UPPSALA
UNIVERSITET

IT 14 039
Examensarbete 30 hp
Juni 2014

High Level Synthesis of FPGA-Based Digital Filters

Gerald Baguma

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

High Level Synthesis of FPGA-Based Digital Filters

Gerald Baguma

This thesis work is aimed at the high level synthesis of FPGA based IIR digital filters using Vivado HLS produced by Xilinx and HDL coder produced by MathWorks. The Higher Layer Model of the filter was designed in Vivado HLS, MATLAB and Simulink. Simulations, verification and Synthesis of the RTL code was done for both tools. Further optimizations were done so that the final design could meet the area, timing and throughput requirements. The resulting designs were later evaluated to see which of them satisfies the design objectives specified.
This thesis work has revealed that Vivado HLS is able to generate more efficient designs than the HDL coder. Vivado provides the designer with more granularity to control scheduling and binding, the two processes at the heart of HLS. In addition, both tools provide the designer with transparency from modeling up to verification of the RTL code.
HDL coder did not meet timing. Vivado HLS on the other hand met the timing requirements. The limitations of each design flow are also discussed in this report. A review of the tools available on the market today was also done and recommendations about them made.
Finally, this thesis work recommends that ABB HVDC should adopt the HLS methodology using Vivado in order to achieve accelerated development. More work should be done to evaluate the possibility of auto C/C++ code generation for RTL synthesis in Vivado. Lastly, an evaluation on the LabVIEW environment should be done as an alternative to the HLS methodology.

Handledare: Jimmy Öhman
Ämnesgranskare: Ping Wu
Examinator: Philipp Rümmer
IT 14 039
Sponsor: ABB AB

Tryckt av: Reprocentralen ITC

Contents

List of Figures

List of Tables

Acronyms

ASIC	- Application Specific Integrated Circuits
AST	- Abstract Syntax Trees
AXI4	- Advance eXtensible Interface 4
AXI4	- Advance eXtensible Interface 4 Lite
AXI4-Stream	- Advance eXtensible Interface 4 Stream
DPM	- Dual Port Memory
DRAM	- Dual Port Random Access Memory
DSP	- Digital Signal Processing/ Digital Signal Processor
DUT	- Device Under Test/Design Under Test
eTDM	-enhanced Time Division Multiplexing
FF	- Flip Flop
FIFO	- First In First Out
FIL	- FPGA In-the Loop
FIR	-Finite Impulse Response
FFT	- Fast Fourier Transform
FSM	- Finite State Machine
GUI	- Graphical User Interface
HDL	- Hardware Description Language
HIL	- Hardware In-the Loop
HLM	- Higher Level Language
HLS	- High Level Synthesis
HVDC	- High Voltage Direct Current
IIR	- Infinite Impulse Response
II	- Iteration Interval

LUT	- Look Up Table
RAM	- Random Access Memory
RTL	- Register Transfer Level
SSA	- Static Single Assignment
TPL	- Textual Programming Languages
VHSIC	- Very High Speed Integrated Circuit
VHDL	- VHSIC Hardware Description Language
VPL	- Visual Programming Language

Chapter 1. Introduction

1.1 Background

Electronic products currently are composed of highly complex designs in such areas as; communication, control, medical, defense and consumer electronics. They feature in applications such as digital signal processing (DSP), communication protocols, soft processors etc. Many DSP algorithms such as FFTs, FIR or IIR, which were previously built using application specific integrated circuits (ASICs) can be built on FPGAs with very high flexibility. In addition, these devices offer better economic prospects as compared to the ASICs. Consequently designs that were previously implemented on ASICs have experienced a move to the reconfigurable technology [27]. These designs have become increasingly complex and are stretching the boundaries of device density, design performance and device power consumption. It's also always the objective of designers to minimize costs by utilizing device resources appropriately to meet design objectives. In addition, given the shortened windows of design development time, it's very important to hit the target for the design objectives within the allocated time and schedule. Many downstream problems can be avoided with an appropriate methodology during the design flow. By taking appropriate steps early in the design phase, significant design productivity and minimized iterations can be achieved. It is therefore important to utilize tools that offer a good design methodology and provide proper estimates of project viability, cost and design closure early in the design phase. Product designers currently are looking for tools that can provide accelerated design productivity with a very high degree of reliability.

In the applications of ABB HVDC (High Voltage Direct Current), voltage and current measuring IO-units in the Modular Advanced Control for HVDC system (MACH) [22] perform digital filtering of analogue signals after analog to digital conversion. The filtering in the digital domain is done by in Digital Signal Processors (DSP) and / or Field Programmable Gate Arrays (FPGA). An efficient way for filter designing is using VHDL (Very-high-speed-integrated-circuits Hardware Description Language). When filters are implemented in FPGAs the corresponding VHDL-code is usually written at Register Transfer Level (RTL) which is thereafter synthesized into logic gates. This means that the filter architecture and characteristics need to be determined before the implementation is done. Also, once the implementation is done an architectural change on the filters may cause a large impact on the implementation, and may result into a change of most of the RTL-code.

There is plenty of High Level Synthesis (HLS) tools available for FPGA design on the market today e.g. HDL coder tool, Vivado HLS tool, Catapult e.tc. An HLS tool usually takes in a higher level language description, for example in C, C++, MATLAB /Simulink or System-C and then based on directives, translates the high level code into RTL-code which can then be synthesized into logic gates. With this methodology one can easily make changes in an algorithm and/or directives and have the tool automatically regenerate the RTL-code.

This methodology offers great benefits such as late architectural or functional design changes without time consuming in rewriting of RTL-code, algorithms can be tested and evaluated early in the design cycle and development of accurate models against which the final hardware can be verified. In addition to this, if by any means the process yields good performance designs, then the amount of errors can be significantly reduced. This methodology may however pose challenges as the generated RTL-code might not be as effective (in terms of area, timing, throughput etc.) as hand written RTL-code, handmade changes in the generated RTL-code (for timing, area etc.) could be lost once the code is regenerated and the code for FPGA-vendor specific tools may not easily be portable to another vendor. By leveraging on the benefits of HLS and the design methodology, this thesis aims at choosing the best tool chain for use in RTL code synthesis. An analysis regarding the device area usage, timing, latency and throughput shall be done. The resulting resource efficient design shall be tested on the PS74x IO boards for voltage and current measurement in the MACH system.

1.2 Problem Statement

Product design implementation on FPGAs, usually involves writing VHDL-code at Register Transfer Level (RTL) which is then synthesized into logic gates. Complexity and strict time schedules have resulted into shortened development cycles. In addition, architectural and functional changes later during design are time consuming since RTL-code has to be changed or re-written. This methodology also provides designers with no mechanism with which algorithms can be tested and evaluated early in the design process. Developers also need to formulate appropriate models upon which the final hardware can be verified, which the above method does not support.

In order to combat these limitations, adequate tools and a proper methodology need to be found out and used, for designers to establish proper project time estimates, project viability, cost and design closure so as to be able to achieve accelerated design productivity while product quality and time to market remain uncompromised.

1.3 Objective

The main objective of this project is to convert ABB's hand coded VHDL Infinite Impulse Response (IIR) filter design into a Higher Level Model (HLM) design using both the Xilinx tool chain [23] and the Mathworks tool chain [24]. The RTL-code generated through HLS shall then be compared to the current hand coded filter design in terms of performance. Measures such as area, timing, throughput, latency and correctness of the resulting RTL-VHDL code shall be evaluated by running the design through the ABB's target system.

In particular, the specific objectives of this thesis work were:

a) To convert the ABBs Infinite Impulse Response (IIR) filter designs currently implemented in RTL-code to a Higher Level Model (HLM) and use Xilinx High Level Synthesis (HLS) and the ultrafast design methodology to generate corresponding RTL-code. The generated code shall be implemented on an existing ABB IO board in the

MACH system, and the evaluation of its performance shall be made.

b) To implement a corresponding MATLAB /Simulink Model of the same filter described in (a) and by utilizing the Mathworks Hardware Description Language (HDL) coder, generate the corresponding RTL-code. The generated code shall be implemented on the existing ABB IO board, and the evaluation of its performance shall be made.
c) To compare the Xilinx Vivado HLS tool and the Mathworks HDL Coder tool. The benefits and drawbacks realized from each design and implementation shall be clearly documented.
d) To test the filter designed using the Vivado HLS tool on the Xilinx Zynq System on Chip. This shall be targeted for a project already designed and implemented on the Xilinx Spartan-6 that has been converted to the Zynq development board at ABB Corporate Research.

1.4 Scope

This thesis is concerned with converting an existing IIR filter into a Higher Layer Model (HLM) and testing how this model can generate code that scales to the handwritten code in performance. Measures such as area, throughput, Latency, timing and correctness were evaluated and the results tested using ABB's PS74x modules. The methodology for design adopted by each tool was evaluated and the conclusions were drawn on which kind of methodology suits the needs of developers.

1.5 Tools

The IIR filter was designed using the filter design and analysis tool in in MATLAB. The filter design tool also contains a library of numerous methods for testing and evaluating the performance of filters. These together with MATLAB /Simulink environment were used to design, simulate and analyze the filter.

The Mathworks' HDL coder and Xilinx's Vivado HLS tool were used for high level synthesis of RTL-VHDL. The VHDL synthesized was simulated using ModelSim and ISIM and the best design was tested by running the implementation through ABB's PS74x modules.

In addition, both the Vivado HLS design and the HDL coder design were synthesized using the ISE studio so that a proper evaluation of the design could be performed. The ISE® Design Suite is an industry proven Electronic Design Automation (EDA) tool for the Xilinx all programmable devices [25]. Additional settings were added during this phase so that better logic could be synthesized. It is important to note that each tool is capable of invoking ISE from within itself. However it might be helpful, if other additional settings are needed, to directly configure a project in ISE.

1.6 Ethical Considerations

During the course of this thesis there was no intentional negligence leading to fabrication of the scientific message or a false credit or emphasis given to a scientist by properly referencing the accredited scholars whose area of scholarly was of significance to this thesis work. In particular all the patent and intellectual property rights as specified under the terms of use of the associated documents were adhered to. The work presented in this thesis was thoroughly reviewed to ensure that there is no Intentional distortion or misrepresentation of any quoted research.

1.7 Methodology

The work done was benchmarked against seven milestones:

1) Relevant research and development
2) Design
3) Modeling and simulation of the digital filter
4) Vivado design flow analysis; HDL coder design flow and analysis
5) Evaluation and comparison of the design flows offered by the tools
6) Hardware Integration and testing
7) The conclusion and recommendations

Each milestone (M) presents a series of work packages that represent a complete set of actions geared to achieving a goal. In addition each of the work packages (WP) has a deliverable/outcome that is attached to it as presented in the table below;

Table 1.1 Summary of methodology

Milestone	**Activity**	**Deliverable**
M 1.0 Relevant research and development	**WP 1.1** Digital filter design and realisation	**D 1.1** Understand the tools and methods for design
	WP 1.2 The synthesis task	**D 1.2** Understand the core of research topic
	WP 1.3 Scheduling and Scheduling Algorithms	**D 1.3** Algorithm in use established
	WP 1.4 Allocation and Allocation Algorithms	**D 1.4** Algorithms in use established
	WP 1.5 Analysis of current research and research gap presentation and Approach development	**D 1.5** Concrete measure and method of analysis
	WP 2.1 Design, modeling and simulation of the IIR digital filter.	**D 2. 1** IIR filter model

M 2.0 Design, modeling and simulation of the digital filter	**WP 2.2** Comparison of the model with the current filter	**D 2.2** Equivalent filter model
M 3.0 Vivado design flow and analysis	**WP 3.1** Synthesis methodology in Vivado	**D 3.1** Characterization of method and equivalent filter design implementation
	WP 3.2 Area optimizations in Vivado	**D 3.2** Area optimized design
	WP 3.3 Latency and throughput optimization in Vivado	**D 3.3** Latency and throughput optimized design
	WP 3.4 Timing optimization in Vivado	**D 3.4** Design with timing constraints met
	WP 3.5 Comparison of the optimized filter with the current filter solution	**D 3.5** Performance statistics relating the synthesized design with the current design
	WP 3.6 Results summary and tool evaluation	**D 3.6** Overall summary of the tool's quality and performance attributes
M 4.0 HDL coder design flow and analysis	**WP 4.1** Synthesis methodology in HDL coder	**D 4.1** Characterization of method and equivalent filter design implementation
	WP 4.2 Area optimizations in HDL coder	**D 4.2** Area optimized design
	WP 4.3 Latency and throughput optimization in HDL coder	**D 4.3** Latency and throughput optimized design
	WP 4.4 Timing optimization in HDL coder	**D 4.4** Design with timing constraints met
	WP 4.5 Comparison of the optimized filter with the current filter solution	**D 4.5** Performance statistics relating the synthesized design with the current design
	WP 4.6 Results summary and tool evaluation	**D 4.6** Overall summary of the tool's quality and performance attributes
M 5.0 Evaluation and comparison of the design flows offered by the tools	**WP 5.1** Comparison of area optimizations	**D 5.1** Statistics for area reductions and discrepancies
	WP 5.2 Comparison of latency and throughput optimizations	**D 5.2** Statistics for latency and throughput reductions and their discrepancies
	WP 5.3 Comparison of timing optimizations	**D 5.3** Statistics for timing improvement and their discrepancies
M 6.0 Hardware integration and testing	**WP 6.1** Implementation of the cascade filter	**D 6.1** source code for the cascade filter

	WP 6.2 Integration of the cascade filter into the current solution	**D 6.2** source code for the updated solution
	WP 6.3 Hardware testing	**D 6.3** Performance results of the solution on hardware
M 7.0 Conclusion and recommendation	**WP 7.1** Conclusion and recommendation	**D 7.1** Deductions on the best design flow
	WP 7.2 Future work	**D 7.2** Recommendations for future research

1.8 Report Outline

This thesis is divided into seven chapters and two other sections, one for references and the other for appendices.

Chapter one provides an overview of this thesis. It establishes the background of this thesis work and presents a summary of the methodology. The chapter also establishes the Justification, Ethical Considerations and scope for this thesis work.

Chapter two reviews the relevant research and development that was used in specifying the problem, obtaining and evaluating an appropriate solution. It establishes the background concepts of High level synthesis in general and leverages on these concepts to draw meaning to the existing problem and solution. The main purpose of this chapter was to draw a ground criterion for evaluating the technique that each of the tools utilizes to realize RTL level code. In addition, this chapter serves to establish the current research gap and to also reach to a meaningful conclusion as to whether this kind of design methodology is "ripe" enough for industrial applications.

Chapter three focuses on the design, modeling and simulation of the digital filter. It also seeks to compare the results of the model with the current filter implementation.

Chapter four presents the design flow used by the Vivado HLS tool. Its core goal was to implement a C/C++ model of the filter; perform a C/C++ simulation and a Co-Simulation using ModelSim/ISIM and optimize the design to at least meet static timing constraints. In addition this chapter gives an extensive discussion about the results of analyzing the performance statistics to ensure that all the improvements and reductions in performance with respect to the handwritten RTL were reported. Projections (especially when the tool algorithms and decisions were not clear) and explanations for the reasons why the performance was so were also clearly investigated and elucidated.

Chapter five deals with the design flow used by MathWorks' HDL coder. Its core goal was to implement a MATLAB /Simulink model of the filter; perform a simulation and a Co-Simulation using ModelSim and optimize the design to at least meet static timing constraints. In addition this chapter focuses on the results of analyzing the performance statistics to ensure

that all the improvements and reductions in performance with respect to the handwritten RTL were reported. Projections (especially when the tool algorithms and decisions were not clear) and explanations for the reasons why the performance was so were also clearly investigated and elucidated.

Chapter six focuses on the implementation of a cascaded IIR filter of order 14, integrating it into the current design and acting out tests on the resulting solution.

Chapter seven offers conclusions on the design methods employed by each tool and recommendations for the choice of tool that was found to be best suited for use by designers or the absence thereof.

A list of the references used in carrying out this thesis work is provided after chapter seven. In the appendices all code listings are documented.

Chapter 2. Relevant Works

2.1 Introduction

The FPGA circuit design flow that has become mainstream for decades is where, the hardware designer manually refines the behavioral system specifications down to the RTL. From that point, RTL synthesis and PAR (Place and Route) complete the design flow. In addition verification of these designs has to be done to match the behavior with the specifications and remove discrepancies where necessary [3].

From the deductions of Moore [10], increasing functionality can be integrated on a single chip. A similar increase in design teams to match device functionality is not only unpractical but also uneconomical. This means that design productivity must be improved. With the increasing transistor count, therefore, it's reasonable to say that a tradeoff of chip area for increased productivity may be in order.

The Synthesis task, as investigated in this thesis, is to take a specification of the behavior required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behavior while satisfying the goals and constraints. In this sense, behavior means the way the system or the components interact with the other components in the system as described further in [1] [3]. The synthesis that is evaluated in this thesis work only focusses on the algorithmic level as described in [1] [3], and seeks to evaluate the best possible design flow that can reliably achieve the goals described above. In summary, the goal of this thesis is to evaluate, through implementation of an IIR digital filter, how HLS can accelerate productivity and also how different design flows can meet the design constraints of timing, area and throughput.

2.2 FPGA design Overview

The field-programmable gate array (FPGA) is a semiconductor device that can be programmed after manufacturing. Instead of being restricted to any predetermined hardware function like an application specific integrated circuit (ASIC), an FPGA allows a designer to program product features and functions, adapt it to new standards, and reconfigure the hardware technology for specific applications even after the product has been installed in the field-hence the name "field-programmable" [28][29]. The FPGA configuration is generally specified using a hardware description language (HDL). FPGAs can be used to implement any logical functions that ASICs perform. In addition, the ability to update the functionality after shipping offers advantages for many applications as compared to the ASICs. Specifically FPGAs offer the following advantages as compared to the ASICs.

- Rapid prototyping
- Shorter time to market
- The ability to re-program in the field for debugging
- Long product life cycle to mitigate obsolescence risk

FPGAs contain programmable logic components called "logic blocks or Logic elements", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" - somewhat like many changeable logic gates that can be inter-wired in many different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory [28]. *Unlike previous generation FPGAs using I/Os with programmable logic and interconnects, today's FPGAs consist of various mixes of configurable embedded SRAM, high-speed transceivers, high-speed I/Os, logic blocks, and routing* [28] [29] [30].

FPGAs have continued to evolve and as Altera states in [28], the devices have become more integrated. Hard intellectual property (IP) blocks built into the FPGA fabric provide rich functions while lowering power and cost and freeing up logic resources for product differentiation. Newer FPGA families are being developed with hard embedded processors, transforming the devices into systems on a chip (SoC) [29] [30] [31].

2.3 The Synthesis Task

The system to be designed is usually represented in a high level language typically C/C++, SystemC, MATLAB, Simulink models [1] [2]. As described in more detail in [1], the first step to high level synthesis is usually the compilation of the formal language into internal representations. Typically parse trees and graphs are used. These graphs depend on the internal ordering of the operations carried out in the program.

The specification is initially designed at a level for human readability and not for direct translation into hardware, so optimizations have to be performed first. Typically transformations like compiler-like optimizations as deadcode elimination, constant propagation, common sub-expression elimination, inline expansion of procedures and loop unrolling. Local transformations, especially those that are much more specific to the hardware being used are also added to the optimization routines.

As described in [1] [4], then comes the core of transforming behavior into structure; Scheduling which consists of assigning operations to control steps i.e. a fundamental sequencing unit in synchronous systems which also corresponds to a clock cycle, allocation which consists of assigning operations to hardware.

The aim of scheduling is to minimize the number of control steps needed for the completion of the program given some constraints on the hardware resources. In essence if more speed is required then more resources have to be utilized. This can be achieved by packing the control graph as tightly as possible i.e. observing only the essential dependencies required by the data flow graph and by the loop boundaries.

The aim of allocation is to minimize the amount of hardware needed by the program. The hardware usually consists of functional units also referred to as cores (such as Adders, multipliers, pipelined multipliers), memory elements (such as block RAMs) [4] and

communication paths (such as multiplexers and buses). Normally manufacturers provide a separate library that is supported for each kind of hardware technology [1] [4]. Since minimizing them together is usually a complex process, most systems minimize them separately as described in [1]. The goal is such that mutually exclusive operations can share functional blocks or cores. The optimization then focuses on grouping these cores such that the minimum number of groups of functional units that are mutually exclusive results. If certain cores in a specific technology library can perform certain operations, then the goal is to group these operations together. Once this optimization problem is solved then the given schedule is minimal. This process also translates to minimizing the amount of storage elements as well as the communication paths.

An important point for the designer to know is that manufactures typically allocate cores depending to the device in question. Effectively the synthesis problem for each device family is different since a different set of libraries supported for each device family is made available. The delays associated with each core (functional unit) affect which cores can be scheduled in a single clock cycle. The created schedule depends upon what cores the operators are bound to. The scheduling process takes into consideration the effects that binding (allocation) has on the design.

In memory allocation values that are generated in one step and used in another step are allocated storage. Particular attention is paid to minimization of registers and access times i.e. simplifies communication paths. Also, communication paths should be selected so that the functional units can support data transfers as necessitated by the schedule. The most simple of which is only multiplexers, and it offers the highest throughput. Buses which can be seen as shared multiplexers allow for minimization of resources but at the expense of throughput. A more reasonable design policy could be to use a hybrid of the both schemes.

The system must then decide how each component of the data path is implemented. This is called binding. Typically the process involves the use of specific device libraries that correspond to the device technology. As thus it can be described as the process where the scheduled operations are bound to specific hardware implementations (or cores) from the technology library [4]. However as described in [1], libraries may limit the possibilities of efficient solutions but are a necessary evil.

Once the schedule and data paths have been chosen, it’s then necessary to synthesize a controller that will drive the data paths as the schedule necessitates it. If hardwired control is chosen a control step corresponds to a state in controlling a finite state machine. Once the interface is known, then the FSM can be synthesized using the known methods and techniques, including the state encoding and logic optimizations [1].

The major constraint as stated in [1], for most of these tools, is the enormous number of design possibilities. However, this thesis also uncovers that given proper directives and constraints to

the input source code or model, a good tool should be able solve the problem of an infinite design space.

2.4 Place of the Designer and Tool Quality Attributes

As stated in [3] that, *the quality of the final result does not only depend on the HLS tool used but also on the integration with the downstream design flow and design libraries.* In this section, the key desired attributes in a tool are explained. Key issues such as; *how the designer should input design specifications and constraints; how the HLS tool should output results, what decisions the designer should make and what information the designer needs in order to make them, and how the system is to explain to the user what is going on during the design process [1]*. This section is the very embodiment of the measure that is used to evaluate the HLS tool performance by serving as a benchmark for the desired traits and behavior.

2.4.1 Definition Language and Ease of Implementation.

Ideally HLS should bridge the gap between algorithm design and hardware design. It should open up hardware design to system designers that have until now no hardware design experience. The tool should offer the system designer a design flow such that hardware design can start from the code written by the algorithm designer rather than having to re-implement the design in a different language. In addition, tools that provide expressiveness multiplicity (i.e. support both Textual Programming Languages (TPL) as well as Visual Programming Languages (VPL)) are ideal. It is also obvious that certain restrictions may be obvious [3], on the high level language however these restrictions should not take away the expressiveness of that language (i.e. should be as loose as possible).

As explained in [3], *the connection between the language in which we think/program and the problems and solutions we can imagine is very close, but also because an overly restrictive design language may make programming a certain behavior very hard (and frustrate the designer during the process).*

2.4.2 Tool Complexity, User Interface and Documentation

As stated in [3], a good HLS tool should have a reasonably flat learning curve. This can be achieved with a simple to use graphical user interface that guides the designer through the design flow. Tool functionality should be easily visible and accessible to the designer. Modifications should take less effort. More complex tool features, as well as a scripting interface for batch processing, should be available for the advanced user but, at the same time, default tool settings should give less experienced designers a head start. In addition concise documentation is essential, not only on how to use the tool but also on how to write and fine-tune source code. It should also give the designer ability to influence the decisions taken during binding and allocation with no ambiguity.

2.4.3 Support for Arbitrary Data Types and Cores

In most common HLPs, available data types are usually the supported types of the instruction processor (int, float etc.) and aggregates thereof. In hardware, the only primitive data type is a bit. Support for complex data types by RTL synthesis tools is usually limited to integers. The

hardware designer has full freedom to determine ranges of these integers (i.e. the number of bits). A good HLS tool should allow the designer to make this choice of arbitrary data type lengths and let them validate this choice at the source code level. In addition support for additional data types such as fixed point or floating point is also an asset as it eases the transition from algorithm to RTL design.

In addition, it is also desirable that the tool provides a starting point to the designer, to utilize the commonly used IP cores and algorithms. There should be possibilities to choose a given functional unit among many for a given technology, say a specific type of DPM RAM core among many. It is also preferred that the designer has access to libraries that may aid algorithm development, for example signal processing, communication, image processing, control etc.

2.4.4 Design Space Exploration Capabilities

As stated in [3], HLS as a design methodology provides significant design exploration capabilities. It leverages on the idea that, *rather than making architecture decisions upfront and coding RTL by hand, HLS allows evaluation of a number of architectures to choose the best one with respect to the design specifications.* Since HLS tools differ significantly in the way they guide this process, there has to be clear capabilities that can lead to accelerated productivity in the increasingly complex designs. It should be possible to re-target the algorithm on different hardware technologies quite easily. Comparisons and analysis of different hardware solutions should be clear and easy to understand for a system designer. In this thesis, a tool that is able to achieve, maximal separation between the behavior (source code) and design constraints (architecture) was being sought after.

2.4.5 Design Verification

With the HLS methodology, clear design interfaces to the algorithm (Device Under Test (DUT)) can be built in a high level language. This interface serves to provide a robust frame work to build test cases for the algorithm. This way, the HLS tool can speed up verification by generating testbenches together with the designs. This solution type where the test vectors used to validate the source code can be re-used to validate the RTL Level code is one of the key features that the HLS tool should provide. As stated in [3], *the most advanced automation of functional verification can be achieved by integrating the source code (as a golden reference) and the generated design (as a device under test) into one testbench. In that case, the simulator can apply the same stimuli to both reference and DUT and report discrepancies.*

2.4.6 Correctness of the Tool

It is also important to determine whether the tool generates correct designs. In this thesis, focus is put on comparing the results of verification of an existing design with a generated design. A simple idea as stated in [3], shall be used; i*f the tool is correct, the design will be correct as well, and a generated design that passes verification also validates the correctness of the tool. In this way, the probability of errors in a generated design is lower than in handcrafted RTL code.*

2.4.7 Performance of the Generated Design:

It is of utmost importance that the generated design be evaluated in terms of latency, area, throughput and timing after placing and routing. To do this well, a reference handwritten, and well optimized RTL level design of a filter was used as a reference. In addition, the design chosen for this investigation is simplistic enough (i.e. not trivial but at the same time simple enough to be implemented with reasonable effort) so as to facilitate thorough investigation of the tools capabilities.

2.5 State of the Art Tools

This thesis work serves to investigate the Vivado HLS tool by Xilinx and the HDL tool chain by Mathworks. Other major players in the market today are listed below. The choice of the tool set for investigation (i.e. Vivado, HDL coder and Verifier) was heavily determined by the hardware designs at ABB HVDC. In addition, this thesis relies on previous research as documented in [3] [2]. For toolsets that were fundamentally found by previous research as lucking in development methodology as well as performance, a further investigation was not done and is not recommended.

Catapult C - Catapult C is owned by Calypto Design Systems, acquired from Mentor Graphics in August 2011. Catapult C accepts a large subset of C, C++ and SystemC. It offers a great deal of synthesis functionality as described in [3]. This tool maybe investigated further for design realization advantages, however research in [3], reveals an underperformance as compared to autopilot an earlier form of Vivado. It is therefore the recommendation of this thesis work that further investigation of this tool not be done.

Compaan - This tool is developed by Compaan design. It is designed for Xilinx FPGAs and works together with the Xilinx tool chain. As stated in [3], this tool can only generate the communication infrastructure but not processing elements. The design work at ABB encompasses both aspects and therefore this tool is not a suitable candidate for consideration.

C-to-Silicon - is a recent HLS tool from Cadence. It uses SystemC as its design language. For designs written in C, C++ or TLM 1.0, CtoS generates a SystemC wrapper. This tool is not recommended for further investigations because of the limitations detailed in [3].

CyberWorkBench (CWB) - is an HLS tool from NEC. C, SystemC and Behavioral Description Language (BDL) are supported. It has enormous advantages as detailed in [3]. More investigation to determine the tools performance with reference to HDL coder and Vivado HLS may be done, however research in [3], reveals an underperformance as compared to autopilot an earlier form of Vivado. It is therefore the recommendation of this research that further investigation of this tool not be done.

DK Design Suite - DK Design Suite is an older HLS tool that was acquired in 2009 by Mentor Graphics. Because of the limitations specified in [3], this tool is not recommended for further investigation

Synphony C Compiler - Synphony C Compiler is Synopsys' HLS tool, based on the former PICO tool which they acquired from Synfora in 2010. Design languages are ANSI C and C++. This tool may not be considered for further investigation since the results of earlier investigations as detailed in [3] [2], indicate an underperformance compared to both HDL coder and Autopilot, an earlier form of Vivado

ROCCC – is an open source HLS tool from the University of California at Riverside (UC Riverside). Commercial support is available from Jacquard Computing, Inc. ROCCC uses eclipse (www.eclipse.org) as its design environment. This tool is however not recommended for further investigation because of the limitations detailed in [3].

LabVIEW – is a development environment by National Instruments (NI) and integrates numerous tools that engineers and scientists need to build a wide range of applications in dramatically less time. The LabVIEW environment is designed for problem solving, accelerated productivity and continual innovation. LabVIEW supports platform independent RTL code generation. More so, the inbuilt methodology supports an all in one solution, from conceptualization to deployment. However since the environment is built for NI hardware additional DLLs have to be built to create a bridge between LabVIEW and custom hardware. Further investigation of this tool is highly recommended as there might be a likelihood for significant improvement in development time.

Other tools include Bluespec, Impulse CoDeveloper etc. This is by no means an exhaustive list of all the tools that are available on the market today, however it is a very good representation of the major players in the field of HLS today.

2.6 Filter Realization

As a general rule linear time-Invariant (LTI) systems can be classified into either finite impulse response (FIR) or infinite impulse response (IIR) depending on whether their operations have a finite or infinite response duration. Additionally depending on the application and hardware the filtering operation can be organized to operate either as a single block or as a sample by sample process. With block processing, the input signal is considered to be a block of many samples. Essentially the block is filtered by convolving it with the filter input and the output is also obtained as a block of samples. In cases where the input is very large, it can be broken down into multiple blocks, filtered and then the output blocks pieced together again. This can be implemented by ordinary convolution or fast convolution algorithms.

In the sample processing case the input samples are processed one at a time as they arrive at the input. In this scenario the filter operates like a state machine by utilizing the current sample together with current internal state of the filter to compute the current output sample. It also updates the current internal state in preparation for processing of the next sample. This thesis work expounds on the concepts of the sample by sample processing technique to develop an HLM for a filter used in an instrumentation application on the MACH 2 platform.

In general FIR filters have an impulse response $h(n)$ that extends over a finite duration interval say $0 \leq n \leq M$, and is identically equal to zero elsewhere i.e. $\{h_0, h_1, h_2, \ldots h_M, 0, 0, 0, 0, \ldots\}$. M is referred to as the order of the filter and the impulse response coefficients $[h_0, h_1, h_2]$ are referred to as filter coefficients. In general the filter equation for the FIR filters is given by Equation (2.1).

$$y(n) = \sum_{m=0}^{M} h(m)x(n-m) \tag{2.1}$$

Or, explicitly as

$$y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) + \cdots + h_Mx(n-M) \tag{2.2}$$

Thus the I/O equation is obtained as a weighed sum of the present input sample and the past M samples.

IIR filters on the other hand have the impulse response $h(n)$ that extends over an infinite duration defined over the infinite interval $0 \leq n \leq \infty$. In general the equation for IIR filters is given by

$$y(n) = \sum_{m=0}^{\infty} h(m)x(n-m) \tag{2.3}$$

Or, explicitly as

$$y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) + h_3x(n-3) + \cdots \tag{2.4}$$

This I/O equation is not computationally feasible since practical systems cannot deal with an infinite number of terms. Therefore, practical implementations normally restrict their attention to a subclass of IIR filters in which the infinite number of filter coefficients $\{h_0, h_1, h_2, \ldots\}$ are not chosen arbitrarily, but rather they are coupled to each other through constant coefficient linear difference equations. With this subclass of IIR filters, their I/O equation can be rearranged as a difference equation allowing the efficient recursive computation of the output $y(n)$.

Practical implementations are normally concerned with filters that have an impulse responses $h(n)$ that satisfy the constant-coefficient difference equations of general type:

$$h(n) = \sum_{i=1}^{M} a_i h(n-i) + \sum_{i=0}^{L} b_i \delta(n-i) \tag{2.5}$$

The convolution equation can be written as:

$$y(n) = \sum_{i=1}^{M} a_i y(n-i) + \sum_{i=0}^{L} b_i x(n-i) \tag{2.6}$$

Or, explicitly as

$$y_n = a_1y_{n-1} + a_2y_{n-2} + \cdots + a_My_{n-M} + b_0x_n + b_1x_{n-1} + \cdots + b_0x_{n-M} \tag{2.7}$$

And in general one can think of FIR filters as a special case of IIR filters whose recursive terms are absent i.e. $a_1, a_2, a_3, \ldots, a_M = 0$

This thesis work is concerned with IIR filters since the current design implementation is an IIR filter. The equivalent transfer function of the IIR filter can be represented as;

$$H(z) = Y(z)/X(z) = \frac{b_0+b_1Z^{-1}+b_2Z^{-2}+b_3Z^{-3}+\cdots+b_LZ^{-L}}{1+a_1Z^{-1}+a_2Z^{-2}+a_3Z^{-3}+\cdots+a_MZ^{-M}} \quad (2.8)$$

For a second order filter the general form of the transfer function is Equation (2.9).

$$H(z) = Y(z)/X(z) = \frac{b_0+b_1Z^{-1}+b_2Z^{-2}}{1+a_1Z^{-1}+a_2Z^{-2}} \quad (2.9)$$

In addition, filters can be realized in different ways such as:

1) Direct form
2) Canonical form
3) Cascade form

The IIR filter for with which this investigation is performed is a cascade of seven second order sections, realized in direct form II (canonical form). The design is targeted for the 45nm low power process technology FPGAs which are optimized for; cost, power, performance and most efficient utilization of low-power copper process technology. The filter is targeted for use in the voltage and current measuring IO-units in the MACH control system that performs digital filtering of signals after analog to digital conversion.

The system of equations (difference) for the second order section of the filter are shown in Equation (2.11).

$$d_0(n) = gain * x(n) - a_1d_1(n) - a_2d_2(n)$$

$$y(n) = b_0d_0(n) + b_1d_1(n) + b_2d_2(n)$$

$$d_2(n+1) = d_1(n)$$

$$d_1(n+1) = d_0(n) \quad (2.11)$$

The cascade realization form of a general second order function assumes that the transfer function is the product of such second order transfer functions as shown in Equation (2.10). In general any transfer function that can be expressed in the form of Equation (2.8) can be factored into second order filters with real valued coefficients.

$$H(z) = \prod_{i=0}^{k-1} H_i(z) = \prod_{i=0}^{k-1} \frac{b_{0i}+b_{1i}Z^{-1}+b_{2i}Z^{-2}}{1+a_{1i}Z^{-1}+a_{2i}Z^{-2}} \quad (2.12)$$

The current design is of order 14 and therefore, since the cascade is of second order sections then $k = 7$. It is important to note that the difference equation in Equation (2.11) is an equivalent form of the transfer function in Equation (2.9).

$$H(z) = \prod_{i=0}^{6} H_i(z) = \prod_{i=0}^{6} \frac{b_0 + b_{1i}Z^{-1} + b_{2i}Z^{-2}}{1 + a_{1i}Z^{-1} + a_{2i}Z^{-2}} \quad (2.13)$$

The design upon which the first evaluation of the tools is made is the second order section of the filter as shown in the diagram in figure 1. This filter is an anti-aliasing filter that's placed between the DSP and the ADC. The ADCs sample at approximately 500 kHz whereas the filter outputs to the DSP run at a configurable rates depending on the measuring board. Typical values could be 50, 100 or 125 kHz. This is the fundamental reason why the anti-aliasing filter is needed between the DSP and the ADCs. The design of the filter is such that all the coefficients and internal states (delay registers), are stored in RAM. The coefficients and decimation parameter of the filter (g_0, a_1, a_2, b_0, b_1, b_2, d_s) are configurable externally by the DSP and it is therefore possible to change the characteristics of the filter depending on which measuring board the communication board is piggybacked on. The filter block performs data filtering (anti-aliasing) as well as storage of filtered values and internal states into the DRAM.

The current implementation of the design utilizes one multiplier, one adder and one subtractor in a pipelined fashion in order to save the FPGA resources. It is the goal of this investigation to be able to achieve these resource restrictions while satisfying the timing of 100MHZ.

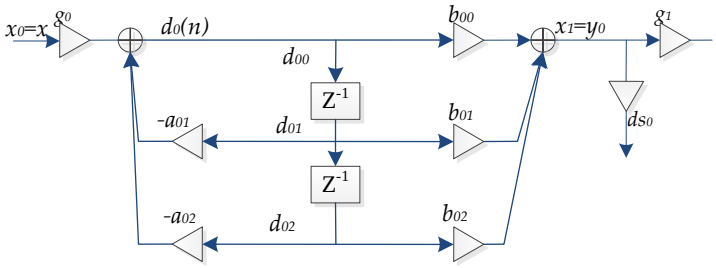


Figure 2.1. Second Order Section (SOS) of the IIR filter

Chapter 3. Vivado High Level Synthesis

3.1 Introduction

Vivado High Level Synthesis is Xilinx's HLS tool for transforming a C, C++ or SystemC specification into an RTL implementation which can then be packaged as an Intellectual Property (IP) core or exported as RTL source code for synthesis. This adds an extra layer of abstraction above the traditional RTL coding approach. The fundamental reason why the FPGA community has moved from one abstraction level to the next is to manage the complexity of the designs. The move is such that each added abstraction layer hides some complexity of the corresponding implementation step.

The RTL description captures the desired functionality by defining datapath and logic between boundaries of registers. RTL synthesis creates a netlist of Boolean functions to implement the design. The focus of the RTL abstraction layer is to define a functional model for the hardware [4]. A functional specification would therefore remove the need to define the register boundaries in order to implement a desired algorithm. The designer's goal is now focused on only specifying the desired functionality.

In Vivado HLS, moving up the design hierarchy to use the functional specification for creating RTL descriptions provides productivity in both verification and design optimization. As discussed in detail in [4], this move significantly creates the following benefits,

- Acceleration in simulation time by using a functional C language-based specification and the resultant earlier detection of design errors.
- High-Level Synthesis shortens the manual RTL creation process and avoids translation errors by automating the creation of RTL from a functional specification.
- High-Level Synthesis automates RTL architecture optimization, allowing multiple architectures to quickly be evaluated before committing to an optimum solution.

C based entry is the most popular mechanism to create functional specifications and Vivado HLS currently supports to a synthesizable level all three C input standards(C, C++ and SystemC). This enables it to simulate C code with minimal modifications.

As described in [4], the Vivado HLS tool performs two distinct kinds of synthesis on the design;

- Algorithm synthesis takes the content of the functions and synthesizes the functional statements into RTL statements over several clock cycles. This type of synthesis typically builds the algorithm and is significantly affected by the interface level synthesis described below.
- Interface synthesis transforms the function arguments (or parameters) into RTL ports with specific timing protocols, allowing the design to communicate with other designs in the system. It's worth repeating in this report [4] that interface synthesis can be performed on global variables, top level function arguments and the return values of

the top level functions. The types of interfaces that can be synthesized are wire, Register one way and two way handshakes, Bus, FIFO and RAM. In addition, functional level protocols that dictate when a function can start or end can be synthesized.

3.2 Optimizations in Vivado HLS

In order to generate optimized RTL code, Vivado HLS uses a number of optimizations such as dataflow pipelining, function pipelining, resource limitation, loop unrolling, memory partition etc. These optimizations are specified as directives using tcl scripts, or can be embedded in the source code. An in-depth explanation of the optimizations employed by Vivado HLS can be found in [4], however it suffices at this moment to make mention of the most important optimizations that have been used in this design. These include:

1) Dataflow pipelining
2) Function pipelining
3) Latency optimizations
4) Loop rolling and unrolling
5) Loop pipelining
6) Iteration Interval reduction or increase
7) Array mapping to RAMs
8) Resource limitation or sharing

The same description of these optimizations is used for the HDL Coder tool where applicable.

3.2.1 Pipelining in Vivado

Pipelining in Vivado HLS can be applied as an optimization between functions or the operations within a function. It can also be applied to the operations inside a loop or between loops. In each case, this optimization increases throughput, by ensuring that the function, loop or operation is not required to wait until a previous function, loop or operation has completed all its operations before it can begin. When the pipelining is applied as an optimization between functions, it's referred to as dataflow pipelining. Dataflow pipelining transforms a sequential implementation of functions into a parallel architecture as shown in the figures 3.1 and 3.2.

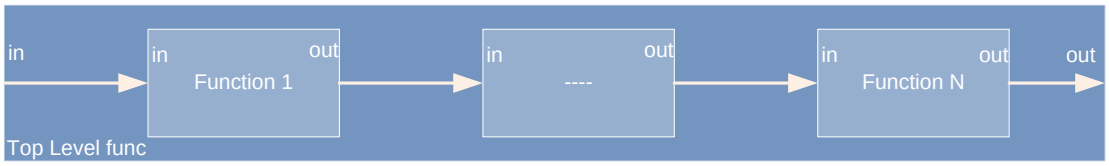


Figure 3.1 Sequential function implementation (adopted from [4])

Figure 3.2 Parallel process architecture after dataflow pipelining (adopted from [4])

The inner workings of this optimizations can be visualized in the figure 3.3 below;

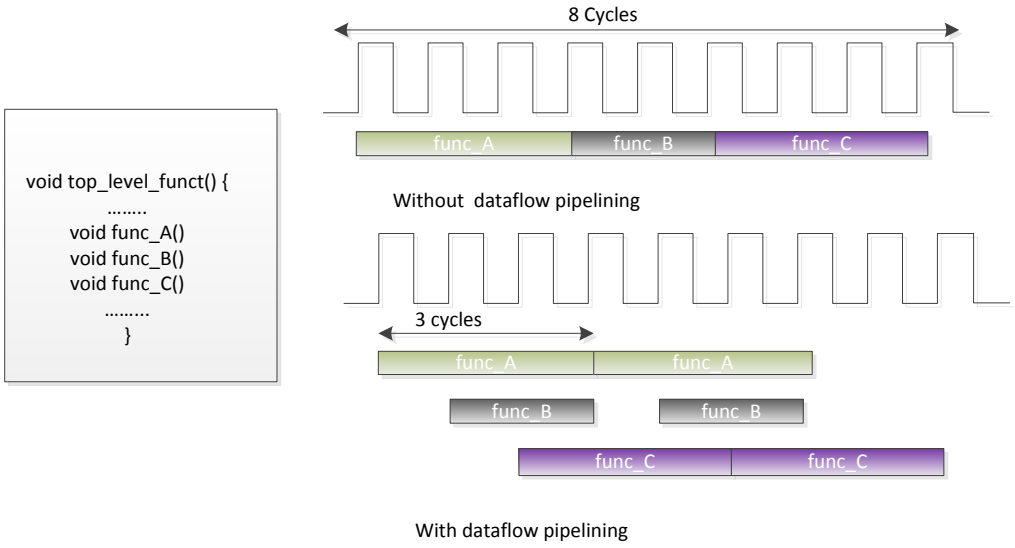


Figure 3.3 Illustration of data flow pipelining (adopted from [4])

At function level, function pipelining optimizes the operations within the function such that operations with no resource contention or data dependency execute concurrently as shown in figure 3.4. As detailed in [4], operations or functions which exhibit dependencies can be pipelined by increasing the initiation interval. The initiation interval is the number of cycles between the input reads.

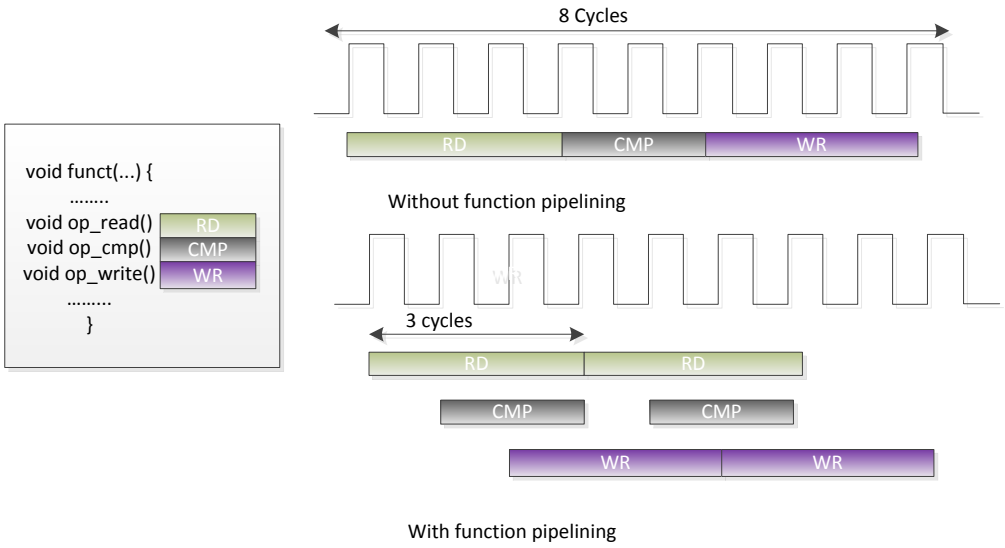


Figure 3.4 Illustration of function pipelining (adopted from [4])

Similarly, when pipelining is applied as an optimization between loops, it's referred to as loop dataflow pipelining as shown in figure 3.5. It is important to note as in [4], that data flow pipelining can only be applied to a region that contains only loops or functions.

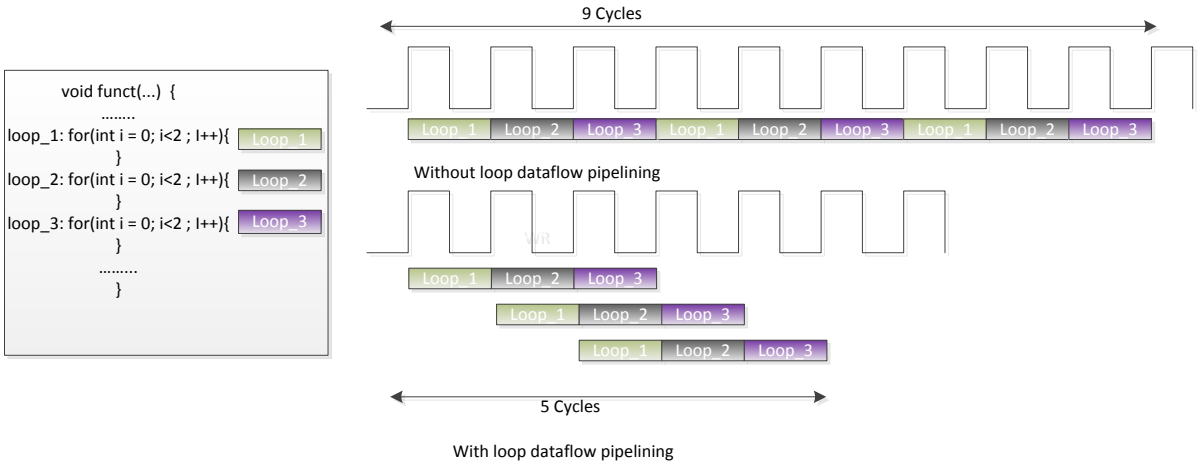


Figure 3.5 Illustration of loop dataflow pipelining

Loop operations are generally executed sequentially, this behavior translates to a sequential RTL architecture. However, as concurrency is always desired in RTL implementations to improve throughput, loop pipelining, as shown in figure 3.6, can go a long way in ensuring that the design objectives of throughput are met in Vivado HLS. Data dependencies in loop operations also limit the extent of pipelining and can be harder to manage as explained in details [4].

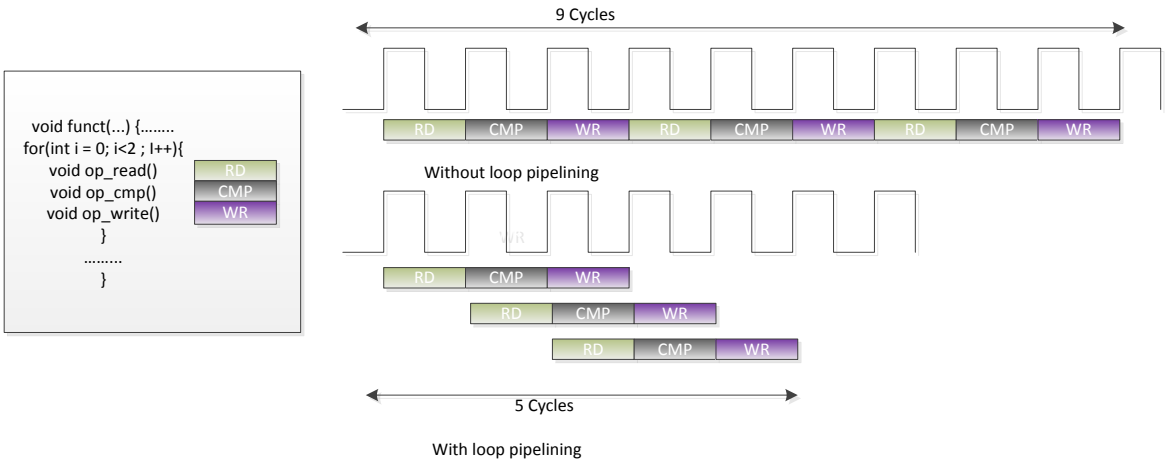


Figure 3.6 Illustration of loop pipelining

3.2.2 Array Mapping to RAMs

As described in [4], the default behavior of Vivado HLS is to map arrays onto RAMs. Typical optimizations for Vivado HLS are shown in the table below; more information on the use of these optimizations can be found in the user manual.

Table 3.1 Array mapping optimizations (Adopted from [4]).

Directive	Description
Resource	Specify which hardware resource (RAM component) an array maps to.
Array_Map	Reconfigures array dimensions by combining multiple smaller arrays into a single large array to help reduce RAM resources and area.
Array_Partition	Control how large arrays are partitioned into multiple smaller arrays to reduce RAM access bottleneck. Also used to ensure arrays are implemented as registers and not RAMs.
Array_Reshape	Can reshape an array from one with many elements to one with greater word-width. Useful for improving RAM accesses without using many RAMs.
Stream	Specifies that an array should be implemented as a FIFOrather than RAM

3.2.3 Loop Optimizations in Vivado

As described in [4], the default behavior of Vivado HLS is to keep all loops rolled. All unrolled loops create at least one state in the design FSM. When there are multiple sequential loops this can create additional unnecessary cycles [4]. Loop merging goes a long way to eliminate such additional cycles. Additionally, it requires extra cycles to move into rolled nested loops. Loop flattening can go a long way in addressing this particular problem. Dataflow pipelining and loop pipelining have been discussed previously, and these optimization greatly improve the throughput of the design. More details on other directives can be found in [4], however the summary below in table form gives the optimizations supported by Vivado.

Table 3.2 Loop optimization directives (adopted from [4]).

Directive	Description

Unrolling	Unroll for-loops to create multiple independent operations rather than a single collection of operations.
Merging	Merge consecutive loops to reduce overall latency, increase sharing and optimization.
Flattening	Allows nested loops to be collapsed into a single loop with improved latency and logic optimizations
Dataflow	Allows sequential loops to operate concurrently
Pipelining	Used to improve initiation interval by performing concurrent operations
Dependence	Used to provide additional information that can be used to overcome loop-carry dependencies.
Tripcount	Provides user override of iteration analysis
Latency	Specify a cycle latency for the loop operation

The above brief discussion on loop optimizations just gives anecdotes of how the listed optimizations can be beneficial to a designer once used appropriately. It is important to note that this list is in no way an exhaustive list of all the optimizations supported by Vivado HLS. The list in this report simply serves to list the most important optimizations used in this particular design. In addition, these optimizations may in a similar way apply to HDL coder. The reader of this report is therefore advised to cross-check for a complete list of all the available optimizations in the respective tools.

3.3 Control and Data path Extraction

At a functional level, extraction of flow control information is the first thing that the Vivado HLS tool does. An example of loop control behavior is given in [4] and clearly illustrates how the tool is able to extract Finite State Machine (FSM) behavior from a Finite Impulse Response (FIR) filter function.

Vivado HLS matches the behavior of the gcc compiler [4]. Based on this it is reasonable to say that the source code is parsed and the corresponding Abstract Syntax Tree (AST) is generated. Then, a Static Single Assignment (SSA) form is generated where significant optimizations of the source code can be performed. The source code is then be converted back into a tree such the scheduling and binding processes can be executed to generate the RTL code.

It is important to note that the designer needs to follow the syntax rules established by Xilinx regarding, function declaration, variable assignment and declaration, pointer handling and class

operations in C++ otherwise the RTL code implementation shall not be correct. The compiler may optimize away some of the important characteristics of the source code. More details of the supported behavior can be found in the Vivado HLS user manual [4].

3.4 C/C++ Code Generation from MATLAB and Simulink for use in Vivado

The Mathworks C/C++ coder provides a very fast way of generating code. Since the Mathworks tool chain features a large toolset for developing algorithms, it is very desirable that the C/C++ code be generated directly from a design model and directly into Vivado HLS for synthesis and implementation of design cores. This flow however presents challenges;

- The code obtained has to be edited to support the syntax rules implemented by Vivado HLS.
- The code obtained may significantly be hard to understand or edit. It may also be the case that the resulting design is relatively complex compared to an equivalent handwritten implementation, as was the case for this design. If this is the case, it would further complicate the application of optimization directives to the code.

However it is reasonable from the design view point to say that this is a very attractive design flow which may reduce the development time significantly. The position that this thesis work affords is that the designer has to ultimately make the decision as to whether;

- The complexity resulting from the generated code presents a lesser challenge compared to the alternative of writing the code.
- The work required in editing the data types from MATLAB /Simulink presents a time gain in design implementation

In this particular design, the time gain that would result from utilizing the generated code was significantly less. This is partly because the DUT was a very simple model of a second order filter. This particular solution could however be very attractive for larger designs.

3.5 C/C++ Simulation

The Vivado HLS tool supports C/C++ design simulation as is detailed in [4]. This thesis leverages on this documentation, designer experience and the tests carried out to conclude that the tool gives the designer enough functionality and ease to perform good model level simulations.

3.6 Co-Simulation

The Vivado HLS tool supports Co-simulation with ModelSim, ISIM, XSIM and Riviera. It also supports verification of the generated design against deviation from the C/C++ or SystemC model by performing post synthesis verification as is detailed [4]. This research leverages on this documentation, designer experience and the tests carried out to conclude that the tool gives the designer enough functionality and ease to perform good co-simulations.

3.7 Scheduling and Binding in Vivado HLS

Scheduling and binding are the processes at the heart of Vivado High-Level Synthesis as previously established. During the scheduling process Vivado HLS determines which cycle operations should occur. The decisions made during scheduling take into account, the clock frequency and clock uncertainty, timing information from the device technology library, as well as optimization directives.

Binding is the process that determines which hardware resource, or core, is used for each scheduled operation for example, High-Level Synthesis automatically determines if an adder and subtractor will be used or if a single adder-subtractor will be used for both operations. Because the decisions in the binding process can influence the scheduling of operations, for example, using a pipelined multiplier instead of a standard combinational multiplier, binding decisions are considered during scheduling.

3.8 Area Optimization in Vivado HLS

The design objective of this exercise in terms of area optimization is to be able to utilize one multiplier, one adder and one subtractor in a pipelined fashion in order to save the FPGA resources. This measure comes as a design specification set in the current solution to minimize resource usage on the FPGA. It suffices to say therefore that the tool chosen for use ultimately should be able to meet at least these basic requirements for this simplistic design.

Vivado HLS comes with a set of directives that a designer can use to control the operators, resources, the binding process and the binding effort level [4]. The first activity is to limit the number operators used in the design. In order to do this Vivado uses the command

```
set_directive_allocation [OPTIONS] <location> <instances>
```

This command specifies the instance restrictions for resource allocation. It defines, and can limit, the number of RTL instances that are used to implement specific functions or operations. For example, if the C/C++ source file has five instances of a function "mult", the set_directive_allocation command can be used to ensure there is only one instance of "mult" in the final RTL. What this means in effect is that all the four instances will be implemented using the same RTL block. More details of the command can be found in the Xilinx man pages for HLS. The set_directive_allocation command can either be embedded in the source code as #pragma or placed in the directives tcl file.

Both options are configurable both using the HLS command line interface or the HLS GUI. In this exercise, the first option is preferred so that the directives are carried over across different solutions.

The tables 3.3 to 3.4 show the amount resource utilization after RTL code generation by the Vivado HLS tool.

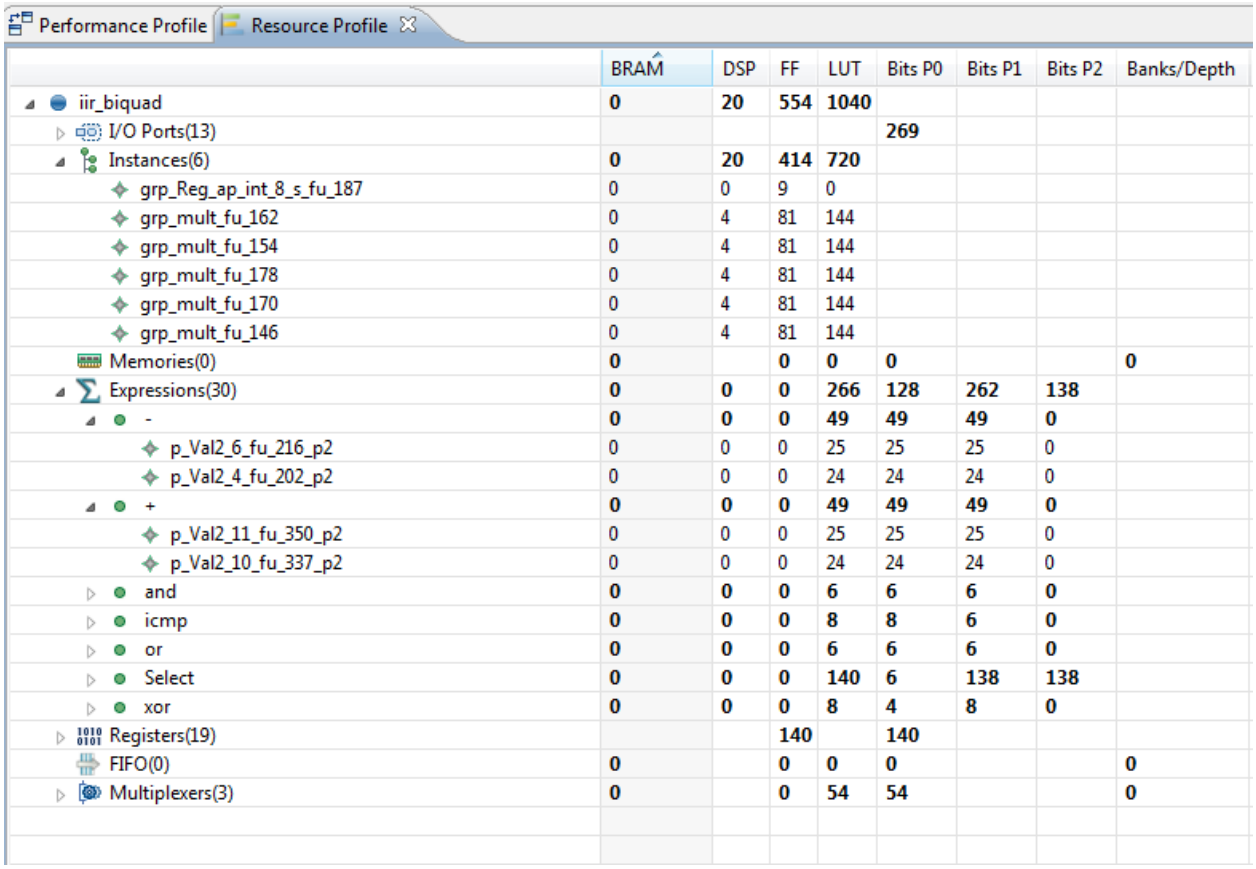
Table 3.3 Instance details in the design

Instance	Module	BRAM_18K	DSP48A	FF	LUT
grp_Reg_ap_int_8_s_fu_187	Reg_ap_int_8_s	0	0	9	0
grp_mult_fu_146	mult	0	4	81	144
grp_mult_fu_154	mult	0	4	81	144
grp_mult_fu_162	mult	0	4	81	144
grp_mult_fu_170	mult	0	4	81	144
grp_mult_fu_178	mult	0	4	81	144
Total	6	0	20	414	720

Table 3.4 Utilization estimates of the design

Name	BRAM_18K	DSP48A	FF	LUT
Expression	-	-	0	266
FIFO	-	-	-	-
Instance	-	20	414	720
Memory	-	-	-	-
Multiplexer	-	-	-	54
Register	-	-	140	-
ShiftMemory	-	-	-	-
Total	0	20	554	1040
Available	116	58	54576	27288
Utilization (%)	0	34	1	3

An analysis view of the tool is also provided in the capture from Vivado HLS shown in figure 3.7.



Performance Profile | Resource Profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth
iir_biquad	**0**	**20**	**554**	**1040**				
I/O Ports(13)					**269**			
Instances(6)	**0**	**20**	**414**	**720**				
grp_Reg_ap_int_8_s_fu_187	0	0	9	0				
grp_mult_fu_162	0	4	81	144				
grp_mult_fu_154	0	4	81	144				
grp_mult_fu_178	0	4	81	144				
grp_mult_fu_170	0	4	81	144				
grp_mult_fu_146	0	4	81	144				
Memories(0)	**0**		**0**	**0**	**0**			**0**
Expressions(30)	**0**	**0**	**0**	**266**	**128**	**262**	**138**	
-	**0**	**0**	**0**	**49**	**49**	**49**	**0**	
p_Val2_6_fu_216_p2	0	0	0	25	25	25	0	
p_Val2_4_fu_202_p2	0	0	0	24	24	24	0	
+	**0**	**0**	**0**	**49**	**49**	**49**	**0**	
p_Val2_11_fu_350_p2	0	0	0	25	25	25	0	
p_Val2_10_fu_337_p2	0	0	0	24	24	24	0	
and	**0**	**0**	**0**	**6**	**6**	**6**	**0**	
icmp	**0**	**0**	**0**	**8**	**8**	**6**	**0**	
or	**0**	**0**	**0**	**6**	**6**	**6**	**0**	
Select	**0**	**0**	**0**	**140**	**6**	**138**	**138**	
xor	**0**	**0**	**0**	**8**	**4**	**8**	**0**	
Registers(19)			**140**		**140**			
FIFO(0)	**0**		**0**	**0**	**0**			**0**
Multiplexers(3)	**0**		**0**	**54**	**54**			**0**

Figure 3.7 Resource profile of the design

The figure 3.7 and the tables 3.3 - 3.4 show that the number of DSP48A used is very high, approximately 34% of the entire available DSP48A resources available on chip, which is not acceptable in this design. One can deduce from looking at the function instances that the increased number of DSP48A usage is due to the number of multiplication instances. There are approximately 5 multiplication instances and each uses 4 DSP48A. The objective at hand therefore is to reduce this number to just 4 instances which correspond to only one multiplier implementation in the design. By setting the allocation directive to limit the number of the function instances to 1, the instances are automatically reduced by Vivado HLS

The following #pragma directive was added to the C++ source file

```
#pragma HLS ALLOCATION instances=mult limit=1 function
```

In effect the results in tables 3.5 - 3.6 results are obtained. A capture from Vivado's resource viewer is also shown in figure 3.8.

Table 3.5 Instances of the design after function instance optimization

Instance	Module	BRAM_18K	DSP48A	FF	LUT
grp_Reg_ap_int_8_s_fu_201	Reg_ap_int_8_s	0	0	9	0
grp_mult_fu_153	mult	0	4	150	144
Total	2	0	4	159	144

Table 3.6 Utilization estimates after function optimization

Name	BRAM_18K	DSP48A	FF	LUT
Expression	-	-	0	266
FIFO	-	-	-	-
Instance	-	4	159	144
Memory	-	-	-	-
Multiplexer	-	-	-	54
Register	-	-	160	-
ShiftMemory	-	-	-	-
Total	0	4	319	464
Available	116	58	54576	27288
Utilization (%)	0	6	~0	1

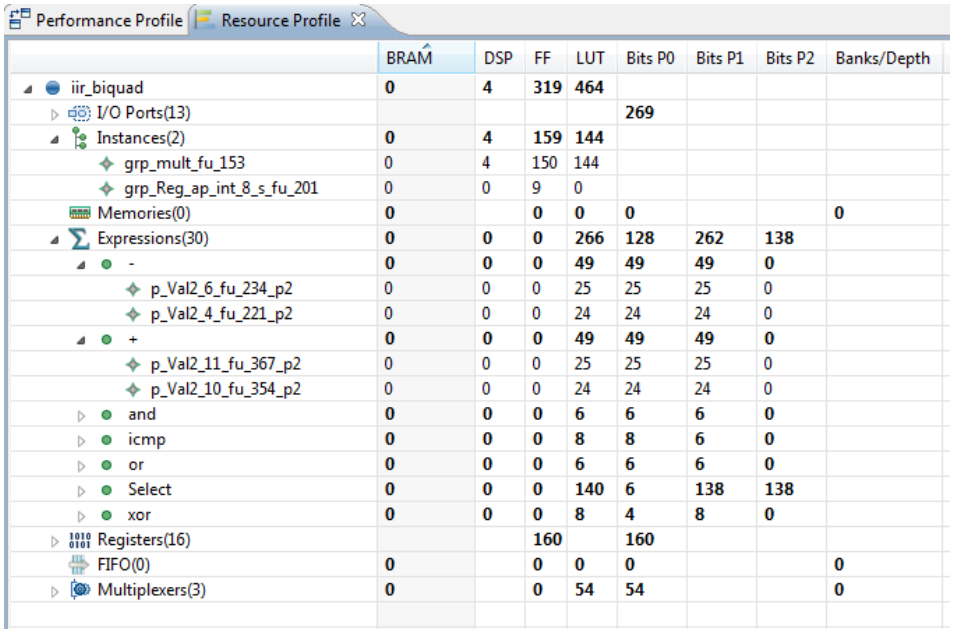


Figure 3.8 Resource profile after function instance optimization

From the tables 3.5 - 3.6 and the figure 3.8 above, Vivado HLS tool has been able to reduce the number of multipliers to only one, which corresponds to the grp_mult_fu_153 function instance. In addition a considerable reduction in the amount of resources has been achieved. The number of DSP48As has been reduced to just 4, which was the target objective. In addition, a considerable reduction of the number of flip flops (FF) and Look Up Tables (LUTS) is observed. The LUTS are reduced from a total of 720 to just 144 and the FF are reduced from a total of 414 to just 159, with just a command in the design. The total of DSP48A usage has reduced to just 6%. The target objective has been achieved, with no significant strain and effort on the part of the designer. In addition one can reasonably deduce from this activity that various configurations of instances can be chosen and implemented on the fly using such commands and Vivado HLS will try to schedule and bind these operations as requested.

Further analysis of the design reveals two addition and subtraction operations. From the objective specification, the aim is to reduce these operations to just one of each. Using the similar directives, Vivado HLS is instructed to reduce these operations as follows;

```
#pragma HLS ALLOCATION instances=sub limit=1 operation
#pragma HLS ALLOCATION instances=add limit=1 operation
#pragma HLS ALLOCATION instances=mul limit=1 operation
```

The resulting operation reduction is shown in the tables 3.7 and 3.8.

Table 3.7 Operations before optimization

Variable Name	Operation	DSP48A	FF	LUT	Bitwidth P0	Bitwidth P1
p_Val2_10_fu_337_p2	+	0	0	24	24	24
p_Val2_11_fu_350_p2	+	0	0	25	25	25
p_Val2_4_fu_202_p2	-	0	0	24	24	24
p_Val2_6_fu_216_p2	-	0	0	25	25	25

Table 3.8 Operations after optimization

Variable Name	Operation	DSP48A	FF	LUT	Bitwidth P0	Bitwidth P1
grp_fu_159_p2	+	0	0	25	25	25
grp_fu_173_p2	-	0	0	25	25	25

As can be seen from the tables 3.7 and 3.8, the operations have been successfully reduced to just one add and one subtraction operation. This finally achieves the area optimizations on the design since the objective was to ensure that only one of multiplier, adder and subtractor were to be used for the second order section of the IIR filter. It also suffices to say at this point that the design objectives have been achieved with no significant constraint. The tool identifies with a language of directives through which the user can, with fine granularity, be able to control what the scheduler and binder should do. In this particular scenario, by directing the number of operations, one is able to control the process of scheduling and allocation. These two processes are the core of HLS and it is desirable that the position of the designer should be as dynamic as possible as earlier on established. For more details on the area utilization for this design, see Appendix A2.

3.9 Latency and Throughput Optimization in Vivado HLS

The design progress has so far not given any attention to throughput and latency. As can be seen in the tables 3.9 and 3.10, the design has a latency of 61 clock cycles and an iteration interval of 62, which essentially means that the design has been synthesized as a sequential function description. The same is the case for the function instance in table 3.10. In addition to a long latency, the Iteration Interval (II) is also long. The design objective is to be able to achieve at least 19 clock cycles of Latency and an iteration interval of at least 6 to maximize

throughput. As earlier stated, this measure is a design specification set in the current solution to minimize resource usage on the FPGA.

Table 3.9 Latency and Iteration Interval (II) before optimization

Latency		Interval		Type
min	**max**	**min**	**max**	
61	61	62	62	none

Table 3.10 Instance latencies and iteration intervals before optimizations

Instance	Module	Latency		Interval		Type
		min	**max**	**min**	**max**	
grp_mult_fu_186	mult	9	9	9	9	none
grp_Reg_ap_int_8_s_fu_234	Reg_ap_int_8_s	2	2	2	2	none

Conventional methods for design throughput maximization revolve around making a parallel architecture of the design. Since the implemented designed (deductively) is serial, the first objective is to turn the design into a parallel architecture. This can be achieved using pipelining.

Vivado HLS tool supports the use of latency constraints on a function. When a maximum and/or minimum constraint is placed on the function, High-Level Synthesis tries to ensure all operations in the function complete within the range of clock cycles specified [4]. More details on latency optimizations are given in [4], however it's important that the behavior of the tool is noted also in this report as well. The next four paragraphs explain how Vivado HLS behaves if it is unable to meet latency constraints;

If the reason for failing to meet latency is due to a timing violation (the logic delay exceeds the cycle time and hence more clock cycles are required) Vivado HLS allows local timing violations and adhere to the constraint. The extent of the allowable local timing violations is controlled automatically by Vivado HLS and there is no user control.

The optimization strategy here is to meet the latency constraint and see if the timing violation can be corrected during RTL logic synthesis. If the timing violation is too great to be met using logic synthesis, review the techniques in the "Logic Structure Optimizations" in [4]

If Vivado HLS cannot meet the latency constraint due to data dependencies or if there are multiple timing violations, or if the timing violations are large, Vivado HLS automatically relaxes the latency constraint and tries to achieve the best possible result. If a minimum latency constraint is set and Vivado HLS can produce a design with a lower latency than the minimum required it inserts dummy clock cycles to meet the minimum latency.

In the filter design, function pipelining was turned on for both functions. In addition the latency constraint was set to a maximum of 25, which at most should allow for an iteration interval of 6 clock cycles. The directives specified were;

```
#pragma HLS PIPELINE
```

```
#pragma HLS LATENCY
```

The rationale behind not specifying latency figures is to let the tool minimize the latency to the best achievable value for this design. Similar reasoning applies to the iteration interval for pipelining. The results for the latency and throughput optimizations are shown in tables 3.11 and 3.12.

Table 3.11 Top level function cycle values after latency and throughput optimizations

Latency		Interval		Type
min	**max**	**min**	**max**	
19	19	6	6	function

Table 3.12 Function instances after latency and throughput optimizations

Instance	Module	Latency		Interval		Type
		min	**max**	**min**	**max**	
grp_mult_fu_206	mult	7	7	1	1	function
grp_Reg_ap_int_8_s_fu_252	Reg_ap_int_8_s	1	1	1	1	function

With these constraints, Vivado HLS was able to achieve exactly the required performance specification. It's important to note however that a better latency of 18 can be achieved but at the expense of the iteration interval which effectively reduces the throughput of the design. The resulting Latency tables for this design configuration are shown in the tables 3.13 and 3.14 and the resulting delay for the consecutive iterations increases to 18+11=29 as compared to 19+6=25.

Table 3.13 Top level function performance with suboptimal optimization

Latency		Interval		Type
min	**max**	**min**	**max**	
18	18	11	11	function

Table 3.14 Function instance performance with suboptimal optimization

Instance	Module	Latency		Interval		Type
		min	**max**	**min**	**max**	
grp_mult_fu_206	mult	7	7	1	1	function
grp_Reg_ap_int_8_s_fu_254	Reg_ap_int_8_s	1	1	1	1	function

3.10 Timing Optimization in Vivado HLS

Vivado HLS allows the designer to specify a clock upon which the timing constraints are based. Multiple clocks can be specified for a particular solution. Only a single clock is supported for C and C++ designs. Vivado HLS uses the concept of clock uncertainty since it can only estimate the timing of operations in the design but cannot know the final component placement

and net routing: as such, it cannot know the exact delays. Using the clock uncertainty, it is able to deduce the usable clock period as detailed in [4].

More information about achieving timing is given in [4], however these important points as explained therein will be highlighted once more in this report

The timing information used for the RTL operators and registers is defined by the library. The libraries are all pre-characterized and stored within High-Level Synthesis. High-Level Synthesis will always aim to meet latency, throughput (initiation interval) and the timing constraints. However, even when High-Level Synthesis cannot meet constraints, it will always output an RTL design.

- If High-Level Synthesis cannot meet a throughput constraint due to a data dependency (for example, if a throughput of one is required but it requires two cycles to read a value from memory) it will automatically improve the initiation interval until a design can be realized.

- If the timing constraints inferred by the clock period cannot be met High-Level Synthesis will issue message, "@W[SCHED-644]..", and output a design with the best achievable performance

In this design the clock constraint was set 100MHZ. See Appendix A1 for a detailed timing analysis from the synthesis tools. The design was able to achieve a minimum period of 8.688ns (Maximum frequency: 115.101MHz). This design therefore meets the timing constraints.

3.11 Performance Comparison with Current solution

3.11.1 Area Optimization

Table 3.15 below shows a comparison between the generated design and the hand coded design.

Table 3.15 Showing resource usage statics for the current design and the generated design

Resource	Current Design	Generated Design
Slices	150	160
LUTS	277	398
FF	486	417
DSP	4	4
BRAM	0	0
SRL	18	19

Overall the Vivado HLS tool achieves almost the same resource utilization as the handwritten code. In the table above, there is; a rise in the used slices by 10, a raise in the used LUTS by 121, a reduction in the number of FF by 69 and a raise of SRLs by 1. In percentages of available resources on the device, as calculated by ISE, the device utilization of the two designs is the same. In conclusion, it is reasonable to say that one can achieve very good area optimization with appropriate directives in the C/C++ design.

3.11.2 Timing Optimizations

Both designs exceed the timing requirement by a very good margin, however the current handwritten design achieves a higher operation frequency of 129.467MHZ as compared to 114.338MHZ in the generated design. It is however important to note that the timing requirements of the generated code can be altered as quickly as the design can be regenerated which is comparably difficult when it comes to adjusting timing requirements for handwritten RTL designs in general. For more details about the timing results see, Appendix A3.

3.11.3 Latency and Throughput Optimizations

Overall the latency and the iteration interval for the current design and the generated design are equal, however the value for the delay register zero of the filter gets ready after 14 clock cycles in the generated design as compared to 8 clock cycles in the handwritten RTL code. This value cannot be improved since latency and throughput constraints cannot be specified for this single output value. The tables below illustrate the Latency and throughput values.

Table 3.16 Latency and iteration interval values for the current design

Latency		Interval	
min	max	min	max
19	19	6	6

Table 3.17 Latency and iteration interval values for the generated Design

Latency		Interval	
min	max	min	max
19	19	6	6

3.12 Overall Comparison with Current solution

The filter values for the generated design agree with the values from the current design. The cosimulation feature was used as well as a hand coded test bench.

3.13 Design Flow evaluation

The Vivado HLS tool Starts from generic C/C++ or systemC code, it converts each datatype to arbitrary-precision datatypes. The algorithm and interface synthesis are then performed on the design. It essentially explores the architecture of the design, builds the actual implementation and then implements interfacing details. Vivado HLS extracts the control and datapath behavior of the DUT; the control behavior is located in loops and conditions which are mapped to a Finite State Machine (FSM). Data path evaluation is achieved by loop unrolling and evaluation of conditions. Based on user constraints e.g. for latency and resource usage, scheduling and binding are performed to generate RTL code.

Vivado HLS is significantly easy to learn and use. It offers three perspective to the developer. A debug perspective where the C/C++ design can be checked for correctness. This perspective greatly supports the designer in producing correct models for synthesis.

The synthesis perspective is ideal during the process of performing C simulations, Co-simulations, Synthesis and design implementation. In this perspective a designer can in addition to performing the fore mentioned tasks, be able to view reports, add various solutions featuring various design configurations, compare different solutions and many other simplistic tasks as detailed in [4]. In addition, the tool also features easy to use manual pages for the HLS directives that can be accessed in all the perspectives.

The analysis perspective offers a feature rich view to analyze designs. It offers a resolution up to clock level, for operation and core resource examination. It offers design tracking and coverage right from the input C/C++ or SystemC code to the generated RTL code (VHDL and Verilog). This way a designer can cross-reference the input model with the generated code. In addition, it provides detailed resource and performance profiles, hierarchical diagrams, function and module hierarchies etc., more details of the extent of the support can be found in the user manual. The tool therefore encapsulates a feature rich designing and analysis perspective. It also fully supports code cross referencing, right from the input model to the generated RTL code.

The generation procedure can be approached either using the GUI or command line interface with the help of tcl commands. It is therefore possible for the designer to script the commonly used directives and optimizations for future applications on similar designs, should he/she wish. In addition the tool provides great support for most RTL code tasks like mapping to RAMs, pipelining, FIFOs etc. More details of the extent of support can be obtained from the user manual.

To enable concurrency, Vivado HLS provides automatic pipelining for functions and loops. Design exploration is also possible using a variety of FPGAs, a number of solutions, different clocks, various constraints etc. After RTL code generation, a design report is generated which gives information about the estimated clock period, the latency (best/average and worst-case) and a resource estimation in a very summarized and easy to interpret format.

For simulation and verification, Vivado HLS allows the reuse of the C-code testbench by automatically generating a wrapper that enables communication between the generated RTL and the C testbench. Reusing the C testbench for RTL simulation significantly reduces the verification time. Vivado HLS also provides a library of classes for conversions between floating Point and fixed point numbers using various techniques of rounding. More details of the techniques supported can be found in the users guide. Essentially as detailed in [11], by Supporting both the ISE® and Vivado design environments Vivado HLS provides system and design architects alike, with a faster path to IP creation by:

- *Abstraction of algorithmic description, data type specification (integer, fixed-point or floating-point) and interfaces (FIFO, AXI4, AXI4-Lite, AXI4-Stream)*
- *Directives driven architecture-aware synthesis that delivers the best possible QoR*
- *Fast time to QoR that rivals hand-coded RTL*
- *Accelerated verification using C/C++ test bench simulation, automatic VHDL or Verilog simulation and testbench generation*

- *Multi-language support and the broadest language coverage in the industry*
- *Automatic use of Xilinx on-chip memories, DSP elements and floating-point library*

Vivado HLS however generates complex variable names which makes the generated RTL level code hard to read and understand easily. In summary the model employed by Vivado HLS is shown below

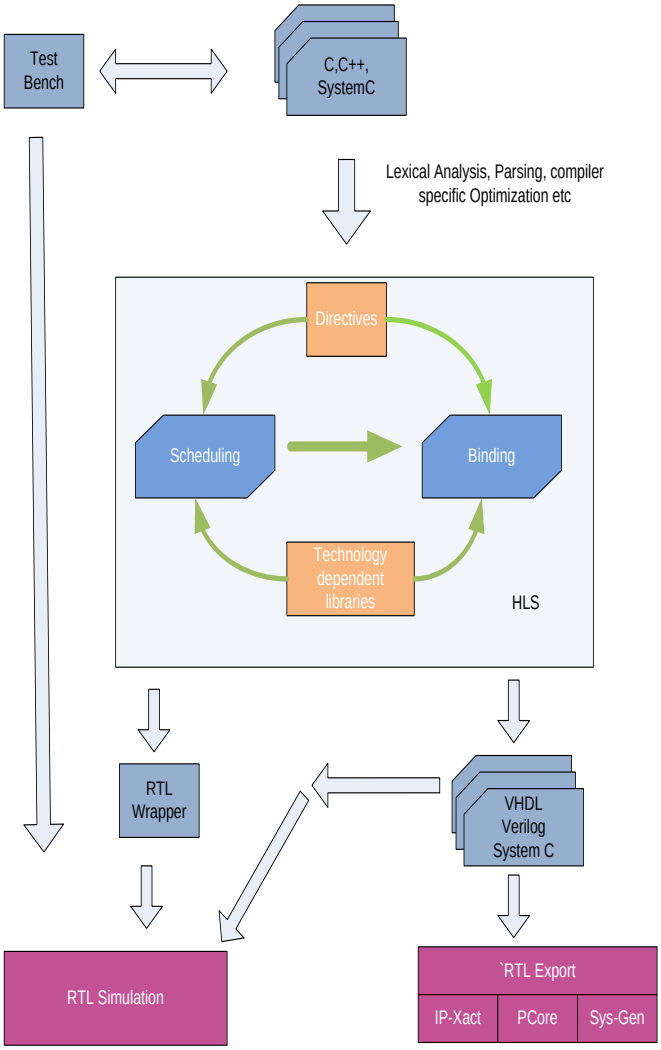


Figure 3.9 Vivado HLS development model

The entire process for Vivado HLS RTL code generation can be summarized with the flow chart in figure 3.10. It is important to note, that the flow chart merely provides a guide line for the designer and is in no way a master piece that should be followed step by step. At any point during the development process, the designer can add or remove relevant steps and tools as they see fit for the particular project/product they are developing.

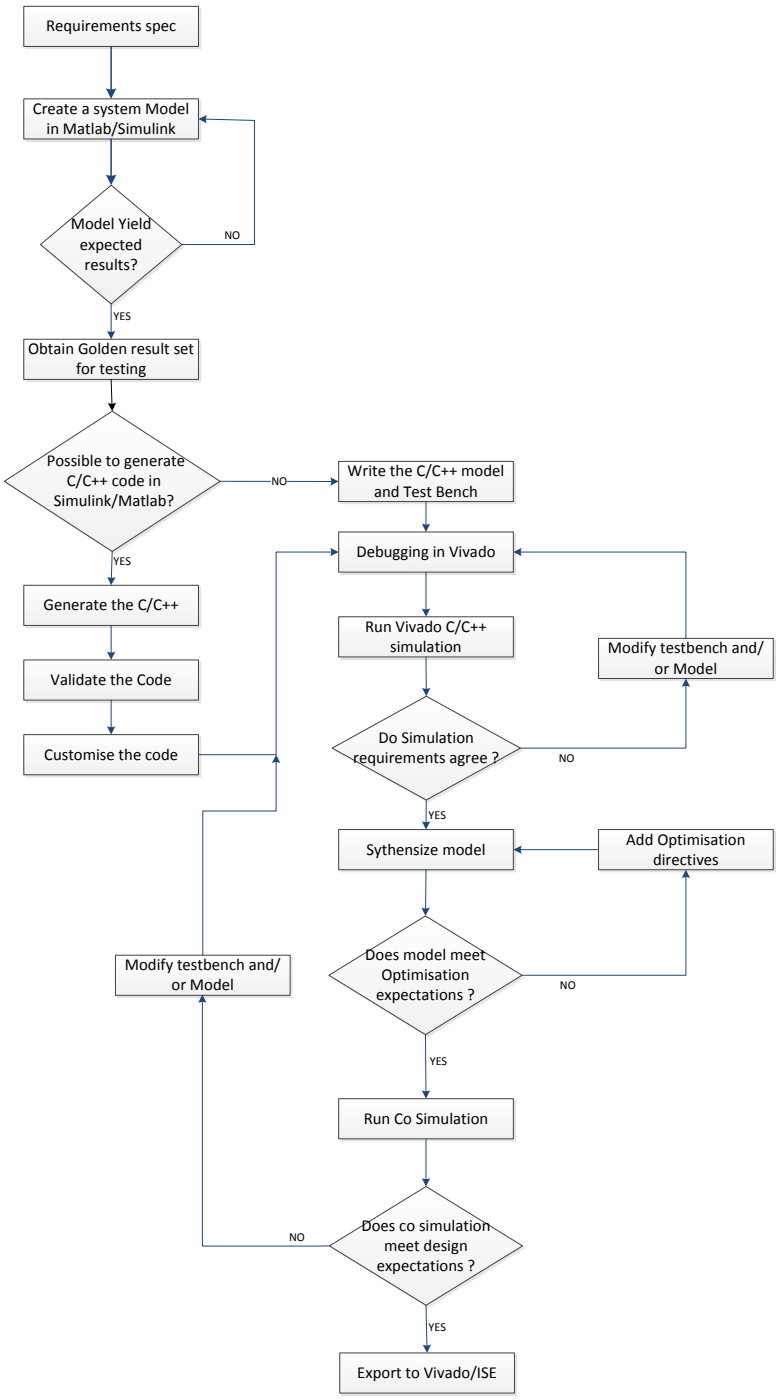


Figure 3.10 Showing steps for the Vivado HLS design flow

Chapter 4. HDL Coder

4.1 Introduction

The HDL coder tool embedded in the MATLAB/Simulink environment lets a designer generate synthesizable HDL code for FPGA and ASIC implementations in the following steps [17]:

- Build a model of the design using a combination of MATLAB code, Simulink and Stateflow charts.
- Optimize the design to meet area-speed objectives
- Generate the design using the integrated HDL workflow advisor for MATLAB and Simulink
- Verify the generated code using HDL verifier.

HDL coder also features a Workflow advisor for automating the FPGA design process from MATLAB algorithms and Simulink models into Xilinx and Altera FPGAs. The HDL Workflow Advisor integrates all steps for traditional FPGA design process, and also includes the following features [15]:

- Checking the Simulink model for HDL code generation compatibility
- Generating HDL code, an HDL test bench, and a cosimulation model
- Performing synthesis and timing analysis through integration with Xilinx ISE and Altera Quartus II
- Estimating resources used in the design
- Back annotating the Simulink model with critical path timing

The HDL tool in MATLAB leverages on the HDL Workflow to guide the designer during the process of generating HDL code. The HDL Workflow Advisor automatically converts MATLAB code from floating-point to fixed-point and generates synthesizable VHDL and Verilog code. Similarly the Workflow Advisor can generate VHDL and Verilog code from Simulink and Stateflow. The power of the tool lies in its ability to generate code from algorithms built using a library of more than 200 blocks, including Stateflow charts. In addition, the MATLAB language gives designers the capability to model their algorithm at a high level using abstract MATLAB constructs. This, together with the huge library provides complex functions, such as the Viterbi decoder, FFT, CIC filters, and FIR filters, for modeling signal processing and communications systems and generating HDL code [13].

In addition to target-independent, synthesizable VHDL and Verilog code; Code generation support for MATLAB functions; System objects and Simulink blocks, Mealy and Moore finite-state machines and control logic implementations using Stateflow; Workflow advisor for programming Xilinx and Altera application boards; Resource sharing and retiming for area-speed tradeoffs; the tool features Code-to-model and model-to-code traceability for DO-254 and Legacy code integration. 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 [12] [13].

HDL code Verification is also greatly supported by the tool. HDL Coder generates VHDL and Verilog test benches for rapid verification of generated HDL code. Designers are able to customize an HDL test bench using a variety of options that apply stimuli to the HDL code. They can also generate script files that automate the process of compiling and simulating the generated code in HDL simulators. HDL Coder works with HDL Verifier to automatically generate two types of cosimulation models [16]:

- HDL cosimulation model, for performing HDL cosimulation with Simulink and an HDL simulator (HDL coder supports Cadence Incisive, Mentor Graphics' ModelSim and Questa simulators).
- FPGA-in-the-loop (FIL) cosimulation model, which is used for verifying the DUT with Simulink and an FPGA board

HDL coder documents the generated code in an HTML report that contains hyperlinked HDL code and a table of generated HDL files. Hyperlinks in the HDL code allow the designer to navigate to the corresponding MATLAB algorithm or Simulink blocks that generated the code. The tool supports code traceability for applications that adhere to DO-254 standard by enabling designers to: [18] [12]

- Navigate to MATLAB code from generated HDL code
- Navigate between Simulink blocks and generated HDL code for bidirectional tracing
- Insert user-controlled comments and descriptions to improve code readability

In addition to the above features, the Simulink Verification and Validation tool box allows HDL Coder to embed system requirements as comments within HDL code generated from Simulink or Stateflow. This helps designers achieve complete transparency throughout the entire workflow, from system requirements to generated HDL code. More details about the functionality supported by HDL coder can be found in [19].

This thesis work aims at establishing how the VHDL code generated by HDL coder can meet hardware resource constraints. Optimizations of particular interests as earlier discussed, are area, throughput, timing and latency. Furthermore, this section assumes that the reader has already reviewed the section on Vivado HLS tool. Definitions clarified earlier in that section are therefore not repeated in this section. The DUT is the second order IIR filter described in section 2.5 and the procedure for evaluation of the tool follows the requirements in section 2.3.

4.2 Area Optimization in in HDL coder

HDL coder fundamentally uses the sharing optimization to ensure resource re-use in the generated RTL code. In addition, MathWorks advises designers to follow the following guidelines when implementing designs in MATLAB/Simulink [19]:

- Input and output data should be serialized since parallel data processing structures require more hardware resources and a higher pin count.
- Designers should use add subtract algorithms instead of algorithms that use functions like sin, divide and modulo. This is because add and subtract operations use fewer hardware resources.
- Designers should avoid large arrays and matrices since they require more registers and RAM for storage.
- Code should be converted from floating-point to fixed-point since floating-point data types are inefficient for hardware realization. HDL coder provides an automated workflow for floating-point to fixed-point conversion as discussed earlier.
- In addition unrolling loops increases speed at the cost of higher area; unrolling fewer loops and enabling the loop streaming optimization conserves area at the cost of throughput.

By default, HDL implements hardware that is a 1-to-1 mapping of Simulink blocks to hardware module implementations. The resource sharing optimization enables users to share hardware resources by enabling an N-to-1 mapping of 'N' functionally-equivalent Simulink blocks to a single hardware module. The user specifies 'N' using the 'SharingFactor' implementation parameter [19] [20].

In addition since the design does not have vector inputs, the streaming optimization to conserve area is not applicable in this design and therefore is not discussed. Further information about this design optimization can be obtained in [21].

4.2.1 Sharing to Realize an N-to-1 Mapping

The filter block uses six multiplier blocks and it's therefore reasonable to assume that using a 'SharingFactor' of six should result into a single multiplier. In addition, the filter block contains two blocks of each of the adders and subtractors. Similarly, it is reasonable to think that, setting a 'SharingFactor' of 2 for both the adders and subtractors should result into a design with only one adder and one subtractor. The filter design also includes two converter blocks. In the implementation of RTL code, such blocks are converted into adders. In a similar way, it's reasonable to assume that with a sharing factor of 2, these may again be reduced to just one adder. The above is the objective of area reduction.

The sharing optimization is implemented using time-division multiplexing. Simulink requires the outputs of the shared resource to be sampled at the predefined sample rate, so HDL Coder overclocks the shared resource at a faster rate than the data rate. In general an N-way shared resource results in N times overclocking in Simulink [21] [19]. Known limitations for this optimization can be found in [19], however the obvious limitation being that the design cannot be oversampled more than the supported clock speed on the target device [19].

In this design, the shared architecture, which includes the shared resources, multiplexer-serializer at the inputs and demultiplexer-deserializer at the outputs, should be able to operate at 6 times the rate of the input data, because 'Sharing factor' = 6.

As stated in [19], during the model validation process, the delay balancing feature in Simulink (turned on in this design) is essential for HDL Coder to automatically balance delays within the IIR filter model. HDL Coder can introduce additional delays in the HDL implementation for a given model. These delays may be introduced by either certain block implementations or by optimizations for the purpose of improving the efficiency of the hardware implementations. However, introducing delays on only certain paths can result in a functional behavior that is different from the original intent of the user model, thereby violating functional equivalence between the original user model and the HDL implementation.

Delay Balancing is a feature supported by HDL Coder for automatically balancing such newly introduced delays across all cut-sets, ensuring that functional integrity is preserved with reference to the original model. This equivalence relationship can be confirmed by invoking the validation model workflow that enables the user to visualize the HDL Code-generation model, the delays introduced by implementations and those introduced by delay balancing and verify the equivalence relationship with the original model [19]. The validation model generated in this design was checked to ensure that the output actually corresponds to the expected output. The observation was that the functional equivalence of the model is maintained during the process of delay balancing.

4.2.2 Block Support, Atomic Subsystems and Extensions

According to [19], HDL Coder supports resource sharing of 4 block types: product, gain, atomic subsystem, and MATLAB function. Sharing functionally for the product and gain blocks means that the multipliers in the HDL implementation will be shared. Two product blocks are functionally equivalent if:

a) Data types of their inputs and outputs are identical.

b) Block parameter settings are identical.

c) HDL block properties are identical.

According to [19], for a gain block, functional equivalence additionally requires that the constant value and data types are also identical. However, if the gain constant data types are identical for two gain blocks with different gain constant values, HDL Coder can share them. Similarly, if a gain block can be implemented as a product block with a constant input, and it has the same data types as another product block in the design, the coder can share them.

The third block type, atomic subsystem, is useful for sharing functionally equivalent islands of logic encapsulated inside atomic subsystems. Two atomic subsystems are functionally equivalent and can be shared if:

a) Their Simulink checksums are identical

b) Their HDL block properties are identical.

The fourth block type, a MATLAB function block, can be shared if it is stateless i.e. does not contain persistent variables, or if its persistent variables are not updated in one call to the function and read on a subsequent call. If the designer wants to share multipliers within a single MATLAB function block, he/she can set its SharingFactor block property.

Based on the above descriptions from mathworks, it's reasonable to conclude therefore that the only candidates for sharing in the model are the multiplier blocks.

4.2.3 Analysis of results

Without the sharing factor the design is able to achieve the following hardware resource usage as viewed from the code generation report.

Table 4.1 Resource usage before sharing

Multipliers	6
Adders/Subtractors	6
Registers	21
RAMs	0
Multiplexers	0

The table 17 above clearly shows that the number of multipliers (corresponding to DSP48As since the signal processing parameter is turned to on in the model) used is very high, approximately 41% of the entire available DSP48A resources available on chip, which is not acceptable in this design. One can deduce therefore from looking at the instances that the increased number of DSP48A usage is due to the number of multiplication blocks in the model. Each multiplier block corresponds to 4 DSP48As. There are approximately 6 multiplication blocks and these will result into 24 DSP48As. The objective at hand therefore is to reduce this number to just 4 DSP48As instances which correspond to only one multiplier implementation in the design. By setting the sharing factor to 6 the number of multiplier blocks in the design could be reduced to 1. The synthesis and mapping results clearly demonstrate the EDA tool results. Important to note at this point is that HDL coder does not give actual resource usage estimates in a summarized form as Vivado HLS. The designer has to therefore invoke the required EDA tool and examine the reports generated by the tool for thorough conclusions on resource usage.

The table 4.2 below shows resource utilization after resource sharing. The case for the multipliers is fairly trivial, since the multipliers are 6, a sharing factor of 6 results into one multiplier being shared in the design. Figure 4.1 shows how the HDL coder implements sharing at the Simulink level. The design however uses more registers as can be seen by the increase in the number of the registers and flip-flops used by the design. However this is expected, and the end result is conserving more multipliers which are in less numbers as compared to flip flops, the tool is able to reduce the overall area usage.

Table 4.2 Resource usage after sharing

Multipliers	1
Adders/Subtractors	6
Registers	28
RAMs	0
Multiplexers	18

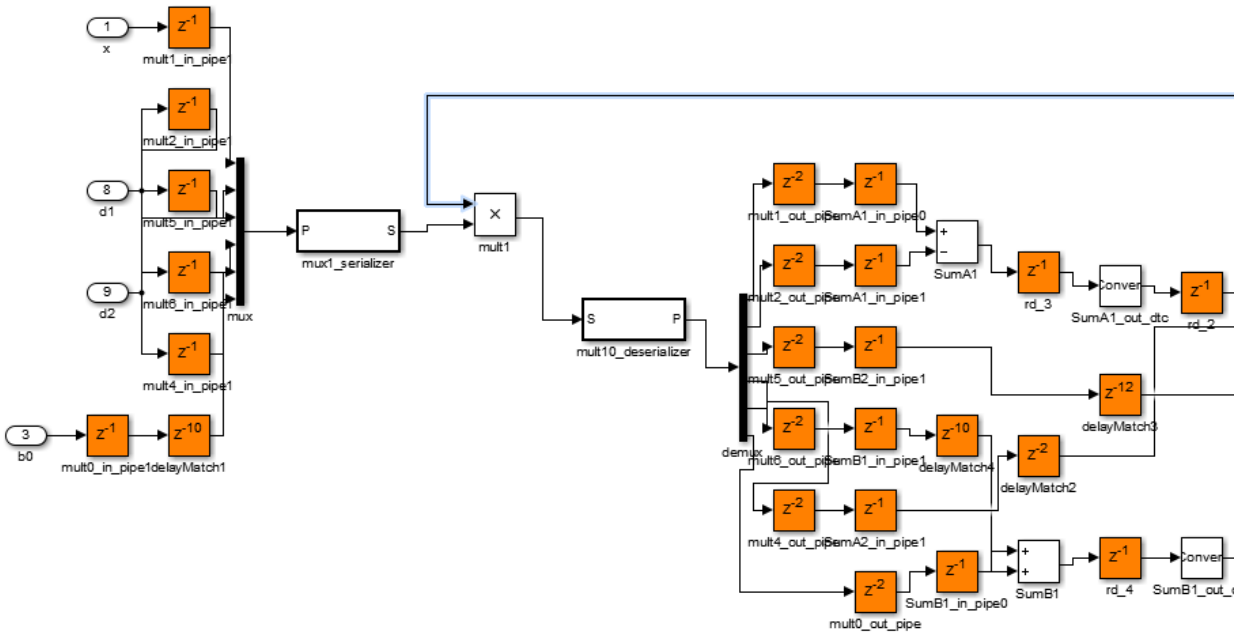


Figure 4.1 Illustration of N to 1 sharing in Simulink

When the adders and subtractors were grouped into subsystems so that a sharing factor of two could be realized, HDL coder did not share these resources on the ground that no candidates were found that matched the criteria and hence it deduced that no legal sharing factors were available for the subsystem.

The case for the converter blocks is not so trivial, when successfully shared, the sharing factor for the multipliers did not work. In effect, HDL coder notices a feedback loop in this arrangement. A sharing factor of five could however be achieved yielding only two multipliers and 5 adders/subtractors. However this design configuration is much more area expensive when compared to the design configuration with 1 multiplier and 6 adder/subtractors. A similar scenario arose when an attempt to perform loop streaming to minimize area was attempted.

An important observation at this point is, that in as much as the tool may realize considerable area savings using the sharing factor, the designer has to also invest considerable effort in identifying the patterns in the subsytems. It is also realistic to say that even after these exhaustive trials, the tool does not guarantee that the goal will be obtained even if it is achievable using the alternative approach of hand written RTL code. Also, some of these bottlenecks were expected as is revealed in the tool limitations in sub section 4.2.2.

Table 4.3 shows the results after design synthesis using ISE. For details of this synthesis see Appendix B1.

Table 4.3 Showing resource usage after design synthesis

Resource	Number
Slices	340
LUTS	679
FF	1132
DSP	4
BRAM	0
SRL	28

In conclusion, synthesis and mapping results from ISE reveal considerable reduction in the area used by the design after sharing, which is a significant plus on the tool. Significant to these reductions and increases in these designs however is but one important design achievement, the number of DSP48A1s used for the design has been reduced. Without a sharing factor up to 41% of the DSP48A1s on the FPGA are used. With a sharing factor, this percentage reduces to just 6%, which is a considerable reduction in the resources. Correspondingly there is an increase in the number of LUTS, SRLs, FF and Slices in the design by a factor of approximately 2 overall. If ignored this could result into increasingly large designs.

4.3 Comparison of Area Statistics with the Current Implementation

Table 4.4 HDL coder resource usage comparison with the current design

Resource	Current Design	Generated Design
Slices	150	340
LUTS	277	679
FF	486	1132
DSP	4	4
BRAM	0	0
SRL	18	28

Overall the HDL coder generates VHDL code with a high resource usage. In the table above, there is a rise in the used slices by a factor of 2.3, a raise in the used LUTS by a factor of 2.5, a raise in the number of used flip-flops by a factor of 2.3 and a raise of SRLs by a factor of 1.6. The DSP481As are however the same. In conclusion, it is reasonable to say that code generation with HDL coder results into increased resource usage.

4.4 Latency and Throughput in HDL Coder.

In order to increase throughput, HDL coder gives the option of pipelining. This option is handled together with improving timing in the section below.

In addition the coder specifies a maximum computation latency parameter which enables designers to specify a time budget for the HDL coder when performing a single computation. Within this time budget, the coder does its best to optimize the design without exceeding the maximum oversampling ratio. When the designer sets a maximum computation latency, N, each Simulink time step takes N time steps in the implemented design. In Essence what this means is that the coder implements a design which captures the DUT inputs once every N clock cycles, starting with the first cycle after reset. The DUT outputs are held stable for N cycles. The requirement of this filter is that the design should be able to have at least an iteration interval of 6, so the maximum computational latency was set to 6 as shown in figure 13 below.

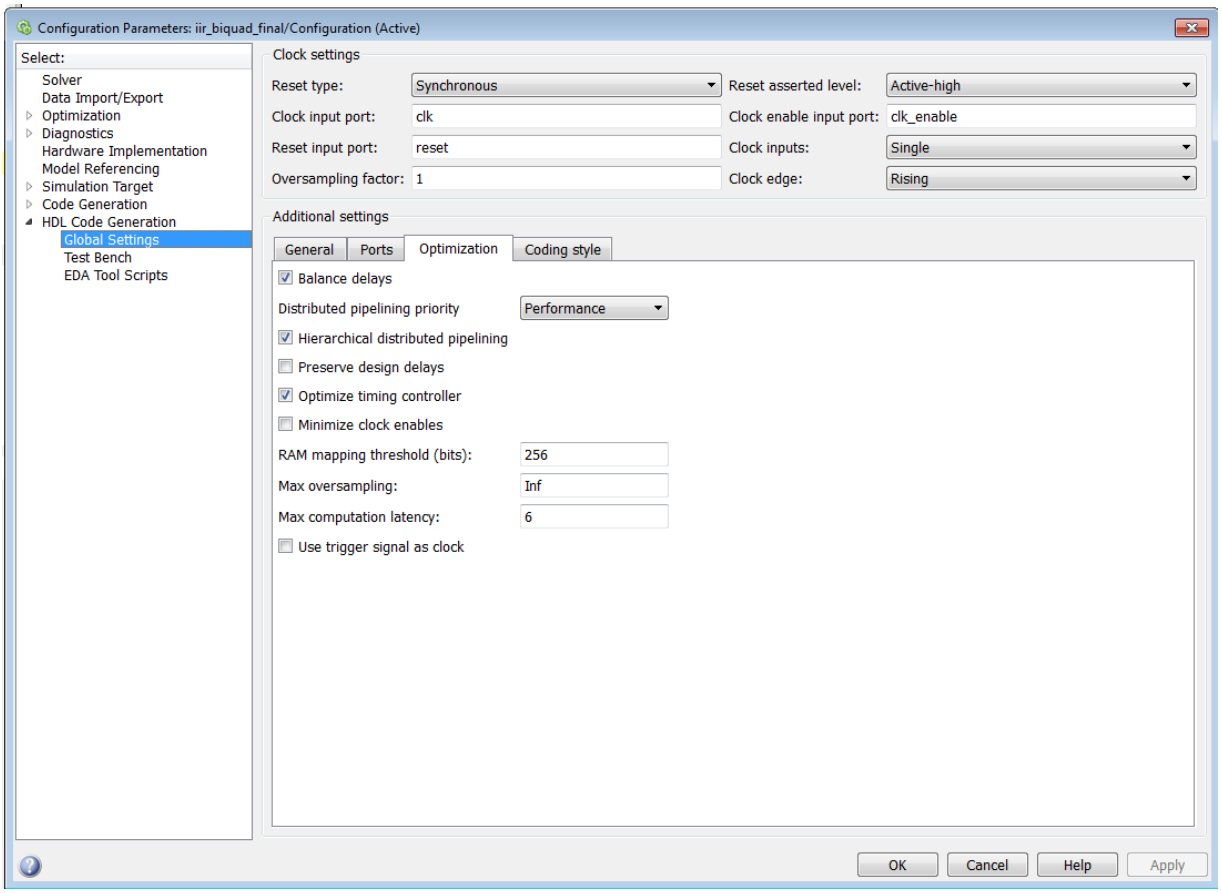


Figure 4.2 Illustrating maximum computational latency in the HDL Coder options

HDL coder does not give a latency summary of the resulting design immediately after code generation, consequently, the designer has to perform co-simulation of the design in order to get design estimates of latency. This thesis work maintains the fact that the designer should be able to get these measures even before such testing is done as can be observed in the methodology by Vivado HLS. This is important because it significantly shortens design cycles. The coder however supports the, "Annotate Model with synthesis Result", feature that is able to trace the longest delay path in the design for optimization, but this is after mapping of the design. In practice designers may want to do get estimates of smaller parts of the design/core before testing. This is not supported by the HDL coder.

In addition, absence of such a report might make in-depth analysis of the design not possible. For large or delay sensitive models a designer may need to know which operations, at RTL, are processed in which clock cycle early in the design process. This information can later be used in constraining the subsystems of the design further to meet pipelining, Iteration Interval, overall latency etc., demands of the design.

4.5 Timing Optimization in HDL Coder

HDL coder utilizes the concept of distributed pipelining which is a subsystem-wide optimization to achieve high clock speed hardware. By turning on 'Distributed Pipelining', the coder redistributes the input pipeline registers, output pipeline registers of the subsystem and the registers in the subsystem to appropriate positions to minimize the combinatorial logic between registers and maximize the clock speed of the chip synthesized from the generated HDL code [21].

To increase the clock speed for any given design, the designer can set a number of pipeline stages for any subsystem. Without turning on distributed pipelining, the specified number of registers will be added to each of the output ports of the subsystem [21] [19].

Once distributed pipelining is turned on, the registers in the subsystem, including output pipeline registers and input pipeline registers, will be repositioned to achieve best clock speed. It is equivalent to retiming at subsystem level [21] [19].

4.5.1 Opportunities for Distributed Pipelining Across Subsystem Hierarchies

Since distributed pipelining is a subsystem-level parameter, different subsystems at different levels of the hierarchy can specify different pipeline stage values and different distributed pipelining settings. By default, the coder distributes only registers of the specified subsystem in this subsystem, not through the lower level subsystems. If cross hierarchy distribution is desired, users can set the distributed pipelining' for lower subsystems to 'on', then turn on the global option hierarchical distributed pipelining. When the local and global options are on, the entire subsystem, including the lower level subsystems, will be considered as a single subsystem when registers are distributed [21] [19]. To maximize the speed for the design distributed pipelining was turned on. In addition, hierarchical distributed pipelining was turned on to ensure that all the lower level subsystems benefit from register redistribution for maximum speed.

4.5.2 Analysis of results

The affirmed realization so far implements a sharing factor of 6 to reduce the area usage on the FPGA. When distributed pipelining is turned off on the filter, after mapping, ISE is able to achieve the following timing information

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)
```

```
Constraints cover 31266453 paths, 0 nets, and 4028 connections

Design statistics:
   Minimum period:  16.737ns{1}   (Maximum frequency:  59.748MHz)
   Minimum input required time before clock:  19.691ns
   Maximum output delay after clock:  16.597ns
```

As a first step to increase the clock speed 3 input and output pipeline registers were added to the design. The timing results obtained after synthesis and mapping are,

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 31294441 paths, 0 nets, and 5205 connections

Design statistics:
   Minimum period:  15.305ns{1}   (Maximum frequency:  65.338MHz)
   Minimum input required time before clock:  15.464ns
   Maximum output delay after clock:  10.369ns
```

The obtained results show an improvement in the timing achieved by the design. As a next step, distributed pipelining is turned on and the rationale is that the added pipeline registers can be further redistributed for better clock speeds. An illustration showing distributed pipelining at the model level is shown in figure 4.3.

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 31148965 paths, 0 nets, and 5500 connections

Design statistics:
   Minimum period:  14.880ns{1}   (Maximum frequency:  67.204MHz)
   Minimum input required time before clock:  17.503ns
   Maximum output delay after clock:  13.200ns
```

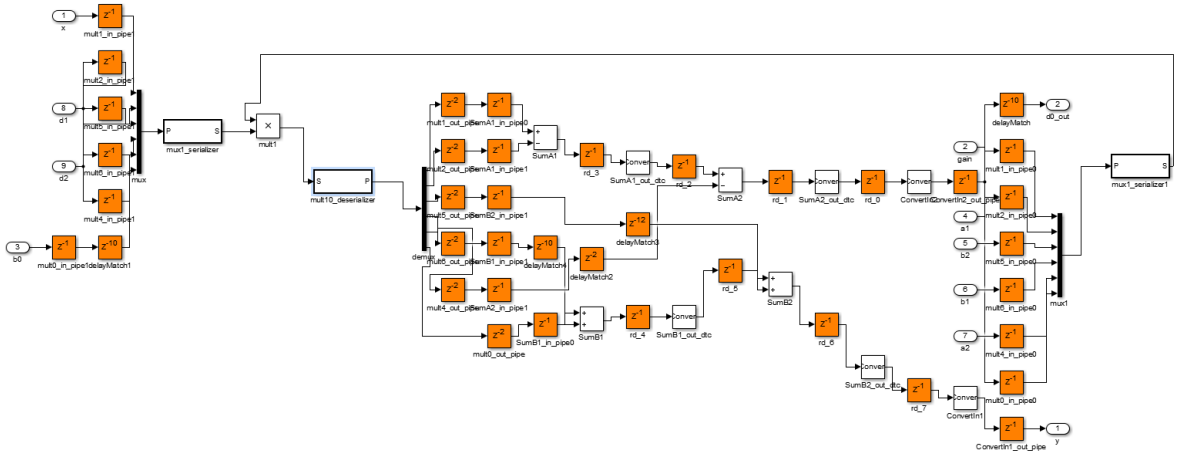


Figure 4.3 Illustration of distributed pipelining in the Simulink model of the IIR filter

Observation after this optimization is that HDL coder is able to increase the speed of the design to 67.204MHZ. As a next step the pipeline registers were subsequently increased, however any further increase in the pipeline stages results in a lesser clock speed as can be seen with a value of just 4. Similarly any value less than 2 results into a slower design, as thus, it's logical to say that the value of three gives the optimum possible pipeline stages for this design at the subsystem level. It is important to note that the rationale of choice of the value 3 is empirical and is for this particular design and does not in any way apply to other designs made in Simulink.

Timing summary with 2 pipeline stages:

```
-------------------------------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 30673953 paths, 0 nets, and 5210 connections

Design statistics:
   Minimum period:  16.118ns{1}   (Maximum frequency:  62.042MHz)
   Minimum input required time before clock:  19.351ns
   Maximum output delay after clock:  15.117ns
```

Timing summary with 4 pipeline stages:

```
-------------------------------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 31148687 paths, 0 nets, and 5394 connections

Design statistics:
   Minimum period:  15.591ns{1}   (Maximum frequency:  64.140MHz)
   Minimum input required time before clock:  15.915ns
   Maximum output delay after clock:  10.627ns
```

As a next step to improve timing, the synthesis result is annotated with the model to trace the path with the most delay. The rationale is that more pipeline registers could be added to optimize the performance of the design. Up to 3 pipeline registers are added to the lower level Simulink blocks along the critical path.

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 30557843 paths, 0 nets, and 6167 connections

Design statistics:
   Minimum period:  14.943ns{1}   (Maximum frequency:  66.921MHz)
   Minimum input required time before clock:  16.828ns
   Maximum output delay after clock:  12.765ns
```

The design however does not perform any better as can be observed in extract from the timing report after mapping above. With back annotation however, a different critical path comes up, and similarly more pipeline registers are added. The rationale is still the same with each addition, more registers could potentially decrease the overall delay. However after an exhaustive iteration over the whole design, with various configurations of input and output pipeline registers on the filter blocks, it was found out that the design does not yield any value greater than the 66.921MHZ. The statistics obtained after the whole iteration are shown below.

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 30557910 paths, 0 nets, and 6069 connections

Design statistics:
   Minimum period:  14.957ns{1}   (Maximum frequency:  66.858MHz)
   Minimum input required time before clock:  17.281ns
   Maximum output delay after clock:  13.188ns
```

This brings the conclusion that the maximum achievable clock rate of the filter design with sharing turned on is approximately 67.204MHz. The resulting increase in the registers after mapping and synthesis is shown in Appendix B2. It also goes without saying that the iterations for tuning a complex design might be enormously big.

The algorithm that the HDL coder utilizes to increase timing unfortunately did not meet the timing constraints of at least 100MHZ that was targeted at the beginning of this exercise. HDL coder fails in the second goal set out to achieve. Whereas it might be illogical to totally conclude that HDL coder cannot achieve a better timing optimization, one can reasonably conclude after such an exhaustive search that the effort in this particularly simplistic design might well surpass the effort of writing the RTL level code.

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 30557910 paths, 0 nets, and 6069 connections

Design statistics:
   Minimum period:  14.957ns{1}   (Maximum frequency:  66.858MHz)
   Minimum input required time before clock:  17.281ns
   Maximum output delay after clock:  13.188ns
```

Analysis of resource usage statistics (see Appendix B2) reveals an increase in the number of slice registers up to 3%. The number of LUTS also increases up to only 3%. Memory increases up to 1%. Overall the increase after timing optimizations is negligible and one can reasonably deduce that the resource usage has almost remained constant after an increase in speed from 59.748MHz to 67.204MHz. The goal however of 100MHZ is not met.

In an effort to achieve timing, the option of increased parallelism was explored. The rationale being that, if more multipliers are added to the design, a faster clock speed can actually be achieved. The resulting timing report is shown below.

```
Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    7.282|         |         |         |
---------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 22723 paths, 0 nets, and 4326 connections

Design statistics:
   Minimum period:   7.282ns{1}   (Maximum frequency: 137.325MHz)
   Minimum input required time before clock:   5.625ns
   Maximum output delay after clock:   8.846ns


-----------------------------------Footnotes-----------------------------
-----
1)  The minimum period statistic assumes all single cycle delays.

Analysis completed Sun May 18 15:48:42 2014
------------------------------------------------------------------------
--
```

Indeed as can be seen from the results the design is able to achieve timing requirements as anticipated. As described in [21] resource sharing results into oversampling by a factor of N to generate an area-optimized implementation with the original latency and if the coder cannot identify N shareable resources, it shares as many as it can, but it still oversamples by a factor of N. In essence resource sharing creates the following costs;

- Uses more multiplexers and may use more registers.
- Reduces opportunities for distributed pipelining or retiming, because the coder does not pipeline across clock rate boundaries.
- Multiplies the clock rate of the target hardware by the sharing factor.

Since the coder oversamples the input to the multiplier by a factor N, if the design is to operate at 100MHZ, the multipliers on the Spartan 6 have to operate at 600MHZ, which is not possible. The coder provides an option for the designer to constrain the maximum oversampling that can be implemented during code generation. This however means, in this particular scenario, with serialization and deserialization the DUT can never meet timing.

In addition the option of redistributed pipelining to increase speed may have the following significant limitation [21].

- The pipelining results might not be optimal in hardware because the operator latencies in the target hardware may differ from the estimated operator latencies used by the distributed pipelining algorithm.
- The coder distributes pipeline registers around the following blocks instead of within them:
    - Sum (cascade implementation)
    - Product (Cascade implementation)
    - Zero-Order Hold

4.6 Performance Comparison with Current solution

The filter values for the generated design agree with the values from the current design, however the cosimulation was not carried out for the MathWorks tool chain because of absence of a license for HDL verifier. The student edition does not also have a license for the verifier. Resource is generally poor in HDL coder as compared to the current design as can be seen in table. Timing can only be achieved after a considerable increase in DSP481As.

4.7 Design Methodology evaluation

HDL coder starts from MATLAB code, Simulink / State Chart designs, it converts the datatypes in the design to fixed point datatypes. The HDL coder algorithm parses the design and extracts the control and datapath behavior of the DUT by building a control flow graph of the design. The tool then performs the necessary optimizations and through scheduling and binding it generates VHDL or Verilog for the DUT.

The MATLAB/Simulink environment requires good knowledge in model based design as well as knowledge in MATLAB programming language. The HDL coder tool semantics do not require much strain on the part of the designer who is already well versed with the MATLAB/Simulink environment since the tool uses the language constructs of the MATLAB/Simulink environment. However for the new beginner, the tool environment may present a significant challenge to learn and adopt easily.

As detailed in [21], and the MATLAB/Simulink documentation, the environment provides the user with a very exhaustive analysis and verification platform. It supports DUT simulation, cosimulation as well as hardware in the loop testing. The designer is able to analyze the DUT under a multitude of test signals all of which can be generated locally in the environment. In addition, designers are able to use other third party simulators like Modelsim to test the generated designs using the HDL verifier. Furthermore, the MATLAB/Simulink environment provides an ideal platform for algorithm design. The design studio provides complete transparency in the design methodology from requirements analysis up to implementation of the designs. In addition the tool supports various configurations of the workflow advisor.

Designers can configure the workflow advisor to generate; IP cores, Generic ASIC/FPGA, FPGA Turnkey, Simulink Real-Time FPGA I/O, FPGA in-the-Loop testing.

The generation procedure can be approached either using the GUI or the MATLAB command prompt. Designers can easily change and save entire design configurations on the fly. In addition the tool provides great support for most RTL code tasks like mapping to RAMs, pipelining, FIFOs etc. More details of the extent of support can be obtained from [21].

To increase design speed optimization and throughput, HDL coder supports adding IO pipeline stages, distributed pipelining, loop unrolling etc. To reduce the area, HDL coder supports a number of optimizations which include: Loop streaming; resource sharing; mapping matrices to block RAMS etc. More information on the extent of support can be found in [21]. HDL coder also generates easy to read VHDL code which makes it easy for experienced designers to easily devise new strategies for optimizing the code using HDL coder. In addition, the tool allows designers to integrate legacy code into their synthesis designs, this goes a long way in making generated designs much easier to integrate with already existing designs.

In summary, a flow chart detailing the design flow using HDL coder is shown below. It's important to note however that this is not the only design flow that is possible using MATLAB/Simulink. Other design flows are possible as detailed in [2]. Furthermore since an earlier analysis of these design flows did not clearly show performance improvements as mentioned in chapter 2 of this report, a further analysis of these flows was not done during thesis. A repetition of these design flows is also not recommended as the results will potentially be same.

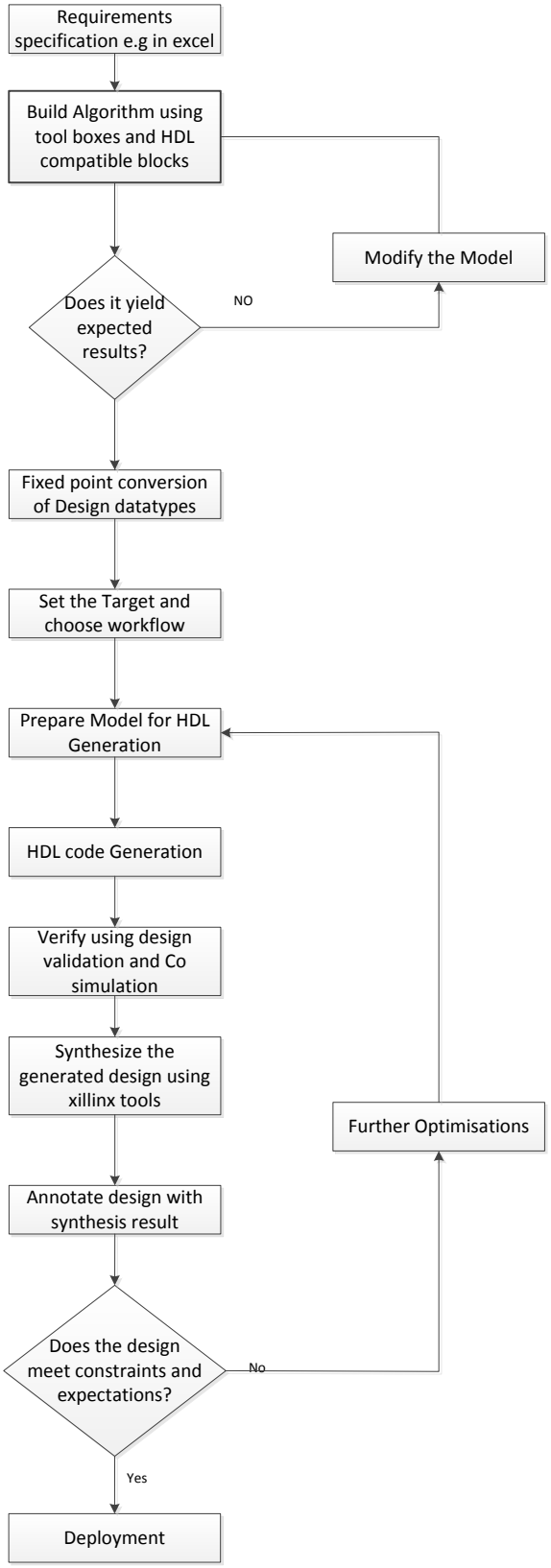


Figure 4.4 Flow chart illustrating the design flow in Simulink.

Chapter 5. Evaluation and comparison of the Design Flows

5.1 Comparison of Area Optimizations

Both HDL coder and Vivado HLS environments provide various options of reducing the area usage on the FPGA. More details on this can be found in [4] for Vivado HLS and [21] for MATLAB/Simulink. A significant difference, as can be noticed from the design optimizations carried out in this design, is that Vivado HLS gives the designer very easy to use and elaborate text commands to direct the scheduling and binding process. In addition, the tool performs area optimizations with significant knowledge of the underlying technology on the FPGA specified in the solution. The designer is able to specify explicitly the number of cores or/and operations he/she wants Vivado HLS to use and the tool tries its best to implement the constraints. HDL coder on the other hand relies on the sharing optimization which is a sub-system level optimization. Once a sharing factor has been specified, the designer relies on the tool to evaluate the possibility of sharing depending on the factored number of blocks considered. Simulink in effect implements a serializer and de-serializer by oversampling the input values and downsampling the output values of the shared resource respectively. In addition, Vivado HLS provides resource estimates with reference to the specific FPGA used in the solution whereas inasmuch as MATLAB/SIMULINK does the same, it does it in a way not specific to the hardware technology. In essence it is not straight forward to the designer as to whether the resulting resource usage shall translate to the same on the FPGA. In addition the designer cannot specify particular resources or cores for the scheduler and binder to use during the process of RTL code generation since it is not aware of the hardware technology.

5.2 Comparison of Latency and Throughput optimizations

Both HDL coder and Vivado HLS environments allow the designer various choices for use to reduce latency and improve throughput. More details on this can be found in [4] for Vivado HLS and [21] for MATLAB/Simulink. Designers using Vivado HLS can be able to specify explicitly what measures of optimizations like Latency, Iteration Interval the tool should use in terms of clock cycles. The tool is able to do so because it has prior knowledge of the component delay estimates of possible cores on a given FPGA. HDL coder supports the same functionality for the Iteration Interval but does give an option to specify the latency of the design. The designer relies on the algorithm in HDL coder to schedule to its best capacity. In addition the analysis report from HDL coder does not show the estimated Latency of generated design after code generation. This maybe a valuable input for the designer early in the designer process without requiring him/her to simulate the RTL code before a conclusion on such measures can be reached. In addition, wave forms for bigger designs can be at times hard for system engineers to understand.

5.3 Comparison of Timing optimizations

Vivado HLS provides a detailed analysis of timing for the DUT. In addition to providing a summary in terms of latency, maximum achievable clock of the design and Iteration Interval, it provides a detailed analysis in form of a timing diagram. The designer is able to explicitly examine in which clock cycles particular operations happen and modify or approve design directives accordingly early in the design process. Such information is missing in HDL coder. It's only after synthesis that a designer can examine the synthesis report to see how well his design measures in these estimates. In addition, the reports generated by ISE are normally large and require the designer to spend more time trying to obtain this information.

5.4 Interface Synthesis

Vivado HLS supports the synthesis of both single and double handshake interfaces. The application designer simply specifies appropriate directives to the tool. HDL coder on the hand relies on the user to provide any handshake signals. It however provides the clock enable signal for the design.

Chapter 6. Hardware Integration and Testing

6.1 Integration of the Cascade Filter into the Current Solution

As mentioned in section 2.5, IIR filter design, the cascade is combination of seven second order sections, with the ability of being singly configured by the DSP. In addition, the filter application provides a mechanism for filter data storage to the dual port RAMs. The output values stored are decimated accordingly to the rates supported by the DSP or eTDM core on the FPGA.

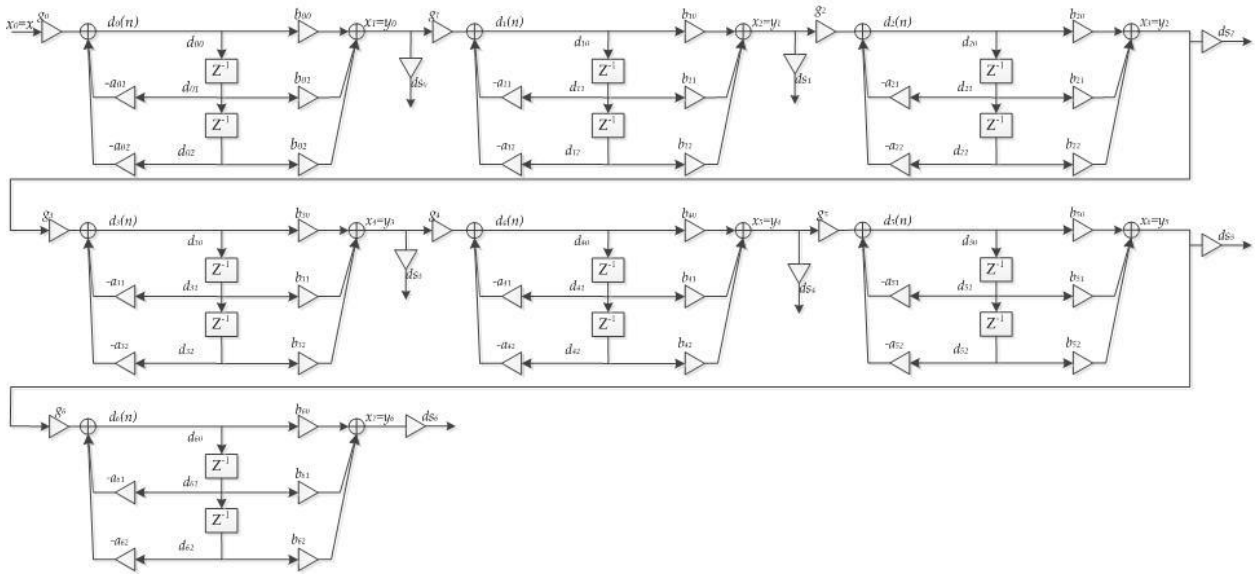


Figure 6.1 Cascade of the SOS IIR filter

6.2 Hardware Testing

Since the design produced by HDL coder did not meet timing and also resulted into an area expensive implementation, further analysis and deployment of the cascade filter was not done. The design produced by Vivado HLS was further upgraded to generate an IIR filter of order 14 as shown in figure 6.1. The filter generated is configurable by the DSP. Characteristics such as order, cutoff frequency, sampling frequency etc. are all changeable as is the case for the current implementation. The core/design was then integrated into the current FPGA solution on the PS741 voltage measuring board for hardware testing.

A simplistic test to compare the current design and the generated design was to implement a module in the PS74x module to measure both the maximum and minimum values of the filter. Then using a stimulus from the frequency generator into the ADCs (with varying inputs e.g. step, impulse etc. and frequencies), both implementations can be investigated to see if the results match.

The module for the calculating the maximum and minimum values was successfully integrated into the project and the solution implemented. However due to complexities in the communication cores and DSP applications on the board, the final runs on the hardware could not be done. It is also important to note that other communication and signal processing infrastructure on the board are beyond the scope of this thesis work.

It is the recommendation of this research therefore that ABB AB continues the runs necessarily to validate the hardware implementations. The solution with the measuring infrastructure was implemented.

The graphs and test data resulting from the logic implementation and simulation are not included in this report because they are clearly ABBs property. However an attempt has been made in the analyses to state whether the expected results match with the actual results obtained during any process.

Chapter 7. Conclusion and Recommendation

7.1 Conclusion

This work has shown that using high level synthesis in FPGA application development significantly achieves accelerated product development cycles. Product developers can be able to estimate hardware resources early in the development stage and be able to select the required hardware with certainty. In addition to minimizing development time, the HLS methodology reduces the amount of errors in the development process. This further reduces hard to find bugs in the final electronics. Adopting such a methodology further simplifies the testing and verification process since all the high level code and models are directly verified using appropriate test benches which are generated during code generation.

This work has also established that the Vivado HLS tool provides the designer with a mechanism for influencing the HLS process (scheduling and binding) with significant granularity as compared to the HDL coder. A designer can explicitly specify the number of operations, specific cores, function instances, RAM cores, communication interfaces etc. quite easily in Vivado HLS as compared to the HDL coder. In addition the tool provides the designer with a detailed analysis of the design with clock level granularity i.e. the designer is able to establish quite easily which operations are performed in which clock cycle and which variables, either in the source code or the generated code, that are affected. It is important to note here that the Mathworks environment greatly supports this kind of functionality, tracking through the entire development process right from requirements to RTL code, however it does not offer the detailed design summary of the design parameters for each clock cycle.

This work has also shown that the designer may be able to achieve the design objectives of area, throughput, latency and timing in an easier way in Vivado HLS as compared to HDL coder. This may be fundamentally because the aspects of pipelining, resource usage etc., are handled in a much better way in Vivado HLS compared to HDL coder. In addition, the tool provides the designer with a direct and easy way to specify constraints.

In addition, the MATLAB /Simulink environment provides a multitude of tool boxes that are especially designed to support algorithm design and development. Control, signal processing, image processing etc. are very well developed in MATLAB/Simulink as compared to the Vivado HLS tool. In addition, the MATLAB /Simulink environment does not only rely on a textual language, but also utilizes Simulink and Stateflow which are graphical (model) based programming environments. This gives system designers a number of ways to think about and develop algorithms specific to a given problem or application. This in effect speeds up productivity more.

The HDL coder workflow also supports both Altera and Xilinx FPGAs and seamlessly integrates into their respective synthesis tools. This gives developers a wider coverage of hardware technology. In addition, the tool supports addition of legacy code for final design synthesis.

This thesis work has established that HLS can in general be used to generate algorithms for all kinds of applications and in fields such as signal processing, control, Image processing, communication etc. This thesis work has also established that, not all problems met by FPGA/ASIC engineers today can be solved by simply constructing a HLM for synthesis. As such, more often than not, engineers need to construct glue logic for the cores which they generate. In addition, because of the wide sample space of design solutions and architectures, not every form of expression of design can be built in an HLS tool. As such engineers need to devise a solution in this circumstance and this may necessitate writing VHDL or Verilog.

In addition, it is very much desirable that an HLS tool should completely abstract the details of the hardware technology. This gives system engineers the ability to be able to develop hardware without a lot of knowledge about the underlying hardware technology. Essentially they concentrate on the algorithm only, rather than both the algorithm and the RTL language's form of expressiveness. But is it safe to say that system developers can, at this level of the HLS technology, start to develop hardware without knowing the details of the hardware technology? If no, then how much knowledge should the system developers have for them to develop very well optimized hardware solutions? Will this result in overall economic gain in product development or loss? This thesis work finds the answer to these question not trivial, however, the following pointers can be helpful to think about.

- The designer still needs to know the exact budget and core limitation of the hardware they are targeting. They need this knowledge to even better optimize the algorithm they are developing sometimes. True as it may be, that the designer will not write VHDL/Verilog, he/she may not be exempt from the knowledge of the hardware technology and more so the details.
- Since some parts of the project may require writing RTL code, the designer is not yet totally exempt from the use and detailed knowledge of hardware.
- All the tasks that FPGA/ASIC engineers meet today are not yet supported by HLS tools. Designers may for example still need to construct constraints, Instantiate IP cores, analyze simulations produced by third party companies like Mentor graphics. In many cases, they also need to write test benches for the designs in which the generated cores are instantiated.
- All tools definitely require very good knowledge in the fields for which the final products are to be used for example, a system engineer may need to know signal processing if they are to develop a core targeting signal processing applications, or Communication, for a core targeting the same. True as it might be, that for some applications, the work is reduced by availability of already written libraries, the designer still has to know how to use these libraries. Designers also need to learn how to effectively use these environments, and more so, get conversant in the languages these tools use. Essentially as earlier on stated in section 2.5 of this report, a good tool should be easy to learn.
- It may be hard to estimate the exact amount of loss or gain that can result, however it might be helpful to remember that poorly optimized designs, either as a result of a bad tool or lack of enough knowledge by the designer, can result into unnecessarily large

FPGA /ASIC chips. This may in the long run reduce significantly even the economic advantages of producing on a large scale.

This thesis work has also established that the MATLAB/Simulink environment gives designers a very rich platform to analyze and develop good algorithms in a multitude of disciplines. It provides designers with both model and textual based programming languages to use thus increasing the ways in which developers can express themselves. This however means, that developers have to learn more and that the environment may present challenges for new users due to its complexity. In some cases, identifying patterns in models for design optimizations (constraints) may increase development time as was seen in this design. Vivado HLS on the other hand predominantly provides for textual programming only. It however provides visual aids to help analyze the designs and provides an easy to use syntax (using pragmas or tcl scripts) for specifying design constraints. It also features some libraries such as signal processing, communication etc., to help designers accomplish algorithm development. As such, it is much easier to learn and use Vivado HLS compared to MATLAB/Simulink. It is important to note that, MATLAB/Simulink provides more support for an algorithm developer and in some cases experienced developers may prefer to use both tools when developing algorithms.

7.2 Recommendations

This work has established that the Vivado HLS tool generates RTL code that is better optimized when compared to code generated by the HDL coder in terms of area, latency, throughput and timing. It has also demonstrated that it is easier for the designer to specify directives with better granularity in Vivado HLS as compared to HDL coder. In addition, clock cycle level analysis is better performed in Vivado HLS than in HDL coder. It is the recommendation of this research therefore that the Vivado HLS tool be chosen as the tool for use in HLS at HVDC

7.3 Future work

There is number of tools on market today for generating C/C++ code. This thesis work therefore recommends that more work should be done in establishing the best possible tool that can be used to generate C/C++ algorithms for use in Vivado HLS. As a starting point, MATLAB/Simulink environment provides the MATLAB /Simulink Coder that can generate platform independent C/C++ code.

This thesis work also recommends that the LabVIEW environment be significantly analyzed to evaluate how measures of Latency, throughput, timing and area compare with Vivado HLS. Since the environment supports both textual and graphical programming, it is reasonable to think therefore that the tool provides a very good platform for system designers to develop algorithms. In addition, the tool features a number of tool boxes to sufficiently supplement the developers' efforts in implementing various algorithms which particularly makes it an attractive tool to investigate.

References

[1]. M. C. McFarland, et al "Tutorial on high-level synthesis", Proceedings of the 25th ACM/IEEE Design Automation Conference, June 1988.

[2]. R. Zoss et al, "Comparing Signal Processing Hardware-Synthesis Methods Based on the MATLAB Tool-Chain," 2011, January IEEE Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA)

[3]. W. Meeus et al , "An overview of today's high-level synthesis tools" Journal for Design Automation for Embedded Systems; September 2012, Volume 16, Issue 3, pp 31-51

[4]. Xilinx (2013, June 19), Vivado Design Suite User Guide on High Level Synthesis, UG902 (v2013.2). Available: http://www.xilinx.com/support/documentation/sw_manuals /xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[5]. P. Coussy et al , "An Introduction to High-Level Synthesis," Design & Test of Computers, IEEE (Volume:26 , Issue: 4), August 2009

[6]. M. Haldar et al, "FPGA Hardware Synthesis from MATLAB," IEEE, Fourteenth International Conference on VLSI Design, January 2001

[7]. C .Tseng, " Automated Synthesis of Data Paths in Digital Systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-5, 3, July 1986, pp. 379-395.

[8]. H. Flamel et al , "A High-Level Hardware Compiler," IEEE Transactions on CAD CAD-6, 2 , March 1987, pp. 259-269.

[9]. T.J.Kowalski, "An Artificial Intelligence Approach to VLSI Design" Kluwer Academic Publishers, Boston, 1985.

[10]. G.E. Moore, "Cramming more components onto integrated circuits." Proceedings of the ieee, vol. 86, no. 1, january 1998, pp. 144–144116

[11]. Xilinx. (2014, May 8), "Vivado High-Level Synthesis" [Online], Available: http:// www.xilinx.com/products/design-tools/vivado/integration/esl-design/

[12]. The MathWorks Inc, (2014, May 17). "HDL Coding standards" [Online], Available: http:// www.mathworks.se/products/hdl-coder/description7.html

[13]. The MathWorks Inc, (2014, May 17), "Generating HDL Code" [Online], Available: http:// www.mathworks.se/products/hdl-coder/description2.html

[14]. The MathWorks Inc, (2014, May 17), "Optimizing HDL Code" [Online], Available: http:// http://www.mathworks.se/products/hdl-coder/description3.html

[15]. The MathWorks Inc, (2014, May 17), "Automating FPGA Design" [Online], Available: http:// www.mathworks.se/products/hdl-coder/description4.html

[16]. The MathWorks Inc, (2014, May 17). "Verifying HDL code" [Online], Available: http:// www.mathworks.se/products/hdl-coder/description5.html

[17]. The MathWorks Inc, (2014, May 17), "Key Features" [Online], Available: http:// www.mathworks.se/products/hdl-coder/description1.html

[18]. The MathWorks Inc, (2014, May 17), "Documenting and Tracing HDL code" [Online], Available: http:// www.mathworks.se/products/hdl-coder/description6.html

[19]. The MathWorks Inc, (2014, May 17), "HDL Coder User's Guide" [Online], Available: http://www.mathworks.cn/help/pdf_doc/hdlcoder/hdlcoder_ug.pdf

[20]. The MathWorks Inc, (2014, May 17), “Resource Sharing for Area Optimization” [Online], Available: http:// www.mathworks.se/help/hdlcoder/examples/resource-sharing-for-area-optimization.html?prodcode=HD&language=en
[21]. The MathWorks Inc, (2014, May 17), “ Distributed Pipelining for Speed optimization” [Online], Available: http:// www.mathworks.se/help/hdlcoder/examples/distributed-pipelining-speed-optimization.html?prodcode=HD&language=en
[22]. L. Araujo et al, "MACH2-modular advanced control 2nd edition," Transmission and Distribution Conference and Exposition, Latin America, 2004 IEEE/PES, vol., no., pp.884,889, 8-11 Nov. 2004
[23]. Xilinx Inc, (2014, May 18). “Vivado High Level Synthesis” [Online], Available: http:// www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm
[24]. The MathWorks Inc, (2014, May 18), “Generarate Verilog and VHDL code for FPGA and ASIC designs.” [Online], Available: http:// www.mathworks.se/products/hdl-coder/
[25]. Xilinx Inc, (2014, May 18), “ISE Design Suite” [Online], Available: http:// www.xilinx.com/products/design-tools/ise-design-suite/index.htm
[26]. National Instruments Corporation, (2014, May 20). LabVIEW. [Online], Available: http:// www.ni.com/labview/
[27]. Findley Media Ltd, (2014 June 12), “FPGAs are displacing asics and assps in high data rate networking applications.” [Online], Available: http:// www.newelectronics.co.uk/electronics-technology/fpgas-are-displacing-asics-and-assps-in-high-data-rate-networking-applications/51275/#sthash.nB2BTY55.dpuf http://www.newelectronics.co.uk/electronics-technology/fpgas-are-displacing-asics-and-assps-in-high-data-rate-networking-applications/51275/
[28]. Altera Corporation, (2014 June 22) , FPGAs, [Online], Available: http:// http://www.altera.com/products/fpga.html
[29]. Wikimedia Foundation Inc., (2014 June 22), “Field-programmable gate array”, [Online], Available: http://en.wikipedia.org/wiki/Field-programmable_gate_array
[30]. Xilinx Inc, (2014, June 22), “All programmable SoC.” [Online], Available: http:// www.xilinx.com/products/silicon-devices/soc/index.htm
[31]. Altera Corporation, (2014 June 22) , “SoC Overview” , [Online], Available: http:// www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html

Appendix A1

```
================================================================================
==
Timing constraint: TS_clk = PERIOD TIMEGRP "ap_clk" 100 MHz HIGH 50%;
For more information, see Period Analysis in the Timing Closure User Guide
(UG612).

 19793 paths analyzed, 2135 endpoints analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors, 0 component
switching limit errors)
 Minimum period is   8.746ns.
--------------------------------------------------------------------------------
-----

Paths for end point
grp_mult_fu_206/iir_biquad_mul_23s_23s_46_6_U1/iir_biquad_mul_23s_23s_46_6_
MulnS_0_U/Mmult_tmp_product (DSP48_X0Y10.B1), 23 paths
--------------------------------------------------------------------------------
-----
Slack (setup path):     1.254ns (requirement - (data path - clock path skew
+ uncertainty))
  Source:               ap_CS_fsm_FSM_FFd3 (FF)
  Destination:
grp_mult_fu_206/iir_biquad_mul_23s_23s_46_6_U1/iir_biquad_mul_23s_23s_46_6_
MulnS_0_U/Mmult_tmp_product (DSP)
  Requirement:          10.000ns
  Data Path Delay:      8.688ns (Levels of Logic = 3)
  Clock Path Skew:      -0.023ns (0.235 - 0.258)
  Source Clock:         ap_clk_BUFGP rising at 0.000ns
  Destination Clock:    ap_clk_BUFGP rising at 10.000ns
  Clock Uncertainty:    0.035ns

  Clock Uncertainty:          0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ):  0.070ns
    Total Input Jitter (TIJ):   0.000ns
    Discrete Jitter (DJ):       0.000ns
    Phase Error (PE):           0.000ns

  Maximum Data Path at Slow Process Corner: ap_CS_fsm_FSM_FFd3 to
grp_mult_fu_206/iir_biquad_mul_23s_23s_46_6_U1/iir_biquad_mul_23s_23s_46_6_
MulnS_0_U/Mmult_tmp_product
    Location             Delay type         Delay(ns)  Physical Resource
                                                       Logical Resource(s)
    -------------------------------------------------  -------------------
    SLICE_X21Y46.BMUX    Tshcko                0.461
ap_CS_fsm[2]_PWR_4_o_equal_63_o
                                                       ap_CS_fsm_FSM_FFd3
    SLICE_X21Y53.C2      net (fanout=143)      1.283   ap_CS_fsm_FSM_FFd3
    SLICE_X21Y53.C       Tilo                  0.259
d1_in_V_read_reg_517<3>

Mmux_grp_mult_fu_206_a_V1061
    SLICE_X27Y18.C5      net (fanout=31)       3.212
Mmux_grp_mult_fu_206_a_V106
    SLICE_X27Y18.C       Tilo                  0.259
Mmux_grp_mult_fu_206_a_V241
```

```
Mmux_grp_mult_fu_206_a_V241
    SLICE_X27Y18.B4      net (fanout=1)        0.327
Mmux_grp_mult_fu_206_a_V24
    SLICE_X27Y18.B       Tilo                  0.259
Mmux_grp_mult_fu_206_a_V241

Mmux_grp_mult_fu_206_a_V243
    DSP48_X0Y10.B1       net (fanout=1)        2.479
grp_mult_fu_206/iir_biquad_mul_23s_23s_46_6_U1/iir_biquad_mul_23s_23s_46_6_
MulnS_0_U/b_i<1>
    DSP48_X0Y10.CLK      Tdspdck_B_B0REG       0.149
grp_mult_fu_206/iir_biquad_mul_23s_23s_46_6_U1/iir_biquad_mul_23s_23s_46_6_
MulnS_0_U/Mmult_tmp_product

grp_mult_fu_206/iir_biquad_mul_23s_23s_46_6_U1/iir_biquad_mul_23s_23s_46_6_
MulnS_0_U/Mmult_tmp_product
-------------------------------------------------  ------------------------
---
    Total                                          8.688ns (1.387ns logic,
7.301ns route)
                                                           (16.0% logic, 84.0%
route)

--------------------------------------------------------------------------------
-----
```

Appendix A2

```
================================================================================
==
Device Utilization Summary in the generated design:
================================================================================
==

Slice Logic Utilization:
  Number of Slice Registers:                   417 out of  54,576    1%
    Number used as Flip Flops:                 417
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                        398 out of  27,288    1%
    Number used as logic:                      370 out of  27,288    1%
      Number using O6 output only:             319
      Number using O5 output only:              21
      Number using O5 and O6:                   30
      Number used as ROM:                        0
    Number used as Memory:                      19 out of   6,408    1%
      Number used as Dual Port RAM:              0
      Number used as Single Port RAM:            0
      Number used as Shift Register:            19
        Number using O6 output only:             3
        Number using O5 output only:             0
        Number using O5 and O6:                 16
    Number used exclusively as route-thrus:      9
      Number with same-slice register load:      8
      Number with same-slice carry load:         1
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                   160 out of   6,822    2%
  Number of MUXCYs used:                        80 out of  13,644    1%
  Number of LUT Flip Flop pairs used:          516
    Number with an unused Flip Flop:           127 out of     516   24%
    Number with an unused LUT:                 118 out of     516   22%
    Number of fully used LUT-FF pairs:         271 out of     516   52%
    Number of slice register sites lost
      to control set restrictions:               0 out of  54,576    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                       279 out of     320   87%

Specific Feature Utilization:
  Number of RAMB16BWERs:                         0 out of     116    0%
  Number of RAMB8BWERs:                          0 out of     232    0%
  Number of BUFIO2/BUFIO2_2CLKs:                 0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:             0 out of      32    0%
  Number of BUFG/BUFGMUXs:                       1 out of      16    6%
    Number used as BUFGs:                        1
    Number used as BUFGMUX:                      0
```

```
  Number of DCM/DCM_CLKGENs:                    0 out of       8    0%
  Number of ILOGIC2/ISERDES2s:                  0 out of     376    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:        0 out of     376    0%
  Number of OLOGIC2/OSERDES2s:                  0 out of     376    0%
  Number of BSCANs:                             0 out of       4    0%
  Number of BUFHs:                              0 out of     256    0%
  Number of BUFPLLs:                            0 out of       8    0%
  Number of BUFPLL_MCBs:                        0 out of       4    0%
  Number of DSP48A1s:                           4 out of      58    6%
  Number of ICAPs:                              0 out of       1    0%
  Number of MCBs:                               0 out of       2    0%
  Number of PCILOGICSEs:                        0 out of       2    0%
  Number of PLL_ADVs:                           0 out of       4    0%
  Number of PMVs:                               0 out of       1    0%
  Number of STARTUPs:                           0 out of       1    0%
  Number of SUSPEND_SYNCs:                      0 out of       1    0%

=========================================================================
```

Device Utilization Summary in the current design:

```
=========================================================================

Slice Logic Utilization:
  Number of Slice Registers:                  486 out of  54,576    1%
    Number used as Flip Flops:                486
    Number used as Latches:                     0
    Number used as Latch-thrus:                 0
    Number used as AND/OR logics:               0
  Number of Slice LUTs:                       277 out of  27,288    1%
    Number used as logic:                     243 out of  27,288    1%
      Number using O6 output only:            213
      Number using O5 output only:              0
      Number using O5 and O6:                  30
      Number used as ROM:                       0
    Number used as Memory:                     18 out of   6,408    1%
      Number used as Dual Port RAM:             0
      Number used as Single Port RAM:           0
      Number used as Shift Register:           18
        Number using O6 output only:            2
        Number using O5 output only:            0
        Number using O5 and O6:                16
    Number used exclusively as route-thrus:    16
      Number with same-slice register load:    16
      Number with same-slice carry load:        0
      Number with other load:                   0

Slice Logic Distribution:
  Number of occupied Slices:                  150 out of   6,822    2%
  Number of MUXCYs used:                       44 out of  13,644    1%
  Number of LUT Flip Flop pairs used:         492
    Number with an unused Flip Flop:           64 out of     492   13%
    Number with an unused LUT:                215 out of     492   43%
    Number of fully used LUT-FF pairs:        213 out of     492   43%
    Number of slice register sites lost
      to control set restrictions:              0 out of  54,576    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
```

```
IO Utilization:
  Number of bonded IOBs:                        275 out of     320   85%

Specific Feature Utilization:
  Number of RAMB16BWERs:                          0 out of     116    0%
  Number of RAMB8BWERs:                           0 out of     232    0%
  Number of BUFIO2/BUFIO2_2CLKs:                  0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:              0 out of      32    0%
  Number of BUFG/BUFGMUXs:                        1 out of      16    6%
    Number used as BUFGs:                         1
    Number used as BUFGMUX:                       0
  Number of DCM/DCM_CLKGENs:                      0 out of       8    0%
  Number of ILOGIC2/ISERDES2s:                    0 out of     376    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:          0 out of     376    0%
  Number of OLOGIC2/OSERDES2s:                    0 out of     376    0%
  Number of BSCANs:                               0 out of       4    0%
  Number of BUFHs:                                0 out of     256    0%
  Number of BUFPLLs:                              0 out of       8    0%
  Number of BUFPLL_MCBs:                          0 out of       4    0%
  Number of DSP48A1s:                             4 out of      58    6%
  Number of ICAPs:                                0 out of       1    0%
  Number of MCBs:                                 0 out of       2    0%
  Number of PCILOGICSEs:                          0 out of       2    0%
  Number of PLL_ADVs:                             0 out of       4    0%
  Number of PMVs:                                 0 out of       1    0%
  Number of STARTUPs:                             0 out of       1    0%
  Number of SUSPEND_SYNCs:                        0 out of       1    0%


Overall effort level (-ol):   High
Router effort level (-rl):    High
```

Appendix A3

```
================================================================================
==
Post-PAR Static Timing Report for the generated design:
================================================================================
==
All constraints were met.

Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock ap_clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
ap_clk         |    8.746|         |         |         |
---------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 19793 paths, 0 nets, and 2299 connections

Design statistics:
   Minimum period:   8.746ns{1}   (Maximum frequency: 114.338MHz)


------------------------------------Footnotes-----------------------------
--
1)  The minimum period statistic assumes all single cycle delays.

Analysis completed Mon Apr 07 15:18:01 2014
--------------------------------------------------------------------------
--


================================================================================
==
Post-PAR Static Timing Report for the current design:
================================================================================
==
All constraints were met.


Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
```

```
clk             |    7.724|         |         |         |
----------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 17411 paths, 0 nets, and 1826 connections

Design statistics:
   Minimum period:   7.724ns{1}   (Maximum frequency: 129.467MHz)


------------------------------------Footnotes-----------------------------
-----
1)  The minimum period statistic assumes all single cycle delays.

Analysis completed Mon Apr 07 15:00:25 2014
--------------------------------------------------------------------------
--
==========================================================================
==
```

Appendix B1

Synthesis Summary with no sharing factor as obtained from ISE

```
================================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Multipliers                                          : 6
 23x23-bit multiplier                                  : 4
 23x23-bit registered multiplier                       : 2
# Adders/Subtractors                                   : 6
 23-bit adder                                          : 2
 43-bit adder                                          : 2
 43-bit subtractor                                     : 2
# Registers                                            : 253
 Flip-Flops                                            : 253
================================================================================
```

Mapping summary with no sharing factor as obtained from ISE

```
================================================================================

Target Device  : xc6slx45
Target Package : fgg484
Target Speed   : -3
Mapper Version : spartan6 -- $Revision: 1.55 $
Mapped Date    : Wed Apr 02 11:53:57 2014


Design Summary
--------------

Design Summary:
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                   135 out of  54,576    1%
    Number used as Flip Flops:                  91
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:               44
  Number of Slice LUTs:                        146 out of  27,288    1%
    Number used as logic:                      131 out of  27,288    1%
      Number using O6 output only:               7
      Number using O5 output only:              42
      Number using O5 and O6:                   82
      Number used as ROM:                        0
    Number used as Memory:                       0 out of   6,408    0%
    Number used exclusively as route-thrus:     15
      Number with same-slice register load:     13
      Number with same-slice carry load:         2
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                    51 out of   6,822    1%
  Number of MUXCYs used:                       136 out of  13,644    1%
  Number of LUT Flip Flop pairs used:          188
    Number with an unused Flip Flop:            66 out of     188   35%
    Number with an unused LUT:                  42 out of     188   22%
    Number of fully used LUT-FF pairs:          80 out of     188   42%
    Number of unique control sets:               1
    Number of slice register sites lost
```

```
      to control set restrictions:              5 out of  54,576    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                      257 out of     316   81%

Specific Feature Utilization:
  Number of RAMB16BWERs:                        0 out of     116    0%
  Number of RAMB8BWERs:                         0 out of     232    0%
  Number of BUFIO2/BUFIO2_2CLKs:                0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:            0 out of      32    0%
  Number of BUFG/BUFGMUXs:                      1 out of      16    6%
    Number used as BUFGs:                       1
    Number used as BUFGMUX:                     0
  Number of DCM/DCM_CLKGENs:                    0 out of       8    0%
  Number of ILOGIC2/ISERDES2s:                  0 out of     376    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:        0 out of     376    0%
  Number of OLOGIC2/OSERDES2s:                  0 out of     376    0%
  Number of BSCANs:                             0 out of       4    0%
  Number of BUFHs:                              0 out of     256    0%
  Number of BUFPLLs:                            0 out of       8    0%
  Number of BUFPLL_MCBs:                        0 out of       4    0%
  Number of DSP48A1s:                          24 out of      58   41%
  Number of ICAPs:                              0 out of       1    0%
  Number of MCBs:                               0 out of       2    0%
  Number of PCILOGICSEs:                        0 out of       2    0%
  Number of PLL_ADVs:                           0 out of       4    0%
  Number of PMVs:                               0 out of       1    0%
  Number of STARTUPs:                           0 out of       1    0%
  Number of SUSPEND_SYNCs:                      0 out of       1    0% 0
out of       1    0%
```

Synthesis Summary with a sharing factor = 6 as obtained from ISE

```
Advanced HDL Synthesis Report

Macro Statistics
# Multipliers                                   : 1
 23x23-bit registered multiplier               : 1
# Adders/Subtractors                           : 6
 23-bit adder                                  : 2
 43-bit adder                                  : 2
 43-bit subtractor                             : 2
# Counters                                     : 1
 3-bit up counter                              : 1
# Registers                                    : 1113
 Flip-Flops                                    : 1113
# Multiplexers                                 : 186
 1-bit 2-to-1 multiplexer                      : 174
 23-bit 2-to-1 multiplexer                     : 10
 46-bit 2-to-1 multiplexer                     : 2
```

Mapping Summary with a sharing factor = 6 as obtained from ISE

```
Target Device  : xc6slx45
```

```
Target Package : fgg484
Target Speed   : -3
Mapper Version : spartan6 -- $Revision: 1.55 $
Mapped Date    : Wed Apr 02 12:14:36 2014

Design Summary
--------------

Design Summary:
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                 1,176 out of  54,576    2%
    Number used as Flip Flops:               1,132
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:               44
  Number of Slice LUTs:                        679 out of  27,288    2%
    Number used as logic:                      623 out of  27,288    2%
      Number using O6 output only:             495
      Number using O5 output only:              42
      Number using O5 and O6:                   86
      Number used as ROM:                        0
    Number used as Memory:                       0 out of   6,408    0%
    Number used exclusively as route-thrus:     56
      Number with same-slice register load:     54
      Number with same-slice carry load:         2
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                   340 out of   6,822    4%
  Number of MUXCYs used:                       136 out of  13,644    1%
  Number of LUT Flip Flop pairs used:        1,203
    Number with an unused Flip Flop:            86 out of   1,203    7%
    Number with an unused LUT:                 524 out of   1,203   43%
    Number of fully used LUT-FF pairs:         593 out of   1,203   49%
    Number of unique control sets:               5
    Number of slice register sites lost
      to control set restrictions:              28 out of  54,576    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                       257 out of     316   81%

Specific Feature Utilization:
  Number of RAMB16BWERs:                         0 out of     116    0%
  Number of RAMB8BWERs:                          0 out of     232    0%
  Number of BUFIO2/BUFIO2_2CLKs:                 0 out of      32    0%
  Number of BUFIO2FB/BUFIO2FB_2CLKs:             0 out of      32    0%
  Number of BUFG/BUFGMUXs:                       1 out of      16    6%
    Number used as BUFGs:                        1
    Number used as BUFGMUX:                      0
  Number of DCM/DCM_CLKGENs:                     0 out of       8    0%
  Number of ILOGIC2/ISERDES2s:                   0 out of     376    0%
  Number of IODELAY2/IODRP2/IODRP2_MCBs:         0 out of     376    0%
  Number of OLOGIC2/OSERDES2s:                   0 out of     376    0%
```

Number of BSCANs:	0	out of	4	0%
Number of BUFHs:	0	out of	256	0%
Number of BUFPLLs:	0	out of	8	0%
Number of BUFPLL_MCBs:	0	out of	4	0%
Number of DSP48A1s:	4	out of	58	6%
Number of ICAPs:	0	out of	1	0%
Number of MCBs:	0	out of	2	0%
Number of PCILOGICSEs:	0	out of	2	0%
Number of PLL_ADVs:	0	out of	4	0%
Number of PMVs:	0	out of	1	0%
Number of STARTUPs:	0	out of	1	0%
Number of SUSPEND_SYNCs:	0	out of	1	0%

Appendix B2

```
================================================================================
Mapping resource summary after timing optimization as obtained from ISE
================================================================================

Target Device  : xc6slx45
Target Package : fgg484
Target Speed   : -3
Mapper Version : spartan6 -- $Revision: 1.55 $
Mapped Date    : Wed Apr 02 13:05:45 2014

Design Summary
--------------
Number of errors:      0
Number of warnings:    0
Slice Logic Utilization:
  Number of Slice Registers:                 1,718 out of  54,576    3%
    Number used as Flip Flops:               1,696
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:               22
  Number of Slice LUTs:                        853 out of  27,288    3%
    Number used as logic:                      711 out of  27,288    2%
      Number using O6 output only:             623
      Number using O5 output only:              42
      Number using O5 and O6:                   46
      Number used as ROM:                        0
    Number used as Memory:                      24 out of   6,408    1%
      Number used as Dual Port RAM:              0
      Number used as Single Port RAM:            0
      Number used as Shift Register:            24
        Number using O6 output only:             2
        Number using O5 output only:             0
        Number using O5 and O6:                 22
    Number used exclusively as route-thrus:    118
      Number with same-slice register load:    116
      Number with same-slice carry load:         2
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                   456 out of   6,822    6%
  Number of MUXCYs used:                       180 out of  13,644    1%
  Number of LUT Flip Flop pairs used:        1,668
    Number with an unused Flip Flop:           103 out of   1,668    6%
    Number with an unused LUT:                 815 out of   1,668   48%
    Number of fully used LUT-FF pairs:         750 out of   1,668   44%
    Number of unique control sets:               6
    Number of slice register sites lost
      to control set restrictions:              26 out of  54,576    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                       257 out of     316   81%

Specific Feature Utilization:
```

```
Number of RAMB16BWERs:                          0 out of     116    0%
Number of RAMB8BWERs:                           0 out of     232    0%
Number of BUFIO2/BUFIO2_2CLKs:                  0 out of      32    0%
Number of BUFIO2FB/BUFIO2FB_2CLKs:              0 out of      32    0%
Number of BUFG/BUFGMUXs:                        1 out of      16    6%
  Number used as BUFGs:                         1
  Number used as BUFGMUX:                       0
Number of DCM/DCM_CLKGENs:                      0 out of       8    0%
Number of ILOGIC2/ISERDES2s:                    0 out of     376    0%
Number of IODELAY2/IODRP2/IODRP2_MCBs:          0 out of     376    0%
Number of OLOGIC2/OSERDES2s:                    0 out of     376    0%
Number of BSCANs:                               0 out of       4    0%
Number of BUFHs:                                0 out of     256    0%
Number of BUFPLLs:                              0 out of       8    0%
Number of BUFPLL_MCBs:                          0 out of       4    0%
Number of DSP48A1s:                             4 out of      58    6%
Number of ICAPs:                                0 out of       1    0%
Number of MCBs:                                 0 out of       2    0%
Number of PCILOGICSEs:                          0 out of       2    0%
Number of PLL_ADVs:                             0 out of       4    0%
Number of PMVs:                                 0 out of       1    0%
Number of STARTUPs:                             0 out of       1    0%
Number of SUSPEND_SYNCs:                        0 out of       1    0%
```