs. You can control HDL architecture and implementation, highlight critical paths, and generate hardware resource utilization estimates.

HDL Coder provides traceability between your Simulink model and the generated Verilog and VHDL code, enabling code verification for high-integrity applications adhering to DO-254 and other standards.

HDL Coder produces vendor independent RTL. Hence the generated RTL can be ported to FPGAs of any platform and even an ASIC based on the design requirements. This allows the same design to be reused for future projects and also enables ease to change platforms when necessary. The level of detailing in the vendor dependent tools is a lot and requires deep knowledge of the hardware to implement them. Whereas, in HDL Coder, there are limited options available for block configuration. This drastically reduces the design time. Moreover HDL Coder is highly recommended for fast FPGA prototyping.[7]

5.4 FPGA Fabric

The overview of the resources utilized by the different vendor is given below and the terms are explained briefly. Since this thesis work focuses on porting the design on a FPGA these terms are important to understand the resource utilization metrics mentioned in the further chapters.

The synthesis tool by the vendor uses the RTL produced from the HLTs and maps it onto the FPGA fabric. Since the FPGA fabric is different for different hardware vendors, the RTL written for a particular FPGA design will not map well with any other FPGA fabric leading to inefficient resource usage. For example, to convert from a Xilinx to Intel design flow all the Xilinx primitives and Xilinx specific IP's needs to be replaced by the Intel ones. Also, because the standard multipliers sizes within DSP are different across devices. HDL Coder creates the RTL based on the synthesis tool selected and thus maps the same design on different FPGA's. The design generated in this thesis are designed to be implemented on the Xilinx Ultrascale+ xczu15eg-ffvb1156-2-i device and the Intel Arria 10 AX115R3F40I2LG.

5.4.1 Xilinx

Configurable logic blocks (CLB)
Every CLB in the Zynq Ultra-scale+ architecture contains one slice with 8 six-input LUTs and 16 storage elements. The LUTs are organized as a column with an 8-bit carry chain per CLB, called CARRY8. Wide-function multiplexers combine LUTs to create any function of seven to nine inputs, or some functions of up to 55 inputs. SLICEL is the name used to describe CLB slices that support these functions, where the L is for logic. These are mapped as CLB LUTs by the synthesis tool. The LUTs in a SLICEM, where the M is for memory, can be configured as a look-up table, 64-bit distributed RAM, or a 32-bit shift register. These are mapped as CLB Registers by the synthesis tool.

Block RAM
The block RAM in Zynq UltraScale+ architecture-based devices stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM.

Arithemetic DSPs
The Zynq UltraScale+ devices have many dedicated low-power DSP slices, combining high speed with small size while retaining system design flexibility. The DSP slice in the UltraScale+ architecture is defined using the DSP48E2 primitive and the slice is referred to as either DSP or DSP48E2 by the synthesis tool. Each DSP48E2 slice has a two-input multiplier followed by multiplexers and a four-input adder/subtracter/accumulator. The DSP48E2 multiplier has asymmetric inputs and accepts an 18-bit two's complement operand and a 27-bit two's complement operand. The multiplier stage produces a 45-bit two's complement result in the form of two partial products. These partial products are sign-extended to 48 bits in the X multiplexer and Y multiplexer and fed into four-input adder for final summation. This results in a 45-bit multiplication output, which has been sign-extended to 48 bits. Therefore, when the multiplier is used, the adder effectively becomes a three-input adder. [18]

5.4.2 Intel

Combinational ALUT:
Combinational ALUT is used to indicate half-Adaptive logic moduleALMs used in the design. The ALM consists of 8 input fracturable look-up table with four dedicated registers.

Dedicated logic registers:
Dedicated Logic Registers refer to the four registers in the ALM, these are dedicated to being registers.

Together, the combiantional ALUT's and dedicated logic registers form an ALM in the Intel Arria 10. The Arria 10 uses an enhanced 8-input ALM with 4 registers and they are grouped as combinational ALUT.

Block memory bits:
The Intel Arria 10 devices contain two types of memory blocks: MLAB and M20K (blocks of dedicated memory resources). The block memory indicates the number of bits used by the design in both the above stated memories.

DSP blocks:
The Intel Arria 10 variable precision DSP blocks support fixed-point arithmetic and floating-point arithmetic. It could map two 18 x 19 multipliers or one 27 x 27 multiplier per DSP block. [10]

5.5 Chapter Summary

This chapter provides the background on the design methodologies currently used in Ericsson for a model based design workflow with vendor dependent tools. This can be reduced by using HDL Coder workflow which is presented as the proposed workflow. This eliminates the re-design time to target a different vendor. This chapter also presents the different FPGA fabric and terminologies covered in the thesis.

6 MathWorks HDL Coder

HDL Coder from MathWorks embedded in the MATLAB/Simulink environment allows a designer to generate synthesizable HDL code for FPGA and ASIC implementations in the following steps:

- Build a model of the design using a combination of MATLAB code, Simulink and State flow charts.
- Simulate and check behaviour.
- Make the model HDL compliant.
- Convert to fixed point if needed.
- Generate HDL Code and synthesize.
- Optimize and iterate the design to meet area-speed objectives.
- Generate the design using the integrated HDL Workflow Advisor for MATLAB and Simulink targeting different vendors.
- Verify the generated code using HDL verifier.[1]

The HDL Coder is basically a compiler which translates the Simulink model to a data and control flow intermediate representation and performs many structural optimizations to produce the RTL.Whereas for the HDL Coder workflow, the code is converted to a control flow representation and then this goes through the data flow graph conversion before the structural optimization kicks in. Once the model is designed as per specifications, design space exploration can be done on them. The design points can be check pointed using the commands *hdlsaveparams* and *hdlrestoreparams*. The workflow advisor also helps to anotate the model with the synthesis results and helps you in further analysis.[5]

6.1 HDL Supported Blocks

HDL Coder can generate code from algorithms built using the HDL Coder library of blocks. These blocks are compatible with HDL Code generation and pre-configured with HDL-friendly settings. The HDL Coder library blocks come with Simulink, so you can share your models and collaborate with colleagues who may not have access to HDL Coder. Sub-libraries such as *HDL RAMs* and *HDL Subsystems* provide blocks specific to HDL applications, e.g. RAMs, and subsystems with synchronous enable/reset control inputs. Additional blocks for signal processing, communications and computer vision are also available. Apart from these toolboxes, HDL Coder also supports Stateflow. This allows state charts, state transition table and truth table to be implemented on HDL Coder.

[1] The HDL Verifier is not within the scope of this thesis work.

In HDL Coder, it is also possible to place a MATLAB Function block in which an algorithm can be written and RTL code can be produced provided the supported MATLAB HDL functions and constructs are used. When there is a need to create control logic that is not available in standard Simulink blocks but do not want to put the logic together with smaller adders or products. For instance an if-else mux or a simple finite state machine, you can use a MATLAB Function block embedded in the Simulink model. This is an easy way to create custom logic using MATLAB but integrated with the algorithm.

6.2 Design Considerations

Reset
There is no need to consider the resets and enables in the design as HDL Coder takes care of it internally. While a synthesis tool can faithfully implement either synchronous or asynchronous reset logic, matching the reset type to the underlying FPGA architecture will result in better resource utilization and performance. It is very important that the proper reset is selected as a improper reset selection can lead to poor and inefficient hardware mapping. Following are the reset that should be used based on the design target.
Xilinx -> Synchronous
Intel -> Asynchronous

Multi-rate modelling
Multi-rate modeling is possible but only with the Simulink HDL flow. The sample rates are shown with different coloured signals. The timing logic for different rates are automatically taken care of by HDL Coder. Additionally, no explicit clock system is needed. HDL Coder creates a timing controller based on the selected clock input selection. If a single clock system is selected for a single rate design, one sample time maps to one clock cycle in HDL, whereas, for a multi-rate model fastest sample time maps to one clock cycle in HDL. Blocks operating at a slower sample time have the same clock in HDL but gated with different clock enable based on the rate it is operating.

Block design
Functions are implemented as smaller blocks and placed inside subsystem which enable ease in design. Each subsystem has some parameters and optimization options. Some parameters are set as *Inherit* by default, this means that it is inherited by the hierarchy above. This is an important part in optimizing the design.

RTL code
The RTL code can be produced in VHDL and in Verilog. One unique thing about HDL coder is that it produces code-to-model and model-to-code traceabilty. This enables designers to move through their models seamlessly from requirements to HDL code, while maintaining the capability to add legacy code to the design. Additionally, HDL Coder creates different files in the RTL code for different subsystem which makes it easily accessible and readable. Thus, it can be good to make individual subsystems for big functionality.

Data types
If the model designed is using floating point data types, the MATLAB Fixed-Point

Designer from MathWorks can be used for changing the design to a fixed point one which is needed to generate the HDL code using HDL Coder toolbox. Single precision floating point is supported by HDL Coder though the use of Native Floating Point. When simulating the model more precision is obtained when floating points are used but since efficient and highly optimized HDL designs are needed it is more often converted to fixed point design to be implemented on the hardware. In HDL Coder a mix of floating point and fixed point is possible. [7]

6.3 HDL Workflow Advisor

The tool also seamlessly integrates the selected FGPA synthesis tools and create projects and can run the whole synthesis/implementation from within MATLAB. The HDL Workflow Advisor checks if the model is compatible for HDL code generation from HDL Coder and generate vendor independent HDL. Along with the HDL code, it can also generate HDL testbench and a co-simulation model. It is possible to run synthesis and implementation for a vendor specific FPGA the design could target. Two of the possible vendors that is being evaluated by this thesis and the tool can target are Xilinx Vivado and Intel Quartus. HDL Workflow Advisor generates the RTL for a system or subsystem then creates a project for the selected vendor. It is also possible to get the results of the synthesis and implementation runs in the Workflow advisor without launching the synthesis tool manually. Which makes it easier to use and compare. The project can be opened from within the Workflow Advisor and all the intricate details can be obtained. The settings for the parameters and the setup can be extracted or another setup can be imported in the Workflow adviser.

6.4 Optimization Options

HDL Coder offers several point and click optimisations that can be applied across the whole design, and it is very important that one has the understanding of what each change does before using it. HDL compatible blocks, subsystem or systems have additionally HDL block properties. These are instrumental in optimizing the design and exploring the design space.

A entire model properties can be adjusted using the HDL coder properties. Following are the list of HDL Coder optimizations on system level that are good to know to understand the tool better. The general optimization options are:

- Balance Delays
  When certain optimisations such as pipelining or resource sharing are enabled, or specify certain block implementations and code is generated, HDL Coder introduces pipeline delays along certain signal paths in your model. The code generator detects these pipeline delays introduced along one path and then inserts matching delays on other paths.
- Map pipeline delays to RAM
  Map pipeline registers in the generated HDL code to RAM. Certain speed or area

optimisations such as pipelining and resource sharing, or specify certain block implementations that can insert pipeline registers in the generated HDL code. Save area on the target device by mapping these pipeline registers to RAM.

- Transform non zero initial value delay
  This option optimizes delay blocks with non-zero initial condition. By using this transformation, HDL Coder can perform optimisations such as sharing, distributed pipelining, and clock-rate pipelining more effectively, and prevent an assertion from being triggered in the validation model.
- Multiplier partitioning threshold
  This parameter specifies the maximum input bit width for a multiplier. If at least one of the inputs to the multiplier has a bit width greater than the threshold value, the code generator splits the multiplier into smaller multipliers. To improve hardware mapping results, set the multiplier partitioning threshold to the input bit width of the DSP or multiplier hardware on your target device.

The pipeline optimization options are:

- Hierarchical distributed pipelining
  Hierarchical distributed pipelining extends the scope of distributed pipelining by distributing delays across subsystem hierarchies. This optimization moves the delays within a subsystem while preserving the hierarchy. It is used to reduce the timing, slack and latency of the design. This parameter within optimization decrease the slack by inserting pipeline registers in the critical paths within the subsystem.
- Hierarchical Distributed pipelining priority
  Numerical integrity:
  This option uses a conservative retiming algorithm that does not move registers across a component if the functional equivalence to the original design is unknown.
  Performance:
  This option uses a more aggressive re-timing algorithm that moves registers across a component even if the modified design's functional equivalence to the original design is unknown.
- Clock-rate pipelining
  If your design contains multicycle paths, use clock-rate pipelining to insert pipeline registers at a clock rate that is faster than the data rate. This optimization improves the clock frequency and reduces the area usage without introducing additional latency. Clock-rate pipelining does not affect existing design delays in your model.
- Adaptive pipelining
  This inserts pipeline registers to the blocks in your design, reduce the area usage, and maximize the achievable clock frequency on the target FPGA device. Adaptive pipelining makes the tool inserts pipelines between arithmetic operations, when rounding or saturation is used and after a down sample and transition block. Additionally, pipelines get added within LUTs, multipliers and rate change blocks. [7]

The last optimizing option is the the resource sharing options. The requirement for sharing factor between blocks is that [4]:

- Data types of their inputs and outputs are identical.
- Block parameter settings are identical.
- HDL block properties are identical.

The resource sharing options are:

- Share Adders
  Enable this parameter to share adders with the resource sharing optimization. Resource sharing identifies addition and summation blocks in your design and replaces them with a single block in the hardware. This optimization saves area on the target FPGA device.

Similar options are there for the multipliers, multiply blocks, atomic subsystems, gain blocks, MATLAB function blocks and floating point IP's. There are different requirements for different types of blocks which is described by Mathworks. The resource sharing options impacts the frequency of operation and thereby timing of the system, which has to be kept in consideration. The target frequency would be the frequency it is synthesized multiplied by the sharing factor.

The use of sharing factor is to reduce the amount of hardware components. For example when multiple multipliers are used then each multiplier can be mapped to DSPs on the device, depending on options for the design. This is relevant for devices with limited amount of DSPs on the board. With a sharing factor, a rise of registers used can be predicted, but most often FPGA devices have much more registers available than amount of DSPs. [6].

The adaptive pipelining and balance delays are enabled by default. The default design thus has optimizations enabled due to adaptive pipelining and delay balancing and the latency due the same have been considered. HDL Coder offers a variety of block level optimizations and varies between different blocks. The list is quite extensive and all cannot be covered here. The ones used in this thesis are as follows:

- Hierarchy flattening:
  Hierarchy flattening enables you to remove subsystem hierarchy from the HDL code generated from your design. The HDL Coder considers blocks within a flattened subsystem to be at the same level of hierarchy, and no longer grouped into separate subsystems. This consideration allows the coder to reorganize blocks for optimization across the original hierarchical boundaries, while preserving functionality.
- DSPStyle:
  DSPStyle enables you to generate code that includes synthesis attributes for multiplier mapping in your design. You can choose whether to map a particular block's multipliers to DSPs or logic in hardware.

- UseRAM:
  The UseRAM implementation parameter enables using RAM-based mapping for a block instead of mapping to a shift register.

The above stated system level and block level optimization options, where ever applicable have been used in this thesis. The optimizations for XSG takes place more on a block level and the designer needs to have a high level of understanding for tuning these parameters. With block level optimizations the configurations of for example a bigger FIR filter or a small adder, these configurations vary the synthesis results for the specific block. There are also a lot of options for the designer to choose from. Moreover, system level optimizations are not present and the designer has to take care of them manually. System level optimization effects many blocks within a system, for example pipelining, this will result in synthesis changes on a system level. Different configurations has different advantages. DSPB also have similar optimizations as XSG with the exception that a system or a subsystem can be scheduled for a certain number of clock cycles.

6.5 Logic Tests

The HDL Coder has two different workflows: Simulink HDL workflow and MATLAB HDL workflow. The Simulink HDL workflow can be equated to the HLS tools XSG and DSPB. The subsystems are implemented more or less like IP's and RTL are created for each of them separately unless specified otherwise. The blocks within the subsystem generate lines of code which is one of the reasons the code is traceable to the block level. The RTL code generated is dynamically generated in HDL Coder and is much cleaner and understandable compared to the other HLT. The various compiler optimization techniques have been presented below and the results for both workflows have been included. In these various techniques used to check efficient hardware generation [1].

6.5.1 Hardware generation transformation tests

Bit width analysis test:
HDL Coder does bit width analysis automatically if the designer wants. It is also possible to use the Fixed-Point Designer to determine the bit widths of all the signals in the design. This can be used to avoid complexities like overflows, bit growth effects,etc where a lot of design time is spent to ensure that these do not cause problems in the design. The designer can also choose to change the assigned bit width by forcing a specified bit width in the signal attributes of the block. On updating the model, the bit types are recalculated and updated.

Bit reversal test:
As for testing bit-level optimization a four bit reversal implementation was made using the HDL supported blocks. The inputs are sliced to extract each bit and concatenated to give a reversed four bit result. The code generated uses the same approach as the described in the model. The input is first sliced and then the outputs of the slices are then concatenated in RTL using the & operator. There is also a MATLAB function block example provided by MATLAB which reverses a 32 bit input. The code declares

constants and performs a series of bit operations for getting the reversed result which generates unnecessary complicated hardware. The optimal implementation would have been to use wires between the inputs and the outputs as in 1.

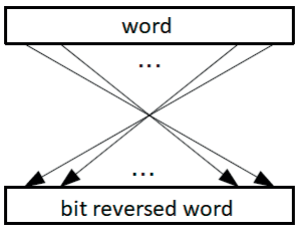

Figure 1: Optimized bit reversal [1]

Multiply/divide by 0,+1,-1 and Operator Strength Reduction(OSR) techniques:
There are two options to go by with this design. Either to use a product block or to use the gain block. The Simulink HDL Coder workflow as mentioned earlier generates lines of code and hence the product block is always interpreted as is and irrespective of any number you multiply with, the workflow will generate constants and multiply it with the inputs. There is an exception here that the product block is supported with this optimization in the the native floating point mode if the single data type is used but not for other types of inputs. The other option is to use the gain block which does provide the same functionality and generates more optimized HDL code. There is also an option of changing the block properties 'ConstMultiplierOptimization' to CSD or FCSD which replaces multiplier operations with additions of partial products produced by CSD or factored CSD implementation. The MATLAB function block has similar optimization options in its block properties and generates similar code based on the inputs assigned to it. The optimal way in implementing these techniques if it is multiplied by zero would be not to generate any hardware, a wire in case its multiplied by one and taking the two's complement of the number in the case it is multiplied by minus one.

Add with 0 test:
Both workflows produce a constant and add them to the input using a adder. This is a poor implementation but in order to provide traceablility this has to be done. It is also important to note that this operation should not be performed by the designer as it does not really make an sense to do it. The optimal implementation in this case would be a wire.

Constant folding test:
If there is an expression declared with constants, this technique computes the expression and directly uses the computed values wherever it is declared. Both workflows make use of constant unfolding and use the result of the constant equation in the generated RTL code. This technique works as expected and provides constants in the generated HDL code.

Common Sub-expression elimination test:
In this technique, the common sub-expressions are found out and assigned to a temporary

instance instead of computing the common sub-expression for each iteration. For eg, $a = b \times c + d$ followed by $a = b \times c + f$ does the computation $b \times c$ twice. This can be reduced by declaring a temporary variable t as $t = b \times c$ followed by $a = t \times d$ and $c = t + f$. The MATLAB HDL workflows do not support common sub-expression elimination. As for the Simulink HDL workflow, this can be done directly at the model level by connecting the result from $b \times c$ to the other inputs.

Code motion test:
Code motion is a technique which changes the order of execution of instruction in a program. It performs speculation and tries to shorten the critical path by adding registers. Since Simulink HDL workflow is a MBD approach, the hardware generated resembles the

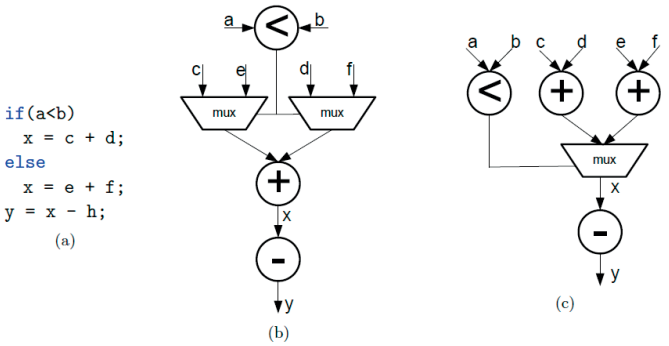

Figure 2: Code motion. [1]

model and for this optimization to take place the optimized design should be designed. As for the MATLAB HDL workflow, the compiler does not handle this optimization and produces hardware as (b) in the above figure, 2. The optimized hardware would be as shown in figure (c) which has a low critical path compared to (b).

Constant Propagation test:
This can be understood from the equations $x = 4$ and $y = x + 7$. This should be simplified by the compiler by assigning $y = 11$. In this way, the constant propagates as and when it occurs next. The Simulink HDL flow similar to the previous cases, the optimized model needs to be designed. Optionally, the workspace variables can be used for defining constants and used in the blocks. The MATLAB HDL flow handles this case and makes use of the propagated constant in the HDL generated.

Loop unrolling test:
Two different loop optimization supported in HDL Coder are: Loop streaming and loop unrolling. The Simulink HDL workflow, by default unrolls the hardware. For example, if given a vector input, HDL Coder unrolls the vector and repeats the hardware for each index in the vector. This can be changed if the streaming option is enabled and a streaming factor is mentioned.

The MATLAB HDL workflow the unroll option does not create a control algorithm as expected but instead produces the same hardware as in the case of the rolled loop. The

only difference is in terms of the HDL code produced, the unroll option when turned on, has individual assign statements for each output whereas, in the other case, for generate is used to produce the HDL code.

6.5.2 Analysis of the above tests

Both the workflows have limited support for efficient hardware generation and can not be really classified as an HLS tool which offers highly efficient RTL. Traceability in HDL Coder comes at a cost and in most cases end up producing inefficient HDL code and relies on the designer to create an optimized design.

There is lack of integration of some of the feature tests conducted above in HDL Coder, as the RTL generated from these tools is often below par in performance in comparison with a hand written, fine tunable RTL code. The results from HDL Coder depends on the design and the experience of the designer with the tool. The design should be broken down into small blocks and these blocks should be optimized for the target hardware and then integrated together to get the optimal results. The difference in performance can be reduced by expressing the design in a more detailed manner but this lowers the level of abstraction considerably. The results from this section should be considered while designing the designs presented until the tools get updated and incorporates these features in them.

6.6 Reports

To check the resulting model and to investigate the result of the optimization methods it is a good idea to go through the generated reports and check the generated model. The reports that get generated by HDL Coder are traceabilty, high level resource utilization, high level timing report, optimization report and web view model. The optimization options in the generated report shows the optimizations that were successful and the generated model shows all the added pipelines and the matched delays. The generated model can be found in the project folder created.

6.7 Resource Mapping

This is one of the most important aspects to be considered while designing the system especially for this thesis as this is a major point of concern. The mapped hardware should be efficient and the resource used during the design should map effectively to the FPGA fabric. The block level optimizations UseRAM and DSPStyle have been used wherever applicable to get the most from the DSP resources of the FPGA.

1. RAM mapping
   One of the most common sources of area inefficiency when first targeting hardware is large register banks. Often these can be more efficiently implemented on the FPGA device by targeting its block RAM resources.

2. DSP mapping
   In addition to multipliers, DSP blocks in modern FPGA devices provide many other dedicated resources, such as adders and pipeline registers, for high-performance signal processing computations. The following guidelines will help you leverage those resources for designs containing common DSP operations. [7]

   - **Reset:**
     Use the global reset type, or set the HDL property *ResetType* to none for multiplier or adder pipeline registers. Using an incorrect reset type prevents DSP block resources from being fully utilized.
   - **Multiplier word size:**
     18 x 18-bit multiplication is a good starting point, as it usually maps well into FPGA DSP blocks. However, the architecture of the target device needs to be checked to maximize the built-in multiplier resources. When a multiplication exceeds the built-in word size, the synthesis tool either implements the extra bits in FPGA fabric, or splits the operation into multiple DSP blocks, resulting in lower speed and higher area.
   - **Fixed point settings:**
     Use full-precision fixed-point for any multiply/add operations, add output pipeline registers, and then perform the necessary rounding/saturation after the final output register with a *Data Type Conversion block*. Alternatively, use rounding/saturation on a product/add block, and let HDL Coder automatically insert pipeline registers between the multiplier and output logic via *Adaptive Pipelining*.
   - **Pipeline registers:**
     Using all available pipeline registers in the DSP blocks will give you highest clock frequency at the expense of increased latency. On the other hand, exceeding available pipeline registers may prevent synthesis tools from mapping all math operations within the DSP, resulting in sub-optimal area and timing performance. Choose pipelining level that matches the DSP architecture of your target device and your hardware requirements; adjust them as needed based on synthesis results.

6.8 Chapter Summary

In this chapter, literature on the HDL Coder is collected and presented in a way a person with a basic knowledge of hardware can use Simulink with HDL Coder in a smooth and hassle free way.

7 Experimental Design, Power Meter

The purpose of this experimental design was to introduce ourselves with HDL Coder. Firstly, Simulink to HDL code workflow was used to learn how to combine HDL Coder blocks to get the required functionality. Once a bit accurate and cycle accurate functionality was achieved, synthesis of the design was done and the results were compared in terms of size, timing and RTL readability. Later the optimization options that can be implemented on the design were evaluated.

To do the comparison, both the HDL Coder designs were synthesized for the selected devices using Xilinx Vivado and Intel Quartus. Once the functionality is verified both flows, the designs should be optimized for timing and area. The new values obtained can be compared with the same designs in XSG and DSPB. Using HDL Coder, one single model can be used to target both vendors, then synthesis can be done independently in Vivado or Quartus. The experimental design test are all run with a frequency of 122.88 MHz which is the frequency of operation. For achieving the best resource mapping with the tool the parameter for reset type was put to *Synchronous* for the HDL Coder designs that target Xilinx vendor and *Asynchronous* for the Intel vendor designs.

7.1 Design and Implementation

Power meter is used for measuring the power of the I-Q data in a design. When the transmitter or receiver is being used, the power meter makes it easier to investigate the power used in the circuit. It presents the real power magnitude from signals. The magnitude can be calculated with the equation below.

$$M = (a^2 + b^2)$$

This power meter has two input signals apart from the clock, enable and reset. The signals are the I and Q data from the TX or RX and are the signals to be squared, added and accumulated, The reset, clock and enable signals are self explanatory. The output of the design is the result signal. In the magnitude equation, the a and b are the input signals. The square is made with two multipliers, one for each signal. The input to both the multipliers is the same signal which results in the square. The output from the multipliers gets added. The result gets accumulated with the previous result. The whole system is an enabled system. The output gets cleared when the reset goes high.

This functionality was used as an inspiration for the first HDL Coder design to understand the workflow and was compared to its XSG and DSPB workflow tools. Simulations were done to confirm that the design gives the correct outputs. One of the major differences between a design done in XSG/DSPB and a design in HDL Coder is that the enable and reset is done automatically, as mentioned in part 6. Therefore the enable and reset signals used in this power meter is not included in the block design but will end up in the RTL code generated. Therefore, the amount of pins into the Simulink block only has to be the two inputs that is to be multiplied, as seen in figure 3.

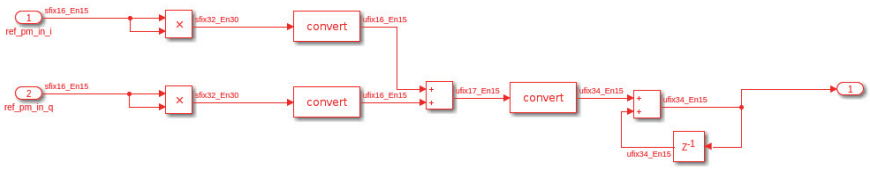

Figure 3: Power meter block design.

A HDL Coder design was made using the MATLAB HDL workflow to evaluate how does it compare to the Simulink HDL workflow. A MATLAB code as a function was written and the results were compared. The design of a power meter with only a MATLAB Function block within the Simulink environment can be seen in figure 4, and the MATLAB code can be viewed in appendix 12.1. The function arguments and the return type signifies the inputs and the outputs to the design respectively. Since the design is to be done in fixed point, the *fi* method is used to define the data types used in the design. The registers in the design can modelled using the *persistent* keyword. There are other methods to model registers as well and they are *coder.hdl.pipeline()* and *dsp.Delay(expr,no_of_registers)*. The code is written so that the data flow can be visualized and only the final output is given to a register.

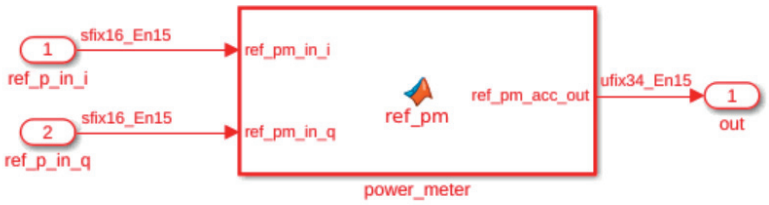

Figure 4: Power meter MATLAB function.

To investigate the design options further, a model of the power meter was made with both HDL Coder blocks and MATLAB function blocks within the Simulink environment. MATLAB code for the multipliers and adders were written and were replaced with these MATLAB function blocks in the design, see figure 5. This was done to investigate how both designs work together. The code for the multipliers and adders are presented in appendix 14. The rest of the blocks and settings for this design is the same as the first design, the block design.

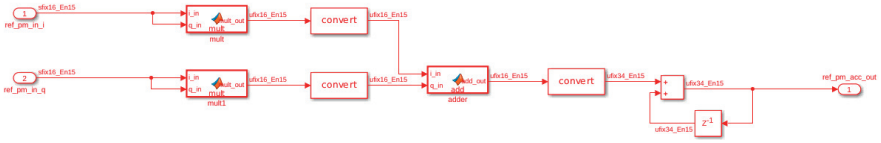

Figure 5: Power meter MATLAB functions and block design.

7.2 Synthesis

To synthesize the HDL Coder designs, the HDL Workflow Advisor was used. Also, to check if the design is compatible for HDL Coder the commands *checkhdl('path')* and *makehdl('path')* can be used on MATLAB command window. The Workflow Advisor, as mentioned in section 6.3, can produce RTL, synthesis and implementation reports according to the synthesis tools specified in Workflow Advisor. With this tool, it becomes possible to receive various reports from both HDL Coder tool and the ones produced from the selected vendor. Therefore, to compare the results from the power meter in Xilinx the same synthesis tool Vivado was applied with the same device.

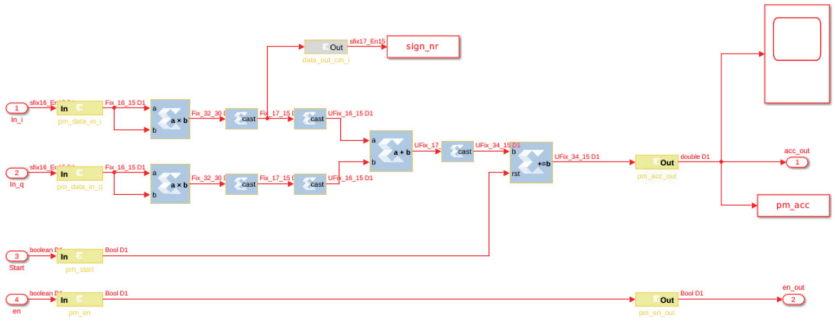

Figure 6: Power meter design in XSG.

Moreover, the synthesis is run with the corresponding Xilinx and Intel power meter designs. For the Xilinx design made using XSG the HDL netlist is extracted which also creates a project in Vivado. The complete design can be viewed in figure 6. The reports are generated to view the timing and the resource utilization for the selected FPGA. The results from the power meter in Xilinx measured with a frequency of 122.88 MHz is shown in table 1.

The RTL produced from XSG is exported as a HDL netlist and the HDL code is barely readable or understandable. The different blocks used in the design generates complicated IP's and the HDL for the power meter shows just the port mapping to these IP's. Moreover, the code can not be observed to be a general power meter code.

The Intel design of the power meter can be viewed in figure 7. In comparison to the HDL

Resource Summary (#)	XSG design
CLB LUTs	41
CLB Registers	54
DSPs	2
Carry8	8
Block RAM Tile	0
Timing Summary (ns)	
Requirement	8.138
Data Path Delay	4.053
Slack	4.085

Table 1: Power meter in XSG

Coder model there is not that much extra blocks needed. The prior effect is the two extra inputs and input and output blocks for the system. In DSPB, it is possible to put an extra function block to get a scheduled architecture, that is for balancing and insert delays where it is most optimized. For the Intel design made using DSPB the project, no reports need to be generated, since DSP Builder within Simulink can keep track of utilization and timing as and when the block are added to the design. The results for the power meter in Intel run with the same frequency as the Xilinx design can be viewed in table 2.

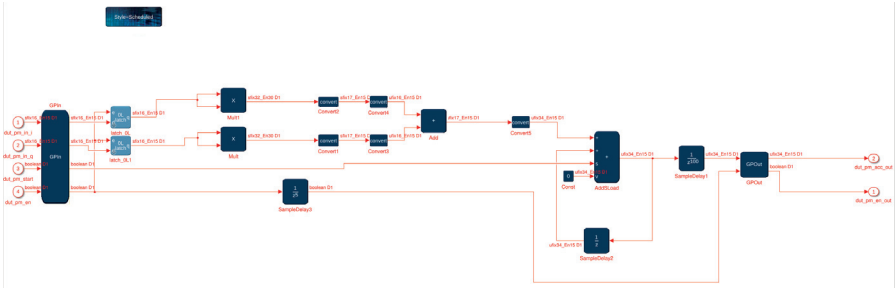

Figure 7: Power meter design in DSPB.

Resource Summary(#)	DSPB design
Combinational ALUTs	66
Dedicated logic regs	167
Block memory bits	0
DSP blocks	2
Timing Summary (ns)	
Period	8.138
Worst case slack:	
Setup slack	5.702
Hold slack	0.038

Table 2: Power meter in DSP Builder.

The RTL code produced from the design made in DSPB is more readable than the one produced in XSG. It is more than connected blocks and includes traceable functions and comments to know what operation in the design corresponds to the RTL code. However, it is not as readable as the HDL Coder produced RTL. The timing for the Intel device is presented in setup and hold slack, which is defined by:
Setup Slack= Data Required Time - Data Arrival Time
Hold Slack= Data Arrival Time - Data Required Time
Zero setup slack indicated that the design is exactly working at the specified frequency and no margin available. Negative setup slack implies that the design can not manage the constrained frequency and timing, also called setup violation. A similar hold violation exists for negative hold slack.

7.3 Optimization

For small designs, there are not much optimization options available. However, all the optimization options offered by HDL Coder were explored. First, each of the three HDL Coder design was made, verified and optimized individually, to make sure the best case for all the parameters for the individual design. The optimization options that is appropriate for a design this size is to use pipelines, sharing factor and the flattened hierarchy option provided by HDL Coder. How this is done can be read about in section 6.4.

The use of sharing factor is suppose to reduce the amount of multipliers used and therefore the amount of DSP blocks used. With use of sharing factor a rise of registers used can be predicted, but most often devices have much more registers available than amount of DSPs. An experiment with a sharing factor of two was done for the block design and the MATLAB function and block design, to investigate if the area would get reduced. Though, it has to be kept in mind that the double frequency is needed for these designs.

The optimization option for distributed pipelining was tried on all three of the designs. Though, it was noticed that the timing was not that effected compared to the extra registers added for the designs. The option for using a flattened hierarchy was tried as well, to distribute the registers across the subsystem to optimize the timing.

7.4 Results Xilinx

A power meter design was a reasonable design to start because of the logic and size. The block design was easy to make and understand. Most of the results from it are relevant for this thesis work. The resulting block designs looks very similar, the one implemented in HDL Coder, XSG and DSPB. The logic can be understood from all three designs. Though, the design part with the reset and timing is understandable in RTL produced by HDL Coder but less so in XSG and DSPB. The design with a MATLAB code function block is not as clear as the one where only blocks are used. It took more time to produce the MATLAB code compared to designing with HDL Coder blocks. The chart in figure 8 show the synthesis results before optimization for the HDL Coder designs and after optimization, and the results are discussed in the sections below. All the designs met

the timing constraints. The table 3 show the slack for each test. Considering timing the block design with distributed pipelining, the MATLAB function design with pipelining and MATLAB function and block design works best. The worst is the unoptimized MATLAB function design. As for a design with area constraints the designs to choose from could be block design and MATLAB function design because using only 5 DSPs and low amount of resources otherwise.

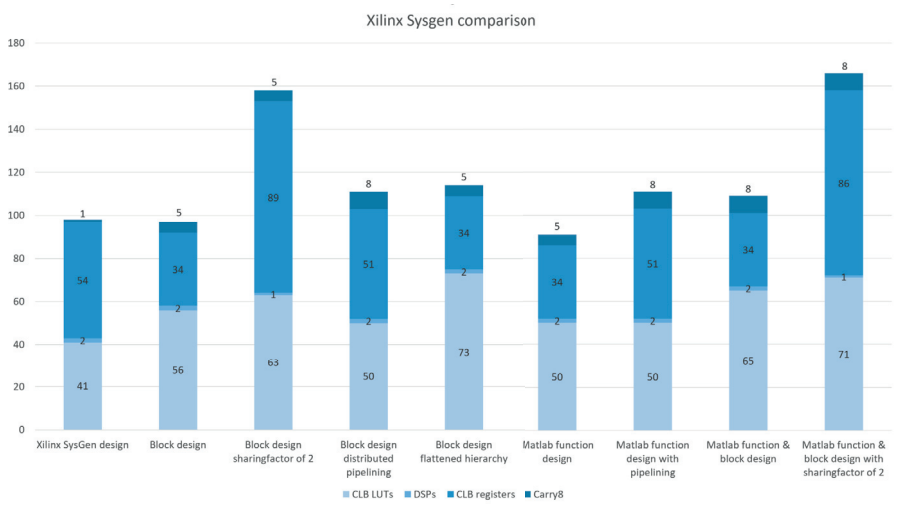

Figure 8: Xilinx comparison with HDL Coder models and optimizations.

The block design corresponds to the design consisting of Simulink HDL synthesizable blocks whereas in the MATLAB function based design it consists of a MATLAB function block within the Simulink model where the code written generates the synthesizable HDL. A mixture of the block design with MATLAB function blocks have also been tried out and is named MATLAB function and block design. In the below table, the results for the different optimization tried with the block design and MATLAB function block have been summarized.

	Slack (ns)
XSG	4.085
Block design	5.193
Block design sharing factor of 2	6.724
Block design distributed pipelining	6.989
Block design flattened hierarchy	5.225
MATLAB function design	4.723
MATLAB function design with pipelining	6.989
MATLAB function and block design	7.05
MATLAB function and block design with sharing factor of 2	5.234

Table 3: Xilinx slack comparison with HDL Coder models and optimizations.

7.4.1 HDL Coder Block Design

To optimize the HDL Coder block design was not that complicated, and the different options were easy to use. When optimized for timing an improved slack value was received even better than for the XSG model. With distributed pipelining and the area increase somewhat with added registers, see figure 8. Also there was nothing that could be changed in the design to change the resource utilization, for example map more to registers than LUTs. The design included two multipliers that with a sharing factor of two could make the design use only one multiplier. The synthesis results of this case showed that the number of DSPs needed for the design was reduced to one but the amount of registers and LUTs had to increase. However this impacts the clock frequency and the design needs to be run at twice the frequency to produce the same results as in the simulations. The table also shows the effect a flatten hierarchy has on the synthesis results. The flattened hierarchy adds extra logic units to handles clock-rate regions and optimizations across subsystems. As a result, the slack is improved and the amount of registers increased.

The RTL produced from the block design is easy to understand. The code is separated depending on the subsystem and the logic from the blocks are easy to understand and trace in the code.

7.4.2 HDL Coder MATLAB Function Block Design

The pipeline registers were added to the MATLAB function design to optimize it further and making it resemble to the XSG power meter as close as possible. These are added after the inputs are squared, added together and at the output. The MATLAB code can be viewed in appendix 13.1. On adding the pipelines in the MATLAB code for the function blocks the following results were obtained. As expected the number of registers goes up with a great improvement in timing. Observe that the timing and resources used for the pipelined block design and the pipelined MATLAB function block are the same.

7.4.3 HDL Coder MATLAB Function and Block Design

The third design with both HDL Coder blocks and MATLAB function blocks was also optimized regarding timing and area. The initial slack of the system is rather high and therefore a pipelined experiment was not needed. Although, the initial resource values are quite large. A test with sharing factor was done. It got an increase in area compared to the initial synthesis values, for the results see figure 8.

The RTL code produced is also very easy to understand and follow. In difference to the block design two other RTL entities were created, one for the multipliers and one for the adders. Also an overlooking top structured was produced very similar to the one for the block design.

7.5 Results Intel

The design made in HDL Coder match both the Intel and the Xilinx design. The chart in figure 9 show the synthesis results before and after optimization for the HDL Coder designs targeting the Intel devices with the Quartus synthesis tool. The table 4 shows the timing for each test, which shows that all the designs met the timing constraints. In this case the MATLAB function and block design and MATLAB function design with pipelining shows the best timing results and should be used if the design has strict timing constraints. As for area it is clear that the MATLAB function design is the best.

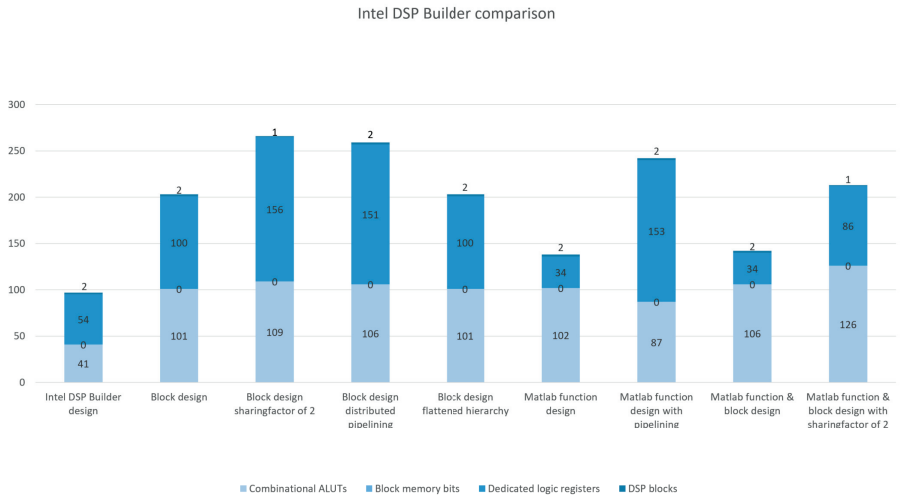

Figure 9: Intel comparison with HDL Coder models and optimizations.

	Setup slack (ns)	Hold slack (ns)
DSPB	5.702	0.038
Block design	4.889	-0.248
Block design sharingfactor of 2	4.889	-0.455
Block design distributed pipelining	4.889	-0.455
Block design flattened hierarchy	4.889	-0.248
MATLAB function design	6.084	-0.004
MATLAB function design with pipelining	6.095	-0.294
MATLAB function and block design	6.29	-0.247
MATLAB function and block design with sharinfactor of 2	4.331	-0.455

Table 4: Intel slack comparison with HDL Coder models and optimizations.

7.5.1 HDL Coder Block Design

The figure 9 shows the results of the same experiments made in section 7.4.1 but in this case targeting an Intel FPGA synthesized using Quartus. It is observed that neither of the optimization methods did improve the amount of registers or LUTs needed. The test with sharing factor did as expected reduce the number of DSPs from two to one, but with an increase in registers needed. Though, the timing is not affected with the use of pipelines only an increase of registers needed same for flattened hierarchy.

7.5.2 HDL Coder MATLAB Function Block Design

The result for the Intel device using the MATLAB function block is rather well compared to the block design. As observed both MATLAB function block designs use less registers, which could be because adaptive pipelining do not interact with MATLAB function blocks. Since the MATLAB code for this design is the same, the pipeline registers as in the case of Xilinx, are added after the inputs are squared, added together and at the output. The result of distributed pipelining is similar to the same done for the block designs.

7.5.3 HDL Coder MATLAB Function and Block Design

As mentioned before, it was noticed for this case as well that the design are to small to improve using optimization. In figure 9 the results for the third design using a sharing factor of two is presented. The results for slack is initially good but a sharing factor of 2 effect both the timing and area a lot.

7.6 Chapter Summary

It was concluded that the design was too small to make a valuable impact with the optimization methods provided by HDL Coder. When an optimization with pipelines were made for both vendors the impact on timing was good compared to the increase in area. Interesting was that the same result was not observed for the Quartus vendor design, compare chart 8 and 9. The increase of registers and LUTs seems to be the same between the vendors when it comes to use of sharing factor. For the timing as well the effect seems to be almost the same, though it is difficult to conclude because the design had positive slack from the beginning. The numbers between the vendors are not comparable because of the different size of LUTs and registers that is used for the different vendors, see background section 5.4.

8 Design of Down Sampling Filter Chain

In this chapter an experiment with a down sampled filter chain is presented. First, a small description of the implemented design is presented. Further, the first synthesis results and optimization options used are evaluated. Lastly, the results are presented, discussed and conclusions are made.

8.1 Design and Implementation

The filter chain created consists of logic blocks and filter chains which interpolates/decimates the signal in multiple steps. The purpose of the design is for communication therefore both a TX and RX part is needed. For the TX part the steps are interpolator, VGA and a FT, as seen in 10. The interpolator constructs new discrete data points and upsamples the sample rate. The VGA lowers the power in the signal and the FT place the samples within a specific range decided by the NCO. The RX consists of the same VGA and FT but instead of the interpolator it has an decimator, that re-samples and downsamples the signal. The design of the TX and RX part are based on the logic presented in figure 10, observe the sample times for each system. The design of the filter chain can be found and described in appendix.[1] A reference model in Simulink was used to model the design using the HDL Coder supported blocks. The reference model was not designed with the hardware in mind and hence there were a lot changes that was needed to be able to model it in a hardware friendly way. The reference model was made in double precision then converted into fixed point for FPGA implementation to be more hardware efficient. The first problem faced was the lack of the complex phase shift HDL Coder supported block and this was modelled using a gain block with the multiplication factor as exponent of the phase gain. This results in a small precision loss due to the quantization error. Additionally, the RX design was made with 6 filters instead of 3 that is made in the corresponding designs in XSG and DSPB.

The earlier power meter design was a single rate design whereas this design had operating regions at three different rates. The operating frequency of the design was still at 122.88 MHz as used previously but the inputs are decimated and interpolated using different filters to handle rates of 30.72 MHz and 61.44 MHz. This is implemented in HDL Coder using a timing controller which samples the different subsystems at different rates as mentioned above. There is also an option of enabled multiple clock in HDL Coder which makes use of different clock inputs to handle the different rates.

[1] Information of the specific logic are not to be described because the design is used in an Ericsson product and can therefor not be disclosed.

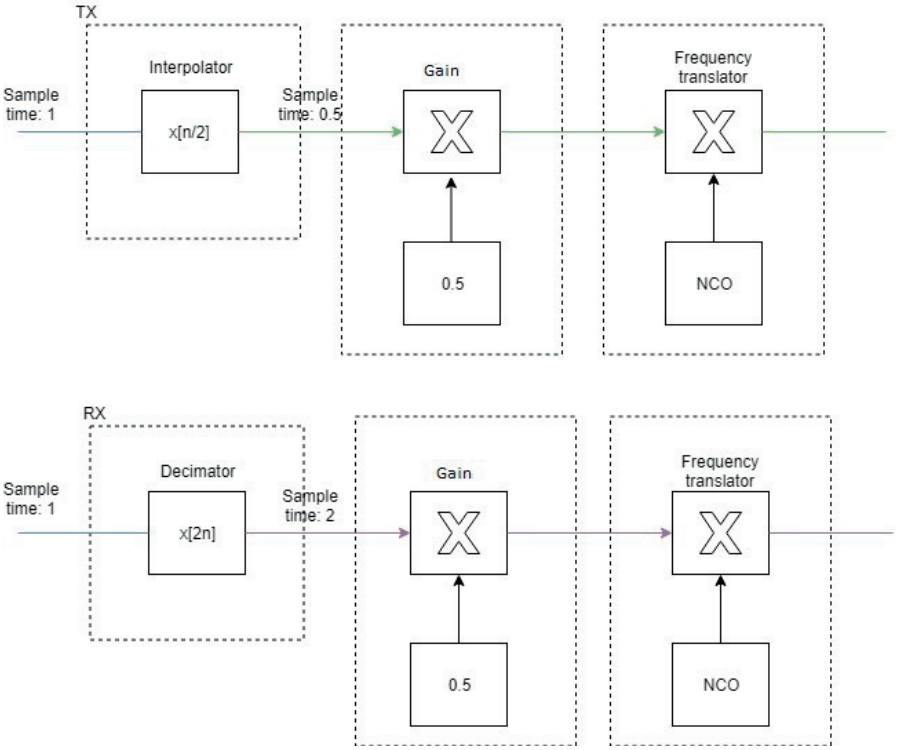

Figure 10: One TX and one RX design.

8.2 Synthesis

The synthesis was done with the default blocks settings and default optimizations enabled in HDL Coder. This design does not meet the timing requirements set for it and provided negative slack for the design. The slack for the two initial run where -9.048 ns targeting Xilinx device and -23.948 ns for the Intel device. The small experiment with the power meter, in the previous section, timing was not a problem though with this relatively bigger design the first synthesis that was run had timing problems. It is noticed that the design needs to be optimized by architecturally changing the design, optimize the blocks used and general optimization methods provided by HDL Coder to be comparable to the Xilinx and Intel vendors. In the next section the optimization methods used to fix timing and make the design synthesizable is described in detail.

8.3 Optimization

The conclusion from the first synthesis was that with a design this size some time has to be spent on optimization. Exploring the various block optimization properties, it was

narrowed down that by enabling the input pipeline registers for the interpolators and decimators the timing requirements are met. This adds to latency as expected and is reported by the tool. All the HDL supported Simulink blocks have HDL block properties and have a handful of options that can be selected based on the design requirement. The design was also optimized for HDL Coder. For example there were a number of product blocks in the reference model which were multiplied by constants and these were replaced by gain blocks to get a more optimized RTL as mentioned in section 6.5. For the architectural optimizations it is important to simulate and make sure that the functionality does not change. Apart from the above mentioned architectural optimizations, the general optimization options for HDL Coder were tried out.

System Level Optimizations:
The possible optimizations on system level are fully investigated in this experiment. For all the designs except the ones without the optimization adaptive pipeline is used. Adaptive pipeline adds pipelines based on the targeted synthesis tool and target frequency characterized by MATLAB. Because of the filters in the design the system has multiple rates, the optimization option clock rate pipelining is used on one of the designs. To be able to use clock rate pipelining an oversampling factor has to be set to more than one depending on the sample rates and the frequency. For our design the oversampling factor was set to 4. Clock rate pipelining adds registers to slower paths of the design and it works together with adaptive pipelining therefore both options are enabled. To optimize the timing, hierarchical distributed pipelining is used and set to *Numerical integrity* for one test. For the design presented two input and output registers were added to move the delays within the design on both the TX and RX part. Though, multiple options with the pipelining on the subsystems was tested and only the best result is presented. Also a flattened hierarchy can reduce the slack but for this design if all the subsystem boundaries were removed, it was impossible to understand the generated model and the RTL produced therefore the results are not included in this thesis.

HDL Coder also supports area optimizations with the use of streaming and resource sharing. The inputs to the system needs to be a vector for the streaming to take place which was not the case in our design. Besides, since the clock rate and data rate had to be the same these optimizations could not be performed on this design.

Block level optimizations:
Moreover, the design could be even more optimized on block level. By changing the block parameters that specific block's characteristics or mapping can be changed which could be vital to fit on to a device. To investigate this flexibility DSPStyle and RAM mapping can be used. Using DSPStyle mapping increase amount of DSPs used in the system which result in less registers and LUTs needed. DSPStyle properties were put on the smaller subsystems and the operators where it was possible. As for RAM mapping, the property was enabled for all the registers in the design and the minimum width of the RAM was set to 1 to push the tool to map the other generated registers to RAM. The more use of RAM mapping result in less other resources needed for the design.

8.4 Optimization Results Xilinx

In figure 11 the most explanatory results of this experiment is presented. The numbers are in percent compared to the same category for the XSG design. It is visible that the amount of registers and LUTs used are somewhat more, though that could be explained by the less amount of RAM used in the HDL Coder design. In the case with *RAM mapping* a try with as much logic as possible mapped to RAM which shows a decrease in both LUTs and registers used. The reason for the high amount of DSPs in the HDL Coder designs could be the partial use of DSP to perform computations which is done more efficiently in the case of XSG which makes use of the entire DSP slice. In the figure the Carry8 category is excluded. That is because the values where not in proportion, the result for all the designs is around 3100 % compared to XSG 100 % for all the designs. This could be owed to the high number of DSPs and the LUTs for the routing of the fast carry logic. Moreover, the increased resource values are also effected by the design miss with the amount of filters used.

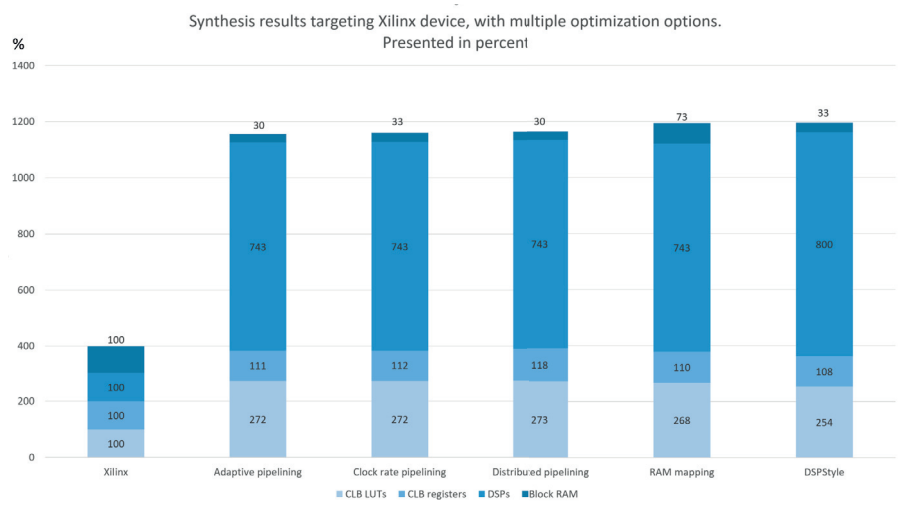

Figure 11: Xilinx comparison with HDL Coder models with different optimization options.

	Slack (ns)
Xilinx	5.053
Adaptive pipelining	3.055
Clock rate pipeling	3.267
Distributed pipelining	3.273
RAM mapping	3.267
DSPStyle	3.267

Table 5: Slack results from XSG design and initial HDL Coder design aimed for Xilinx device.

Overall it is clear that the optimization methods is not improving the design that much. Clock rate pipelining improved timing somewhat by inserting delays that map to regis-

ters. In table5 it could be observed that distributed pipelining improved timing somewhat. For the increase of amount of registers the increase in timing was reasonable but it could be pushed more by adding more registers. In this design, block level optimization options and flexibility was tested. The XSG design has more logic mapped to RAM and for HDL Coder that amount was not possible to map. On block level only the inserted delays where possible to map to RAM manually in addition to the system level RAM mapping option. The case *RAM mapping* show the design with the most possible logic mapped to RAM which became a fair amount. For that case the registers and LUTs decreased. As for DSPStyle, it could be concluded how the resource utilization can be changed. If less LUTs or registers are needed for a design then DSPStyle is a good compliment with RAM mapping. Though, normally it is the amount of DSPs that is critical. Overall the impact of the optimization methods where quite small on the standard filter block. If the optimized filter blocks or an own lower level design would have been done then the optimization results might would have been improved, but this is out of the scope of this Master Thesis.

8.5 Optimization Results Intel

The resource utilization compared to the DSPB design can be view in figure 12. First, it is visible that the categories presented is double the DSPB designs resources except for amount of DSPs which is more than four times the value. Second, the category block memory bits could not be included in the chart cause it was out of proportion compared to the rest. The final values was 1513 block memory bits in the DSPB design and around 114700 bits for the HDL Coder design. Though it is important to know that the value is in bits and a small block mapped makes a big difference, for example the 1513 bits where generated from only 4 delays. With the traceability of the Quartus it was possible to trace the high amount of block RAM. The big NCO block HDL Coder used resulted in approximately 49 000 bits and in the design there where 2. The remaining 15 000 originated from the delays used in the design and the delays added by the optimization methods and balance delays. The DSPB design was highly optimized and used a custom NCO which resulted less block memory bits, therefore the results for the HDL Coder design seems incomparable. Comparing the timing for the designs it is visible that the DSPB design has both values positive whereas HDL Coder designs gives a negative hold slack. As explained in chapter 7 the hold slack can be negative it is the setup slack that implies the setup violation.

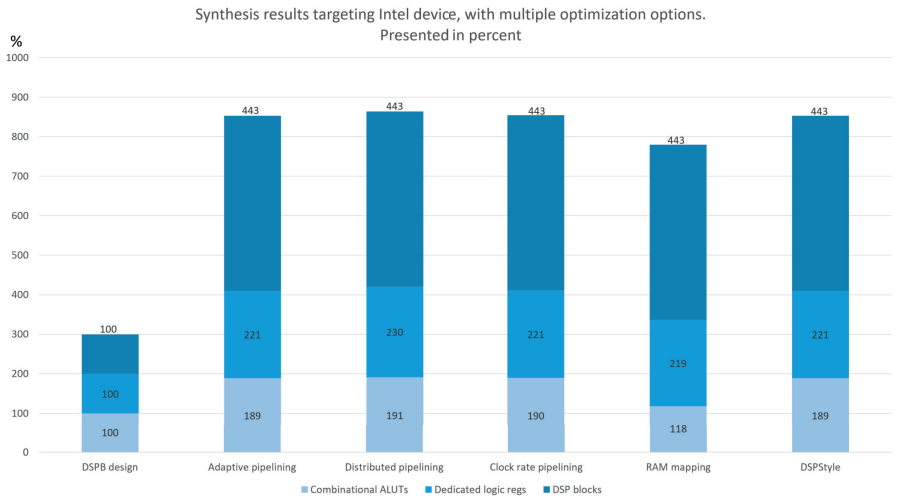

Figure 12: Intel comparison with HDL Coder models with different optimization options.

Slack (ns)	Setup	Hold
Intel	2.892	0.035
Adaptive pipelining	4.209	-0.519
Clock rate pipeling	4.226	-0.519
Distributed pipelining	4.149	-0.519
RAM mapping	4.145	-0.519
DSPStyle	4.209	-0.519

Table 6: Slack results from DSPB design and initial HDL Coder design aimed for Intel device.

The overall view of the optimization options is that they have a very small influence in actual resources. It is viewed that distributed pipelining have an negative impact on the timing. Though, clock rate pipelining give a better result. Clock rate pipelining should make the registers more effective in the regions with slower clock rate, but it seems mostly that registers is added. As for the block level optimizations, RAM mapping is very effective and DSPStyle have barely have any impact.

8.6 Chapter Summary

The results of this experiment can conclude that, when using the standard filter block, HDL Coder does not produce results comparable to XSG and DSPB for optimized filter designs. Though, have in mind that these tools are highly optimized for their devices as for HDL Coder only can be generally optimized. It is concluded that the results for targeting the Xilinx FPGA is more optimized than the Intel one. Also remember that more decimators is needed for the HDL Coder design, which has an impact on the result. Compared with the optimization impact for the power meter this experiment had a bigger change for the utilization and timing, though, less than expected. The design

might have to be even bigger. It was observed that the design did not have any problems fitting on to the FPGAs. This design was a block of the entire system implementation on the FPGA. To challenge the device size and investigate more in the optimization options a branched design was made where each branch included one filter chain from the design in this experiment, this can be viewed in 9.

9 Branched Filter

In this section a bigger design is presented, experimented on and optimized, a branched filter. The implementation is described. For this experiment only a comparison with the Xilinx correspondence is used. Then the synthesis and first synthesis result is presented. Lastly the optimization options used and the results are presented and discussed.

9.1 Design and Implementation

The design that was made is a three branched filter chain. Which means that the figures 13 and 14 are made times 3 for each part. In each of the *Block*s the RX and TX filter from section 8 is placed, with some adjustments making them a one stage filter instead of three. This result is 7 blocks for each branched system. For the TX subsystem the sample rate on the input start at 8 then it gets interpolated down to 1. Whereas the RX start at 1 and decimates through 3 stages of decimators to the sample rate of 8. Additionally, the TX branch has to include a CC block to merge the signals correctly, which is only a complex adder. The same branched filter chain was made in both HDL Coder and XSG.

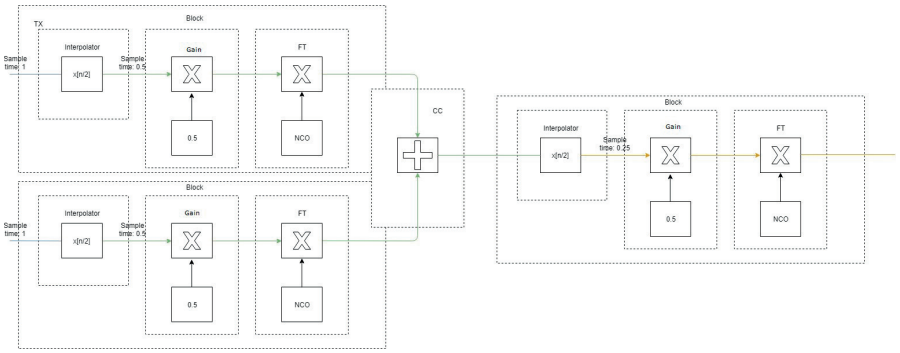

Figure 13: TX branch.

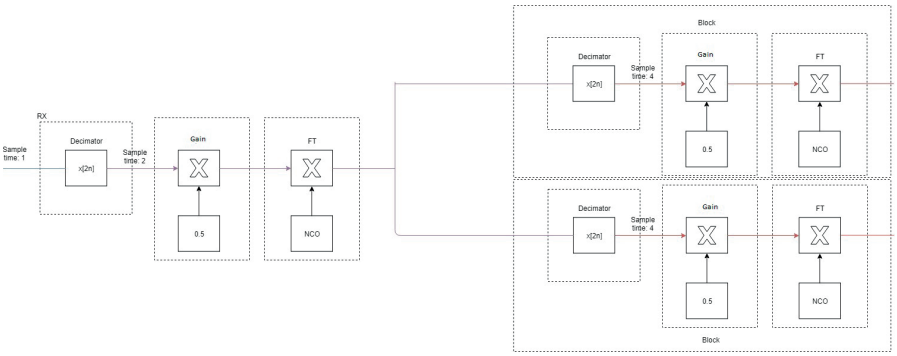

Figure 14: RX branch.

9.2 Synthesis

The synthesis flow was run in the same way as in section 7.2 but with only the Xilinx part. It was expected that the first synthesis of the HDL Coder design would have negative slack, because of the negative slack of the initial design in the small filter chain in the previous chapter. The slack now was -6.019 compared to the XSG design with 6.84. Therefore optimization is needed for this design, which is described in the next section.

9.3 Optimization

Primarily the timing had to be solved thereby the input and output pipeline registers were enabled for the decimators and interpolators in all the blocks. It is noticed that block level optimization options takes more time to change and evaluate for the bigger design. Scripts can be used to change multiple parameters in short time. Because of that the smaller blocks in the branch is originated from the filter chain design there was no design optimizations to be done. Moreover only system level and block level optimizations are tried out, which are described in section 8.3. The options that is used is adaptive pipelining, distributed pipelining on system and block level and RAM mapping on system and block level. Clock rate pipelining was investigated if it could be used, but because there are multiple sample rates then multiple clocks had to be used for which the oversampling factor can not be set to more than 1.

9.4 Optimization Results

The conclusion that is taken from table 7 is that the slack for XSG design is not achievable by the HDL Coder design. As for the utilization the values for everything except DSPs and Carry8 is not that bad. The explanation for the high amount of DSPs is that the interpolator and decimator blocks used in HDL Coder is unoptimized compared to the ones used in XSG design. The utilization for the blocks in HDL Coder produce 32 DSPs

each versus 6 the RX and 2 for the TX in the Xilinx design. There is one in each block that result in that the difference increase a lot fast. As for the amount of Carry8s the interpolator and decimators used in HDL Coder design are introducing 5 DSPs where the same block in XSG does not use any. Additionally, the NCO blocks introduces more DSPs depending on the sample rate for the HDL Coder design. One effecting factor is also that the XSG branched design is made by us and the logic between the blocks might not be optimized.

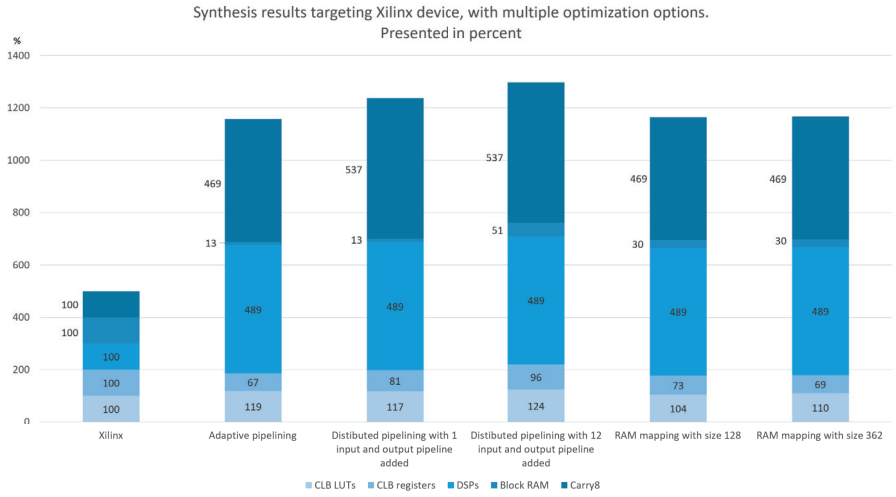

Figure 15: Xilinx comparison with HDL Coder branched design with different optimization options.

	Slack (ns)
XSG	6.84
Adaptive pipelining	4.967
Distributed pipelining with 1 input and output pipeline	4.97
Distributed pipelining with 12 input and output pipeline	4.97
RAM mapping with size 128	4.97
RAM mapping with size 362	4.97

Table 7: Slack results from XSG design compared with HDL Coder with different optimization options.

Comparing the optimization results for the filter chain in section 8 versus the ones for this experiment now some effect can be guaranteed. The adaptive pipelining and with default options produce good results by itself. Slack got increased with distributed pipelining and one extra input and output pipeline per filter block. Though, when amount of pipelines increased the timing did not get effected. It was enough to activate the *Hierarchical Distributed Pipelining* to get maximum slack. RAM mapping increased somewhat when it was enabled for every block possible, but not as much as what is possible for the XSG design. When varying the *RAM mapping threshold* for the constraint *Map pipeline delays*

to RAM it was found that it is possible to change the register amount and LUT amount. Presented in the chart are only two of the experiments. When the threshold was varied in the optimization for the filter chain in the previous section the result barely changed.

9.5 Chapter Summary

It is visible that the results are improved compared to the smaller designs and that the optimization options take effect. It was noticed that the big differences for one of the 7 filters got increased a lot, but all the values are explainable. Limitations for how much logic that could be mapped onto RAM existed, which makes it difficult to compare as well because it is not 1 RAM equals 1 register. Also limitations for the timing that could not get more improved. Though, comparing to the previous results with the power meter and smaller filter chain the results are more reasonable for this bigger design.

10 Overall Analysis and Conclusion

In this section a discussion about the MBD tool HDL Coder will be presented and compared with the XSG and DSPB design tools. Tips and tricks for designing with HDL Coder are provided and problems faced during the thesis with both MATLAB and HDL Coder. Lastly, a conclusion section is presented to sum up the thesis work and possible future work in this area is described.

10.1 HDL Coder

The results about the HDL Coder tool as a high level tool is based on sections 7, 8 and 9. Additionally, the experiments with logic test in section 6.5 and the MATLAB code designs in chapter 7 is used for evaluating HDL Coder as a HLS tool. As a high level tool HDL Coder can result in much more productive methodology for FPGA projects. Because that the tool is not vendor specific, a model can create designs for both Xilinx and Intel FPGAs. Though, the downside of the tool not being vendor specific is the less optimized resource mapping to the device, because it is not specific as XSG and DSPB is for their devices. In the experiments some of the values were increased a lot compared to the designs in XSG and DSPB. As explained earlier these results can be made more comparable in HDL Coder by using another filter structure and the optimized HDL filter block. This however requires more hardware specific knowledge. For HDL Coder a big part of optimizing the design was effected by redesigning and changing the blocks used. The designing in HDL Coder varies compared to XSG and DSPB when necessary parts of the design is done automatic, for example the clock systems. Therefore our opinion is that the abstraction level for HDL Coder is higher than the XSG and DSPB vendors. Additionally, the optimization options that was available was easy to use and do not exist in any other high level tool. So HDL Coder can be used with relative low knowledge of hardware and there is not many options in each of the subsystems to choose from compared to its vendor dependent counterpart. Thus, making it easy to use to create a functional design from specifications could be done quite easily. The generated design and RTL is readable and traceable. For the bigger branched design, it was noticed that the optimization options had a greater impact and the flexibility was better than for both the smaller designs. Though, looking at the results from the experiments it is showed that the optimization possible for HDL Coder versus designing in XSG or DSPB is not enough to get good utilization and timing results.

The architectural optimizations are more powerful and will lead to better hardware mapping on the FPGA fabric. For example, a complex multiplier in HDL Coder can be implemented as a single product block which decomposes to four multiplier and two adders in the generated model. This can also be implemented using the three multipliers and five adders and needs to be explicitly specified in the design in order to save the critical DSP resources on the FPGA. Moreover, if specific DSP blocks in the FPGA are to be targeted the design needs to be more elaborated and should match closely to the FPGA architecture. Another example would be the multiplication with constants. The gain blocks should be used for these kind of operations with the CSD option enabled compared to the product block. This optimization utilizes shifts and adds to implement

a product instead of using the DSP resource for the same. MATLAB code can also be added using MATLAB function block within Simulink and should be used only if the given functionality can not be implemented using HDL Coder supported Simulink blocks.

XSG and DSPB use hardwired blocks but HDL Coder take the functionality and analyzes it on a system level before producing RTL. They provide much more granularity and control. The blocks from these tools come with many options to pick and chose from and one needs a much higher level of technical expertise, understanding of the blocks and options. There blocks are also highly optimized for the vendor hardware and requires a bit of understanding and manual work to achieve similar results in HDL Coder. The dierence in performance can be reduced by expressing the design in a more detailed manner but this lowers the level of abstraction considerably. This could be a bottleneck while designing in HDL Coder as for the time taken to design. Both, DSPB and XSG are having a lower abstraction level for a design that targets their own hardware, with HDL Coder, it is vendor independent and designer needs to take additional efforts to map it to the right resources.

Unique to HDL coder is the code to model and model to code traceabilty. The Native Floating Point library and automated fixed point conversion using the Fixed Point Tool from reference floating point model are also exclusive in HDL Coder. However, the native floating point library results in bulky and inefficient design so it is only to be used when the design really needs it. HDL Coder also does Delay Balancing where the tool automatically inserts delays on parallel data paths. Additionally, the simulation environment and the possibilities provided with HDL Coder are really good. Moreover, the same design can also be ported to an ASIC.

The main difficulty using HDL Coder is mapping the design to the right resources. For example, if you have a chain of DSP blocks, where a single summation ends up outside a DSP block, it completely breaks the chain in or chain out and puts all summation in logic instead of DSP. It was found that the most productive way of working with HDL Coder would be to work on small isolated blocks and get them to properly map onto the hardware resources. After they are optimized for the selected hardware, they can be included in the larger system. Though, with the knowledge gathered throughout this thesis it still remained a issue to effect the mapping of the resources. More general issues are presented in section 10.5.

In section 6.5, efficient hardware generations techniques were checked with HDL Coder and it turns out that HDL Coder does not consider these optimizations and produces inefficient hardware for some cases. This also can be overcome by designing with more detail and having a hardware thinking while designing. For the tool vendors, this thesis will be a stimulus to further improve their tools and make MBD the tool of choice for hardware design in the near future.

Some important factors for using HDL Coder tool is the user experience of it, if it takes a lot of time to learn or design then it would not be a productive tool anymore. Some of the relevant parameters for designer experience are discussed in the below section.

10.2 Comparison of the Different Vendors

Based on our experiences we have evaluated and prepared radar charts based on the above parameters. Seen in figure 16 is the evaluation results gotten from this thesis. The ratings are from 0 to 5 where 5 is the best and the motivations for the ratings is described below.

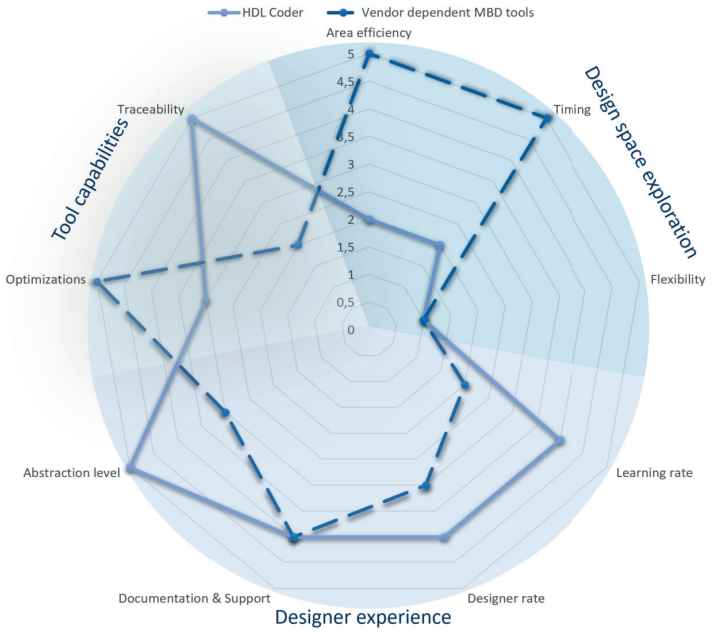

Figure 16: Our experience with HDL Coder compared to vendor dependent high level tools.

Abstraction level
The abstraction level of HDL Coder is higher than for XSG and DSPB and much higher than hand written RTL. In comparison to hand written RTL there is no need to know RTL design to create a working design. However, some hardware design knowledge is profitable to make a better design. The abstraction level is raised by the automated options and the easy-to-use optimization options provided by the tool. The clock, enable and reset networks are created automatically. Moreover, delay balancing is possible for the entire design, which is usable for big designs. Also, enabling pipelining for timing optimization is rewarding and simple to use. Though, that these options are available automatically also result in that it is more difficult to effect the for example clock network or pipelining done by the tool.

Learning rate
Because of the high abstraction level the tool it is easy to learn. The simulation methods are easy to use for verifying the designs and to give an understanding. It is not necessary to have deep knowledge in writing RTL to create a doable design, which saves learning

time. Though, more optimization options are available and it is important to understand when to use what and how. Therefore, the first optimized and working design can take time, but it is assumed to take less time than for XSG and DSPB. When the HDL Coder tool was tested as a HLS tool a lot of time had to be spent to learn how to. How to write the code and understand the output.

Designer rate
Compared to XSG and DSPB less time can be spent on designing with HDL Coder. Though, one conclusion from the experiments in section 8 and 9 is that the designs have to get optimized. The optimization time is greater than the design time because the many options on both system level and block level. That is the cause if HDL Coder is used for MBD. Though, designing with HLS takes more time and the optimization options are not that many but difficult to use. To simulate the design is quick and easy to use, for verifying the functionality. Also, to synthesize and get results is very fast and smooth. There is no need to create an own project in Vivado or Quartus because it is done by HDL Coder Workflow Adviser. This saves time in evaluating the utilization and timing results.

Documentation and Support
MathWorks have very good documentations online for all available tools. Simulink and MathWorks provide lots of HDL Coder examples and tutorials for learning. It is also possible to download specific examples for designs to improve knowledge of how to best design. Additionally, a description of every possible block exist combined with examples of how to use them.

Design space exploration
The design space exploration consists of the synthesis results in terms of area efficiency (as in resource utilization), timing (slack) and flexibility (mapping between available resources). These results is based on the results gathered from the experiments done in this thesis. It was noticed that the area and timing results where not as good as the other two high level tools. It was not predictable of how large a design was going to be when synthesized or how the optimization options was gonna effect the results. Though for the bigger branched filter chain design some values was improved compared to the corresponding XSG design and the result was realistic and comparable. As for flexibility it was noticed that there where limitations for how much it was possible for the designer to effect the mapping of the resources. For example the timing results for the branched design in section 9, where the slack could not increase more.

Designer Experience
Some of these parameters is discussed in the previous part 10.1 with HDL Coder in perspective. To start with learning rate is fairly high for HDL Coder design tool but less so for optimization methods and using it as a HLS tool. Though, it is understood that for other high level tools the design time is increased with the complexity of the blocks and timing systems which makes HDL Coder seem quick. The same goes for how much time it takes to make one design (designer rate), the higher complexity the more time it takes. For other high level tools there is more to keep track of in the designs, like clock, enable and reset systems. Therefore, the design rate is less than for HDL Coder. Additionally, documentation and support for most tools are the same and generally very good. Lastly, the abstraction level of high level tools is increased in general but the

experience of HDL Coder in this thesis gives the impression of that it is even higher. This is because of all the automatic options and optimization that is available for HDL Coder that is not general for other high level tools. For example balancing delays is possible on system level for HDL Coder, on subsystem level for DSPB and not at all XSG design.

Tool capabilities
As mentioned there are a lot of different capabilities for optimimization a HDL Coder design and it is possible to trace design-to-RTL and RTL-to-design. In the experiments it was noticed an increase in optimization performance made by the tool for a bigger design. For these chosen filter experiments it was found out that a XSG and DSPB design can get highly optimized. The optimization techniques availabe in HDL Coder, as mentioned, is easy to use. Additionally, the system level optimizations for other HLS tools are absent and the designer has to consider this in the design. The traceability for HDL Coder is really good. Both the code and resources are traceable to the design and specific blocks and subsystem, which is an usable quality when optimizing the resources. Whereas the XSG and DSPB only can trace resources.

The learning rate, designer rate and the abstraction level along with the documentation & support constitute the designer experience. The optimizations offered by the tool and the data types supported by the tool have been classified as tool capabilities broadly.

10.3 Methodology

In the background section, two different types of methodologies for designing is discussed, in section 5.2 and 5.3. One other method is the traditional handwritten RTL. Based on the experience gathered in this thesis the pros and cons table 8 is prepared considering design for FPGA.

Methodology	**Pros**	**Cons**
Hand written HDL workflow	Highly tunable and optimizable. Full control over mapping resources.	Time consuming. Inefficient to reuse code across vendors.
Vendor dependent workflow	Highly efficient resource mapping. Reusable design.	Time consuming but better than hand written. Designs can not be used across vendors.
Vendor agnostic workflow	Quick and easy to design. Highly reusable cross designs.	Not efficiently resource mapped, less control over mapping. Can be used across vendors.

Table 8: Methodologies pros and cons.

10.4 Tips and Tricks

This work presented in this thesis is carried out in MATLAB 2018a within a Linux virtual server based system. Mention before it is not optimal to run MATLAB on a virtual server since it is not supported. In this thesis some other versions were also tested but it got concluded that the newest version had the most options and updated blocks that should be used. Therefore, it is recommenced to use the latest versions of HDL Coder while designing. The most effective way to optimize and compare different designs was to use scripts for the runs. The commands *hdlsaveparams* and *hdlrestoreparams* should be used to take snapshots of the different optimization runs. These can be used as different check points in the design and helps in exploring the design space much better. Also, to effectively simulate the design the Signal Analyzer and Logic Scope provides a easy way of debugging during the design phase.

In this thesis a few tips were collected to ease the HDL Coder design phase even more, these are listed below.

- To create the most optimized design when down sample or rate transmission blocks are used is to place a delay block directly after it. Another option is to enable adaptive pipelining where this correction is done automatically. For the upsample, the rate transition block or the repeat blocks should be used.
- HDL Coder do not integrate delays inside blocks for pipelining, this must be done with the block specifications if it is possible.
- Use only booleans as control signals for logical operations and avoid other mathematical operations on the control signals.
- If saturation of integer overflow is not necessary then turn it off, for a more optimized result.
- To model synchronous enable or reset ports on blocks place a State Control block in the subsystem (at the same level or higher) and configure it as synchronous. Additionally, on the blocks an optional enable or reset can be enabled to get the needed input.
- Another design tip is that HDL Coder removes all redundant logic in the design. This results in that outports are always needed to get valid results.
- In most cases it is better to use gain block when multiplying with constants with block properties for the 'ConstMultiplierOptimization' as auto and 'DSPStyle' to 'on' where applicable. This generates optimized HDL compared to a product block.
- A MATLAB function block is easiest used when there is an easy algorithmic implementation to be done where there does not exist any ready block for that specific function. When using a MATLAB function, instead of using *fimath* then use *hdlfimath* whenever possible. Additionally, uncheck *Allow direct feed through* in the block setup if the code only uses registers.

- If the design synthesized with HDL Coder does not meet the requirement constraints, highly optimized blocks from XSG and DSPB can be used. Also hand written code or code generated from a third party HLS tools can be used as a black box which can be integrated in HDL Coder.
- Output ports in enabled and triggered subsystems must have Initial output set to 0. Hence if these need to be assigned initial values they have to be modelled in HDL Coder.
- It would also be good to mention sample rates for each block if designing for multi rate model because if it is inherited on using the subsystem later, it could give errors.
- One more way to reduce the resources would be enable the options 'Minimize global resets' and 'Minimize clock enables'.
- Ensure proper type of reset is used for the selected FPGA for optimal resource mapping.

There where many difficulties of when to use individual optimization methods. When pipelines is needed in the design, the block properties are changed to add input and output pipeline registers. In this case it is a good practice to enable *Balance delays* so that all parallel paths are delay balanced. If adaptive pipelining is enabled it will add pipeline registers between critical points in the design and many of these are very specific cases and not easy to know about. Adaptive pipelining will also contribute to minimizing the critical path and therefore improve timing results. As for resource sharing it should be used carefully because it reduces area at the cost of operating frequency. It the frequency of the design is to be unchanged then it is recommended to keep these settings turned off and the *Sharing Factor* and *Streaming Factor* is kept at 0. Though, it is always a good idea to investigate the possibility. Additionally, it is also possible to share entire subsystem if the same functionality is used more than once by configuring the subsystem as an atomic subsystem.

Moreover, based on experience it is always better to generate all the reports HDL Coder has to offer and go through the generated model and the reports. The important things to observe in the reports are the optimization options that have taken effect in the model and latency in terms of clock cycles due to delay balancing. One interesting thing to note here is if distributed pipelining has been enabled, whether after the optimization the added registers have increased or decreased. The critical path shown is based on characterized model and hence is not always matching the critical path synthesized by the design. The latency in the optimization report should be checked and made note of before moving to the synthesis stage. Always check the report generated by HDL Coder for warning and messages. Some of these warning or messages might be critical for the design.

10.5 Problems Faced with MATLAB and HDL Coder

Below we present the errors and possible workarounds that we have encountered during our experiences with HDL Coder. To start with, HDL Coder partially supports floating

point, but not 'double'. Also the Fixed-Point Designer can be used to convert the floating point model to fixed point one. Introduction to fixed and floating point can be read in section [16]. When precision is vital for the design Native Floating Point can be used to solve the problem.

MATLAB is not tested for use on virtual machines. This resulted in some problems with MATLAB and some features in the tool that would have made work easier did not work. For example, the workspace variables that gets saved in MATLAB can be shown incorrectly in the variables pop-up window. Since Ericsson used these workspace variables to log and check functionality with respect to the reference model. This made it difficult to compare two variables from a run. Optionally, Simulink has an option of logging signals with the *Logic analyzer scope* and displaying them. This can come in very handy as the output for the simulation would be exactly the one expected out of the system.

Delay matching for enabled based subsystem. If there are same enabled based subsystems with the exact same functionality, HDL Coder throws out a error and complains that delay matching is not possible. The workaround here was to add a gain block of 1 where ever it complains. This fix does not affect the functionality but creates additional lines of code and also prevents the block re-usability which in turn impacts productivity.

10.6 Conclusions

The main problems for this thesis work labelled *Performance Evaluation of MathWorks HDL Coder as a Vendor Independent DFE Generation* were:

- Investigate a new methodology with a vendor agnostic workflow.
- Evaluate the performance of HDL Coder as an HLT and compare it with its vendor dependent counterparts,

It can be concluded that HLT tools are suited for FPGAs because of its reusabilty, ease to procure and can be used to make a prototype much quicker and easier, especially when time to market is of paramount importance. The model designed can also act as a golden reference model in case the design needs to be further optimized. The tools evaluated have improved design productivity many folds. The major time spent with these tools are for optimizations for meeting the timing and/or for better mapping on the FPGA fabric.

In this thesis an evaluation of different HLT for FPGA prototyping and design. The tools evaluated are XSG, DSPB and HDL Coder with the later being more focused upon due to its feature of producing vendor independent RTL and ease of re-targeting. HLS tools in general, raise the abstraction level for designing digital circuits. HDL Coder also provides the most natural way of representing hardware because the model created is very similar to the implementation. Hence, high level tools can reduce the burden on the designer to a great extent and allow the designer to design on a higher level without having much knowledge of the RTL. A few designs with increasing complexity were made and compared with the results obtained from HDL Coder.

The MBD workflow offered reasonable performance with greater ability to parameterize a design, visualize and re-use portions of an existing design. However the hardware generated by the vendor dependent tools are highly optimized in terms of resource usage and performance. These features could be a bottleneck for the selected FPGA. HDL Coder is useful to users new to hardware description and can be used in situations where power, area, or timing constraints are slightly relaxed. In conclusion, HDL Coder would be a good option to begin a design for a first prototype on the FPGA. Depending on the availability of resources and the margin for performance, one can use the vendor specific HLT's and other form of hardware description.

During our work, we have discovered a number of challenges that HDL Coder currently has and have been listed below.

- Learning the tool. It was needed to know as much as to say that the tool is evaluated properly.
- Handle and understand all of HDL Coder optimization techniques. Difficult to know when to be done.
- Learn enough of DSPB and XSG designs to understand their synthesis values.
- Two ways to optimize, redesign architecture or optimization options provided by tool.

10.7 Future Work

This thesis was preliminary study of the HDL Coder tool and made by us as students with little to no experience of MBD. The verification of the design is not performed in this thesis and can also be undertaken as a future work. This thesis emphasizes on the design aspect of a given functionality on a FPGA. Not all parameters for example power were explored and hence could be a point of discussion.

- The procedure that was used in this thesis is not the most optimal. A done reference design and RTL was given and the functionality was analyzed then redone in HDL Coder. Whereas, a HDL Coder design should be made in an earlier stage. When the functionality of the system or component is specified then it is to be implemented in HDL Coder so that the design do not get influenced by for example the RTL or another design made in XSG or DSPB. A future work of this could be that a specified functionality is given and three designs with the same functionality is made in HDL Coder, XSG and DSPB. Then these designs are synthesized and the results gets evaluated and compared.
- Evaluate designs using a mix of HDL Coder and DSP Builder or SysGen blocks. During this thesis it was understood that it was possible to combine the HDL Coder Simulink blocks with blocks from another vendor, for example XSG or DSPB. In areas where another vendor has produced a very efficient block that could be used in a HDL Coder block design and generate a more optimized result. For example in this thesis the knowledge about XSG and DSP optimized filter blocks was gained

and that they could be used instead. This was never tried and is a good future work to evaluate.

- Make a block level comparison for the different vendors. Compare individual blocks for different vendors and observe the generated RTL. Some block level optimization comparison could be interesting. The result of this could be used in the previous dot for future work.
- For bigger companies, a library of HDL Coder blocks and bigger designs, systems and components are needed. For reuseability a collection of designs could be done, with different optimization properties that could be included in the designs.
- Investigate compiler optimization options for efficient hardware generation.

11 References

[1] Naeem Abbas. "Acceleration of a bioinformatics application using high-level synthesis". PhD thesis. École normale supérieure de Cachan-ENS Cachan, 2012.

[2] Hikmat N Abdullah and Hussein A Hadi. "Design and Implementation of a FPGA Based Software Defined Radio Using Simulink HDL Coder". In: *Engineering and Technology Journal* 28.23 (2010), pp. 6750–6768.

[3] Matthew S Allen. "Performance Assessment of Model-Driven FPGA-based Software-Defined Radio Development". In: (2014).

[4] Gerald Baguma. *High Level Synthesis of FPGA-Based Digital Filters*. 2014.

[5] HDL Coder. *Accelerate Design Space Exploration Using HDL Coder Optimizations*. Accessed: 2019-04. 2014. URL: https://se.mathworks.com/videos/accelerate-design-space-exploration-using-hdl-coder-optimizations-81998.html?elqsid=1557227738933&potential_use=Student.

[6] Florent De Dinechin and Bogdan Pasca. "Large multipliers with less DSP blocks". In: *Field Programmable Logic and Applications*. IEEE. 2009.

[7] Jack Erickson. *HDL Coder Evaluation Reference Guide*. Accessed: 2019-04. 2019. URL: https://www.mathworks.com/matlabcentral/fileexchange/58941-hdl-coder-evaluation-reference-guide.

[8] Deepak Gopi. "Digital front end for base-station RF". PhD thesis. Rutgers University-Graduate School-New Brunswick, 2011.

[9] Tim Hentschel, Gerhard Fettweis, and W Tuttlebee. "The digital front-end: Bridge between RF and baseband processing". In: *Software defined radio: enabling technologies*. John Wiley & Sons, 2002, pp. 151–198.

[10] Intel. *Arria 10 Overview*. Accessed: 2019-04-20. 2019. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf.

[11] Intel. *Introduction to DSP Builder for Intel FPGAs*. Accessed: 2019-05. 2019. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/dspb/hb_dspb_intro.pdf.

[12] Joonas Järviluoma. "Rapid Prototyping from Algorithm to FPGA Prototype". In: (2015), p. 60.

[13] Wim Meeus et al. "An overview of today's high-level synthesis tools". In: *Design Automation for Embedded Systems* 16 (2012), pp. 31–51. DOI: 10.1007/s10617-012-9096-8. URL: https://doi.org/10.1007/s10617-012-9096-8.

[14] Valerie Youngmi Sarge. "Evaluating Simulink HDL coder as a framework for flexible and modular hardware description". PhD thesis. Massachusetts Institute of Technology, 2018.

[15] Rahul K Shah et al. "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)*. IEEE. 2017, pp. 346–351.

[16] *Specify Data Types Using Data Type Assistant*. Accessed: 2019-04. 2019. URL: https://se.mathworks.com/help/simulink/ug/specify-data-types-using-data-type-assistant.html.

[17] Xilinx. *System Generator for DSP*. Accessed: 2019-04. 2019. URL: https://www.xilinx.com/products/design-tools/vivado/integration/sysgen.html.

[18] Xilinx. *Zynq Ultracale+ Datasheet*. Accessed: 2019-04-20. 2019. URL: https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf.

12 Appendix A

12.1 Power Meter MATLAB Code

Code 1: Initial design of the power meter

```
function ref_pm_acc_out  = ref_pm(ref_pm_in_i, ref_pm_in_q)

F1 = fimath('OverflowAction','Saturate','RoundingMethod','Floor');
F2 = fimath('OverflowAction','Wrap','RoundingMethod','Floor');

persistent s7
if isempty(s7)
    s7 = fi(0,0,34,15,F1);
end

i1 = fi(ref_pm_in_i,1,16,15,F2);
q1 = fi(ref_pm_in_q,1,16,15,F2);

i2 = fi(i1*i1,F1);
q2 = fi(q1*q1,F1);

i3 = fi(i2,1,17,15,F1);
q3 = fi(q2,1,17,15,F1);

i4 = fi(i3,0,16,15,F1);
q4 = fi(q3,0,16,15,F1);

s5 = fi(i4 + q4,F1);

s6 = fi(s5,0,34,15,F1);

s7 = fi(s6 + s7,0,34,15,F1);

ref_pm_acc_out= s7;
end
```

13 Appendix B

13.1 Power Meter MATLAB Pipelined Code

Code 2: Initial design of the power meter

```
function ref_pm_acc_out = ref_pm_sub(i, q)

F1 = fimath('OverflowAction','Saturate','RoundingMethod','Floor');
F2 = fimath('OverflowAction','Wrap','RoundingMethod','Floor');

persistent s7 i2 q2 s5;
if isempty(s7)
    s7 = fi(0,0,34,15,F1);
    i2 = fi(0,1,32,30,F1);
    q2 = fi(0,1,32,30,F1);
    s5 = fi(0,0,17,15,F1);
end

i1 = fi(i,1,16,15,F2);
q1 = fi(q,1,16,15,F2);

i2 = coder.hdl.pipeline(fi(i1*i1,F1),3);
q2 = coder.hdl.pipeline(fi(q1*q1,F1),3);

i3 = fi(i2,1,17,15,F1);
q3 = fi(q2,1,17,15,F1);

i4 = fi(i3,0,16,15,F1);
q4 = fi(q3,0,16,15,F1);

s5 = coder.hdl.pipeline(fi(i4 + q4,F1));

s6 = fi(s5,0,34,15,F1);

s7 = coder.hdl.pipeline(fi(s6 + s7,0,34,15,F1));

ref_pm_acc_out = s7;
end
```

14 Appendix C

14.1 Power Meter MATLAB Code

Code 3: Multiplier in HDL Coder

```
function mult_out  = mult(i_in, q_in)
F1 = fimath('OverflowAction','Saturate','RoundingMethod','Floor');
mult_out = fi(i_in * q_in,0,16,15,F1);
end
```

Code 4: Adder in HDL Coder

```
function add_out  = add(i_in, q_in)
F1 = fimath('OverflowAction','Saturate','RoundingMethod','Floor');
add_out = fi(i_in + q_in,0,16,15,F1);
end
```

Printed by Tryckeriet i E-huset, Lund 2019

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2019-721
http://www.eit.lth.se